A GENERALIZED EXECUTION MODEL FOR NESTED DATA-PARALLEL COMPUTING

By
Dean Engelhardt

April 18, 1997

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ADELAIDE

Contents

List of Tables

List of Figures

Preface

Of the many attempts to simplify the task of programming today's parallel multiprocessor architectures, the most successful is the paradigm of Data-Parallelism. Much of the appeal of the model lies in its high-level view of the parallel machine coupled with its efficient mapping to a large class of real-world architectures.

While the Data-Parallel model has been very effective at allowing highly parallel specification of operations across rectangular arrays, its applicability to programs which use less regular data structures is very limited. This factor compromises the utility of the model: many important scientific problems based heavily upon irregular data structures (e.g., sparse matrix computations, finite element irregular mesh codes) cannot be efficiently parallelized using existing Data-Parallel techniques.

The purpose of this thesis is to explore extensions to the Data-Parallel model which make it more amenable to these kinds of irregular problems. In particular we consider the paradigm of Nested Data-Parallelism proposed by Blelloch, investigating techniques for efficiently mapping programs with such nested parallelism onto traditional multiprocessor architectures. Through mathematical analysis we derive a novel implementation of such features which makes use of a multi-threaded model of computation upon each processor of a multiprocessor machine. We describe a realization of this execution model which we have constructed for the Thinking Machines CM-5 and detail how we have used this system as the basis for the implementation of a simple language with Nested Data-Parallel features. To demonstrate the validity of our approach we benchmark the performance of several programs compiled using this language system, comparing the figures obtained with those for equivalent programs compiled in the CM Fortran and NESL systems. For the real-world irregular codes we examine, our unoptimized compiler output running on the CM-5 threading system delivers performance that is competitive with both existing systems, surpassing each in certain situations.

Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Dean Engelhardt
April 18, 1997

Acknowledgments

There are of course a great many people to whom I am indebted in various ways for their support during the years that I have been engrossed in the work reported herein.

First and foremost I owe thanks to Catherine Edis, my warm and loving SO, for many years of support — both tangible and intangible — and for being there through everything. Without her deep friendship and moral support, I am not sure I would ever have reached the stage where completing this project first seemed possible.

Massive thanks are also due to Andrew Wendelborn who proved to be a thesis supervisor with an inspired ability to provide just the right amount of guidance at just the right time. His talents as an organizer, publications co-author, scholarship co-ordinator, and mentor have made the past years of study far less complicated and far more comfortable than they might have otherwise have been.

Thanks also are due to Michael Oudshoorn for informally co-supervising the visualization aspects of this project.

I would also like to thank all those who have been unfortunate enough to share an office with me over the past years, and who have been forced to endure sometimes incoherent rantings, mad scribblings on white-boards and workstations that make strange noises at odd times. A long time sufferer in this regard is fellow Adl project researcher Bradley Alexander. Others include (in no particular order): Paul Roe, Chris Nowak, Josh Leane, Sam Bushell, Matthew Wilson, Heath James and Ken Hawick.

For providing an often-stable CM-5 for the many hours of performance analysis undertaken for Chapter 8 of this thesis, I would like to thank Francis Vaughan of the South Australian Centre for Parallel Computing and Lindsay Hood formerly of TMC Australia.

Several members of the Scandal team at CMU (the makers of NESL) were very helpful in promptly answering questions via email and looking into installation problems and other less identifiable difficulties. Thanks are due to Jonathan Hardwick, Jay Sippelstein and Guy Blelloch. In a similar vein, Seth Goldstein, from Berkeley also deserves thanks for prompt information on active messages and TAM.

Chapter 1

Introduction

The efficient programming of highly parallel computers — those which offer tens, hundreds or thousands of processors connected by a high-speed network — is perhaps one of the most difficult problems facing modern scientific computing. It is widely recognized that such machines hold enormous potential to provide radically improved execution speeds for many scientific programs, thus enabling larger problems to become computationally feasible. Yet all too often this potential is unrealized, with programmers who port their codes to parallel architectures observing only a moderate improvement in performance.

By far the greatest factor contributing to this phenomenon is the complexity inherent in the programming models offered for programming parallel machines. The earliest attempts at constructing such models involved a simple grafting of parallel features (e.g., communications primitives) onto the traditional sequential model of execution as embodied in languages such as C and Fortran. Such systems, many of which are still advocated, have the advantage that they are easy for a programmer familiar with sequential programming to learn. However, as such programmers soon discover, the models do little to mask the inherent complexity involved in programming many individual processors, each of which may execute instructions entirely independently and which may, at any time, interact in an arbitrary fashion. By and large this complexity (many times greater than that of programming a uniprocessor machine) must be managed by the programmer. On top of this, the programmer is also charged with the task of making his or her implementation of the desired algorithm perform well by ensuring that all processors in the system remain gainfully busy throughout the execution, and that the overheads (e.g., due to communication between nodes) are not great. Considering the massive intellectual effort required to construct a program under these models that is correct in all possible cases and also efficient, it is not surprising that few such programs exist.

The *Data-Parallel* programming model [52, 121] represents a simpler alternative to the inherently complex task of programming under such models. Its simplicity derives from the provision of a higher-level view of parallel programming which abstracts away many of the details of the parallel architecture. The fundamental principle underlying this paradigm is that a potentially high degree of parallelism can be achieved through partitioning a program's large data aggregates across many nodes of a parallel machine, and then framing a computation in terms of whole-aggregate operations whose (purely serial) per-element computations are relatively independent, and which may thus be performed in parallel. A Data-Parallel (DP) program is made up of a sequence of such high-level collection-oriented operations through which runs a single conceptual path of control. The intellectual effort required to construct such a program is considerably less than that needed to co-ordinate the multiple concurrent control paths of the serial-derived paradigm. Thus, the DP model is easier to program.

However, the higher-level abstraction afforded by the paradigm means that the ultimate performance of a DP program is determined by how it is mapped by compiler onto the parallel hardware. Typically the high-level operations offered by a DP language are chosen such that they are highly regular in both their patterns of computation and their parallelism. This makes the compilation task simpler, since such forms have clear — and at the same time efficient — mappings onto parallel architectures. The compiler technology required to generate such mappings is well understood, and has been incorporated into a number of very efficient DP implementations (e.g., Thinking Machine's CM Fortran [127] and C* [101, 102, 118, 125], as well as various HPF [57] systems). These implementations adopt a parallelizing strategy which seeks to cast the program in a form amenable to execution under either the *Single Program Multiple Data (SPMD)* or *Single Instruction Multiple Data (SIMD)* model of parallel computing. The approach revolves around the partitioning of data aggregates within the DP program across a number of disjoint memories, each of which is closely associated with a processor. The program is compiled into a sequence of sub-steps, each corresponding to a DP operation in the source. The various processors of the parallel machine progress through this sequence in lock-step, globally synchronizing at the end of every sub-step (to eliminate any possibilities of race conditions). It is in the execution of the sub-steps that the parallelism of the implementation is realized: each DP operation is implemented as a co-operative operation between processors, in which each processor is charged with executing the serial per-element operations for those aggregate indices stored within its associated memory.

In the sections which follow we describe several languages which have, in the past, been used for the expression of DP execution, as well as offering details of the concrete machines to which such programs have been mapped. Following this historical appraisal of the DP paradigm, we offer a contemporary analysis of its suitability as a general-purpose medium for parallel execution, noting a broad class of important applications which are not amenable to traditional DP expression because they embody parallel forms which are irregular. The remainder of this thesis is devoted to investigating techniques for supporting such irregularity while retaining the general flavour of DP execution which makes it an effective and scalable technique for parallel computation.

1.1 Hardware Implementations of Data Parallel Execution

Historically, the success of the DP model of parallel computing can largely be attributed to the development of computer architectures which directly embodied executions models — the SIMD and SPMD paradigms — useful for supporting DP. Where implementations targetting less-specialized parallel hardware (e.g., *Multiple Instruction Multiple Data (MIMD)* multiprocessors) were required to synthesize or simulate such modes of execution, DP systems producing code for specialized SIMD or SPMD machines could take advantage of hardware-optimized facilities especially catering to the styles of interaction and synchronization found within those models. This level of direct support proved an enabling factor in DP's emergence as a technology for high performance programming.

Below we briefly survey several of the SIMD and SPMD architectures which have proven successful targets for implementation of the DP style of execution.

1.1.1 SIMD Hardware Solutions

By definition, a SIMD computer is one that applies a single machine instruction to a number of different data elements simultaneously. Typically, such a machine is organized as a collection of simple computational elements, each with an associated memory which stores the data element upon which that node will act during a computation. Upon each clock cycle, an instruction is broadcast to every computational node whereupon it is carried out in the context of the registers and/or memory that is owned by the node. A point of machine synchronization occurs at the end of each machine instruction: a new instruction will only be broadcast after all nodes have completed the previous instruction. This global synchronization is provided directly in hardware.

The considerable research interest in SIMD machines prevailing in the 1970's and 1980's partly represents prevailing attitudes towards theoretical design of parallel computers, but is more representative of engineering issues such as the price-performance trade-offs in hardware available at the time [132]. In short, it was only by the cojoining of a large number of very simple processing elements that a cost-effective parallel computer could be constructed.

One of the first designs to embody this principle in a commercially-released product was the ICL Distributed Array Processor (DAP) [132, 55, 97]. This architecture was characterized by very simple computational elements (single bit-serial processors) connected together in a two-dimensional grid of processors. The earliest DAP configurations comprised square grids of 1024 (32×32) nodes — later versions allowed 4096 elements to be connected. The DAP was originally conceived as a dedicated special-purpose array processor which would be connected to an ICL mainframe (in much the same way that vector processors were added external to earlier architectures to provide high-speed vector computation). Effectively the DAP functioned as a special unit of the mainframe's memory — values could be read to and written from the individual banks of memory associated with each processing node. Special instructions issued by the mainframe would cause some or all of the processors to perform computations in the context of their local memory and registers in a SIMD fashion. Data could be moved between processors along the network which connected each with its nearest grid neighbours to the North, South, East and West. The limited nature of this connectivity (and the lack of a general router) meant that only simple and regular patterns of communication could be supported.

The Thinking Machines CM-2 [7, 132, 133, 122] is a later SIMD architecture which embodies many of the same notions, but also adds new features to the design. Like the DAP, the CM-2 is composed from a large number of single-bit processing elements, each with an associated memory. Commercial systems were sold with between 4096 and 65536 nodes. To improve the computational performance of the nodes, Thinking Machines chose to group the processors into neighbourhoods of 32 elements and associate a Weitek-based floating-point co-processor with each such neighbourhood. The design was flawed by the fact that the data-path between the nodal memories and the FPU did not supply enough bandwidth to feed data to the unit at a rate which achieved the theoretical MFLOPS rating of the Weitek chip. Nodes of the CM-2 are joined by a hypercube network which could support both regular (e.g., nearest neighbour) and general communication, the latter due to the presence of a router chip on each node. It should be noted, however, that the cost of such general communication is several orders of magnitude greater than that of using regular modes. As with the DAP, the CM-2 is properly envisaged as the back-end for a conventional serial processor. At various times CM-2 systems were delivered attached to Sun 4 work stations, Vaxes and Symbolics 3600-series Lisp Machines.

Contemporary with the CM-2 were the various SIMD architectures produced by the MasPar Computer Corporation, in particular the MP-1/2 [7, 132, 16, 77] designs. These were multiprocessor machines based around collections of 4-bit processors arranged in a unique network configuration called the *X-Net*. In essence the X-Net consists of a two-dimensional grid of processors, each of which has direct connectivity with its nearest eight neighbours. In hardware, however, the node only has four data-paths extending from it — each of these leads to a routing node which can switch information through the network. A high-speed global router present within the X-Net allows the MasPar machines to implement general communication as well as the regular styles of interaction found in the DAP (although at a significantly greater cost). Configurations of the MP-1 and MP-2 typically consisted of as many as 16384 4-bit nodes, each of which integrated a floating-point/integer unit to improve its computational power. The nodes executed instructions broadcast from a DECstation 5000 front end, in a SIMD fashion.

The characteristics of SIMD multiprocessors such as the DAP, CM-2 and MasPar offer considerable benefits for the efficient implementation of DP operators. The synchronization required for correct execution of such an operator can be easily derived from the synchronization that occurs naturally within the machine upon the execution of each instruction. Furthermore, the patterns of communication typically found in common DP operations display significant regularity, making their implementation under each of the three architectures very efficient. These two areas of direct hardware support are jointly responsible for the very high levels of DP performance achieved on such architectures — programs which are *purely* DP can be compiled into very efficient forms. Such hardware is, however, quite inefficient as a basis for less-structured forms of computation. This leads to the non-DP sections of a program (e.g., vector indexing operations) becoming very costly to the point of seriously degrading overall program efficiency. Given the fact that few practical programs are completely composed from DP features, many have seen this as a serious limitation on the utility of SIMD hardware. This factor can be identified as one of the forces — along with significant changes in the cost/performance of off-the-shelf serial processors — which motivated a distinct trend away from this style of specialized parallel architecture.

1.1.2 SPMD Hardware Solutions

Changing processor technologies, coupled with the general observations concerning the difficulty in obtaining good performance for real-world problems executed upon SIMD machines, has lead computer architects to consider alternatives for supporting the DP paradigm in hardware. The obvious candidates, general-purpose MIMD machines, embody an unstructured style of execution with little concession for low-cost synchronization. For these reasons such machines do not mesh well with the DP model, and typically perform poorly when called upon to emulate a DP style of execution (usually by synthesizing appropriate synchronization protocols in software). Experience has shown that a better architecture for the execution of DP operators can be obtained by specializing a general MIMD machine by adding hardware which directly implements these protocols of global synchronization. Further hardware optimizations can be made by assuming the default DP situation of each node executing an identical code image. We characterize architectures derived from such specializations as *SPMD machines* since (as was the case in the SIMD model) all processors are effectively executing the same program across different data in parallel.

However, unlike the execution found in SIMD machines, the SPMD architecture has no implicit per-instruction synchronization: the nodes proceed independently in their execution except at points in the program where special machine instructions cause them to synchronize with the other nodes. We call this model of execution *lock-step*.

The principal advantage of the SPMD architecture (from the perspective of implementing DP) is that, while it retains the cheap synchronization which made the SIMD machines efficient executors of DP operations, the underlying similarities to MIMD machines offer good performance to non-DP sections of a program.

One architecture which embodies this direction in architectural evolution is the Thinking Machines CM-5 [7, 124, 95, 33], the (rather dissimilar) successor to the CM-2. The CM-5 is composed of a relatively small number (from 32 to 1024) of large computational elements (SPARC processors with an optional bank of 4 high-performance vector units), each with an attached bank of memory (32Mb), connected by a high-speed fat-tree [71, 45, 70] network. At this level of description the system resembles a general distributed memory MIMD machine. However, there are two important differences. Firstly, the CM-5 enforces that each of the processor nodes must each hold a copy of a single code image, although its execution of instructions from that image is, by default, independent. Co-ordination of nodes within the machine is implemented by a second network (the *control* network) that co-exists with the primary fat-tree network, and similarly links all nodes. This network provides rapid global synchronization of all nodes in the machine, as well as directly implementing several features useful for DP execution. These include hardware implementations of global broadcasts, global reductions (e.g., a global OR operation in which each node contributes one bit) and parallel prefix operations[1]. Furthermore, the control network supports variants of these operators which are *segmented*: each processor can set a flag indicating that it is the first element in a new segment of the input data — computations for each segment are treated independently[2]. The control network is ignored by most machine instructions executed by the node, with only a select few synchronization and global-computation (e.g., parallel prefix operations and parallel reductions) instructions activating its facilities. This model of nodal execution is precisely the lock-stepped evaluation described previously: nodes execute independently of one another, except at explicit points of synchronization.

[1]See Section 5.2.5 or [17] for a description of this class of DP operations.

[2]The segmented vector model is described in detail in [17]; we return to a discussion of segmented DP operations in Section 2.3.

The control network facilities of the CM-5 permit very efficient implementations to be made for DP operations, in much the same way as the cheap synchronization of the CM-2 was an enabling factor in providing DP implementations on that architecture. However, the fat-tree network offers a communication network which, while optimized for regular communication (e.g., nearest neighbour communication), delivers a high-bandwidth medium for point-to-point communication. General communications across this network are only a factor of four worse than the optimal case, as compared to a factor of 100 or more for the SIMD machines. This offers considerable opportunities for the efficient execution of the non-DP segments of a program (e.g., dereferencing a single index of a data aggregate), making the CM-5 considerably better at executing practical problems (both DP and non-DP).

1.2 Data-Parallel Languages

A number of programming languages have been proposed which centre upon DP constructs as the primary source of parallelism in programs. Such parallelism is obtained by the specification of whole-aggregate operations across aggregates which can be distributed between nodes of a parallel machine. The majority of DP languages provide similar features for the introduction of these types of aggregates and operations [55] — most can be characterized as providing the following common facilities:

1. **Aggregates as elemental data objects:** the ability to pass whole aggregates as function arguments, and the ability to syntactically refer to an entire aggregate (e.g., by providing the aggregate name without any index dereference) within an expression.

2. **Bulk subselection from aggregate objects:** the ability to obtain the values for an entire aggregate dimension as a single operation, and often the ability to be able to further refine this selection by specifying a subrange of indices.

3. **Aggregate expressions and conformity:** the ability to specify an operation across all indices of an aggregate simply by applying the operation to the unindexed aggregate name, and also the ability to apply simple binary operations (e.g., plus) to pairs of conformant aggregates. The semantics of these latter operations are defined by pairwise matching aggregate indices.

4. **Coercion of aggregates to obtain conformity:** the ability to be able to either increase the rank of an aggregate (by copying values from the existing aggregate along a new dimension) or to decrease the rank of an aggregate (by collapsing an aggregate dimension by combining values using some function — e.g., summing along a particular dimension).

5. **Expression indexing:** the ability to apply all the normal styles of aggregate indexing to expressions which represent aggregate values.

6. **Aggregate Assignment:** the ability to be able to assign an aggregate valued expression to a specified conformant aggregate. The semantics of such an assignment are element-wise; each element in the destination aggregate is overwritten with the value of the corresponding element in the aggregate-valued expression.

7. **Parallel Operations Across Aggregates:** a set of standard DP operations which enable a programmer to introduce explicit parallelism through collection-oriented expressions; these include common (commutative) reductions of aggregates, parallel prefix [52, 68, 76, 83] operations, and permutation operations.

These language features collectively provide an environment in which explicitly parallel specification can be made. The precise manifestation of these concepts in DP languages — including the choice of the types of aggregates to allow parallel expression across — is governed to a large extent by the historical development of this class of languages and the computational forms they were originally designed to express. We turn now to a brief survey of these factors, and a description of the common DP languages they have produced.

1.2.1 Historical Development of DP Languages

The development of DP languages can be considered to have been driven by two distinct forces: the need to develop a medium for programming newly-constructed SIMD machines, and the desire to construct a conceptually simple framework for expressing parallel operations. The first of these influences historically dominated the early evolution of the paradigm, and has largely been responsible for forging the broad characteristics of the DP languages which are today widespread and popular. The desire for simplified parallel execution has been responsible both for the development of several research-based augmentations to the basic DP language model, and ultimately to the refinement and modern evolution of the model through absorption of ideas from such research systems.

In this section we briefly chronicle this history and evolution of DP languages.

Conceptual Origins

The basic concepts underlying DP can be identified in early *collection-oriented* [115] languages such as APL [60] and SETL [108]. Within these languages, features were provided for the expression of whole-aggregate operations as single source-level operations. Conceptually, an aggregate was a first-class citizen in these languages (rather than merely a bundle of data elements) and features were provided for applying operations directly to an aggregate. The semantics of such collection-oriented operation were typically defined to imply that an identical computation was to be performed for each index of the collection. Also present were language features to reduce the dimension of an aggregate (e.g., sum all elements of a vector). These features are all reminiscent of the DP style of programming, despite the fact that these languages were typically designed not for their opportunities for parallelism, but rather for their expressive and abstractive power.

Languages to Program Early SIMD Machines

When SIMD architectures began to be developed in the 1970's as the results of research into cost-effective ways to build parallel computers, the development of an appropriate paradigm for programming such machines became an important priority. Early prototype systems were coded directly in assembly language; however, with the possibility that such machines could become commercial products came the necessity to develop languages and compilers which could effectively exploit the potential of the architectures. In developing these systems, vendors most frequently targetted Fortran as a basis for the language they sought to develop. This choice was driven largely by the language preference and familiarity of their intended market (scientific programmers). The DP extensions to the base language were derived very simply from an appraisal of the styles of execution for which the underlying SIMD machine was optimized. Such machines were clearly intended as fast array processors — thus, parallel constructs across arrays were introduced. The fact that the nodes of the machine were always required to execute the same sequence of instructions gave their parallel execution a very regular flavour, well suited to problems which called for all array elements to be acted upon in an identical (or similar) fashion. That is, in a DP fashion. Thus, DP features were added to the language to allow for the convenient expression of these types of programs (which best exploited the target machines).

An early example of a vendor-supplied DP language targetting a specific SIMD architecture is DAP Fortran [59] (a dialect constructed for the ICL DAP, described in the previous section)[3]. The language exploited parallelism from operations across arrays whose values were partitioned across the machine. A severe limitation placed upon such partitioned aggregates was that they could only be of a size less than or equal to the number of nodes in the machine. That is, the largest array that could be considered on the early DAP machines was 32×32; in later machines arrays up to 64×64 could be acted upon. This imposition was clearly a product of the way in which the computation was mapped onto the SIMD machine: each processing element was responsible for acting on a sole aggregate element. DP operations in the language supported the following types of operations across partitioned aggregates:

- broadcast a value to be stored in each aggregate element or sub-aggregate (e.g. give each row of a 2D matrix the vector value `[4,5,6]`),
- perform parallel assignments between aggregates of conformable shape,
- compute an array-section or slice,
- express computation of shifted forms of an aggregate (e.g., cyclically shift all indices of the vector one place to the right)
- perform a given scalar operation on all indices in parallel (e.g., compute the `sin` of all indices of a matrix, storing the results in a new matrix)
- combine aggregates (or dimensions of a multi-dimensioned aggregate) in parallel according to some function (e.g., sum all rows of a 2D matrix to give a vector of sums)

[3]The DAP Fortran language bears many similarities to earlier research languages developed to program the Illiac IV [7, 15]. Specifically, it is influenced by the CFD Fortran [93, 120] system constructed for that architecture. The principal innovation of the DAP language over this predecessor was the ability to consider more than one dimension of an aggregate as a candidate for parallelism.

The DP features of the language were introduced either through a special indexing notation (e.g., `A(,1)` referred, by a notation of subscript elision, to the first column of the matrix `A`) or through a set of inbuilt aggregate-level functions. Figure 1 gives some examples of DAP Fortran's DP features at work. To allow for more complex operations to be specified, the language allowed for many of these styles of DP operation to be combined with conditionals which defined the subset of the processors which would execute the operation. For example, it was possible to specify within an aggregate-level assignment that only those array elements currently with values less than 0 should receive a new value. In this case, this would define a DP operation in which only a subset of the machine would participate.

Later SIMD-targetted Languages

As the facilities offered by SIMD architectures developed, vendor-supplied Fortran compilers evolved to add features exploiting the new hardware-supported functionality. Early improvements over the DAP Fortran (e.g., CM Fortran [127, 128] and MasPar's MPF [55]) eliminated the restriction on the size of distributed aggregate which could be acted upon. These languages took advantage of special hardware present in machines such as the CM-2 and MasPar which implemented a model of *virtual processors*. Conceptually in such languages a DP operation across a grid utilizes as many virtual processors as there are elements in the aggregate — as in DAP Fortran, each such processor is responsible for computations across only that element. Hardware within the machine maps virtual processors onto actual processors, allowing each physical processor to take the role of many virtual processors. This advance in DP languages introduced a degree of architectural independence — no longer was the size and configuration of the machine explicitly defined within the program. Furthermore it permitted code to be easily transported between machines of different (physical) size.

The CM Fortran language also introduced an alternate form of DP notation based around the `forall` loop, a generalized parallel loop across indices of an array. The expressiveness of this concept extends the range of problems which can be expressed as DP programs; the fact that the language permits for the nesting of `forall` loops offers interesting possibilities for supporting computationally complex or irregular codes (see Section 2.1.1).

```
INTEGER A(3,3), B(3,3), C(3,3), V(3), W(3)

V = (1,2,3)
```
$\Rightarrow \quad V = [\,1\;2\;3\,]$

```
W = (1,1,1)
```
$\Rightarrow \quad W = [\,1\;1\;1\,]$

1. Broadcast Operations
```
A = XPND (V,1,3)
```
$\Rightarrow \quad A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$

```
B = XPND (W,1,3)
```
$\Rightarrow \quad B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

2. Aggregate Level Operations
```
C = A + B
```
$\Rightarrow \quad C = \begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$

3. Computing Array Slice
```
V = C (,1)
```
$\Rightarrow \quad V = [\,2\;2\;2\,]$

4. Computing Cyclic Shift
```
C = C (,+)
```
$\Rightarrow \quad C = \begin{bmatrix} 4 & 2 & 3 \\ 4 & 2 & 3 \\ 4 & 2 & 3 \end{bmatrix}$

5. Element-wise Fortran Primitive
```
B = LOG (B)
```
$\Rightarrow \quad B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

6. Reducing Dimension
```
W = SUM (C,1)
```
$\Rightarrow \quad W = [\,12\;6\;9\,]$

Figure 1. Examples of DAP Fortran's DP Features

Another facility which appeared first in this generation of DP languages was the ability to specify permutation-like operations on vectors and matrices. These typically appeared in the language either as intrinsic functions (e.g., the parallel `SEND` operations of CM Fortran) or as special indexing operations in which an integer vector (rather than a scalar) appears as the index. Figure 2 illustrates this latter (vector-valued subscript) notation as used by CM Fortran. Facilities are also commonly available in this language for specifying generalized permutations in which values could collide (and be combined by a function) at their destination. It seems clear that these kinds of features entered DP languages primarily because of the presence of generalized network routing hardware in the machines to which they were targetted (e.g., the CM-2 and MasPar).

```
INTEGER V(6), W(6), R(6)

V = (11,12,13,14,15,16)
⇒        V = [ 11 12 13 14 15 16 ]

W = (1,5,4,3,6,2)
⇒        W = [ 1 5 4 3 6 2 ]

R = V(W)
⇒        R = [ 11 15 14 13 16 12 ]
```

Figure 2. Examples of CM Fortran's Permutation Operations

DP Languages Derived from C and Lisp

Simultaneous with these advances in vendor-supplied languages, several DP languages were proposed which developed the paradigm in other directions. A number of definitions (and implementations) were made for DP extensions to languages other than Fortran, but which supplied similar styles of array-based operation. The C* [101, 102, 121, 118, 48, 125] language was an early example of such an extension, which sought to extend ANSI C by adding a special class of parallel operations (specified using a special style of indexing) across a program's (implicitly) distributed arrays. Similar efforts in merging DP features with Common Lisp lead to Connection Machine Lisp [53, 55] and later *LISP [123, 121].

Experiments with Parallel-Prefix Operations

As well as developing new bases for DP languages, researchers in the 1980's also explored new modes of expressing parallel computation in the DP style. The most successful of the proposals to emerge from this work was that of the *Parallel-Prefix* (or *scan*) operator [52, 68, 76, 83]. The semantics of this operation involved the computation of a series of partial sums to be computed according to some (associative) combining function. For example, a `plus_scan` operation, when applied to a vector `[1,2,3]` would produce the vector of partial sums `[1,3,6]`. Although such an operation seems, at first glance, inherently serial (the result of each addition must be computed prior to its input into the next addition), such is not the case: techniques developed by Hillis and Steele [52] (described further in Section 5.2.5) allow such operations to be calculated in $O(n \log n)$ on an n processor SIMD machine. Furthermore, research into the expressive power of such operations showed them to be surprisingly general, with the ability to concisely express complex operations such as parallel radix sorting, language parsing and even list traversal [52, 17]. The demonstration of parallel prefix operations as practical expressions of parallel operations, which also operated in a broadly DP fashion, lead to their introduction into a number of languages, including CM Fortran and *LISP. It also influenced the design of the CM-5 hardware, as described in Section 1.1.2 previously.

Standardizing DP Fortran

With the proliferation of many different dialects of DP Fortran, all based on similar notions and abstractions, came the desire to standardize. The end result of considerable discussion by ANSI group X3J3 was the language Fortran 90 [81]. This standard embodied the DP features introduced with DAP Fortran, and added the machine-independence of languages such as CM Fortran. As with the majority of DP Fortran dialects, much of the parallel specification is represented within array indexing operations. Special parallel intrinsics (e.g., matrix multiplication, dot products, summation of all matrix elements, circular and end-off shifts) are also provided. Notably, the standard does not explicitly include parallel prefix operations.

Recent Directions in DP Fortran

Subsequent to the codification of the Fortran 90 standard, a number of new DP dialects have emerged. Of these, the most well-known is High Performance Fortran (HPF) [57, 141, 75], an extension to the standard which seeks to offer the programmer direct control over a number of issues concerning the mapping of the program to an underlying machine. As we have described previously, the common implementation path for DP languages on parallel machines centres upon the decomposition of array structures across the various memories of the underlying machine. In HPF source-level features are available which can be used to guide the compiler in choosing the proper data decomposition. Such features are a reaction to the difficulties encountered by compiler-writers in producing a system which can automatically make high-quality data layout choices by analytical means. In HPF, a programmer can specify a `DECOMPOSITION` which is an abstract entity which looks similar to an array but which implies no storage. Such declarations are useful since arrays within the program can be specified as being `ALIGN`ed with a particular `DECOMPOSITION`. If two or more vectors (or matrices) are `ALIGN`ed with the same `DECOMPOSITION`, the compiler can deduce that those data aggregates should be decomposed identically: that is, if vectors `V` and `W` are identically aligned, it is always true that the i^{th} element of `V` is stored within the same memory as the i^{th} element of `W`. This implies that DP operations such as `V = 2*W` involve no inter-node communication and are thus very cheap to execute. The syntax of the language allows for other types of decomposition information to be specified through `ALIGN` statements, including the ability to align a one-dimensional aggregate with one of the dimensions of a multi-dimensioned matrix, and the facility to denote that one matrix should be decomposed in a manner which makes it aligned with the transpose of another. While the addition of such a specificational feature to the Fortran 90 standard can be viewed as a slight compromise to that language's high level abstraction (i.e., the principle of coding without regard to the underlying machine), the assistance which such directives provide in the compilation process is significant in improving program performance.

Research Directions for DP Languages

Fortran 90 and HPF represent the state-of-the-art in DP languages as currently used for scientific computation. Evolution of the ideas introduced over three decades ago have resulted in languages which are both mature in expressive power and easily mappable onto a broad range of parallel hardware. However, as we will explore in the section which follows, the domain of problems for which such languages can provide efficient solutions is limited by the nature of the DP model which underlies them. Since the model was founded upon an assumption of regular parallelism, its expressiveness and efficiency do not extend to cases where irregular parallel forms appear within a program. Examples of these kinds of irregularities arise naturally in a number of important scientific problems. To allow for the expression of highly parallel forms for such practically significant problems, a number of languages have been proposed in which the general concepts of DP are present in a generalized form (c.f., the descriptions of the languages NESL and Adl in Sections 2.1.1 and 7.1 respectively). The bulk of this thesis is devoted to exploration of these alternative DP models, and towards the construction and evaluation of concrete instances of those alternatives.

1.3 Limits of Traditional Data Parallelism

The DP model and its traditional implementation under the SIMD and SPMD models provides a proven high-performance platform for the specification of parallel algorithms across large rectangular arrays of data. DP operations are provided which perform a serial per-element computation for every index of such an array in parallel. Others consider parallel operations across sub-selections of these regular data structures, or parallel reductions of array values. All up, the paradigm offers a broad range of parallel tools for modelling algorithms which exclusively make use of rectangular arrays.

While many scientific computations may be reasonably framed in such a form, there exists entire domains of scientific modelling — those which consider irregularly structured data — that are not amenable to expression under the DP model. Examples of such areas include:

- sparse-matrix computation (where grids are commonly represented as ragged arrays such as that shown in Figure 3),
- computational chemistry simulations (where molecules are represented by a structuring of simple aggregates),

- finite element computations across irregular meshes.

For these irregular problems of scientific computing the DP model offers only slight opportunities for parallel expression of operations. This is an artifact of the nature of the parallel operations available under the model: all deal exclusively with applying a serial computation across indices of a regular data structure. If we use DP operators to express an operation across the sparse matrix represented by the ragged array in Figure 3 — say, the multiplication of every matrix element by 2 — we must frame a parallelization either as a single DP operation across the outer (horizontal) vector, or a series of DP operations across the inner (vertical) vectors. In the first case we would be specifying a computation where 8 serial per-element computations are simultaneously active (each considering the indices of an inner vector in series). The second situation involves a sequence of parallel operations: the first comprising 4 simultaneously active per-element computations, the next consisting of 8, and so on. While each of these DP forms is a parallel expression of the problem, neither exposes the full parallelism of the problem — the multiplication of each of the matrix elements by a constant is completely independent of any other element's multiplication, therefore all 40 multiplications could be performed in parallel.

It is factors such as this which lead to the DP model being of limited use to scientific programmers considering the parallelization of their irregular codes. Instead, the developers of such systems are obliged to use either construct for themselves an appropriate parallel execution model for their program from low-level parallel constructs, or manipulate their problem to allow for an expression in some other high-level parallel paradigm (e.g., an implicitly parallel functional model, or a parallel object-oriented model). For many problems neither of these approaches is entirely satisfactory — a more attractive alternative would be the development of a generalized DP model which could support parallelism across irregular structures.

1.4 Thesis Overview

This thesis investigates ways of extending the simple DP model such that it can be made more amenable to the expression of irregular scientific problems such as those identified above.

Chapter 2 describes an extended DP paradigm which goes a long way towards catering to these application areas, by allowing DP operations across irregular data

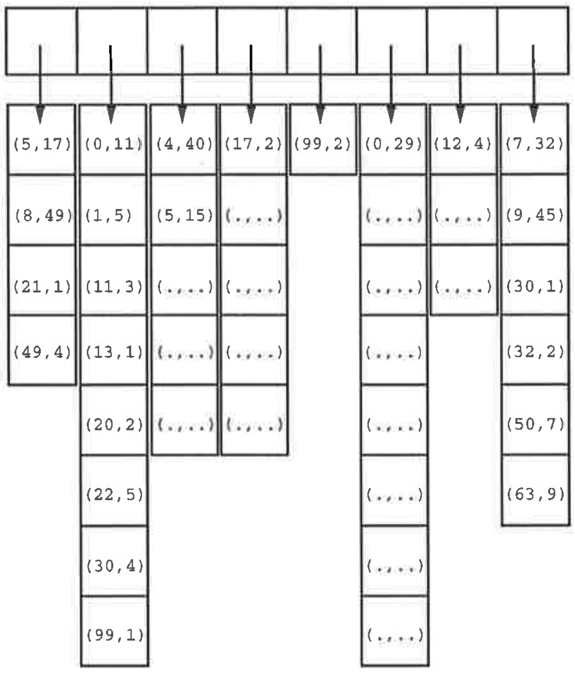


Figure 3. An Irregular Data Structure Representing a Two-Dimensional Sparse Matrix

to be expressed as a nesting of normal DP operations. We survey several previous approaches [22, 96, 26, 127] that have been undertaken in attempting to implement a nested DP model on traditional multiprocessor machines.

Finding that none of the extant methodologies provides support for all aspects of irregular computing, we undertake, in Chapter 3, a formal analysis of the DP paradigms under consideration in an effort to distill a generalized execution model. We present a comprehensive mathematical model of the DP and NDP paradigms, as well as formalizing the concept of distributed execution. The ultimate product of this analysis is a pair of NDP execution models which have guaranteed properties of deadlock-avoidance for any pattern of data layout.

It should be noted that, while the material presented in Chapter 3 forms an important part of the argument presented in this thesis, its subject matter is exclusively technical and largely self-contained. Readers may elect to skip over this discussion upon their first reading, concentrating on the products of our detailed mathematical analysis — the candidate NDP implementations (summarized in Section 3.5) — rather than the process by which they were derived. The reader may later return to the body of Chapter 3 in order to gain a precise understanding of the mathematical framework of definitions and derivations which underlies the thesis.

A practical evaluation of the candidate NDP implementations derived from our mathematical model is presented in Chapter 4. We focus on the predicted performance characteristics of each in the presence of irregular computation. At the end of our analysis, we choose one of the two candidates — a multi-threaded strategy — as the basis for subsequent work, based upon its capacities for masking latencies introduced by program irregularities.

Chapter 5 describes an abstract machine which serves as a realization of the chosen (multi-threaded) model on a multiprocessor distributed memory machine. Our abstract model is highly flexible and permits generic specification of DP operations which may operate over vectors forming part of a complex data-structure.

Considerations germane to the implementation of this model are discussed in Chapter 6, in which a prototype implementation targetting the Thinking Machines CM-5 [124] is presented. The implementation includes several features specifically catering towards irregular patterns of execution, including the provision for arbitrarily distribution of aggregate elements across the machine. The execution model embodied by this implementation bears some similarity to a number of previous multi-threaded architectures [86, 30, 113, 99, 61, 43, 87, 8], both hardware and software. Chapter 6 presents a broad comparative survey of such systems in order to characterize their relationship to our model.

As a case study describing how our execution model supports the irregular extensions to the DP model, Chapter 7 describes how a simple language based upon this extended model may be efficiently mapped onto our CM-5 environment. The performance of the language implementation is analyzed in Chapter 8, where a number of regular and irregular computations are benchmarked. To allow comparison, we consider the performance of equivalent programs expressed in an optimized traditional DP system and another research prototype implementation of the extended DP model.

Finally, Chapter 9 draws conclusions from the work reported, highlighting a number of significant contributions we have made in providing a highly parallel basis for the DP specification of irregular scientific programs. We also discuss several areas in which the existing systems and methodologies could gainfully be further investigated.

Chapter 2

Nested Data-Parallelism

In the previous chapter we saw that, despite its successes at providing a high-performance basis for simple and regular scientific computations, the traditional paradigm of Data-Parallel execution is not easily applied to the parallelization of less regular computations. Specifically, the flat model presented by these paradigms does not mesh well with the structured data aggregates (potentially of irregular or dynamically altering size) found within irregular scientific algorithms. This mismatch makes it difficult (or impossible) for a programmer using the flat DP model to build a computational form of such an algorithm which exposes and makes use of all the parallelism inherently available within the algorithm. That is, due to limitations in the model's expressibility, many opportunities for parallel execution are lost.

A simple generalization of the traditional DP model, *Nested Data-Parallelism* [22, 17, 96, 26], has been proposed as a means of augmenting DP expressibility to eliminate many of these forced serializations. This chapter introduces the paradigm and describes how its application leads to natural and highly-parallel specification of parallel algorithms across irregularly-structured data. Following this is a discussion of techniques that have been adopted to implement the generalized scheme on normal multiprocessor architectures. None of the approaches is without flaw and none precisely meet the requirements of irregular scientific computing. Thus, we close the chapter with a discussion of the need for a more general execution model to underlie the Nested Data-Parallel model.

2.1 Concepts of Nested Data-Parallelism

The basis for the paradigm of Nested Data-Parallelism (NDP) lies in a simple generalization of the traditional (or *flat*) model of Data-Parallelism. The flat model derives parallelism by applying the same sequence of computation to every element of a data aggregate in parallel, insisting that this per-element computation is purely serial. The NDP paradigm, conversely, allows for the per-element computation of a DP operator to itself contain parallel constructs in the form of other DP operator instances. That is, the paradigm permits DP operators to be *nested* in a manner analogous to the nesting available for other language constructs (e.g., serial loops).

```
A1 = [1,2,3,4,5]
A2 = [3,7]
A3 = [2,2,2,2]
A = [A1,A2,A3]

function g(x:integer -> integer) = x+1;

function f(v:integer[] -> integer[]) =
         map (g,v);

B = map (f,A);
```

Figure 4. A Nested Data-Parallel Program

To illustrate the concept, consider the example program (written in a simple functional DP pseudo-code) shown in Figure 4. This program considers a parallel operation across an aggregate `A` made up of a nesting of one-dimensional vectors (i.e., a *ragged array*) which computes a new vector nest whose elements have values one greater than the corresponding entries in the source. The operation is specified in terms of a simple "apply-to-all" DP operator called `map`. The semantics of this operator are such that it accepts two arguments, a function and a vector and applies the function to every element of the vector *in parallel*, returning a vector formed from the results of the function calls. In the last line of the sample program we see this operator applied to the ragged array `A` — this expression describes a parallel operation in which three calls to the function `f` are made in parallel, the first receiving the argument `A1`, the second receiving `A2` and the third receiving `A3`. This is standard

Data-Parallel execution. What distinguishes this program as NDP, is that the per-element computation of this operator (i.e., the function `f`) itself contains a DP operator in the form of another `map` expression. This function definition stipulates that for each of the calls to `f` (themselves executing in parallel), a number of calls to the function `g` (one per index of `f`'s argument vector) should be made in parallel.

It is important to note that the semantics of such a nesting of DP operations implies the existence of *multiple* levels (or dimensions) of parallelism. In the example code, two levels of parallelism exist: parallelism between instances of the function `f` and parallelism within each instance of `f` (between invocations of the function `g`). We describe the former as comprising the *outer parallel dimension* of the computation while the latter we term the *inner parallel dimension*. How such multi-level parallelism is mapped onto a real-world multiprocessor is a question of implementation, and will be discussed in Sections 2.2 and 2.3 below.

2.1.1 Nested Data-Parallel Programming Languages

A number of languages support the specification of NDP operations through the structured composition of flat DP operators. These we term *Nested Data-Parallel languages*. Several such languages are described briefly below in terms of the constructs they provide and the scope they offer for NDP expression.

NDP Fortran Dialects

As described in Section 1.2.1, there exist many dialects of Fortran where DP features are added either through the provision of special indexing instructions or through parallel loop constructs across arrays. Those that offer loop constructs (e.g., CM Fortran with its `forall` construct) often cater for their nesting within a program. This, in essence, defines an NDP computation. However, while such languages permit the specification of multiple dimensions of parallelism within a computation, their compilers are rarely capable of exploiting such parallelism. This is typically due to the analytical complexities arising from the fact that these are imperative languages — it is often difficult for a compiler to determine which of the instances within the nested parallel loop have interdependencies. Faced with such difficulties compilers usually adopt a simple-minded serialization scheme for such NDP loops (such as those described in Section 2.2)[1].

SISAL

The SISAL [41, 116, 78] and SISAL 2 [24] languages have their basis in the high-level expression of dataflow semantics and are not explicitly DP languages. They do, however, provide a number of DP-like operations across the language's vector and array types. Particularly of interest is the parallel `forall` iterator which allows a loop body to be instantiated independently for every element of a range or for every index of a vector. This construct comprises three parts: a generator which specifies the range of the iteration, a loop body which details the per-instantiation computation, and a return value section which describes how the results from the loop body instances should be combined to form a single result value for the loop itself. This combination step represents a reduction operation similar to those found in most DP languages (c.f., the rank-reducing operations described for flat DP languages in Section 1.2): SISAL offers a typical selection of combination functions for this reduction including summation, selection based on some criteria (e.g., return the minimum result), and the construction of an array from the set of individual results.

SISAL permits the nesting of `forall` loops in two ways. The first is through the complete textual encapsulation of one loop within a loop body expression of another loop. The second nesting form makes use of a special syntactic form in which a single body and return value section are specified but the generator includes multiple ranges (separated by the keyword `cross`). Both forms of specification are completely general and may be used to specify NDP iterations across irregularly structured data.

Paralation Lisp

Paralation Lisp [103] is a dialect of Common Lisp which adds a new data constructor, the *field*, which allows for the specification of an array-like ordered collection of elements. Field elements may be of any type including field types, thus allowing for the construction of irregular nested aggregates. The language permits type heterogeny between elements of the same field.

[1]For example, although the CM Fortran language allows for nested `forall` loops, the current compilers for the language [127, 128] treat such constructs conservatively, making no attempt to parallelize either the outer or inner loop.

Two DP primitives are defined across Paralation Lisp's fields: the `elwise` operator, and the `<- (move)` operator. The former represents a parallel iterator which accepts a set of fields and a *body expression*, computing a result field by applying the body expression to each element of each argument field. This is equivalent to one or more instances of the apply-to-all operator `map` introduced in the pseudo-syntax of our earlier example.

A second DP primitive offered by the language, the `move` operator, allows for a generalized permutation (not necessarily one-to-one) of a field to be computed in parallel. The operator accepts a source field and a `mapping` (computed using the Paralation Lisp primitive `match`). This latter argument may be considered as a bundle of one-way arrows connecting elements of the source field with positions in the resultant field. During the move operation, each element of the source is pushed down the arrows extending out from its location. Several arrows departing the same location implies a concurrent read (duplication) of the source element; multiple arrows arriving at a single result position implies a combination (i.e., a reduction) of several source elements. The latter situation is handled by the user specification of a reduction function to combine colliding elements.

Both `elwise` and `<-` operations provide opportunities for general expressions to be defined as per-element computations (for `elwise` this constitutes the body of the iteration, for the move operation it derives from the possibility for a user-specified combination function). Specifically these per-element expressions may themselves contain `elwise` and/or `<-` operators, thus expressing NDP computation. Given the nature of the field constructor (as a one-dimensional aggregate), such nesting is important to the specification of parallel operations across the nested fields used to represent complex (possibly irregular) data structures. An NDP implementation of a (subset of) Paralation Lisp is described in [22] and [17].

Nesl

The NESL [18, 20, 19, 26] language of Blelloch is designed specifically with NDP specification in mind. Parallelism in the language arises explicitly through operations across homogeneous *sequences* of values. Irregularly structured data may be represented as a nesting of sequences; parallel operations across such aggregates may be specified by a nesting of the operators available for iteration across simple unstructured sequences.

The fundamental DP operation in the language is the functional flat DP `apply-to-all` construct specified by a syntax reminiscent of the list-comprehensions provided by Miranda [131] and Haskell [58]. The NESL expression:

```
{ negate(a) :  a in [3, -4, -9, 5] };
```

demonstrates this construct in a flat DP context, specifying the parallel application of a function `negate` to each element of a sequence of length 4. As with list-comprehensions, NESL's `apply-to-all` may also include a clause for sub-selecting (filtering) which elements of the source sequence to consider in creating the result. For example, the expression:

```
{ negate(a) :  a in [3, -4, -9, 5] | a < 4 };
```

is a specification that the function `negate` should only be activated for the three sequence elements whose values are less than 4.

The language also provides parallel permutation operations similar to the move operator of Paralation Lisp. The NESL construct `->`, called a *parallel get*, accepts a `value` sequence (of any base type) and an `index` sequence (of base type integer) and computes in parallel a new sequence derived by selecting, for each position in `index`, the value at that index of `value`. For example:

`values` = $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$
`indices` = `[3,5,2,6]`
`values -> indices` = $[a_3, a_5, a_2, a_6]$

A related operation, the *parallel put* (denoted `<-`) is also supplied by NESL. This operator takes a sequence of values (the *destination sequence*) and a sequence of `(integer, value)` pairs. Each element `(i,v)` in the latter sequence causes the i^{th} element of the destination sequence to be overwritten with the value `v`.

Finally, the language also provides a number of important DP operations to compute *parallel prefix* (or *scan*) operations [17] across sequences. These operators, (`plus_scan, max_scan, min_scan, or_scan, and_scan`) compute in parallel a new sequence which contains values which are partial accumulations of the elements of the original using a particular accumulation function. For example, the `plus_scan` operator, when applied to a sequence, generates the sequence of all partial sums — that is, the i^{th} element in the result sequence is the sum of all source sequence elements up to the i^{th} position.

While most of NESL's DP constructs do not allow for general per-element expressions, and thus do not allow for nesting, the `apply-to-all` operator clearly does support such a paradigm. This is the source of the languages NDP features: if the expression to the left of the colon in the `apply-to-all` form itself involves a DP operation (whether it be a put, get, scan, or another `apply-to-all`) the computation is NDP. The add-one operation across the ragged array structure outlined in Figure 4 can be concisely written in NESL as:

```
B = { { i+1 :  i in v } :  v in A }
```

Section 2.3 below describes a strategy for the implementation of NESL's NDP features.

Comparative Summary of NDP Languages

As we have seen in the preceding discussion, most NDP languages implement a common set of DP operations — an apply-to-each construct, a parallel permutation, and several reduction and parallel prefix operations. The flexibility afforded the programmer in the specification of such operation, however, varies between languages and between operators within the same language. Whereas a programmer may be allowed to specify an arbitrary computation as the per-element code for one type of DP construct, the same language may restrict the choices of per-element code for another construct to a small set of system-defined options. Table 1 summarizes, for each of the NDP languages considered thusfar (and also for the Adl language introduced in Chapter 7), the availability of different constructs and the flexibility afforded in their use.

DP Operation	CM Fortran	SISAL	Paralation	NESL	Adl
apply-to-each (body code)	yes (user-def)	yes (user-def)	yes (user-def)	yes (user-def)	yes (user-def)
permutation (combiner)	yes (+,max,...)	no	yes (user-def)	yes (system-def)	no
reduction (combiner)	yes (+,max,...)	yes (+,max,...)	yes (user-def)	yes (+,max,...)	yes (user-def)
scan (combiner)	yes (+,max,...)	no	no	yes (+,max,...)	yes (user-def)

Table 1. Comparing DP Operations Available in NDP Languages

A DP operation in which the programmer is permitted full flexibility in the specification of the per-element computation is marked "(user-def)" in the table, while those which restrict such specification to a small set of system-defined options appear with the label "(system-def)" or a sample of the valid options "(+, max, ...)". It is important to note that the introduction of NDP into a program is only possible through use of an operator which allows user-specified per-element computation. Thus languages which provide a richer vocabulary of such operators allow for greater range of NDP expression.

2.2 NDP Implementations via Serialization

As we noted earlier, one of the defining characteristics of the NDP paradigm is that it naturally introduces multiple levels (or dimensions) of parallelism into a computation. In implementing the NDP paradigm upon a conventional multiprocessor architecture, it becomes necessary to map these independent dimensions of parallelism onto the single real dimension of parallelism offered by the hardware.

The most straightforward approach to this mapping problem — one adopted in many compiler systems (e.g., [127, 137, 125]) — is to seek a means of expressing the parallelism afforded by an NDP expression in terms of the (well understood) flat DP model. Once in this form the computation has only a single dimension of parallelism and is thus readily mapped onto a hardware implementation. We call such an approach an *implementation via serialization* because it reduces the parallelism within the computation by serializing all but one parallel dimension of the computation.

```
A1 = [1,2,3,4,5]
A2 = [3,7]
A3 = [2,2,2,2]
A = [A1,A2,A3]

function f(v:integer[] -> integer[]) =
   { for-serial i = 1,...,length(v)
        result[i] = v[i] + 1;
     return (result);}

B = map (f,A);
```

Figure 5. The Inner Serialization of the Ragged Array NDP Code

If we consider the simple example NDP code from Figure 4 it is clear that there are two parallel dimensions expressed by the computation. One possible implementation method is to arrange for a compiler to serialize the inner instance of the `map` DP operation, leaving one parallel dimension (that afforded by the outer `map`). Operationally, this leaves us with a program equivalent to that shown in Figure 5. This program fits into the flat DP model since it incorporates a single DP operation whose per-element computation is purely serial.

An alternative approach a compiler might adopt in implementing our simple NDP program is to leave the inner DP operation as a parallel computation and serialize the outer iteration of the nest. This yields a computation operationally equivalent to the form shown in Figure 6. Again, this program fits into the confines of the flat DP paradigm since it can be viewed as a sequence of flat DP operations. Specifically, the serialized form does not violate the flat DP assumption that only a single DP operation is active at any point in time.

```
A1 = [1,2,3,4,5]
A2 = [3,7]
A3 = [2,2,2,2]
A = [A1,A2,A3]

function g(x:integer -> integer) = x+1;

function f(v:integer[] -> integer[]) =
         map (g,v);

B = { for-serial i = 1,...,length(A)
        result[i] = f(A[i]);
     return (result);}
```

Figure 6. The Outer Serialization of the Ragged Array NDP Code

It is clear than all approaches which implement NDP by reduction to a flat DP form inherently involve the elimination of some parallelism available for exploitation in hardware. Whether or not this reduces the performance obtained in the execution of a NDP program on a particular hardware architecture depends on the size of the unserialized dimension relative to the number of processing elements within the machine. It is possible for an NDP specification to offer more parallelism than may be realized on a given number of processors; in that case, a serialized version of the code may not suffer a significant performance penalty due to the loss in available parallelism. Alternatively, a serialization may reduce the amount of parallelism within an NDP code to a level where the processors of the machine are not kept busy. In that situation a loss in performance is inevitable.

2.2.1 Perils of Serialization

The serializing approach to implementing NDP places a heavy burden on the NDP compiler — when compiling an NDP expression it must decide which of the available parallel dimensions is to be preserved and which others are to be serialized. The ultimate performance displayed by the program may be seriously affected by this decision since it indirectly determines how much parallelism is available to be exploited within the computation. It is clear that it is desirable that the parallel dimension chosen for preservation by the compiler be as large as possible. In some cases the exact extents of the dimensions may be statically determinable (e.g., a parallel loop from 1 to 100) although typically the extents of any given parallel dimension will depend either on the size of an aggregate in the program, or upon the dynamic value of some scalar variable or aggregate element. That is, in most cases this information is not available statically and thus the compiler cannot make use of such precise knowledge of the situation. Instead, compilers must adopt a heuristic approach (driven by analysis of patterns of aggregate access) in deciding upon a serialization. In general such conclusions, based on imperfect knowledge of the actual situation that will prevail at runtime, may prove unreliable, leading to poor performance of the compiled program.

The situation becomes more complex when we consider programs whose data (over which NDP computation is defined) changes shape as a computation unfolds. Consider, for example, the recursive functional quicksort program shown in Figure 7 for NESL. The argument to the quicksort function, `s`, represent a sequence of items to be sorted. The first step of the computation checks the length of the input sequence (`#s`) to see whether a singleton has been passed. Assuming that the sequence contains at least two elements, the function continues by selecting a random element as a pivot (line `5`) and computing the subsequence of `s` which contains elements whose value is less than `pivot` (called `les`), the sub-sequence of elements greater than `pivot` (called gtr) and the subsequence of elements equal to pivot (lines 5, 6 and 7 respectively). All of these subsequence computations are DP operations, making use of the apply-to all operator with its selection clause.

Line 8 of the program specifies the NDP section of the computation: a nested sequence of length 2 is formed (the first element is the subsequence les, the second is the subsequence gtr) and a DP operation is specified across this new sequence. The per-element computation performed during that operation is a recursive call to qsort which is, as we have seen already, a function containing parallel code. Thus the specified operation is NDP. The value returned by this operation (and stored in result) is a new nested sequence of length 2, the first element of which contains a sorted version of les, the second containing the sorted form of gtr. Thus to compute the total sorted form of the input sequence s, all that is required is for the sorted version of les (i.e., result[0]) to be concatenated with the subsequence of s which had values equal to pivot and further concatenated with the sorted form of gtr (i.e., result[1]).

```
1     function qsort(s) =
2     if (#s < 2) then s
3     else
4       let pivot = s[rand(#s)];
5           les = {e in s| e < pivot};
6           eql = {e in s| e == pivot};
7           gtr = {e in s| e > pivot};
8           result = {qsort(v):  v in [les,gtr]}
9       in result[0] ++ eql ++ result[1];
```

Figure 7. A Recursive Functional Quicksort in NESL

Figure 8 shows the tree of calls to qsort generated for a top-level call to sort a sequence of ten elements. Within each of these blocks, there exists parallelism in the form of the operations to split the sequence around the pivot (lines 5, 6, and 7). Between blocks at the same level of the tree there is a second dimension of parallelism. This is introduced by the apply-to-all operator in line 8 which specifies that both recursive calls should be made in parallel with one another.

If we consider the serializing implementations of this simple NDP code it quickly becomes apparent that neither the outer nor the inner serialization is completely satisfactory. If we serialize the inner operations (i.e., the DP operations within a

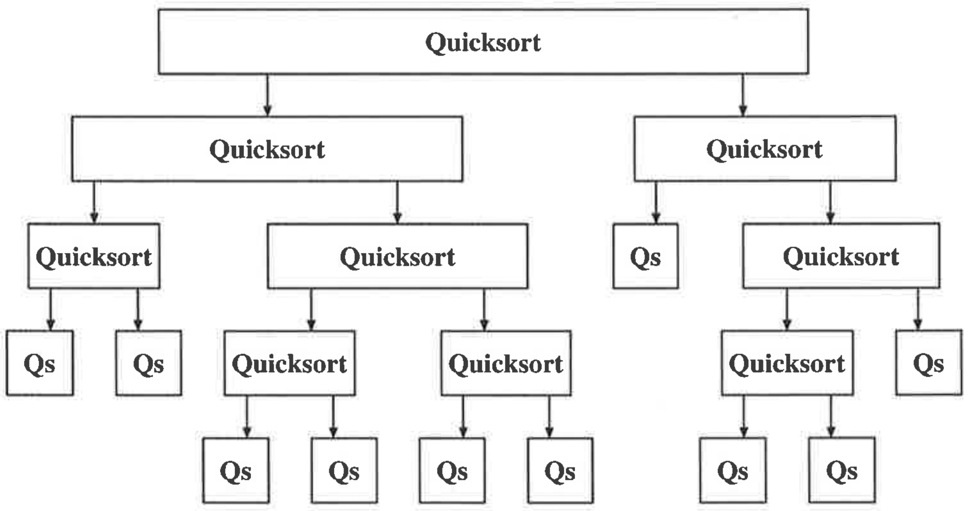


Figure 8. A Call Tree for the Recursive Quicksort

block) then the parallelism of the computation is very poor during early steps, due to the small number of blocks. Near the leaves of the tree, however, this serialization retains a good amount of parallelism: there are many blocks and the (serialized) work contained within each is small[2]. If we consider the outer serialization of the `qsort` (i.e., the serialization of parallelism between blocks) we note the opposite trend in parallelism. At the root of the tree the blocks are large, indicating a large amount of available parallelism, while towards the leaves of the tree the blocks (and thus the opportunities for parallelism) become very small.

Figure 9 illustrates these trends. The graphs, obtained via a simulated execution of the various serializations[3], display the degree of parallelism at various times during computation of quicksort over ten elements, for both the inner and outer serialization. Also shown at the bottom of the figure is the theoretical parallelism that would be attained by exploiting both parallel dimensions simultaneously. In these parallelism profiles, the horizontal size of a block represents the amount of serial work for a given application of the function `qsort`: we assume that it is possible to construct

[2]Whether this situation leads to good performance or not depends on how efficiently the implementation can manage and/or schedule the large number of very fine-grained computational elements.

[3]The simulated parallelism profiles were obtained by considering the units within the Quicksort call tree (Figure 8) and their inter-dependencies. The blocks from the call tree were then placed upon the graph axes in positions which adhered to the serialization under consideration and also maintained inter-block dependencies.

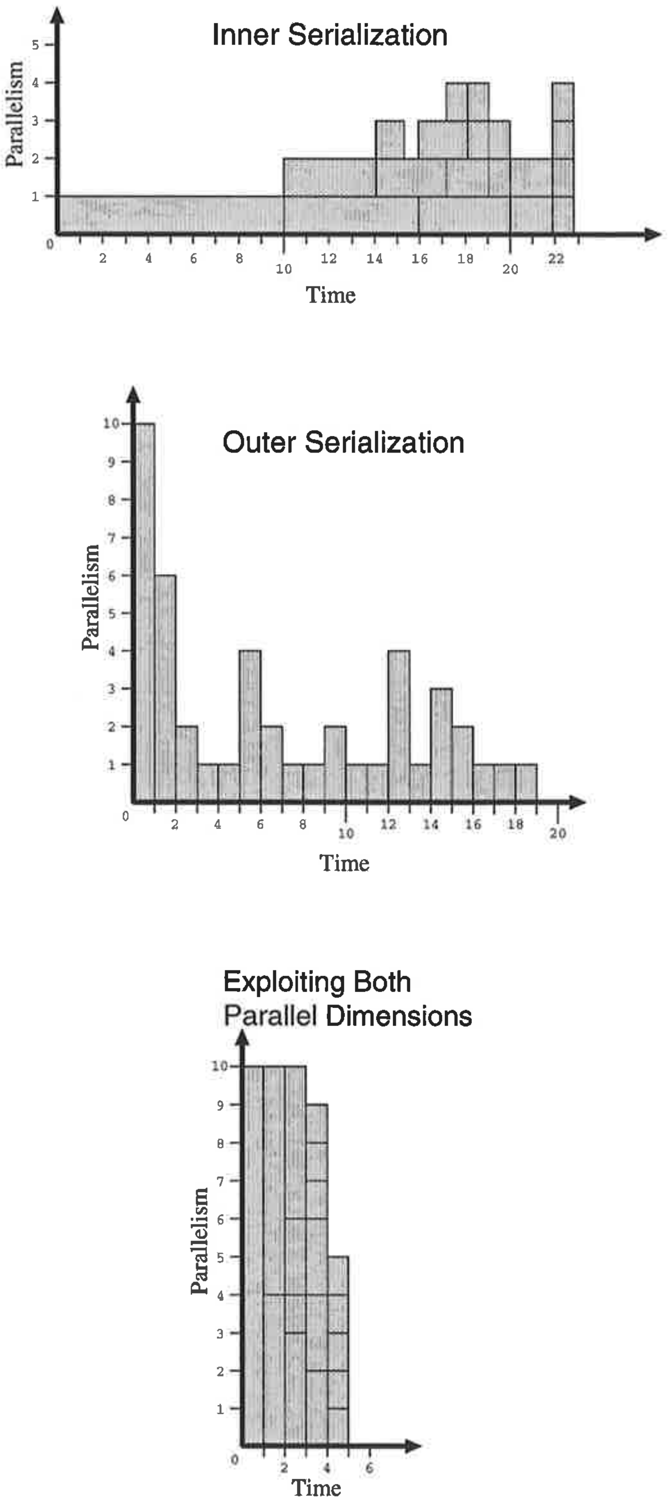


Figure 9. Parallelism Profiles for Different Implementations of Quicksort

a serialization which is $O(n)$ in the size of the input vector. The degree to which blocks are vertically stacked indicates the number of evaluations which are occurring in parallel. As motivated above, the outer serialization approach displays poor parallelism at the beginning of the computation but good parallelism later in the execution; the opposite general trend is observed for the inner serialization.

It is clear that the changing sizes of the sequences passed to `qsort` at different times during the computation leads to differing demands as to which serialization is preferred. This demonstrates that for such a program none of the available serializations will deliver optimal performance — the sample serialization-based executions shown in the figure consume 23 and 19 time units, compared to a theoretical minimum execution time of 5. To implement programs with this style of dynamically changing (or unfolding) parallelism in a manner which delivers a performance close to this optimum, the implementation must necessarily exploit all available levels of parallelism.

2.3 Structure Flattening

The serializing implementations of NDP we have discussed so far have been based around the philosophy of applying a transform to constructs with more than one dimension of parallelism, to yield a semantically equivalent expression which has a single parallel dimension. These resultant expressions can be implemented directly in terms of the unstructured DP model. A variation upon this approach is the *structure flattening* methodology described by Blelloch *et al.* [17, 22, 20, 96]. This approach similarly focusses upon translating NDP constructs into forms expressible in a simpler DP model; however, rather than targetting the traditional unstructured DP model of computation, Blelloch's structure flattening transformations make use of the more expressive *segmented vector model* [17]. This model adds one important feature to traditional the DP paradigm: the ability to specify that a data aggregate (vector) is logically segmented into a number of sub-aggregates (segments). DP operations across segmented aggregates treat each segment as an independent unit of data, computing results for each segment concurrently but independently. Operators which involve accumulations only consider the accumulation of values within a single segment: there is never a carry over of values from one segment's computation to another's.

```
A = [0,1,2,3,4,5,6,7,8,9]
seg = [2,3,1,4]

B = segmented_plus_scan (A,seg)
⇒ B = [0,1 |,| 2,5,9 |,| 5 |,| 6,13,21,30]
```

Figure 10. A Segmented scan Operation

Figure 10 illustrates the concept of a segmented flat DP operation. The vector A is a simple one-dimensional aggregate containing 10 integer values; seg is a description of how those ten indices of A are logically divided into four segments. According to the specification, the first segment of the vector A consists of its first two indices, the second segment comprises the next three, the third contains only a single index (with value 5), while the final segment contains the remaining four indices. We consider a segmented DP operation **segmented_plus_scan** which computes partial sums in the same manner as the normal plus_scan DP operator, but which treats the computation across each segment of the vector A as separate. That is, the operator generates a sequence of values which are the partial sums of source indices within a particular segment. The bottom section of the figure shows the results of the segmented scan (the segmentation of the result is shown with | symbols). The first two result indices are the partial sums from the first segment of A (i.e., the values 0 and 1). The next three result values describe partial sums of A's second segment (the values 2, 3 and 4). The single index which follows is the partial sum of the singleton third segment, while the remainder of the result vector describes the partial sums of the final segment (the values 6, 7, 8 and 9).

Considering this example operation from a slightly different perspective, the motivation for utilizing the segmented vector paradigm becomes apparent. The computation we have just described can be considered to be operationally equivalent to a code in which four flat scan operations are concurrently executed, each across a different segment (sub-vector) of the vector A. The property of the segmented vector model which indicates that each of these individual segment-operations must be performed entirely independently, implies a conceptual parallelism between these operations. That is, this specification is effectively equivalent to an NDP computation (an apply-to-all whose body is a flat scan operator).

A number of implementations [17, 22, 96] of NDP have sought to use this equivalence to provide a mapping from multiple conceptual dimensions of parallelism to a single hardware dimension. The compilation technique adopted in these systems, called *structure flattening*, involves reducing any nested aggregates present within an NDP program into a single one-dimensional vector of values. The original structure and dimensionality of each aggregate is preserved through a series of n segment descriptor vectors (where n is the dimension of the original aggregate). The values contained within the data vector are all those present in the innermost dimension of the original structure, ordered so as elements from the same original vector are adjacent. The first segment descriptor describes the structure of the innermost level of original nesting, the second descriptor specifying the structure of the next level out, and so on.

The principle is best illuminated by example. Consider the data structure shown in Figure 11(a). The structure flattening technique translates this structure into the three vectors shown in Figure 11(b). The `data` vector in this diagram shows the single vector of values produced by this method, while the `segdes1` vector describes the lengths of the innermost nested sub-vectors. The `sedges2` vector describes the entire original aggregate as a single segment.

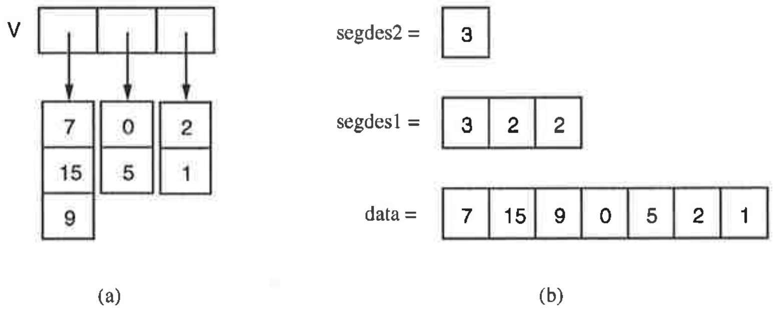


Figure 11. Example of Blelloch's Structure Flattening

Associated with this process of flattening aggregates into segmented vectors is a second translation procedure which converts NDP operations within a program into equivalent segmented DP operations over the flattened data. In the case of the CMU structure flattening implementation of NESL, this is achieved by modifying those user-defined functions referenced within a DP operation such that all sub-steps of the nested call become either segmented vector operations or calls to other previously-transformed user functions.

The structure flattening approach to implementing NDP represents a significant improvement over the serialization-based implementations described in the previous section. The amount of parallelism presented by an NDP operation for exploitation in hardware is based entirely on the length of the data vector for the source aggregate, regardless of how that vector is logically segmented. If we consider again the quicksort NDP program from Figure 7 we can see that this equates to the simultaneous exploitation of parallelism from more than one dimension. Operations within an invocation remain parallel (they are the per-segment operations in the transformed version) while parallelism between invocations is also exploited (more than one segment is being computed simultaneously). Thus the flattening-based implementation of this program displays optimal parallelism both at the root and leaves of the call tree.

2.3.1 Conditional Execution

The fundamental assumption made by the mapping of NDP computation onto the segmented vector model is that an NDP operation may be characterized by a number of *structurally identical* operations applied in parallel. Such an operation is easily translated into the application of a segmented version of that operation applied across a flattened form of the original aggregate.

While many NDP codes adhere to this kind of behaviour, the fact that languages allow for arbitrary user-defined computation (possibly including conditionals) to be inserted as the per-element computation of a DP operator can lead to difficulties. Consider for example the NESL code shown in Figure 12. In this program the inner computation of an NDP is made conditional upon the length of an inner vector. If the vector is of length less than two, a vector of partial sums is generated using the **plus_scan** operator; otherwise an **apply-to-all** operation is executed to generate a result value. It is clear that in this program the per-element computation of the outer **apply-to-all** operator violates the assumption of structural homogeneity: it may occur that when executed, certain of the inner computations will invoke instances of **plus_scan** while others will follow the control path which applies the inner **apply-to-all**. That is, the operations applied in parallel may not necessarily be invocations of an identical operator. Because of this, the program cannot be directly implemented by a process of flattening its structures and converting its NDP operations into segmented vector operations.

```
function cond (vec) =
  if #vec < 2 then plus_scan (vec)
  else {x+1:  x in vec}

function do_stuff (vvec) =
  {cond(z):  z in vvec}
```

Figure 12. Mixing NDP and Conditionals

To implement conditional execution within NDP operations, implementations based on structure flattening introduce an operation called *packing* which is reminiscent of techniques applied in vectorizing compilers (such as described in Chapter 5 of [107]). The essence of the methodology is that the compiler inserts code which, at the time that a conditional is evaluated, creates a temporary segmented vector which contains only those segments for which the predicate evaluates true. This temporary vector is distributed across the entire machine by communications operations. The segmented operation defined by the true branch of the conditional is then applied to that temporary vector, the results being copied back into the vector of results under construction (again, potentially involving communication). An analogous process of temporary vector creation, distribution, application and gathering then takes place for segments for which the predicate was false.

In general, this protocol of packing-based conditional execution is expensive (see the experimental evaluation and discussion in Chapter 8, where the technique is shown to introduce large overheads into a number of executions) as a result of the communications induced by the redistribution and gathering phases. Techniques for optimizing or eliminating such operations remain an active field of research.

2.4 NDP as an Efficient Basis for Irregular Scientific Computing

As we have described previously, one of the principal strengths of the NDP model of computing is its ability to concisely express parallel forms for irregular algorithms.

Such algorithms are typically difficult or impossible to parallelize under traditional DP models. Given this significant power of expressiveness for irregular cases, it seems logical that an implementation of the NDP model should provide efficient execution for irregular codes.

In Sections 2.2 and 2.3 we have described the two prevalent implementation models for NDP, each with their own performance idiosyncrasies. We now turn to the evaluation of each of these techniques from the perspective of providing good parallel performance for irregular codes. The metrics we use to qualitatively gauge this performance are the degree of parallelism obtained by the execution of the NDP code, and the overheads (such as communications cost) introduced in supporting the irregular aspects of the computation.

Considering the broad application area of irregular scientific computing, we note that a computation may display irregularities in one of three forms:

1. irregularly structured data aggregates, possibly of dynamically varying size or shape;

2. irregular (possibly statically unpredictable) aggregate access patterns;

3. non-uniform paths of execution, introduced through the presence of conditionals.

For an implementation of NDP to be a successful basis for irregular computation it must provide some degree of low-cost support for each of these types of computational irregularity.

From our discussion of serialization (Section 2.2.1) it is clear that the implementations of NDP via simple serialization of parallel dimensions do not perform well in the presence of the first type of irregularity described above. Specifically such implementations are likely to be unable to exploit the maximum parallelism available in an NDP code with this kind of irregularity, leading to performance which is sub-optimal. Based on this observation we would consider such implementations a poor basis for irregular scientific computing.

The structure flattening approach is specifically designed to overcome the problems faced by serialization solutions. It realizes a maximum amount of parallelism regardless of the shape or dimensionality of the data being worked upon, even in the case where these quantities vary dynamically. This property is enough to make the structure flattening implementation a good basis for codes whose irregularity is of the first type enumerated above, that is irregularity through data structure.

The technique is significantly less supportive of the other two forms of irregularity. Irregular patterns of access are, to some degree, considered by the model in its provision for optimized parallel *put* and *get* operations. However, the most common form of access — the simple aggregate dereference — is not specifically optimized. On a parallel architecture for which memory access is non-uniform (e.g., a distributed memory multiprocessor) the cost of a simple dereference can vary widely depending upon which memory space stores the specified aggregate element. Optimizing access patterns in such systems becomes a task of arranging for the memory spaces accessed by processor i to be as "close" as possible to processor i; for aggregate dereference this becomes a task of *partitioning* the aggregate across the various memories such that those elements required by processor i are contained in a memory close to that processor.

If we consider irregular patterns of access, this means we must consider irregular patterns of data decomposition. There is no support for such complex data placement strategies in the current NDP implementations based on structure flattening: the single data vector that results from the flattening process is partitioned into several *blocks* of adjacent elements with each block assigned to the memory of one processing element. This decomposition is clearly very regular and leads to high overheads in codes which rely heavily on irregular patterns of aggregate dereference, since such accesses must frequently employ slow (communication-based) mechanisms to retrieve values from non-local memories. Chapter 8 presents a quantitative analysis of the remote access costs incurred by several irregular NDP programs due to various schemes of data partitioning.

The structure flattening implementation of NDP also does not fare well in the presence of the third type of irregularity we have noted above — non-uniform control paths. The presence of conditional execution within an NDP operator must be handled in the structure flattening implementation by recourse to its protocol of packing. As described in the preceding section, this operation has the potential to introduce high overheads due to costs involved both in distributing each of the packed sub-vectors across the nodes of the machine and gathering the result values back to the appropriate memories. These operations are communication-intensive, and are a major source of inefficiency in control-irregular NDP codes implemented using structure flattening.

Based upon the analysis we have made of the two existing implementation techniques for NDP, we can draw several significant conclusions. Firstly, it is clear that neither of the existing approaches is supportive of all three types of irregularity that we have identified as common in irregular scientific computing. Therefore, we assert, none of the existing techniques implements the NDP model in a manner which realizes its potential as a basis for highly parallel, efficient irregular computing. The analysis we have undertaken in this section does, however, offer some pointers as to what qualities an NDP implementation should possess to better cater to this domain of parallel computing. Such an implementation should:

- realize a high degree of the parallelism inherent in an NDP operation, even in the presence of irregular data whose extents are statically unknown or which are dynamically varying
- support complex or arbitrary decompositions of aggregates across memories (to allow for expression of a low-cost partitioning in the presence of irregular patterns of aggregate access)
- allow for heterogeneous simultaneous application of DP operations (to provide an efficient basis for control-irregularities)

In the chapters which follow we outline the design and realization of an NDP implementation strategy which embodies these three principles and is thus, we propose, a good basis for irregular scientific computing.

Chapter 3

Designing a New Paradigm for NDP

In the previous chapter we considered the paradigm of Nested Data-Parallelism from the point of view of efficient implementation of irregular computation. In our analysis we highlighted a number of limitations inherent in current approaches to compiling NDP. These included: a lack of support for complex data decomposition, and poor ability to support evaluations in which a number of heterogeneous operations are simultaneously active. While these limitations are not a serious hindrance to efficient execution of regular NDP codes, programs which display any degree of irregularity — in either data structures or computational patterns — are severely penalized in their observed performance. In short, the current implementations of NDP do not provide an efficient framework for executing irregular computation of the kind found in many important areas of scientific computing (e.g., sparse matrix operations, finite element meshes, computational chemistry kernels).

In this chapter we turn our attention to the design of a new NDP implementation whose characteristics make it a better substrate for irregular execution. We specifically consider implementations which target distributed memory multiprocessor machines, since these are the architectures for which the DP philosophy is particularly well-suited, although many of our design considerations are equally applicable to shared memory machines.

We begin our process of design by describing the fundamental principles of NDP and distributed execution in terms of a mathematical model which permits convenient proofs of several important properties of different possible NDP implementations. The model we propose is simple (based on set theory and a generic concept of algorithmic execution), concise, and tailored specifically to the task of expressing key aspects of NDP executions such as potential for deadlock.

Building on the mathematical foundation this model defines, we construct a series of NDP execution paradigms, each of which possesses certain provable properties. We begin with the construction of a paradigm which defines the minimum semantics for DP execution, then note some deficiencies and propose augmented paradigms which fully describe DP computation. At the end of this analysis we are left with two NDPs paradigm which may provably execute any NDP operation to completion irrespective of the distribution of computation across the machine. One is a single threaded paradigm; the other incorporates multi-threading. An applied analysis of both is undertaken in Chapter 4 with respect to usefulness as a basis for high-performance irregular NDP programming.

3.1 Basic Concepts

We begin by making some basic definitions that will prove useful in later theorems. Here we consider mathematical representations for the concepts of vectors and their distribution into disjoint memory spaces of a distributed memory multiprocessor. These notions will form the basis for our discussion of Data-Parallelism and distributed execution.

3.1.1 Vectors

We firstly define the notion of the vector, a single-dimension data aggregate of the type found in most DP programming languages, over which DP operations may be defined.

Definition 3.1 *A* **vector** *is a finite sequence of elements, each of a common type. The* **length** *(cardinality) of a vector V is denoted V_{len}. The i'th element of a vector (called the i'th* **index***) is denoted $V[i]$. The first index of any vector V is $V[0]$ and the last index is $V[V_{len}-1]$. The set of all indices of V is denoted V_{ind}.*

3.1.2 The Distributed Memory Machine

Since the basis of DP execution is the realization of the parallelism present in collection-oriented operation, it is useful to define a framework for this type of execution. We begin by designing a formalism designed to capture the semantics of an underlying machine. We choose to limit our attention to distributed memory machines, since these are the architectures most commonly considered for DP execution.

Definition 3.2 *A* **distributed memory machine** $\mathcal{M}$ *is a sequence* $(N_i)_{i=0}^{size-1}$ *of 3-tuples of the form* (P_i, M_i, S_i^t)*. Each element of this collection is called a* **processing node** *and its position within the sequence is termed the processing node's* **identity***. The* P_i *element of each tuple is termed the node's* **processing element***, the* M_i *element its* **memory space** *and the* S_i^t *component its* **current state***. The number of processing nodes within a distributed memory machine is termed the* **size** *of the machine.*

Our definition of a distributed memory machine describes any computational device composed from a number of processors, each closely associated with a private/local bank of memory. In particular, it describes the real-world class of distributed memory (scalable) supercomputers including the SIMD and SPMD machines described in Sections 1.1.1 and 1.1.2 respectively. Note that the existence of a communications network is implicit in the definition; we assume that there exist means by which the processing element P_i on node i may influence the state object S_j^t of a different (i.e., remote) node j. For the discussion which follows, the mechanism by which such transformation occurs (e.g., by the transmission of messages through a particular topology of network) is not important.

The central element of the distributed memory machine we have defined is the processing element, a representation of the computational hardware present on the node and which can execute instructions and define a thread of control. Within our formal framework we choose to model this process of nodal execution as a discrete series of transitions altering the state of the node.

Definition 3.3 *Each processing element* P_i *can be transformed at time* t *by the application of an* **instruction** I*, generating a subsequent state at time* $t+1$*. Equationally* $S_i^{t+1} = P_i(S_i^t, I)$*.*

3.1.3 Partitioning Vectors Across the Distributed Machine

As mentioned earlier, the goal of this mathematical analysis is to model parallel operations across data. Typically such operations are granted parallelism by the distribution of the input data across the various memory spaces of the machine. It is useful to describe such opportunities for parallelism — we do this by introducing mechanisms for modelling the decomposition of a vector across memory spaces of our model machine.

Definition 3.4 *The memory space M_i of a node is a finite set of vector indices. The indices within M_i are said to be* **owned** *by the processing node i.*

Definition 3.5 *A* **partitioned vector** *is a vector whose elements are all contained within memory spaces of the distributed memory machine. Each such vector V has an associated* **partitioning function** *P_V which maps the j^{th} index of V to the node identifier i, where the memory space of node i (i.e., M_i) contains that index of V.*

Example 3.1: Figure 13 shows the relationship between vector partitioning and memory spaces. In the upper panel we see three vectors X, Y and Z of varying length. We attribute each with a partitioning function (as shown in the middle panel) resulting in the assignment of each index of the three vectors being assigned to one of the memory spaces of the distributed memory machine. The bottom panel shows the resultant mapping of indices to memory spaces.

As we have defined the concept of the partitioning function associated with each partitioned vector, it is a mechanism for mapping each vector index to exactly one node's memory. If we look at this from the opposite perspective it means that the memory of a given node is composed entirely of indices (from partitioned vectors) which some partitioning function has mapped to that memory. We can formalize this concept as follows.

Theorem 3.1 *If $\mathcal{V}$ is the set of all partitioned vectors currently in existence, then*

$$M_i = \bigcup_{v \in \mathcal{V}} \{z \in v : P_v(z) = i\}$$

Proof: This follows from the definitions of partitioned vector and distributed memory machine. If we consider a single partitioned vector $v \in \mathcal{V}$ and its associated

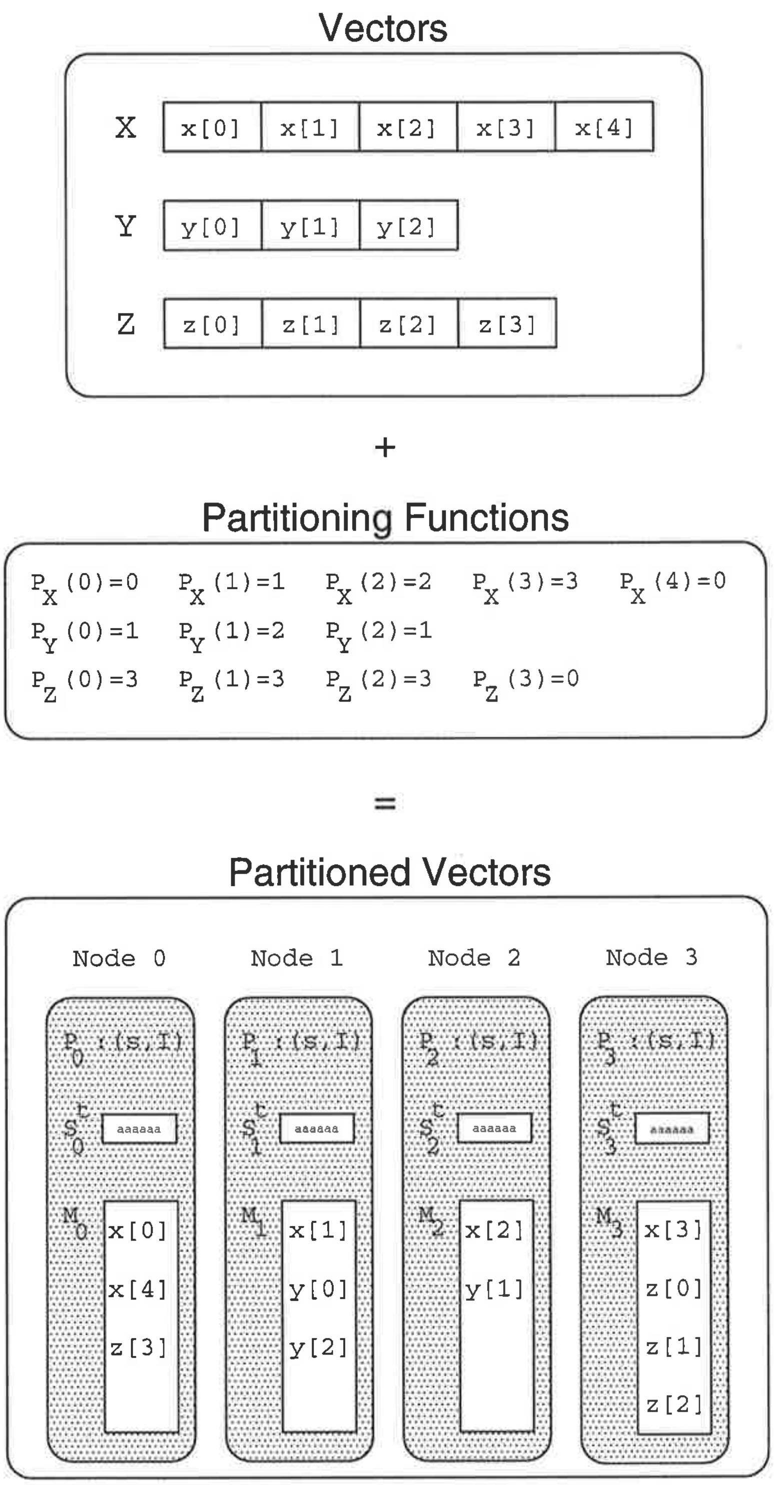


Figure 13. Partitioning Vectors into Memory Spaces

partitioning function P_v then we can define the subset $(M_i)_v$ of v's indices which P_v indicates should be stored in a designated memory M_i. $(M_i)_v = \{z \in v : P_v(z) = i\}$. This set is clearly a subset of M_i. If we consider such subsets for every $v \in \mathcal{V}$, their union is the set of all existing aggregate indices assigned by partitioning to node i. By definition, this set is also M_i. $\Box$

3.2 Modelling Data-Parallelism

Armed with definitions of partitioned aggregates and distributed memory machines, we are now in a position to discuss the semantics of distributed execution for Data-Parallel operations over partitioned aggregates. We begin by defining a mathematical representation for a Data-Parallel operator as a specialized form of a general computation. From there we consider the possible distributed executions of such forms.

3.2.1 The DP Operator as a Specialized Co-operative Computation

In its most general form, the concept of a computation can be considered to be little more than a process by which an underlying machine is transformed from one state to another by the application of a collection of instructions. Clearly the execution embodied in a DP operation falls within this definition, hence it is worthwhile formalizing the concept as part of our developing theory of DP execution.

Definition 3.6 *A* **computation** *is a function $\mathcal{C}$ which maps a set of processing nodes $\mathcal{P}$ at time t and a set of vector* **inputs** *$\{V_i\}$ to the same set of processing nodes at a later time $t + z$ with a different state S_i^{t+z}:*

$$\mathcal{C} : \{(P_i, M_i, S_i^t), i \in \mathcal{P}\} \times \{V_i\} \mapsto \{(P_i, M_i, S_i^{t+z}), i \in \mathcal{P}\} \textit{ for some } z \in \mathbf{N}$$

We call the set $\mathcal{P}$ the **participant set** *of the computation. A computation is said to be* **co-operative** *if $\mathcal{P}$ contains more than one member. A computation whose participant set has but one member is termed* **local***.*

It is worth noting that this is a very general definition of the concept of computation, one which embodies two common forms observed in programming models: the serial computation and the co-operative parallel computation. The former includes every type of computation supported by traditional (non-parallel) programming languages, while the latter describes paradigms where a single conceptual algorithm is decomposed into a set of communicating sequential computations.

Example 3.2: Let M be a distributed memory machine of four nodes, N_0, N_1, N_2 and N_3. If we consider a computation $\mathcal{C}_1$ which maps the single node N_0 (at time t) and the vector V to a future instance of N_0 in which S_0^{t+z} contains an element called **sum** which is the sum of all indices of V stored in M_0. The participant set for this computation is $\{N_0\}$, which has only a single element. Therefore the computation $\mathcal{C}_1$ is a local computation.

Consider instead a computation $\mathcal{C}_2$ which maps the four nodes of $\mathcal{M}$ (at time t) and a set of four vectors $\{V_0, V_1, V_2, V_3\}$ to a future instance of the four nodes in which each S_i^{t+z} has an element called **sum** which holds the sum of all indices from the vector V_i. That is, node N_0's state contains the sum of all indices of V_0, node N_1's contains the sum of all indices of V_1, and so on. The participant set of this computation is $\{N_0, N_1, N_2, N_3\}$, hence $\mathcal{C}_2$ is co-operative.

While it is clearly true that a DP operation is a type of computation, there are a number of properties that distinguish it from other forms that are equally termed computation. Thus it is useful to define the DP operation as a specialization of the general concept. Looking back to the descriptions of DP languages and operators presented in Section 1.2, a number of distinguishing features become apparent. Firstly, the DP operation is intended as a parallel (or co-operative) computation — it exists within a language as an explicit opportunity for expressing parallelism. Secondly, a DP operation is explicitly tied to a particular argument aggregate — it is defined as a parallel operation in which a given serial component computation is performed for each of the indices of the input vector. Finally, as described in Chapter 1, the DP operation is inherently synchronizing — none of the serial sub-components can finish until all have completed their serial execution.

Formalizing these concepts, we obtain the following definition for a DP operation:

Definition 3.7 *A* **Data-Parallel (DP) Operation** *is a co-operative computation Ψ which has three properties:*

1. *$\exists$ a vector V which is designated as the* **primary** *input of Ψ. We use the notation $\Psi(V)$ to describe the primary input of a DP operator Ψ.*

2. *$\Psi(V)$ may be decomposed into a set of* **sub-operations** *or* **responsibilities** *$\Psi_i(V[i])$, one such sub-operation per index of V. Each such responsibility is a local computation. Equationally:*

$$\Psi^{\text{co-op}}(V) = \{\Psi_i^{\text{loc}}(V[i]), \quad i \in [0, \ldots, V_{len} - 1]\}$$

3. *a distributed execution (see Definition 3.14 below) of Ψ must be synchronizing (in the sense of Definition 3.15).*

Example 3.3: Consider again the distributed machine $\mathcal{M}$ of four nodes. Let V be a vector of length four, partitioned so that each node owns one index of V. Let $\mathcal{C}_3$ be a computation which maps all four nodes (at time t) plus the single vector V to a future version of the four nodes in which each state object S_i^{t+z} contains an element called *sum* with identical value, namely the sum of all elements of V. This computation is co-operative (there are four nodes in the participant set) and closely associated with a single vector (i.e., V). Furthermore, we can express $\mathcal{C}_3$ in terms of a set of four local computations (responsibilities) as follows:

$\mathcal{C}_3^j$ is a local computation which maps a node N_j (at time t) plus the vector index $V[j]$ stored within that node's memory space (M_j) to a future instance of N_j which has a state element *sum* with a value equal to the sum of all indices of V. There are four such local computations $\mathcal{C}_3^0, \mathcal{C}_3^1, \mathcal{C}_3^2$, and $\mathcal{C}_3^3$, one per index of V. We can define informal semantics for these serial computations:

- $\mathcal{C}_3^0$ begins by creating three elements v_1, v_2 and v_3 in the state of the node N_0 which executes it. The serial computation then waits until all of these have been filled with values, then computing *sum* $= v_1 + v_2 + v_3 + V[0]$ (where $V[0]$ denotes the value stored in the zeroth index of V). Once this is computed, the value *sum* is copied into the state objects of each node $N_i, \quad i = 0, \ldots, 3$.

- $\mathcal{C}_3^j, \quad j = 1, 2, 3$ copies the value of the jth index of V into the v_j element of N_0's state.

From the definition above, $\mathcal{C}_3$ would be a DP operator assuming it were executed on a synchronizing execution (e.g., if we assume that the operation were executed in such a way that all nodes synchronize on completion). The example computations $\mathcal{C}_1$ and $\mathcal{C}_2$ from the Example 3.2 are not DP operators: the first is not a co-operative computation, the second does not have a single primary input V.

Each responsibility within a DP operation represents a serial computation which takes place during the execution of an operation. From the definition of the responsibility it is clear that each such serial computation has a dependence upon one index of the input vector, namely that index for which this responsibility was generated (during the decomposition). We call this kind of data dependence a *close association*, with the assumption that the designated index is pivotal to the computation embodied by the responsibility and will likely be used many times within that computation. Because there are no restrictions on the kinds of operations that may appear within a responsibility's serial computation, it may arise that there are any number of other data dependences which must be fulfilled before that computation can complete. We term these other dependences *loose associations* and assume that, while they are necessary to the computation, the data they define is likely to be used only a few times. Formalizing the concepts into the mathematical framework, we get:

Definition 3.8 *The responsibility $\Psi_j(V[j])$ which corresponds to index $V[j]$ (of Ψ's primary input) is said to be* **closely associated** *with that index. Any other associations of Ψ_j are termed* **loose associations***; these may be with other indices of V or indices of other vectors.*

Example 3.4: In the previous example's definition of the responsibilities of $\mathcal{C}_3$, each of the $\mathcal{C}_3^j$'s is closely associated with one vector index $V[j]$. The responsibility $\mathcal{C}_3^0$ is furthermore loosely associated with the indices $V[1], V[2]$ and $V[3]$ since its execution cannot complete until it has received these values.

3.2.2 Executing a DP Operation on a Distributed Memory Machine

Now that we have defined Data-Parallel operations and their decomposition into a set of local (serial) sub-computations, we may consider the execution of these sub-computations (or responsibilities) on a distributed memory machine. We do this by defining the notion of a *schedule* of responsibilities maintained by each node, a list of responsibilities which the node's processing element will at some time be applied to. Once we have defined this notion we may consider the process of mapping responsibilities to positions within such lists.

Definition 3.9 *Each node N_i of the Distributed Memory Machine maintains (as part of state S_i^t) a list $\mathcal{L}_i^t$ of responsibilities that will, at some time t, be passed to the node's processing element and thus transform the local state. $\mathcal{L}_i^t$ is called node i's* **local schedule**. *The order of responsibilities within $\mathcal{L}_i^t$ describes the temporal ordering of the associated transformations. A node's local schedule at time 0, $\mathcal{L}_i^0$ is called its* **initial schedule**.

Example 3.5: Considering again the example of $\mathcal{C}_3$, the DP operator defined previously, applied across a vector V of four elements, one possible set of initial schedules is:

$$\mathcal{L}_0^0 = [\mathcal{C}_3^0], \quad \mathcal{L}_1^0 = [\mathcal{C}_3^1], \quad \mathcal{L}_2^0 = [\mathcal{C}_3^2], \quad \mathcal{L}_3^0 = [\mathcal{C}_3^3]$$

Another possible set is:

$$\mathcal{L}_0^0 = [\mathcal{C}_3^0, \mathcal{C}_3^1], \quad \mathcal{L}_1^0 = [\mathcal{C}_3^2, \mathcal{C}_3^3], \quad \mathcal{L}_2^0 = [], \quad \mathcal{L}_3^0 = []$$

The former set of initial schedules defines a distributed execution in which all four nodes of the machine each execute the serial code from one responsibility. The second set of schedules defines a different situation: nodes 2 and 3 perform no work in the execution of the operator, whereas nodes 0 and 1 are each called upon to perform two responsibilities. These nodes first execute (at time $t = 0$) the code for the responsibility at the head of their schedule, and then at some later time execute the code for their second responsibility.

The local schedule structure defines a mechanism for sequencing the serial code (from responsibilities) which a node is called upon to execute during a DP operation. The policy which is employed in the definition of each node's initial schedule is an important factor in how the execution will proceed. We formalize the concept of such a policy as follows:

Definition 3.10 *A* **distribution** *of a DP operator $\Psi(V)$ is a function $\mathcal{D}$ that maps each responsibility $\Psi_i(V[i])$ to a unique position within exactly one initial schedule $\mathcal{L}_j^0$.*

This definition merely states that a policy of responsibility distribution may be defined in terms of an arbitrary function. This function is analogous to the concept of the partitioning function for the dispersal of vector indices throughout the memory spaces of the machine. Like the partitioning function, the distribution function can specify arbitrarily complicated patterns of decomposition. In some instances it is useful to restrict the possible range of distributions to enforce a particular property of an execution. One common limitation often placed upon a distribution is that it should map all responsibilities with close associations to index I onto the node whose memory contains I. This is called an *owner computes* distribution. There is an implicit heuristic assumption that the cheap satisfaction of the close association for those responsibilities will lead to an overall cheaper computation.

In our analysis we will usually consider the general case of arbitrary distributions, although in several instances we will consider the particular case of owner computes. In the practical application of this theory made in Chapter 4, we return to a discussion of the benefits of such distributions in real-world implementations.

Definition 3.11 *A distribution $\mathcal{D}$ of a DP operator $\Psi(V)$ is said to be* **owner-computes** *iff*

$$\mathcal{D}(\Psi_i(V[i])) \in \mathcal{L}^0_{P_v(i)} \qquad \forall i \in [0, \ldots, V_{len} - 1]$$

That is, every responsibility closely associated with a vector index in node N_i's memory must be mapped into node N_i's initial schedule by $\mathcal{D}$.

Example 3.6: Consider the vector V of four elements, partitioned by the function $P_v : x \mapsto M_x$ across a distributed memory machine of four nodes. Index $V[0]$ is contained within M_0, index $V[1]$ within M_1 and so on. Consider again the DP operator $\mathcal{C}_3$ from Example 3.3 and the two possible distributions of responsibilities to initial schedules presented in Example 3.5. The first of these distributions is owner-computes, since each node's initial schedule $\mathcal{L}^0_j$ contains only responsibilities whose close association is with indices stored within M_j. That is, node N_0's schedule contains $\mathcal{C}^0_3$ which is closely associated with $V[0]$, node N_1's schedule contains $\mathcal{C}^1_3$ which is closely associated with $V[1]$ and so on.

Conversely, the second distribution shown in the example above is not owner-computes since both nodes N_0 and N_1 have, within their initial schedule, responsibilities with close associations to indices of V which are not stored within the local memory. Specifically, $\mathcal{L}^0_0$ contains $\mathcal{C}^1_3$ which has its close association with

$V[1]$ (stored in node N_1's memory space). Similarly, $\mathcal{L}_1^0$ contains $\mathcal{C}_3^2$ (closely associated with an index in M_2) and $\mathcal{C}_3^3$ (closely associated with an index in M_3).

Previously we have informally referred to the mechanism by which the serial computation from a responsibility is executed upon a node of the distributed machine. We turn now to a formalization of this concept in preparation for our description of the execution of an entire DP operation. We also consider a definition of what it means for an entire DP operation to have completed its execution. Again this is a useful concept leading up to the definition of a full DP execution.

Definition 3.12 *A responsibility Ψ_i is said to have been* **applied** *when its associated transformation has been induced upon a processing node. The responsibility Ψ_i is said to be* **complete** *at the time, T, when the* entire *co-operative computation Ψ (i.e.,* all *responsibilities of the DP operator) has been applied. We define a predicate* **complete**(Ψ_i, t) *to be true iff at time t, Ψ_i is complete.*

Building a Distributed Execution Formalism

To continue with our mathematical framework for describing DP, we must turn our attention to describing the dynamic features of the paradigm. That is, we must detail the mechanism by which a distributed DP operation (i.e., a set of initial schedules) is executed by nodes of a distributed memory machine. To do this we must describe an algorithm for distributed execution.

At its simplest, we can consider a distributed execution of a set of initial schedules to be a simple three-step iterative process stepped-through by each node of the machine:

1. initialize the node
2. if the local schedule is non-empty, apply the head element and remove it.
3. if the local schedule is empty, terminate, else return to Step 2.

This model captures the semantics of a real-world multi-processor machine: on each iteration of the algorithm all nodes fetch a schedule element (instruction) from the local schedule (local program) and execute it. The node terminates its execution when the local schedule (local program) is exhausted.

While this definition of a distributed execution is sufficient to model a machine at a coarse level, it ignores one important aspect of an execution central to DP styles of execution: inter-node synchronization. As noted in the definition of the DP operator (Definition 3.7), an important part of modelling the semantics of such an operation is an adherence to the fundamental DP property that all collaborators (i.e., responsibilities) synchronize prior to any of them completing their execution. The simple model we have described above allows no form of co-ordination between nodes stepping (independently) through the algorithm and hence the model is unable to express DP execution.

We thus consider generalizing the simple model to allow it to express the style of nodal interaction required by the DP paradigm. The first concept we choose to add to the simple execution model is that of *nodal control blockage*, that is a nodal state in which further execution is disallowed until such time as some event has taken place. In DP styles of execution such blockages commonly arise as the result of inter-node synchronization, thus it is important that such a state is expressible in our formalism. We choose to model nodal control blockage by augmenting each node with a flag which denotes its readiness. The intended role of the ready flag in the distributed execution is simple: whenever a node is marked as *ready* (i.e., ready flag is set), it is in a position whereby it may execute an element from its schedule. When the node is *unready* (i.e., ready flag is clear), no such execution is possible; that is, the node is blocked.

A second factor that must be considered in our model is the specification of a predicate by which a blocked node can become unblocked, that is a mechanism for defining the dependency which caused a given blockage. In the DP model, synchronizations occur between nodes which are executing responsibilities from the same DP operation. A useful basis for modelling such a dependency is to record which responsibilities which have been applied (executed) on a given node but which have yet to be synchronized with their peers. These waiting responsibilities are the sources of blockages in DP executions. We add this form of record to our model in the form of the *applied set* into which responsibilities are added following their application, and from which responsibilities are removed when a synchronization is completed. This formalism provides a convenient mechanism for the specification of DP-style synchronization: we can state, for example, that node N_a which has responsibility Ψ_a (from DP operator Ψ) in its applied set will be blocked (ready flag set false) until such time as all other responsibilities of Ψ appear in some node's applied set.

In order to keep a clean semantics (with single assignment) in our execution modelling we also define a second set for each node, the *newly applied set*, a temporary structure in which associations can be recorded immediately after being applied on a node (before being moved into the applied set defined for the next cycle through the algorithm).

Collectively we formalize the notion of a distributed memory machine augmented by ready flag, applied set and newly applied set as follows:

Definition 3.13 *A* **distributed execution environment** Θ *consists of a distributed memory machine $\mathcal{M}$, each of whose nodes N_i have been augmented with three additional properties:*

- *a* **ready** *flag,*
- *an applied set A_i^t, and*
- *a newly applied set $\bar{A}_i^t$.*

Each of these properties is defined at a series of discrete time steps $t = 0,\ t = 1, \ldots$

The additions we have made to the nodes of the distributed memory machine permit us to specify a new iterative model for distributed execution which has the potential to define forms of synchronization useful to DP operation. We do this by specifying the distributed execution as an informal four step iterative process carried out by each node of the distributed machine:

1. initialize the node;
2. if the node is ready and has a non-empty schedule,
   (a) apply the head element of the schedule,
   (b) move the head schedule element to the newly applied set, and
   (c) mark the node unready;
3. if the node has at least one responsibility outstanding
   (a) add the newly applied set to the applied set,

   (b) remove responsibilities from the applied set for which synchronizations have now completed

   (c) determine whether the node should be returned to the ready state;

4. if the local schedule is empty, terminate, else return to Step 2.

This definition includes two major decision points which dictate the flow of the algorithm, namely:

- the point at which we must determine which responsibilities to discard from the node's applied set, and
- the point at which we must choose whether the node is to be marked ready (and thus continue executing responsibilities).

In any given distributed execution, policies must be defined which dictate the decision making process which takes place at both these points within the algorithm. For convenience we describe the former decisions as being dictated by a *discard policy* and the latter by a *continuation policy.* Clearly there are several possible candidates for each policy; the adoption of each combination delivers an execution with different characteristics. We can describe a simple uncoordinated execution (i.e., the distributed execution we began with) by defining policies of discarding the entire applied set, and of always indicating continuation by marking the node as ready in Step 3 of the algorithm. As we demonstrate in following sections, it is also possible to describe more complex synchronizing operations by choosing different policies.

We now make a formal definition of our concept of the distributed execution in line with the informal outline we have already made.

Definition 3.14 *A* **distributed execution** $\Delta(\beta, \Gamma, \Theta)$ *is an iterative algorithm which transforms a distributed execution environment* Θ *at time* $t = 0$ *to the same environment at a later time* $t = T$. *Two characteristic functions* β *and* Γ *define aspects of the execution. The function* β *is called the* **discard function** *while* Γ *is termed the* **continuation predicate**.

The algorithm is defined by five steps, with each node of the machine independently beginning the execution at the start of Step 1:

1. *each node N_i of Θ is:*
   - (a) *marked as* **ready**,
   - (b) *has a local time counter t_i set to* 0,
   - (c) *and is given an* **applied set** A_i^0 *and a* **newly applied set** $\bar{A}_i^0$ *both of whose value is ϕ.*

2. *each node N_i of Θ* which is ready *and which has a non-empty local schedule $\mathcal{L}_i^{t_i}$ performs the following sub-steps:*
   - (a) *examines the responsibility Ψ_j at the head of $\mathcal{L}_i^{t_i}$ and:*
   - (b) *uses Ψ_j to compute a new state: $S_i^{t_i+1} = P_i(S_i^{t_i}, \Psi_j)$*
   - (c) *adds Ψ_j to N_i's newly applied set $\bar{A}$: $\bar{A}_i^{t_i} = A_i^{t_i} \cup \{\Psi_j\}$*
   - (d) *removes Ψ_j from $\mathcal{L}_i$ to form a new schedule: $\mathcal{L}_i^{t_i+1} =$ **tail** $(\mathcal{L}_i^{t_i})$*
   - (e) *marks itself* unready.

   All nodes which were ineligible to perform these steps simply copy:
   $\bar{A}_i^{t_i} = A_i^{t_i}$, $S_i^{t_i+1} = S_i^{t_i}$ *and* $\mathcal{L}_i^{t_i+1} = \mathcal{L}_i^{t_i}$.

3. *all nodes N_i for which $\bar{A}_i^{t_i} \neq \phi$:*
   - (a) *discard responsibilities from $\bar{A}_i^{t_i}$: $A_i^{t_i+1} = \bar{A}_i^{t_i} \backslash \beta(\bar{A}_i^{t_i})$*
   - (b) *calculate $\Gamma(A_i^{t_i+1})$ to determine whether the node N_i is permitted to* continue *its execution*
   - (c) *nodes N_i for which $\Gamma(A_i^{t_i+1})$ evaluates true are marked* ready

   All nodes which were ineligible to perform these steps simply copy:
   $A_i^{t_i+1} = \bar{A}_i^{t_i}$.

4. *all nodes N_i increment their local time counter t_i by* 1

5. *if $\exists$ a processing node for which $\mathcal{L}_i^{t_i}$ is not empty, return to Step 2, otherwise terminate.*

We now consider the specialization of this general execution which corresponds to the fundamental DP style of synchronizing execution. In earlier discussion we stated this property informally by noting that a node N_a which has responsibility Ψ_a (from

DP operator Ψ) in its applied set should be blocked (made *unready*) until such time as all other responsibilities of Ψ appear in some node's applied set. In the terms of our mathematical framework, such a statement is a definition of discard policy, thus we formalize it by defining a particular discard function $\beta_{\mathbf{sync}}$ which embodies this principle.

Definition 3.15 *A distributed execution in a distributed execution environment Θ is called* **synchronizing** *if a responsibility Ψ_j is only discarded when complete. That is,*

$$\beta_{\mathbf{sync}}(A_i^{t_i}, t_i) = \{\Psi_i \in A_i^{t_i} : \mathbf{complete}\ (\Psi_j, t_i)\}$$

In developing our model of a distributed execution, we adopted the applied set as a repository of responsibilities waiting at some barrier point for their siblings to be applied. Another way of viewing the elements of this set is as blocked threads of control: each applied set responsibility corresponds to a control flow which cannot continue until some synchronization event has taken place (represented by its removal from the set by the discard policy). This point of view is useful because it allows us to reason about the *threadedness* of an execution, that is the degree to which a node is called upon to manage multiple concurrent control threads at points during the execution. We can define this concept of threadedness very simply in our model as being the maximum size to which an applied set is allowed to grow while the algorithm is being stepped-through. More formally:

Definition 3.16 *We define the* **threadedness** $\mathcal{T}$ *of a distributed execution Δ to be the maximum size which any node's applied set A_i^t will assume during the execution. That is, if an execution takes place between time $t = 0$ and $t = T$ across a distributed memory machine of size s, then*

$$\mathcal{T} = \max_{i \in [0,\ldots,s-1]} \max_{t \in [0,\ldots,T]} |A_i^t|$$

We call a distributed execution with $\mathcal{T} = 1$ **single threaded***; all others are* **multi-threaded**.

The principal utility in computing the threadedness of an execution model relates to estimations of how easily that model may be mapped onto a real piece of hardware (see Chapter 4 for such a discussion). As we noted in our review of common parallel hardware for DP execution (Sections 1.1.1 and 1.1.2), most such machines have node which support only a single thread of control in hardware. Thus, we would intuitively expect that an execution model which is multi-threaded (i.e, which allows the applied set to grow to two or more) would be more expensive to implement on such a machine than a single-threaded model. Alternatively such multi-threaded models may be quite easily realized on a multi-threaded architecture such as those which have recently been proposed by the parallel hardware [8, 88, 117, 87] and software [30, 23, 114, 85, 99] communities. A brief appraisal of several such models appears in Section 6.3.

3.2.3 A Synchronizing Distributed Execution for DP Operators

We have now modelled enough concepts to allow us to consider defining a paradigm of distributed execution for Data-Parallel operators. The first paradigm we consider modelling is the traditional locked-step SPMD style execution commonly used in implementations of DP (see Section 1.1.2). This represents a well-understood approach to the parallel execution of DP operations and as such is a good starting point for our exploration of the execution requirements for NDP operations. We begin by describing the traditional execution in terms of our mathematical framework and go on to prove properties for the execution. We are particularly concerned with exploring the conditions under which the execution fails to terminate, developing a refinement of our first-cut model (using a concept of aggregating responsibilities) which is provably deadlock-free for any distribution of data and computation within a system. In later sections this model is evaluated to determine how it copes with the execution of multiple simultaneous DP operations (i.e., the NDP case).

From our definition of what we mean by the term DP operator, we know that such an execution must be synchronizing: this defines β, one of the two policy functions to the distributed execution algorithm. Thus all that is required is the definition of a continuation predicate Γ and we will have constructed a distributed execution over DP operators. We know from our discussion of SPMD architectures that their hardware execution model is single-threaded. We can introduce this property into the mathematical model by specifying that a node with a non-empty applied set (i.e., one which is already waiting upon a synchronization) is disallowed from executing further work from its schedule. This is a continuation policy — we can express its semantics within our definition of Γ.

Definition 3.17 A Simple Synchronizing DP Execution*:*
We define a synchronizing execution Δ_{sync} by making a functional definition of the discard function β, and defining a continuation predicate Γ.

$$\beta_{sync}(A_i^{t_i}, t_i) = \{\Psi_i \in A_i^{t_i} : \textbf{complete } (\Psi_j, t_i)\}$$

$$\Gamma_{sync}(A_i^{t_i}) = \begin{cases} true & if\ A_i^{t_i} = \phi \\ false & otherwise \end{cases}$$

Example 3.7: Figure 14 shows how the execution we have defined, Δ_{sync}, can represent the SPMD-style evaluation of a DP operator Ψ across a vector V of length four. The pictured machine has four nodes, each of which is granted an initial schedule containing one responsibility from Ψ (the number of responsibilities being determined by the vector length).

The top section of the figure shows the state of the nodes following their initialization by the first step of the distributed execution algorithm. Each node has had its applied set and newly applied set to empty, its time counter t_i set to 0 and the *ready* flag set.

The middle panel shows the same machine after algorithm Step 2 has taken place on each node. Since all nodes were marked ready, all are given the opportunity to apply the first responsibility from their schedule and move that responsibility into the newly applied set. This leaves each node with an empty schedule and a cleared *ready* flag.

The final section of the figure shows each node after the third algorithm step has been performed (including the evaluation of the discard and continuation functions — thus the annotation of the arrows by β and Γ). The newly applied set (a singleton) is added to the applied set of each node; elements are then removed based upon the application of the discard function β. From Definition 3.17 we know that responsibilities are discarded only when the entire DP operation is complete. In this instance, all four responsibilities from Ψ are in applied sets, thus the operation is complete. This means that all responsibilities are removed from applied sets (representing the completion of a synchronization and the unblocking of each of the nodes of the machine). Left with an empty applied set, the continuation policy defined by Γ_{sync} marks each node ready.

Following this step the time counter would be incremented and the local schedules inspected. Since all are empty, the execution would terminate (if one or more responsibilities had remained, a branch back to algorithm Step 2 would have occurred instead).

At the completion of the algorithm the four node distributed memory machine has fully executed the DP operation Ψ by applying each of the responsibilities (i.e., the execution of the serial code for each vector index) and then synchronizing all participants at a barrier.

One of our goals in defining our synchronizing distributed execution was that it should model a single-threaded parallel scheme. As mentioned previously this is an important property to be able to assert since it indicates that the model has control similarities with the hardware of a broad class of parallel machines including SIMD and SPMD architectures. This similarity is a good indicator that an efficient implementation of the model is possible for this class of machines.

Intuitively the current definition of the execution exhibits the single-threadedness on account of the fact that it disallows a nodes with non-empty applied sets from becoming ready (and thus executing further responsibilities). We now formalize this intuition and prove it for our mathematical definition.

Theorem 3.2 *Δ_{sync} is single-threaded.*

Proof: We prove this theorem through an induction. We begin by denoting the n'th cycle through the iterative loop of Δ_{sync} as C_n. To prove the single-threadedness of Δ_{sync} we must prove that for all $C_n, |A_i^n| \leq 1$ for every node of the machine.

We begin by considering C_0, that is the situation that prevails prior to any cycles of the loop, when only algorithm Step 1 has been carried out. In this case, we know that each of the applied sets A_i^0 is empty; and thus has a cardinality of 0. That is, $|A_i^0| \leq 1$ for every node.

Next we prove that if we assume the proposition for cycle C_n then we can prove it also holds for C_{n+1}. We consider the case for an arbitrary node N_i. We look at the applied set A_i^{n+1} for that node at the algorithm step where we return to the beginning of the loop. Either one of two cases holds true for this node's applied set: either $A_i^{n+1} = \phi$ or $|A_i^{n+1}| = 1$. We know that these are the only two cases, since the inductive assumption tells us that the cardinality of the set is ≤ 1.

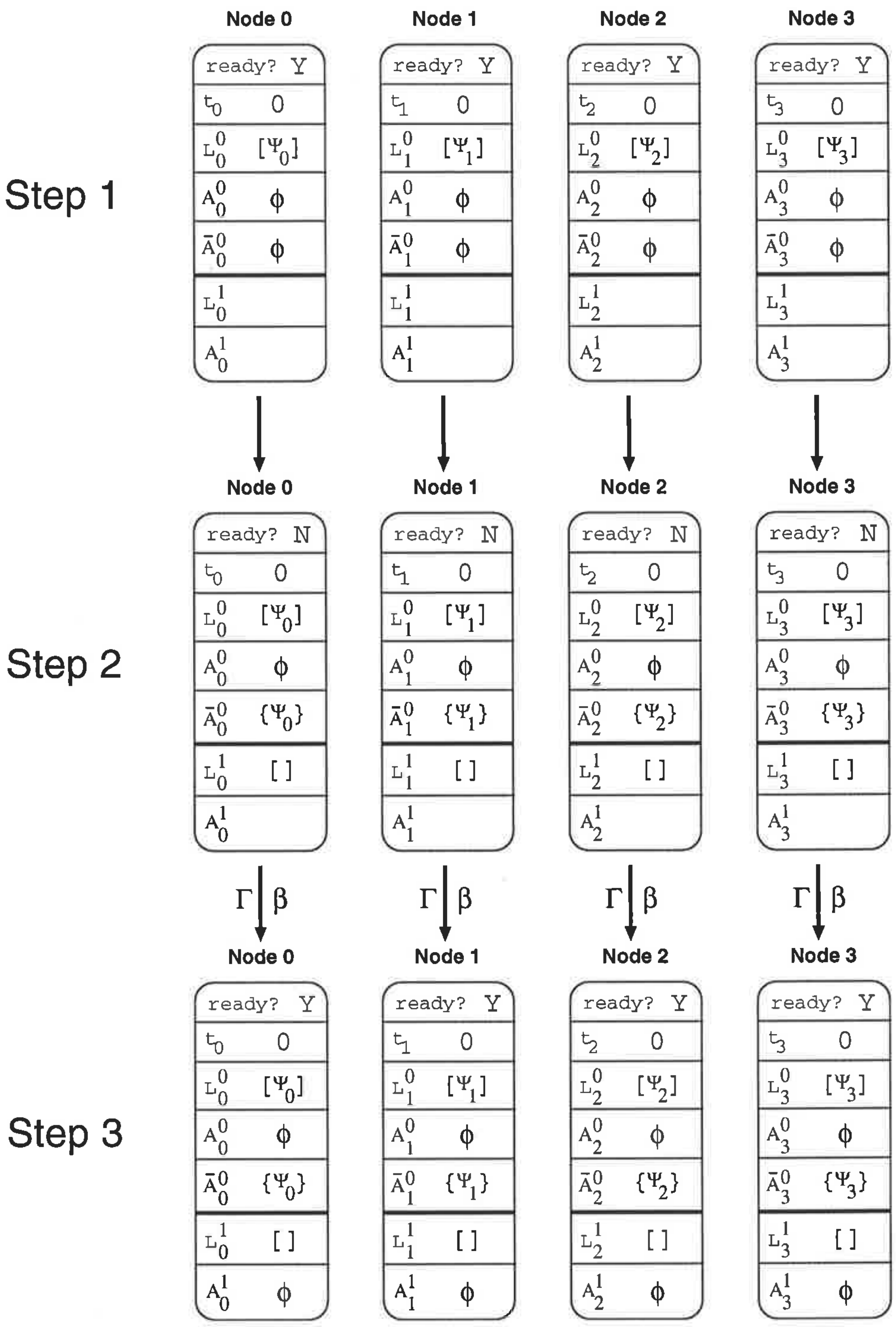


Figure 14. An Example Time-Step ($t_i = 0$) of the Simple Synchronizing DP Execution Δ_{sync}

Case $A_i^{n+1} = \phi$:
In this case, we consider what must have occurred during Step 3 just prior to this loop's completion. During that step, the continuation predicate $\Gamma_{\text{sync}}(A_i^{n+1})$ would have been evaluated — clearly from Definition 3.17, this evaluation would return *true*. Thus node N_i completes iteration C_n marked as ready.

Now let us consider the C_{n+1} cycle through the loop. Since N_i is ready, it may potentially partake in the sub-steps of algorithm Step 2 if it still has elements in its local schedule. If we assume that no such element exists, then the path of the algorithm simply copies $\bar{A}_i^{n+1} = \phi$ which causes Step 3's sub-steps to be skipped ($A_i^{n+2} = \bar{A}_i^{n+1}$) and C_{n+1} finished with $|A_i^{n+2}| = 0 \leq 1$.

Alternatively, if we assume that $\mathcal{L}_i^{n+1}$ has at least one element, then the sub-steps of Step 2 are followed. The effect of this is to define $\bar{A}_i^{n+1} = \phi \cup \{\Psi_j\}$. That is, at the end of these sub-steps there is exactly one member of the applied set. Since this set is now non-empty, the sub-steps of algorithm Step 3 are performed. The effect of this on A_i^{n+2} is to remove 0 or more elements from $\bar{A}_i^{n+1}$ (depending on the value of $\beta(\bar{A}_i^{n+1}, t_i)$).

Hence, in either case, at the end of iteration C_{n+1} there is at most 1 member of the set A_i^{n+2}. That is, the proposition to be proven holds true for C_{n+1}.

Case $|A_i^{n+1}| = 1$:
By similar analysis to that presented for the previous case, we can deduce that the application of the continuation predicate Γ_{sync} that took place during C_n would necessarily have evaluated to *false* (since A_i^{n+1} is non-empty). Therefore, at the end of the iteration C_n, node N_i would be marked unready. Therefore, none of the sub-steps of Step 2 would be followed during C_{n+1}. Instead, $\bar{A}_i^{n+1}$ would simply be defined to be equal to A_i^{n+1}. This would mean that $\bar{A}_i^{n+1}$ would be a set of one element and hence the sub-steps of Step 3 would apply. As in the previous discussion, the effect of these sub-steps on A_i^{n+2} are to remove 0 or more members from $\bar{A}_i^{n+1}$ based on the discard function β. Regardless of the exact number of members discarded, by the time the loop is complete, $|A_i^{n+2}| \leq 1$. That is, the proposition holds true for C_{n+1}.

We have shown that for both possible cases, assuming the proposition for C_n implies it to be true also for C_{n+1}. This, coupled with the above proof for the base case and the Principle of Induction, allows us to deduce $|A_i^n| \leq 1$ for all $n \geq 0$. Since node N_i is an arbitrary node of the distributed machine, it must hold that this property is true for all nodes. Therefore, Δ_{sync} is single-threaded. □

In defining our synchronizing execution model we have effectively defined a process whereby a computation can be decomposed into a number of serial chunks and distributed across a machine (into the initial schedules of the various nodes). These individual computations have an important interdependency by virtue of the fact that they collectively define a DP operation; all must synchronize at a barrier before completion. Since such synchronization implies the complete blockage of control on a node it becomes important to ensure that we never arise at a situation where all nodes block — since such an event defines a deadlocked system (which will never complete the assigned computation). We would like to be able to discover what kinds of conditions such deadlock can occur, so that we can consider means of ensuring that such situations do not arise.

In the application of the simple synchronizing execution to the evaluation of a single DP operation, the only way that a deadlock may arise is by a single node being granted more than one responsibility for the operation. In that instance, once the first responsibility is executed by the node it is moved into the applied set, blocking control on the node. The blockage can only be resolved when all responsibilities have been applied; but the second responsibility granted to the designated node cannot possibly be applied until after the blockage has been cleared. Thus we arrive at a circular dependency which signifies a deadlock. We formalize this argument in terms of our model:

Theorem 3.3 *Distributions $\mathcal{D}$ of a DP operator Ψ which assign more than one responsibility Ψ_i to any given node, will always cause the distributed execution Δ_{sync} of Ψ to deadlock.*

Proof: Let Ψ_a and Ψ_b be two responsibilities of Ψ which are contained within $\mathcal{L}_i^0$, the initial schedule of a particular node N_i. Let p_a be the position of Ψ_a in the list, p_b be the position of Ψ_b. Without loss of generality we assume $p_a < p_b$.

Now consider the situation where, after a number of cycles through the iterative loop of Δ_{sync}, N_i has removed Ψ_a from its schedule $\mathcal{L}_i^n$, placing it instead into its applied set A_i^n. . This action renders N_i unready, since Γ_{sync} evaluates to *false* for $A_i^n \neq \phi$.

Next we consider what needs to take place for the head of $\mathcal{L}_i^{n+1}$ to be passed to the processing element P_i. By observation, it is clear that such an event takes place only when the node is permitted to follow the sub-steps of algorithm Step 2. That is, it occurs when N_i is *ready*. But by the nature of Γ_{sync} that cannot happen until the applied set A_i is empty, which can clearly only be realized by a call to β_{sync} evaluating to a set containing Ψ_a (since we know this responsibility is a member of A_i^n).

Examining the definition of β_{sync} it becomes apparent that Ψ_a will only be a member of the discard set when *every* responsibility of Ψ is contained within an applied set A_j^n. Specifically, it can only occur when Ψ_b is within such a set. By inspecting the algorithm again, it is clear that it is only ever possible for Ψ_b to be a member of A_i^n (that is, the applied set for the node which had Ψ_b in its schedule).

Therefore, for Ψ_a to be removed from A_i^n, that set must also contain Ψ_b. But, by Theorem 3.2, Δ_{sync} is single-threaded and hence its applied set can never contain more than one responsibility. This implies that Ψ_a and Ψ_b can never be members of the same applied set; hence (by the argument above), Ψ_a can never be a member of the discard set generated by β_{sync}. Thus, N_i can never become ready and the responsibility at the head of $\mathcal{L}_i^n$ can never be executed. Thus, the execution of Δ_{sync} will not terminate. □

Implication: This result suggests that the simple synchronizing DP execution Δ_{sync} is only a useful execution when the distribution $\mathcal{D}$ ensures that no node's initial schedule has more than a single responsibility from a given DP operator Ψ. Specifically, it is not a useful model in the case that the length of the close association vector of Ψ is greater than the size of the machine, since in that case no such distribution $\mathcal{D}$ exists. This is clearly a deficiency of our simple synchronizing model, one which now seek to address.

3.2.4 Avoiding Deadlock by Responsibility Aggregation

The limited utility of our first attempt at defining a distributed execution arose from the fact that DP operators involving more responsibilities than the machine size would lead to non-termination of the execution. Under the proposed system of decomposition of DP operators, this would inevitably arise for every DP operator acting over a vector with more indices than there are nodes in the machine.

Our approach to overcoming this limitation is to aggregate responsibilities into larger entities. We then consider a simple modification to the previously defined execution models which substitutes these aggregated responsibilities in place of normal responsibilities as elements in the initial schedule. The fact that there is always fewer of these macro-responsibilities than normal responsibilities (the exact number is defined by the nature of the aggregation), suggests that it should be possible to arrange for the execution of a DP operator over a large vector to run to conclusion.

We start by defining an aggregation of responsibilities, which we term a *macro-responsibility.* The concept is simple: a macro-responsibility is merely a set of responsibilities from the same DP operation. We formalize the notion as follows:

Definition 3.18 *A* **macro-responsibility** $\Psi_n^\square$ *of a DP operator* $\Psi(V)$ *is a set of responsibilities* $\Psi_i(V[i])$*. The set of* $\Psi_n^\square$*'s closely associated indices of* V *is denoted* $V\{i_1, i_2, \ldots, i_r\}$ *and is called the macro-responsibility's* **close association set.**

The computational semantics associated with a macro-responsibility are assumed to be derived from the semantics of the component responsibilities by a process of serialization. Thus, the serial computation of a macro-responsibility can be considered to be given by the serial computations for the individual responsibilities concatenated to one another by a sequencing operation. In terms of our mathematical model, this concept can be represented as follows.

Definition 3.19 *A macro-responsibility* $\Psi_n^\square$ *may be passed to a processing element, at which time it will induce a state change which is a composition of the state changes associated with each of the responsibilities* Ψ_j *it contains. That is if* $\Psi_n^\square = \{\Psi_{i_1}, \ldots, \Psi_{i_r}\}$*, then*

$$P_i(S_i^t, \Psi_n^\square) \equiv P_i(P_i(\ldots P_i(P_i(S_i^t, \Psi_{i_1}), \Psi_{i_2}) \ldots), \Psi_{i_r-1}), \Psi_{i_r})$$

Where the right-hand side of the equivalence contains exactly r *applications of the processing element* P_i*.*

An important point to note is that there is no notion of synchronization between the responsibilities within a given aggregation. That is, if a DP operator Ψ is decomposed into a series of macro-responsibilities, we define a protocol of barrier synchronization between those macro-responsibilities but each macro-responsibility can execute as a single serial code (despite the fact that it is, in essence, a collection of responsibilities). This defines a slightly looser notion of DP synchronization than considered previously, although one that still maintains the required properties of race avoidance[1].

Using the Concept of Responsibility Aggregation in the Previously Developed Theory

The previous theory we have developed concerned the execution of serial components of co-operative computations, these individual computational elements being derived from the per-element computation of a DP operation. We have now developed a new type of serial computation by the aggregation of such computational units into coarser grained macro-responsibilities. We now consider the substitution of such computational elements into our existing models of computation. That is, we return to our previous descriptions of executions for schedules of responsibilities and replace the role of the responsibility with our new computational element, the macro-responsibility. By making this simple substitution we develop a theory for the distributed execution of macro-responsibilities.

In particular, we can derive an algorithm for such an execution (by taking Definition 3.14 and rewriting "macro-responsibility" wherever the "responsibility" concept occurs in the original) which has identical structure to our model of responsibility execution. It can be shown that the theorems proved for this earlier execution also hold true for the derived execution for macro-responsibilities. We can consider a *Simple Macro-Responsibility Synchronizing DP Executions* $\Delta^{\Box}_{\mathrm{sync}}$ by a similar rewriting of Definition 3.17.

From earlier consideration, we know that such an execution has the following properties:

1. it is single-threaded (from Theorem 3.2), and

2. it will deadlock for any distribution which assigns more than one macro-responsibility for an operator Ψ to a single node (from Theorem 3.3).

We turn now to an investigation of techniques for ensuring that deadlock cannot occur in our execution of macro-responsibilities. We know from the second derived theorem listed above that we can guarantee a deadlock free execution if we can

[1]This is achieved by virtue of the fact that all responsibilities in a macro-responsibility are executed serially on the same single-threaded node and thus cannot possibly generate erroneous execution due to interleaving or other timing-related phenomena.

guarantee that a distribution will never assign more than one macro-responsibility from a given DP operation to any node.

We develop a technique for ensuring such properties of a distribution by limiting the kinds of aggregations we allow for our responsibilities. To guarantee that no node is granted more than one macro-responsibility for an operation, we must first guarantee that the decomposition of the operation is into a number of parts less than or equal to the number of nodes. One technique for enforcing such a property is the insistence that we decompose our DP operations into macro-responsibilities which are defined by close associations to indices stored in a given node's memory. That is, we define our first macro-responsibility to incorporate all responsibilities closely associated with the indices of node 1, our second macro-responsibility to group those responsibilities closely associated to node 2, and so on. Such a decomposition generates at most one macro-responsibility per memory (and thus per node), so the total number of macro-responsibilities will never exceed the number of nodes. We formalize this concept in our theory by introducing the notion of *maximal* decompositions:

Definition 3.20 *A macro-responsibility $\Psi_n^{\square}(V\{i_1, \ldots, i_r\})$ is termed* **maximal** *if its close association set $V\{i_1, \ldots, i_r\}$ includes every index of V which is held within a given memory space M_i. That is, the close association set for the macro-responsibility is given by:*

$$V\{i_i, \ldots, i_r\} = \{v \in V_{ind} : P_v(v) = z\}$$

where V_{ind} is the set of indices of V and z is a (fixed) node of the machine (i.e., the node which contains all indices closely associated with $\Psi_n^{\square}$).

The decomposition of a DP operator Ψ into macro-responsibilities is called a **maximal decomposition** *iff it describes Ψ as a set of maximal macro-responsibilities.*

We now have a mechanism for ensuring that the number of macro-responsibilities for a DP operator is less than or equal to the number of nodes in the machine. This goes some way towards guaranteeing deadlock-free execution. We must also, however, limit our distribution of these macro-responsibilities such that each node is only assigned at most one of these macro-responsibilities. The simplest way to introduce such a limitation makes uses of the fact that each macro-responsibility generated by a maximal decomposition is inherently and uniquely related to a single node of the machine. That is, the node whose indices each responsibility in the aggregation have close associations with. If we force our distribution to assign each macro-responsibility to this related node's schedule, then the uniqueness of the relationship guarantees that each node will be given at most one macro-responsibility from the decomposition. Since such a distribution maps the computation for a macro-responsibility to the same node which holds its closely associated indices, we term the distribution *owner-computes* (c.f., Definition 3.11).

Definition 3.21 *A macro-responsibility distribution $\mathcal{D}^\square$ is owner-computes iff it maps each macro-responsibility $\Psi_n^\square(V\{i_1,\ldots,i_r\})$ to some position within the initial schedule $\mathcal{L}_i^0$ of the node whose memory M_i contains all the indices $V[i_1],\ldots,V[i_r]$. in $\Psi_n^\square$'s close association set.*

Our intention in defining the concepts of maximal decomposition and owner-computes distribution was to alleviate the possibilities of deadlock in execution of initial macro-responsibility schedules. We now formalize this desired property in a theorem and prove that the facilities we have provided are sufficient to guarantee that it holds for an arbitrary DP operation.

Theorem 3.4 *Every maximal macro-responsibility decomposition of a DP operator $\Psi(V)$ whose distribution $\mathcal{D}^\square$ is owner-computes will execute to completion on $\Delta_{sync}^\square$.*

Rationale: Assume that the indices of the vector V are distributed across exactly n nodes $N_{i_1}, N_{i_2}, \ldots, N_{i_n}$. We call this set of nodes the *owner set* of V. Now, if we consider a decomposition of Ψ into maximal macro-responsibilities, there must (by definition) be exactly n resultant macro-responsibilities. That is, there will be one macro-responsibility $\Psi_j^\square$ for each of the n memory spaces M_j corresponding to nodes in the owner set of V.

Now we come to the distribution of the n macro-responsibilities $\Psi_k^\square$. We are given that the distribution $\mathcal{D}^\square$ is owner computes, implying that node N_i will be granted all macro-responsibilities which have close associations with indices in its memory space M_i. But by the argument given above, we know that for a node N_i, there is exactly one maximal macro-responsibility with associations to its indices of V (this was how we defined the decomposition). Thus the distribution $\mathcal{D}^\square$ will grant each of the n nodes $N_{i_1},\ldots,N_{i_n}$ an initial schedule $(\mathcal{L}_j^0)^\square = [\Psi_j^\square]$, that is a list of one macro-responsibility. All nodes not in V's owner set are granted empty initial schedules.

After one cycle through the iterative loop defining our distributed execution (i.e., the iterative algorithm from Definition 3.14), each of these n nodes will have removed its macro-responsibility $\Psi_k^\square$ from its schedule and added it to the newly applied set $\bar{A}_i^t$. The discard function $\beta_{\text{sync}}^\square$ will indicate that each of these n nodes may remove these applied macro-responsibilities (since all macro-responsibilities of Ψ have been applied). Thus $\Gamma_{\text{sync}}^\square$ returns *true*, the nodes are each marked *ready*, and the execution is allowed to terminate (since each node now has an empty schedule).

Note that the assumption of owner-computes is actually stronger than necessary: all that is required is a distribution $\mathcal{D}^\square$ which never maps more than one macro-responsibility of Ψ into any given initial schedule $(\mathcal{L}_i^0)^\square$. $\square$

Implication: We have now constructed a framework under which *any* DP operator may be fully executed to completion. All that is required is that the operator be maximally decomposed into macro-responsibilities, and then for those aggregated forms to be distributed in an owner-computes fashion. Since the execution underlying this framework is single-threaded, the paradigm should be easily mapped onto traditional multiprocessor architectures. Indeed, if we inspect the implementation strategies adopted by real-world DP language systems for such distributed architectures (such as the simple SPMD DP execution model alluded to in Chapter 1), it is clear that most adhere to the exact same framework that we have proposed here.

Although our approach does not consider the costs of computation, it is obvious from inspecting the concrete example systems using this paradigm that it can be made efficient. In part this arises from the owner-computes property which places the computations closely associated with a data element on the node for which the demands for that element can be satisfied without inter-node communication. Other, non owner-computes, solutions also exist (as motivated in the preceding proof) — these involve distributing maximal macro-responsibilities between nodes in a way such that at least two nodes are granted the macro-responsibility arising from close associations with indices from a different memory. That is, two or more nodes are charged with the task of performing computations whose entire input set is off-node. Intuitively we would expect such solutions to be significantly more costly than the owner-computes alternative.

3.3 Extending to the Nested Data-Parallel Case

So far we have considered only the execution of single flat DP operators. Our motivation in such analysis was to explore successful techniques for implementing such executions with an eye towards utilizing such approaches in the design of an execution of NDP operators. In this section we explore a simple extension of the concept of the DP operator which allows its consideration under the models we have already described. Through analysis of the application of such models to NDP operators, we discover limitations of the earlier approaches and gain insight into the properties a distributed execution of an NDP operator should possess.

3.3.1 Modelling NDP Operations

The mathematical framework described in the previous sections gives a complete formalism for decomposing DP operations into a collection of serial computations which, when co-operatively performed across a distributed machine, implement the operation. This corresponds to the flat model of DP described in Chapter 1. The primary goal of this analysis is, however, the description of NDP operations and it is to this we now turn. As described in Section 2.1, an NDP operation is an extension of a traditional DP operation whereby the various units which co-operate to implement the operation are themselves DP operations (rather than simply serial blocks of code). We use this characterization to make a formal definition of such operators:

Definition 3.22 *A Data-Parallel Operation Ψ is called* **nested** *if each of its component responsibilities Ψ_j are themselves Data-Parallel Operations. Ψ is termed the* **outer** *DP operation of the nest; Ψ_j is termed an* **inner** *DP operation. By Definition 3.7, such inner operations may themselves be decomposed into responsibilities. The responsibilities of an inner DP operation Ψ_j are denoted $(\Psi_j)_k$.*

The decomposition rule for NDP operations may be written:

$$\Psi^{\text{co-op}}(V) = \{\Psi_j^{\text{co-op}}(V[j]), \quad j \in [0, \dots, V_{len} - 1]\}$$

$$\Psi_j^{\text{co-op}}(V[j]) = \{(\Psi_j^{\text{loc}})_k(V[j][k]), \quad k \in [0, \dots, V[j]_{len}]\}$$

This may be read as: the NDP operator Ψ (a co-operative computation) can be decomposed into exactly V_{len} responsibilities, each of which is itself a DP *operation (i.e., a co-operative computation). These inner DP operations may each be decomposed into $V[j]_{len}$ responsibilities, each of which is a local computation.*

It is clear from this definition that a fully decomposed NDP operation consists of a number of sets of responsibilities corresponding to the inner operations of the nest, plus a group of responsibilities which represent the outer operation. This latter collection has an important semantic role in the modelling, namely the representation of synchronization which must take place between collaborators as the very last phase of the NDP operation (necessary since the outer operation is itself a DP operation).

To begin reasoning about executions of NDP operations, we must define a mechanism whereby these various sets of responsibilities are mapped into the initial schedules of the machine. We extend the concept of the distribution (see Definition 3.10) to accommodate the nested case: an NDP distribution comprises a collection of $N+1$ normal DP distributions (where N is the number of inner DP operations of the nest). One of these distributions describes the spread of responsibilities for the outer operation (these represent the outer synchronization), while the remaining N each describe how responsibilities for one of the inner operations should be distributed. Formally:

Definition 3.23 *A distribution $\mathcal{D}_{nest}$ for a NDP operation $\Psi(V)$ consists of a set of distributions, one for* each *of the inner operations Ψ_j plus one for the outer operation Ψ. $|\mathcal{D}_{nest}| = V_{len}+1$. All such distributions must adhere to the following rule:*

- *all responsibilities from the outer operator Ψ must appear within the resultant schedules $\mathcal{L}_i^0$* later than *any responsibilities for the inner operations Ψ_j.*

Example 3.8: Let Ψ be an NDP operator over the nested vector $V = [[1,2,3],[4,5],[6,7],[8]]$. The individual responsibilities of Ψ are themselves DP operators: Ψ_0, Ψ_1, Ψ_2, and Ψ_3. Each of these operators can itself be decomposed into responsibilities $(\Psi_j)_k$. We consider the process of distributing these inner responsibilities across a machine of size four. We define $\mathcal{D}_{\text{nest}}$ to be this distribution, consisting of five individual distributions $\mathcal{D}_i$, one for each inner operator Ψ_i plus one distribution $\mathcal{D}^{\text{outer}}$ for the outer operator (to represent its synchronization).

We declare $\mathcal{D}^{\text{outer}}$, the distribution of responsibilities from the outer operation Ψ in such a fashion that each Ψ_j is mapped into the schedule for node j. That is, Ψ_0 is granted to node 0, Ψ_1 to node 1, and so on. Since the responsibilities we are discussing here represent the synchronization of the outer DP operation, the positions occupied by these responsibilities within the various schedules must be *after* any responsibilities for inner DP operations. This is a simple representation of the dependency which the outer operation has upon the inner operations: the former cannot possibly complete before each of the latter has completed.

Furthermore we declare $\mathcal{D}_0$ to distribute the three inner responsibilities $(\Psi_0)_j$ as follows: $(\Psi_0)_0$ is allocated to node 0's schedule, $(\Psi_0)_1$ to node 1's schedule and $(\Psi_0)_3$ to node 2. The declarations of $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$ are made in such a way that the following mapping prevails:

$$(\Psi_1)_0 \mapsto \text{node } 3, \quad (\Psi_1)_1 \mapsto \text{node } 0, \quad (\Psi_2)_0 \mapsto \text{node } 1,$$

$$(\Psi_2)_1 \mapsto \text{node } 2, \quad (\Psi_3)_0 \mapsto \text{node } 3$$

Depending on the exact list positions which these distribution functions assign to each responsibility, one possible set of initial schedules for the machine would be:

$$\mathcal{L}_0^0 = [(\Psi_0)_0, (\Psi_1)_1, \Psi_0], \quad \mathcal{L}_1^0 = [(\Psi_0)_1, (\Psi_2)_0, \Psi_1],$$

$$\mathcal{L}_2^0 = [(\Psi_0)_2, (\Psi_2)_1, \Psi_2], \quad \mathcal{L}_3^0 = [(\Psi_1)_0, (\Psi_3)_0, \Psi_3]$$

It is worthwhile noting that each of the nodal initial schedules now contains responsibilities for a number of DP operations (in contrast to the simple cases we have explored previously where only one DP operation was distributed). This factor, we shall see, has ramification upon the execution of such NDP distributions and in particular the limitations that must be put in place to ensure a deadlock-free execution.

Observation: Once an NDP operation Ψ has been decomposed fully into responsibilities and distributed by $\mathcal{D}_{nest}$, we have produced a set of initial schedules $\mathcal{L}_i^0$, one per node, which are in a form ready for input into the distributed executions $\Delta(\beta, \Gamma, \mathcal{M})$ we have previously defined (in Definition 3.17) for non-nested DP operations.

Implication: All properties and restrictions shown true for the execution of simple DP operators on Δ also apply to such executions of NDP operators. Specifically:

1. such executions are single-threaded

2. inner distributions which map more than one responsibility from a given DP operation onto any node of the machine will generate deadlock in the execution.

As with the consideration of executing flat DP operators, this second property poses a severe limitation to the applicability of our model. If either the inner or outer vector of a nest has more elements than the size of the machine, then it will inevitably be true that one node is assigned more than one responsibility from the DP operation across that vector, and hence full execution will not be possible.

In the flat DP case, we overcame this problem by considering an aggregating approach to responsibilities. We now turn to the application of this same approach to the inner DP operations of a nest.

3.3.2 Macro-Responsibilities for NDP

In dealing with executions of flat DP operations, we found that responsibility aggregation was a useful technique for eliminating deadlock possibilities arising from multiple responsibilities from a single DP operation being assigned to a single node. We can apply these same concepts to our emerging NDP execution model in order to solve these same problems.

We do this by changing our model for decomposing NDP operations: rather than dividing all inner DP operations from the nest into individual responsibilities, we consider their division into maximal macro-responsibilities (see Definition 3.20). The same approach is adopted in the decomposition of the outer DP operation. Once all operations have been divided into maximal macro-responsibilities we distribute the various sets across the nodes using a distribution $\mathcal{D}^{\square}_{\text{nest}}$ which is owner-computes. We then use the simple synchronizing macro-responsibility execution (i.e., the algorithm from Definition 3.14 expressed in terms of macro-responsibilities) to execute the initial schedules generated by this distribution.

For the flat DP case, where we were dealing with the execution of only a single DP operation, these precautions — maximal macro-responsibility decomposition plus owner-computes distribution — were enough to guarantee deadlock-free execution. However, the fact that we are now performing a number of synchronizing operations in an interleaved fashion, means that these measures are no longer sufficient to guarantee that cyclic dependencies (and thus deadlock) cannot arise during the execution. A formulation of this result, and an illustration of how such dependencies can arrive, are found below.

Theorem 3.5 *There exist macro-responsibility distributions $\mathcal{D}^{\square}_{nest}$ for a nested DP operation $\Psi(V)$ for which the simple synchronizing maximal macro-responsibility based execution $\Delta^{\square}_{sync}$ will not complete.*

Rationale: We prove this by demonstrating one distribution which introduces a cyclic dependency during an execution of $\Delta^{\square}_{\text{sync}}$.

Consider a distributed machine of two nodes across which two vectors $V1$ and $V2$, each of four indices, is distributed. The partitioning of $V1$ and $V2$ is such that node 0 holds indices $V1[0], V1[1], V2[0]$ and $V2[1]$ while node 1 holds $V1[2], V1[3], V2[2]$ and V2[3]. Next consider an NDP operation Ψ which has two inner operations, one of which takes place across $V1$, the other of which is across $V2$.

In such a system there are three DP operations: the outer Ψ and the two inner operations Ψ_0 and Ψ_1. Each of these we decompose according to a maximal macro-responsibility decomposition. This leaves us with the following sets of macro-responsibilities:

- $\Psi^{\square}_0$ and $\Psi^{\square}_1$ (representing the outer operation)
- $(\Psi_0)^{\square}_0 = \{(\Psi_0)_0, (\Psi_0)_1\}$ and $(\Psi_0)^{\square}_1 = \{(\Psi_0)_2, (\Psi_0)_3\}$ (representing the first inner operation)
- $(\Psi_1)^{\square}_0 = \{(\Psi_1)_0, (\Psi_1)_1\}$ and $(\Psi_1)^{\square}_1 = \{(\Psi_1)_2, (\Psi_1)_3\}$ (representing the second inner operation)

The first member of each of these groups has associations with the indices of node 0, while the second has only associations with node 1. Thus any distribution which maps the first member of a pair to node 0's schedule and the second to node 1's schedule is owner computes.

Consider now a single such distribution $\mathcal{D}^{\square}_{\text{nest}}$ which generates the following (owner computes) initial schedules for nodes 0 and 1:

$$\mathcal{L}^0_0 = [(\Psi_0)^{\square}_0, (\Psi_1)^{\square}_1, \Psi^{\square}_0], \quad \mathcal{L}^0_1 = [(\Psi_1)^{\square}_0, (\Psi_0)^{\square}_1, \Psi^{\square}_1]$$

Note that this definition assigns node 0 with a macro-responsibility for the first inner DP operation, followed by one for the second DP operation. Conversely, the schedule of node 1 contains a macro-responsibility for the second operation followed by one for the first.

We now turn to the execution of these schedules. This begins by each node applying the first of its macro-responsibilities and then blocking at a synchronization point (in algorithmic terms, each node moves a macro-responsibility from the head of the schedule to the applied set, evaluates the discard function to discover that neither is permitted to remove elements from that set, and finally applies the continuation function which dictates that the node should be in an unready state).

If we consider the situation on node 0: it has blocked because the first inner DP operation Ψ_0 requires synchronization. Thus node 0 will only be unblocked at some time after the other node's section of Ψ_0 has been applied. Until that time node 0 will be idle.

If we consider the situation on the other node: this node has blocked because of the synchronization required for the second inner DP operation Ψ_1. This node cannot resume execution until such time as node 0 performs its part of this inner operation.

We can see that this is a cyclic dependency and defines a deadlock. The fact that there is at least one such distribution which generates such a result even in the presence of a maximal decomposition and an owner-computes distribution, proves the theorem. □

It seems clear from the preceding proof that the standard macro-responsibility approach fails for NDP operations because it does nothing to avoid cyclic dependencies which arise from the relative ordering of different operations on different nodes. We now construct a means for limiting distributions of NDP operations in such a way that we can ensure that such deadlock cannot arise within our model.

We begin by defining a simple mechanism for determining whether a distribution specifies a consistent ordering between two DP operations on all nodes of the machine. We say that a distribution is *pairwise sorted* with respect to two DP operations if, considering those nodes of the machine which have responsibilities for both operations, all such have those responsibilities in the same order within their schedule. Distributions which do not maintain this property for some pair of DP operations are guaranteed to deadlock in the same fashion as the distribution shown in the preceding proof.

Definition 3.24 *A macro-responsibility distribution $\mathcal{D}^{\square}_{nest}$ of a nested DP operation $\Psi(V)$ is said to be* **pairwise sorted with respect to** k **and** l *iff the following relationship exists between the initial schedules $(\mathcal{L}^0_i)^{\square}$ of the distributed memory machine's nodes:*

Let $\mathcal{N}$ be the set of nodes to which $\mathcal{D}^{\square}_{nest}$ has mapped macro-responsibilities of both Ψ_k and Ψ_l $(k, l \in [0, \ldots, V_{len} - 1], k \neq l)$. Then if $z \in \mathcal{N}$,

$$\mathbf{pos}\;((\mathcal{L}^0_z)^{\square}, (\Psi_k)^{\square}) < \mathbf{pos}\;((\mathcal{L}^0_z)^{\square}, (\Psi_l)^{\square}) \Longleftrightarrow \mathbf{pos}\;((\mathcal{L}^0_i)^{\square}, (\Psi_k)^{\square}) < \mathbf{pos}\;((\mathcal{L}^0_i)^{\square}, (\Psi_l)^{\square}), \forall i \in \mathcal{N}$$

where $\mathbf{pos}\;((\mathcal{L}^0_i)^{\square}, (\Psi_j)^{\square})$ *denotes the position of node N_i's sole macro-responsibility $(\Psi_j)^{\square}_n$ of Ψ_j within the list $(\mathcal{L}^0_i)^{\square}$.*

The notion of pairwise sortedness is a means of expressing that for a given pair of DP operations, a consistent ordering (for their macro-responsibilities) exists across the entire machine. This is enough to guarantee that no deadlocks will be introduced into the computation because of conflicts in the sychronization of those two operations. We need to broaden this concept, however, to deal with an arbitrary number of DP operations. For an NDP operation to have no ordering-induced deadlocks it must be true that for any two DP operations defined within the nest, a pairwise ordering is guaranteed by the distribution. We term distributions which can define this kind of operator-wide sortedness to be *totally sorted.*

Definition 3.25 *A macro-responsibility distribution $\mathcal{D}^{\square}_{nest}$ of a nested DP operation $\Psi(V)$ is said to be* **totally sorted** *iff $\forall k, l \in [0, \ldots, V_{len} - 1], (k \neq l)$, $\mathcal{D}^{\square}_{nest}$ is pairwise sorted wrt k and l.*

We have now arrived at a point where we have defined a means of specifying a distribution which can guarantee that no deadlock arises through differing orderings within schedules. We now add this notion to our basic macro-responsibility execution and show how the resulting model is guaranteed deadlock free for all NDP operations.

Theorem 3.6 *A single-threaded synchronizing distributed execution $\Delta^{\square}_{sync}$ can complete the execution of an NDP operator Ψ on a distributed memory machine of size N only if:*

1. *$\Delta^{\square}_{sync}$ is the simple macro-responsibility synchronizing DP execution derived from substituting "macro-responsibility" in place of "responsibility" in Definitions 3.14 and 3.17, and*

2. *the outer operation Ψ is decomposed into N or fewer macro-responsibilities $\Psi_m^\square$, and*

3. *this set of macro-responsibilities is distributed such that no node has more than one within its schedule, and*

4. *Ψ's inner DP operations Ψ_j are each decomposed into fewer than N macro-responsibilities $(\Psi_j)_n^\square$, and*

5. *these sets of macro-responsibilities are distributed by a totally sorted distribution $\mathcal{D}_{nest}^\square$ which assigns no more than one macro-responsibility from Ψ_j to any node*

Rationale: We know that to be called synchronizing, a distributed execution must have a discard function of the form:

$$\beta(A_i^{t_i}, t_i) = \{\Psi_l^\square \in A_i^{t_i} : \mathbf{complete}(\Psi_l^\square, t_i)\}$$

for a scheme of responsibility decomposition (or aggregated responsibility decomposition) of a DP operator into sub-computations called $\Psi_l^\square$.

We also know from earlier analysis that an execution can be termed single-threaded if it has a continuation predicate that causes a node to block (i.e., remain unready) whenever the local applied set A_j^t is non-empty. That is, the function Γ must return *false* whenever A_j^t is non-empty. In the case where the applied set is empty, the function must return *true* for at least some of its applications; otherwise a node N_i would never become *ready* and hence never complete the execution of its schedule. Since Γ is a function of the applied set, if it returns *true* for some empty A_j^t it must do so for all empty A_j^t. Hence, by requiring single-threadedness, we have effectively defined:

$$\Gamma_{\text{sync}}^\square(A_i^{t_i}) = \begin{cases} \text{true} & \text{if } A_i^{t_i} = \phi \\ \text{false} & \text{otherwise} \end{cases}$$

Thus, merely by the assumption of properties of synchronizing and single-threaded execution, we have fixed the two auxiliary functions of the distributed execution algorithm. However, variance still exists between executions by virtue of the kinds of elements contained within the nodal schedules. For example, we have seen already the cases where each list held individual responsibilities and the case where it held macro-responsibilities. Different properties were proved for these two executions. The task, therefore, for the remainder of this proof is to establish that an execution with

β and Γ as defined above can always complete execution of an NDP operator if: its schedule elements are macro-responsibilities; there are fewer than N of them for both the outer operation and each inner operation; and the distribution of such elements to initial schedules is by a totally-sorted distribution which grants no more than one outer macro-responsibility to any schedule. We prove each of these requirements in turn.

Requirements 1 and 2: Limits on Decomposition and Distributions of Outer Operator Ψ:

Recall that the outer operator Ψ is itself a DP operator. Thus all properties proven previously for the execution of DP operators also holds true for Ψ. Theorem 3.4 and its proof describe how, for the macro-responsibility based distributed execution, termination is not possible when one node N_i has been granted more than one macro-responsibility from a DP operator. Applying this result in the context of Ψ gives the stated result: to terminate, an initial schedule $(\mathcal{L}_i)^\square$ cannot have more than one macro-responsibility from the outer operator Ψ.

Requirement 3: Decomposition of $\Psi_j(V[j])$ into $\leq$ N Macro-Responsibilities:

For Ψ to execute to completion, each of the inner DP operators Ψ_j must execute to completion. From an earlier analysis we know that the algorithm at the heart of the execution will not complete if the initial schedule of any node N_i holds more than one responsibility/macro-responsibility for the same DP operator Ψ_j. Thus, if we wish a given inner DP operator Ψ_j to be completely executed, we must ensure that no node has more than one of its sub-computations. If there are N nodes, this means that there must be at most N sub-computations.

Given the fact that we can make no assumptions about the extents of the vector $V[j]$, this means that the decomposition of Ψ_j must be into macro-responsibilities, and further that there must be fewer than N such macro-responsibilities in each decomposition.

Requirement 4: The Distribution $\mathcal{D}^\square$ must be Totally Sorted:

We prove that this is a requirement for complete execution of the NDP operator Ψ by contradiction. We assume that there is a distribution $\mathcal{D}^\square$ of macro-responsibilities which will complete under a single-threaded synchronizing execution, but is not totally sorted. The fact that $\mathcal{D}^\square$ it is not totally sorted means that there exists one pair k

and l which are not pairwise sorted. That is, if $\mathcal{N}$ is the set of nodes whose initial schedules have macro-responsibilities from both Ψ_k and Ψ_l, then $\exists$ nodes z_1 and z_2 in $\mathcal{N}$ for which:

$$\mathbf{pos}\ ((\mathcal{L}^0_{z_1})^\square, (\Psi_k)^\square) < \mathbf{pos}\ ((\mathcal{L}^0_{z_1})^\square, (\Psi_l)^\square) \not\Leftrightarrow \mathbf{pos}\ ((\mathcal{L}^0_{z_2})^\square, (\Psi_k)^\square) < \mathbf{pos}\ ((\mathcal{L}^0_{z_2})^\square, (\Psi_l)^\square)$$

Specifically it may be true that z_1 and z_2 have initial schedules such that in $(\mathcal{L}^0_{z_1})^\square$, Ψ_k's macro-responsibility precedes Ψ_l's, while in $(\mathcal{L}^0_{z_2})^\square$ the Ψ_l's macro-responsibility occurs earlier in the list. This is precisely the situation explored in the Proof for Theorem 3.5, which showed such schedules could never complete on a synchronizing single-threaded execution. This contradicts the assumption that the non-totally sorted $\mathcal{D}^\square$ under consideration would complete for such an execution. Therefore we can conclude that no non-totally sorted distributions exist which lead to full execution on the synchronizing single-threaded execution.

We have thus shown that the desired three properties must hold for an NDP decomposition to execute to completion on a synchronizing single-threaded execution. $\square$

Corollary: The totally sorted distribution of the maximal macro-responsibilities of $\Psi(V)$ will always execute to completion on a single-threaded synchronizing execution.

Rationale: By the definition of maximal decomposition of a DP operator, each of the inner DP operators $\Psi_j(V[j])$ will be decomposed into exactly N_j macro-responsibilities, where N_j is the number of memory spaces which hold indices of $V[j]$. Clearly each of these N_j must be less than N (since N is the count of all nodes, and hence all memory spaces).

The rest follows from Theorem 3.6. $\square$

We have now reached a scheme for NDP execution which we have proven to always terminate across any DP operator. That is, we have arrived at a design which meets the requirements with which we began this chapter. The scheme guarantees completeness of execution by imposing a number of restrictions on the general distributed execution. Specifically, we are required to ensure that the decomposition of every DP operation in a nest consists of a number of macro-responsibilities which is less than or equal to the number of nodes in the machine. Furthermore, these macro-responsibilities must be distributed in a fashion which grants no more than one macro-responsibility from an operation to any given node. Finally, the scheme places a further requirement upon the distribution: it must be totally sorted, that is we must somehow eliminate the possibility of node N_a's schedule listing a macro-responsibility for Ψ_p before one for Ψ_q while at the same time node N_b defines the opposite relative orientation for its macro-responsibilities of Ψ_p and Ψ_q.

We know from an earlier result that the scheme in question is single-threaded. Thus we term this implementation of NDP the *Single-Threaded Sorted Paradigm (STSP)*. The properties of the scheme suggest that it would offer a suitable framework for cheaply implementing NDP on a conventional multi-processor archiecture (i.e., one composed of single-threaded nodes). We examine the practical considerations of defining such an implementation (with especial reference to efficient support for irregular computation) in the following chapter.

3.4 A Multi-threaded Alternative

The executions we have considered thusfar in this chapter have been limited to those that arise when each node of a distributed memory machine embodies a single-thread of control. We began modelling such executions based on the fact that they represent the forms of execution most closely related to a large class of real-world multiprocessor architectures. This semantic closeness argues in favour of a straightforward and efficient implementation of the models upon such hardware. However, in the process of augmenting our basic single-threaded model to ensure deadlock-free execution of first DP and then NDP codes, restrictions and complexities were added to the basic formalism. These introduce implementation costs which, to some extent, offset this basic low-cost mapping process.

It is therefore worthwhile returning to our original assumption — that our executions must be single-threaded — and considering the consequences of relaxing this limitation. In this section we undertake such an analysis, investigating the possibilities that a multi-threaded approach affords. We begin by defining a simple multi-threaded execution model in terms of our existing theory and then continue by proving that this very basic definition is sufficient to guarantee deadlock-free executions for both DP and NDP operations, irrespective of decomposition and distribution.

3.4.1 Modelling Multi-Threaded Execution

The basis for defining a multi-threaded model of distributed execution exists already within the mathematical framework laid down in Section 3.2.2. The concept of an execution model's threadedness is described in Definition 3.16 which relates back to the notion of the model's applied set as a repository for blocked threads of control.

To define a multi-threaded model of distributed execution all that is required is that we return to our distributed execution algorithm (Definition 3.14) and define a continuation protocol which allows for a node's applied set to grow beyond a singleton. That is, we must define a continuation function Γ which enables a node to execute a further schedule element (i.e., become *ready*) even in the case where one or more previous responsibilities are still awaiting sychronization events. Because the model must still enforce a DP style of synchronizing execution, we must still adopt a discard function based upon operation completion (i.e., the function from Definition 3.17). We formally define our first-cut multi-threaded execution as follows:

Definition 3.26 A Simple Multi-threaded Synchronizing DP Execution: *We define a multi-threaded synchronizing execution $\Delta^{\mathcal{M}}_{sync}$ by making a functional definition of the discard function $\beta^{\mathcal{M}}_{sync}$, and defining a continuation predicate $\Gamma^{\mathcal{M}}_{sync}$.*

$$\beta^{\mathcal{M}}_{sync}(A_i^{t_i}, t_i) = \{\Psi_j \in A_i^{t_i} : \textbf{complete } (\Psi_j, t_i)\}$$

$$\Gamma^{\mathcal{M}}_{sync}(A_i^{t_i}) = \mathit{true}$$

Multi-threaded execution is introduced into the model by the fact that the continuation policy (defined by Γ) indicates that a node should remain *ready* irrespective of the contents of its applied set. That is, even if there are blocked threads of control upon the node (represented by responsibilities in the applied set), the execution is still permitted to execute a further thread of control (by applying a further responsibility). This aspect of the design marks it as multi-threaded. We formalize this rationale into a formal proof:

Theorem 3.7 *$\Delta^{\mathcal{M}}_{sync}$ is multi-threaded.*

Proof: All that is required is that we prove that the execution can cause the applied set A_i^t of at least one node to contain more than one element. We choose to prove this by proving that $\Delta^{\mathcal{M}}_{\mathrm{sync}}$ can generate applied sets of any cardinality ≥ 0.

Let $P(n)$ be the proposition that $\Delta^{\mathcal{M}}_{\text{sync}}$ can generate an applied set A^t_i containing n elements.

$P(0)$ is true, since the applied set of each node begins as the empty set.

Assume that $P(n)$ is true. That is, at some time t at least one node's applied set A^t_i contains exactly n members. We assume, without loss of generality, that this occurs on a node N_i which has at least one element remaining in its local schedule $\mathcal{L}^t_i$.

At the end of the last iteration of the algorithm, $\Gamma^{\mathcal{M}}_{\text{sync}}$ would have been applied to A^t_i, returning *true*. Thus, at the end of this cycle of the algorithm, node N_i is marked *ready*. Consider the next cycle of the algorithm. The sub-steps of Step 2 will be followed, since the node is *ready*. As part of these sub-steps, node N_i will remove the head element from its schedule, apply it to the local processing element and add it to the newly applied set $\bar{A}^{t+1}_i$. Since this set will thus be non-empty, the sub-steps of algorithm Step 3 will be followed, and the discard function $\beta^{\mathcal{M}}_{\text{sync}}$ will be called. Depending on the extent to which the DP operations being co-operatively executed have completed, this set will contain 0 or more applied responsibilities.

Consider the case where this set is empty. Then the applied set A^{t+1}_i assumes the value of $A^t_i \cup \{\Psi_j\}$. That is, at the end of the cycle, the cardinality of the applied set will be $n+1$. Thus we can deduce that it is possible for the execution to generate an applied set of size $n+1$. That is, $P(n+1)$ is true.

By the principle of induction, the fact that $P(0)$ is true and that $P(n) \Rightarrow P(n+1)$ implies that $P(n)$ is true for all $n \geq 0$. $\square$

3.4.2 Multi-Threaded Execution of Flat DP Operations

We now consider how the multi-threaded execution we have defined can be applied to the evaluation of flat DP operations. We assume that such operations are decomposed into a collection of responsibilities (as outlined in Definition 3.7) which are distributed into the initial schedules of the various nodes. The execution proceeds by following the algorithm described in Definition 3.14 with policy functions $\Gamma^{\mathcal{M}}_{\text{sync}}$ and $\beta^{\mathcal{M}}_{\text{sync}}$ from Definition 3.26.

We are interested in learning the conditions under which such an execution will encounter a deadlock. If we consider the situations which caused cyclic dependencies in our single-threaded models for DP execution, we see that all are centred upon factors related to a node being unable to execute further responsibilities while awaiting a synchronization. In the multi-threaded model there is no such blockage of the entire node's control and thus no opportunity for deadlock due to a node's inability to execute a later responsibility because of the synchronization requirement of a former responsibility. Since such dependencies represent the only deadlock cases for the flat DP model, we would intuitively expect that a model which inherently avoids such dependencies — such as our multi-threaded execution — would be naturally deadlock-free for any distribution of a DP operation's responsibilities. We now formally prove this intuition.

Theorem 3.8 $\Delta^{\mathcal{M}}_{sync}$ *can execute to completion* any *distribution $\mathcal{D}$ of a flat DP operator Ψ.*

Rationale: Consider a decomposition of Ψ into a set of local sub-computations which are distributed across nodes of a machine of size s by a distribution $\mathcal{D}$. We consider the initial schedules $\mathcal{L}^0_i$ of these nodes, defining a quantity T, the *execution time* as follows:

$$T = \max_{i \in [0,\ldots,s-1]} \mathbf{len}\ (\mathcal{L}^0_i)$$

where **len** is the length function for lists.

Now consider the execution of a single cycle of the algorithm. Since $\Gamma^{\mathcal{M}}_{\mathrm{sync}}$ is *true* for all applied sets, we can deduce that every node N_i of the machine will be tagged *ready* at the beginning and end of every iteration of the loop. Since every node will be ready at the beginning of Step 2, every node with a non-empty schedule will shorten its schedule by one, adding the head element to the newly applied set. After the T'th cycle, every node N_i will have emptied its schedule and the algorithm will complete.

Note that this is independent of the distribution $\mathcal{D}$ chosen for the sub-computations. Specifically, the execution still completes if a node N_i is granted more than one sub-computation from the same DP operation. □

3.4.3 Multi-Threaded Execution of NDP Operations

In the previous section we considered the application of our multi-threaded execution model to the evaluation of flat DP operations. We turn now to a similar analysis of the model's ability to handle the NDP case.

We model an NDP operation in a fashion identical to that considered earlier for the single-threaded execution (Definition 3.22). Each of the various outer and inner DP operations of the nest are decomposed into responsibilities and then distributed into the initial schedules of the various nodes. Multi-threaded execution is then simulated by following the distributed execution algorithm (Definition 3.14) with the multi-threaded policy functions (Definition 3.26) substituted.

In the consideration of deadlock conditions for our single-threaded models, NDP operations were problematic on two counts. Firstly there was the possibility of cyclic dependency due to a node being called upon to execute two responsibilities from the same operation — where the second could not be performed until after the first had synchronized with it. We have already shown in the previous section that the multi-threaded execution manages this case without deadlock. A second cause of cyclic dependency in NDP execution was the possibility of inconsistent ordering of responsibilities in the various nodal schedules. This type of deadlock is also naturally avoided in the multi-threading execution by virtue of the fact that no node is ever disallowed from continuing executing its schedule even when a synchronization is pending for one of its threads.

By this argument we would expect our simple multi-threaded execution to also avoid deadlock for any distribution of an NDP operation. We now prove this assertion.

Theorem 3.9 $\Delta^{\mathcal{M}}_{sync}$ *can execute to completion* any *distribution* $\mathcal{D}_{nest}$ *of an NDP operator* Ψ

Rationale: This situation is similar to that described in the proof for the previous Theorem. If we decompose each inner DP operator Ψ_j into sub-computations and distribute these across nodes N_i, we can define an execution time T in a fashion similar to above. By an identical argument, the execution of this distributed computation will complete after exactly T iterations of the algorithm.

Note that this result holds true even for distributions $\mathcal{D}_{\text{nest}}$ which assign more than one sub-computation for a single DP operator Ψ_j to a single node. It also holds true in the case of a non-totally sorted distribution. □

Implication: If we were to implement this execution model on a distributed memory machine which supported multiple concurrent threads on each node, we would have an environment in which both DP and NDP operations would execute to completion for any possible distribution of responsibilities. That is, the multi-threaded execution we have examined in this section forms another design which meets the requirements with which we began this chapter. The implementation of this candidate NDP scheme, termed the *Multi-Threaded Paradigm (MTP)*, is considered in the following chapter, with a specific emphasis on the costs and benefits associated with supporting irregular styles of computation.

3.5 Summarizing the Mathematical Analysis

In this chapter we have provided a lengthy formal treatment of the distributed execution of DP and NDP operations. Beginning with models for key concepts such as the distributed aggregate and the decomposition of a DP operation, we described an algorithm which can be considered as a mathematical model of a real-world execution. The exact nature of the execution modelled under our formalism is guided by two policies: one governing the synchronization between nodes, the other defining the situations in which the node's control should be blocked. By varying these policies we can arrive at model executions with widely varying characteristics.

We began by considering a very simple synchronizing execution which is functionally identical to the SPMD style found in common flat DP implementations on machines such as the CM-5. Both real and modelled executions portray single-threaded computational forms. Analysis of our model showed that it could only be guaranteed deadlock free if we adopted a mechanism for aggregating the individual per-node computations in a way which ensured that each node only had one role to play in the co-operative computation. With this limitation in place, the model provided a good basis for executing flat DP operations.

We turned then to the consideration of NDP computation, at first applying the same model found successful for flat DP execution to the new context. Analysis showed that even given the safeguards put in place to ensure deadlock in the flat DP case, this single-threaded model had difficulties in completing the execution of NDP computations distributed in certain fashions. Specifically we discovered that in the NDP case, a second type of cyclic dependency could be set up in the machine by virtue of a lack of common consistent ordering within the schedules. To eliminate the possibility of such situations arising we further restricted our model by placing strong requirements of ordering upon the distribution of work. The resulting model, the *Single Threaded Sorted Paradigm (STSP)*, can guarantee deadlock-free execution for any distribution of an arbitrary NDP computation.

In response to the complex restrictions which were required to build a single-threaded environment which could safely execute NDP operations, we considered an alternative approach — a multi-threaded model of distributed computation. While such a model is less well suited to implementation on SIMD or SPMD hardware, it naturally free from the kinds of cyclic dependencies which plague the single-threaded models. Thus it is capable of executing both DP and NDP operations to completion without any further need for semantic mechanism or limitations on distribution. This simplicity in some ways recommends the *Multi-Threaded Paradigm (MTP)* as a basis for an NDP implementation.

In the next chapter we analyze these two deadlock-avoiding strategies for NDP execution from a practical perspective, deducing performance costs and characteristics for each. We carefully consider the impact of program irregularity upon the two execution models, highlighting the costs incurred due to synchronization and remote-access. Based upon the results of this analysis we build a case for choosing one of the strategies — the Multi-Threaded Paradigm — as a more suitable basis for the execution of irregular scientific programs.

Chapter 4

Applying the Mathematical Analysis

In the previous chapter, we considered mathematically the problem of executing general NDP operators across vectors (and vector nests) of arbitrary partitioning. We introduced a number of formalisms which abstractly represent aspects of real-world distributed execution and and DP/NDP operations. In this chapter we relate this mathematical model back to actual hardware and software implementations by evaluating the approach from two practical perspectives.

Firstly, we consider the model's general applicability as a descriptive tool for DP execution models, focussing on the modelling of software and hardware implementations of DP; we show explicitly relationships between our formalism and several such implementations as discussed previously in Chapters 1 and 2.

Following this analysis, we consider our abstract model from a second important perspective: that of estimating the performance characteristics of real-world (distributed memory) implementations of NDP execution as described by the model. In Chapter 3, we identified two models — the "Single-Threaded Sorted Paradigm (STSP)" and the "Multi-Threaded Paradigm (MTP)" — which guarantee termination for all possible input partitioning. We now consider performance realizable in implementations of each of these models by firstly identifying the principal functionalities the models demand of an implementation, and then estimating the cost of supplying such key functionalities. We describe a loose cost metric for implementations — both in a general form (irrespective of the target architecture), and also for the particular case of implementations targetting SPMD architectures[1].

We build a case for choosing one of the two candidate models as the most likely to deliver good performance in a SPMD environment, based on the estimates of implementation overheads derived from the SPMD metrics. The choice we make of abstract model (the MTP) and target architecture (SPMD) forms the context for the work reported throughout the remainder of this thesis.

4.1 Modelling Existing DP Paradigms

The formalisms developed in the previous chapter are very general in their expression of DP and the execution of DP operations — most existing hardware and software implementations of DP are expressible within the framework. We turn now to the task of constructing such a descriptive form for several well-known DP paradigms. These representations form a good basis for comparisons between the various DP paradigms and the two candidate NDP paradigms (the STSP and the MTP) derived from our modelling work, defining a place for each of the two new solutions in a spectrum of DP/NDP implementations.

4.1.1 Modelling SIMD Hardware: the CM-2

As described in Section 1.1.1, the SIMD model of computation, as typified by the hardware architecture of the Thinking Machines CM-2, is one in which all nodes of a parallel machine execute the same code stream, synchronizing after each instruction. We can model such behaviour in terms of the Simple Synchronizing Distributed Execution (Definition 3.17). We begin by modelling each machine instruction M_i in a CM-2 program as a DP operation M'_i, which we may decompose into exactly N responsibilities (where N is the number of CM-2 nodes to be modelled). Each of these N responsibilities of M'_i has the computational semantics of the source instruction M_i — that is, our conceptual DP operation is made up of N copies of the computation for M_i. We can think of these individual responsibilities as formal representations of

[1]We choose to concentrate attention on SPMD implementations of our models for two reasons. Firstly, such architectures are commonly considered to be highly conducive to extracting good performance from DP execution. A second reason lies in the common availability of such architectures in the scientific computing community, the audience most likely to find NDP languages of immediate use.

the instruction copies that are broadcast to the nodes during a real SIMD execution. Mathematically we can represent the modelling as:

$$M_i \textbf{ is modelled by } \text{DP operation } M'_i = \{M_{i_0}, M_{i_1}, \ldots, M_{i_{N-1}}\}$$

We distribute the responsibilities from our conceptual DP operation M'_i into the initial schedules of our N nodes in such a way that the j^{th} responsibility from M'_i (the DP operation representing the i^{th} instruction) is placed in the schedule of node j at position i. That is, exactly one copy of machine instruction M_i's computation is placed at each node's schedule (at position i), simulating the broadcast of the instruction across the nodes of the machine.

If we consider the global program as a sequence of instructions $M_0, M_1, \ldots, M_{r-1}$ (each modelled as a DP operation), we can deduce the schedule that such a distribution (broadcast) generates for each node of the machine. An arbitrary node j is allocated the initial schedule $\mathcal{L}^0_j$ as follows:[2]

$$\mathcal{L}^0_j = [M_{0_j}, M_{1_j}, \ldots, M_{r-1_j}]$$

Note that this schedule mirrors exactly the global CM-2 program we are modelling: each node is called upon to issues copies of the same instructions in the same order.

In this model of the CM-2, an execution begins with each abstract node applying the responsibility representing its copy of the first machine instruction M_0. The nodes move this responsibility into their local applied sets and are unable to continue execution until such time as these sets are emptied by discard. This occurs only when all nodes have completed the execution of their copy of M_0 (i.e., their responsibility of M'_0). The process then begins anew for the next schedule element, and so on until the entire series of machine instructions has been executed by each node. Clearly, in this execution model, every machine instruction represents a point of synchronization in the execution, which is precisely the style of execution embodied within SIMD hardware such as the CM-2.

4.1.2 Modelling SPMD Hardware: the CM-5

The CM-5 represents a class of hardware architectures targetting the traditional SPMD styles of execution (see Section 1.1.2) in which each node of the parallel

[2]Recall that in our notation (Definition 3.9) $\mathcal{L}^t_k$ represents the list of responsibilities awaiting execution on node k at time t.

machine begins with an identical code stream from which it executes instructions independently. Contained within the code stream are a number of barriers, points at which all nodes in the machine must synchronize before continuing.

We model this model of execution by again beginning with the Simple Synchronizing Distributed Execution (Definition 3.17). We consider the CM-5 instruction set as comprising two types of instructions — those that introduce a global synchronization point (e.g., the barrier synchronization, hardware implemented parallel prefix operations) and those that do not (e.g., simple ALU instructions and memory operations).

In line with the approach adopted in the modelling of the CM-2, we represent operations from the former class as DP operations which are decomposed into exactly N responsibilities (where N is the number of CM-5 nodes modelled). As before, such individual responsibilities are merely copies of the machine instruction, and have semantics identical to that of the instruction itself. We can consider such elements as copies of the machine instruction that are broadcast (one to each node) during the execution. In formal terms we define the modelling for a synchronizing instruction M_i as follows:

M_i **is modelled by** DP operator $M'_i = \{M_{i_0}, M_{i_1}, \ldots, M_{i_{N-1}}\}$ if M_i is synchronizing

Alternatively, if the instruction M_i is not a synchronizing instruction, we model it as a set of N local computations — that is, a set of N responsibilities which are not part of any DP operation, and which may thus execute independently without any required synchronization. Each responsibility in the set is given the computational semantics associated with the original instruction. We can again think of these as being copies of the instruction which are to be broadcast to the various nodes of the machine. Mathematically we describe the modelling as:

M_i **is modelled by** the set $\{L_i^0, L_i^1, \ldots, L_i^{N-1}\}$ if M_i is not synchronizing

The independence of the responsibilities within such a set (i.e., each may be executed without a co-ordination with other set members) gives the desired property of execution without synchronization. In terms of our mathematical model, the responsibilities from such sets may be removed from a node's applied set at any time after their execution, without need to wait for a synchronization event to take place. This property can be represented in terms of the **complete** predicate: passing responsibilities corresponding to independent local computations (i.e., set members) to **complete** at any time after the execution of such responsibilities will always yield a *true* value.

As the final stage in modelling the CM-5 we define a distribution for assigning the responsibilities for each instruction to the initial schedules of the nodes. We can again consider this operation to be analogous to a broadcast of machine instructions to computational elements in the CM-5. We define our distribution in two parts, one rule for responsibilities arising from synchronizing instructions (i.e., parts of a DP operation) and another for those generated during the modelling of non-synchronizing instructions (i.e., responsibilities which have no inter-relationship). Mathematically, our rules are:

$$\mathcal{D}_{\text{CM-5}}\ M_{i_j} \mapsto \text{ position } i \text{ in } \mathcal{L}_j^0$$
$$\mathcal{D}_{\text{CM-5}}\ L_i^j \mapsto \text{ position } i \text{ in } \mathcal{L}_j^0$$

These functions collectively implement a simple scheme: if we consider our global CM-5 program to be composed of machine instructions $M_0, M_1, \ldots, M_{r-1}$, then the distribution guarantees that every node is granted exactly one copy of each of these instructions. Furthermore the nodal copies are ordered in exactly the same way as the original program — the local version of M_0 heads the schedule, followed by the representation of M_1 and so on.

Thus, for example, a CM-5 instruction stream consisting of four non-synchronizing instructions M_0, M_1, M_2, M_3 followed by a synchronizing instruction M_4, would result in each node j being granted the following initial schedule:

$$\mathcal{L}_j^0 = [L_0^j, L_1^j, L_2^j, L_3^j, M_{4_j}]$$

It is important to note again that each of the responsibilities L_i^j is a local computation, and is not part of any DP operation. This means that the predicate function **complete** always returns true for a responsibility L_i^j at any time after it has been applied. The responsibilities M_{i_j} are, however, components of a DP operation; thus the predicate **complete** returns true only after all responsibilities of M_i have been applied. This has the desired effect that nodes in the distributed execution may independently apply (and discard) any of the responsibilities implementing a non-synchronizing instruction. When a responsibility implementing a synchronizing instruction is reached, however, all nodes must complete the instruction before any are allowed to discard the responsibility and continue with the execution.

4.2 Reasoning About Performance for Abstract Model Implementations

The principal goal of the process of modelling undertaken in the preceding chapter was the design of an execution environment whose characteristics were conducive to the evaluation of irregular NDP codes. To date we have used our mathematical framework to deduce two models — the STSP and the MTP — which are potential solutions to this problem, by virtue of their abilities to guarantee deadlock-free evaluation even in the presence of arbitrarily complex distributions of computation (such as those which might be required to optimize an irregular execution). However, as they stand, the formalisms we have defined incorporate no mechanism for reasoning about the practical utility of our candidate solutions, particularly the degree of performance one could expect from an implementation of each of the models on a real-world machine. Thus, we have no basis for deciding which of the execution models — the STSP or MTP — is the better basis for implementing irregular NDP.

In this section we offer a series of evaluation criteria which highlight aspects central to actual performance in implementations of our abstract models. We make use of the descriptions of SIMD and SPMD hardware (also formal definitions within the single framework of our mathematical analysis) from the preceding section, noting that the actual performance obtained by abstract model X executing on hardware of type Y can be considered in terms of the cost of mapping the formalisms of X onto Y. The analysis which follows considers the issues involved in such mappings of the STSP and MTP on a target machine, identifying the key areas of functionality that must be addressed in an actual implementation. We focus particular attention on how well we would expect implementations of the different paradigms to perform in the presence of irregularities of the types identified in Section 2.4.

As a particular case we consider the mapping of both the STSP and MTP environments for NDP onto a general SPMD machine (with characteristics identical to our description for the CM-5 in Section 4.1.2). We choose this type of architecture because of it represents a broad class of parallel architectures currently in use for high performance scientific computing. Consideration of the costs of providing the various key functionalities for each of the models on such a machine gives us an impression of the degree of performance that each model of NDP execution could obtain on a real-world SPMD architecture. Particularly we can determine how such costs would be affected by program irregularitities. Based upon such factors we can make an estimation regarding which candidate model is likely to provide the best foundation for high-performance irregular execution on a SPMD machine.

4.2.1 Principal Cost Factors

If we inspect the mathematical forms of the distributed execution (Definition 3.14) and the simple synchronizing DP execution (Definition 3.17) it is clear that there is certain functionality assumed of the environment in which the execution is taking place. On a basic level there is an assumption of a control flow on each of the nodes of the machine, and some kind of mechanism for applying responsibilities. On a higher level there are a number of model-specific functions assumed of the environment. We can classify these latter demands into three broad categories:

1. The nodal scheduling of responsibilities;
2. The maintenance of global knowledge;
3. The satisfaction of each responsibility's associations.

The first of these categories embodies both the maintenance of the nodal local schedule structure and the possible administration of multiple threads of control (should the applied set ever be permitted to grow beyond a singleton). The STSP places greater demands on the aspect of local schedule structure, since it requires that the local schedule data structure remain sorted (in the sense of Definition 3.25) at all times, even in situations of dynamic update. Conversely, while the MTP assumes little of its local schedule it relies heavily on a node's ability to deal with multiple active threads of control. We can thus define the cost of implementing the schedule-related functionality of an abstract model as consisting of a cost involved in management of the local schedule plus a cost involved in supporting the threadedness of the model. The magnitudes of each of these costs depends on the degree of support offered to each operation by the target hardware.

The second category we identify, the maintenance of global knowledge, derives from the necessity to implement the predicate **complete**, which determines whether all responsibilities of a DP operation have yet completed. It represents a vehicle for synchronizing the co-operative participants of the operation just prior to the completion of the operation (a fundamental requirement for DP style of execution). The style of interaction (as defined in the predicate and its use within the algorithmic description of distributed execution) is reminiscent of classical *barrier* [63, 7, 14] synchronization. Thus we can consider the cost of maintaining global knowledge in an implementation of an abstract model as being equivalent to the cost of implementing a barrier synchronization between a given set of nodes (those co-operatively computing a DP operator). Again, the precise magnitude of this cost is governed by the degree to which such operations are optimized in the target environment.

Our final identified functionality involves the satisfaction of the close and loose associations of each responsibility. These are data dependencies (see Definition 3.8) which must be met before the computation represented by the responsibility can execute to completion. In an implementation of an abstract model a cost will be incurred in retrieving the data required to satisfy the associations of a responsibility. Depending on the location (within the distributed environment) of the node executing the responsibility and the memory which holds the desired data, this retrieval may involve inter-node communication. Thus the cost of satisfying an association within an implementation of an abstract model is driven by the cost of inter-node communication, the frequency with which such communication is required and the degree to which the latency of remote access is visible.

If we take into account all three of these categories of assumed functionality, we can postulate a cost metric for an implementations of an abstract model on a machine:

$$\begin{aligned}\textit{Implementation cost} = {} & \textit{(thread-support cost + schedule maintenance cost) +} \\ & \textit{barrier synchronization cost + visible association latency}\end{aligned}$$

As we have noted above, the magnitude of each of these component costs will depend heavily on the degree to which the demands of the model are compatible with the facilities offered by the machine. For example, we would expect that an implementation of the MTP targetting a machine with hardware support for multi-threading (see Section 6.3.5 for a description of several such machines) would incur a smaller *thread-support cost* than a similar implementation targetting a SIMD machine (where no explicit threading support is present). Thus to evaluate the overall cost of an implementation of model A on machine M we must estimate the various component costs in the context of A and M and sum them (with an appropriate weighting). We informally consider two such evaluations in the following section: those which describe the implementation of the STSP and MTP on a SPMD multiprocessor.

4.2.2 Implementations on a SPMD machine

As we have described in the introductory chapter of this thesis, the SPMD class of parallel architectures are commonly considered to have the most potential for high-performance execution of DP programs. This fact, coupled with the broad availability of SPMD machines in the scientific computing community, makes it appropriate that we focus special attention on the implementation of our models on this class of architecture.

The analysis we present is divided into a discussion of the relative costs incurred by virtue of the satisfaction of each of the principal areas of model-specific functionality identified in the previous section. We particularly note the degree to which additional costs are introduced by the presence of program irregularities. Following a discussion of each of the component costs, we offer a rationale for choosing one implementation over the other as a basis for high-performance irregular computation on a SPMD machine.

Basic Modelling Cost

The principal difference between the STSP and the MTP lies in their basic model of execution: the former adheres to a single-threaded form of nodal execution, while the latter assumes a multi-threaded capability of its nodes. As described in the formalism of Section 4.1.2, the nodes of a SPMD machine (such as the CM-5) are distinctly single-threaded. This suggests that the thread-support cost involved in the implementation of the STSP on such a machine would be trivial, while the same cost of a SPMD implementation of the MTP would be relatively high. The latter implementation would necessarily involve a software layer to provide multi-threading capabilities on each node; the functionality and cost of that layer would be similar to that observed in existing threading libraries such as Culler's TAM [44, 119, 31, 30], P-RISC [86, 85], Charm [64, 111, 112, 113], Pebbles [99, 100, 10, 109], and the MIT Cilk [23, 62] system. Later chapters of this thesis (Chapters 5 and 6) describe both an abstract and concrete form of the precise functionality demanded by the MTP implementation.

A second aspect of the basic cost of modelling the candidate execution paradigms is the overheads involved in management of the *local schedule* data-structure. In the MTP, no special functionality is required in this management (beyond the simple addition and removal of responsibilities) and indeed such a structure will likely already exist within the software multi-threading layer for the management of local threads (c.f., the task pool element of the nodal state defined in Section 5.1). Thus, the management costs incurred by the MTP can be assumed to be negligible. The STSP, however, places a particular demand upon its nodal `local schedule`, namely that it should always remain totally sorted (in the sense of Definition 3.25). Thus a special protocol must be instituted in the management of that implementation's scheduling structure. A number of factors complicate this process, including the possibility that a real NDP program may dynamically add responsibilities at arbitrary (unpredictable) times during the execution. Further complexity arises from the possible existence of conditionals within an NDP nest which effectively eliminate any chances of discovering static information regarding the order of dynamic additions. Factoring these issues into the protocol of schedule management, we are forced to adopt a dynamic scheme where some form of sorting operation occurs upon each addition to the `local schedule` structure. The cost associated with such a protocol must be considered to be significant on account of the frequency with which such operations would arise.

Global Knowledge Cost

The STSP and MTP place identical demands upon an implementation with regards to the spread of global knowledge. In both, the semantics of the operation can be considered to be functionally identical to a barrier synchronization between the nodes co-operating on a single DP operation. Furthermore, as described in the formalism of Section 4.1.2, many SPMD machines incorporate special global synchronization operations (including barrier synchronization) directly in hardware. Thus, on the surface, it would seem that both STSP and MTP implementations would incur only a moderate cost due to global knowledge overhead. However, it must be recalled that both the MTP and STSP models place few restrictions on the distribution of computation across the machine, allowing the likely possibility that the set of nodes we wish to engage in a barrier is a proper subset of the machine. In this instance the whole-machine synchronization embodied in the hardware of the SPMD machine does not represent the proper interaction and is thus unsuitable for use as an implementation of global knowledge spread in the model.

This presents an interesting trade-off in the implementations: we can choose either to restrict the allowed distribution of computation to ensure that each DP operation is co-operative between all nodes (and thus allow the use of hardware synchronization), or we can retain the generality of distribution and synthesize the barrier from primitive point-to-point communications primitives. The former approach minimizes the synchronization cost but reverts the execution model to a simple serialization (see Section 2.2.1 for an argument against such approaches), unable to exploit multiple parallel dimensions. Since, as we have argued previously, such an execution is unsuitable to irregular NDP computation, we must necessarily consider the second approach — the synthesis of barriers from individual communications (see Section 5.2.2 for a thread-based description of such a synthesis). It is clear that such a mode of global knowledge spread introduces a significant latency, equally high for both STSP and MTP implementations. However, some proportion of this cost can be masked by the MTP implementation's ability to execute other program threads while a given thread is blocked at a synchronization point.

Association Cost

During the execution of a responsibility within either the STSP or MTP implementation, it may be necessary to retrieve data from off-node to satisfy the close and/or loose associations of the responsibility. The distributed nature of the SPMD architecture dictates that such an operation necessitates inter-node communication and thus introduces a latency. In both implementations it may be possible to reduce this overhead by choosing judicious distributions of data and computation such that few off-node requests are needed. Such an approach is especially tractable for minimizing the cost associated with close associations, since the dependency between a responsibility and its closely associated aggregate element is statically known. We can ensure that each responsibility is distributed to the node which owns its closely associated element by limiting our model to *owner computes* distributions (Definition 3.11). However, it is typically impossible to statically deduce the dependencies represented by loose associations, thus we must always expect a certain degree of latency to be introduced in the satisfaction of such associations. As was the case with the global knowledge operations, the amount of latency suffered by the STSP and MTP implementations would be similar (given the same distributions of data and computation), however the MTP implementation has the opportunity to mask some or all of this overhead by executing other threads. This mechanism has the potential to make the MTP considerably more efficient in the execution of irregular NDP codes, since such programs will likely define a high number of loose associations for constituent responsibilities.

Irregular NDP Execution on SPMD Machines

Considering all component costs for both STSP and MTP implementations on SPMD architectures, it is clear that each has its strengths and weaknesses. While the STSP's basic single-threaded control is provided inexpensively in a SPMD implementation, the model suffers notable costs due to the presence of communications-induced and synchronization-induced latency, and also must enforce a relatively expensive protocol to ensure program sortedness. Conversely, an implementation of the MTP must provide a complete multi-threading software layer upon each node of the machine, with all the overheads such a system introduces. However, the MTP implementation can make use of inexpensive mechanisms for schedule management and also has the potential to mask some or all of the latencies introduced by synchronization and communication.

Returning to the original goal of this analysis — to choose an environment for irregular NDP execution — we can interpret these performance considerations in light of the kinds of operations we wish optimized in our implementation. Section 2.4 identifies three common forms of irregularity in NDP codes: irregular data structures, irregular access patterns and irregular control. The first of these is equally well managed by the implementations of STSP and MTP; the latter two introduce high overheads into the STSP implementation but may be handled cheaply by the MTP implementation. This difference marks the MTP as the superior environment for irregular execution: although it bears a constant overhead due to thread support, the presence of program irregularities introduces few additional costs. Thus an implementation which can minimize the threading overhead offers great potential as a environment which delivers good performance irrespective of irregularity.

Throughout the remainder of this thesis we will focus our attention on a closer investigation of MTP implementations for SPMD architectures. In the following chapter we expand upon the functional requirements of the MTP, defining an abstract machine upon which the MTP can directly be implemented. We use this design as a basis for a low-overhead implementation of threading, described in Chapter 6. Later chapters of the thesis consider an evaluation of an actual implementation of the MTP upon this environment, defining a NDP language system (Chapter 7) and demonstrating that it delivers a high degree of performance for irregular codes (Chapter 8).

Chapter 5

A Framework for Multi-threaded Expression

We have now provided a case, in terms of a detailed mathematical analysis of general NDP execution on a distributed memory machine, for considering a multi-threaded basis for such execution on a SPMD architecture. To further develop this idea into a fully-fledged execution environment useful for describing actual NDP operations, we need to elaborate the Multi-Threaded Paradigm (MTP) presented and analyzed in Chapters 3 and 4 into an appropriate medium of general and detailed multi-threaded expression. Specifically we need to define a SPMD abstract machine supporting the notion of multi-threading in a simple and potentially efficient manner. The latter requirement underlies the need we observed previously (in Section 4.2.2) for low thread management overheads to make an implementation of the MTP practically viable.

In this chapter we describe a simple thread-based SPMD abstract machine which adheres to the spirit of the execution we described earlier in mathematical form, while offering low practical overheads. Our discussion details the nodal data structures, instruction set and model of execution of the simple machine, noting how many of these concepts are directly analogous to forms present in the earlier mathematics. To demonstrate the generality and utility of our definition, we go on to describe how highly-generic partitioning-independent forms for three common DP operators (map, reduce and scan) can be expressed as programs for this abstract machine.

5.1 A Threaded Abstract Machine

We now describe a distributed memory multi-threaded multi-processor machine which promises low practical overheads: an abstract machine based upon the principles of SPMD execution and non-preemptive multi-threading. The former influence stems from the DP nature of the codes we expect our machine to ultimately execute. The latter factor is a reaction to the need for low administration cost: whereas a system in which executing threads are preempted must necessarily store the full state of machine (including registers) at points of context switch, a non-preemptive scheme need only store the (much smaller) state which the thread itself explicitly identifies as necessary for continuing its execution. This factor, coupled with simpler control requirements, means that switching costs are considerably lower for non-preemptive systems of thread scheduling.

At the top-most level of abstraction we define our machine in terms of an abstract machine *program* and a set of *processing nodes*, thus:

Let $\Omega(P, N)$ be an abstract machine.
Where $P = \{c_i\}$ is an abstract machine program,
$N = \{n_i\}$ is a set of processing nodes

The notion of an abstract machine program is defined as follows: an Ω-program consists of a collection of disjoint *threads* c_i, each a sequence of abstract machine instructions. Threads may be *activated* and passed initial state information, which will be encapsulated throughout the lifetime of the activation. Each thread activation (or simply activation) executes the instructions from its corresponding thread, in sequence, until it reaches a distinguished *terminate* instruction. The thread activation realizes the concept of responsibility (or macro-responsibility) in our mathematical model (c.f., Definitions 3.7 and 3.18) — both represent the computational building blocks which describe the unfolding of a dynamic distributed execution.

In accordance with the SPMD nature of Ω, each node n_i holds a copy of the same (global) program P, but activates threads from that program independently of other nodes. Every node commences its execution by activating the same distinguished startup thread c_0. The node's execution of the program P continues until such time as the thread c_0 terminates its activation.

At any time between the initial activation of c_0 and the completion of the program P, the computational state of a node n_i is described by a tuple of the following form:

$\eta_i(\tau) = (h_i, t_i(\tau), r_i(\tau), s_i(\tau), b_i(\tau))$

Where h_i is the unique host id of the node
$t_i = \{a_{ij}(s_1, \ldots, s_n)\}$ is a set of thread activations, each encapsulating its own state variables
$r_i = \{(k_{ij}, v_{ij})\}$ is a set of result values
s_i is a memory pool dedicated exclusively to n_i
b_i is the single element of t_i that n_i's instruction stream is currently being drawn from

The purpose of each of these state elements, and the modes in which the machine Ω may manipulate them are discussed in the Sections which follow. We can identify a correspondence between the state elements and mathematical constructs presented in our earlier analysis: t_i is equivalent to the nodal local schedule $\mathcal{L}$ (Definition 3.9), Ω's nodal memory pool s_i is the precise analogue of the nodal memory space M_i of Definition 3.2; the remaining elements of $\eta_i(\tau)$ can be considered elements of the state S_i^t in that same description.

5.1.1 Execution Model and the Task Pool

The nodal model of execution embodied by Ω differs from that traditionally associated with SPMD machines (see Section 1.1.2). A thread which has become active on a node n_i executes until such time as it explicitly requests suspension (by issuing an abstract machine instruction) or terminates. In the course of its execution the activation may request the activation of other threads, however such activations do not occur immediately (as would function activations on a conventional machine). Rather, new activations are added to the node's *task pool* t_i, a data structure from which a new "executing" activation is chosen (non-deterministically) whenever the node becomes idle. A thread may have many activations within a single task pool, each corresponding to a distinct dynamic invocation and possessing its own internal state.

While this generalized model of execution seems to lend the model somewhat of a MIMD flavour, all nodes of our abstract machine are constrained to each execute threads drawn from a *single* program, albeit with less clearly defined lock-stepped phases of execution than normally present in SPMD models. Indeed, the execution ordering of thread activations within our model is non-deterministic, yet our paradigm remains fundamentally SPMD.

Every activation within a node's activation pool t_i is in exactly one of three states: either *pending*, *active* or *suspended*. Each describes a computational situation which a thread may enter during its activation:

- A *pending* activation is one that has been entered into the pool but has yet to be selected for execution by the scheduler. Its encapsulated state is, therefore, the initial state that was passed to it when it was entered into the activation pool.

- An *active* activation is one which is presently being executed by the node. There can be at most one activation within a task pool with this status, the activation b_i.

- A *suspended* activation is one whose thread has already been partially executed, and which voluntarily suspended its execution. Upon suspension, an activation specifies a *key* (or identifier) which denotes the conditions under which its execution may continue.

The fact that Ω supports the notion of thread suspension makes it a *blocking* model of multi-threading in the common terminology of the literature (e.g., [10, 100, 99]).

5.1.2 The Result Pool

In addition to the task pool, each node maintains a *result pool* which serves as a medium for communication and synchronization between activations. Structurally each pool is a set of pairs — the first element of the tuple being an identifying key, the second bearing a value of some expressible data type. By means of abstract machine instructions (see Section 5.1.5), an activation may add a result to either the pool of the node it is presently executing on, or to that of another node. The latter action requires communication (see Section 5.1.4).

The result pool of a node plays an important role in the scheduling of activations when a node becomes idle. Activations which have voluntarily suspended on a key k may not be scheduled until one or more results with key k are present within the node's result pool. At that time, the suspended activation once again becomes *schedulable* (i.e., it is available for the scheduling algorithm to choose when the node falls idle). When such an activation is eventually chosen by the scheduler, it is said to be *resumed*: one result from r_i with key k is removed from the pool and the execution of code from the suspended activation continues from the point at which it voluntarily relinquished control. The result pool value which caused the activation to resume is available for use within this continuing computation.

It is clear that under this scheduling paradigm, the matching of suspension keys with result pool keys plays a role in defining the paths of control and data flow in the program. We assume that abstract machine programs maintain a protocol of key management which guarantees that its allocation of keys from an infinite "key space" is never such that a result entered into a node's result pool with key k and intended for a given thread activation T may erroneously be *intercepted* by a different activation also suspended on k.

5.1.3 Data Types, Aggregates and Partitioning

The machine Ω supports all common basic data types (numbers, characters, etc.) but only provides a single aggregate constructor to group values of these types — the one dimensional (nestable) vector (as described in Definition 3.1). All vectors have a lower bound of 0 and an upper bound of $len - 1$. An important property of the abstract machine is that each of its vector values is a *partitioned aggregate* (c.f. Definition 3.5) — that is, its indices are spread across some or all of the nodes of the machine according to a *partitioning function*. The partitioning function for a vector takes the index and aggregate length as arguments and returns the identity of the node which owns that index.

The individual values which a node n_i is assigned by such partitioning functions reside within that node's memory pool s_i (which we assume to be unbounded).

To allow a program to easily refer to distributed vectors, Ω provides a *vector descriptor* for every vector value. This meta-data object is a three-tuple (id, pf, len), in which id is an identifier which uniquely distinguishes the vector, pf is the partitioning function used to distribute the vector, and len is the length of the vector.

5.1.4 Communication

As alluded to previously (and described in Section 5.1.5), the language of the machine Ω includes *communication instructions*, which may alter the state of a remote node.

The semantics of the inter-node communication is as follows: the node executing the instruction sends a communication event (or *message*) to node n_j, the target of the communication instruction. The execution of that node's *active* activation b_j is momentarily suspended, while the state alteration specified by the instruction is carried out. On the completion of this state modification of n_j, the activation b_j continues from the point at which it left off when it was interrupted. From the point of view of the activation b_j, the state of the node has been seamlessly altered behind the scenes.

In such a fashion, the communications instructions of Ω have the capacity to add entries into a remote node's task pool or result pool. In the former case, the instruction is obliged to provide an initial state for the newly activated thread, while in the latter case, a key must be provided to tag the result value.

5.1.5 Abstract Machine Instruction Set

The abstract machine Ω is imperative, with each node acting as an independent sequential computational element. In addition to the standard operation of assignment, the language of the machine supports traditional algorithmic control constructs (for, while, if) and a number of special instructions which deal with the management of thread activations and communications between nodes. This section provides a semi-formal specification of the semantics of these operations.

As notational conveniences, we adopt the following conventions in expressing the semantics of Ω-instructions and in describing Ω-programs:

- we use a *dot* notation to refer to elements of a *vector descriptor*. Thus, if v describes a vector within an Ω-program, $v.id$ denotes the vector's identity, $v.pf$ denotes the partitioning function of the vector and $v.len$ denotes its length.

- we use a *square bracket* notation to refer to individual indices of a vector. Thus $v[10]$ denotes the eleventh element of the aggregate v. We use this notation to refer to both locations (L-values) and values (R-values). It is an error for an activation executing on a node n_i to refer to an aggregate element not stored on that node (i.e., not present within s_i).

- we use the symbol M to refer, within a thread activation, to the identity of the host executing that activation. Thus, if a thread whose definition contains the assignment $x := M$ is activated on nodes 0 and 1, the first activation will assign $x := 0$ and the second $x := 1$.

We now describe the semantics of the eight special instructions of Ω, offering an informal and semi-formal (symbolic) definition for each. Note that within the semi-formal definitions of operations which manipulate the thread pool, we annotate the thread activations under consideration with their current state and their state variables.

enter activation (t: thread, $s_1, \ldots, s_n$: state vars)
An activation for the thread t is entered into M's activation pool with state initialized by the variables $s_1, \ldots, s_n$
Symbolically: $t_M := t_M \cup \{t\}$.

request (h: host id, t: thread, $s_1, \ldots, s_n$: state vars)
Communication occurs between nodes M and h causing an activation for t to be entered into h's pool with initial state $s_1, \ldots, s_n$
Symbolically: $t_h := t_h \cup \{t[\text{pending}, s_1, \ldots, s_n]\}$.

enter value (k: key, v: value)
Deposit the value v into M's result pool and tag it with key k
Symbolically: $r_M := r_M \cup \{(v, k)\}$.

result (h: host id, k: key, v: value)
Communicate the value v from node M to node h and deposit it in the receiving node's result pool. This result pool entry is tagged with key k
Symbolically: $r_j := r_j \cup \{(v, k)\}$.

suspend activation (k: key) $\rightarrow$ result value
Alters the executing activation's state to "suspended". As explained earlier, this causes the node to schedule a new activation to execute
Symbolically: $t_M := (t_M \backslash \{t[\text{executing}, s_1, \ldots, s_n]\}) \cup \{t[\text{suspended}, s_1, \ldots, s_n]\}$.

terminate activation ()
Immediately remove the executing activation from the node's activation pool and stop executing it. This causes a new activation to be scheduled
Symbolically: $t_M := t_M \backslash \{t[\text{executing}, s_1, \ldots, s_n]\}$.

allocate storage (v: **vector id,** n: **integer)**

This instruction causes a new memory element to be allocated in the node's memory pool s_i. The new entry is tagged with the id of the vector it forms part of, and the index into that vector.

Symbolically: $s_M := s_M \cup (v, n, \phi)$.

new id () → identifier

Causes an identifier to be created and returned. It is guaranteed that the identifier is unique across the entire machine.

5.1.6 Some Useful Set Operations

The following set functions are not part of the language of the abstract machine, but are short hand notations used in the Ω-programs presented in the next section. The operations we describe are define relationships between host ids within a set; most define some ordering of the set which we denote by the sequence $(A_j)_0^{|s|-1}$.

In the thread descriptions which follow, these set operations are used to mathematically model protocols of communication between nodes (e.g., specifying that every node in a set should send a value to the node whose id is next largest in the set). One particular operation, γ is useful for defining protocols for spreading information from a set of source nodes (each of which know a specified value) to a set of destination nodes (each of which must receive this value from) using communications which traverse the shortest possible path through the machine.

pos_in_ordered (s: **set of host ids,** h: **host id) → integer**

The sequence $(A_j)_0^{|s|-1}$ is searched for the entry h and the index A_h of h in the sequence is returned. If $h \notin s$ a distinguished error value is returned.

Examples: pos_in_ordered ({2,8,3,12,4,0}, 2) = 1
pos_in_ordered ({2,8,3,12,4,0}, 8) = 4

index_ordered (s: **set of host ids,** n: **integer) → host id**

The $n + 1$st element of the sequence $(A_j)_0^{|s|-1}$ is returned.

Examples: index_ordered ({2,8,3,12,4,0}, 1) = 2
index_ordered ({2,8,3,12,4,0}, 4) = 8

prev_in_ordered_cyc (s:**set of host ids,** h: **host id) → host id**
succ_in_ordered_cyc (s:**set of host ids,** h: **host id) → host id**

The sequence $(A_j)_1^{|s|}$ is searched for element h – if it is not found, an error occurs. Assuming h is an element of the sequence, let its position in the sequence A be α. That is $A_\alpha = h$. **prev_in_ordered_cyc** returns the $\alpha - 1$ element of A if that exists. In the case that $\alpha = 1$ the operation returns $A_{|s|}$, that is the last element of the sequence. The semantics of **succ_in_ordered_cyc** are analogous. The operation returns $A_{\alpha+1}$ unless $\alpha = |s|$ in which case it returns A_1.

Examples: prev_in_ordered_cyc ({2,8,3,12,4,0}, 8) = 4
prev_in_ordered_cyc ({2,8,3,12,4,0}, 0) = 12

γ (**h: host id, d: set of host ids, s: set of host ids) $\rightarrow$ set of host ids**

For the purposes of this complex set operation, we need to define a metric on host ids (i.e., a way of telling how *close* two host id's are). We define $\mathcal{M}$ to be such a metric. Furthermore we consider only metrics with the property that $\mathcal{M}$ *partitions* the set of host ids. That is, if we choose n host ids, $h_1, \ldots, h_n$, from the set of all such ids H, we can partition the entire space of host id's into n disjoint sets using $\mathcal{M}$. We do this by defining the partition p_i as comprising $\{h_i\} \cup \{x \in H \mid \mathcal{M}(h_i, x) < \mathcal{M}(h_j, x) \forall j \neq i\}$. That is, the set of all host id's closer to the id h_i than they are to any other of the distinguished host ids h_j. Since $\mathcal{M}$ enforces that no id can be equidistant from two of the distinguished ids, every non-distinguished host id must fall into exactly one such set.

The complex set operation γ is a partitioning operator which uses $\mathcal{M}$. Two preconditions to its correctness exist: firstly that the identifier h that appears as its first argument is a member of the set of ids s. Secondly, the two set arguments to γ, s and d must be disjoint.

With these preconditions satisfied, the operator γ returns the subset of d whose elements are each closer under $\mathcal{M}$ to the distinguished id h than they are to any element of $s \backslash \{h\}$. Symbolically:

$$\gamma(h, d, s) = \{x \in d \mid \mathcal{M}(h, x) < \mathcal{M}(j, x) \forall j \in s \backslash \{h\}\}$$

To illustrate the semantics of γ in a practical context, consider the example in Figure 15. The situation we wish to model is the spread of a value common to a set of nodes $\{0, 1\}$ to a set of nodes $\{2, 3\}$. While any of the nodes can communicate the given value to any other, some nodes are "closer" than others — as defined by some metric $\mathcal{M}$. This notion of inter-node "distance" governs the cost of each individual communication. In defining a communication protocol we wish to minimize the total distance traversed by our messages; for this we use the operation γ.

$$\mathcal{M}_0(h_1, h_2) = h_2 - h_1$$

(a)

		h_1			
		0	1	2	3
	0	0	-1	-2	-3
h_2	1	1	0	-1	-2
	2	2	1	0	-1
	3	3	2	1	0

(b)

$$\begin{aligned}
\gamma(0, \{2,3\}, \{0,1\}) &= \{x \in \{2,3\} \mid \mathcal{M}_0(0,x) < \mathcal{M}_0(j,x)\} \, \forall j \in \{0,1\} \backslash \{0\} \\
&= \{x \in \{2,3\} \mid \mathcal{M}_0(0,x) < \mathcal{M}_0(1,x)\} \\
&= \phi \\
\gamma(1, \{2,3\}, \{0,1\}) &= \{x \in \{2,3\} \mid \mathcal{M}_0(1,x) < \mathcal{M}_0(j,x)\} \, \forall j \in \{0,1\} \backslash \{1\} \\
&= \{x \in \{2,3\} \mid \mathcal{M}_0(1,x) < \mathcal{M}_0(0,x)\} \\
&= \{2,3\}
\end{aligned}$$

(c)

Figure 15. Using the Set Operation γ

Part (a) of the figure defines a simple metric for determining the distance between nodes of our abstract machine (obtained by simply subtracting the sender's id from the receivers). The table in Figure 15(b) evaluates the distance between every possible pair of senders and receivers using the metric. Inspecting this table we see that the values which appear within a column are distinct, as are values appearing within a row. This means that $\mathcal{M}$ is a partitioning metric — that is, no node is ever equally distant from two other nodes under $\mathcal{M}_0$.

Figure 15(c) shows two applications of γ which determine a minimal communication protocol. We firstly evaluate the expression $\gamma(0, \{2,3\}, \{0,1\})$ which can be interpreted as an operation to find the subset of target ids (i.e., $\{2,3\}$) which are each closer to node 0 than to any other sending node. This subset represents the targets which should receive their copy of the value to be spread from node 0 (since receiving it from any other potential source would involve a longer distance communication). By applying the mathematical definition of γ we see that under metric $\mathcal{M}_0$ this evaluation produces the empty set, indicating that in our spread protocol node 0 should not perform any of the communications. Conversely, our second application of γ determines which of the target nodes should be supplied values from node 1 — this evaluation dictates that node 1 should satisfy both nodes 2 and 3. Together these evaluations of γ define a complete minimal distance protocol (assuming metric $\mathcal{M}_0$) for the desired communication. In the thread descriptions provided in the next section, no particular metric is assumed: the semantics of the operations are parametric upon the measure of distance.

5.1.7 An Example Ω-Execution

Figure 16 shows a simple program for the abstract machine Ω. The purpose of the program is to enter two thread activations on each node of the machine, and ultimately have each report a greeting message to the user. Of the two threads entered into each node's task pool, one is given an activation id 0 and the other an activation id 1. Thread activations which have the 0 id are permitted to report to the user immediately: those with id 1 must wait for a message to be received from the activation 0 of the node whose node identifier is one less in a cyclic ordering of node ids.

Recall that in this thread description, as in those which follow in the next section, the symbol M is used to denote the id of the node executing the thread activation (see Section 5.1.5).

We consider the execution of this simple program on an instance of the abstract machine Ω with three processing nodes n_0, n_1 and n_2. Figures 17 and 18 give snapshots of each node's task pool and result pool at various times during the execution.

Snapshot 1 shows the abstract machine immediately upon startup: each of the nodes has a single activation of the thread **C0** in its task pool, and each result pool is empty. Each **C0** activation is marked with an A to indicate that it is active. Once each node begins its execution of the instructions from **C0**, it is called upon to enter two activations for the **WORKER** thread, one with activation id 0, one with id 1. The state of the nodes after these additions is shown in **Snapshot 2**. Note that we are using the notation **Worker(0)** to denote the worker activation with id

```
1   C0 ()
2   {
3      enter activation (WORKER, 0, k1)
4      enter activation (WORKER, 1, k1)
5      done : null := suspend activation(k1)
6      done : null := suspend activation(k1)
7      terminate activation ()
8   }
9   WORKER (actid: integer, k: key)
10  {
11     if actid = 0
12        say "hello from thread actid on node" M
13        result (succ_in_ordered_cyc (M), k2, 0)
14        enter value (k, 0)
15     else
16        ready : null := suspend activation(k2)
17        say "hello from thread actid on node" M
18        enter value (k, 0)
19     terminate activation ()
20  }
```

Figure 16. A Simple Abstract Machine Program — hello-multi-threaded-world

0 and **Worker(1)** for the activation with id 1. It is important that these two be distinguished — despite the fact that they are executing the same static code, they are distinct dynamic elements of the computation, each with an independent state. Note also that these worker activations are tagged P for pending.

Once the **C0** activations have entered the workers, they themselves suspend at line 5 of the source program, waiting for a result with key k_1 to appear within the local pool. This causes each node to fall idle, occasioning action by the scheduler to pick a new activation to which processor attention may be focussed. **Snapshot 3** shows the nodes after this scheduling decision has been made. We can see that nodes 0 and 2 have each chosen to make their **Worker(0)** elements active (they have been tagged A), while node 1 has chosen **Worker(1)** — each is a valid choice and the semantics of Ω make no guarantees of a deterministic selection. The worker activations on nodes 0 and 1, with $id = 0$, pass through lines 12, 13 and 14 of the source, writing a greeting

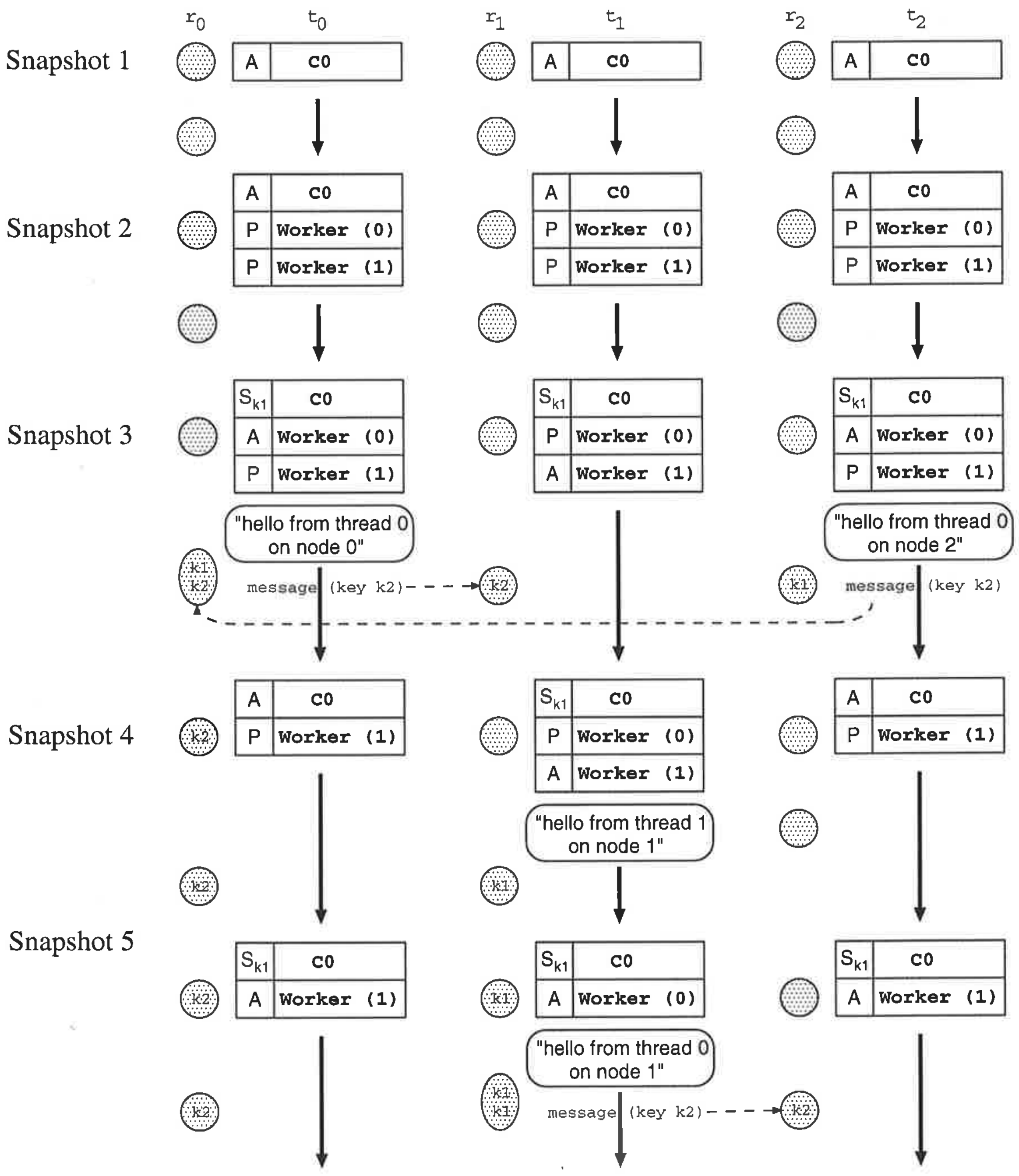


Figure 17. Executing the Abstract Machine Program, Snapshots 1–5

to the user, communicating with the next (cyclic) a message with key k_2, entering a value into the local result pool with key k_1, and finally terminating. This leads to the nodes falling idle. Conversely, the activation on node 1 reaches line 16 of the source, which is a thread suspension instruction on key k_2. Thus this node, too, becomes idle and needs a newly scheduled activation.

In performing the previous work, nodes 0 and 2 entered a value locally with key k_1, the key upon which the **C0** activation is suspended. Hence, that task pool element is a candidate for rescheduling. We see the state of the machine in **Snapshot 4** just after this rescheduling has occurred on both nodes. On node 1, we can see that the previous activity on node 0 has caused a value to appear within r_0 which bears the tag k_2. This is precisely the key upon which **Worker(1)** recently suspended. Thus, at the time we come to choose a new activation to schedule, **Worker(1)** is once again schedulable. The Snapshot shows the case where this resumption has been chosen by the scheduler.

The resumption of **C0** on nodes 0 and 2 allows both to reach line 6 of the source, at which time the activations once again suspend on k_1. On node 1, the resumption of the worker activation causes the user to be signalled, and for a value to be entered locally with key k_1 (lines 17 and 18 of the source, respectively). After this result entry, the worker activation terminates.

Snapshot 5 shows nodes 0 and 2 after their schedulers have chosen to grant processor attention to their pending **Worker(1)** activations. These two activations progress to line 16 of the source before suspending on key k_2. Node 1 also schedules a pending activation, its task pool entry for **Worker(0)**. This activation signals the user, and issues a communication instruction which causes a value with key k_2 to be added to node 2's result pool. The activation then adds a value to the local result pool with key k_1 and terminates. Note that r_1 already contained a result pool entry with key k_1 from the work performed in the previous Snapshot. In the semantics of Ω these two values are distinct despite their identical keys, so each is listed individually in the result pool.

Since both of these two entries in node 1's result pool have keys matching that upon which the node's **C0** activation suspended, then either can be used to resume that activation. **Snapshot 6** shows the case where a random element has been chosen and consumed in such a resumption. This allows node 1's C0 activation to reach line 6 before once again suspending. While this is taking place, the remaining nodes are

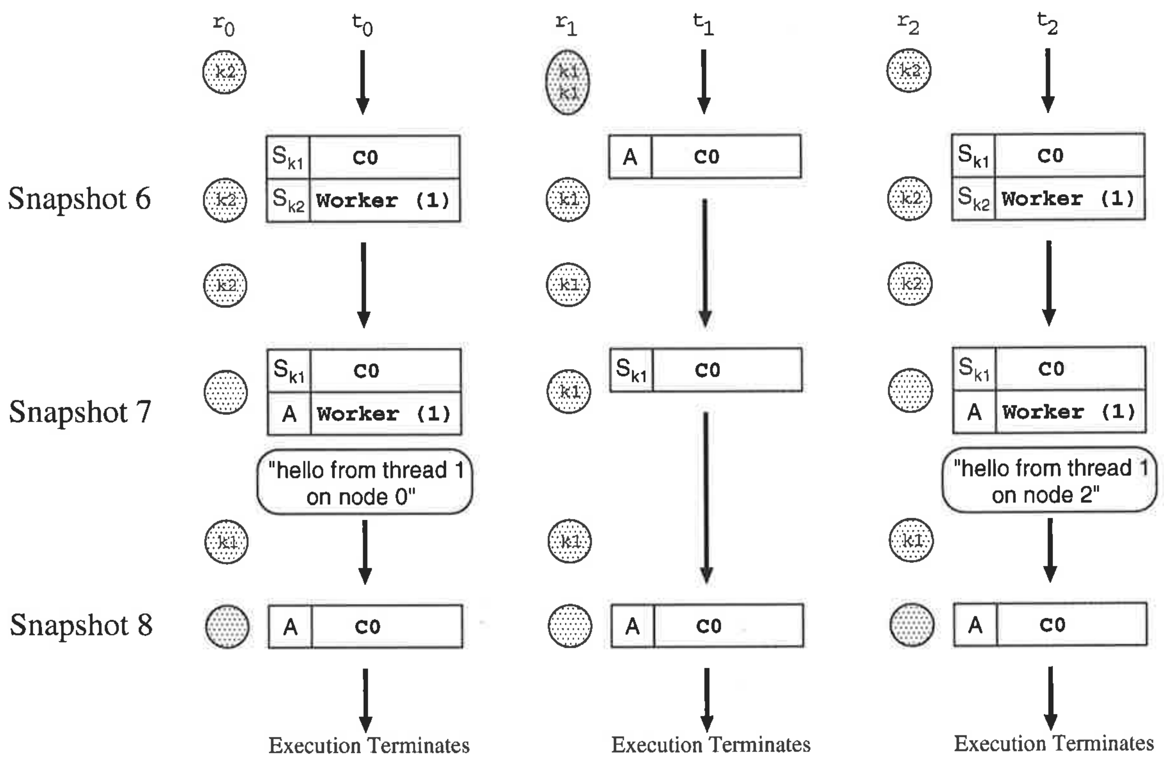


Figure 18. Executing the Abstract Machine Program, Snapshots 6–8

still in the process of choosing a new activation to schedule (and are hence idle). In **Snapshot 7**, these two nodes both resume their activations of **Worker(1)**, possible because each node's result pool contains a value with key k_2. Both signal the user and add k_1-tagged entries to their local pool.

The final Snapshot shows the **C0** activations on each node being resumed from the suspension at line 6. These resumptions can take place due to the presence of a k_1 element in each result pool. As soon as this occurs, the nodes execute the next instruction, **terminate activation**. Thus all nodes complete their execution of the program.

5.2 Threaded Specification of Data-Parallel Operations

In the preceding discussion we have motivated a general model for the execution of NDP within a paradigm of non-blocking multi-threading. To this end we defined a semantic framework for threaded definitions in the form of the abstract machine Ω. With this ground work performed, we may now make a detailed specification of a generalized model of NDP.

We begin by defining a thread-based form for the elemental sub-computations which a DP operation may apply in parallel across indices of a vector; i.e., its responsibilities. In the most general expression of DP, these elements may adopt any computational form which returns a value (to be used in constructing the result of the DP operation). Thus we may think of a responsibility as being represented by a computational function. To promote parallelism, however, we consider only referentially transparent functions (i.e., those which do not have side-effects).

We model function activation in terms of the primitives offered for thread activation. This may be accomplished by placing the Ω-instructions for each DP elemental computation within a unique thread and adhering to the following calling convention.

To "call a function F" (i.e., to activate the elemental computation of a DP operation):

1. enter an activation for F's thread into the local task pool passing a key k_0 as an argument of initial state,

2. suspend activation of the caller on k_0.

The thread form of F models function return (passing the *result* value) by including the following as its last two instructions:

$$\text{enter value } (k_0, result)$$
$$\text{terminate activation } ()$$

Since our model expresses NDP, we must provide for the situation where the elemental computation of a DP operation is itself a DP operation. That is, we should adopt the previously mentioned protocol (division into a separate thread plus calling conventions) in our Ω definitions of DP operations.

We choose to construct such *Data-Parallel Thread* forms in a highly generic fashion, partly to demonstrate the generality of the execution model and partly to promote re-usability of our definitions. Rather than make a definition of a particular instance of a DP operation, we choose to capture the semantics of the operation itself in a form generic upon the possible elemental sub-computations and data layouts with which it may be instantiated. We also make a design decision that our our implementations of DP operations should be owner-computes (c.f., Definition 3.11): this choice is made in an effort to reduce the amount of communication necessary to complete the computation.

It is to the details of constructing generic, nestable owner-computes forms for DP operations that we now turn our attention.

5.2.1 Issues in the Specification of Data-Parallel Threads

The fundamental model of execution embodied in our protocol of DP threads is relatively simple. A DP operator Ψ across a partitioned vector v of length V_n is modelled as an activation of the thread form of Ψ entered into the task pool of every node which owns a portion of v (i.e., the participant set of $\Psi(v)$). The instructions of this DP thread perform, amongst other things, the spawning on each participant node of threads implementing Ψ's elemental computation. One such activation occurs on a node for each local element of V held within that node's memory — thus, during the entire computation exactly V_n such elemental computation activations will be made across the entire machine. It is important to note that there are no limitations upon the nature of these sub-threads; they may be DP threads in themselves, thus implementing NDP.

While this model is straightforward, there are a number of considerations which govern the nature of the information we need to carry through the state of the DP thread activations. Additionally certain constraints enforce unusual modes of thread activation. We now come to a discussion of these factors.

Data Constraints

As intimated above, we model calls to a DP operator in a manner that is similar to calls to a regular function. That is, we enter an activation into the local node's activation pool and then suspend the presently executing activation until the function or DP operator has been completely executed. In terms of Ω, this is achieved by passing a key as one of the initial state variables of the newly activated elemental computation thread, and suspending upon that key. The elemental activation has the obligation to enter a result into the node's result pool once its computation is complete. The value so entered is the result of the DP computation.

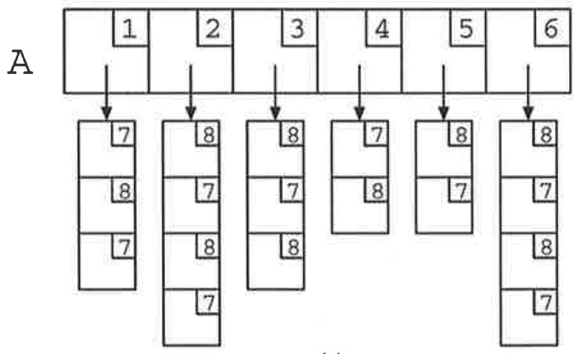
In applying this protocol to DP threads, several subtleties arise because of potential disparity between the set of nodes which compute a result (governed by the owner-computes principle), and the set which require that result (to incorporate into further computation).

Consider the example data and program shown in Figure 19. This shows a simple nesting of an "apply-to-all" DP operation `map` across a nesting of vectors `A`. The outer `map` operation specifies a parallel computation across the outer dimension of the vector, one in which the operator `map (+1)` is to be applied to each index. That is, each of the inner vectors of the nest is acted upon by a parallel operation which applies the (serial) function `(+1)` to every index.

The partitioning of `A` is denoted in the figure by the numbers in small rectangles in each index's upper right.

If we follow the model for DP threads we have defined at the beginning of Section 5.2, the computation of the nested DP operation proceeds as follows. First an activation of the thread form of `map` is entered into the task pool on nodes $1, \ldots, 6$ (the owners of the outer vector). Each of these activations, in the course of its execution, must spawn an activation for its elemental computation. Since the elemental computation is itself a DP operation (`map (+1)`), the owner computes rule suggests that such elemental activations should be made on the set of nodes owning the inner vectors. For the particular partitioning shown in the figure, this means that the six activations for `map (+1)` should each be made on nodes 7 and 8. Thus the computation of each of the inner DP operator instances is co-operative between nodes 7 and 8. However, the results of each such computation is actually required by one of the nodes $1, \ldots, 6$ as part of the calculation of the outer DP operation. We therefore have the situation that for the computation to be completed, each of the elemental activations (resident on nodes 7 and 8) must *communicate* its result back to the node which spawned it.

We term this kind of implicit data dependency a *return value dependency*. In terms of the model as it has thusfar been explained, it is impossible for the sending



(a)

```
map (map (+1)) A
```

(b)

Figure 19. Difficulties in Satisfying Data Constraints

nodes to know which foreign nodes they should send copies of the co-operative result they have just computed.

To facilitate the correct propagation of results in the case where foreign node computation is required, we make use of the concept of *participant sets* (c.f., Definition 3.6). We ensure that every activation maintains a set of host ids for the hosts which will at some time participate in the co-operative computation of which the activation is part. In the example above, the outermost `map` activation has participant set $\Pi_A = \{1, \ldots, 6\}$, while each activation of the inner `map` has participant set $\Pi_{A[i]} = \{7, 8\}$.

Participant sets provide an easy means of determining the return value dependencies of an activation. In essence, each thread activation maintains knowledge of its own participant set (the set of nodes it is currently co-operating with) and passes this set as an initial state argument to any thread activations it makes. When an activation completes its evaluation, it can compare its own participant set to that of its caller. Any nodes which are in the caller's set but not the activation's need to be sent the result of the computation.

Returning to the example in Figure 19, the system works as follows: the starting thread s_0 is, by default, executed on every node. Every node thus enters an activation for `map` passing the set of all host ids as its *caller participants* argument. The outer `map` activations first compute their own participant set $\Pi_A = \{1, \ldots, 6\}$; that is, each of the nodes 1 through 6 will perform one responsibility of the outer map. From our earlier discussion (see Section 5.2), we know that each of these responsibilities will be modelled as a local function call — that is, node 1's `map` responsibility will be a computation local to that node (i.e., will have a participant set of $\{1\}$), node 2's `map` responsibility will have participant set $\{2\}$, and so on. The singleton participant sets are computed by each of the six nodes, and passed to the thread which has been activated locally to perform the responsibility.

During the course of their execution of these thread components of the outer `map`, the nodes are called upon to spawn activations for the inner `map` which is to take place on nodes 7 and 8. During this activation process, each of the outer responsibilities passes its own (singleton) participant set as the caller participant argument to the `map` threads its spawns on the remote nodes. Each of the six `map` operations (all of which are co-operative between nodes 7 and 8) can deduce their own participant set $\Pi_{A[i]} = \{7, 8\}$.

If we consider the situation that prevails at the completion of one of these inner `map` operations, it is clear that the result — which is known to nodes 7 and 8 — must somehow be communicated back to the outer `map` responsibility which spawned the operation. The determination of the protocol of back-propagation proceeds by the comparison of the inner `map`'s participant set ($\{7, 8\}$) and the caller's participant set passed during the activation of the inner `map`. For example, if we consider the inner operation spawned by the outer `map` responsibility executed on node 2 we would be required to compare the caller set $\{7, 8\}$ with the inner participant set $\{2\}$. Determining that there exist elements in the latter (those that require the value) but not in the former (those that already know the value), we would deduce that node 2 must receive a communication from either node 7 or 8 in which the final value of this particular inner `map` was transmitted.

As we have demonstrated by way of this example NDP code, the concept of the participant set provides a convenient means of satisfying return value dependencies. Such dependencies — arising from situations where (due to the owner computes rule) the nodes seeking a result value are different from those generating that value — define a protocol of communication dependent on aggregate partitioning. The comparison of participant sets represents a low-cost mechanism for dynamically deriving such a protocol, thus allowing us to construct DP threads which are generic upon the decomposition of input aggregates.

Control Constraints

In the discussion of data constraints above, we implicitly assumed that whenever a DP thread computed a set of participants, it would necessarily be true that every node in the operators participant set would at some time enter an activation of the thread implementing the operator. We turn now to the discussion of a mechanism to guarantee that this situation always prevails.

In the trivial case that the participant set of an activation about to be spawned is a subset of the participant set of the activation currently executing, the control constraints are easily satisfied by means of traditional algorithmic control constructs (such as `if`). For example, if a co-operative computation with participant set $\{1, 2, 3\}$ wishes to spawn a sub-operation which itself is co-operative across nodes 1 and 2, the outer computation can specify this with a code section which reads:

```
if (executing on node 1 or 2)
    locally spawn part of inner operation
suspend until the inner operation is done
```

However, in cases where some or all of the nodes in the elemental activation's set are not present within the caller's set, a more general mechanism is needed (since the spawning operation has no existing thread of control on some or all of the inner operation's participants). An example of a case which requires this additional generality is the inner `map` activations in Figure 19. If we look at the outer `map` activation executing on each node $1, \ldots, 6$, we see that somehow each of these activations must cause a co-operative computation to be initiated between nodes 7 and 8.

Generalizing this situation, we can consider the invocation of a DP operator with a participant set Π_b by an outer co-operative computation with participant set Π_a. Nodes which are in Π_b but not in Π_a will require some special mechanism to arrange for thread activations to be entered into their local task pool. We call this requirement the *activation constraint* of the operator's invocation.

We satisfy such constraints by using the abstract machine's ability for a node to enter an activation into a remote node's task pool. Once a DP thread begins, it computes its own participant set and compares it to its caller's participant set. Any node which is in the former but not the latter needs to be explicitly activated to satisfy the control constraints. We define a protocol for nodes in the caller's set but not in the operator's set to send remote requests for thread activation to the appropriate nodes. In an effort to keep the remote activation responsibility spread among nodes, we define this protocol in terms of the set operation γ (see Section 5.1.6). In the example DP threads which follow, this protocol is expressed as follows: (where M is the host id of the node executing the activation, p is the activation's participant set, v is the vector being operated upon in parallel, f is the function representing the per-element computation for the operator, and k is a key useful for future synchronization):

```
$np = \{x \in H \mid v.pf(i, v.len) = x$
        for some $i \in [0 \ldots v.len - 1]\}$
$nt = np \cap \overline{p}$
if $M \in p$
    for each $\alpha \in \gamma(M, nt, p)$ request $(\alpha, \mathbf{DP\ OP}, f, v, p, k)$
```

This code fragment from a DP operator thread shows how such threads for operations evaluate their own partipant set (np), how they can compare this with the caller's set of participants (p) to determine the set of nodes (nt) upon which it will be necessary to remotely activate a copy of the DP operator thread. A protocol for deciding which of the current participants should make each of the required remote activations is specified (in the last two lines of the code fragment) in terms of the partitioning set operator γ: this is an effort to minimize the cost of communication involved (c.f., Section 5.1.6).

Note that activations that are entered by this method of remote request do not place their result in the local result pool prior to terminating: had the node in question actually wanted the value being computed, its id would have been in the *caller's participation set* argument of the DP thread, thus its participation in the computation need not have been induced by a foreign node. In terms of the example in Figure 19, this means that the activations of the inner map thread do not conclude by entering results into the pools of nodes 7 and 8. Rather they pass their results back to the calling node (as described in the data constraint discussion) and terminate.

5.2.2 Auxiliary Threads

Before presenting the thread definitions for the DP operators Map, Reduce and Scan, it is necessary to introduce a number of auxiliary threads. These threads implement important basic parallel operations which are commonly used within the description of more complex operators.

Vector Dereferencing

As mentioned in the description of the machine Ω, a node n_i may only directly access vector elements which are stored within its own memory pool s_i. In the case that a node requires a copy of a vector element stored on a foreign node, it may only do so by a split-phase fetch. That is, it must send a request to the node which owns the data, asking that a copy be forwarded. Such a protocol implemented in terms of Ω is shown in Figure 20.

```
DEREF ($v$: vector, $i$: integer, $h$: host id, $k$: key)
{
    if $v.pf(i, v.len) \neq M$
    ⌊ error
    else
    ⌊ result $(h, k, v[i])$
    terminate activation ()
}
```

Figure 20. Remote Dereference Thread

This thread first checks to see whether the vector element being requested is actually present on the node. In the instance that it is not, an error is raised, otherwise the appropriate value is copied into the result pool of the node which made the original request (i.e., **DEREF**'s h argument). Typically this split-phase fetch would appear within the instruction sequence of another thread in the following form, where v is the vector being dereferenced and i is the remote index being requested.

```
$h = v.pf(i, v.len)$
request $(h, \mathbf{DEREF}, v, i, M, k_0)$
val = suspend activation $(k_0)$
```

Synchronizing a Set of Nodes

As has been previously mentioned (Definition 3.7), the parallel semantics of DP operators dictates that they should be synchronizing operations. In terms of Ω's implementation of DP this translates to the requirement that the activation which entered the DP thread should not be able to proceed until after all nodes in its co-operative computation have finished. A relatively simple means of guaranteeing this property is for the DP thread to perform a synchronization of all nodes in the co-operative computation just before control is relinquished. The **SYNC** thread shown in Figure 21 is an expression of such a synchronization.

```
SYNC ($s$: set of host ids, $k$: key)
{
    if $|s| = 1$
        enter value $(k, \phi)$
        terminate activation ()
    $succ =$ succ_in_ordered_cyc $(s, M)$
    $rep := s$
    result $(succ, k_1, M)$
    while $rep \neq \phi$
        $t$ : host id $=$ suspend activation $(k_1)$
        $rep := rep \setminus \{t\}$
        result $(succ, k_1, t)$
    enter value $(k, \phi)$
    terminate activation ()
}
```

Figure 21. Synchronizing a Set of Nodes

The thread is passed a set of host ids; these are the identifiers of the nodes which should be synchronized. It is a precondition of the correct execution of this thread that all nodes in s at some time enter an activation of the thread with identical arguments. Until such time as this precondition is met, the synchronization barrier may not be passed. Additionally the activation which entered the **SYNC** thread should suspend upon the key k at the point in the instruction sequence which truly marks the barrier.

The operation of the thread is: firstly it checks for the trivial case that only one node has been requested to be synchronized. In cases other than this trivial one, the thread proceeds to conceptually arrange the host ids in s into a ring. Communication then proceeds between the node executing the particular activation and its cyclic successor on the ring. The message communicated is simply the host id of the originating node. Upon reception of such a message from a predecessor, an activation removes the id contained in the body of the message from the set of nodes expected to participate in the **SYNC**. The message is then retransmitted to the appropriate successor node. When finally, this set is empty, all nodes must necessarily have entered the **SYNC** and so the thread may signal the calling activation that the synchronization is complete. This it does through passing a null valued result with the appropriate key.

Note that the communications strategy used in the **SYNC** thread to implement a barrier can be improved upon. Alternate communcations schemes, based upon arranging the participant nodes in a tree-like structure, described in the literature [14] involve only $O(\log n)$ communications. Such protocols can be described in Ω, although the resulting threads are considerably more complex than that shown in Figure 21. As we shall see (in Section 6.2.2) when we come to considering the implementation of our DP threads, the issue of low-cost synchronization becomes sufficiently important that we consider alternate approaches which eliminate the need for thread-based protocols of synchronization altogether.

Allocating a Distributed Vector

The act of allocating a vector whose parts are to be distributed among a set of node memory pools, is clearly a parallel operation. Since only a node itself may make an allocation within its own pool, such a vector allocation must necessarily be a co-operative operation. We provide a thread **ALLOC_VECTOR** to perform such an operation; its Ω-thread definition is shown in Figure 22.

The arguments to the thread are the partitioning function and length of the vector to be allocated. From this information it is possible to compute the participant set of the computation np. It is a precondition to the completion of this thread that every node whose id appears in this set enters an activation for **ALLOC_VECTOR** with identical arguments. A second condition is that the activation entering the allocation request must suspend itself on the key k at the point in its instruction stream where the value for the new vector is actually required.

The thread for **ALLOC_VECTOR** may be divided into four portions (marked by bars in Figure 22). Firstly, we compute an identifier for the vector about to be allocated. This identifier must be unique across the entire machine, and be agreed on by all nodes which hold portions of the vector. To achieve a practical solution to these problems, we stipulate that the node which holds the first element actually generates the id and communicates it to all other nodes in np. Following the agreement on the new vector id, all nodes with portions of the vector set about allocating the appropriate amount of space to hold their local sections of the vector. Next we synchronize all nodes in np (since, like true DP operations, this allocation should be

```
     ALLOC_VECTOR ($pf$: partition function, $len$: integer, $k$: key)
     {
         $np = \{x \in H \mid pf(i, v.len) = x$
                 for some $i \in [0 \ldots len-1]\}$
         if $pf(0, len) = M$
             $id$ = new id ()
(1)          for each $\alpha \in np \backslash M$
                 result $(\alpha, k_1, id)$
         else
             $id$ : id = suspend activation $(k_1)$

         for $i \in [0 \ldots len-1]$
(2)          if $pf(i, len) = M$
                 allocate storage $(id, i)$

(3)      enter activation (SYNC, $np$, $k_2$)
         $dum$ : null = suspend activation $(k_2)$

(4)      enter value $(k, (id, pf, len))$
         terminate activation ()
     }
```

Figure 22. Allocating a Distributed Vector

a synchronizing operation). Finally, a vector descriptor is constructed and returned to the calling activation.

5.2.3 The Map Thread

We come now to the semantic description of a simple DP operator, the functional `map`. This operator takes as arguments a vector v of type α and a function f of input type α and return type β, and creates a new vector v' of base type β. The values within the newly constructed vector are given by the relationship $v'[i] = f(v[i]) \ \forall i \in [0 \ldots v.len-1]$. We assume referential transparency of the function f.

We note two properties of this evaluation. Firstly it is always true that $v'.len = v.len$; that is, the input and output vectors have the same length. Secondly the numerous invocations of the function f are independent and may be thus performed in parallel.

Figure 23 defines the semantics of the `map` operation in terms of the abstract machine Ω and the auxiliary threads we have previously defined. The thread takes as arguments the vector being mapped over and the function being mapped, as well as arguments denoting the set of nodes which expect copies of the result, and the key they will be expecting. The latter two arguments are present to satisfy the data and control constraints discussed previously.

```
      MAP ($f$: thread, $v$: vector, $p$: set of host ids, $k$: key)
      {
          $np = \{x \in H \mid v.pf(i, v.len) = x$
                 for some $i \in [0 \dots v.len - 1]\}$
(1)       $nt = np \cap \overline{p}$
          if $M \in p$
              for each $\alpha \in \gamma(M, nt, p)$ request $(\alpha, \mathbf{MAP}, f, v, p, k)$
          if $M \in np$
(2)           enter activation $(\mathbf{ALLOC\_VEC}, v.pf, v.len, k_1)$
              $new$ : vector = suspend activation $(k_1)$
              $c := \phi$
              for $i \in [0 \dots v.len - 1]$
(3)               if $v.pf(i, v.len) = M$
                      enter activation $(f, v[i], i, k_2, \{M\})$
                      $c := c \cup \{i\}$
              while $c \neq \phi$
(4)               $(tmp : \mathtt{val}, idx : \mathtt{index}) :=$ suspend activation $(k_2)$
                  $new[idx] := tmp$
                  $c := c \backslash \{idx\}$
          $rc = p \cap \overline{np}$
          if $M \in rc$
(5)           $new$ : vector = suspend activation $(k_3)$
          else
              for each $\alpha \in \gamma(M, rc, np)$ result $(\alpha, k_3, new)$
          if $M \in p$
              enter activation $(\mathbf{SYNC}, p, k_4)$
(6)           $dummy$ : null = suspend activation $(k_4)$
              enter value $(M, k, new)$
          terminate activation ()
      }
```

Figure 23. Threaded Specification of Map

As with the auxiliary threads presented earlier, there are certain preconditions which need to be satisfied to guarantee correct termination of the **MAP** thread.

Firstly, it must be true that every node whose id appears in the set p at some time enters an activation for **MAP** with identical arguments. A second condition is that the activation which entered the **MAP** thread must suspend upon the key k at the point in its instruction stream where the result of the `map` is actually required.

Consider the definition which appears in Figure 23 as comprising six distinct stages. We describe the purpose and function of each of these individually.

Stage One: Activation (lines 3–7)

The first operation that must be performed by the *MAP* thread is the computation of the participant set for this DP operator. Line 3 of the thread definition illustrates how this set np is evaluated by repeatedly applying the partitioning function of the vector input v to determine the set of nodes which own part of v. Once we have determined the desired participant set, we compare this with our knowledge of the set of nodes which initiated the *MAP*, to determine which nodes need to have remote activations directed towards them to satisfy the *activation constraint* (as described in Section 5.2.1). The set operator γ is used to define a protocol outlining which nodes perform which remote activations.

Stage Two: Allocating the Result Vector (lines 9–10)

The next stage of the `MAP` thread is executed only by those nodes who have been identified as participants (i.e., are in the set np). These nodes now collectively allocate a distributed vector which will (when filled with values) ultimately be returned as the result of the DP operation. The allocation is achieved by each participant node entering an activation for the auxiliary thread **ALLOC_VECTOR** (defined in Section 5.2.2). The initial state arguments passed to these activations specify that the resulting vector should have the same length as the input v and be partitioned according to the same partitioning function $v.pf$. While this second property is not necessary to the semantics of map, it serves to minimize the communications overhead needed to compute values for the new vector.

Following the activation of the **ALLOC_VECTOR** thread, each node's **MAP** activation suspends until the allocation has been performed (and the descriptor for the new vector is written to the local result pool with key k_1). After resuming execution, **MAP** defines the value *new* to carry the value of the new vector's descriptor.

Stage Three: Spawning Instances of the Function Thread (lines 11–15)

Once the new vector is allocated, the nodes in np must spawn an instance of the function thread f for every index of the input vector which is stored locally. Each of these activations represents one responsibility of the DP operator. Each activation of the **MAP** thread determines for itself the indices for which it is required to make function invocations. It does this by looping over the indices of the input vector and determining, by recourse to the vector's partitioning function, whether the index is local (i.e., allocated to node M — the node executing the activation). As an index is determined to be local, it is added to the set c of local indices (line 15) and an activation of the function thread f is entered into the local task pool (line 14). The initial state for each function activation includes the value stored at the appropriate index of v as well as the index itself. Furthermore the function thread is passed a key argument k_2 with the value it returns to the **MAP** thread should be tagged, and a participant set. Since each element of an aggregate is stored on exactly one node, each activation of f needs only involve the participation of the node which owns the associated element of v. Hence the participant set we pass is a singleton.

We place a number of expectations on the function thread f which is activated during the execution of **MAP**. Firstly, any DP operations which are activated by f should be passed f's own participant set argument (i.e., the singleton $\{M\}$). Secondly, results from the computation of the function over the aggregate index passed should be ultimately entered into the local result pool in the form of a two-tuple of the form (r, i) where r is the result value, and i is the index that was passed to the function thread. This tuple should be tagged with the key which was passed to the function thread.

Stage Four: Collecting Function Results (lines 16–19)

In the fourth phase of the **MAP** activation's execution, nodes participating in the DP operation (i.e., those in np) gather the results from the various function threads spawned in the previous stage, copying the returned values into indices of the return vector (allocated in Stage 2). The set of local indices (c) built up during the spawning process is used to determine which indices we must receive results for before this gathering process is complete.

Lines 16–19 of the **MAP** thread implement this process as a while loop predicated upon the emptiness of c. Within the loop, the thread suspends upon the key k_2 (the same key which was passed to each of the function thread activations); execution resumes when one of the function threads has placed a result in the result pool. Due to the non-deterministic scheduling policy of Ω, we cannot assume in which order the different function results will become available. However, as discussed in the previous stage, we expect the return value for the function to be structured as a tuple which includes both the function return value and the index for which the computation was being performed. Thus, as shown in lines 17 and 18 of **MAP**, we can retrieve both of these details from the result (by pattern matching) and use the result's index field (idx) to specify which slot in the result vector to fill with the returned value (tmp).

Once we have received a result corresponding to a given vector index, we remove that index from the set c and return to the head of the loop. Thus, the process of suspension, resumption and slot-filling continues until the values of each of the function threads spawned in Stage 3 have been computed and stored. At this point the computational part of the `map` operation is complete.

Stage Five: Back Propagation of Result Vector (lines 20–24)

As described in Section 5.2.1, the DP operation being performed may be called upon to distribute the result vector (i.e., its descriptor) to satisfy its return value dependency. We compute the set of nodes which were participants in the operator which spawned this `MAP` (described by the set p) and the participants of this operation (np). We assume that all nodes in the latter set already know the descriptor for the return vector, by virtue of the fact that they helped in its allocation. However, nodes in $p \cup np$, must receive the descriptor by communication. These nodes suspend (on key k_3) awaiting the communication (line 22); nodes in np are charged with contacting these nodes according to a protocol defined by the minimum distance operator γ (line 24).

Stage Six: Synchronization of Participants (lines 25–28)

The last stage in the **MAP** thread synchronizes all nodes in p prior to the operation terminating. As described in Chapters 1 and 3, this is necessary for all DP operations (in order to guarantee that no race conditions can arise in a program). We make use of the **SYNC** auxiliary thread defined in Section 5.2.2 — each node in p enters an activation for this thread passing key k_4 (line 26) and immediately suspends on this key (line 27). When the synchronization is complete, each node will receive a result marked with k_4, thus resuming the **MAP**. At this point it is safe for each node in p to enter the result of the map (tagged with the key k passed as an initial argument) into the local result pool (line 28) and terminate.

Note that, as mentioned earlier, nodes in $np\backslash p$ do not participate in this phase, since they did not request the final value of the `map`. Such nodes simply terminate.

5.2.4 The Reduce Thread

We now turn to the description of a more complex nestable DP operator, the tree-based `reduce`. This operator takes as arguments a vector v of base type α, a binary operator $\oplus$ which accepts two arguments of type α and returns a value of type α, and a starting value a of type α. The single value computed by the operator is obtained from a recursive reduction of values from v under the operator $\oplus$. Represented mathematically, this result is

$$((\ldots((a \oplus v[0]) \oplus v[1]) \oplus \ldots v[V_n - 2]) \oplus v[V_n - 1])$$

where V_n is the length of the vector v.

The opportunities for parallelism in the evaluation of this operator appear, on the surface, to be non-existent. The innermost call to $\oplus$ must necessarily occur first, the enclosing call next, and so on. To introduce scope for parallelization, we must make certain assumptions about the operator $\oplus$, that is to narrow the scope of what may be used as a reduction operator. One common limitation used in parallelizing such evaluation, is to assume associativity of the operator $\oplus$. This allows for the rearrangement of the bracketing of the reduction without altering the final value. Hence the reduction may be represented as, for example:

$$((\ldots(a \oplus v[0]) \oplus \ldots) \oplus (\ldots \oplus (v[n-2] \oplus v[n-1])\ldots))$$

This form of the reduction allows significant opportunities for parallel evaluation since it essentially defines a tree of independent applications of $\oplus$.

For the purposes of the particular reduction we define here, we make a further assumption on the operator $\oplus$, namely that it is commutative. This property allows the arbitrary reordering of values in the reduction without altering the final value. That is, the reduction may be equally well represented as, say,

$$((\ldots(v[n-2] \oplus a) \oplus \ldots) \oplus (\ldots \oplus (v[n-1] \oplus v[0]) \ldots))$$

or any other permutation of values. The reason we introduce this restriction is to allow for re-ordering to optimize the communications overhead incurred by the threaded form. Each node's role in the co-operative reduction may be divided into two distinct phases, an on-node reduction and an inter-node reduction. In the first of these, the node reduces all elements of v that it holds locally into a single result. Once this has been done, the second phase combines these nodal results into a single global result through communication. The total number of communications required in this phase is $C - 1$ where C is the number of nodes participating in the computation. This compares to as many as V_n communications (where V_n is the length of the vector) required for the parallel reduction of an arbitrarily partitioned vector under a associative but non-commutative operator. Note that the expression of this latter operation (non-commutative reduction) is also possible under Ω by adopting a technique similar to that described below for the **SCAN** thread (see the discussion on page 142). For the present discussion, however, we shall consider only the commutative case.

Our semantic definition for the parallel reduction operator is shown in Figure 24. As with the **MAP** thread presented previously, we define two preconditions to successful completion of the reduction thread. Firstly, every node in the set argument p must enter the reduction thread with identical parameters. Secondly, the activation entering the **REDUCE** should suspend on the passed key k at the point in its instruction sequence where the value of the reduction is actually needed.

As with our previous DP thread, the definition we present for **REDUCE** may be logically decomposed into six distinct phases (marked by marginal bars in the figure). We describe each stage in turn.

Stage One: Activation

In the first, we compute the participant set np of the reduction and dispatch remote requests. This is performed in a manner identical to the analogous phase of the **MAP** definition.

```
     **REDUCE** ($f$: thread, $v$: vector, $a$: value, $p$: set of host ids, $k$: key)
     {
         $np = \{x \in H \mid v.pf(i, v.len) = x$
                 for some $i \in [0 \ldots v.len - 1]\}$
(1)      $nt = np \cap \overline{p}$
         if $M \in p$
             for each $\alpha \in \gamma(M, nt, p)$ request $(\alpha, \mathbf{REDUCE}, f, v, a, p, k)$
         if $M \in np$
(2)          $myseq =$ pos_in_ordered $(np, M)$
             $i := 0$
             if $myseq = 0$
                 $val := a$
             else
(3)              while $v.pf(i, v.len) \neq M$
                     $i := i + 1$
                 $val := v[i]$
                 $i := i + 1$
             for $x := i \ldots (v.len - 1)$
(4)              if $v.pf(x, v.len) = M$
                     enter activation $(f, val, v[x], k_1, \{M\})$
                     $val$ : val := suspend activation$(k_1)$
             for $j := 1 \ldots \lceil \log_2(|np|) \rceil + 1$
                 if $(myseq \bmod 2^j) = 0$
                     $s =$ index_ordered $(np, myseq + 2^{j-1})$
                     if $s \in H$
(5)                      $t$ : val = suspend activation $(k_2)$
                         enter activation $(f, val, t, k_3, \{M\})$
                         $val$ : val := suspend activation $(k_3)$
                 else if $(myseq \bmod 2^j) = 2^{j-1}$
                     $d =$ index_ordered $(np, myseq - 2^{j-1})$
                     if $d \in H$ result $(d, k_2, val)$
             if $myseq = 0$
(6)              for each $i \in p$ result $(i, k, val)$
         terminate activation ()
     }
```

Figure 24. Threaded Specification of Reduce

The remainder of the reduction thread is only executed by those activations which own part of the vector in question. That is, only by nodes in np. Nodes which activated **REDUCE** because they desired the final value, but do not participate in the reduction (nodes in $p\backslash np$) simply terminate their activations. The final value will be communicated to them by a node in np once the computation has completed.

Stage Two: Determining a Sequence Number

The second portion of the **REDUCE** thread involves each activation deciding upon a *sequence number*. This number determines the role of the activation in the global operation. Sequence numbers are computed by ordering the set np into a sequence and determining where the local node's id appears within this sequence. An activation is granted sequence number 0 if the local node's id occurs first in the sequence, the activation whose local node id appears next would be granted sequence number 1, and so on.

Stage Three: Determining an Initial Value

The local stage of the reduction begins with each activation deciding upon an initial value for its accumulator variable val. It is in this variable that the sub-results of the reduction will be progressively stored. The protocol for deciding upon a starting value for this variable is as follows: the activation with sequence number 0 copies the a activation argument, while all others find their lowest local index of v and copy its value into val.

Stage Four: Reduction of Local Indices

The local reduction then proceeds by continuously searching for locally stored elements of v which have yet to be used in the computation. As each of these is found, the binary function thread f is activated and passed two values: the current val and the particular local element of v that has been discovered. The main **REDUCE** thread suspends until f has computed the combination of the two arguments. This new local sub-result is written to val. This process continues until all local elements have been combined, at which time val holds the reduction of every element of v stored on the particular node.

Stage Five: Global Combination of Nodal Reductions

With each of the activations of **REDUCE** having computed the local reduction of their values, it is then necessary to globally combine these sub-results into the final result. We do this through a tree-like scheme of communications as motivated earlier: the specific protocol, first described by Hillis and Steele [52], is implemented as $\lceil log_2(|np|)\rceil + 1$ passes through a regular communication step. During the first such pass, all activations with odd sequence numbers communicate their *val* variable to the activation which has the previous sequence number. These messages, upon reception by the even sequence numbered activations, are combined with the local *val* through another activation of f. The result of this combination overwrites the even sequence numbered activation's *val* variable. After communicating their sub-results, activations with odd numbers play no further part in the global accumulation, and simply terminate. Those activations which remain executing then pass a second time through the communications step. This time, activations whose sequence number is divisible by 2 but not by 4 send their *val* to the activation with sequence number two less. This process of activations contributing their computed values and terminating continues, until finally only one activation remains active: the one with sequence number 0. The *val* variable of that activation contains the final value of the global reduction. This protocol of inter-node communication is illustrated for a reduction co-operative between seven nodes in Figure 25. The value `r` computed during the final step of this example is the final value of the reduction, i.e., $v0 \oplus v1 \oplus \ldots \oplus v6$.

Stage Six: Back-Propagation of Final Result Value

The final stage of the **REDUCE** thread satisfies the data constraints of the operation, namely that all nodes with id in p should be notified of the final result. This condition is met by the activation with sequence number 0 (the only node to know the final value) communicating the result to every such node. This is shown in the last step of the example in Figure 25.

Note that it is not necessary to explicitly synchronize the nodes in p, as we were required to do for the **MAP** thread. This is because all nodes with ids in the set are implicitly synchronized by their need to receive a communication from a single distinguished node, namely the node executing the **REDUCE** activation with sequence number 0.

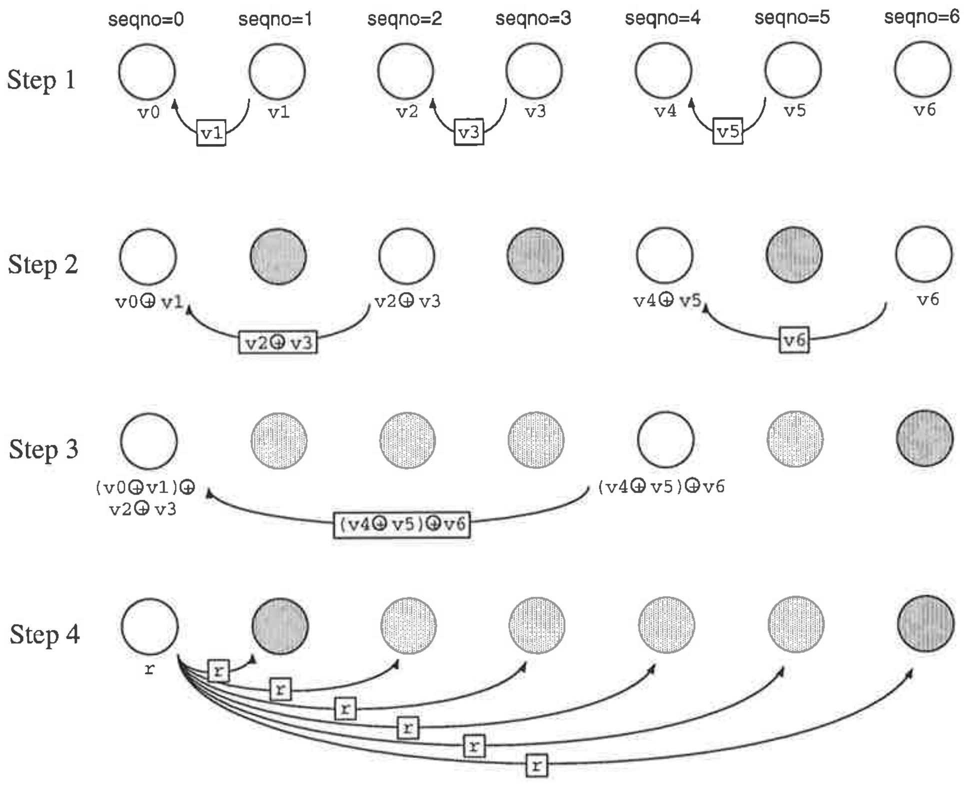


Figure 25. A Tree-Based Communication Scheme for `reduce`

5.2.5 The Scan Thread

To round out our demonstration of Ω's ability to generically model nestable DP operations, we consider a third operator: the `scan` or parallel prefix. This operator is commonly used for a variety of purposes in the literature of traditional DP programming (e.g., [17]). We consider a `scan` operator which accepts three arguments: a vector v of base type α, a binary operator $\oplus$ which accepts two arguments of type α and returns an value of type α, and a starting value of type α. From these inputs it computes a new vector v' which has the same length as v and whose indices contain "running totals" of the indices of v using $\oplus$. That is, the first index of v', $v'[0]$ holds the value $a \oplus v[0]$, the second index, $v[1]$ holds $a \oplus v[0] \oplus v[1]$, the next holds $a \oplus v[0] \oplus v[1] \oplus v[2]$ and so on.

A common means of introducing parallelism into the evaluation of such parallel prefix operations, first identified by Hillis and Steele [52], is to adopt a scheme of interleaved computation steps and tree-based communication. Figure 26 shows such a protocol, illustrated for a co-operative parallel prefix computation across four nodes. Each node of the machine begins with the value of one index of the source vector (represented by the capital letters A, B, C, D). The first node also knows the starting value of the `scan`, st. As the first communication phase, every node which has a right neighbour, informs that neighbour of the values of which it has knowledge. Once this information has been exchanged, the evaluation continues by arranging that every node with a neighbour two places right passes values seen thusfar to that neighbour. Once this set of transfers has taken place, we can see that every node has knowledge of all values needed to compute one index of the resulting parallel prefix: i.e., node 1 has enough values to compute $v'[0]$, node 2 has the values needed to compute $v'[1]$, etc. Had the vector been longer, and hence further communication was required, we would have considered neighbours 4 places distant, 8 places distant, and so on, until each node held enough information to complete its evaluation of one index of v'. In the general case, this parallel scheme for evaluating `scan` over a vector of length V_n, generates $n + 1$ inter-node transfers of information and passes through $\log n$ communication steps.

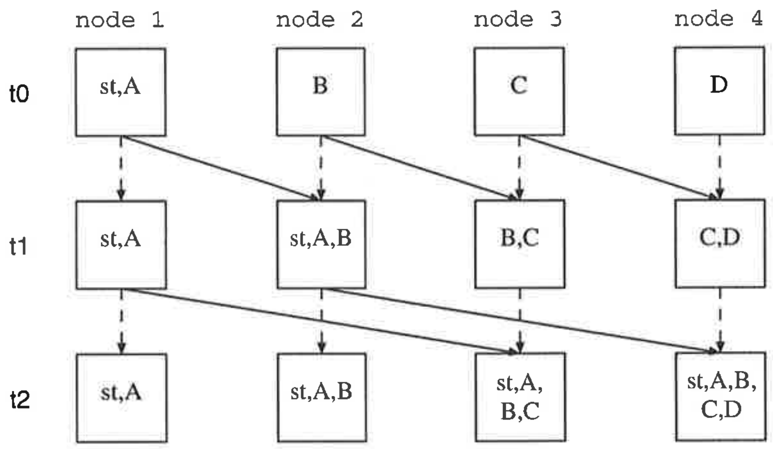


Figure 26. A Parallel Implementation of `scan`

In constructing a thread-based parallel evaluation for `scan` we adopt a similar strategy to the tree-based scheme presented above, augmenting the protocol to take into account the possibility that each node may hold more than one element of the input vector. We do this by retaining the communications structure outlined in the figure, but changing the nature of the elements which participate in the communication — rather than considering nodes interacting in a tree-like fashion, we consider an identical patten of interaction between activations of a particular (auxiliary) thread. Each such activation corresponds to one index of the input, and appears in the task pool on the node that owns that index (according to the owner-computes rule). The interactions shown in Figure 26 then become inter-activation communications, achieved by arranging for a value to be entered into the appropriate node's result pool[1]. Note that, in the case where the sending and receiving activation reside on the same node, no inter-node communication is required to model this interaction

Figures 27 and 28 show definitions for the main **SCAN** thread and an auxiliary thread which forms the elemental computation of this operator. The former of these definitions is essentially identical to the definition of **MAP** from Figure 23 except that rather that enter a user-defined elemental computation f for each local index, **SCAN** enters one activation of **SCAN_AUX** per local index.

This similarity to our earlier definition means that **SCAN** has identical preconditions to **MAP**, and operates in a similar fashion regarding dispatching of remote requests, the allocation of a result vector, suspension awaiting completion of all elemental activations, back-propagation of result values and synchronization.

The actual computation of result index values for the scan is performed by co-operation between activations of the **SCAN_AUX** thread. During the evaluation of a scan across a vector of length V_n, there will be exactly V_n such activations entered into the various task pools of the machine. Every node's pool will hold one **SCAN_AUX** activation for every local index of the input vector held locally on that node.

SCAN_AUX accepts six arguments as its initial state: the binary function f used to combine index values, v the vector over which the scan is taking place, a the initial value of the scan, and i the index of v which this activations corresponds to. Also passed is the standard participant set argument (p) and a key (k) upon which the parent **SCAN** activation suspended.

The first task of the auxiliary thread is to initialize a counter st which represents the current *step distance*, that is the fixed distance (measured in indices) between pairs of activations which will interact in the phase of the scan. Since we are adopting the tree-like system illustrated in Figure 26, our protocol for interaction between **SCAN_AUX** activations will first specify interactions between direct neighbours (i.e., step distance of 1), followed by interactions between activations two indices

[1]Since we adhered to the owner-computes principal when activating the per-index auxiliary threads, we can – at any time — use the partitioning function of the input aggregate to learn which node holds the activation corresponding to a given index.

```
SCAN (f: thread, v: vector, a: value, p: set of host ids, k: key)
{
        $np = \{x \in H \mid v.pf(i, v.len) = x$
              for some $i \in [0 \ldots v.len - 1]\}$
(1)     $nt = np \cap \overline{p}$
        if $M \in p$
          for each $\alpha \in \gamma(M, nt, p)$ request $(\alpha, \textbf{SCAN}, f, v, p, k)$
        if $M \in np$
          enter activation ($\textbf{ALLOC\_VEC}, v.pf, v.len, k_1$)
(2)       $new$ : vector = suspend activation ($k_1$)
          $c := \phi$
          for $i \in [0 \ldots v.len - 1]$
(3)         if $v.pf(i, v.len) = M$
              enter activation ($\textbf{SCAN\_AUX}, f, v, a, i, \{M\}, k_2$)
              $c := c \cup \{i\}$
          while $c \neq \phi$
(4)         ($tmp$ : val, $idx$ : index) := suspend activation ($k_2$)
            $new[idx] := tmp$
            $c := c \backslash \{idx\}$
        $rc = p \cap \overline{np}$
        if $M \in rc$
(5)       $new$ : vector = suspend activation ($k_3$)
        else
          for each $\alpha \in \gamma(M, rc, np)$ result $(\alpha, k_3, new)$
        if $M \in p$
          enter activation ($\textbf{SYNC}, p, k_4$)
(6)       $dummy$ : null = suspend activation ($k_4$)
          enter value ($M, k, new$)
        terminate activation ()
}
```

Figure 27. Threaded Specification of Scan, Part One

apart, four indices apart and so on. In the initialization we thus set up the situation as it will prevail during the first interaction: that is, we set the step distance to 1 (line 4).

The next task that must be undertaken by each instance of **SCAN_AUX** is the determination of an initial value to begin the accumulation process (in the same way as we required an initial value for each **REDUCE** activation). We use a similar approach as adopted in **REDUCE** – for the activation corresponding to index 0 of the source vector we determine the value for *working* by making a call to the binary function f (our accumulating operator) passing the starting value a of the `scan` and the value of index $v[0]$ (lines 6 and 7). For all other activations of **SCAN_AUX** we assign an initial value for *working* simply by copying the source vector index $v[i]$ upon which the activation is operating (line 9).

```
1         SCAN_AUX ($f$: thread, $v$: vector, $a$: value, $i$: integer,
2                             $p$: set of host ids, $k$: key)
3         {
4           $st := 1$
5           if $i = 0$
6             enter activation $(f, a, v[i], k_1, \{M\})$
7             $working$ :val := suspend activation $(k_1)$
8           else
9             $working := v[i]$
10          while $st < v.len$
11            if $(i + st < v.len)$
12              $h := v.pf(i + st, v.len)$
13              if $h \neq M$
14                result $(h, k_2, working)$
15              else
16                enter value $(k_2, working)$
17            if $(i - st \geq 0)$
18              $tmp$ : val = suspend activation $(k_2)$
19              enter activation $(f, tmp, working, k_3, \{M\})$
20              $working$ : val := suspend activation $(k_3)$
21            $st := st \times 2$
22          enter value $(k, working)$
23        }
```

Figure 28. Threaded Specification of Scan, Part Two

The body of **SCAN_AUX** is an iterative loop (lines 10-21) which implements the tree-like communication scheme illustrated in Figure 26. The loop predicate (line 10) specifies that iteration should continue while st, the step distance is less than the length of the vector. Within the loop there are three distinct phases of operation. Firstly, each activation is called upon to determine whether it is required to send a value during this step of the communication protocol. This decisions is made simply by determining whether there exists an activation which is st indices to the "right" (i.e., corresponds to an index of v which is st greater) of the activation being evaluated. If such an activation exists (i.e., if this activation's index of i plus the step distance st is still a valid index of v) then the present activation will be required to send a message to this right neighbour during this step (line 11). To send this message, the activation must determine upon which node of Ω the specified **SCAN_AUX** activation resides. Here we make use of the assumption that every node entered one **SCAN_AUX** activation for each of its local indices of v. Thus, we can use the partitioning function of the input vector to determine the node h which hosts the desired target activation (line 12). If the target node is the same as the node currently executing the activation (i.e., M), then the communication takes the form of an entry into the local result pool (line 14); otherwise it involves the use of one of Ω's inter-node communication operations (line 16). Either form of message is tagged with the key k_2. The appropriate node will, eventually, receive this value when it suspends on k_2 in line 18 (see below).

Once an activation has determined (and satisfied) its obligations as a source of communication in the current step of the tree-like protocol, its next task is to divine whether or not it is required to receive a message. Any activation A which has a neighbour exactly st indices to the "left" is designated to be a receiver during the current step — by virtue of the fact that this left neighbour, when executing lines 11-16 during the current step, would have identified A (which is st places to its "right") as a target for communication. We implement this requirement of message receipt by using Ω's suspension primitive: we know that the sending activation must have forwarded the value tagged with key k_2, therefore we suspend upon that key. After the message has been received (i.e., the key k_2 has appeared in the result pool, and the activation has been resumed), the value contained therein is combined with the current accumulator (*working*) by invoking the thread for the combining function f (shown in lines 19 and 20). The value returned from this function call is written into the accumulator variable.

Once an activation has carried out its commitments to send and/or receive during this step of the protocol, **SCAN_AUX** prepares for the next loop iteration by multiplying the step distance by 2 (line 21). This is a statement that the next step of the protocol will consider neighbour activations which are twice as far apart as those which interacted in this step. If this new step distance is greater than the length of the input vector then the iteration terminates — this accurately models the protocol of Hillis and Steele (pictured in Figure 26). Once the iterative loop has completed, each activation of **SCAN_AUX** has computed one partial sum of the input vector — that is, one element of the result. The activation thus completes its execution by passing this result value back to the activation of **SCAN** which initiated it, which in turn copies the value into a slot of the newly allocated result vector. Back-propagation and synchronization then ensue as described for the **MAP** thread.

It is worthwhile noting that this implementation of the DP `scan` is deterministic in result even in the case of a non-commutative binary accumulation function f. At each step, combinations of sub-results occur between activations which are a fixed (logical) number of indices apart, regardless of whether this involves communication. That is, we attempt no evaluation re-ordering to minimize communication as we did for the **REDUCE** thread (such is only possible for `scan` if we assume a partitioning). The approach we have adopted for the **SCAN** thread definition could thus be used to construct an alternative **REDUCE** definition which does not assume commutativity. Such an implementation would be more general but would exhibit greater overhead in the case where the combining function was commutative.

5.3 Summarizing the Multi-threaded Framework

In this chapter we have introduced an abstract environment for multi-threaded execution which has a close correspondence with the MTP model described mathematically in preceding chapters. Our multi-threaded abstract machine, Ω, represents a high-level framework in which parallel operations may be specified in terms of a blocking thread-based execution model. The machine is built upon principles of SPMD execution and non-preemptive scheduling with an eye towards an efficient implementation capable of expressing DP-style operations.

We have identified a number of issues that are critical to the modelling of DP operations as threads, including activation dependencies and return value dependencies. We have described simple mechanisms for expressing and resolving such dependencies, and have used these approaches to concisely model three DP operations (`map`, `reduce`, and `scan`) as threads. Our definitions of these DP threads are highly generic: the same operational description defines an execution of the operator for any elemental/combining function and also for any possibly layout of the input vector.

In the following chapter we undertake an implementation of the machine Ω on a real-world distributed memory machine, the Thinking Machines CM-5. Using this implementation we can directly code realizations of the **DEREF**, **SYNC**, **ALLOC_VECTOR**, **MAP**, **REDUCE** and **SCAN** abstract thread operations, described in Sections 5.2.2 to 5.2.5, effectively giving us a library of executable generic threads which stand as implementations of DP operations. We show in Chapter 7 that such a library can form the basis for an NDP language implementation.

Chapter 6

AMAM: An Implementation of the Model

The abstract machine Ω described in the previous chapter represents a full operational specification of an execution environment conducive to the efficient Multi-Threaded implementation of Nested Data-Parallelism. As an abstract machine, however, it is not of practical use in the construction of such an NDP system — to achieve such an end, we must construct a concrete instance of the machine Ω on a real-world Distributed Memory multicomputer. This chapter considers such an instantiation, the AMAM (Active Message Abstract Machine), on the Thinking Machines CM-5 Supercomputer [124]. Section 9.1.1 briefly considers a number of alternative instantiations of Ω which could also be readily constructed to take advantage of specific hardware support (e.g., hardware multi-threading).

In our discussion we detail the implementation of the various data- and control-structures present in our definition of Ω, as well as discussing a number of practical optimizations introduced to efficiently support NDP. We also consider how this concrete implementation of a multi-node multi-threading system bears similarities and differences to similar environments (e.g., P-RISC [86, 85], TAM [44, 119, 31, 30], Charm [64, 111, 112, 113], Pebbles [99, 100, 10, 109], Nomadic Threads [61], and the I-Structure Software Cache [74, 43] system) derived from a need to support different (but related) computational needs. Finally, we describe how thread definitions made in terms of the abstract machine Ω may be rendered as executable forms for the AMAM/CM-5. We illustrate the process by inclusion of an AMAM translation of the generic nestable Data-Parallel **MAP** described in the previous chapter.

6.1 A Multi-Threading Environment for the CM-5

Our basic strategy in constructing a concrete instance of the abstract machine Ω is to realize its various nodal data structures and control model directly in C, coupled with a simple communications library. Each of the instructions of Ω which provide functionality beyond simple algorithmic control primitives may be implemented by a C function which provides the appropriate manipulations of such structures and/or control. Thus, ultimately, our implementation of multi-threading constitutes a library of such functions with which computational threads (also represented by C functions) may be linked to form multi-threaded executables.

Figure 29 shows the nodal structure of the AMAM. Many of the entities and data structures present within this architecture are familiar from our earlier abstract model. Below we describe briefly the role of each of these entities in the AMAM system. Detailed discussion of the issues and concepts critical to their implementation and functionality appears in the sections which follow.

The Code

As in the Ω, every node in the AMAM is required to activate and execute threads from a single program. This program is, at its simplest, nothing more than a collection of AMAM threads, one distinguished as the starting (or main) thread which will initiate computation on every node.

The Heap

Each node of the AMAM is assumed to have its own memory space over which it is the sole owner. In the implementation, this is manifested as the nodal heap, from which allocations of space for activation states, storage for local vector indices, and tagged result values are made. In the CM-5 AMAM system, the per-node heap is represented by a boundary-tag pool [90].

The Task Pool

Every thread activation presently unterminated upon a node has a corresponding entry in that node's Task Pool. This data structure, central to the nodal scheduling

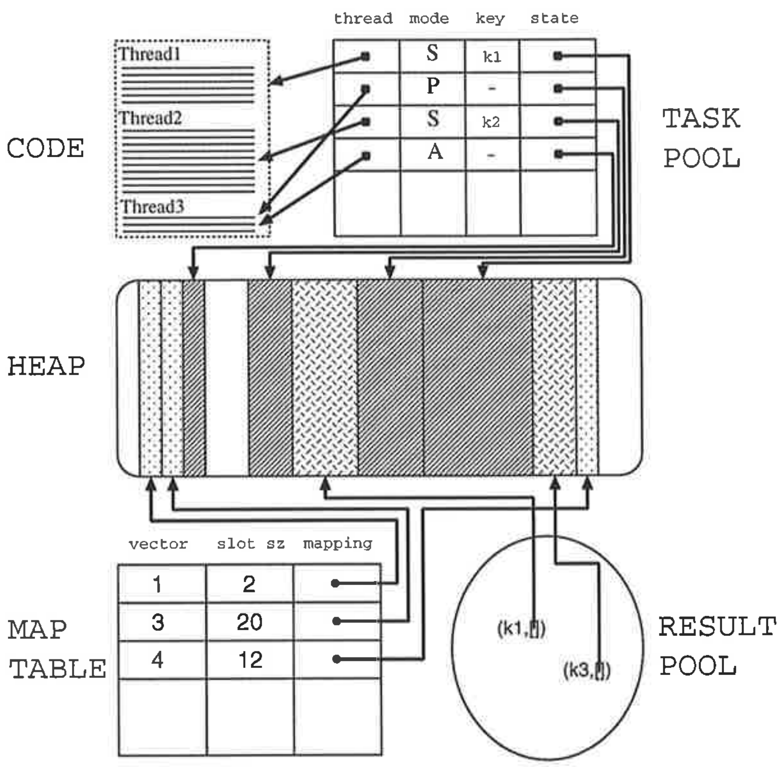


Figure 29. Nodal Structure of the AMAM

semantics (see Section 6.1.1), maintains the following information for each activation:

- a pointer to the code for the thread of which this is an activation; this is useful for passing control to the activation;
- a current mode for the activation denoting whether it is presently being executed (Active), has yet to be executed (Pending) or is only schedulable on arrival of a value (Suspended);
- if Suspended, an activation entry records an identifying *key* which describes the synchronization event which will permit resumption;
- a pointer to a block of memory in the heap which stores that activation's state.

The Mapping Table

As in Ω, the only supported aggregate is the partitioned vector. The Mapping table manages the translation between locally stored aggregate elements and memory addresses in the heap where their values are stored. Given a particular vector and an index into that vector, this table can uniquely determine the local (heap) address of the element provided that it is stored on-node. If a request is made for an off-node element, the translation process simply returns an error value. Section 6.1.2 discusses the manifestation of the partitioned aggregate in AMAM, the nodal storage model for vector indices and the efficient management of the Mapping Table.

The Result Pool

The Result Pool is the structure which provides for inter-activation communication via values tagged with keys. Unlike the Ω result pool, not every communicating value appears within the result pool: certain results, those tagged with special keys, can be directly matched to the activations waiting upon them. The semantics of such keys, their efficiency and generality is discussed in Section 6.1.4.

6.1.1 AMAM Threads and Scheduling

The heart of the AMAM library is the notion of the *thread* as the sole unit of activation. A thread is represented as a C function which adheres to a few simple rules imposed by the scheduler (see below). AMAM allows for the activation of any program thread on any node of the machine. A thread may be activated arbitrarily many times within the system.

The principal variation between the AMAM and Ω lies in the manner in which such notion of thread suspension is supported. Recall that the semantics of Ω-threads allowed for an activation to suspend at any point during its execution. We noted that, in the language of the literature, this made our abstract machine's multi-threading model *blocking*. In implementing the model upon a concrete machine, we must consider how such potential for mid-execution blocking impacts upon the complexity (and thus the overheads) present in the scheduling of thread activations. We now show that an implementation of thread scheduling directly as specified by Ω has high overheads, and proceed to develop a more efficient alternative with equivalent semantics.

Consider a thread T, represented in an executable form by a C function `T_imp`. Each activation of `T_imp` has an explicit state associated with it in the form of a set of state inputs $s1, \ldots, s_n$ which were passed by a parent thread at activation time.

Consider the case where the thread T (and hence the function `T_imp`) suspends its execution mid-way through its execution. At the point in time where an activation of T signals its desire to suspend, the system must arrange for the storage of that activation's state. When, at some later point in time, the activation is resumed we must ensure that further execution takes place within the same context. If we consider the range of information which may be accessed during this continuation it is clear that it includes the state variables $s_i, \ldots, s_n$; hence it is crucial that these be saved at suspension-time. Furthermore, if the C representation of T makes use of any local variables (e.g., for temporary storage) then it is possible for such variables to be accessed in the continuation, hence these must also be saved. Finally, it may be the case that our C code is compiled in such a way that an operation executed after resumption of the activation reads a register which it assumes to contain a value deposited prior to the suspension. Thus, the context we must save upon suspension includes not only the full set of thread state inputs and local variables, but also the full register file of the node. This potentially large amount of information must furthermore be restored at the time an activation is resumed. These necessary state manipulations would have the effect of introducing a high overhead into context-switching and scheduling operations performed within the execution model.

Given the potential costs of directly implementing Ω's notion of blocking threads, we develop an alternative approach for AMAM. Rather than allowing AMAM-thread activations to suspend mid-way in their execution, we enforce that they must run to completion. That is, we instead consider a *non-blocking* model of multi-threading. In such a model, the amount of state information which must be stored upon suspension of an activation is limited to the state inputs of the thread itself — local variables and register values need not be saved (see below). This simplification, coupled with the principle (from Ω) that the threading system should be non-preemptive to reduce switching overheads, leaves us with a very simple scheduling model. Figure 30 shows, in pseudo-code, this scheduler. The definition, effectively, consists of a single iterative loop in which the node continuously chooses one of its local activations, computes the address of the C function which represents the thread corresponding to the activation, and launches this function using C's mechanism for dereferencing pointers-to-functions. Several arguments are passed to the newly activated thread, including a unique number identifying the activation, a pointer to the associated state and (possibly) a pointer to other data items. We assume that at the end of each thread definition there is a call to a `terminate` system function which removes the activation just executed from the nodal task pool (and hence from the list of candidates for future execution) and deallocates the space reserved for the activation's state.

```
AMAM_schedule_loop ()
{
  void (*f)(state_type);

  while (running)
  {
    choose an entry t from the local task pool
    f = t.thread_pointer;
    (*f)(new_sched_id,t.state_pointer);
  }
}
```

Figure 30. A Simple Nodal Scheduler for AMAM

While this definition of an AMAM thread and the scheduling of AMAM activations is concise and devoid of the overhead described earlier, we have arrived at such simplicity by eliminating an important semantic feature of Ω's threads, namely suspension. To be useful as an implementation of our earlier design, we must provide AMAM with a mechanism for modelling such a concept. We do this by means of a simple augmentation of the scheduling model just presented, the addition of the *continuation*. Previously we assumed that every activation would make a call to a `terminate` system function immediately prior to the completion of its execution. We now provide another option: an activation may, as its last operation, instead of issuing a `terminate` make a call to a system function `suspend` passing a key and the name of a thread (the activation's *continuation*). This system call, like `terminate` removes the activation from the node, but unlike `terminate` a suspension also adds an activation for the continuation thread to the local pool. This activation receives the state from the activation which `suspended`, but is also tagged as ineligible to be scheduled until the key passed during the suspension matches one in the local result pool. The ultimate execution of the continuation activation consumes the value which made it eligible for rescheduling. Figure 31 shows a pseudo-code version of the `suspend` system call.

```
AMAM_suspend_activation (key k, thread cont)
{
  state s;

  s = state of calling activation
  remove calling activation from task pool
  locally add activation for cont with state s,
    tagged ''waiting on key k''
}
```

Figure 31. AMAM's Notion of Activation Suspension

With the existence of the **suspend** call, our scheduling algorithm becomes slightly more complex (see Figure 32). Note, however, that this model of suspension and resumption is much less costly than that for *blocking* threads. While there is still a need to save state upon a call to **suspend**, the fact that it is actually a completely new activation that will be our continuation means that we only need to copy the state variables of the activation. That is, we are not required to keep the state of local variables or registers used within the C function representing the thread. The fact that our point of resumption is now a completely new activation means that such an operation can be modelled neatly as another C function call rather than a low-level manipulation of state. In Figure 32 this is manifested in the function calls of lines 15 and 19. In both cases a pointer to an AMAM thread (a C function) is dereferenced and the resultant thread (function) activated and passed three parameters: an identifier generated by the scheduler, the state object of the activation and a result value. The situation in line 15 represents the continuation of a suspended activation, thus this third argument corresponds to the value for which the activation was waiting. Line 19 represents the scheduling of an activation which has not previously executed: there is no relevant result value, thus we pass a NULL as the third argument.

Re-expressing Ω-Programs with Non-Blocking Threads

The difference in the sense in which Ω and AMAM support the suspension of threads means that programs written for Ω may need to be transformed before they can be realized as AMAM codes. In most cases, such a conversion is easily achieved

```
AMAM_schedule_loop ()
{
  void (*f)(state_type, result_type);
  result_type r;

  while (running)
  {
    choose an entry t from the local task pool
    if (t is waiting on a key k)
     if (result pool contains k)
     {
        f = t.thread_pointer;
        r = pointer to result pool entry with key k
        remove (k,r) from result pool
        (*f)(new_sched_id,t.state_pointer,r);
     }
    else
     f = t.thread_pointer;
    (*f)(new_sched_id,t.state_pointer,NULL);
  }
}
```

Figure 32. An AMAM Scheduler Supporting Continuations

through a simple strategy of textually splitting an Ω thread into a number of AMAM-threads, using suspensions in the abstract code as the points of division. Each of the resulting AMAM-threads (except the last) would have a call to AMAM's `suspend` function as its final operation; all but the first would use the result value passed upon resumption within the expression in which the original Ω-suspend occurred. Figure 33 demonstrates the principle.

While this approach is applicable to all Ω-programs, the presence of structured control constructs surrounding a suspension instruction can lead to difficulties. Such cases typically require some re-expression of the program structure (c.f., the discussion of the AMAM `MAP` thread in Section 6.4.2).

Section 6.4 offers AMAM versions of a number of the Ω-thread definitions described in the previous Chapter.

```
Ω-THREAD(s1, s2, ..., sn)
{
  ......instructions (A)......
  s1 := suspend($k_1$)
  ......instructions (B)......
  s2 := suspend($k_2$)
  ......instructions (C)......
  terminate activation ()
}
```

$\Longrightarrow$

```
AMAM_THREAD (sched_id, state_ptr, dummy)
{
  ......instructions (A)......
  AMAM_suspend_activation ($k_1$,AMAM_CONT_1);
}
AMAM_CONT_1 (sched_id, state_ptr, result)
{
  state_pointer->s1 = result;
  ......instructions (B)......
  AMAM_suspend_activation ($k_2$,AMAM_CONT_2);
}
AMAM_CONT_2 (sched_id, state_ptr, result)
{
  state_pointer->s1 = result;
  ......instructions (C)......
  AMAM_terminate_activation ();
}
```

Figure 33. Casting an Ω-program in Terms of Non-Blocking AMAM threads

Prioritizing the AMAM Scheduler

The `AMAM_schedule_loop` defined in Figure 32 makes no attempt to minimize the overhead of key matching, that is the cost involved in the process of determining whether a given activation is reschedulable. As presented, the scheduler chooses an activation non-deterministically from the task pool and, if necessary, checks for the appropriate key in the result pool. The costs of performing this search must be considered to be non-trivial; thus it is important that some effort is made to try to minimize the wasted effort that results from a key matching failure (i.e., searching for a key prior to its entry into the result pool). Given that the times at which results become available is typically impossible to predict (the latency they embody may be unbounded), we are limited in the scope of such optimization.

However, one heuristic which offers improvement is the prioritization of the scheduler. Rather than simply choosing an activation at random from the pool as our candidate for execution, we always choose a pending activation when available, resorting to suspended activations only when left with no pending ones. The motivation underlying this heuristic is twofold: firstly, the cost of scheduling a pending activation is small; it is never necessary to examine the machine state in order to schedule such an activation. Secondly, in delaying the selection of a given suspended activation, we are increasing the probability that the attempt to match its key will succeed.

Figure 34 shows a scheduler which implements the prioritized protocol found in the AMAM. In the CM-5 implementation, there exists a special structure auxiliary to the task pool on each node, whose purpose it is to maintain a list of pending activations. Such a structure allows for fast selection of such an activation for execution. Table 2 gives an approximate count (in terms of memory allocations/deallocations and memory operations) of the cost involved in various aspects of the prioritized CM-5 scheduler. In this table, the symbol r refers to the number of entries in the local result pool and h denotes a hashing constant.

```
AMAM_schedule_loop ()
{
  void (*$f$)(state_type, result_type);
  result_type $r$;

  while (running)
  {
    if (task pool contains a pending activation)
    {
      choose a pending entry $p$ from task pool
      $f$ = $p$.thread_pointer;
      (*$f$)(new_sched_id,$p$.state_pointer,NULL);
    }
    else
    {
      choose an entry $t$ from task pool, suspended on key $k$
      if (result pool contains $k$)
      {
        $f$ = $t$.thread_pointer;
        $r$ = pointer to result pool entry with key $k$
        remove $(k,r)$ from result pool
        (*$f$)(new_sched_id,$t$.state_pointer,$r$);
      }
    }
  }
}
```

Figure 34. A Prioritizing AMAM Scheduler

Scheduler Function	Allocs/Deallocs	Other Ops
Adding Activation to Local Pool	1	10
Suspending an Activation (to a Continuation)	0	7
Terminating an Activation	1	3
Scheduling a Pending Activation	0	7
Selecting a Suspended Activation to Test	0	7
Testing a Direct Key	0	1
Testing a Logical Key	0	$O(r/h) + 2$
Scheduling after a Successful Direct Match	1	3
Scheduling after a Successful Logical Match	1	4

Table 2. Scheduling Costs for the CM-5 AMAM

6.1.2 Storage Model for Partitioned Vectors

Like the abstract machine Ω, its implementation AMAM provides only direct support for a single form of data aggregate, the one-dimension partitioned vector. Such vectors may be nested to form more complex distributed structures. As in our abstract models, we describe the partitioning for an AMAM vector in terms of an associated *partitioning function*, a mapping from a vector index to the node which owns it. In an AMAM program, such a specification typically takes the form of a C function with type signature:

```
host_id partitioning_function (int index, int length)
```

where `length` is the length of the vector. The fact that such functions are parametric upon total aggregate size allows for specification of data layout in the absence of any knowledge of the execution-time dimensions of a vector.

Each node is responsible for storing the indices allocated to it by such partitioning, allocating space within the node's heap. To satisfy requests to access the values of local indices, each node maintains a mapping table which allows it to calculate the heap address of a local index. Since this process of indirection is implicit in every access of a vector index, it is critical that its overhead be minimized. Two systems exist within the AMAM environment to improve efficiency in such computations: a protocol of allocation which bunches indices from a single vector into a contiguous memory block, and a further function associated with each vector to provide cheap offset computation within such a block. We now turn to a discussion of these optimizations.

A Block-based Storage Protocol for AMAM vectors

In Ω the nodal mapping structure was manifested as a table in which an entry was maintained for every *index* stored within the local memory space, recording the address of the allocation for that index. In constructing an implementation of this abstract model, we use a more effective representation which reduces the time required to search the table. A *per-vector* entry is recorded in the mapping table of node N which points to the base of a contiguous block of N's memory in which all its indices of the vector are stored. Placement within such a block is based on increasing vector index. While this scheme places some limitations upon the semantics of distributed AMAM vectors (e.g., all index allocations for a node must occur at one time, thus vectors cannot easily "grow"), the types of problems we intend to solve using the AMAM — DP codes across irregular data structures — will seldom require more general semantics for their aggregates.

Figure 35 illustrates this storage model for a simple vector, `V`, of 9 indices partitioned across 4 nodes. The partitioning function associated with this aggregate, `pf`, maps one index to node 0, three to node 1, two to node 2 and three to node 3. Each node has allocated a block of memory within its own personal memory space to contain these indices. Every such block is conceptually divided into a certain number of *slots*, regions of memory exactly large enough to contain one index of the vector. The figure shows how the ordering which indices been mapped (by `pf`) to a given node adopt in that node's memory block, namely an order sorted on position (index) in `V`. Thus, node 1 (onto which indices $2, 3$ and 8 are mapped) has allocated a block with three slots; the value of vector index 2 is stored in `slot 0`, the value of index 3 in `slot 1`, and so on. A second function `rl` is also associated with the vector — the purpose and functionality of this *relative location function* is described in the following section.

Each node of the AMAM keeps, within its mapping table, both a pointer to the base of block representing its indices of the vector as well as a record of the *slot size* of the block. Pointers to the memory representing an individual index can be computed from these values. Given the *slot address* of an index (i.e., the number of the slot it occupies in the block), the pointer to that index's memory can be cheaply evaluated as:

$$index\_address = base\_address + slot\_address \times slot\_size$$

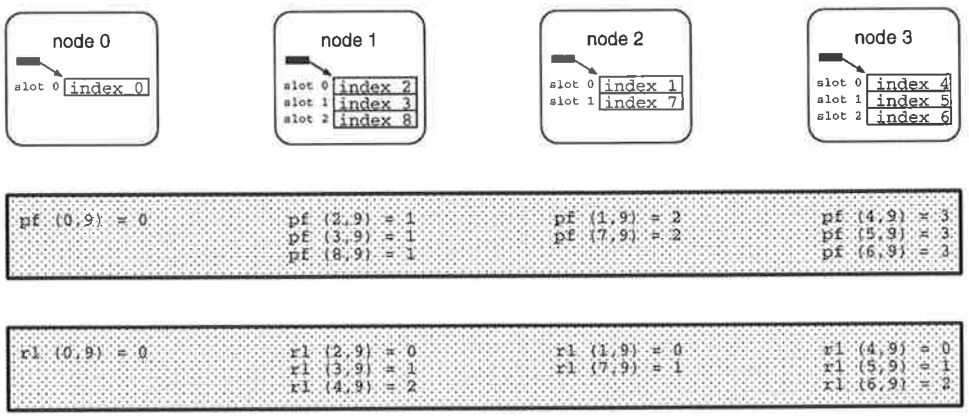


Figure 35. Relationship Between Partitioning Function and Relative Location Function

For this process to be practical we must have a low-cost way of determining the slot address for a given vector index. It is clearly possible to build an algorithm to compute this *relative location* by repeatedly applying the vector's partitioning function to all indices preceding the desired one, counting the number of local indices. However, such an approach is $O(n)$ in the number of local indices and potentially quite expensive for large vectors. Furthermore, this process of slot address determination introduces a wide variance in the cost of accessing a vector index, something that is generally undesirable from the point of view of providing a system with predictable performance. A better approach is to introduce a *relative location function* with each vector which provides cheap (preferably $O(1)$) determination of the slot address for an index. We turn now to a discussion of these functions.

Relative Location Functions

In an effort to optimize the process of slot address computation, we associate a second function (in addition to the partitioning function) with each AMAM vector. This *relative location* function, is a programmer-provided algorithmic means of determining slot addresses in less time than required for the repetitive application function method described previously. Given an index and the vector's length, the function must (cheaply) return the address of the slot in which the index resides (on some node). For every given partitioning function there is an associated algorithmic relative location function; the two functions are closely related. It is important when the programmer is associating such functions with a vector that the partitioning function and relative location do not provide conflicting views of storage layout.

To clarify the relationship between the partitioning function and the relative location function, consider again the example shown in Figure 35. The upper shaded box in this figure shows a definition of a partitioning function `pf`. The nodes above this box show the mapping of locally-held indices to slots. The bottom shaded box shows the relative location function, `rl`, which is paired with `pf`. The construction of relative location functions is discussed in Section A.1

6.1.3 The Communications Subsystem

The abstract machine Ω assumes a communication mechanism which allows transparent update of the state of a foreign node. We implement this flavour of communication in AMAM by means of *active messages* [134]: low-level asynchronous messages whose receipt causes a message handler (specified within the data of the message) to be invoked. On the CM-5 such a protocol of message passing is available as the very lowest (CMAML) layer of the standard communications library CMMD [126].

The Ω machine defines two communications primitives: `request` and `result`, the former an addition to a remote task pool, the latter to a remote result pool. Internal to AMAM we define two message handlers, one for processing request-generated active messages arriving at a node, one to deal with result-generated active messages. The functionality required for each of these handlers is minimal, the creation of a table entry for the appropriate pool and the copying of information from the message into this new entry. Some care must be taken to assure that asynchronous updates do not interfere with other state manipulations interrupted when the message was received. In practice it is easy to achieve such atomicity: our CM-5 implementation explicitly does not use an interrupt-driven protocol of active messages, preferring the alternative polling approach in which a message is only received when a node explicitly polls its network interface for buffered messages. Various studies (e.g., [119, 44, 61]) on the CM-5 have shown this protocol of active message receipt involves considerably less overhead. Care in the placement of polling calls avoids any semantic problems caused by interrupted AMAM nodal state updates.

6.1.4 A Hybrid Model for Keys

As described in Section 6.1.1, the concept of a *suspension key* (or simply *key*) is one central to the specification of correct control flow in AMAM programs. Such tags implicitly specify the schedulability predicate for suspended thread activations. As such they represent a mechanism for synchronization and data flow as well as defining execution order and dynamic control flow for a thread.

In our abstract model, we considered all keys to be logical entities, and thus all determination of schedulability to involve logical scans of the result pool. In practice such searches may prove costly, especially given the fact that there is no guarantee that the work will result in an activation being scheduled. AMAM supports this logical notion of keys directly, and thus retains the specificational generality of the synchronization and data flow model embodied in Ω. However, we also introduce a distinct form of key, the *direct key*, which is less general but for which the matching process occurs directly, outside of the result pool, and is thus considerably more efficient. The characteristics and advantages of both of these types of key are discussed below.

Logical Keys vs Direct Keys

AMAM's *logical key* is represented by a simple four-tuple of integers. When an activation suspends upon such a key, it waits for the appearance of the identical four-tuple as a tag for a value in the local result pool. Such a matching involves a non-trivial amount of computation: the entire result pool (or possibly a subset of the pool defined by a hashing value) must be scanned.

However, logical keys provide the greatest generality in specifying data and control flow. The integers which make up the key are purely logical entities — they have no explicit meaning or relationship to past execution paths. Thus, any two activations which know (by some means) of a given logical key may communicate. No master-slave, or any other, relationship need exist between the two activations.

The second form of key within the AMAM implementation is the *direct key*, which bears close similarity to efficient synchronization mechanisms available in various implementations of dynamic data flow, such as the Id [12, 69] language, several USC software prototypes [43, 61, 74], and the RMIT/CSIRO [1, 2] and EM-4 [140, 104] parallel architectures. Such keys are always created dynamically, through a call to the

AMAM system library. This system call returns a pointer to a local area of memory large enough to hold (a pointer to) a result value, plus a presence bit. Initially the result field is empty and the presence bit off. A result written to the node with a direct key k, causes k's presence bit to become set and the received value to be copied into the memory reserved for the key. Note that such a received value does not appear within the nodal result pool. Once the presence bit of a direct key is set, resumption of the suspended activation is possible. In AMAM it is very cheap for the scheduler to make such a check: the pointer to the key object is stored within an activation's task pool entry, and simply dereferenced to determine schedulability.

There is a tradeoff: The patterns of control and data flow which can be modelled by direct keys are more limited than with logical keys. Because the key is a dynamically created entity, it is impossible for communication to occur from thread activation B to thread activation A unless, at some earlier time, A has communicated the key to B; that is, a direct key cannot model unsolicited communication.

This hybrid system of keys is, we believe, a unique variation on the traditional solutions to tag-matching in a data flow system.

6.1.5 AMAM System Calls

Table 3 describes the interface presented by the AMAM system. We describe briefly the semantics associated with each of these system calls.

Basic System Calls

`AMAM_num_hosts` returns the number of nodes present within the current AMAM.

`AMAM_me` returns the identity of the node on which the call is made. This is the AMAM equivalent of the symbol M used in Ω-programs.

`AMAM_new_id` generates a new identifier.

Threading System Calls

`AMAM_enter_activation` enters an activation for a specified thread into the local activation pool, marking it as pending and associating a given initial state.

AMAM System Calls

System Call	Arguments
AMAM_num_hosts	
AMAM_me	
AMAM_new_id	
AMAM_enter_activation	void *thread (int, void *, void *), void *initial_state
AMAM_suspend_activation	int sched_id, logical key k, void *state, void *continuation (int, void *, void *)
AMAM_suspend_activation_d	int sched_id, direct key k, void *state, void *continuation (int, void *, void *)
AMAM_terminate_activation	int sched_id
AMAM_new_direct_key	
AMAM_enter_result	logical key k, void *value
AMAM_enter_result_d	direct key k, void *value
AMAM_result	int host, logical key k, void *value
AMAM_result_d	int host, direct key k, void *value
AMAM_request	int host, void *thread (int, void *, void *), void *initial_state
AMAM_enter_mapping	vec_id v, int slot_size, void *heap_ptr
AMAM_get_mapping	vec_id v, int slot_addr

Table 3. Summary of AMAM System Interface

`AMAM_suspend_activation` changes the activation's entry in the task pool, by setting the "suspended" status, recording a continuation and a logical suspension key. `AMAM_suspend_activation_d` is identical, but the key upon which the suspension is made is direct rather than logical.

`AMAM_terminate_activation` removes an activation from the local pool.

Key Management System Call

`AMAM_new_direct_key` generates a new direct key by allocating space for a result and tagging it with a presence bit initialized to off.

Results Subsystem

`AMAM_enter_result` places a value into the local Result Table, tagged with the given logical key.

`AMAM_enter_result_d` copies a value into the specified direct key and sets its presence bit to true.

Communication Routines

`AMAM_result` causes an active message to be sent to a remote node which, upon receipt by that node's handler, causes a value to be entered into the result pool and tagged with the specified logical key.

`AMAM_result_d` also causes a message to be sent and a handler to be invoked on a remote node. In this case, however, the job of the handler is to copy the given value into the space allocated for the specified direct key and set the presence bit to true. Note that it is only legal to perform such an action if the direct key passed as an argument was generated on the node to which the value is to be written.

`AMAM_request` sends an active message to the specified node which causes a new activation to be entered into that node's task pool, marked pending, and associated with the initial state sent within the message.

Partitioned Vector Support

`AMAM_enter_mapping` adds a new entry to the local mapping table with the given vector identifier, slot size and pointer to the memory block holding the vector's local indices.

`AMAM_get_mapping` calculates a pointer to a local index of the specified partitioned vector. The system call must be provided with the slot address of the index in question.

6.2 Optimizations for NDP use of the AMAM

The multi-threading implementation we have described is a general-purpose environment for distributed memory computation. That is, although we have designed the AMAM with NDP execution in mind, its functionality as presented is not specialized towards such execution, and it is not surprising that there are situations where the standard AMAM system discourages optimally efficient specification of such NDP programs. For these few cases it is advantageous to define specializations (or augmentations) of AMAM functionality that offer a lower-cost specification. This section presents four such NDP-specific optimizations which are present within the current AMAM; Section 6.4 describes how such forms are used in the optimized expression of DP threads.

6.2.1 Multiple-Direct Keys

In the specification of threaded forms for higher order DP operators it is not uncommon for an activation to spawn many child activations to perform the per-index elemental computations for each local index of a vector. Typically, once the parent has spawned its quota of children, it wishes to suspend until such time as *all* the children have completed an execution. That is, only after each child has contributed a result value, does the parent activation wish its continuation to be scheduled. Examples of such behaviour are evident in the Ω-thread definitions for **MAP** (Figure 23) and **SCAN** (Figure 27). In both these descriptions, the fact that AMAM's suspension call only allows for suspension on the arrival of a single result value, forces us to synthesize a protocol of repeated suspension upon a single key. Thus, rather than an activation simply suspending until it's children have all contributed their result, it is required to: suspend, resume upon receipt of the first child result, suspend again, resume upon receipt of second child result, and so on. This protocol continues until all child values have been received.

In AMAM, this model of suspension awaiting multiple results displays poor efficiency. Because AMAM's threads are non-blocking, we must express the protocol as shown in Figure 36. This figure illustrates an activation waiting on results from N children. The cost of such a wait is high: here we require $N + 1$ activations to be scheduled and N keys to matched as part of that scheduling. Furthermore, the first $N - 1$ activations of the continuation thread `AMAM_CONT` cause very little useful work to be performed (a single operation), leaving us with a high overhead computation.

To reduce the overhead of such waits we introduce the notion of a *multiple-direct key*. Like the direct key discussed earlier, the multiple-direct key is a dynamically generated synchronization point. At the time of its creation, a *write-count* is associated with the key. Whereas suspension upon a direct key has the implied semantics of resumption after the key has been written to exactly once, suspension

```
AMAM_THREAD (sched_id, state_ptr, dummy)
{
  int i;

  ... allocate initial states for N children ...
  for (i=0;i<N;i++)
  {
    child_i_state->arg1 = ...
    child_i_state->key = k1;
    AMAM_enter_activation (AMAM_CHILD,child_i_state);
  }
  state_ptr->counter = N;
  AMAM_suspend_activation (sched_id,k1,AMAM_CONT_1);
}

AMAM_CONT_1 (sched_id, state_ptr, result)
{
  store or consume the result
  state_ptr->counter = state_ptr->counter - 1;
  if (state_ptr->counter > 0)
    AMAM_suspend_activation (sched_id,k1,AMAM_CONT_1);
  else
  {
    all result have arrived, continue computation
  }
}
```

Figure 36. Suspending an Activation in Wait of Multiple Results

on a multiple-direct key implies resumption after the key has been written a number of times *equal to the key's write-count.* Each of the results written to a multiple-direct key are stored in the memory allocated for the key.

Thus the computation expressed in Figure 36 could be more concisely and efficiently expressed as shown in Figure 37.

AMAM's implementation of multiple-direct keys represents them as a block of memory large enough to store (pointers to) a number of results equal to the argument passed during key creation, plus a counter recording how many results are yet to arrive. The process of writing to such a key becomes one of copying a pointer and decrementing the count. Determining the schedulability of an activation suspended

```
AMAM_THREAD (sched_id, state_ptr, dummy)
{
  int i; void *multi_key;

  ... allocate initial states for N children ...
  multi_key = AMAM_new_multiple_direct_key (N);
  for (i=0;i<N;i++)
  {
    child_i_state->arg1 = ...
    child_i_state->key = multi_key;
    AMAM_enter_activation (AMAM_CHILD,child_i_state);
  }
  AMAM_suspend_activation_md (sched_id,multi_key,AMAM_CONT_1);
}

AMAM_CONT_1 (sched_id, state_ptr, *result_array)
{
  all result have arrived, continue computation
}
```

Figure 37. Using a Multiple Direct Key

on a multiple-direct key is cheap: the scheduler needs only look at the count to determine whether it has reached 0.

6.2.2 An Alternative To Barrier Synchronization

It has been noted several times in this thesis that, by their very nature, Data-Parallel computations are synchronizing operations. This arises principally from the need to ensure that an aggregate value created by one DP operation is not consumed by another operation until such time as the entire aggregate has been computed. The second operation must be protected from the possibility that it may attempt to read an index from the aggregate before it has been written.

In providing an execution environment in which DP-style operations may be efficiently evaluated it falls to us to provide a mechanism for making such synchronization low-cost. We have previously considered a synchronization mechanism (for Ω, see Section 5.2.2) in which participants of a DP computation took part in a barrier synchronization, the protocol synthesized from the simple communications primitives of the language. While this is an effective solution, performing such a barrier can be expensive. In this section we consider an alternative approach which allows for a DP style of synchronization to be specified in terms of the presence of a vector id in a node's mapping table. As we shall show, this protocol, reminiscent of classical I-Structures [13] from data flow computing, will often significantly reduce the overhead of synchronizing participants in a DP computation.

Characterizing the Synchronization Requirements for DP

We begin our discussion of global synchronization in DP thread programs by characterizing the need for such synchronization in the execution of such threads.

Figure 38 shows an AMAM program in which global synchronization is important to ensuring correct evaluation. The program in the figure makes use of two DP threads, `MAP` and `DEREF`, the former being an implementation of the DP operator of the same name, the latter being a thread which forwards a copy of an index's value to a foreign node (c.f., the Ω definitions in Section 5.2.3 and 5.2.2).

Consider the situation where the DP `map` operation is co-operative between two nodes, node 0 and node 7. Suppose that our definition of the `MAP` thread does not provide a global synchronization phase, and that node 0 completes its work in the co-operative scheme prior to node 7. Thus control on node 0 passes to the activation of `AMAM_CONT` while node 7 has yet to finish its work on the `MAP`. Now, node 0 causes an activation for `DEREF` to be entered into node 7's task pool. Suppose that for some reason, node 7's activation of `MAP` suspends. It is possible that the activation of `DEREF` will be scheduled on that node, prior to the completion of the `MAP`. Thus we will be reading the value of an index of the `map` result yet to be evaluated.

As this example demonstrates, the purpose of synchronization in DP thread programs is to ensure that a data item (usually a vector index) is never read by one thread activation prior to its value being written by a different activation. By enforcing a protocol whereby the second activation cannot possibly begin until after the first has finished (e.g., by placing a barrier synchronization between the two), we can always ensure that such a read-before-write situation cannot occur.

```
AMAM_THREAD (sched_id, state_ptr, dummy)
{
  void *direct_key;

  direct_key = AMAM_new_direct_key();
  ... allocate state for DP MAP ...
  map_state->key = direct_key;
  AMAM_enter_activation (MAP,map_state);
  AMAM_suspend_activation (sched_id,direct_key,AMAM_CONT);
}

AMAM_CONT (sched_id, state_ptr, result)
{
  vector_descriptor map_res, void *direct_key;

  map_res = result;
  direct_key2 = AMAM_new_direct_key();
  ... allocate state for DEREF ...
  deref_state->vector = map_res;
  deref_state->idx = 17;
  deref_state->key = direct_key2;
  AMAM_request (7,DEREF,deref_state);
  AMAM_suspend_activation (sched_id,direct_key2, AMAM_CONT_2);
}
```

Figure 38. Demonstrating the Need for Synchronization in DP Threads

Synthesizing a Protocol for Synchronization

When considering the definition of DP operations in Ω we addressed the need for global synchronization by constructing a special thread **SYNC** which implemented a barrier synchronization between a set of participants. DP threads which were not naturally synchronizing because of their implementation (e.g., **REDUCE**) were required to activate this synchronization thread prior to the final value of the DP operation being returned to the caller. Applying such an approach to the previous example, node 0 would not have had its activation of `AMAM_CONT` schedulable until both nodes 0 and 7 had completed their work on the `map`.

While such an approach to synchronization is clearly effective at eliminating the possibility of read-before-write situations in a DP thread code, the cost of constructing such a barrier protocol from AMAM's low-level communications primitives would prove prohibitive. Inspecting the **SYNC** thread we defined for Ω, we can see that its synchronization of n participant nodes requires $O(n^2)$ communications and for $O(n)$ activations to be scheduled on each participant node.

Furthermore, this approach to synchronization is proactive in its prevention of read-before-write conditions. The cost of synchronization is incurred even in the case where individual DP operations present within a program do not naturally contain the possibility for read-before-write conditions[1].

These factors lead us to consider a more optimistic scheme of providing synchronization. We take our inspiration from the reactive style of synchronization implicit in the I-Structure [13].

An Optimized Alternative: Presence-based Synchronization

If we consider the data structures and functionality of the AMAM as described in this chapter, we can see that read-before-write errors arise only in the use of the nodal mapping table to compute an address for a vector index. Specifically if we use this table to compute a pointer to a vector index which has yet to be filled by some creating thread, then we have suffered this type of error.

One approach to avoiding such problems is to arrange a protocol whereby a vector's identity is only added to a mapping table *after* the operation which creates the vector is complete. Such an approach allows for a vector's presence in the mapping table to be a predicate which defines whether or not a given dereference attempt will cause a read-before-write error[2]. This predicate may be evaluated whenever a dereference is to be made; attempts which cause no error may be safely satisfied immediately, others can be delayed until such time as the creator operation has completed.

AMAM provides a mechanism for exploiting this style of synchronization, in the form of a system call `AMAM_verify_mapping`. We have used this facility, coupled with a protocol of mapping table entry, to experiment with the construction of DP threads which have significantly lower synchronization cost.

[1]It may arise that a DP operation consuming the output of another can safely have its execution overlapped with that of the producer. This is possible when the consumer's pattern of access of the result is predictable, regular and totally on-node. In such cases, the work involved in synchronizing the producer operation prior to allowing the consumer to begin is entirely unnecessary.

[2]This presumes that a vector identifier is never reused in an AMAM execution and that a program never incorrectly attempts to dereference a vector that has been deallocated.

The `AMAM_verify_mapping` function proceeds by:

- checking to see whether the specified vector id is present in the local mapping table;
- in the case where the id is present, returning a `NULL` value signifying that vector may be dereferenced safely;
- in the case where the id is absent, a new direct key is generated and returned as the result — the calling activation is expected to suspend upon this key which will, at some later time, be filled (by call-back) to signify that it is now safe to perform the dereference.

Section 6.4.1 presents an AMAM translation of the remote vector dereference thread **DEREF** which makes use of this system facility. This thread definition demonstrates how such presence-based synchronization can be used to eliminate read-before-write errors without the need for an expensive protocol of barrier synchronization.

A major advantage of this scheme is that a synchronization cost is paid only when needed. Typically a call to `AMAM_verify_mapping` for a given vector identifier need be made only once within an activation (or series of activations and continuations) prior to the first dereference of a given vector. It is safe to assume that if a mapping was verified at time t, it will be safe to dereference the vector at any times provably after t, since identifiers are never removed from a mapping table except at the time of vector deallocation.

In summary, this alternative approach to synchronizing DP threads offers significant opportunities in reducing the amount of communication required to ensure read-before-write errors cannot occur. In place of a complex protocol of passing messages to synthesize a barrier, we can make use of a cheap system call which determines the safety of dereferencing a vector and can arrange for issuing a call-back message when such a dereference can safely be made. It is clear that this process of notifying activations via call-back must necessarily involve a certain amount of communication in the AMAM. In general, however, observation of AMAM codes in which the presence-based sychronization is used, suggests that such communication constitutes a much lower volume than the synthesis of a barrier[3]. Thus, for such program, our presence-based scheme represents a significantly lower-cost synchronization mechanism for DP threads.

6.2.3 Working With Participant Sets

In the process of meeting the data and control dependencies necessary to fully describe the semantics of Data Parallel operations in Ω, we relied heavily on the notion of participant sets (see Section 5.2.1). Given that we plan to utilize such abstract definitions to provide efficient AMAM implementations of generic partitioning-independent operations, it is useful to augment the AMAM to provide support for operations over such sets. Below we describe the semantics of a small group of system calls, reminiscent of the set operations we defined in Section 5.1.6, which we have chosen to add to our implementation in order to provide efficient manipulation of participant sets.

`AMAM_single_pset` accepts a single host identifier as it argument and creates a new participant set which contains only that identifier.

`AMAM_range_pfun`, when passed a partitioning function and a vector length, produces the set of all hosts who own indices from a vector with that partitioning and that length.

`AMAM_in_pset` is a predicate which determines whether a given host identifier is contained within a given participant set.

`AMAM_and_pset` takes two participant sets and produces their intersection; that is, the set of all host identifiers present in both input sets.

`AMAM_seq_pset` accepts a participant set p and an integer i, and returns the set element which is the i'th smallest (i.e., the i'th member of a sequence formed from the elements of the set).

[3]Presence-based synchronization is used exclusively in the implementations of DP operators used in our language implementation (see Chapter 7) targetting AMAM. Early experiments with performance benchmarking barrier-based versions of these codes yielded results which were an order of magnitude less efficient than the presence-based versions which were later adopted. This difference in performance was due exclusively to a greater communications volume.

In our CM-5 AMAM, we represent processor sets as bit-vectors, leading to efficient implementations of the `AMAM_single_pset`, `AMAM_in_pset` and `AMAM_and_pset` operations. The sequencing operation is more expensive, requiring i bit-shift operations. Determining the range of a previously unseen partitioning function is an operation which inherently requires the function to be evaluated for every index of the vector. An optimization which reduces the cost of subsequent range calculations for the same partitioning function is discussed in the following section.

6.2.4 Inverting Partitioning Functions

An operation that arises often in the coding of Data-Parallel threads is that of determining, given a vector v and its associated partitioning function pf, which of the indices of v are local to the node executing an activation. Such an operation is visible in the Ω definitions for **MAP** and **SCAN** where the vector partitioning function is applied across the entire index space of the vector to compute the set of local indices.

In the course of a DP operation involving a set of partitioning functions, it is likely that each node will perform the same (partial) inverse computation many times. Two factors contribute to this: the likelihood that the same vectors will typically be used in several DP operations over their lifetimes, and the fact that a number of vectors (of the same length) may share the same partitioning function. Every evaluation of partitioning function inverse (for a vector of a given length) will produce the same result.

With this in mind, one approach towards reducing the cost of DP threads is to arrange so that this $O(n)$ (in the length of the vector) computation was only performed once on each given node and its result stored to satisfy future requests to invert the same partitioning function (in the context of an equal length vector). We implement this memoization of inverse computation by providing a system call `AMAM_invert_pf` which accepts a pointer to a partitioning function and a vector length and checks to see whether the inversion of a vector of equal length and equal partitioning has already been performed. If such a previous operation has occurred on this node, the result will reside in a nodal data structure, the *inversion table* — we can thus simply return this value as the result of `AMAM_invert_pf`. In the instance that no prior inversion matches the current request, we must compute the inverse by the normal

$O(n)$ method. The result of this computation is both returned by the system call and added to the local inversion table.

6.3 Overview of Related Architectures

In the definition of AMAM, we have now described a complete environment for practical distributed multi-threaded execution. The implementation we have outlined bears some similarities to a number of other research prototypes which seek to provide a multi-threaded execution as a basis for different models of parallelism (e.g., macro data flow, parallel object oriented programming). In this section we briefly survey the most important of these related architectures, commenting upon their points of convergence and divergence with the AMAM model we have presented.

6.3.1 P-RISC and TAM

TAM [44, 119, 31, 30] and P-RISC [86, 85] are two multi-threaded abstract machines designed and implemented as targets for compilation of the implicitly parallel, higher-order language Id [84]. Each is based upon the principle of macro data flow [69] which aims to provide, on a traditional multiprocessor, a model of execution reminiscent of that found in traditional dynamic data flow. We describe each of these architectures in turn and then comment (in the discussion section on page 174) on the relationship of these designs to the AMAM.

P-RISC

The P-RISC (Parallel RISC) model consists of a number of processing nodes, each of which owns a local memory, a communications network which allows these nodes to exchange messages, and a (logically distinct) global memory. The latter memory, which may actually be partitioned among the nodes, may only be accessed by split-phase instructions. That is references into the global memory space can be assumed to involve long latencies and thus are implemented as separate request and response messages.

The machine executes programs which consist of a number of P-RISC graphs, each representing a unit of computation which can be activated (e.g., a function). Execution begins with one of these graphs being instantiated upon one node of the machine: this involves allocating a block of memory to be that activation's *frame*, a region of local storage. Instructions from this active graph are then executed in sequence. During this execution, the graph may specify the instantiation of other graphs either on the initial node, or on another node. This represents the spawning of a new *microthread* of control. The semantics of P-RISC's microthreads defines them to be run-to-completion (i.e., they are non-blocking threads). The P-RISC allows for an arbitrary number of microthreads to be simultaneously active upon a node of the machine. Each maintains a pointer to the frame for the activation which spawned it; this may be a *global pointer*, that is, a pointer into another node's memory.

Each node maintains a *scheduling queue*, a data structure which records the microthreads which are active upon it, and governs the multiplexing of the node's single thread of control among such microthreads. This is implemented as a simple stack; as new microthreads are activated on a node they are pushed onto that node's stack. When a microthread terminates its execution, the top stack element is popped and becomes the executing microthread. While a node is executing, any messages that are received are queued in another data structure. Each message contains the address of a handler that should be invoked upon its consumption: when the scheduler is faced with an empty stack, it passes control to the handler for one of its queued messages.

Split-Phase access is manifested in P-RISC using a continuation model. When such an access is issued, a message is sent to the appropriate destination which contains, aside from data necessary to the access, a pointer to an instruction in the sender graph from which control should continue after the result becomes available. The sending micro-thread then terminates. At some later time, on the destination node, the message is received and ultimately consumed to pass control to the specified handler. That handler communicates a response to the sending node, with the continuation it received as the handler address.

Synchronization between micro-threads is specified by a `join` instruction which makes reference to a variable in the activation's frame and a count of the number of micro-threads that will participate. The variable is initialized to zero, and incremented when each micro-thread issues its `join`. If the incremented value is less than the count, the micro-thread simply dies. When the last micro-thread has issued the instruction (which makes the variable's value equal the count), it is permitted to proceed beyond the synchronization point.

TAM

The Threaded Abstract Machine (TAM) model proposed by Culler bears many similarities to P-RISC largely due to commonality of backgrounds and intended purposes. As with the former model, TAM is a machine made up of a set of processing nodes, each with an associated memory, and a logically distinct global heap. The latter is assumed to be partitioned between the various nodes.

A TAM program is made up of a collection of *code-blocks*, each a collection of (non-blocking) threads and message handlers (called *inlets*). Upon activation of a code-block, a region of memory called a *frame* is allocated to hold variables local to the code-block. As part of every frame, storage is allocated for a structure called the *local continuation vector* (LCV). This records the activations of each thread within that block, and is the principal resource used in scheduling.

The scheduling model for activations of code-blocks and their threads is hierarchical. At the top level, each node maintains a list of all local code-block activations which are *ready*, that is which have at least one thread activation which may immediately be executed. Execution begins by one of these ready code-block activations being selected. Once this has occurred, an inner scheduling model is applied to schedule threads from this chosen activation's continuation vector. Whenever execution of one thread terminates, the LCV of the current activation is again consulted for a new thread to execute. Only when the current frame has an empty LCV do we consider returning to the outer scheduling algorithm to choose a new code-block activation from the ready queue. The time between applications of this outer scheduling process is called the *quantum* of the chosen frame. The motivation in providing this two-level scheduling policy lies in improving locality by concentrating execution on a single frame for as long as possible (possible since all threads within a code block share its activation frame). This also makes it possible for a compiler to determine that a set of thread activations will execute within the same quantum, and thus permit information to be passed between such threads in registers.

Each code-block defines, in addition to its threads, a set of inlets which act as handlers for messages sent to threads of this code-block from threads of other code-block activations (possibly resident on other nodes). An inlet is activated immediately upon reception of a message, preempting any thread that may be executing at that time. Typically, their role in computation is simply to copy transmitted data into appropriate fields of the receiving frame. Inlets are used to implement split-phase access of elements of the global heap in much the same way as previously described for P-RISC: the address of the sending frame is transmitted with thc message to enable the thread sending the result to determine to which activation's inlet the resultant message should be directed.

Threads are activated by a special *fork* instruction in the TAM language. Such thread activations are modelled as simple additions to the LCV. Additionally, a thread may invoke a code-block by causing the allocation of a new frame (on some node of the machine) and passing an initial state (the arguments to the code-block) as a message to a designated inlet of the code-block.

TAM's threads may be specified to observe a synchronizing behaviour. This is achieved by the introduction of an *entry counter*. The semantics of *fork*ing a thread are defined such that activation of the thread (and hence addition to the LCV) only occurs when this count has been made zero. Attempts to *fork* a thread with a non-zero entry count fail, but cause the count to be decremented.

Discussion

Clearly P-RISC and TAM bear some close parallels to the architecture we have delineated for AMAM. All are multi-node models in which each node provides some support for multiplexing a single thread of control between a number of thread activations. Furthermore, the nature of the threads in all models is non-blocking. However, whereas the models derived from data flow choose to group threads together into larger blocks which share a state object, AMAM opts for a flat model in which each thread activation is responsible for its own distinct state object. This approach is feasible since, as was the case for our abstract model Ω, we intend to use AMAM's threads as representations of relatively large computations (e.g., whole functions or sections of an NDP operation — see Sections 5.2 and 7.3.3 for discussions of such modelling). This contrasts with the finer grain-size expected for P-RISC and TAM, where a thread might represent only a handful of machine instructions. Thus, although AMAM must maintain a number of state objects equal to the number of current thread activations (as opposed to one state object per group of activations), there will be overall fewer activations required to carry out a similar computation.

Hence the number of state objects maintained by each of the systems is roughly comparable.

A noticeable difference between AMAM and the data flow derived models is the pattern of program instantiation. In TAM and P-RISC a program begins as an activation executing on a single node, whereas AMAM's distinguished startup thread is activated on all nodes independently. This difference stems from the form of execution which the multi-threaded environment is intended to model: the former models are designed with a data flow style of execution (complete with forks and joins of control/data) whereas our target is to support an extension of the traditional SPMD model commonly used for executing DP codes.

The nodal scheduling model of P-RISC bears some similarities to that of AMAM particularly the existence of a nodal structure recording activations and a scheduling loop which simply selects an element of this structure. Note however, that the two models handle continuations after split-phase operations in a different way — where AMAM stores the continuation on the node which issued the split-phase operation (and therefore must periodically check for the arrival of the result), P-RISC sends the continuation as data in the requesting message and relies on the thread which forwards the result to pass this continuation back (causing slightly higher communications overheads). TAM's scheduling model is quite different to either AMAM's or P-RISC's in that it makes some effort to take advantage of inter-thread locality — this approach enables threads to share implicit state (e.g., registers) and also reduces the cost of scheduling threads which are related to those just completed. However, the support of such a hierarchical scheduling policy requires a complex dynamic data structure across the machine which may be costly to maintain. TAM's model of split-phase is similar to P-RISC's although rather than passing continuations, the sender gives the local frame address and the specification of which inlet should receive the result. Together these can be thought of as a specification of a continuation equivalent to those found in P-RISC and AMAM.

It is worthwhile noting that all three models provide an efficient mechanism for passing state to a continuation thread: in AMAM this is achieved by storing pointers to the state object and the continuation of an activation which has suspended. TAM and P-RISC achieve this by a storage model which shares frames between all threads in a block, and thus retains state information which becomes local to the continuation when it is ultimately activated. The AMAM model is the more general in that it permits the passage of state information to an arbitrary continuation; the data flow derived models insist upon a continuation being a thread within the same code-block.

Finally it is interesting to note the similarities between the forms of communication and synchronization present within the different models. Clearly all are based upon an asynchronous handler-based style of communication of the sort embodied within Active Messages. The synchronization mechanisms of P-RISC (the `join`) and TAM (*entry counts*) are analogues of the (multiple) direct key mechanism we have described for AMAM thread synchronization. In all three systems, this form of synchronization presumes a relationship between the sending and receiving threads: the dynamic nature of the identities in the synchronization scheme (direct keys, activation frames) means that a given thread can only know the identity of a particular synchronization element if it has previously been passed this information by the creator of that element.

AMAM offers an alternative form of synchronization which does not presume such a relationship, namely the logical key. Such keys are presumed to be static or semi-static entities whose identity can be known to two threads who have no direct relationship (through either previous communication or heritage). Thus it is the case that whereas such threads cannot express communication/synchronization via direct keys (and the facilities of TAM and P-RISC), their interaction can be specified in terms of AMAM's logical keys.

In summary, AMAM, TAM and P-RISC represent models of multi-node multi-threaded computation that are closely related but which cater to disparate needs. The latter models cater towards threads which are very fine grained, all fork from a single starting thread, and only every communicate with the thread which forked them. Conversely, AMAM provides an environment for efficiently supporting somewhat larger grained threads, which need not be activated from a single starting thread and which may communicate in an arbitrary fashion (i.e., any thread activation may communicate with any other, not just its parent activation). These differences are manifestations of AMAM's place as a basis for NDP computation (as compared with TAM and P-RISC's existence as a model for implementing macro data flow).

6.3.2 Charm

Charm [64, 111, 112, 113] is a system providing a distributed memory multi-threaded environment for an object-oriented style of execution. The most recent manifestation of this environment, CAB [114], is particularly oriented towards supporting irregular patterns of computation. It is this version of the system which we describe here.

Like the TAM and P-RISC models we have previously described, Charm/CAB provides the programmer with non-blocking threads called *uThreads* or *atomic computations*, grouped together into larger control constructs called *uProcesses* or *chares*. The uProcess also contains a number of message handlers called *entries*. Each uProcess activation has an associated frame of memory which stores its local variables and state; this memory can be viewed as the encapsulated state of a uProcess *object*, manipulable only by the entries of the uProcess. The uThreads within each uProcess represent finer grained activation units which do not own their own state but have access to a *Shared Data Area (SDA)* of the containing uProcess' frame. Thus the Charm thread model can be said to be hierarchical in the same way that P-RISC and TAM were, and is similarly divergent from AMAM's flat thread model.

The model of computation embodied in Charm/CAB is message-driven. The individual uProcesses are passive entities which, once they have been activated upon a node of the machine, carry out computation only upon receiving a message. Each message contains the address of an entry of the receiving uProcess which corresponds to the new uThread to be activated when eventually the message is consumed. The nodes of the machine each maintain a queue of the messages they have received (both from uThreads executing locally and those active on other nodes) and uses this structure as the principal scheduling data structure. That is, the node consumes a message from the queue (and hence executed one of the uProcess' uThreads) whenever it falls idle. This computational model differs from TAM's messages and AMAM's result-based communication in that it directly links every message to the instantiation of a new thread of control rather than simply using such communications to pass data (which may through fulfilling data dependencies cause a later activation). The Charm message passing model is more closely approximated by AMAM's request-based communications protocol, that is by our mechanism for remote thread activation.

6.3.3 Pebbles

The Pebbles [99, 100, 10, 109] system is another multi-threaded execution model arising from research into introducing a dynamic data flow style of execution into a von Neumann computational environment. At present the architecture is being used as a compilation target for a distributed memory implementation of the SISAL language [78, 41].

Pebbles offers a flat threading model (i.e., one which has no hierarchies of threads as were present in TAM, etc.) which is non-blocking. The intention is that each thread in the system plays the role of a node (or actor) in a dynamic data flow graph. With this in mind, Pebbles threads do not have large state objects (frames) associated with their activations. Rather, each activation maintains a small data structure called a *framelet* which stores only its inputs. Threads are synchronizing in the sense that they may not be scheduled for execution until such time as all input values are available. Thus, the notion of a Pebbles framelet can be compared to AMAM's mechanism for multiple direct keys. There is no Pebbles analogue to AMAM's per-thread state object: all transfer of state information between a thread activation and its continuation occurs through explicit message passing rather than (as is the case in AMAM) through an implicit mechanism of transfer through management of a state pointer.

Different activations of the same thread are distinguished by a unique *colour*; messages directed towards a node carry *tags* which define the precise thread to which they must be delivered. The Pebbles abstract machine incorporates a synchronization unit which performs the matching/synchronization of tags to thread activations. This process is analogous to key matching in AMAM.

6.3.4 Nomadic Threads and the I-Structure Software Cache

A number of prototype multi-threaded multi-node abstract machines [43, 74, 61] have recently been proposed by researchers at the University of Southern California as test-beds for the experimental analysis of novel execution models[4]. These architectures are based upon a generic model of multi-threaded execution [139] similar to that embodied by machines such as TAM and P-RISC. Like these models, the USC machines are

[4]This work derives from earlier research [39, 40] by the same group which considered hardware architectures for macro data flow.

frame-based: that is, each activation of a code block is granted a block of state memory which may be used in the execution of its (potentially many) component threads.

Scheduling within the USC multi-threaded models is simplc: a code-block activation executes until such time as either all its component threads have finished, or all are blocked awaiting a value from another activation. Each activation's frame contains reserved fields which are used for synchronization with child activations — when a child activation finishes, it is obliged to write a value into the appropriate field of its parent activation to inform the parent of its completion. This synchronization mechanism is reminiscent of TAM's frame-based synchronization and AMAM's direct key. As is the case in AMAM, use of the USC system for implementing aggregate-based computation has lead to the adoption of a counter-based synchronization (similar to AMAM's multiple-direct key) to efficiently cater to the joining of multiple child activations.

The systems which have been developed at USC have principally been used to evaluate novel alternatives for multi-threaded execution. Consideration has been made of possibilities for providing nodal caches [74, 43] into which off-node values, gathered by an expensive split-phase fetch, can be stored to reduce the cost of future access. In considering the implementation of aggregate-level operations which have single-assignment semantics, such caching mechanisms can be provided at extremely low cost. A second experimental USC prototype has considered an execution model in which threads are permitted to relocate from node to node in order to satisfy data dependencies. These nomadic threads [61] eliminate the need for expensive fetching of data (usually by split-phase) typically found in thread computations which are owner-computes .

While each of these extensions to the multi-threaded model is far beyond what is presently supported in AMAM, the ideas they embody are compatible with our model. Thus, one possible future direction for the AMAM (see Section 9.1.1) could be the consideration of such functionality.

6.3.5 Multi-Threaded Hardware Architectures

While this discussion of architectures for multi-threaded execution has thusfar limited itself to those solutions which (like AMAM) offer a software-based realization of multi-threading, it is worthwhile mentioning that recent research in hardware architectures has also contributed a number of environments with similar features. A survey of the broad field of hardware multi-threading is beyond the scope of this thesis, however for purposes of comparison we provide a brief overview of two such architectures. Section 9.1.1 provides a brief discussion of implementing the AMAM on such machines.

*T

The MIT *T [87] (pronounced "start") is scalable (distributed memory) machine based around a collection of customized RISC processors attached to a high-performance fat-tree network. The processors are Motorola 88110 microprocessors augmented with custom hardware (the *Message Synchronization Unit (MSU)*) providing functionality for fine-grained communication and synchronization. This unit contains a stack of microthreads, each the activation of a non-blocking thread. Very low cost hardware switching is provided within the MSU to schedule these micro-threads for execution by the CPU.

Communication across the network is provided in the style of active messages, with each message consisting of a continuation (i.e., a micro-thread reference) and one or more values. This style of nodal interaction is reminiscent of that offered by TAM, P-RISC and the remote activation facilities of the AMAM. As in these systems the focus is on modelling high-latency access as split-phase. Receipt of a message by the MSU leads to the execution of a handler thread (similar to a TAM inlet) which copies data into the appropriate frame. Whenever a *T node encounters the end of the micro-thread currently being executed, it may elect to schedule the micro-thread for such a waiting message rather than the default action of popping the local stack of micro-threads.

Later versions of the *T architecture (*T-NG [27] and *T-Voyager [9]) explore the addition of globally coherent nodal caches to reduce the frequency of communications introduced by off-node memory references.

To date the *T family of architectures has been used as a basis for implementations of a number of message passing libraries including Cilk [23, 62] and MPI [80]. Future work will address the issue of compiling implicit and explicit parallel languages to these machines.

Tera

The multi-threaded hardware architecture [8] developed by Tera is a shared memory system, where each of the processor nodes in a three-dimensional mesh network has hardware support for up to 128 concurrent threads. Each active thread is granted a status word (which includes its instruction pointer) and a set of dedicated registers. Inter-thread switching is very efficient with a switch-per-machine-instruction being a viable execution paradigm. Each instruction includes a 3-bit field indicating the (minimum) number of further instructions that can be executed from the current thread before a dependency arises which forces a context switch. This information is used to determine the actual grain size of the thread segments which are executed: there is a hard upper limit of 8 instructions, indicative of the architecture's intended purpose as a basis for fine-grained multi-threading.

Synchronization between Tera threads is described in terms of memory-based operations. Every word of memory has an associated 4 bits of *access state* of which one bit represents presence (in analogy with AMAM's direct key and classical I-structures).

The current multi-threaded multiprocessor architecture offered by Tera are programmed using Fortran, C and C++ (which is automatically parallelized by loop-analysis) into which explicit parallelism can be added by way of pragmas. These languages are sufficient for the expression of some irregular forms parallelism (such as that present in irregular mesh computations); the low-cost synchronization and latency hiding offered by the Tera architecture offer good opportunities for exploiting such parallelism.

6.4 Ω-Threads in AMAM

As presented, the interface and functionality offered by AMAM are close parallels of those central to our definition of the abstract machine Ω. Thus it is unsurprising that the threads and programs we presented previously for Ω can be translated reasonably directly (excepting the need to split Ω threads to make them non-blocking, as explained in Section 6.1.1) into AMAM source. To demonstrate this process, we present AMAM translations of two Ω-threads, the remote vector dereference and the nestable generic Data-Parallel `map`.

6.4.1 Vector Dereference

Figure 39 shows the AMAM form of the vector dereference operator described for Ω in Figure 20. The definition consists of two non-blocking threads `vector_deref` and `vector_deref_0`. The first serves as a vehicle for verifying that the vector for which dereferencing has been requested has been entered into the local mapping table. As discussed in Section 6.2.2, this check is a form of synchronization by name: if the mapping exists already then this implies that the operation upon this node which created the vector is complete. If the mapping is absent at the time that the `vector_deref` activation is executed, this implies the creator activation is still incomplete, hence we must suspend (on a direct key) until such time as we are informed by the system that the creation is complete.

Once the activation of `vector_deref_0` has been scheduled, we know that the mapping for the requested vector must be present within the local mapping table. Thus it is safe to dereference any local index of the vector. Using the vector's relative location function (`state_ptr->v.rl`) we compute which slot corresponds to the index we wish to retrieve. Using the mapping table and some simple pointer arithmetic this enables us to find the heap address representing the desired index. If the calling activation resides on the same node as this activation (i.e., `h == AMAM_me()`) then the result can be passed back by a simple entry into the local result pool. Otherwise, we must use communicate the value using `AMAM_result`.

6.4.2 Nestable Data-Parallel MAP

The AMAM implementation we provide for the nestable generic DP `map` is a direct (non-blocking thread) translation of the Ω-thread defined in Figure 23 with modifications introduced to take advantage of the optimizations discussed in Section 6.2. Figures 41, 42 and 43 give full definitions of the six threads which collectively implement the `map`. Figure 40 summarizes the possible paths an activation of the entry thread `map` may follow between these various threads before termination.

Fields of vector_deref State

v.id	identity of vector to be dereferenced
v.len	length of vector to be dereferenced
v.rl	vector's relative location function
idx	index to be dereferenced
h	id of node who requested the index value
dkey	direct key upon which caller suspended

Thread Definitions

```
void vector_deref (sched_id, state_ptr, dummy)
{
  void *key;

  key = AMAM_verify_mapping (state_ptr->v.id);

  if (key == NULL)
  {
    AMAM_enter_activation (vector_deref_0);
    AMAM_terminate_activation (sched_id);
  }
  else
    AMAM_suspend_activation_d (sched_id,key,vector_deref_0);
}


void vector_deref_0 (sched_id, state_ptr, dummy)
{
  void *buf; int slot_no;

  slot_no = state_ptr->v.rl (state_ptr->idx,
                             state_ptr->v.len);
  buf = AMAM_get_mapping (state_ptr->v.id,slot_no);

  if (state_ptr->h != AMAM_me())
    AMAM_result_d (state_ptr->h,state_ptr->dkey,buf);
  else
    AMAM_enter_result_d (state_ptr->dkey,buf);

  AMAM_terminate_task (sched_id);
}
```

Figure 39. AMAM Implementation of Remote Vector Dereferencing

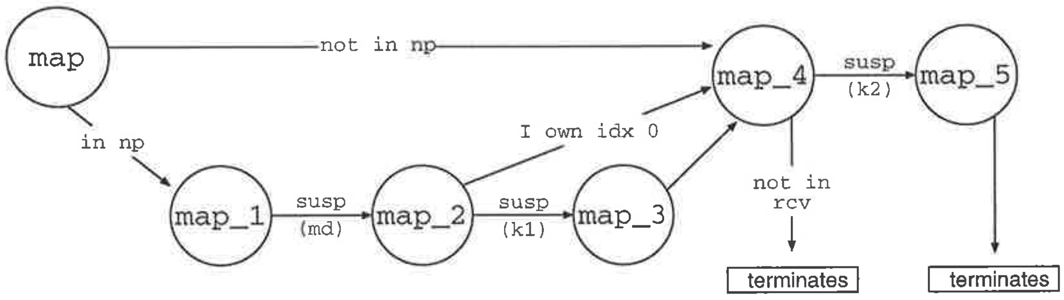


Figure 40. Possible Execution Paths for a `map` Activation

The state object (which shall be denoted `s`) for the `map` thread and its various continuations is shown at the top of Figure 41. This list describes two types of state fields, those which must be supplied by the activation entering the `map` activation (i.e., the initial state) and those which are filled in the course of the computation. In the figure, the former are shown in the top section of the box, the latter at the bottom.

The entry thread into the co-operative computation, `map`, is activated on each of the nodes in the set `s->p` by an activation which thereafter suspends on the direct key passed as initial state (i.e., `s->dkey`). This entering activation is resumed once the `map` has been completely computed. At that time, it will receive a vector descriptor denoting the result of the DP operation.

The first operation performed by each activation of the `map` thread is the calculation of which hosts are in the *participant set*, `s->newp`, of the computation (line `5`). These are the nodes which will co-operatively perform the map operation: by the owner computes rule we determine the set to be all hosts which own a portion of the vector. An optimized computation of this set is offered by the `AMAM_range_pfun` system call (described in Section 6.2.3) which computes the range of a partitioning function for a particular domain (the set of indices of the input vector). Once the participant set has been determined, it is possible to determine, as we did in the Ω implementation of **MAP**, which hosts need to receive messages to satisfy the computation's activation constraint. In Figure 41 this set is called `snooze` (computed in line `6`); we implement the process of making remote activations by using a protocol wherein each `snooze` member receives a message from the member of `s->p` whose id is "closest" to it. A `for` loop (lines `9`–`12`) is used to implement a protocol of activation

— this corresponds to an expression of the set operation γ (see Section 5.1.6) for a particular metric.

Following this work to satisfy the activation constraint, the `map` thread offers two paths down which its activation may proceed. If an activation is a participant in the co-operative computation for the `map` (i.e., it executes on a node whose id is in `s->newp`) then it passes its state to a new activation of `map_1` and terminates (line `15`). Activations which are not participants in the `map` (i.e., node id is not in `s->newp`) pass their state instead to an activation of `map_4` (line `17`). This conditional activation succinctly expresses the fact that these two sets of activations will perform quite different roles in the ensuing computation.

Activations which reach `map_1` have the responsibility of computing the result value of the `map` for the portion of the input vector resident on a single node (the node executing the activation). They begin by computing the subset of that vector's indices which are locally resident — this is achieved by evaluating a (partial) inverse of the partitioning function (line `25`). As discussed in Section 6.2.4 we provide a system call, `AMAM_invert_pf`, to efficiently perform such inverse calculations. Our primary need for determining the set of local indices is to compute the storage requirement for the local indices of the `map` result. We know, from the initial state field `s->slot_size` passed at the beginning of the `map`, how much memory is required to hold a single index. This coupled with the cardinality of the set computed by inverting the partitioning function, yields the total memory requirement for the local section of the result vector. This amount of memory is allocated from the local heap and a pointer to its base stored in the state field `s->block` (line `27`).

Following the allocation of memory to store the result indices, the activations of `map_1` proceed to activate instances of the per-element computation function `s->f` (lines `29`–`37`). One such activation is (locally) entered for every local index of the input. The initial state for each function activation includes a *context* field which holds the slot address of the index in question, an *argument* field containing that index's value, and a singleton *caller's participant set* field. Each activation also contains a reference to a single multiple direct key (created with a count equal to the number of local indices of the input). Once all `s->count` function activations have been made, the `map_1` activation suspends upon this key (line `38`).

We assume a basic model for the execution of the function thread `s->f` in which it accepts an initial state with the fields described above, and (after whatever scalar or

Fields of `map` State

v.id	identity of input vector
v.len	input vector length
v.pf	input vector partitioning function
v.rl	input vector relative location function
f	function argument to the `map`
slot_size	storage requirement for each result index
p	set of nodes which need `map` result
dkey	direct key upon which caller suspended
count	number of indices local to this node
newp	participant set of the `map`
block	pointer to storage for local result indices
md	the multiple direct key
vecid	id of vector created during the `map`
result_v	descriptor for the result vector

Thread Definitions

```
void map (sched_id, s, dummy)
{
  pset snooze; int i,t;

  s->newp = AMAM_range_pfun (s->v.part,s->v.len);
  snooze = AMAM_and_pset (s->newp, AMAM_not_pset(s->p));

  t = AMAM_seq_pset (s->p,0);
  for (i=0;i<AMAM_num_hosts();i++)
   if (AMAM_in_pset(s->p,i)) t = i;
   else if (AMAM_in_pset(snooze,i) && AMAM_me() == t)
     AMAM_request_any (i,map,s);

 if (AMAM_in_pset(s->newp,AMAM_me()))
   AMAM_enter_new_activation (map_1,s);
 else
   AMAM_enter_new_activation (map_4,s);

 AMAM_terminate_activation (sched_id);
}
```

Figure 41. AMAM Implementation of NDP `map`, Part One

thread-based work is required) terminates after having written a value to the direct key it was passed. This value must be of the form *(context, result value)*, where *context* is the same context from the initial state (`u->ctx`) and *result value* is the return value of the function.

An activation reaches `map_2` after all of its child function activations have completed and written their results to the multiple-direct key (thus reducing the key's count to 0). Instead of receiving a single result from the scheduler, activations of this thread receive an array of results, one for each of completed function applications. Each entry of this array is a *(context, result value)* pair, the context being the slot address of the index which was passed to that particular function instance. Since the result vector is partitioned identically to the input vector, this context is also the slot address in the result to which the associated result value should be written. The loop at the beginning of `map_2` (lines 44–48) cycles through the entire result array copying the results into the appropriate slots of the block allocated for the result.

Following the writing of results into `s->block`, the process of evaluating the `map` continues with the determination of a global name for the result vector. The implementation we offer here adopts a simple protocol: the node which owns the first index of the vector is permitted to generate an identity for the vector, which is then communicated to all other nodes in `s->newp` (lines 52–55). Once this activation has completed its broadcast, it has completed its role in the co-operative computation and can thus generate a result vector descriptor (lines 57–58) and make an entry into its node's mapping table (line 59). Following this, the activation passes control to `map_4`.

The `map_2` activations on nodes other than the designated name generator suspend upon a key `k1` until such time as the vector identity has been received (line 64). At this time (when the continuation `map_3` has been scheduled), these activations too may safely generate a return vector descriptor and report the new vector to the local mapping table (lines 69–71). These activations then proceed to `map_4`.

The thread `map_4`, the joining point of all possible paths through the series of threads, implements a protocol of communications to satisfy any return value constraints (as described in Section 5.2.1) for the DP operation. This involves computing the set `rcv` of nodes which have expressed a desire to be informed of the `map` result (i.e., are in `s->p`) but were not participants in the co-operative computation of that result. This set is computed by line 79 of Figure 43. Activations in `rcv` suspend

```
void map_1 (sched_id, s, dummy)
{
  set_of_int idx_temp; int count;

  idx_temp = AMAM_invert_pf (s->v.pf,s->v.len);
  s->count = cardinality (idx_temp);
  s->block = allocate (s->count*(s->slot_size));
  s->md = AMAM_new_multi_direct_key(s->count);
  for (i=0;i<(s->count);i++)
  {
    ... allocate space for a function state u ...
    u->ctx = i;
    u->arg = AMAM_get_mapping (s->v.id,i);
    u->p = AMAM_single_pset (AMAM_me());
    u->dkey = s->md;
    AMAM_enter_new_activation (s->f,u);
  }
  AMAM_suspend_task_md (sched_id,s->md,map_2);
}
void map_2 (sched_id, s, result_array)
{
  func_result *r; host_id h; int i;

  for (i=0;i<s->count;i++)
  {
    r = (func_result *) (tmp[i]);
    copy (state->block+(r->ctx)*s->slot_size,r->val);
  }
  h = s->v.pf (0,s->v.len);
  if (h == AMAM_me)
  {
    s->vecid = AMAM_new_id ();
    for (i=0;i<AMAM_num_hosts();i++)
      if (AMAM_in_pset (i,s->p))
        AMAM_result (i,k1,s->vecid);

    s->result_v = make_vector_descriptor (s->vecid,s->v.pf,
                                          s->v.rl,s->v.len);
    AMAM_enter_mapping (s->vecid,s->block);
    AMAM_enter_new_activation (map_4,s);
    AMAM_terminate_activation (sched_id);
  }
  else
    AMAM_suspend_activation (sched_id,k1,map_3)
}
```

Figure 42. AMAM Implementation of NDP map, Part Two

on key `k2` in wait of the result value, which will be forwarded to them by one of the participants. We use a protocol similar to that presented previously (lines `9`–`12`) for activation to determine which participant satisfies which suspensions (lines `84`–`88`).

Once any required messages have been sent, participant activations in `s->newp` terminate (line `92`). If the activation was also in `s->p`, and thus desired the result value, the result value vector descriptor is written to the direct key upon which the activation initiating the `map` is suspended (line `91`).

Activations which receive the final `map` result via communication (thus reaching `map_5`) similarly write this value to the caller's suspension key and terminate.

6.5 Summary of the Architecture

In this chapter we have described a concrete implementation, the AMAM, of the threaded abstract machine Ω defined and analyzed in Chapter 5. The realization of the multi-threaded execution model on a real-world distributed memory multiprocessor (the Thinking Machines CM-5) is embodied in a library of C data structures and routines. The former provide implementations of the nodal structures found in Ω, while the latter form a suite of system calls which provide the functionality of the Ω instruction set.

To allow for efficient scheduling of threads, we choose to limit AMAM's threads in a way that ensures that they are *non-blocking*; that is, they do not suspend mid-execution. While this innovation complicates the (otherwise trivial) translation of Ω codes into AMAM, the reduced context switching overheads involved in such a system make the system considerably more efficient. Further improvements are gained by introducing a system of prioritized scheduling, in which thread activations which can cheaply be identified as schedulable have priority over those whose schedulability may be costly to determine (e.g., activations which are suspended).

A principal feature of the AMAM is its hybrid model of synchronization keys. In considering our abstract machine Ω we were not concerned with the nature of the keys used within inter-thread synchronization and communication. In our implementation such issues become important: we want to provide a mechanism which allows for very general forms of interaction to be possible, but we also wish to minimize the cost of synchronization which uses our keys. To a certain extent these goals are mutually exclusive. In AMAM, we provide an environment in which two forms of

```
void map_3 (sched_id, s, result)
{
  s->vecid = result;
  s->result_v = make_vector_descriptor (s->vecid,s->v.pf,
                                        s->v.rl,s->v.len);
  AMAM_enter_mapping (result,s->block);
  AMAM_enter_new_activation (map_4,s);
  AMAM_terminate_activation (sched_id);
}


void map_4 (sched_id, s, dummy)
{
  pset rcv; int i,t;

  rcv = AMAM_and_pset (s->p, AMAM_not_pset(s->newp));
  if (AMAM_in_pset (rcv,AMAM_me()))
    AMAM_suspend_activation (sched_id,k2,map_5);
  else
  {
    t = AMAM_seq_pset (s->p,0);
    for (i=0;i<AMAM_num_hosts();i++)
      if (AMAM_in_pset(s->p,i)) t = i;
      else if (AMAM_in_pset(rcv,i) && AMAM_me() == t)
        AMAM_result (i,k2,s->result_v)

    if (AMAM_in_pset (s->p,AMAM_me()))
      AMAM_enter_result_d (s->dkey,s->result_v);
    AMAM_terminate_activation (sched_id);
  }
}


void map_5 (sched_id,s,result)
{
  AMAM_enter_result_d (s->dkey,result);
  AMAM_terminate_activation (sched_id);
}
```

Figure 43. AMAM Implementation of NDP `map`, Part Three

synchronization are possible — synchronization by *logical key* and synchronization by *direct key*. The former can support completely general patterns of interaction, but incurs a potentially substantial matching cost; the direct key approach is less costly in matching, but only allows for interaction between thread activations which have some common dynamic parent.

By providing both methods of synchronization, AMAM allows the programmer (or compiler) to make use of the most efficient method (direct keys) where they are sufficiently expressive, but also have available a more general (albeit less efficient) mechanism when such is required. Experiments with real DP thread programming (see Section 7.3.4) suggests that at least 60% of the synchronizations within such programs may be satisfied by direct keys.

We describe a number of special features which have been added to the AMAM to optimize the execution of threaded NDP codes. These include a form of counter-based synchronization to minimize the overhead of implementing operations, common in DP processing, in which many threads are gathered together at a single join. Another significant augmentation is a presence-based method of synchronizing DP operations in which an I-structure-like protocol is introduced for AMAM's partitioned vectors to guarantee that read-before-write access is disallowed. The use of this facility eliminates the need for synchronization barriers to be present in the implementation of DP operators.

The utility of the AMAM environment is demonstrated by the construction of executable forms for two of the Ω threads defined in the previous chapter: the vector dereference thread and the generic, nestable `map` thread. The close correlation between the AMAM forms and the abstract definitions from which they are derived, highlights the ease with which such translation may be made.

As we shall see in Chapter 7, we can make use of such concrete AMAM implementations of our earlier generic thread definitions as the basis for the convenient and efficient implementation of a fully-featured NDP language.

Chapter 7

Case Study: Implementing a Simple Nested Data-Parallel Language on the AMAM

The multi-threading library AMAM presented in the previous chapter represents a full realization of an environment to support irregular Data-Parallel modes of computation. In the descriptions of the various thread-forms of DP operations we have already made some definition concerning how such computations may be expressed in terms of this multi-threading model. This chapter extends this discussion, detailing techniques and methodologies for mapping a simple yet fully-featured first-order functional NDP language to the AMAM. The language we choose, called Adl, is loosely based on a subset of the SISAL language and bears semantic similarities to NESL. Adl's simplicity means that the basic framework of a compiler may be constructed with a minimal amount of effort, allowing a focus to be placed upon techniques for producing optimized forms for the language's (very general) parallel structures.

We present a brief overview of the Adl language, focussing our attention on its generic Data-Parallel features and facilities for nesting such operations. This is followed by a detailed description of a runtime support system for the language and its construction upon the AMAM. We note that many of the more complex details of this runtime environment (such as the provision of threaded implementations of Adl's various DP operations) can be directly constructed from the generic forms (abstract and concrete) we have already investigated for describing such semantics in terms of our models. Finally we discuss the process of compiling the Adl language into AMAM threads ready for linking with the libraries for the AMAM implementation and the previously constructed runtime environment to build executable forms.

7.1 An Overview of the Language Adl

Adl [4, 98, 6] is a small strict functional language in the style of SISAL. It features a simple type system centering upon three base types (integer, boolean and real) from which more complex types may be derived using two type constructors: the *n-ary tuple* and the *one-dimensional vector*. The former represent the style of untagged structure types common in functional languages, while the latter represents a homogeneously-typed grouping of data elements. Each of these constructors is applicable to values of any type of the language, thus it is possible to specify vectors of vectors, tuples containing vectors, vectors of tuples, and so on. In this fashion arbitrary complex types can be built.

7.1.1 Program Structure

An Adl program consists of a set of function definitions, one of which is distinguished as the *main function* or computational entry-point. Syntactically, a function is declared using a binding operator `:=` as follows:

function_name formal_argument `:=` *function_body_expression*

Function application is represented in the syntax by the specification of the function name followed by the actual argument.

Each function notionally accepts only a single argument, although this may be a tuple-typed value; the language provides structural pattern matching to project elements from the argument. For example, consider the function:

```
add2 (a,b) := a+b
```

This function may be invoked with a single tuple-valued actual parameter, e.g., `(2,3)`. In this case, the structural pattern matching of Adl would establish the associations `a := 2, b := 3` within the execution of `add2`.

User functions within an Adl program are exclusively first-order — the type system does not permit the passing of functions as parameters. A number of language-defined combinators (`while`,`map`,`scan` and `reduce`) are exceptions to this rule.

Adl functions may not contain recursive function invocations. We choose to impose this limitation to encourage an exclusively DP style of programming — much of what is typically expressed in programs via recursion can equally well be cast in terms of Adl's DP operators (described in Section 7.1.3). A DP expression of such computations, although perhaps slightly unfamiliar to most programmers, has the advantage that its parallelism is explicit and easily exploited by a compiler. Furthermore optimization of such non-recursive forms is considerably less complex (this is particularly important for equational approaches to optimization, such as the scheme presented in [6, 5]). The Adl implementation described in this chapter makes some use of the languages non-recursive nature (principally to simplify the protocol of synchronization). However, Section 7.3.4 offers discussion of a simple modification which would allow for recursive functions, thus making the approach practical for the implementation of more general NDP languages.

Adl supports a limited form of universal quantification, called *syntactic polymorphism*, for its functions. In most cases, it is not required for the programmer to specify the type of the argument or return value of a declared function: the system assumes the most general type for such declarations. The exception to this rule is the declaration of the main function for which a concrete argument type must be supplied. This limitation allows for the static determination (by propagating type information through the program) of all types within the program.

A function within the language may introduce values into its inner scope by means of the `let` expression. The syntax is as follows:

`let` *declarations* `in` *result_expression* `endlet`

All declarations made in the first arm of the `let` are separated by a semi-colon, each representing an addition to the local scope of the expression. Declarations may be made with names identical to those in outer scopes, in which case the outer binding is occluded within the scope of the `let` by the inner binding. Once all the bindings specified in the first section of the `let` expression have been introduced, the expression in the second arm is evaluated in the context of this inner scope (and any levels of scope containing the `let`) to arrive at the final value of the `let` expression itself.

7.1.2 Expressions in the Language

Adl defines the standard set of scalar numeric operations (`+`, `-`, `*`, `/`, `^` (power), `\` (integer division), and `mod`) for use within its expressions as predefined infix (type-overloaded) functions. Similarly present are the boolean operations not (`~`), and (`&`), or (`|`) and the typical relational operators (`==`, `~=`, `<`, `<=`, `>`, `>=`). Common transcendental functions (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`) are also supplied and manifest as pre-defined functions using the normal syntax for function application.

Scalar literals may appear in the program, taking the form of either numerics or the boolean literals `true` and `false`. Tuple literals may be specified as a parenthesized list of expressions of any type. Literal vectors are denoted by a list of expressions, all of identical type, surrounded by square brackets.

Conditional evaluation is manifested in the language by means of two pre-defined functions `if` and `while`. The former appears within the program as the following syntactic elements:

`if` *predicate_expr* `then` *expr_1* `else` *expr_2* `endif`

The semantics of this construct are as follows: first the (boolean) predicate expression is evaluated. Based upon the value obtained from this evaluation, either *expr_1* (if predicate evaluated true) or *expr_2* (if predicate evaluated false) is calculated. This represents a non-strict computation; this is the only such exception to the language's rule of strict evaluation.

The `while` operator provides a means of introducing unbounded iteration into an Adl program. It appears within a program as an application of a second-order pre-defined function:

`while` (*iterator_fn*, *predicate_fn*, *initial_value*)

The type rules of the language stipulate that if the initial value if of type T then the iterator must be a unary function from T to T and the predicate must be of type T to `bool`. The value of the `while` is obtained by repeated application of the iterator function. The first such application is made to the initial value supplied; subsequent ones are made to the value derived from the previous application. This process continues until such time as a value is produced which, when passed to the predicate, yields `false`. This final value is returned as the result of the operation.

Adl provides a number of in-built operations which work across vectors of values.

These include the language's Data-Parallel primitives (discussed in the next Section) as well as the operations `#`, `!` and `iota`. The first of these functions, `#`, is a unary prefix operator which returns the length of its vector argument. The `!` is an infix primitive which represents vector dereference; `v!0` refers to the value stored at the first index of the vector `v`, `v!1` to the value at the next index, and so on. The function `iota` is a simple dynamic vector constructor. Given an integer, this primitive constructs a vector of this length filling the i'th index of this new vector with the integer value i. That is, the value of `iota n` is the vector `[0,1,2,3,...,n-1]`.

7.1.3 Data Parallel Operations

DP operations appear within the language as second-order pre-defined functions acting across vectors of values. Three such operations are defined in Adl: `map`, `reduce` and `scan`.

The `map` operator takes two arguments; a unary function f and a vector v of values, and produces a new vector of equal length to v and with index values derived from applying f to the corresponding index of v. That is, if we denote the result vector as v', then

$$v'[i] = f(v[i]) \quad \forall i = 0, \ldots, \#v - 1.$$

Adl's DP `reduce` operator accepts three arguments: a binary function f, an input vector v and a starting value a. From these it computes a single value obtained by accumulating all indices of v plus the value a using the function f. That is, adopting an infix notation for f, we calculate

$$v[0](f)v[1](f)\ldots(f)v[\#v-1](f)a.$$

We assume associativity of the function f.

The last of the language's DP operators, `scan`, implements an arbitrary parallel prefix across a vector of values. Given a binary function f, an input vector v and a starting value a, this operator generates a new vector v' whose indices contain partial accumulation (via f) of v. That is, the i'th index of the result contains the accumulation (via f) of a and all indices up to and including i. That is, the result of a `scan` across vector v using function f and starting value a is the vector:

$$\left[\quad v[0](f)a, \quad v[1](f)v[0](f)a, \quad \ldots \quad , v[\#v-1](f)v[\#v-2](f)\ldots(f)v[0](f)a \quad\right]$$

A Simple NDP Program in Adl

To illustrate how these simple DP primitives can be combined to specify nested DP computations across irregular data structures, consider the example Adl program in Figure 44. The purpose of this program is to add two ragged array structures (i.e., vectors of vectors), `a` and `b` which are of identical shape (each inner vector is of equal length). The program begins (in line 9) by creating a temporary vector using the iota combinator. The values of this vector are the full set of valid indices of the outer vector of the nest a. That is, if we consider a run of the program (pictured in Figure 45) where `a` — and hence `b` — is a vector of 3 inner vectors, then the temporary vector created at line 9 is $[0, 1, 2]$.

```
1     main (a:vof vof int, b:vof vof int) :=
2       let
3         f i := let
4                  g j := a!i!j + b!i!j
5                in
6                  map (g,iota #(a!i))
7                endlet
8       in
9         map (f,iota #a)
10      endlet
```

Figure 44. An Example NDP Adl Program

For each index of the temporary vector, the function `f` is activated (in parallel). Each of these instances is responsible for adding one inner vector of the nest `a` to a corresponding inner vector of the nest `b`. The argument passed to an activation of `f` defines which of these additions it is to perform: the instance with argument `i=0` is responsible for adding the first pair of inner vectors, that with argument `i=1` considers the second pair, and so on. Within the function `f`, the first operation is to again create a temporary vector using `iota`, this time containing the indices of the inner vector under consideration.

Returning to our sample run (Figure 45) we see that each of the three instance of `f` has used the `iota` constructor to create a vector which holds the indices of the inner vector under consideration.

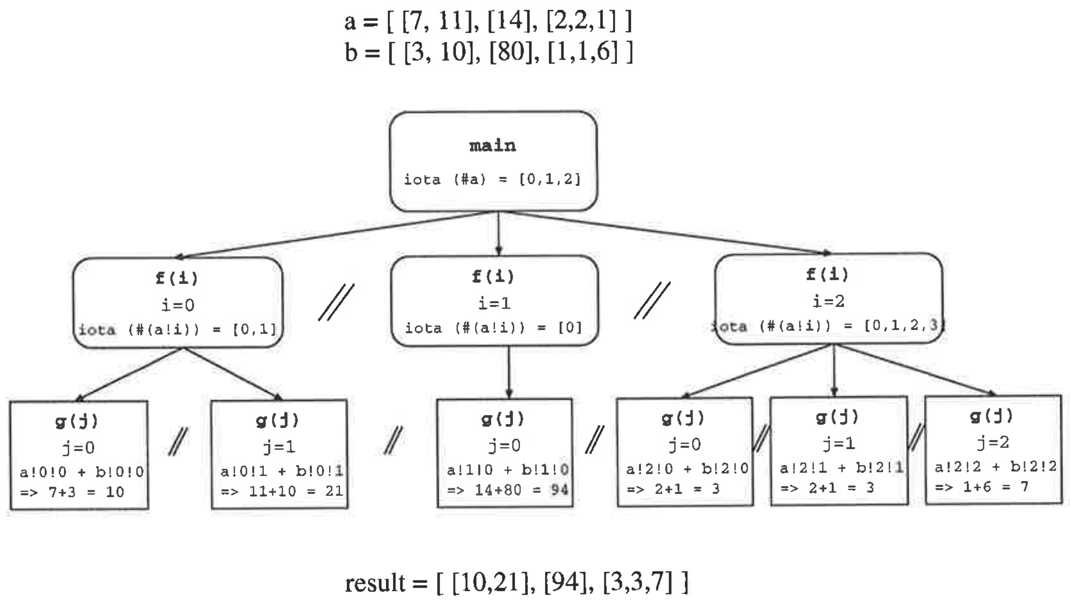


Figure 45. An Evaluation of the Example Adl Program

Each instance of the function `f` uses its temporary vector as the basis for an inner DP operation — a parallel application of the function `g` to each vector index. Each instance of `g` is charged with the task of adding one index from an inner vector of `a` (i.e., a number) to the index of `b` in an identical *position* within the nest. We specify such position as a pair of indices: an index into the outer vector, plus an index into the appropriate inner vector. Within the function `g` of the sample program, we use the argument `j` to specify the latter, while the former is defined by the argument (`i`) to the instance of the function *f* which invoked this instance of `g`. To retrieve the latter, we must make use of Adl's scope rules. Once outer and inner indices are known (and hence a position has been identified), each instance of `g` may perform its addition (line 4) by a process of dereferencing both `a` and `b` with these outer and inner indices.

If we continue the concrete example from Figure 45, we can consider the instance of `f` which was passed the argument 0 to signify that it was responsible for adding the first inner vectors of `a` and `b`. We saw earlier that this function creates a temporary vector $[0, 1]$ and applies `g` to each index. We can consider the two instance of `g` spawned by this instance of `f`: one will receive the argument 0 to signify that it should make an addition of elements at position (`i`=0, `j`=0) by using these indices to dereference both `a` and `b`. The second instance of `g` will concentrate on position (`i`=1, `j`=0).

Ultimately, each of the instances of `g` which have been spawned by the `map` of line `6` will return a value to that operator. These are combined to produce a vector whose values are the pairwise sums of the inner vectors of `a` and `b` for which this instance of `f` was made responsible. In turn, the instance of `f` return these vectors to the `map` at line `9`, where they are combined to produce a nest where the n'th inner vector is the pairwise sum of the n'th inner vectors of `a` and `b`. This nest becomes the result of the program.

7.2 Building an Execution Environment for Adl

The remainder of this chapter describes an approach to constructing a Distributed-Memory parallel implementation of the language Adl [37]. We make use of the approach to NDP offered in previous chapters, exploiting parallelism introduced by the partitioning of Adl's vectors across memory spaces of the machine. Our implementation incorporates all of the various threads — those which implement Adl's DP constructs as well as all those auxiliary to such definitions — specified in previous discussion into a single execution environment which rests atop the AMAM. This environment is represented in real terms as a library of AMAM thread definitions which form a runtime system for Adl (which we call the ADLRTS).

Once we have built such an environment, our task of mapping Adl to a distributed parallel executable becomes one of simple compilation from source to a skeletal AMAM code. Threads from this object code directly implement the scalar operations of the program, making calls to the ADLRTS to accomplish the non-scalar sections. Since the complexities of message passing, participant sets, and various other factors related to parallel thread-based DP execution are encapsulated within the thread definitions of the library, the compiler itself does not need to consider such aspects of the computation. This represents a major simplification in the compilation process.

This section discusses aspects of the Adl execution environment (ADLRTS); the next describes an Adl compiler which targets this system.

7.2.1 A Thread Library to Support NDP

Our approach to implementing Adl centres around the exploitation of the explicit parallelism introduced by the DP operations (involving `map`, `reduce` and `scan`) present within a program. We do not attempt to make use of implicit parallelism (e.g., expression-level parallelism) afforded by the functional nature of the language.

A consequence of this implementation strategy is that it is possible to clearly identify the sections of the Adl program which exhibit parallel behaviour. That is, it is possible to define a small set of *parallel constructs* which embody the only opportunities for parallel execution within a program. This set includes the DP operators, the indexing operator (`!`) and vector creation operators (`iota` and the vector literal constructor). This clear-cut definition of the parallel aspects of the language suggests a particular convenient structure for the Adl language system: a compiler which can generate thread code for the sequential (scalar) parts of the program, coupled with a library of parallel routines (or threads) which may be called (activated) to implement the non-scalar sections of the program. The remainder of this chapter describes an implementation adhering to this structure. The discussion below focusses on the construction of a generic library for the language's parallel features; Section 7.3 offers a full description of a compiler which makes use of these services to provide a complete implementation of the language.

The Basis for a Generic Parallel Library

We have already noted in our description of DP threads (Sections 5.2 and 6.4) that the thread environment of Ω and AMAM allows for very general parallel operators to be constructed. Indeed, we have demonstrated how both machines may be used to specify parallel operations which are generic upon both an elemental function and the partitioning of a data aggregate accepted as argument. Such generic descriptions are precisely what is required for the construction of a concise library of parallel routines to support the parallel execution of an Adl program.

Our work in earlier chapters provides us with ready hand-coded generic thread forms for most of the language's parallel constructs. We can translate these forms into AMAM (Section 6.1.1 describes the straight-forward derivation of AMAM forms for Ω threads) to supply the bulk of this runtime library. To further ease the compilation process we also add three structured scalar operations (`while`, `read` and `write`) to the library. Table 4 shows the full set of ADLRTS operations, each listed with the initial state elements upon which they are parametric. In the sections which follow we briefly describe each of these runtime library threads. We first consider those threads whose functionality pertains to serial operations in the program, following this with a discussion of threads which define basic operations on Adl's partitioned vectors. Finally we describe the generic library threads which implement the language's nestable DP operators.

Thread Name	**State Arguments**
`ADLRTS_while`	`thread *func, thread *pred, void *start_val, void *funcs_link, void *preds_link, void *dkey`
`ADLRTS_read`	`void *store, char *type_descriptor, void *dkey`
`ADLRTS_write`	`void *value, char *type_descriptor, void *dkey`
`ADLRTS_deref`	`vector_descriptor v, host_id h, int idx, void *dkey`
`ADLRTS_alloc_vector`	`pset p, host_id (*part_fn)(int,int), int slot_size, int len, log_key k, void *dkey`
`ADLRTS_iota`	`pset p, host_id (*part_fn)(int,int), int len, log_key k, void *dkey`
`ADLRTS_map`	`pset p, vector_descriptor v, thread *func, int slot_size, void *state_link, log_key k, void *dkey`
`ADLRTS_reduce`	`pset p, vector_descriptor v, thread *func, void *start_val, void *state_link, int slot_size, log_key k, void *dkey`
`ADLRTS_scan`	`pset p, vector_descriptor v, thread *func, void *start_val, void *state_link, int new_slot_size, log_key k, void *dkey`

Table 4. Thread Definitions in the ADLRTS

Serial Operations

The `ADLRTS_while` thread directly implements Adl's `while` construct as a dynamically unrolling series of thread activations. The initial state object of the `ADLRTS_while` thread activation should contain pointers to the threads representing the iterator function and the predicate function as well as a pointer to the value which starts the iteration. For each of the two functions, we also require a pointer to the state of its statically enclosing function thread activation so that scope may be modelled in the invocations of the functions. A full discussion of this mechanism is given in Section 7.3.4 of the description of the Adl compiler. Given these initial state items, the runtime system thread alternatively activates (on the local node) instances of the predicate thread and the iterator thread, storing the value returned by the latter in an internal state element. When eventually, a predicate activation returns a `false` value, the current value of this element is written to the direct key (`dkey`) mentioned in the initial state, thus becoming the return value of the `while`.

File system input to an Adl program is supported by inclusion of the `ADLRTS_read` thread in the runtime library. The input operation does not correspond to an explicit construct of the language, but is implicitly specified by the declaration of a main Adl function which accepts arguments to its execution, which we interpret as data supplied from an external input. The `ADLRTS_read` thread has three input fields in its state object: a pointer to the block of memory in which the parsed value is to be stored, a string which defines (in a simple format) what type of item is to be read, and a direct key which is written when the input operation is complete.

In a similar vein, the library provides the `ADLRTS_write` thread for file output. Again, such an operation does not explicitly occur in an Adl program, but is implied by the fact that the main function returns a value. Our interpretation is that this result should be written to an output file or the screen. The state fields for this thread are similar to those discussed for the input operation, with the exception that a pointer to the value to be written is offered rather than a pointer to a block of memory to be filled.

Vector-Based Operations

The ADLRTS library supports vector dereference, the `!` operator, with the `ADLRTS_deref` thread. This is precisely the thread defined in Section 6.4.1. Its input state consists of a descriptor for the vector being dereferenced (which includes data such as its identity and partitioning function), the index requested and the name of the host who wishes to receive the value. The thread first checks to determine whether the vector in question has an entry in the local mapping table (using the `AMAM_verify_mapping` system call, see Section 6.2.2). If the vector has a local mapping, we can safely dereference it and return the result to the designated node, writing it into the direct key `dkey`. If the verification indicated that the vector could not yet be dereferenced, `ADLRTS_deref` suspends until such time as it receives a call-back to indicate that such access is now permissible. Once this occurs it performs the dereference and returns the result to the appropriate node, writing `dkey` as before.

Allocation of a new vector can be occasioned by activating the `ADLRTS_alloc_vector` thread within the runtime library. This thread, derived from the Ω-program in Section 5.2.2 (Figure 22), is never invoked directly from a compiled Adl program but is used by various other threads within the library. Vector allocation is a parallel operation with a participant set equal to the set of nodes which will, eventually, own portions of the new vector. All nodes in this set must enter an activation of the thread for the allocation to complete. The state arguments prevailing in allocation include the partitioning function, length and slot size of the new vector. In communication between the various instances of the thread corresponding to the same allocation, a logical key mechanism is utilized since the various activations have no direct relationship (i.e., no common ancestor activation) and the communication is unsolicited. Because such a protocol is used, it is important that all activation must be passed an identical logical key. Section 7.3.4 below provides some insight into how this is achieved within the current compiler. After the allocation is complete, the thread activations each write a vector descriptor for the newly created vector into the direct key passed in the `dkey` slot of the initial state.

Adl's simple vector allocation operator `iota` is supported directly by a library thread `ADLRTS_iota`. This has effectively the same interface and semantics as described for the uninitialized vector allocation thread described above. Differences lie in the lack of a slot size field in the input (the new vector's base type is always `int`) and the presence of a per-index write operation upon the allocation of each new block of memory.

Data-Parallel Operations

The runtime library threads `ADLRTS_map`, `ADLRTS_reduce`, and `ADLRTS_scan` provide a direct implementation of the equivalently-named DP operations of the language. The definitions of these threads is derived directly from the descriptions we have provided already for AMAM (Section 6.4.2 describes the `map` implementation) and the abstract machine Ω (Section 5.2.4 offers a version of `reduce`, Section 5.2.5 details a DP `scan`; both may be easily translated into AMAM threads according to the guidelines described in Section 6.4).

Each of the threads are generic upon the function to be applied to indices or sets of indices from the input vector: a pointer to the thread implementing this function is passed as initial state. As described above (and further in Section 7.3.4), to achieve the correct modelling of state for activations of these function threads, we also need our initial state to contain a pointer to the state of the function's static parent. In addition to this genericity, the library threads are independent of the partitioning of the data they act upon; we make the descriptor for the input vector, which includes details of its partitioning and relative-location functions, another initial state parameter to the operations. Each activation of a DP thread from the ADLRTS also receives, as part of its initial state, the set of nodes `p` which wish to be informed of the result of the computation. Since all three thread definitions make some use of unsolicited communication, each requires that a logical key be supplied within the initial state; all logical keys for activations co-operating in the same operation must be identical. Section 7.3.4 describes the mechanism for achieving this in the current system. Finally, to communicate a result back to the invoking activation, the initial state of each activation of `ADLRTS_map`, `ADLRTS_reduce` and `ADLRTS_scan` must incorporate a direct key.

7.3 Compiling Adl into Threads

We turn now to the description of a compilation process which translates an Adl source into a collection of AMAM threads which, when combined with the ADLRTS, provide a distributed-memory NDP execution of the program. Our general strategy, as stated previously, relies upon the collection of hand-coded generic threads within our runtime system to implement the various parallel and vector-oriented aspects of the language. Thus our compilation is charged with the production of an ΛMΛM code which implements only the scalar functionality of the source, and which makes activations of runtime library threads as appropriate.

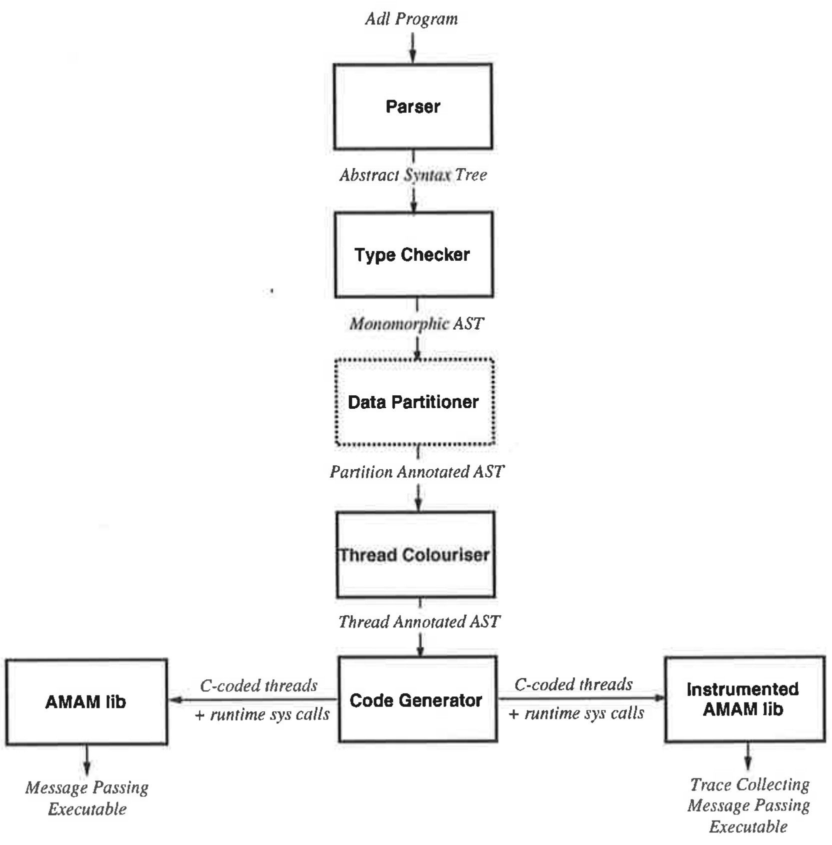


Figure 46. Architecture of the Adl compiler

Figure 46 shows the logical structure of our prototype compiler. The compiler, coded in C to produce a machine independent AMAM object code, consists of a number of functional units, each of which considers a different sub-process of the translation. The sections which follow provide descriptions of the functionality of each and also note the various issues which must be dealt with during the various phases of compilation.

7.3.1 Type Checking and Simple Optimizations

Our first tasks in compiling an Adl program centre around the traditional compiler operations of parsing and type-checking. We derive an abstract syntax tree (AST) representation of the source by means of a simple LL(1) parser. Following this construction, we pass the AST to a type-checking module which walks across the tree, performing two operations. The first is a simple evaluation of type correctness; the second is a process of tree-rewriting to eliminate any polymorphic forms present within the program. As described in Sections 7.1.1 and 7.1.2, user functions and the presence of some overloaded in-built operators may introduce such features. Owing to Adl's limited form of universal quantification, every type within the program may be statically discovered through a process of propagating type information[1] from the top-level function declaration through the various invocations. It is thus possible to determine which sets of types a given function or operator is instantiated with. Given this knowledge we may replace polymorphic function declarations with one or more monomorphic ones (one for each different type instantiation), rewriting calls to the function to point at the appropriate replacement. Similarly, we may replace overloaded operators within the AST with monomorphic forms.

7.3.2 Data Partitioning

In implementing Adl on the AMAM, we identify the Adl vector as a partitioned aggregate over which DP operations can be framed. As described in Section 6.1.2, each such aggregate must be associated with a partitioning function which defines the decomposition of the vector indices across the memory spaces of the machine. Under AMAM's model of execution, such functions are static entities, C routines embedded within the code linked with the AMAM library. Thus, in producing an executable form of an Adl program, we must also consider the generation of such functions and their association with vectors from the program.

This is a critical aspect of the compilation of Adl; the overall performance of a DP program in the language will be sensitive to the placement of data, since this typically determines the volume and nature of the communication patterns present within an AMAM execution of the code.

[1]This process is similar to the instantiation of *parameterized generic domains*, as described in Section 17.3 of [42].

One approach to this specification of data-layout is to implement of an algorithmic method of partitioning function determination and generation. The extensive literature on such schemes (e.g., [46, 105, 67, 92, 47, 89, 72, 56, 65, 66]) makes it clear that generation of an efficient data partition by such means is complex, even when the decision space of partitionings is small (e.g., limited to simple `Block`, `Cyclic` and `Block-Cyclic` decompositions of various sizes). This complexity arises from the fact that the derivation of a data partitioning scheme in which communication is minimal (and thus the overall program cost is minimal) is equivalent to a graph partitioning problem: first, a full graph of the program's data dependencies must be built; then the resultant graph must be partitioned into N groups (one per memory space of the distributed machine) such that the number of inter-group edges is minimized. It has been demonstrated that the generation of such a partitioning is NP-hard [73].

Given the much larger range of partitioning functions available to our Adl compiler (owing to the generality of AMAM partitioning functions), the task of designing an automatic partitioner for this system is likely to be even more problematic than it is for the more restricted cases considered in the literature. Thus, for the present compiler prototype we do not consider the provision of such automatic determination (although the design of such an algorithm is certain worthy of future investigation).

As an alternative, the present prototype Adl compiler requires the programmer to provide a specification of partitioning in the form of a set of program annotations. Given the complexity such specification represents, we choose not to make such annotations as source level (textual) declarations, but rather provide an interactive visual environment where the decompositions can be defined in a more abstract fashion. This X11-based GUI tool, called PFN [38], offers a higher-level view of the design process: a programmer assigns elements of a data aggregate to chosen nodes by gesture in a GUI rather than by writing code, with partitioning functions generated automatically. The operation of PFN is described in detail in Appendix A.

7.3.3 Thread Production

Once an Adl program's AST has been rewritten in monomorphic terms and annotated with descriptions of the data-layout prevailing for each vector, we may begin reasoning about the generation of an AMAM thread program to implement the computation. We approach this code generation problem as a two-phase process:

firstly we determine, by analysis of the AST, how the individual highly-structured computational elements of the Adl program should be split into unstructured threads. This program partitioning information can be considered a further annotation of the AST. Once the entire program has been allocated into a thread the second phase of generation, the actual synthesis of AMAM source, can begin. This section considers the former analytical operation; the actual generation is described in Section 7.3.4.

There are many possible ways in which a given Adl program can be represented in threaded form. The approach we adopt in our current prototype system is a simple syntax-directed scheme in which the division of a program's computation is based exclusively on the position of an expression within the source. No attempt is made to re-order the component expressions of a computation to enhance properties (e.g., length) of the resultant threads. We choose this method purely because of its simplicity: as described in Section 9.1.2, future refinements of the Adl compiler may investigate the application of more complex approaches (such as those found in the macro data flow literature [106, 54, 99]) and the resulting performance benefits.

Within the current compiler prototype, the process of thread production is implemented within the *thread colourizer* module of the system, and involves a two-pass walk across the program AST. During this walk, nodes are *coloured* (annotated) according to which thread of the AMAM object code their implementation should appear within.

The first of the two passes serves to allocate a unique colour to each function within the source program, asserting that all should be represented as an individually activatible computational element in the AMAM output. This is a necessary choice since, as described in Sections 5.2 and 7.3.4 we must make use of AMAM's mechanisms for thread activation to model the notion of function call. Since the prototype compiler does not consider opportunities for function in-lining[2], this means every program function is responsible for contributing (at least) one thread to the object code.

This first pass partitioning does not take into account the possible presence of unbounded latency operations within the various functions of a program. As we

[2]The introduction of such optimization is complicated by our approach to implementing the second-order features of the language as a library of generic routines. Such routines expect a thread (representing a function) as an argument — inlining would involve the elimination of such functions by copying their bodies *into* the generic routine. This is clearly not possible with our current approach.

have seen, the philosophy of AMAM (and of multi-threading in general) is that such operations should be made *split-phase* [11, 12, 32]. That is, the presence of unbounded latency operations (in Adl these include `map`, `reduce`, `scan`, `!`, `iota`, `while`) should define a division in the implementing thread, a voluntary relinquishment of control with an opportunity for later activation of a continuation. Our second colourizing pass performs this sub-division, defining the thread representation of a function containing n unbounded-latency operations as a collection of $n + 1$ threads. The first thread in the series for a function we term its *invocation*, the remainder are called *continuations*. The invocation thread contains the operations of the function up to and including the first unbounded latency operation; here, the invocation thread voluntarily suspends, passing a pointer to the first continuation thread into AMAM. Within the first continuation appear all operations between the first and second unbounded latency operations, followed by a suspension pointing to the second continuation. Thus, each operation is placed into exactly one of the $n + 1$ threads. This process of division is analogous to that used in the translation of Ω threads to AMAM (see Section 6.1.1 for a discussion and Figure 33 for an example demonstrating the resulting thread structure).

Conditionals

Conditionals within an Adl function can complicate this process of division into threads. Consider, for example the fragment of Adl code shown in Figure 47(a). The index operation (`!`) has unbounded latency. One branch of the conditional expression should, by our former scheme, be implemented as a split phase (thread terminating) operation, while the other clearly should not be (since it contains no unbounded latency operations).

Our approach towards handling such cases is to treat each of the possible paths of control as a separate collection of continuations — if one branch of the conditional is taken, control passes through a series of continuations $s_1, s_2, \ldots, s_{n_s}$, while activations pursuing the alternate path proceed on a different continuation series $a_1, a_2, \ldots, a_{n_a}$. Once the conditional has been fully executed, we bring these two divergent paths of control back together in order that subsequent code generation may continue from the base (single dynamic path) situation.

Figure 47(b) illustrates such a scheme as applied to the example: if `p` is true, control passes through thread `T2` followed by `T3`; if `p` is false, we proceed directly to

```
x := if p then v!12 else 100;
y := x+1;
...
```

(a)

```
THREAD T1 (my_id, my_state, dummy)
...
  if (p) then
    { s = make_new_state_object();
       s.vector = v; s.index = 12;
       s.direct_key = generate_new_direct_key();
       s.replyto = my_local_node_id();
       AMAM_enter_request (v.part_fun(12,v.len),
                                   INDEX_THREAD, s);
       AMAM_suspend_activation (my_id,
                                        s.direct_key, T2);
  } else {
       my_state.x = 100;
       AMAM_enter_activation (T3, my_state);
       AMAM_terminate_activation (my_id);
  }
}

THREAD T2 (my_id, my_state, result)
{ my_state.x = result;
   AMAM_enter_activation (T3, my_state);
   AMAM_terminate_activation (my_id);
}

THREAD T3 (my_id, my_state, dummy)
{ my_state.y = my_state.x + 1;
...
```

(b)

Figure 47. Implementing Conditional Split-Phase

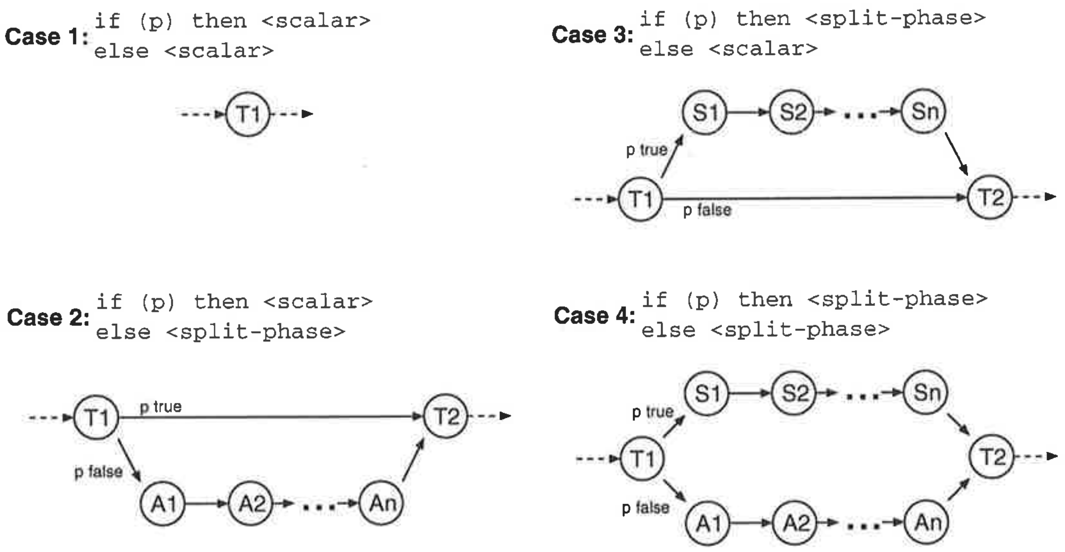


Figure 48. Execution Paths For Conditional Split-Phase

`T3` (the joining point).

Figure 48 shows schematically the thread structures generated by the four possible combinations of scalar and split-phase (unbounded latency) computation within a conditional. Case 3 demonstrates the situation prevailing in Figure 47. In each of the graphics of Figure 48, the circles correspond to threads within the AMAM program; those labelled `T1` refer to the implementation of computation preceding the `if` (and the evaluation of the predicate `p`), those labelled with an `A` or `S` represent threads from the subsequent or alternative execution paths respectively. Note that all four structures provide a joining point between the two execution paths, allowing a single representation of the expressions following the conditional.

The complexity of handling the conditional case arises from the non-strict semantics of the Adl `if`, necessary to ensure that only the single (appropriate) arm of the conditional is executed.

7.3.4 Code Generation

With the issue of code partitioning resolved, the annotated AST representing the program is passed to the code generation unit of the compiler. For the most part the operation of this unit is unremarkable — each AST node is associated with a code template which implements it. Complex operations like `map` are implemented as simple invocations of generic runtime system threads which embody the functionality of the operator. Despite the simplicity of this scheme, a number of interesting issues arise during the code generation: two are outlined below.

Modelling Function Application and Scope

As alluded to earlier, we choose to model the activation of a function f directly in terms of the activation of the thread implementing it. In order to model the function return flow of data, we introduce the following protocol:

- The threads `F_1, C_2, ... C_n` which model the function f each have as part of their state an object named `key`.
- The activation of the function f is translated into three stages:
  1. Creation of a state object `s` which contains copies of the function arguments and a new direct key `key_x` (created by a call to `AMAM_new_direct_key`)
  2. A call to `AMAM_enter_activation (F_1,s)`, immediately followed by
  3. A call to `AMAM_suspend_activation (key_x, caller_continuation)`
- The return of the function f is modelled as a two-stage process:
  1. A call to `AMAM_enter_result (state->key, result_val)` passing the function result, immediately followed by
  2. A call to `AMAM_terminate_activation ()`

The handling of functions is also complicated by the need to model reference to values available through the language's scope rules. Two possible alternatives exist for handling such accesses. The first involves the complete elimination of implicit scoping by means of the traditional lambda-lifting techniques [94] present in many functional language implementations. The second involves the management of a static link of the kind expounded extensively in the traditional compiler literature (e.g., [3]). Such a link effectively gives a function thread a means of accessing state objects encapsulated by the function thread which statically contains it.

While the former approach seems attractive in that it requires no runtime management, it has the side-effect of increasing the size of the state objects required for threads representing functions (since these functions now effectively have a greater number of explicit arguments). Thus, the amount of memory consumed by such state objects is increased, as is the overhead involved in explicitly copying values into the initial state fields (since there are now more fields).

With these considerations in mind, we choose to adopt the static-link based approach to scope-modelling. The specifics of the methodology are extensively documented in the compiler literature, centreing around the computation (at function-activation time) of a pointer to the state of the function thread activation statically enclosing the newly-activated function. This approach is suitable to our Adl implementation because normal applications of Adl functions are always modeled as the *local* activation of the thread form of the function. Thus a state pointer computed by the thread activating the new function thread (a pointer to an address in the local heap) is still meaningful in the context of the new function thread activation, since that activation executes on the same node as the caller, and thus in the presence of the same heap.

To model the second-order constructs of Adl (i.e., `map`, `scan`, `reduce`, and `while`) we implement a protocol where, in addition to specifying a pointer to the threaded form of the function argument, the initial state passed to such constructs also includes a static link pointer to the designated function's (static) parent. Using this pointer, the thread implementing the operator can establish the proper static linking for any instances of the function it chooses to activate.

Copying State to Satisfy Activation Constraints

On occasion it is necessary to produce a remote copy of the state object of an activation (e.g., to invoke a copy of the present activation on a remote node). In producing such a copy it is critical to take into account the static link field of the state: the resulting copy must still provide an access mechanism to obtain the state of the static parent, in the form of a pointer into the local memory. A simple-minded copy, in which an image of the original state is transmitted to the remote node, is not enough — the static link field of such a copy remains a pointer into the memory space of the originating node, and is not useful for dereferencing on the target node. We address this problem by introducing an alternate (somewhat more complex) protocol for copying state between nodes.

The need to produce copies of state objects arises exclusively within the thread-forms of second-order operators. As discussed in Section 5.2.1, an important aspect of the thread modelling of DP operations is the need to satisfy an activation constraint. In AMAM-thread realizations of DP operators, this constraint is met by arranging for some nodes to be remotely activated by their peers so that they may perform their part of the co-operative computation. See lines `9–12` of the AMAM **MAP** thread (Figure 41 of Section 6.4.2) for an example of such activation. During the remote activation of a thread, the invoking activation sends a copy of its own state to the remote node, to be used as the initial state of the new activation. This state object will contain a pointer to the static parent's state of the function argument of the operator, which is a pointer into that node's heap. The copy of the state object which is passed to the thread activation on the remote node also contains this pointer, but now it is of no meaning, since that activation's execution is taking place in the context of a different nodal heap. Thus whenever this remote instance of the operator thread makes an activation of the function argument thread, that activation will receive a static link which is invalid.

Figure 49 illustrates the circumstance in which problems of this kind arise. The first panel shows an instance of the **MAP** thread executing on Node 1 of a machine. The state object for this activation resides at address `6` in that node's heap. To satisfy the activation constraint of the DP operation, it becomes necessary to remotely activate a copy of the **MAP** thread with identical state. The second panel shows the situation after this remote activation has been made — note that the static link field of the newly activated state (which retains the same value as the pointer field in the original state) now points to an unallocated region of Node 2's heap. When Node 2's `MAP` instance launches the per-element computation function `foo`, that activation has an invalid static parent link.

The prototype Adl compiler avoids such erroneous instances by introducing a means of constructing a *closure* for a function, that is a structure containing all values within scope of that function. When a remote instantiation of a DP operator takes place, the closure for the function argument is built and transported to the remote node. Upon receipt of this data, the remote node copies the closure into its heap. Then, upon activation of the DP operator thread on that node, it is arranged so that the `link` field of its state points to the closure copy within the local heap. This operation represents a moderate amount of overhead (in both communications and computation), but is typically required infrequently (if at all) in the course of a program's execution.

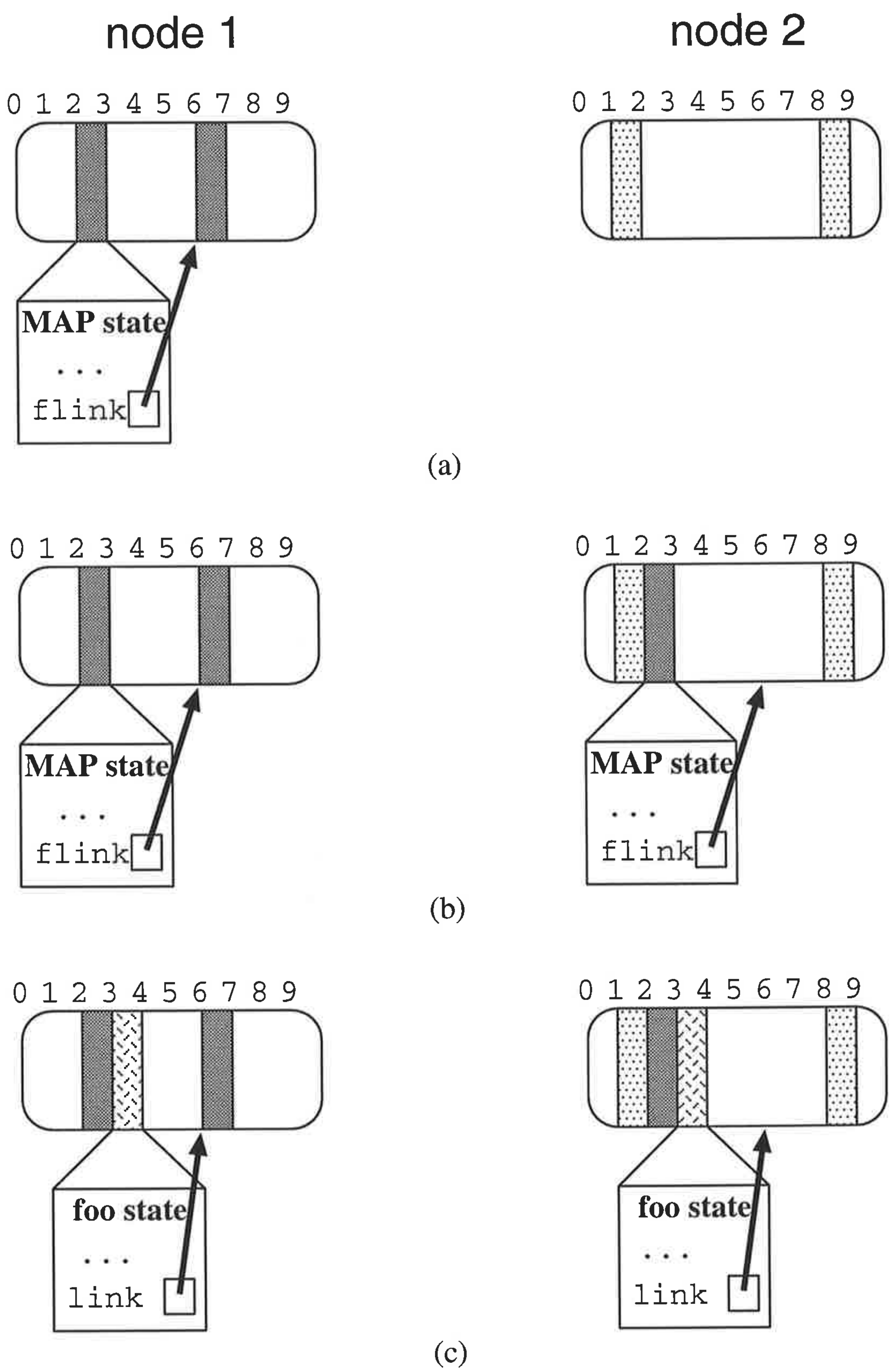


Figure 49. Difficulties in Implementing Static-Link Based Scoping for First Order DP Constructs

Key Management Protocol in Adl and its Runtime System

In Section 6.1.4 we have described the hybrid nature of synchronization keys in the AMAM system. The principal aspect of the scheme is the presence of a cheap synchronization mechanism (direct keys) and a more costly, but also more general mechanism in which synchronization is performed by matching logical keys (or tags). The former system is limited to the specification of interactions between thread activations which share a common ancestry (e.g., one is the activating parent of the other). Logical keys allow completely general specification of interaction, even between threads which have no historical relationship.

In compiling Adl to AMAM, as well as in the construction of the ADLRTS described in the previous Section, the design of a protocol of key use and logical key form is important.

In the context of the Adl compiler, we need consider only the implementation of synchronization required to implement simple expressions — all other functionality is encapsulated within the runtime library. As described earlier, our approach to compiling Adl expressions containing unbounded-latency operations (including DP operations) is to make such computations split-phase. That is, to suspend evaluation of the expression immediately after a call to a runtime or function thread, and continue only once that new thread activation has completed and supplied a value. Since all such flows of data/control are effectively call-return in nature, all are amenable to specification through AMAM's direct keys — that is, the code generated by the Adl compiler does not need to resort to the more general mechanism of logical keys. All code produced by the compiler uses a method of split-phase control-flow modelling identical to that defined earlier in this Section for functions.

Within many of the system threads in the Adl Runtime system the situation is more complex. Each data-parallel operator under AMAM is modelled as a set of co-operating thread activations whose interactions have communications requirements which are often only satisfied through unsolicited communication. In the current library of system threads approximately 40% of the inter-thread data flows are of this nature. Direct keys are not sufficient in these cases — a protocol of logical keys must be defined. However, several aspects of Adl (including its lack of recursion) allow us to synthesize such a scheme statically through compiler analysis. Such a *static logical key* protocol, while less efficient than direct keys, remains considerably more open to efficient implementation than the most general case. This technique for matching optimization may also be applicable to more general languages under certain circumstances.

In the current Adl compiler the following static system is used. Firstly, every static occurrence of a data-parallel operator within the program is assigned a unique *operation number* by the compiler. This integer, passed to the thread activations which implement that operator, forms part of the four-part logical key used for suspending the system thread implementing the operator. In many cases, this number is sufficient to uniquely specify the desired control and data flow within the data-parallel thread. In some circumstances, other key-fields must be defined. Commonly, the *expected sender* (i.e., the id of the node from which a value is to be received) is specified. Similarly the *vector id* of the aggregate being operated upon is used in some cases to disambiguate between similar operations across different data. The fourth field is occasionally used by the implementation to further encode a dynamic *context*.

This scheme is sufficiently general to allow for static logical keys to be defined and used in all synchronizations within the Adl runtime library not amenable to specification via direct keys. Specifically it permits patterns of communication (such as tree-based reductions) that are not based purely upon solicited communication. Importantly, it achieves such generality without the need for a co-ordinated "global key space." The static logical keys are defined directly in terms of the (identical) program running on each node of the AMAM, allowing a key to be globally known without the need for such information to be distributed or managed at runtime. This scheme of compiler-synthesized logical keys represents a major reduction in the overheads of synchronization management, and hence is a major factor in reducing the cost of our thread-based implementations on Nested Data-Parallelism.

Significantly, the definition of this protocol of logical keys is the only place within the described Adl compilation process where the language's lack of recursion makes for significant simplification. The fact that we may assume that a DP operator statically nested within an Adl function f (and not within another DP operation) can only have a single activation at any point in time, allows for that operator's thread to be uniquely identified by a static operator number and a vector identifier. In Section 9.1.2, we propose a simple extension to the described key protocol which would disambiguate the multiple simultaneous activations of the same static DP operator, and thus support recursive languages.

7.4 Summary of the Language Implementation

In this chapter we have described a full implementation of a simple second-order NDP language, Adl, using the AMAM as a target thread-based execution environment. The language contains three nestable DP operators — `map`, `reduce` and `scan` — each of which is generic upon a combining function. The function passed to a DP operator may itself contain further DP operators, thus introducing the possibilities for NDP execution.

To simplify the task of compiling Adl to a thread-form suitable for execution on the AMAM, we adopt an approach which makes use of the demonstrated ability for AMAM threads to be written in a highly generic fashion. Specifically, we use the generic forms we have already constructed for the operators `map`, `reduce` and `scan` (see Sections 5.2.3 to 5.2.5), collecting AMAM versions of these threads (and others pertaining to parallel operations) into a runtime library (the ADLRTS). The task of compiling Adl then becomes one of translating a source program into a simple skeleton AMAM program which implements the scalar parts of the program, activating threads from the ADLRTS whenever a parallel operation must be performed.

We have described a prototype Adl compiler which undertakes this process of translation. While the present system does not consider the automatic generation of data partitioning, the Adl environment includes a graphical interactive tool to make programmer specification of such details simple and intuitive. The compiler manages the synthesis of structure in the thread program by determining which expressions from the source program are mapped into which threads in the object code. The current prototype uses a simple syntax-based method which first allocates each source function to a thread and then splits that thread at points of unbounded latency. We consider the case of conditionals containing split-phase operations, providing a number of templates which allow for multiple paths of thread continuations.

The code generated by the Adl compiler is compact, explicitly containing only the scalar parts of the program and invocations of ADLRTS threads to perform parallel sections. Functions are modelled as threads within the AMAM code, scope within such functions being implemented by a traditional static link. Care must be taken when copying state objects between nodes to ensure that this link is correctly manipulated. Also present in the object code produced by our prototype compiler is an implicit protocol of synchronization keys (see Section 6.1.4). All interactions modelled within the skeleton AMAM code for a program are call-return in nature (since all operations involving more complex parallel interactions have been collcctcd into the ADLRTS), thus the compiler makes exclusive use of direct keys within its output.

In the following chapter we examine the performance displayed by the (unoptimized) output from our Adl compiler, comparing actual timings derived from the CM-5 AMAM against equivalent programs written in other DP languages and executed upon the same hardware. In analyzing our implementation we note a number of performance idiosyncrasies which can be attributed to the particular manner in which we have approached the task of compiling Adl. Specifically we note a number of aspects of the thread-based approach which offer highly efficient execution of irregular codes.

Chapter 8

Experimental Evaluation of the AMAM

In previous chapters we have outlined an execution environment constructed atop the Thinking Machines CM-5, and given a detailed account of how this environment has been used to implement a general paradigm of Nested Data-Parallel execution. We have considered a simple NDP language with generalized operators, Adl, and detailed an approach to compilation targetting this execution model. We turn now to an evaluation of the effectiveness of our approach, specifically analyzing the performance displayed by a compiled Adl program executing on the CM-5 AMAM.

The experimental study which follows is divided into two parts: a qualitative consideration of certain aspects of the execution (specifically communications bulk and patterns), and a quantitative, comparative analysis of observed execution times. For each evaluation we provide measurements derived from the execution of several DP programs, both flat and nested, regular and irregular.

8.1 Qualitative Visual Analysis of AMAM's NDP Execution

It is clear that, in a number of senses, the execution model embodied in AMAM and used as the source of NDP execution in the Adl system is more complex than existing paradigms of DP or NDP execution (such as those described in Chapters 1 and 2). Specifically, our approach allows each of the nodes within the parallel machine considerably more autonomy than is typical of SPMD systems in that they are independently responsible for scheduling their own execution of sub-computations from a number of simultaneously active DP operations. Furthermore, the generic threads we have constructed to collaboratively implement such operations are independently responsible for dynamically generating the required communications pattern to evaluate the operator over a vector whose partitioning is learned only at runtime. Such aspects of our implementation are very different from the traditional SPMD models based upon a well-defined series of operations which are executed in a deterministic order in lock-step by each node of the machine.

As we have shown in preceding chapters, the unique features of our model are useful in the construction of a clean and concise model of NDP expression. However, for this same model to be a practically useful underpinning for real NDP evaluation, we must establish that it displays the same desirable qualitative properties that traditional systems of DP (and NDP) execution provide. Accordingly, we must demonstrate:

- a degree of regularity in patterns of communication,
- efficient exploitation of the underlying machine,
- communications strategies which scale with problem size, and
- a high degree of performance predictability.

This section describes a series of experiments which establish these properties for the CM-5 AMAM. Our approach is to augment our abstract machine implementation to enable it to collect time-stamped trace data describing critical events during the execution of an abstract machine program. We focus our attention on communications events, although we also consider the costs incurred by attempts to match logical keys and by the scheduling of operations and receipt of messages. From the collected trace data we derive visualizations and animations of the executing DP program. These visualizations encapsulate qualitative aspects of the execution — by a comparative analysis of patterns within such visualized traces we derive properties of the model.

8.1.1 Instrumentation of the AMAM

To enable the tracing and ultimate visual analysis of executing AMAM programs, we augment the C-code comprising the AMAM such that during the execution of a thread program, certain time-stamped events are recorded into a *trace buffer*. Upon completion of the program, the contents of this memory are written to a trace file which can subsequently be examined, collated and animated. In the present implementation, two classes of events are considered: communications events and mode-change events. The semantics captured by each of the event trace categories are detailed below.

For the most part, the tracing carried out within the AMAM system has only slight perturbations on system performance. The network hardware of the CM-5 incorporates a high resolution timer for each node which is read directly to obtain time stamps. The very low cost of this timer read (effectively a single read from a memory-mapped hardware register) makes problems of nodal clock drift relatively insignificant. To further eliminate intrusiveness on the part of the logging system, all time-stamped traces are buffered in efficient static data-structures optimized for speed of trace entry.

Communications Events

A *communication event* occurs whenever an AMAM node sends or receives an active message, whether it be to implement a data-communication (i.e., the insertion of a tagged value into a remote node's result pool) or to accomplish remote-activation of a thread (i.e., the insertion of a new activation into a remote node's activation pool). Each event generates a time-stamped log entry which records the identity of the node on which it occurred, the size of the message sent or received, and an identifier used to uniquely identify the message's content. This information, once collected and collated post-run forms the basis for the visualization and analysis of communication patterns and volume.

Mode Change Events

In addition to studying communications effects in the AMAM, we are also interested in gaining a qualitative feel for the overheads inherent in the model's execution of thread programs. To this end we define a conceptual model in which each node is, at any point in time, said to be in one of five *modes* of execution:

- executing code from a user thread,
- interpreting a message received from another node,

- matching keys to determine which activation to execute next,
- rotating the task pool to find the next executable activation, and
- implementing the logic of the scheduling loop.

Within the AMAM we collect trace events recording changes in mode. That is, on each occasion that a node switches from one of these execution modes to another, an event is recorded which captures the node's identity and the new mode it is entering. Post-run analysis of this information allows accurate plots and measures of node utilization.

8.1.2 Experimental Suite

We now consider the execution, and subsequent trace analysis, of a small suite of Adl programs compiled using the system described in Chapter 7. We concentrate on simple DP and NDP codes involving the `map` and `scan` operators (whose thread implementations are described in Sections 6.4.2 and 5.2.5 respectively). The programs we consider are intended to highlight the performance characteristics of each of these operators in isolation, as well as providing a basis for investigating the performance displayed by a nesting of the two. To this end, we consider traces derived from executing the following three DP codes:

1. `map (+1) v`: a simple non-nested application of the `map` operator which produces a vector in which all the elements of `v` have been incremented.
2. `scan (+) 0 v`: a simple non-nested application of the `scan` operator which generates a vector of running totals of values from the vector `v`.
3. `map (scan (+) 0) vv`: a nested data-parallel computation which generates, given a vector of vectors, the running totals for each of `vv`'s inner vectors.

As input to the flat DP programs, we shall consider a vector of length 64 partitioned according to a `block` strategy which allocates equal subranges of indices to each of the 32 nodes of a CM-5 AMAM.

The NDP program is considered acting upon a nest of vectors with outer length 8 and whose inner vectors are each of length 8. Figure 50 shows a representation of this structure which describes the data partitioning for this input aggregate. In this diagram, each rectangle corresponds to a vector index, the number contained by the box denoting the node to which that index was allocated. The boxes at the very top of the figure show the partitioning for the outer vector, those below the arrows show the partitioning details of each of the 8 inner vectors.

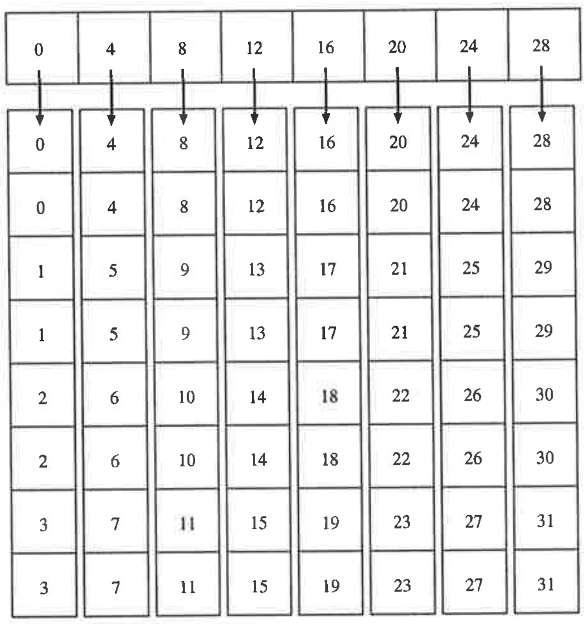


Figure 50. Partitioning For the 8 × 8 Nested Vector Input

8.1.3 Space-Time Visualization of Communication

In this section we consider the visualization of patterns of communication generated during the execution of our suite of DP-thread programs on the AMAM. As a means of producing these graphical representations we make use of the ParaGraph [50, 49, 51] parallel visualization environment.

The Flat `map` Program

We first consider an analysis of the communications regularity of each of our three test programs. To do this, we execute each program on a 32 node CM-5 instrumented version of AMAM, collecting communication event traces during the run. We then animate these trace files using the *Space-Time* plot offered by ParaGraph. The space-time visualization shows each of the 32 nodes along the vertical axis and time along the horizontal axis (increasing to the right). Each communication is shown as a diagonal line originating at the sender node (at the time that the message was sent) and ending at the receiver node (at the later time that the message was received).

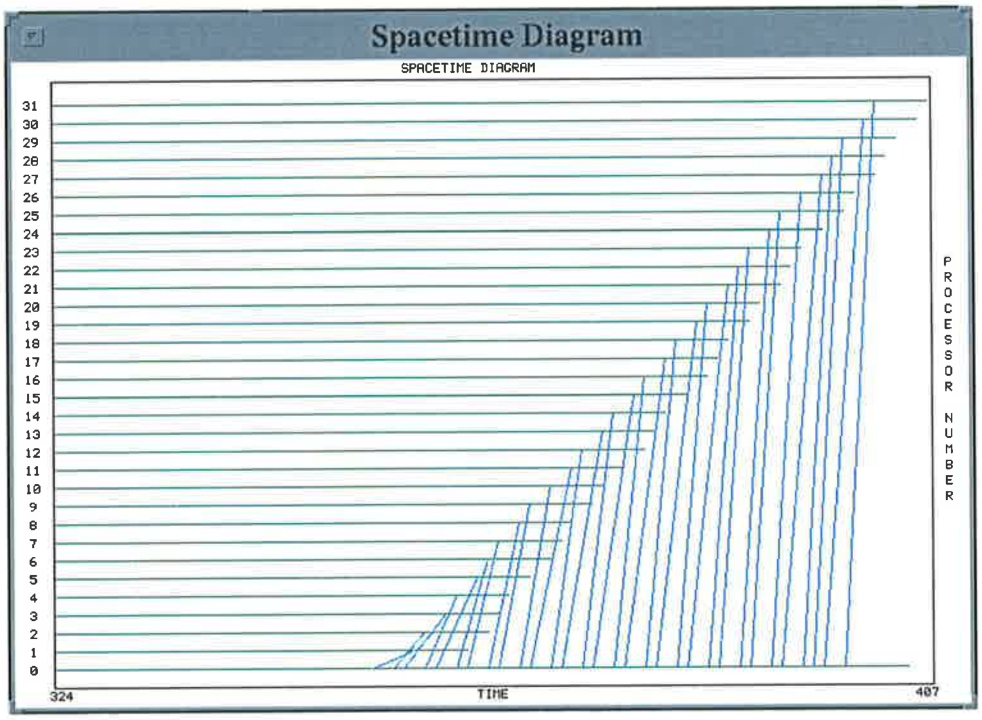


Figure 51. Space-Time Plot for Flat `map`

Figure 51 shows the space-time visualization derived from tracing the execution of the flat `map` program. This plot clearly demonstrates a very simple communication structure for this program, namely a broadcast from one node to all others. This structure may easily be explained as a manifestation of the need for the **MAP** thread (defined in Figures 41–43, Section 6.4.2) to communicate a globally-unique name for the result vector to all nodes which participated in its computation. In the program in question, the `map` operation is co-operative between all 32 nodes of the machine (since each owns indices of the input vector); thus all nodes are in the operation's participant set. Global uniqueness of name for the new vector is achieved by designating one member of this set as the *naming participant* and arranging for that node to inform all other participants via communication. In the **MAP** thread, we adopt the protocol of choosing the participant with smallest node ID to be naming participant. Thus we see in the figure communication emanating from node 0 destined for all other 31 nodes of the machine.

As described in Section 6.2.2, this protocol of name discovery also serves as a mechanism of synchronization to avoid potential problems of read-before-write access to vector indices.

The Flat scan Program

Figure 52 shows a ParaGraph space-time trace visualization for the flat data-parallel operator `scan`. The visualization shows two communications phases within the operation, both demonstrating a high degree of regularity. During the first of the phases, each participating node is involved in a tree-like pattern of communicationshown by *red* lines in the figure As the first step in this tree, all nodes communicate with immediate neighbour participants. Then each communicate with participants two distant, then four distant and so on until, in the fifth and final stage of the combination, each of the nodes $0, \ldots, 15$ communicate with the node 16 places distant. This ordered series of communications is precisely the pattern of evaluation illustrated in Figure 26 (Section 5.2.5) as a parallel implementation of `scan`. Again, the fact that the input vector is partitioned across all nodes of the machine means that all nodes are participants in the computation of the DP operator.

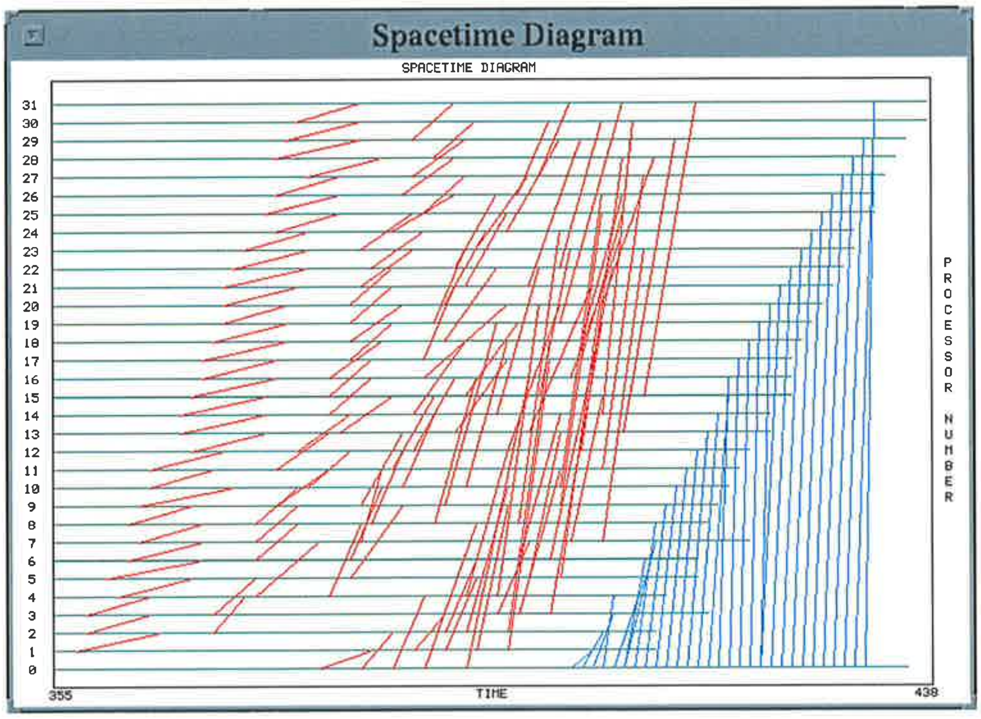


Figure 52. Space-Time Plot for Flat `scan`

After the completion of the first regular communications phase, a second phase occurs in which one node broadcasts to all others. These messages appear as *blue* lines in the space-time plot. The pattern and purpose of this phase is identical to that discussed for the flat `map` program above.

The Nested `map-scan` Program

The third example program, the nested `scan`, produces a considerably more complex trace as shown in Figure 53. However, the illustrated pattern of communication remains highly structured and regular. Furthermore it is possible to explain this detailed trace directly as a composition of the previous two.

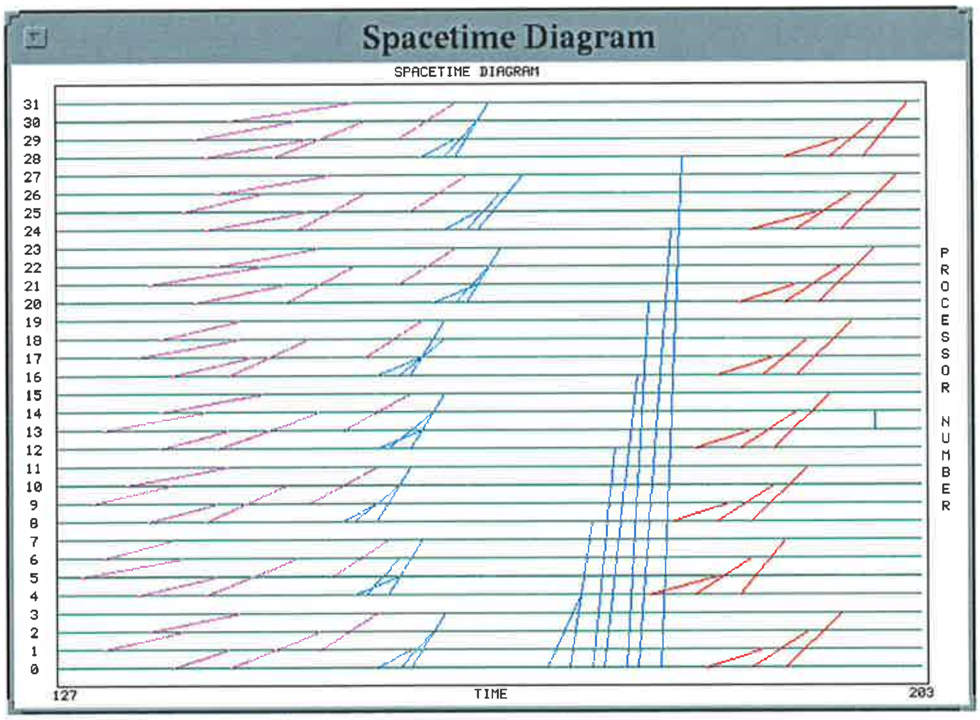


Figure 53. Space-Time Plot for Nested `scan`

Prior to undertaking such an explanation, it is important to recall the partitioning of the aggregate over which the NDP program is operating (described in Figure 50). Our observations concerning the previous traces centred around the notion of the set of participants co-operating in the computation. Thus, it is important that we identify, given the particular partitioning of the nested aggregate, such participant sets for each data-parallel operator being executed. Firstly, we observe that the outer vector of the nest is partitioned across eight nodes; $0, 4, 8, 12, 16, 20, 24, 28$. Each of the eight instances of the inner data-parallel operator (the `scan`) operates over one of the inner vectors, each of which has its own unique partitioning scheme. Thus the participant set of the first `scan` instance would be $\{0, 1, 2, 3\}$, the second would be $\{4, 5, 6, 7\}$ and so on. That is, each `scan` is executed as a co-operative computation over a group of four adjacent nodes. There is no overlap between groups: every node is a participant in exactly one of the inner operations.

With these notions of per-operator-instance participant sets in mind, the NDP trace can be explained as follows. The first set of communication (the *purple* lines) represent the tree-like reduction phases of the eight `scans` (see Section 5.2.5 for a discussion of the communications requirements of `scan`). Recall that each `scan` manifests here as a co-operation between only four nodes (the owners of one inner vector) thus the tree-like communication involves only two steps.

Immediately following the collection of *purple* lines, the figure shows a set of *blue* lines representing the broadcast stages of each of the eight `scans`. As expected, each broadcast consists of exactly three communications: each from a single member of a group of four (the naming-participant) to another member of that same group.

Following on the completion of each of the `scan` instances, the trace animation features a second set of *blue* lines, those corresponding to the broadcast at the very end of the outer `map` operation. Recall that the participant set of the `map` instance is $\{0, 4, 8, 12, 16, 20, 24, 28\}$, hence it is exactly these nodes that take part in the synchronization. Node 0 acts as the naming-participant; all others receive the name via communication from that node.

At the very end of the ParaGraph trace visualization, a set of *red* lines is visible. These are the communications steps required to satisfy the *return value dependency* of the computation. As described in Section 5.2.1, these communications are a product of the fact that the set of nodes computing the result value of the `map` is not the same as the set of nodes which desire that result (i.e., all 32 nodes). Since the nodes outside the `map` participant set did not learn the result through being one of the nodes co-operatively computing it, these nodes must be forwarded the result from a participant node.

Conclusions from the Space-Time Analysis

Each of the three space-time visualizations generated from the AMAM trace for a DP program display clear regularity in the patterns of generated communications.

Furthermore, by comparing the space-time visualizations for the flat DP operations with that for the nesting of those operations it is clear that the NDP visualization can be considered as a convolution of the visualizations for each of the component flat operators. This composibility suggests a degree of performance predictability for the environment.

8.1.4 Visualization of Aggregate Communications Volume

We next consider the same communications traces gathered from the execution of each of the three sample programs (on a 32 node CM-5 instance of AMAM) from a different viewpoint: that of overall communications volume. Our motivation in making such an analysis comes from the concern that our NDP model, which generates communications dynamically through a consideration of partitioning function, may lead to an excessive or unpredictable volume of communication. This would serve to imbue our model with a significant performance opacity: that is, it would be difficult for programmers to accurately gauge a predicted performance for their programs.

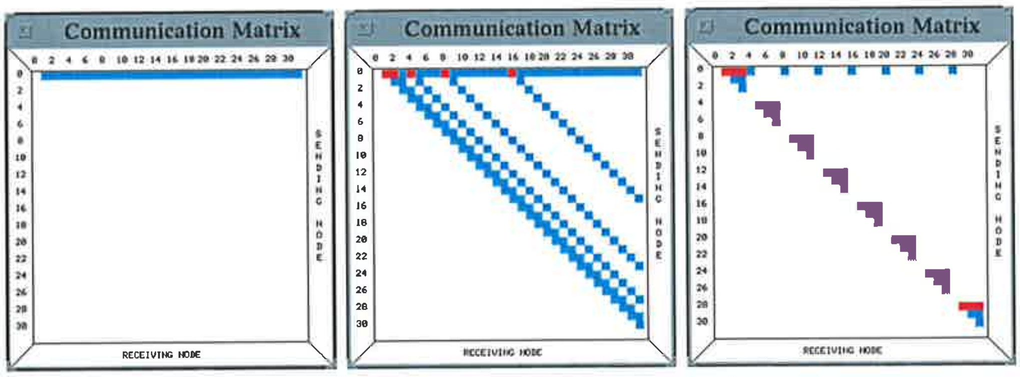


Figure 54. Communications Volume Plots

Figure 54 shows three ParaGraph *Communication Matrix* visualization, each an illustration of the overall communication volume of one of the three sample programs. These representations plot sending nodes along the vertical and receiving nodes along the horizontal. Each point in the grid is coloured according to the aggregate communications volume from the particular sender to the the particular receiver. A white block signifies no communication — other colours represent measured communications volume, with blue being a small volume and colours approaching red indicating an increasing volume.

The left-most of the three matrices shows the communications volume for the non-nested `map` program. As would be expected from earlier traces and descriptions, this is characterized by a simple low-volume broadcast: shown as a horizontal line of blue blocks. The middle matrix illustrates the communication generated by the flat `scan` — a set of highly regular tree-like combinations (the diagonal lines) and a single broadcast operation (horizontal blue band, identical to the `map` trace).

As was the case with the Space-Time diagrams above, the appearance of the `map-scan` program's volume matrix is readily explained as a composition of those for its individual operators. The nested operation involves eight instantiations of the `scan` operator, each working with an aggregate partitioned over four nodes. If we thus consider taking the matrix for the flat `scan`, scaling it down (so as to make it an operation over 4 rather than 32 nodes) we have an approximation of the matrix for one of these inner `scans`. Examination of the NDP volume plot (rightmost in Figure 54) reveals it to be almost entirely composed of eight such scaled matrices, lined up along the diagonal. The outer `map` is represented by the blue blocks along the top of the diagram (the broadcast) and the red highlights on the upper section of the eight triangles (the result propagation).

Conclusions from the Communications Volume Analysis

Each of the communication matrix visualizations generated from AMAM trace show a relatively small and predictable volume of communication generated for each program. Again, as with the space-time visualizations, there is a high degree of composibility in the observed communications volume — that is, given the observed communication volume and pattern for two component DP operations under AMAM, we can make good predictions regarding the performance of a NDP operation formed by nesting one within the other.

8.1.5 Visualization of Processor Utilization

As a final experiment we consider an analysis of the utilization of the AMAM nodes during the execution of a NDP operation, namely the nested `scan` program. As mentioned previously, the instrumented AMAM is capable of gathering traces of mode-changes in each node. A plot of these using ParaGraph's Gantt Chart display reveals general trends in the proportion of time being used for user work and system work. Figure 55 shows such a plot for the NDP test program. The nodes of the machine appear on the vertical axis of the graph, time appears along the horizontal, increasing to the right. Coloured blocks show what type of work the node was involved in at the particular point in time. The meaning associated with each of the five colours is summarized in Table 5.

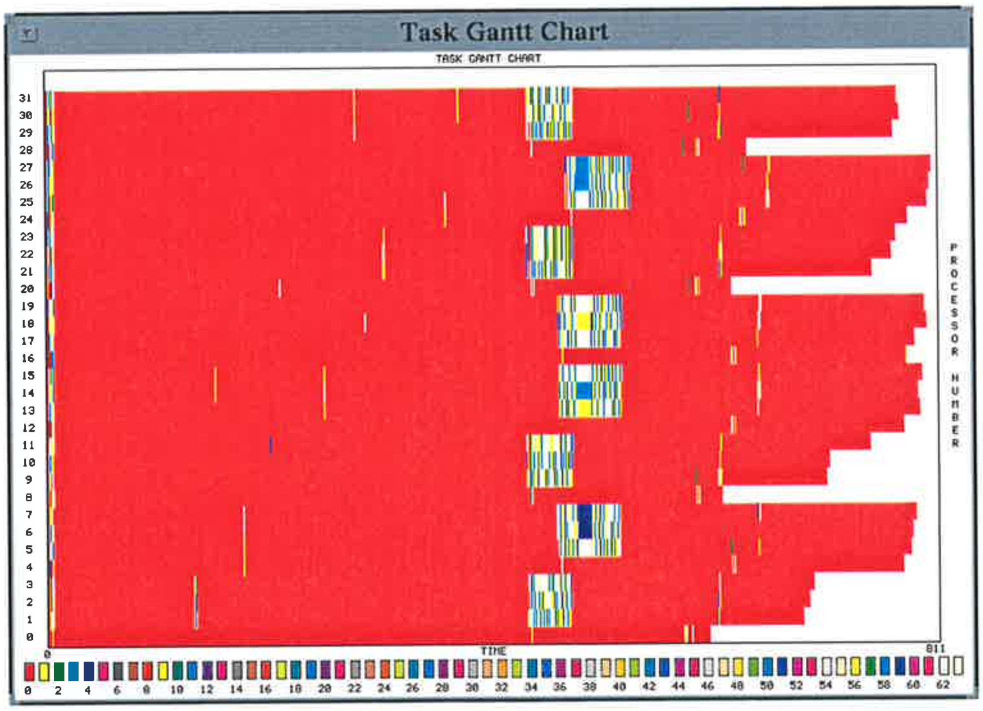


Figure 55. Processor Utilization for Nested scan

System Function	*Colour*
user_work	Red
schedule_loop	Yellow
interrupt_handler	Green
pool_rotation	Cyan
matching_keys	Dark Blue

Table 5. Colour Mappings for Utilization Plot

Conclusions from the Visualization of Processor Utilization

While it is difficult to extract meaningful general trends from the fine structure of this detailed plot of the inner workings of AMAM, two important observations can be made. The first concerns the overall proportion of user work to system overhead. It is clear from the overwhelming dominance of the very heavy shade in the plot that the majority of the runtime on all nodes was engaged in the execution of user tasks. That is, compared to the time spent on thread-management and key-matching overheads, the time spent in useful work is very high.

A second property highlighted by the plot in Figure 55 is that, for this program, the amount of work involved in fetching a thread for execution is very small compared to the amount of work involved in executing it. The large system-dominated blocks in the middle section of the visualization represent points at which no user work was available to execute. This corresponds to the periods when non-naming participants of the various `scan` operations were blocked waiting for a broadcast message. It is interesting to note that little system overhead is visible during the value exchange portion of each `scan` when threads are similarly blocked awaiting messages. The sole difference in this case is the existence of other ready threads of work which can be scheduled to mask this latency.

8.1.6 Related Work

A more comprehensive tracing and visualization system [110] has been constructed for the CM-5 AMAM system as part of a related research project. This system aims to provide a higher level visualization of processor utilization which is tied closely back to DP constructs in an Adl source.

8.1.7 Summary of Qualitative Results

The experiments described in this analysis have appraised a number of practical aspects of the CM-5 AMAM's performance from a qualitative (visualizing) perspective. This approach permits easy recognition of important trends and regularities in the measured performance of executing DP and NDP programs.

The first important conclusion we can draw from the experimental evidence is that the observed overhead incurred due to the complexity of our model (i.e., nodal multi-threading) is reasonably low. Our utilization plot clearly demonstrates that even for a complex NDP program which assigns many threads to each node, the overwhelming majority of runtime is spent performing user work.

In considering the volume and pattern of the communication generated for the sample programs compiled under our thread-based approach, we have demonstrated a number of important properties. We have firstly shown that there is a high degree of regularity in the protocols of communication synthesized by our generic DP threads, traces from the each of the DP and NDP programs featuring simple patterns of low-volume communication. The trace generated for the `map-scan` program is of particular interest, since it highlights the composibility of our model's communication structures: the pattern obtained in the nesting of `map` and `scan` is a simple (and highly regular) combination of the patterns displayed by the operators individually.

It is also evident from our animations of communication trace that the patterns synthesized in our thread implementation of DP operators retain the familiar communications patterns associated with those operator. The tree-based protocol commonly used to serve the communications needs of the `scan` operator (see Section 5.2.5) is clearly evident in both codes involving that operator. This is an important result since it suggests that our generic thread implementations of the DP `scan` operator is displaying the same scalability as traditional implementations of the operator; such scalability is one of the key properties provided by flat DP.

Overall, the results from our qualitative analysis argue in favour of the practical utility of our approach to implementing NDP. In considering the performance characteristics of a small suite of representative DP and NDP programs, our analysis has demonstrated that two fundamental properties of conventional DP — communications regularity and scalability — are preserved under our (more general) model. These attributes are central to the popularity and high performance of the flat model (as described in Chapter 1); their preservation in our system is clearly desirable. We have also demonstrated our system to have a high degree of performance predictability, stemming from the observed composibility of NDP operator performance traces under our system. This serves to lessen the opacity of the system to the programmer: an understanding of the communications characteristics of each NDP operator allows accurate prediction of the the aggregate characteristics of any nesting of those operators.

8.2 Quantitative Analysis of Adl Performance

One of the motivating factors in choosing a paradigm of non-preemptive nodal multi-threading as the basis for this work was the suggestion that such a model would map easily and efficiently onto existing distributed memory multi-processors. In this section we evaluate the validity of this premise by undertaking an analysis involving the absolute measurement of the performance (execution time) of several DP codes written in Adl, compiled into AMAM threads and executed on the CM-5 prototype of the AMAM. No optimization was made to the raw AMAM threads produced by our unsophisticated compilation process.

We particularly concentrate upon the measurement of NDP programs which embody irregular patterns of computation. These are the class of problems which the work described in this thesis aims to support.

For purposes of comparison the scientific programs coded in Adl and evaluated have also been translated to NESL and CM Fortran [127] (a superset of Fortran 90 [81] and High Performance Fortran [57]). These programs have been executed and profiled upon the latest releases of the respective CM-5 implementations (the CMU NESL system and the heavily optimized production CMF version 2.3.0 compiler). To enable a fairer comparison, none of the systems were permitted to make use of the CM-5's vector hardware.

The programs we have considered for each of the three language systems represent natural expressions of the benchmark tests rather than heavily optimized codes in which the structure of the underlying problem has been obscured. In a number of places within our analysis we note opportunities for constructing forms from which the language system may be able to extract more parallelism at the cost of readability and clarity. However, in the comparative results reported in this chapter, no such optimizations have been applied.

All tests described in this section were performed on a 32 Node CM-5 made available by the South Australian Centre for Parallel Computing.

8.2.1 Basic Data-Parallel Performance

We begin our analysis of Adl/AMAM performance by considering the execution cost of simple flat DP operations. Our execution environment was not specifically implemented with an eye towards optimal execution of such forms, and is thus likely to display significantly worse performance than systems optimized for such operations (e.g., CM Fortran). In spite of this, these flat DP tests are useful from the practical perspective that even in many irregular NDP programs there are commonly phases of execution which involve flat DP modes of execution. These results are also useful as a basis for interpreting later experiments which consider nests of these flat DP operations.

One important factor to keep in mind in the comparative analyses which follow is the existence of CM-5 hardware support for certain fundamental DP and DP-like operations (see Section 1.1.2). These facilities are made available by the existence of a second inter-node communications network (the *control network*) which can efficiently: synchronize all nodes at a barrier; broadcast a single result to all nodes; combine a value from all nodes to produce a single result; and compute certain parallel prefix (`scan`) operations. As we discussed in Section 4.2.2, the machine Ω (and thus its implementation AMAM) cannot make use of such primitives, since they support fully general data decomposition. These hardware operations are, however, available to the CM Fortran and NESL systems. This difference proves to be a major factor in the relatively poor performance results obtained for the flat DP Adl programs we consider below.

Flat `map` Performance

Our first test program is a simple application of the apply-to-all operation `map`. The operation considered is the simple addition of 1 to every element of a vector of real numbers.

The Adl, NESL and CM Fortran versions of the code are reproduced in Section B.1.1 of the Appendices. Table 6 summarizes the timings obtained by running the three programs across aggregates of 100 and 10000 elements.

For the CM Fortran program we considered two versions: one which makes use of the general parallel iterator `forall` and another which implements the operation in terms of an aggregate-level assignment.

For the Adl program we considered a number of different block and cyclic partitions of the input vector. The value shown in the table reflects the minimum time for these various partitionings (in both cases this occurred when the input was partitioned in a cyclic fashion across all 32 nodes). Figure 56 shows costs for the various partitionings in the 100 index vector tests. This shows timings for divisions of the input vector into 32, 16, 8, 4, and 2 blocks as well as cyclic assignment of indices across 32, 16, 8, 4, and 2 nodes. Also plotted in the graph is the timing achieved when all data is allocated to a single processor (and hence the operation is serial).

Total Execution Time (in milliseconds)				
Size	Adl (32N)	NESL	CMF (forall)	CMF (!forall)
100	4.550	0.668	0.042	0.041
10000	40.149	0.844	0.353	0.354

Table 6. Comparative Performance of the Flat `map` on a 32 Node CM-5

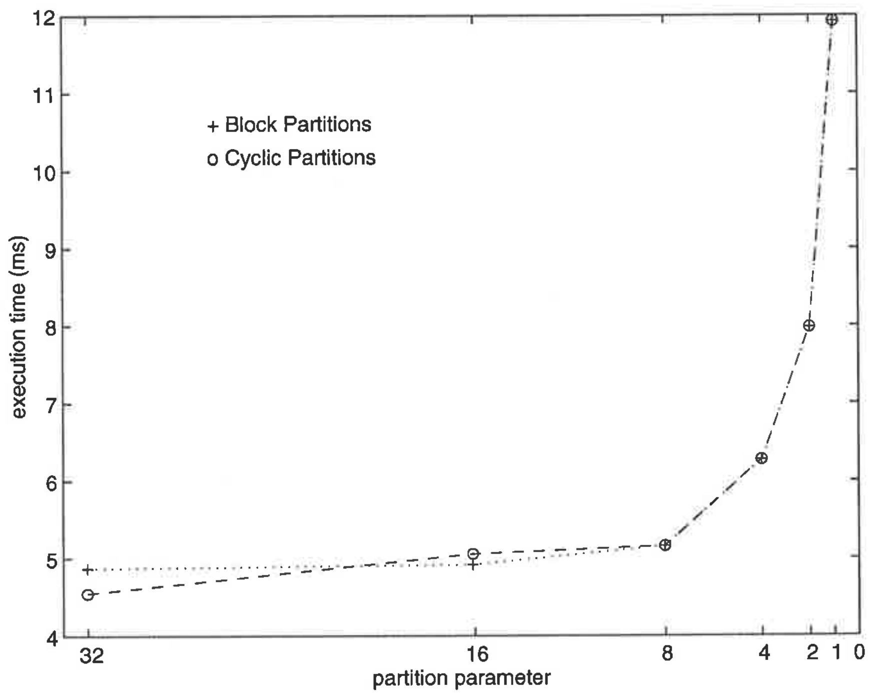


Figure 56. Graph of Adl Flat `map` Performance for Varying Partitioning

It is clear from the performance measurements reported in the table that the Adl program is considerably less efficient than either the NESL or optimized CM Fortran codes. The timings for the AMAM program are approximately 100 times those of the CM Fortran code and between 8 and 50 times those of the NESL programs. In part this cost is the result of the overheads involved in the Adl code's invocation of a function for each index mapped over — neither the NESL or CM Fortran programs implement the addition operation by function. However, undoubtedly the largest contributor to this relatively poor performance is AMAM's costly synchronization as opposed to the hardware-supported synchronization available to the other two programs. Given the very fine-grained nature of the function mapped across input indices (i.e., a single addition) this overhead of the `map` thread is a significant part of the observed cost.

An additional manifestation of these overheads can be seen in the graph of Adl performance versus partitioning (i.e., degree of computational distribution). The speedup ratio observed between the serial (single node) program and that operating on all 32 nodes is only approximately 2.5 due mostly to the fact that synchronization cost increases with the number of participant nodes.

Flat `scan` Performance

We now consider the non-nested DP performance of a second of Adl's input constructs, the `scan`. The simple program tested in this analysis constituted a single application of the operator with the accumulating function +. That is, our program represented a code to compute the running sums of the real-valued elements of the input vector.

Table 7 shows the results obtained from running codes for the various language systems (code for each is given in Section B.1.2).

As before, the Adl experimental suite included runs of the program for a variety of input partitionings, based on both cyclic and block patterns. Figure 57 shows a graph of the performance displayed for the same set of partitionings discussed in the previous analysis.

Total Execution Time (in milliseconds)			
Size	Adl (32N)	NESL	CMF
100	7.931	0.414	0.275
10000	748.403	0.984	0.713

Table 7. Comparative Performance of the Flat `scan` on a 32 Node CM-5

Again these figures show our prototype Adl system displaying noticeably poorer performance than NESL and CM Fortran in this test of flat Data-Parallelism. For the smaller problem our system is approximately 20 times slower than NESL and 28 times slower than CM Fortran. Figures for the larger problem describe the Adl scan an even poorer comparative situation (700 times slower than NESL and over 1000

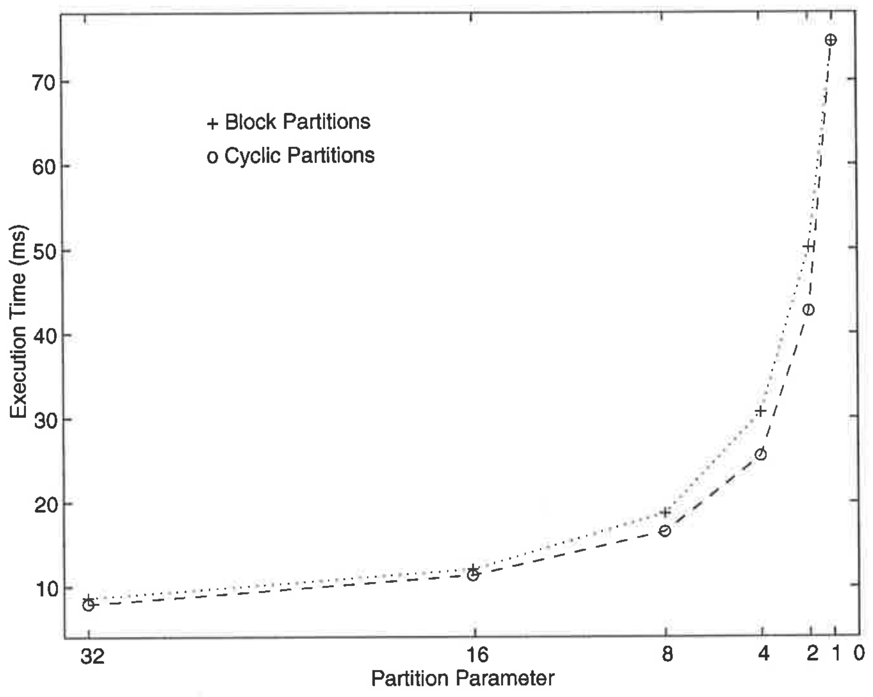


Figure 57. Graph of Adl Flat `scan` Performance Varying Partitioning

times slower than CM Fortran). Some of the loss in performance may be attributed to synchronization cost as before (due to the mismatch between AMAM's execution model and the specialized synchronization hardware provided by the CM-5), and also to the cost of invoking a user-function for each combination of sub-results. A further factor lies in the CM Fortran and NESL programs' ability to make use of CM-5 control network hardware to directly implement the `scan`.

The difference in hardware utilization also explains the different orders of complexity displayed by the programs: the CMF and NESL codes, both of which can make use of a hardware `scan` instruction display the $O(\log n)$ complexity provided by that hardware implementation; the Adl program (which uses no hardware support) displays a considerably poorer $O(n)$ performance trend due to the overwhelming cost of synthesizing a barrier synchronization.

Inspecting Figure 57 it is apparent that the current implementation of `scan` is, however, displaying a better scaling with machine size than was attained in the earlier experiment with `map`. The execution time displayed when the input aggregate is partitioning across 32 nodes is approximately 10 times that of the undistributed (serial) computation.

8.2.2 Flat Irregular Program Performance

As a final flat DP program, we consider a less regular DP operation in which values from a primary input vector are permuted according to a second input vector. The lengths of the input and permutation vectors are not necessarily identical, allowing the possibility that a given input index may be required for more than one position in the result vector, or that a given input index may not appear in the output. In the DP literature this operation is sometimes called a parallel *get* operation, or in the language of DP Fortran systems a *vector valued subscript*.

In Adl the program is implemented as a `map` over the partition vector which calls a function retrieving the appropriate index using the `!` operation. Note that some or all of these indexing requests will require inter-processor communication to be satisfied. The semantics of the operation make it impossible to statically optimize this communication (or the partitioning which generates it), since the desired motion of data is unknown until such time as the permutation vector becomes available (i.e., at runtime). The Adl program as evaluated uses simple block partitioning (across all 32 nodes) for both input and permutation vectors.

We evaluate two NESL programs which implement the permutation; one which uses the same approach as the Adl program (i.e., a `map-index` paradigm), and a second which uses an inbuilt NESL primitive (`->`) to specify the parallel `get`. The CM Fortran code uses a vector valued subscript approach. All code is presented in Section B.1.3.

Table 8 shows the results derived from evaluating the various programs for a number of differently sized input and partition vectors. Tests were run which introduced the three possible cases (input vector size $<$ permutation vector size, input vector size $=$ permutation vector size, and input vector size $>$ permutation vector size) for two sizes of problem.

Note that for a number of tests, it was impossible to complete the computation in NESL due either to VCODE difficulties or other problems leading to incorrect result values being generated.

This experiment, in contrast to the simple flat DP computations we have considered so far, is one in which irregularities in the computation introduce an

Vector Sizes		Total Execution Time (in milliseconds)			
Input	Perm	Adl (32N)	NESL (m-i)	NESL (->)	CMF
100	50	5.103	5.524	0.683	0.175
100	100	5.191	7.348	0.722	0.194
50	100	5.334	5.689	0.664	0.203
1000	500	31.000	232.826	n/a	0.260
1000	1000	53.000	495.038	n/a	0.397
500	1000	89.000	252.093	n/a	0.407

Table 8. Comparative Performance of the Permutation Program on a 32 Node CM-5

unpredictable amount of latency into the execution. The nature of the Adl/AMAM implementation of the program is that each node will host a number of threads (one per locally-held vector index) and will thus be able to mask some or all of this latency. Thus we would expect Adl's performance to be considerably more competitive in this test than observed previously.

Inspecting the table we see that the Adl program displays performance that is markedly better than the `map-index` implementation in NESL — the margin is small for the first set of tests, but large for the second. At its best the Adl permutation outperforms this NESL code by a factor of approximately 9. By comparison, the other NESL program (which uses the intrinsic parallel `get` operation) displays much better performance than Adl for the tests in which it ran to completion. This difference in NESL performance is most likely due to the existence of a highly optimized specialized executable primitive for the `get` operation. The CM Fortran runtime library also provides such a primitive, the product of much optimization effort; thus unsurprisingly the CM Fortran permutation program is also notably faster than the Adl version. It would clearly be possible to add a parallel *get* operator to Adl, implementing it as a generic ADLRTS thread — we would expect an Adl code using such an operator to be more competitive with the NESL and CM Fortran programs benchmarked in this experiment.

8.2.3 Simple Nested DP Performance

The remainder of our analysis of the Adl implementation considers the system's facilities for implementing NDP operators directly as simultaneously-active sets of

AMAM threads. We begin with a simple application of DP nesting, a parallel iteration (`map`) across a vector of vectors, during which a vector of running-sums (i.e., the result of a `scan`) is computed for each inner vector.

In Adl and NESL this computation is concisely expressed as a nesting of the DP operations considered individually in the early sections of this analysis. Such a nest exposes a high degree of parallelism: the individual co-operative computations to implement each inner scan may be simultaneously active within the machine. Such semantics cannot be expressed for CM Fortran owing to that system's flat model of DP execution — one of the dimensions of parallelism must be serialized. We consider both possible serializations. Source code for each, and for the Adl and NESL versions may be found in Section B.2.1.

The generality of the AMAM system affords great flexibility when we come to specifying the partitioning of the input aggregate (and hence the computation) for the Adl program. We may specify different partitioning functions for each vector in the nest; that is, we can make a specification of the layout for the outer vector which is independent to the layout for the first inner vector, which is independent to the partitioning for the second inner vector, and so on. To investigate the impact these partitioning decisions have upon the observed performance of the NDP computation, we considered an entire range of partitioning possibilities and executed the resulting AMAM programs across three sizes of input.

The partitionings considered were derived by first considering block partitionings of the outer vector across 32 nodes, 16 nodes, 8 nodes, 4 nodes, 2 nodes and 1 node (i.e., serialization of outer DP operations). For each of these candidate outer partitionings we considered six patterns of inner partitioning, namely: partitioning each inner vector across all 32 nodes, partitioning each across a set of 16 nodes, and so on down to considering each inner vector to be resident upon a single processor (i.e., serialization of each inner operation). In implementing these inner partitionings we attempted to spread the indices of the inner vectors evenly across the machine. Thus, if we were considering 8 inner vectors each to be partitioning into 4 blocks on the 32 node CM-5, we would assign the blocks for the first vector to one set of four adjacent processors, those for the next vector to a different set of four processors, and so on.

The many partitioning functions resulting from this process were generated interactively using the PFN tool (see Appendix A) and linked with the compiler output to create binaries implementing the various partitioning policies. These were run across three differently sized inputs: a nest of 16 vectors each of length 16, a nest of 32 vectors of length 32, and a nest of 64 vectors 64 long. Tables 9, 10 and 11 show timings for these various versions of the Adl `mapscan` program. The data is plotted in three-dimensions to display trends and highlight the relative magnitudes of the variations in performance. These are shown in Figures 58, 59, and 60.

Total Execution Time (in milliseconds)						
Inner	Outer Partitioning					
Partitioning	block_32	block_16	block_8	block_4	block_2	serial
block_32	13.841	39.515	43.316	45.341	52.441	65.941
block_16	23.676	26.878	29.286	33.616	36.219	68.394
block_8	17.501	17.321	20.109	17.445	30.956	55.940
block_4	11.398	13.295	11.754	18.996	33.774	63.981
block_2	11.699	9.855	14.804	25.804	47.284	91.245
serial	11.366	11.699	21.548	34.167	67.743	128.533

Table 9. Performance of the Adl 16 × 16 `mapscan` Varying Partitioning

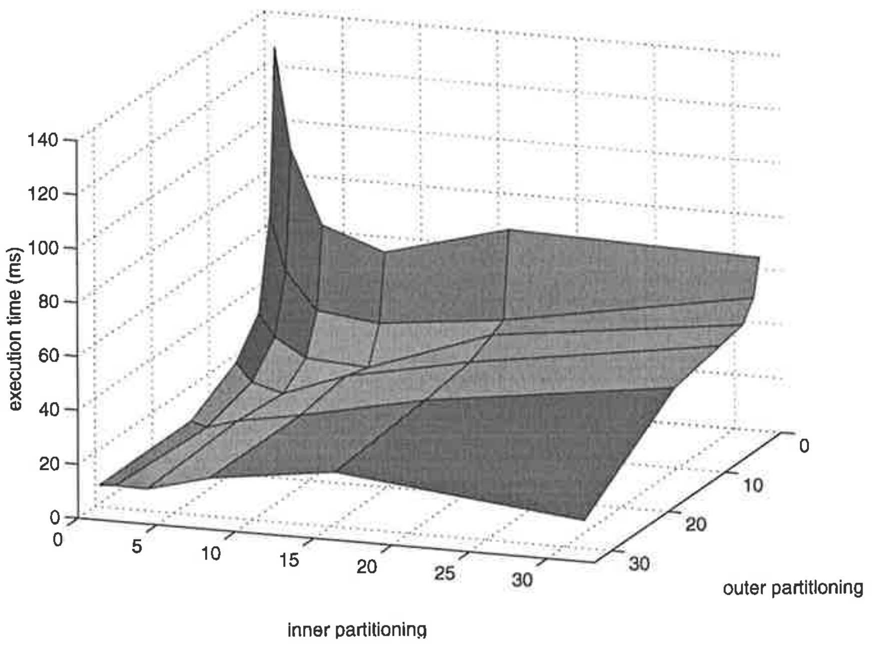


Figure 58. Plot of 16 × 16 Adl `mapscan` Performance Varying Partitioning

Total Execution Time (in milliseconds)						
Inner	Outer Partitioning					
Partitioning	`block_32`	`block_16`	`block_8`	`block_4`	`block_2`	`serial`
`block_32`	38.499	101.496	110.518	114.545	148.871	210.687
`block_16`	64.238	66.905	70.687	83.946	82.783	157.150
`block_8`	44.949	42.430	54.459	40.942	79.447	161.912
`block_4`	38.753	39.345	30.090	57.067	111.409	219.805
`block_2`	29.647	26.856	46.800	94.626	185.525	383.236
`serial`	21.368	37.902	83.897	148.874	310.727	595.816

Table 10. Performance of the Adl 32 × 32 `mapscan` Varying Partitioning

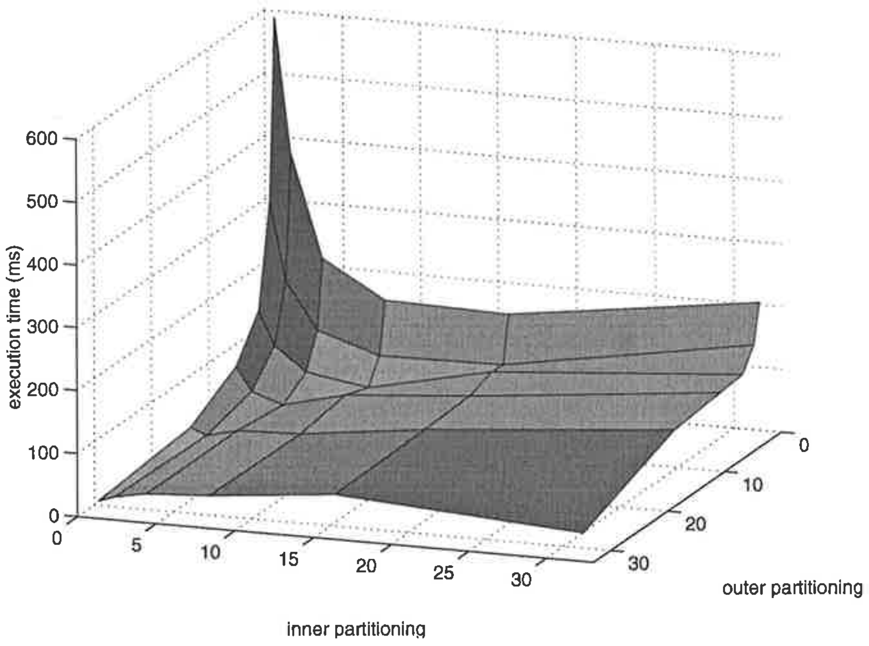


Figure 59. Plot of 32 × 32 Adl `mapscan` Performance Varying Partitioning

Total Execution Time (in milliseconds)						
Inner	Outer Partitioning					
Partitioning	`block_32`	`block_16`	`block_8`	`block_4`	`block_2`	`serial`
`block_32`	144.574	335.315	342.738	356.402	424.919	519.673
`block_16`	240.221	236.330	232.764	262.897	216.112	459.037
`block_8`	175.220	165.700	178.041	140.129	267.998	547.542
`block_4`	125.173	137.174	114.510	218.455	441.454	900.670
`block_2`	105.917	102.657	201.089	410.145	827.670	1768.604
`serial`	89.526	179.164	341.931	674.609	1416.946	2744.999

Table 11. Performance of the Adl 64 × 64 `mapscan` Varying Partitioning

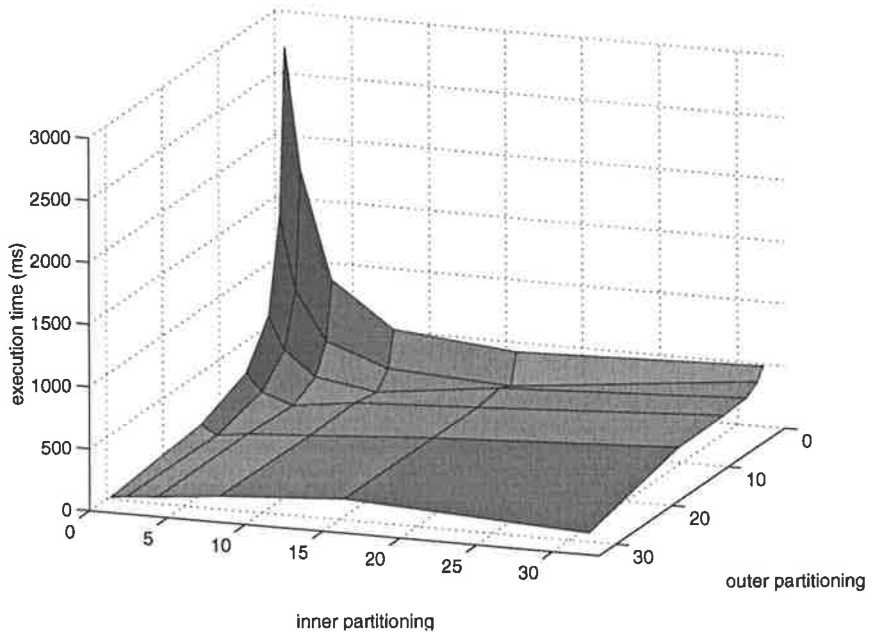


Figure 60. Plot of 64 × 64 Adl `mapscan` Performance Varying Partitioning

These tables and the corresponding plots document a non-trivial relationship between partitioning and observed performance, highlighting a fine tradeoff between the advantages of distributing computation and the additional costs of synchronizing a greater number of participants. The three sizes of experiments display similar trends in their manifestation of these cost factors. One noticeable difference, however, lies in the absolute sensitivity of performance to slight changes in partitioning — for the small problem, a small differential in partitioning-space leads to a large change in execution cost (as illustrated by steep gradients in the plot); for increasingly larger input, this effect becomes less noticeable (and thus the plots show a flatter landscape). This decreased sensitivity derives from the greater opportunities for latency hiding that arise in the larger computations.

Comparing the minimal execution times for each of the three experiments with the completely serialized execution time (i.e., the time derived from `serial` partitionings for both inner and outer vector) we see that the nested computation displays much better speedup than was observed for the earlier flat DP programs. For the 16×16 run, the Adl program achieves a maximum speedup of 13 on 16 nodes of the machine. The 32×32 experiments yield a maximum speedup of 27.9 over 32 nodes. Similarly, the 64×64 run achieves a 30.7 times speedup when using 32 nodes.

Table 12 takes the best Adl performance times for the three sizes of problems (the `block_16` $\times$ `block_2`, `block_32` $\times$ `serial`, and `block_32` $\times$ `serial` times respectively) and compares them to timings measured for the NESL program and the two semi-serialized CM Fortran programs.

Total Execution Time (in milliseconds)				
Input Nest Size	Adl	NESL	CMF (outer)	CMF (inner)
16×16	9.855	0.141	10.621	4.831
32×32	21.368	0.156	21.966	13.041
64×64	89.526	0.218	46.199	50.360

Table 12. Comparative Performance of the `mapscan` (NDP) Program on a 32 Node CM-5

The general conclusion that can be drawn from these results is that the Adl/AMAM program displays approximately the same order of magnitude performance as the two CM Fortran programs but is 100 – 400 times slower than the NESL program. The high performance of the the NESL code is achieved by the structure flattening approach described in Section 2.3. The segmented operation which results is efficiently supported on the CM-5 by specialized primitives. The poor timings displayed by CM Fortran may be completely related to lost opportunities for parallelism by virtue of the system's lack of support for NDP. The Adl system's performance is, as before, hampered by the system's inability to make use of the CM-5 control network for efficient synchronization. The system does, however, display more competitive timings than those obtained for flat DP owing to greater latency-hiding opportunities.

8.2.4 A Simple Finite Element Mesh Computation

To consider the performance of the Adl/AMAM system in a more practical (application-oriented) domain, we have considered two examples of sizeable real-world irregular scientific programs in our benchmarking. The first of these considers a simple finite element computation which solves the two-dimensional Laplace equation across an arbitrarily complex, irregularly sized triangular mesh.

The Adl version of this program, called `meshcomp` represents the mesh in a natural form — as two vectors, one defining the nodes of the mesh (in terms of their position in the plane, their initial value and an identifying number) and another defining the triangular mesh elements (as a vector denoting the three nodes which bound the element and an identifying number). The source for the program (reproduced in Section B.2.2) comprises 244 lines of Adl which, after compilation, is implemented in approximately 2800 lines of AMAM.

The NESL version of the `meshcomp` program very closely approximates the Adl version, the equivalent DP features and data representation being used in both. The CM Fortran version of the program, however, is radically different in both regards as a consequence of the lack of provision for NDP. This limitation forces a significant amount of code to be written in a serial form, and also imposes a less natural representation of the mesh (as a disjoint collection of three one-dimensional arrays and a single two-dimensional array). To gain parallelism in some phases of the computation a function f to be applied to all elements was necessarily in-lined as a series of identically ranged `forall` statements. Each `forall` implements one expression from f storing the result in a temporary array. Thus the CM Fortran program makes use of approximately a dozen extra distributed arrays to arrive at the final result. While a less space-intensive solution may be possible, it would likely be achieved by eliminating parallelism from the code (and thus presumably reducing the overall efficiency of the program).

Code for the NESL and CM Fortran programs is available in Section B.2.2.

In evaluating the performance of these programs across a small mesh of elements, we considered timings for a number of phases of the computation in order to allow for finer comparison.

- **Phase One** incorporates the construction of the *stiffness matrix* for the computation. This involves an iteration across all elements which itself includes an iteration across three integers. In Adl and NESL these iterations can be implemented in terms of a nesting of `map` constructs; in CM Fortran the inner loop is serialized (by unrolling). Note that the small extent of the inner dimension (always three elements) means that such unrolling should be considered as only a slight loss of parallelism.

- **Phase Two** involves the first part of the construction of the *diagonal matrix* for the iteration. The computation is a nested loop with outer length equal to the number of elements and inner length equal to the number of nodes. Again Adl and NESL implement this using a nesting of `map`s; CM Fortran must serialize one of these dimensions of parallelism.

- **Phase Three** completes the construction of the diagonal matrix by collapsing the aggregate computed in the previous phase. Adl uses `reduce` to achieve this; NESL uses its inbuilt `flatten` operation. The CM Fortran program does not need to perform any extra work in this phase due to a clever choice of data representation[1].

- **Phase Four** counts the number of nodes which are on the boundary of the grid (as defined by a specific initial value). Adl implements this as a `reduce` across

[1]We make use of the fact that the matrix `stiff` is a two-dimensional Fortran structure from which we can extract sub-vectors from both the rows and columns of the array. We can use a technique of row-based sub-selection, coupled with a DP `mask` to conveniently isolate and sum the particular array elements germane to each element of the diagonal matrix. This permits us to build this result incrementally in-place. This method is not easily applied to the Adl and NESL versions owing to their vector-of-vectors representation and the single assignment semantics of the language — sophisticated compiler analysis would be prerequisite to such optimizations.

a vector constructed with `map`. NESL and CM Fortran use `map`-like operators coupled with a predefined `count` operator.

- **Phase Five** sets initial value for every node using the boundary conditions. This is a simple iteration across nodes and is implemented as a flat DP operation for all programs.
- **Phase Six** comprises the body of the computation which implements the Laplace iteration. An outer while loop performs iterations of the PDE solver until such time as values have converged. Each iteration involves approximately six DP operations (including one nest of two `map`s) and many indexing operations.

Table 13 shows the performance results for each of the phases of computation as well as the total execution time for each of the programs.

Total Execution Time (in milliseconds)			
Program	Adl	NESL	CMF
`meshcomp` Phase One	20.756	36.247	12.739
`meshcomp` Phase Two	15.714	67.750	26.038
`meshcomp` Phase Three	8.974	12.333	
`meshcomp` Phase Four	4.000	0.987	0.093
`meshcomp` Phase Five	2.483	11.643	0.098
`meshcomp` Phase Six	260.013	734.960	175.413
Total `meshcomp` Time	311.940	863.920	214.382

Table 13. Performance of the Finite Element Mesh Code on a 32 Node CM-5

The overall timings displayed in the table for the various `meshcomp` programs suggests that the Adl/AMAM system achieves performance comparable to or better than either NESL or CM Fortran for irregular scientific computations. Analyzing the timings for each phase we can gain insight into the factors leading to this result.

In executing Phase One of the `meshcomp` computation, each program must implement an NDP operation (or a serialized version of it). Thus we might expect the performance displayed by each program to be similarly ranked to those realized in the `mapscan` program previously. However, in this program the inner dimension of parallelism is uniformly short (three elements), hence the cost associated with serializing it is not as high as in the previous experiment. Also, the operation applied during the outer parallel loop, includes several vector dereferences for each iteration. Thus the observed performance also bears some resemblance to the relative figures obtained for the `permutation` flat DP program.

Considering Phases Two and Three together we see that the Adl performance is slightly better than that displayed by CM Fortran and considerably better than the performance of the NESL program. The CM Fortran program loses efficiency because it serializes the outer loop of the parallel nest; the NESL program displays poor machine utilization because of the need to resort to its protocol of packing (see Section 2.3.1) to implement the conditionals nested within the inner `map` operation.

Phase Four of the NESL and CM Fortran codes achieve considerably better performance than the Adl version by virtue of the specialized parallel `count` primitive they may make use of. This operation is optimized and supported by the CM-5 control network hardware (as one of the combine-from-every-nodes operations). Conversely, the Adl program uses an application of the general `reduce` operator whose implementation has no particular hardware support.

In executing Phase Five, the observed performance of the three programs varies considerably with CM Fortran executing the flat DP operation quickly, and Adl executing it approximately 25 times slower. This differential is roughly in keeping with the figures measured earlier for flat DP operations. Out of place, however, is the NESL program whose cost is approximately 5 times that of the Adl code. Again this is a product of the packing operation used within the NESL implementation to handle conditional execution within DP operators. The fact that without the presence of such a conditional, we would expect NESL's performance to be of the same order of magnitude as CM Fortran's, suggests that the cost of this operation is very high.

The final phase of the `meshcomp` program incorporates nested DP, a high degree of indexing and several conditionals. Our earlier analysis suggests that the latter two of these factors would contribute to poor performance of the NESL system relative to the Adl/AMAM system. This is what we observe. The CM Fortran program suffers from serialization of the single nested operation but still performs marginally better than Adl due largely to lower cost execution of the various non-nested DP operations executed during an iteration of the outermost (serial) loop. It may be possible to improve the degree of exploited parallelism (and hence the observed performance) of the CM Fortran program by a process of restructuring the program and its data structures. One possibility would be to represent the program's two-dimensional arrays in a flattened form (as vectors) and expressing matrix operations in terms of expressions across subsequences of the corresponding vectors. Any such improvements in the code's exploitable parallelism, however, would come at the cost of readability and and clarity of expression due to the limitations of the flat DP model and the compilers which are implementing it.

8.2.5 A Molecular Chemistry Kernel

As a second test of the Adl system's performance for practical irregular scientific computations, we consider the kernel of a molecular chemistry program for the computation of bonded-forces between atoms in a molecule. Input data describe a set of atoms and an arbitrary set of bonds between them. In Adl and NESL this sparse connectivity matrix is represented as a ragged two-dimensional array. The CM Fortran program uses an unwieldy representation in which the sparse matrix is flattened such that its values reside in a single one-dimensional array with a second array describing the segmentation of this flattened form. Such an unnatural form is required for any parallelism to be realized in the program, since it is impossible to directly express parallel operations over ragged structures in the language — instead we must represent such data structures in terms of regular (rectangular) vectors and matrices, across which a rich variety of parallel operations are available. In contrast the NDP code implementing the Adl and NESL versions is considerably clearer in data and code structure and exposes a greater degree of parallelism. Particularly noticeable are the NDP systems' ability to cleanly describe highly parallel operations over the connectivity matrix in terms of a nesting of DP operations even in the face of the irregular (and statically unknown) shape of the input. The CM Fortran program was constrained to a single axis of parallel execution across this aggregate, thus forcing the serialization of large sections of the computation.

The source for all three versions of the `forces` program is reproduced in Section B.2.3. Table 14 summarizes the results of timing experiments in which the various programs were used to compute the bonded-forces within molecules of varying size (from 16-atom molecules up to 512-atom molecules). The molecules considered were randomly generated such that each atom was connected to approximately $\frac{1}{32}$nd of all others. All programs were then run on a 32 node CM-5 using the same set of randomly-created molecules. For several of the tests, the installed NESL system was unable to complete execution due to the occurrence of a VCODE interpreter error.

Total Execution Time (in milliseconds)			
Molec Size	Adl	NESL	CMF
16 atoms	12.245	20.150	66.674
32 atoms	19.963	33.663	160.000
64 atoms	42.216	n/a	178.389
128 atoms	105.416	n/a	484.733
256 atoms	318.636	n/a	2558.237
512 atoms	966.826	n/a	16911.795

Table 14. Performance of the Chemistry Kernel on a 32 Node CM-5

The structure of the `forces` computation is such that it may be represented as a nesting of three DP operations: an outer `map` across all atoms, within which is contained both a `map` across the connectivity list for a given vector, and a `reduce` to add the individual forces derived from each bond. The inner `map` operation contains an indexing instruction — from the earlier experiment with the `permute` program, we would expect such a structure to cause serious loss of efficiency in the NESL program. This is borne out in the figures in Table 14 where those NESL runs which completed showed a performance about 70% worse than Adl's. The results also show even poorer performance manifested by the CM Fortran version of the program. This is a product of the need to serialize one of the dimensions of parallelism. While it would be preferable to retain the outer parallelism (i.e., the loop across the entire set of nodes), the opportunity for the various atoms' connectivity lists to be of heterogeneous length makes the expression of such parallelism problematic in CM Fortran[2]. Instead, we serialize the outer loop and retain the parallelism across the individual connectivity lists.

This program provides a good example of the nature of problem which the Adl/AMAM model of NDP execution was specifically designed to execute efficiently (i.e., a completely irregular code across a ragged structure which makes use of unpredictable indexing). The impressive relative performance of the unoptimized Adl environment in this test provides some evidence of the validity of our approach.

[2]It may be possible to partially parallelize such operations by considering an even less natural and flexible representation of data. However, such a program will necessarily sacrifice a great deal of readability in its attempts to express such parallelism in terms of the vocabulary of flat DP.

8.2.6 Summary of Quantitative Results

The experiments reported in this section comprise a broad application of the DP and NDP aspects of our system and two others in a variety of situations. The tests were all performed on a 32 node CM-5: an architecture which provides a degree of hardware-support for certain operations which arise frequently in the evaluation of flat DP under the traditional lock-stepped SPMD model of execution. These facilities are not compatible with the style of execution embodied by the AMAM environment, hence are not available for use. In the benchmarks we have considered, this has in certain cases manifested in poor performance of Adl relative to the CM Fortran and NESL systems which may engage this specialized hardware.

In tests of flat DP performance the effects of this differing level of hardware support is most abundantly apparent. The Adl versions of the basic DP operator tests perform approximately two orders of magnitude poorer than the comparable programs in NESL and CM Fortran. When we consider an irregular flat DP program (a parallel `get`) Adl's performance is more competitive, at times bettering that of the `map-index` NESL program. This latter code seems to particularly suffer due to the presence of indexing within the DP operation, suggesting that the mechanism used within the CMU NESL system for implementing such dereference/communication is worse than the remote-thread approach of Adl.

The experiments which consider nested DP evaluation show the Adl system in a better light. The `mapscan` program, a regular nesting of operations, shows Adl's performance to be of the same order of magnitude as the CM Fortran versions of the code, each of which had been partially serialized to fit in with the flat DP paradigm offered by the CM Fortran environment. In this test, however, the NESL system shone brightest, outperforming both Adl and CM Fortran by two orders of magnitude. This can be attributed to a successful application of the principle of structure flattening to this simple nest of DP operators.

In the more complicated NDP examples, which consider practical applications of NDP in the realm of irregular scientific programming, the NESL approach seems less effective. In the first of these tests, the irregular fine-element mesh computation, the presence of conditionals in many of the DP nests forces the NESL system to resort to its technique of packing which, for the current implementation, seems to introduce significant overheads. The presence of indexing seems, as before, to also be a contributor to that system's poor performance. Conversely, the Adl system performs well in the presence of such irregularities, realizing timings that are comparable to those of the CM Fortran mesh computation.

The final test considers a simpler NDP code which incorporates two dimensions of parallelism (both of considerable extent), one of which contains indexing. For this experiment the Adl system records the greatest efficiency — NESL suffers due to the presence of the index operators within the DP nest, and CM Fortran loses considerable opportunities for parallelism by serializing the outer dimension of the computation.

Overall, it can be concluded from these tests that the design decision (made in Section 4.2.2) to base our model of DP execution on a loose implicitly synchronizing model of execution causes the Adl system some loss in performance on the CM-5 due to the system's incompatibility with some of the specialized hardware available on that architecture. However, such considerations seem to impact heavily only on the simple flat DP codes evaluated — for NDP programs, particularly for those displaying a high degree of irregularity (introduced by the presence of indexing and/or conditional expressions), the system displays efficiency which is comparable to, or better than, either existing system[3]. This is in spite of the relatively unoptimized nature of the Adl environment (compared specifically to the heavily optimized CM Fortran system). These good performance figures, achieved for precisely the class of problems for we set out to provide efficient support, demonstrate the potential of our approach, and recommend it as a general-purpose approach to the realization of high-performance NDP execution for real-world irregular computations.

[3]The performance characteristics of our implementation — namely a high machine utilization in NDP computation, but relatively poor handling of the flat DP case — suggest that it may be worthwhile investigating a hybrid execution model which combines both multi-threaded execution (for NDP code segments) and traditional SPMD execution (which optimizes flat DP). While a detailed discussion of such a hybrid is beyond the scope of this thesis, some brief notes on such an implementation are given in Section 9.1.2.

Chapter 9

Conclusions and Scope for Future Work

This thesis has investigated techniques for implementing the NDP paradigm in a manner which delivers good performance for programs which display irregularity in their execution. We began our analysis by determining the areas in which existing approaches to NDP implementation are deficient in their support of irregular programs, noting the qualities an implementation should possess to be a good basis for irregular scientific computing. By developing mathematical models of distributed execution and DP, we were able to prove a set of requirements prevailing upon distributed implementations of flat DP. Extending this work to consider the case of NDP, we discovered that further properties were required of an implementation to guarantee termination irrespective of data partitioning. We ultimately derived two NDP execution models — the Single Threaded Sorted Paradigm (STSP) and the Multi-Threaded Paradigm (MTP) — which provide those necessary properties.

For reasons of predicted efficiency we identified one of these candidates — the MTP approach to NDP implementation — as preferable, owing to its possibilities of masking latencies introduced by unpredictable access patterns. Expanding upon the concepts present within this approach we proposed a high-level multi-threaded abstract machine architecture atop which NDP translations could be specified in a clear and concise manner. Finding our architecture to be very general and flexible, we developed abstract machine forms for a number of important DP constructs in a manner which allowed for the arbitrary nesting of parallel operators, genericity across the per-element computation of the operator, and independence (or genericity) across the data decomposition chosen for the aggregates under consideration.

We then turned to the task of providing a concrete implementation of our abstract machine, constructing the AMAM (Active Message Abstract Machine) on the Thinking Machines CM-5. The implementation assumes nothing of the underlying architectures except the existence of an asynchronous mode of communication in the style of active messages, and thus represents a portable multi-threaded execution environment. The implementation embodies a number of features which make it optimized for NDP execution, most notably a unique hybrid system of synchronization which allows for low-cost specification of call-return type communication, yet also allows more general communication at a slightly higher cost. The AMAM also retains the quintessential properties of our abstract model: fully asynchronous execution, arbitrary interaction between threads, and the ability to specify partitioning independent operations.

To demonstrate that this environment is suited to providing a basis for NDP execution we considered a full implementation of a simple NDP programming language (Adl) atop the AMAM. Noting that the generality of our threading environments provided for very general specifications to be constructed (as demonstrated in earlier modelling), we began our language implementation by constructing a library of generic threads, collectively implementing for the various parallel constructs of the language. The task of compilation then became one of synthesizing a skeletal thread program which implemented the scalar sections of the Adl code and made invocations of the generic library threads when complex interaction was called for. Issues which were dealt with in the construction of such a skeletal object code included: the need to provide a mechanism for modelling scope, a protocol of keys to permit minimum-cost synchronization, and the construction of thread templates to implement conditional paths of split-phase control.

To evaluate the success of our approach to NDP implementation, we undertook a series of experiments to analyze the performance of several DP, NDP and irregular NDP codes, compiled using our Adl prototype compiler and executed atop the AMAM. For comparison, semantically equivalent programs were also constructed in CM Fortran (a highly-optimized traditional DP compiler) and NESL (another research implementation of NDP). These tests showed that while Adl/AMAM performed considerably worse than the competitors on flat DP programs and (to a lesser extent) on regular NDP codes, it performed very competitively for the class of problems —

irregular NDP codes — for which it was specifically designed.

9.1 Directions for Future Research

As with all research, the work described in this thesis represents a starting point for further study. We now outline some possible future investigations.

9.1.1 AMAM

Other applications of the AMAM: the execution model embodied within Ω and its implementation AMAM, is a general one. While the AMAM includes a number of features which optimize Data-Parallel styles of execution, it would be quite possible to consider framing other types of parallel execution (e.g., that derived from parallel object oriented programming [25, 82, 138], or macro-dataflow derived execution [106, 54, 99]) in terms of its threaded execution. It would be interesting to see how broad the applicability of AMAM's multi-threading could be made.

Reducing Communication through Nodal Caching: the present AMAM implementation fetches the value of a remote vector index every time it is required, regardless of the fact that the value may not have changed in the time between accesses. A protocol of caching remote vector indices could be introduced to eliminate such wasteful remote access. When considering languages based upon single-assignment semantics (such as Adl) in which the values of indices is never updated, such a system could be implemented with minimal coherency overheads.

Investigation of Load Balancing Techniques: the basis for the abstract machine Ω, and thus for the implementation AMAM, is that DP execution should be purely owner-computes. This means that the load balance across a machine is typically governed by the partitioning of the data being operated upon. Given the complex nature of the interaction between partitioning in performance, and the irregular (usually statically unpredictable) accesses and computation patterns of irregular scientific programs it will be interesting to relax this rule, either by allowing dynamic modification of partitioning functions, or instituting a protocol where the evaluation of a sub-computation may be farmed off-processor with the result sent back to the origin.

Implementing AMAM on Multi-Threaded Hardware: as described in Section 6.3.5, recent research in multiprocessor hardware has yielded a number of prototype architectures which implement a multi-threaded paradigm directly in hardware. An implementation effort which merged the ideas for thread modelling and high-level NDP execution paradigms presented in this thesis with efficient hardware-supported thread-switching and synchronization would provide an interesting counterpoint to our present CM-5 system. Issues that would need to be addressed by such an implementation include: an appropriate model for AMAM's unique hybrid synchronization paradigm, the management of thread state while an AMAM continuation awaits scheduling, and the generation of sufficiently many threads to keep a fast-switching node busy across program-introduced latencies.

9.1.2 The Adl Compiler

More Sophisticated Compilation Strategies: the Adl compiler, as presented, remains a prototype which considers only a few simple optimizations. The development of the suite of analyses and optimizations within its framework could lead to notable improvements in the observed performance of most Adl programs, particularly those in which there are few opportunities for masking latencies (introduced by inefficient object code). Areas worthy of consideration include:

- a more sophisticated scheme of thread generation which considers the reordering of expressions to maximize thread length;
- the application of traditional optimizations for single-assignment aggregates (e.g., update in place analysis): it is not clear whether such techniques, designed for very different (serial) execution environments, would be beneficial or detrimental to Adl/AMAM performance;
- the development of strategies to minimize closure sizes, eliminating costly communication during remote activations.

Analytical Determination of Data Decomposition: the present Adl compiler insists upon the programmer-specification of data layout for most of the aggregates in a program. Although a graphical tool, PFN (see Appendix A), greatly simplifies this process, such specification should ultimately be automatically generated within the compiler. The large domain of possible partitionings (the AMAM supports arbitrary decomposition), coupled with the language's support for irregularity suggests that this analysis will not be straightforward.

Consideration of Recursive Adl: as described in Chapter 7, the Adl language is defined without recursion. The compilation strategy for the language outlined in this thesis only utilizes this property in one place: the specification of a logical key protocol. A compiler for recursive Adl could likely be constructed purely through the expansion of this protocol. The principal issue which such an implementation would need to address is ensuring that two dynamic instances of the same DP operator acting across the same input vector use distinct logical keys. This could be achieved by adding extra context information into the keys used in DP operations, taking into account the dynamic instance of the user-function which spawned the operations. One possibility would be to arrange for each function instance within a recursive call chain to be assigned a unique integer (its *instance number*): this integer would then be used as a context field in the logical keys used in all DP operators spawned by that instance.

Compiling to a Hybrid AMAM/Traditional Execution Model: the quantitative performance figures obtained for the CM-5 AMAM in Chapter 8 suggested that while the environment provides efficient execution of NDP operations, its performance for flat DP execution lags behind that offered by traditional (lock-stepped SPMD) execution models. It may be worthwhile considering a hybrid execution model; that is, an environment in which flat DP segments of a program are compiled into regular locked-step SPMD operations, while NDP sections are retained as threads. The reconciliation of these two, very different, execution styles into a single framework is quite challenging.

One possible approach is to consider a program to be made up of a number of disjoint units, each of which is either serial, flat DP or NDP. Each unit would be compiled into a form which is optimal for the type of operations it contains — traditional lock-stepped SPMD for flat DP units, threads for NDP units. Program transition between units would necessitate some style of global synchronization (since the underlying execution model should change for all nodes simultaneously). In some cases this may introduce additional overheads, although it should be noted that units which have DP or NDP operators as their final computation would already perform such a synchronization step as part of that operation.

9.1.3 Performance Evaluation and Visualization

Tying Visualization Back to User Functions: while the existing provisions for visualizing Adl execution are useful for learning details concerning the overall communications cost of a program, this feedback principally concerns a lower level of abstraction than that at which an Adl program is constructed. Thus, the information provided by such visualizations is unlikely to be very useful to a programmer of the system in determining which regions of his or her program are inducing poor performance. The development of a more abstract (high-level) view of the executing program, in which cost is tied more closely back to program-level operations, would greatly assist performance tuning in Adl and other AMAM-implemented systems.

9.2 Contributions

The work reported within this thesis constitutes a number of significant contributions to the field of irregular scientific computing, and specifically to the study of implementing the NDP model of computation. In summary, we have:

- offered a formal mathematical model for Data-Parallelism and Nested Data-Parallelism, and a model for describing the implementation of such systems on a distributed memory architecture;
- motivated and described a novel implementation strategy for NDP which is based upon a multi-node multi-threaded model of execution;
- provided an abstract basis for such an execution in terms of the multi-threaded architecture Ω whose basic functionality is derived from our mathematical modelling. Ω is unique in its high-level semi-formal view of a complex multi-threaded environment and its consideration of arbitrary decomposition of data aggregates;
- derived formal definitions (in terms of our abstract machine Ω) for a number of operations common to DP and NDP programs using our thread-based NDP implementation strategy. These operational representations (thread programs)

constitute highly generic forms which are independent of the partitioning of the data aggregates with which they work;

- described an implementation of the machine Ω and our threaded implementation of NDP on a real-world distributed memory multiprocessor (the Thinking Machines CM-5). This implementation is completely portable to any distributed memory machine that offers asynchronous communications in the style of active messages;
- introduced a hybrid scheme of synchronization which offers a very low-cost synchronization mechanism for (common) types of simple, regular interaction but also provides a more costly mechanism which can be used in the specification of more general interactions;
- demonstrated how this hybrid synchronization allowed for significant performance benefits in the implementation of DP threads;
- introduced a simple NDP language — Adl — and described a technique for translating the language into a threaded form suitable for direct execution upon our multi-threaded model and its implementation;
- described how the complex aspects of an NDP language implementation can be encapsulated into a library of generic runtime routines;
- demonstrated the practical applicability of our implementation approach as an efficient basis for irregular computation by benchmarking several compiled Adl programs with equivalents written in CM Fortran and NESL, noting that for real-world irregular NDP codes the Adl implementation achieves performance which is comparable (and on occasion superior) to both competitors.

9.3 Final Observations

The work reported in this thesis offers a significant contribution to the field of irregular scientific computing by means of demonstrating an environment wherein irregular algorithms specified in a highly parallel high-level form (i.e., Nested Data-Parallelism) may be executed efficiently regardless of their irregularity. The multi-threaded nature of our approach makes it tolerant to irregular and unpredictable patterns of access by virtue of its ability to mask the latencies introduced by such forms. We believe, after having conducted the parallelization of a number of irregular codes, that this kind of tolerance is absolutely critical to the success of any implementation catering to irregular language features. While compiler analysis can help in eliminating some of the overheads introduced by program irregularities, there exist many situations in such programs where data-dependencies cannot be statically determined. Analytical approaches offer little hope in eliminating unbounded latency operations caused by such unpredictable program behaviour. It is only when we provide a system which can tolerate such unforeseen expensive operations — as ours can — that reasonable performance can be attained. It is, we believe, in the refinement and consolidation of latency-tolerating models of parallel execution such as AMAM's that the future of high-performance irregular scientific computing truly lies.

Appendix A

Visual Partitioning Specification

One of the significant features of the execution environment we have designed, implemented and benchmarked in this Thesis is its facility to generalize data partitioning within a DP environment. Where traditional DP execution environments constrain the division of data aggregates to simple pre-defined patterns, our work aims to cater to arbitrarily complex partitioning of data across processing elements. Recent research [28, 135, 79] into applying Data-Parallelism to irregular scientific problems suggests that such complexity is necessary to making the implementation efficient.

As described in Chapter 6, declarations of data layout for AMAM programs take the form of arbitrary *partitioning functions*, mathematical mappings from an index of the aggregate to the identity of its *owner* (the processing element which stores the value of that index). Each aggregate is associated with one such function at the time of its creation. In the current implementation of the model, such functions are represented by an executable form (a C function) which the programmer must supply at compile-time.

The process of designing an executable partitioning function for an AMAM aggregate is complicated by three considerations: Firstly, as demonstrated in several of the experimental evaluations reported in Section 8.2, the overall performance of a DP program under AMAM is often very sensitive to the placement of data, thus making the decisions made in partitioning function design critical to program efficiency. Secondly, the practical act of realizing a chosen partitioning design is susceptible to all the normal problems of coding. Logical errors can lead to unexpectedly poor performance, or in some cases may violate constraints of the system (e.g., how many nodes are present) leading to unpredictable runtime errors which are usually difficult to isolate. Finally, the fact that the NDP runtime libraries typically apply a coded partitioning function many times during the execution of a complex NDP operation, makes it crucial that the function is coded as efficiently as possible.

Clearly, the perils associated with this approach to partitioning specification call for the burden to be removed from the programmer through automation.

As described in Section 7.3.2, the full automation of this process — which revolves around the construction of compiler optimization strategies for designing partitioning functions based on sophisticated analysis in the presence of an accurate cost metric — is a complex and ill-understood problem, well beyond the short-term goals of the research presented in this Thesis. However, this does not rule out the possibilities of partial automation of the design process, or the provision of an environment which facilitates construction at a more abstract level. This appendix describes research undertaken to construct a prototype tool that aims to provide such an environment.

The environment we consider, called PFN, consists of an interactive X11-based visual tool which permits the definition of data decomposition to be made through gesture rather than through the authoring of code for partitioning functions. The executable partitioning functions implementing the visual specification are generated automatically from the abstract specifications made by the programmer. This process is carried out within the tool and makes use of a set of optimized code templates provably free of logical errors.

The sections which follow recap some aspects of data placement specification in AMAM (specifically the nature of partitioning functions and relative location functions), describe a mechanism for attaching partitioning annotations to aggregates whose quantity remains statically unknown, and detail the operation of PFN in the specification and automatic generation of partitioning code.

A.1 Representing Data Placement in the Adl System

Section 6.1.2 provides a detailed description of how the partitioning of a vector's indices is specified within the AMAM execution environment. Specifically detailed in this description are the roles played by the partitioning function and the associated relative location functions.

Partitioning functions constitute a mechanism for discovering which node of the AMAM owns a given index of a vector. Such functions accept two arguments (the index whose owner we wish to compute, and the length of the vector) and return a host identifier.

Figure 61 shows some examples of partitioning functions valid within the Adl implementation. These descriptions make use of the system constant `ADLRTS_num_hosts` which is equal to the number of processing elements present within the system. The first two functions implement the traditional Block and Cyclic decompositions as found in almost every DP model. The third demonstrates a variant to the Block partition which "interleaves" the first half of the set of blocks with the second half. The final function demonstrates a completely irregular partitioning.

Relative Location Functions

Relative location functions encapsulate a different kind of information concerning the location of a vector index within the AMAM system. Whereas the partitioning function gives details concerning which memory space an index resides in, the relative location function allows discovery of the slot address within that memory space which has been used to store the value of the index in question. The concrete form of the relative location function accepts the same two arguments as a partitioning function and returns an integer slot address (see the discussion in Section 6.1.2 for a more comprehensive discussion of these aspects of the AMAM storage model). For every partitioning function there is an associated relative location function: each vector which has been decomposed according to a partitioning function f must also be annotated with that f's associated relative location function.

Figure 62 shows computational forms of the relative location functions paired with `pf_block` and `pf_cyclic` from Figure 61. The remaining two partitioning functions defined in Figure 61 have relative location functions identical to `rl_block` (since these partitionings are merely node permutations of the simple Block partitioning).

Note that the exercise of constructing a relative location function which incorporates the precise semantic information needed to allow it to be paired with a given partitioning may require a reasonable amount of intellectual exercise. Many opportunities exist for subtle logical errors to be introduced into the system through obscure mismatches between the paired functions. The system assumes logical consistency in the `(pf,rl)` function-pair annotation of a vector over all indices of the

Block Partitioning:

```
host_id pf_block (int i, int l)
{
   float block_size;

   block_size = ceil ( (float) l / (float) ADLRTS_num_hosts );

   return ( (int) ((float) i / block_size) );
}
```

Cyclic Partitioning:

```
host_id pf_cyclic (int i, int l)
{
   return ( i % ADLRTS_num_hosts );
}
```

"Interleaved" Partitioning:

```
host_id pf_interleave (int i, int l)
{
   float block_size;
   int block_no;

   block_size = ceil ( (float) l / (float) ADLRTS_num_hosts );
   block_no = (int) ((float) i / block_size);

   if (2*block_no < ADLRTS_num_hosts) return (2*block_no);
   else return (2*block_no - ADLRTS_num_hosts +1);
}
```

Complex Partitioning:

```
host_id pf_complex (int i, int l)
{
   int block_size;

   block_size = ceil ( (float) l / (float) ADLRTS_num_hosts );
   switch (i/block_size) {
   case 0:  return (0);
   case 1:  return (8);
   case 2:  return (11);
   case 3:  return (7);
   }
}
```

Figure 61. Example Partitioning Functions

Block Partitioning:

```
int rl_block (int i, int l)
{
   int block_size;

   block_size = ceil ( (float) l / (float) ADLRTS_num_hosts );
   return ( i % block_size );
}
```

Cyclic Partitioning:

```
int rl_cyclic (int i, int l)
{
   return (i/ADLRTS_num_hosts);
}
```

Figure 62. Example Relative Location Functions

vector. Unpredictable runtime errors may arise in the situation where each function of a pair, even for a single vector index, imparts a different view of how data is laid out within the various memory spaces of the machine.

A.1.1 Partitioning a Vector Nest

In previous discussion of partitioning, we have considered only the situation where partitioning specification is to be made for vectors of simple elements. For that case, it is clear that a programmer may specify a full partitioning for a program P simply by denoting which `(pf, rl)` pair will partition the output of each of P's partitioning-introducing instructions. Such a specification can be viewed as a form of program annotation, that is one of associating function pairs with static program-level entities.

The situation is not so straightforward when we consider the partitioning specification of a program's input. Adl permits input to the top-level program to be of any type, including structured types built from nested vectors and (possibly) tuples. Since such program inputs are to be partitioned like any other vector within a program, it becomes necessary to provide partitioning associations (i.e., `(pf,rl)` pairs) for them. However, in the case of nested vector input, we have a situation where the concrete (runtime) instances of the inner vectors have no direct analogue in the static program.

To illustrate, consider the case of a program whose input is defined to be a single vector of vectors. In the Adl syntax, such a program would be declared:

`main input:vof vof int =` *function definition*

That is, the structure of the aggregate to be built is implicit in the type declaration of the main function's input. If this program is invoked with an input of `[[1,2],[2,3],[11]]`, a total of four vectors ($v_{outer}, v_{in1}, v_{in2}$, and v_{in3}) would be created at program startup. If, alternatively, the program were passed an input of `[[1],[2],[3],[4],[5],[6]]`, the creation of seven partitioned vectors ($v_{outer}, v_{in1}, \ldots, v_{in6}$) would ensue.

It is clearly impossible to associate the dynamically generated individual inner vector entities with static entities within the source. The fact that their number is statically indeterminable ensures this.

Given this limitation, it is clearly not possible to make partitioning assignments to individual inner vectors of an input-nest by means of program annotation. How then can the necessary specification of partitioning be made for these vectors?

One straightforward approach is to address the unknown set of inner vectors collectively by simply (and statically) associating a single `(pf,rl)` function pair with the combined set. The semantics of such an assignment is simple; each of the statically-unknown number of inner vectors will (upon its creation) receive the *same* partitioning association: `(pf,rl)`. Since there is only one assignment for the entire set of inner vectors we can associate this partitioning assignment with the (inner instance of the) `vof` constructor in the program input type specification[1]. Under such an approach, the task of specifying data partitioning for nested vector inputs may be made through static annotation. But, we have achieved this static specifiability by adopting a very simplistic model of inner vector partitioning. Specifically, we have assumed that each of the inner vectors must necessarily be assigned the exact same `(pf,rl)` pair, and hence be partitioned across the machine in an identical fashion. This is a severe limitation on partitioning expressibility in that it denies the possibility of specifying highly efficient heterogeneous partitioning of inner vectors.

Previous research into the impact of partitioning on program performance [35, 34, 36] suggests that there are many problems whose efficient computational solution depends upon the ability to partition data aggregates in non-regular (heterogeneous)

[1]The partitioning assignment for the outer vector is *always* a single `(pf,rl)` pair and can thus always be associated with the outer instance of the `vof` type constructor in the program signature.

patterns. Such situations can arise because of inherent irregularities in data structures (e.g., complex graph structures) or because of irregularities in the computation taking place (e.g., codes which perform high-cost work over one part of a structure and low-cost work across the remainder). For programs displaying these kinds of irregularities, an approach which limits partitioning to regular heterogeneous patterns will inevitably produce either an unnecessary amount of communication (to satisfy demands for off-node data), or a poor balance of computational load (from nodes being assigned unequal work due to computational irregularity).

For this reason we reject the simplistic scheme of static homogeneous partitioning of inner vectors. In its place we refine the notion of the partitioning association for vector nests, generalizing it to become a static form generic enough to permit some heterogeny among the partitioning of inner vectors. We call such a form a *partitioning scheme* for the vector nest.

We define a partitioning scheme for a vector of vectors to consists of three elements:

- A partitioning association `(pf,rl)` for the outer vector of the nest,
- A finite set of partitioning associations from which each inner vector will (at the time of its creation) receive exactly one, and
- A strategy function which determines, for each inner vector (i.e., each index of the outer vector), which association from this set it will receive.

Figure 63 shows an example of a partitioning scheme. In this sample scheme, the outer vector is defined to be partitioned according to a cyclic partitioning. Two possible partitioning associations are defined for the inner vectors of the nest, namely `(pf_inner_1, rl_inner_1)` and `(pf_inner_2, rl_inner_2)`. The first of these associations defines a traditional Block decomposition of a vector, the second defines a partitioning where all elements are mapped to a single node (23) — such a decomposition is often called Serial. As the final component of the scheme, we define a strategy function `strat`.

Consider the partitioning that would be produced if a program whose input was annotated with this partitioning scheme were invoked with the argument `[[1,2],[2,3],[11]]`. Clearly the outer vector v_{outer} would receive the association `(pf_outer, rl_outer)`; that is, it would be partitioned cyclically. The three inner vectors (v_{in1}, v_{in2} and v_{in3}) would each receive one of the two possible inner

1. Outer Vector Partitioning

```
host_id pf_outer (int i, int l)
{ return ( i % ADLRTS_num_hosts ); }

int rl_outer (int i, int l)
{ return ( i / ADLRTS_num_hosts ); }
```

2. Set of Inner Partitioning Possibilities

```
host_id pf_inner_1 (int i, int l)
{  float block_size;

   block_size = ceil ( (float) l / (float) ADLRTS_num_hosts );
   return ( (int) ((float) i / block_size) );
}

int rl_inner_1 (int i, int l)
{  float block_size;

   block_size = ceil ( (float) l / (float) ADLRTS_num_hosts );
   return ( i % block_size );
}

host_id pf_inner_2 (int i, int l)
{ return (23); }

int rl_inner_2 (int i, int l)
{ return (i); }
```

3. Strategy Function

```
(pf,rl) strategy (int index)
{
   switch (index % 2) {
   case 0:  return ((pf_inner_1, rl_inner_1));
   case 1:  return ((pf_inner_2, rl_inner_2));
   }
}
```

Figure 63. Example Partitioning Scheme

partitioning associations mentioned above. Applying the strategy function for each, we can determine that v_{in1} (index 0 of the outer vector) and v_{in3} become block partitioned, while v_{in2} is given the serial partitioning association.

As alluded to in the example, we associate exactly one partitioning scheme with the *entire* structured input of a program. That is, the association can be defined as a static annotation of the full top-level declaration of the program's main function. At runtime, when the concrete vectors corresponding of this input are constructed, the constructing operation is obliged to make use of the scheme's strategy function to determine a partitioning association `(pf,rl)` for each inner vectors. This mechanism permits concrete inner vectors of the input nest to be partitioned heterogeneously, while retaining the notion of partitioning specification as a program annotation..

The previous discussion concerning partitioning schemes has addressed the situation where the program input is a two-deep nesting of vectors (i.e., a vector of vectors of some base type). We can easily generalize the concept to arbitrarily nested aggregates by refining the definition of a partitioning scheme in this case. We define a generalized partitioning scheme (for a vector nest deeper than two) to consist of:

- A partitioning association `(pf,rl)` for the outermost vector,
- A finite set of partitioning *schemes* from which each inner vector nest will (at the time of its creation) receive exactly one, and
- A strategy function which determines, for each inner vector nest (i.e., each index of the outer vector), which scheme from this set it will receive.

Under this recursive definition of partitioning scheme, a nest of vectors of depth n is assigned a single partitioning scheme P. At the time of construction of the nest, when sub-nests of depth $n-1$ (i.e., the indices of the outermost vector) are being built each receives a partitioning scheme, P_i, from the defined set embodied by the second part of P's scheme. The particular scheme associated with each inner nest is decided by application of the P's strategy function. Each of these nests of depth $n-1$ contain a number of sub-nests of depth $n-2$; each such nest is assigned a partitioning scheme, P_{i_1,i_2}, based upon their parent vector scheme's strategy function and set of scheme possibilities. This process continues recursively until eventually, the construction of vectors of non-structured elements is entailed. At this point, the parent partitioning schemes ($P_{i_1,i_2,\ldots,i_{n-2}}$) are again consulted to determine the strategy for associating partitioning information to each of these inner-most vectors. However, in this situation, these parent vector strategy functions return `(pf,rl)` pairs rather than further partitioning schemes. Thus the actual value-bearing vectors of the deeply nested input structure each receive a concrete partitioning association upon their creation.

A.2 Visual Specification

As presented in the previous section, the task of designing the partitioning details of an Adl program place quite onerous demands upon the programmer, namely to design an efficient data layout given the pattern of data usage within the program, and also to code such policies to be efficiently executable and logically correct. It is clear that there exists much room within this approach for introducing automation to reduce the amount of effort needed to make such specifications. While it is an ultimate goal of the Adl project to completely eliminate the explicit programmer specification of data layout altogether (by introducing sufficiently sophisticated compiler analysis), there is scope for more short-term solutions to aspects of this problem.

This section describes a prototype tool called PFN coded partly in C with user interface sections written in Tcl/TK [136, 91]. This environment aims to automate the more mechanical stages of the partitioning specification process, namely the coding of partitioning functions and partitioning schemes (from a well-defined design) in an efficient and logically consistent manner. The tool provides a high-level graphical view of the design problem, allowing the programmer to make specifications by gesture rather than by coding. This leaves the programmer free to focus his or her intellectual effort upon the decisions of data layout policy.

A.2.1 A Conceptual Framework for Abstract Specification

The fundamental concept underlying the PFN tool is that of abstract representation of Adl vectors. When we consider the task of statically specifying properties (e.g., to which node each should be assigned) of individual indices of a concrete vector V it becomes immediately obvious that the lack of static knowledge concerning the size of V disallows any kind of direct specification via annotation of indices. If we cannot statically know how many such indices will exist, we cannot know how many indices are available for annotation, and thus cannot properly make a definition.

To overcome this problem we need to define a mechanism for abstracting over the actual dimension that a vector V will assume at runtime. We define an *abstract form* of V (denoted V'), a one-dimensional aggregate of known length n whose indices are abstract entities called *partitioning elements*. Each of these n partitioning elements ($V'_0, V'_1, \ldots, V'_{n-1}$) is an abstraction across a (dynamically calculable) set of indices from the concrete vector V. Together these n sets partition the whole index space of V. That is, if we consider a particular concrete instance of V, then we can say that every index $V[i]$ of this instance is *contained by* exactly one partitioning element. Furthermore we define this notion of containment such that every concrete index $V[i]$ contained within a partitioning element V'_j *inherits* any properties defined for V'_j.

Consider the example shown in Figure 64. The uppermost section of the figure shows a definition of an abstract decomposition V' of an as-yet unconstructed vector V. In this static specification we have chosen to divide our abstract vector into four partitioning elements V'_0, V'_1, V'_2, V'_3. At runtime, when the concrete vector V is built, this means that we will conceptually divide its indices into exactly four sets (according to some mapping). The second box of the figure illustrates one possible instantiation of the concrete vector, where it assumes a length of nine. In this dynamic situation, the bottom section of the figure shows how the indices of V *might* be divided into four sets. Each shaded circle is here representing the set associated with one of the four partitioning elements of V'.

The task of mapping (at runtime) concrete vector indices into the sets associated with each partitioning element falls to a *repetition pattern*. Every abstract vector must have exactly one such pattern associated with it. The role of this entity is to provide a mapping function which takes as argument a concrete vector index and returns the partitioning element whose set that index is to be assigned. No restrictions are placed upon how such a mapping is computed. Thus, there are clearly an infinite number of possible repetition patterns[2]. For use within our visual specification tool, however, we define only two.

A **Block** repetition pattern is the analogue of the traditional block-wise decomposition. Associating the abstract form of a vector with this pattern means

[2]Such patterns are effectively analogues of the partitioning function — instead of partitioning concrete indices to a fixed number of processing nodes, we are considering mapping the same domain onto a fixed number of abstract sets.

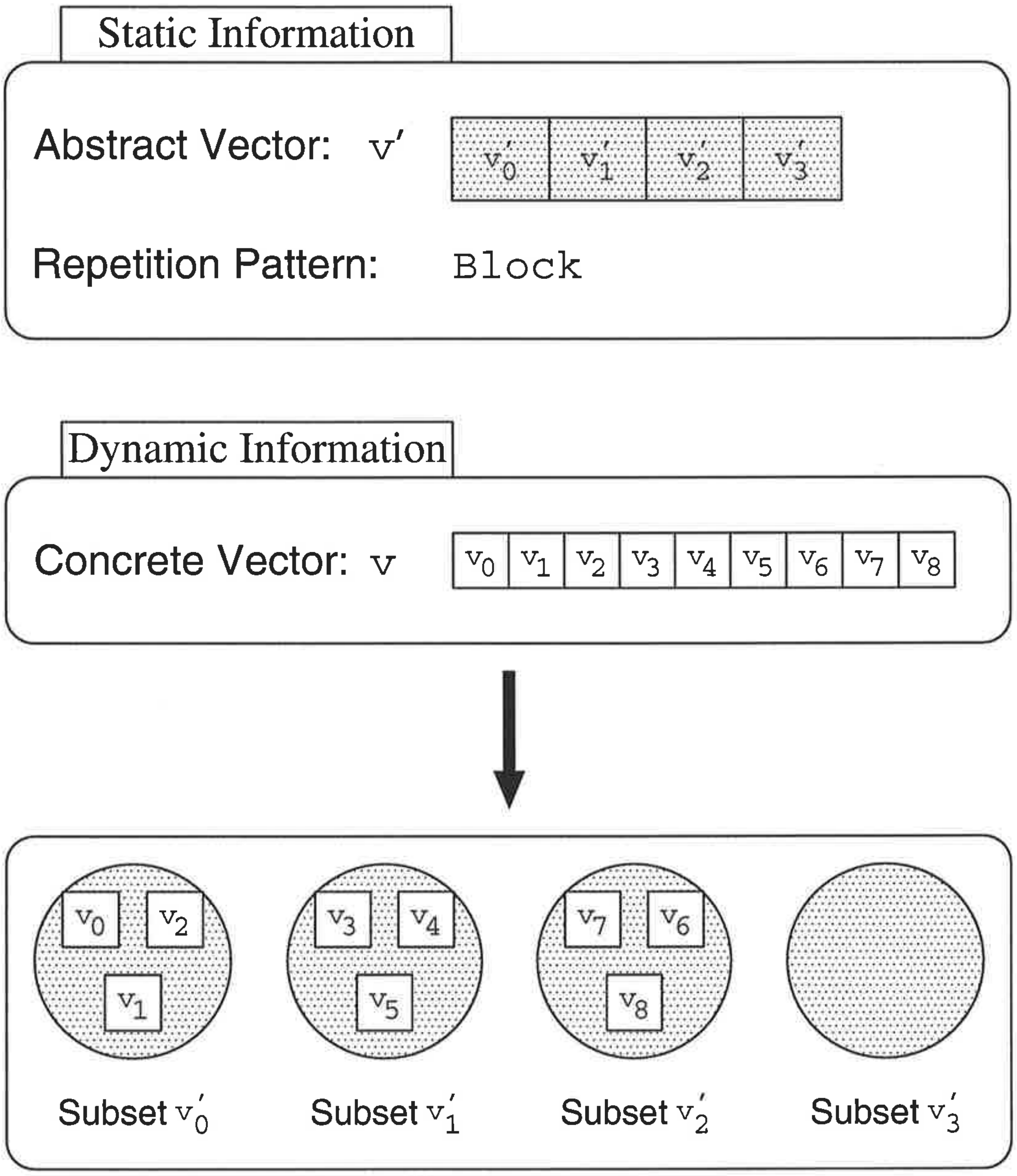


Figure 64. Decomposing a Vector into Abstract Partitioning Elements

that, at the time of its creation, the indices of the concrete vector will be conceptually divided into a number of blocks (ideally one block per partitioning element in the abstract vector), each a segment of the index-space wherein members are adjacent.

The second repetition pattern, **Repeat**, maps indices to sets in a fashion based on the traditional Cyclic decomposition. At the time of its creation, a concrete vector whose abstract form had been marked with this pattern, has its first n indices (where n is the number of partitioning elements in the abstract vector) mapped directly to the set with corresponding number. That is, index zero is mapped to the set for partitioning element 0, index one is mapped to set 1 and so on. The same pattern of mapping is then applied over the next n indices (index n is mapped to set 0, $n+1$ to set 1, etc.) and so on until the entire space of concrete indices is assigned to sets. Equationally, this pattern can be represented by the function $i \to i \bmod n$.

Returning the the example in Figure 64, we can see how the decision to assign the abstract vector V' the **Block** repetition pattern has lead to the mapping of concrete indices into the four partitioning element sets. Note that the division of nine indices into four Blocks means a block size of three indices, hence set V'_3 receives *no* concrete indices. If, however, the concrete vector V was instead of length 8, all four sets would have been granted a block of two adjacent indices.

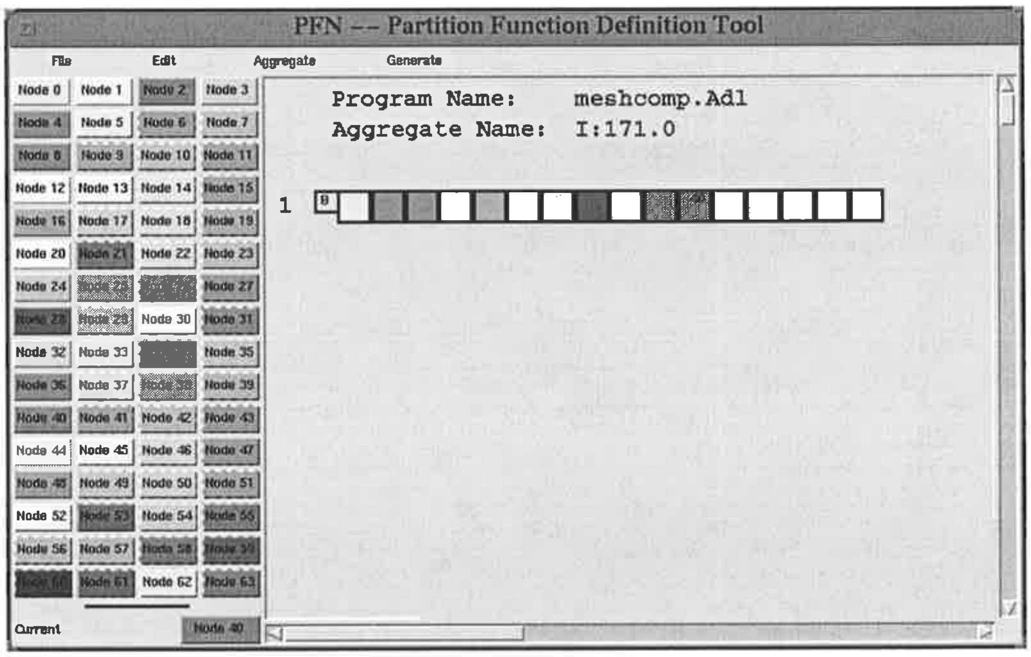
The concepts of an abstract representation for a vector, and of decomposition of that form into a known finite number of partitioning elements, are important to our model of static specification. They provide a mechanism for taking an abstract specification of property assignment across abstract vectors and implicitly transforming it into a generic definition of similar assignment across all possible concrete instances of the program's vectors. The former (abstract) specification is possible through a direct annotation approach, since the number of partitioning elements in each abstract vector is a known static quantity. The generic form, however, is an effective specification of properties for any possible instantiation of the program's data structures. For any given concrete vector generated during a run of the program, there is an abstract definition and an associated repetition pattern. At the point in time that the vector is to be created, the actual dimensions become known and thus the mapping between concrete indices and partitioning elements can be generated (by applying the repetition pattern). Given this mapping, each concrete index can be said to be contained by exactly one partitioning element. Thus each concrete index inherits the properties from one such element.

If we return to the example in Figure 64, we can define a property, *home*, for each partitioning element which denotes the node on which indices contained by the partitioning element reside. For example we could annotate partitioning element V_0' with a home Processor 4, V_1' with a home Processor 0, V_2' with a home Processor 7 and V_3' with a home Processor 8. As noted above, such an annotation can be viewed as a generic specification of the property *home* for indices of any possible concrete instantiation of V. If we consider the concrete vector shown in the second box of Figure 64 and the subsequent division of indices into partitioning elements, we can see that our abstract definition has effectively defined the *home* property for every index of V. Specifically, indices v_0, v_1 and v_2 have a home of Processor 4, indices v_3, v_4 and v_5 have a home Processor 0, indices v_6, v_7 and v_8 have a home Processor 7. Despite our declaration that one partitioning element of V' should have a home Processor 8, no index of this instance of V has such a property because the index subset assigned to V_3' by the **Block** repetition pattern is the empty set.

As an aside, it is worthwhile noting that because an abstract vector can be divided into *any (finite) number of* partitioning elements, such a process of association is no less general than a direct approach of assigning properties to concrete indices.

A.2.2 Defining of Simple Partitioning: A Sample PFN Session

Figure 65 shows the PFN tool during the specification of partitioning for a simple vector of base values. The window is divided into two frames, the *canvas* (to the right) and the *palette* (to the left). The former makes up the workspace of the tool, displaying the abstract representation of the vector currently under consideration. The name of the abstract vector (and the program it is defined within) is displayed at the top of the canvas; below it is shown a graphical representation of the aggregate, a horizontal rectangle divided into a finite number of smaller boxes. Each of these boxes represents one partitioning element of the abstract form. A smaller box is attached the the very left of the collection of partitioning element boxes — this is the repetition pattern box, and contains a single letter proclaiming whether the abstract vector is currently associated with the **Block** (B) repetition pattern or the **Repeat** (R) pattern.



```
Program Name:      meshcomp.Adl
Aggregate Name:    I:171.0
```

Figure 65. Using PFN to Specify a Partitioning Function

The palette, the set of coloured buttons along the left hand edge of the PFN window, describes the partitioning choices currently available. Each button represents a processing node within the target machine, or more accurately the *memory space* owned by that processor. Each of these disjoint memories is a potential target for a partitioning assignment. We choose to assign a unique colour with each of these processor memories, denoted by the colour of the corresponding button within the palette. During a PFN session, we will use these colours to describe which data is assigned to which memory space.

The fundamental operation performed within the tool is the assignment of partitioning elements to processing nodes. As described in the previous section, if we assign a property to every partitioning element of an abstract vector then we have *also* defined this property for every concrete vector which instantiates that abstract form. Specifically, if we annotate each element with the identity of the memory space which will contain it during a run of the program, then we have effectively defined a full partitioning of any concrete vector instances across those memory spaces. That is, we have defined a partitioning function for the vector.

Within PFN, the association of a partitioning element p to a processing node N_i

is represented graphically by filling p's box on the canvas with with the colour for N_i. The user interacts with the system to define such an association by clicking inside a partitioning element's box. Such a gesture causes the node association for that element to be replace with an association to the *current node*, the processing node we are currently working with. The box becomes filled with the corresponding colour. The identity of the current node is shown below the palette; a visual reminder of its identity is also afforded by the mouse cursor which always assumes the colour of the current node. The user may alter the Current Node at any time by simply clicking on a button of the palette.

To assist the user in determining the association of a partitioning element already coloured, whenever the mouse pointer is moved into such a box, PFN displays the number of the node to which the element has been assigned.

Altering the repetition pattern associated with a vector representation upon the canvas is achieved by clicking within its small repetition box. This gesture causes the vector to be associated with the next pattern in the cycle of available options. The present tool supports only the **Block** (B) pattern and the **Repeat** (R) pattern.

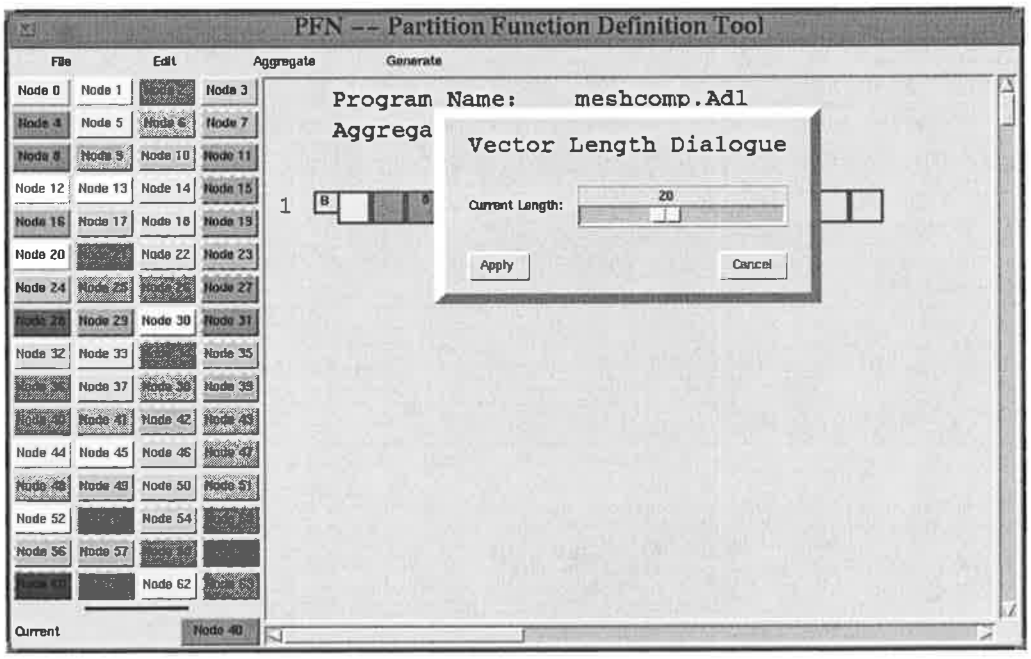


Figure 66. Redefining the Number of Elements for an Abstract Vector

The PFN tool also allows the user to redefine the number of partitioning elements into which a vector is divided. When an aggregate is first granted an abstract representation, it is defined to consist of a default number of partitioning elements. A user may "resize" this abstract form (i.e., define it to consist of a different number of partitioning elements) at any time by clicking with the right mouse button anywhere within the abstract vector's graphical form. Such a gesture pops up a dialogue incorporating a slider with which a new length may be set. Figure 66 shows such a resizing operation in progress. In the case where the representation is made longer, the newly added elements are initially unassigned. Where the length has decreased, allocation information for elements beyond the new length limit is lost.

Other editing operations useful to the definition of partitioning are also supplied within the PFN canvas. A user can middle-button drag the mouse across a region to mark a set of partitioning elements as *selected*. Disjoint selections are also permitted — middle-button dragging with the shift key down adds all elements within the drag-defined rectangle to the selection. An option to select all partitioning elements is supplied on the Edit pull-down menu.

Once a selection has been defined, two sets of operations may be performed on the specified element set. Firstly, the assignments of colours to elements within the selection can be *cycled* either left or right. In the former case this results in every element in the selection receiving the colour of the closest selected neighbour to the right. The selected element which is right-most on the canvas is granted a new colour derived from the left-most selected element's starting colour. The cycle rightwards is identical except that every selected element receives the colour of the closest selected left neighbour element.

In addition to colour rotation, the selection can be used to move and copy segments of a vector assignment. PFN supports the notion of an abstract vector *clipboard* to which the *cut* and *copy* operations from the Edit pull-down assign elements. The cut operation writes the selected elements to the clipboard and then clears each of them of its former colour allocation (marking it unassigned). The copy operation works similarly, but does not clear the selected elements. Once abstract vector elements are on the PFN clipboard they may be viewed in the *Clipboard View* window (activated from the Edit pull-down) where they appear as a vector of coloured rectangles similar to their representation upon the canvas.

The elements in the clipboard can be pasted into a current canvas selection, overwriting the colour associations of the selected vector elements with colours (assignments) drawn from the clipboard: the left-most selected element of a vector receives the colour (assignment) of the left-most clipboard item, and so on. In the case where there are more coloured elements in the clipboard than selected elements, some of the clipboard elements (the right-most ones) are not copied onto the canvas. Alternatively, if the selection is larger than the number of clipboard elements, all elements are copied once from the clipboard onto the canvas leaving some of the selected canvas elements (those beyond the length of the clipboard segment) unaltered.

Illustrations of PFN's partitioning element selection functionality and the workings of the PFN clipboard-related operations are given in Section A.2.3 below.

A.2.3 Defining a Partitioning Scheme

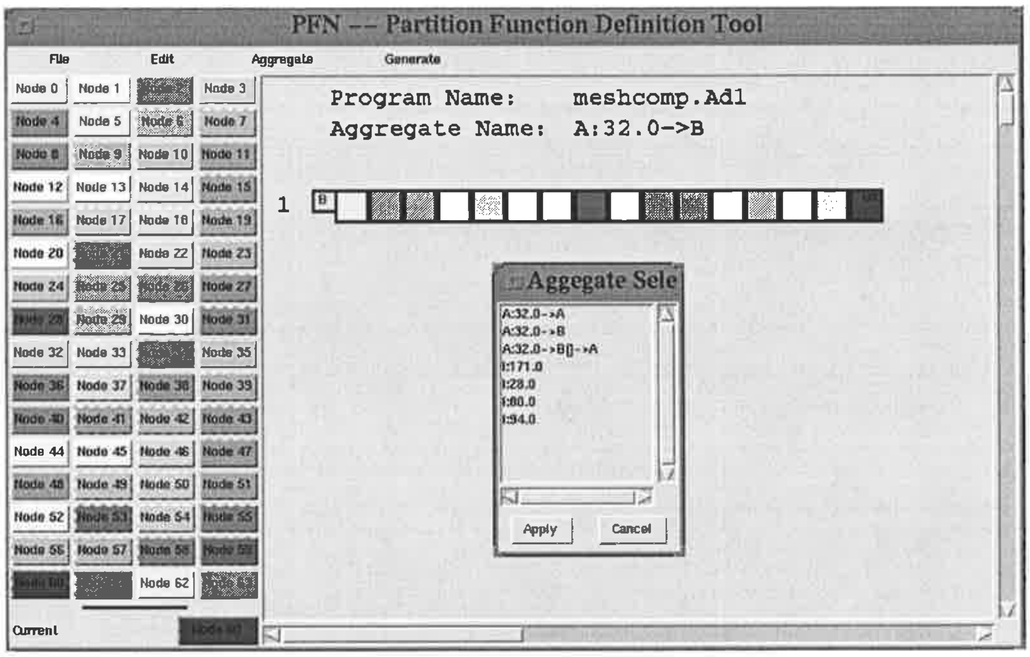
As described in Section A.1.1, it may arise during the partitioning specification of a program that we need to define a partitioning scheme for a structured program input rather than a simple partitioning function. This section describes how, by simple extension of the model presented previously, PFN can elegantly handle this more complex case.

Recall that a partitioning scheme consists of a standard declaration of partitioning for the outer vector, a set of possible partitionings for the inner vectors and a strategy function which chooses a member of this set for every element of the outer vector. We address the definition of each of these three elements in turn.

Defining Partitioning for the Outer Vector

It is clear that the outermost vector of a nest is nothing more than a simple vector of elements. Its partitioning can easily be specified in PFN using the facilities described previously. That is, the outer vector can be placed upon the canvas as a simple abstract vector and divided into a user-specified number of partitioning elements. These elements may then be associated to processing nodes by gesture. Furthermore a repetition pattern may be defined for the outer vector by clicking within a repetition pattern box affixed to the abstract vector representation.

Figure 67 shows such the PFN window during the declaration of a scheme's outer vector partitioning.



```
Program Name:     meshcomp.Adl
Aggregate Name:   A:32.0->B
```

```
A:32.0->A
A:32.0->B
A:32.0->B[]->A
1:171.0
1:28.0
1:80.0
1:94.0
```

Figure 67. Using PFN to Specify the Outer Partitioning of a Nest

Also shown in this figure is the *aggregate selection pop-up*, a dialogue which a user may activate (from the toolbar) to select which of the program's aggregates is presently visualized on the canvas. We note in this example, the dialogue selection pop-up shows an entry for the vector currently under consideration, that is the vector called `A:32.0->B`. Also within the list is a similar name, `A:32.0->B[]->A`. This second entry refers collectively to the *child set* of the vector; that is the set of vectors which are themselves the indices of the outer vector `A:32.0->B`. Figure 68 illustrates this relationship between the entities. The fact that a child set exists for `A:32.0->B` denotes that vector must be the outer vector of a nest.

Defining the Set of Inner Partitioning Possibilities

The task of defining a set of possible sub-vector partitionings can be thought of as a two stage process: firstly, we decide *how many* inner partitionings we want to be able to choose from, and secondly we make definitions for this number of abstract vector partitionings. In extending PFN to allow the specification of partitioning schemes it is clearly necessary that both activities be somehow modelled.

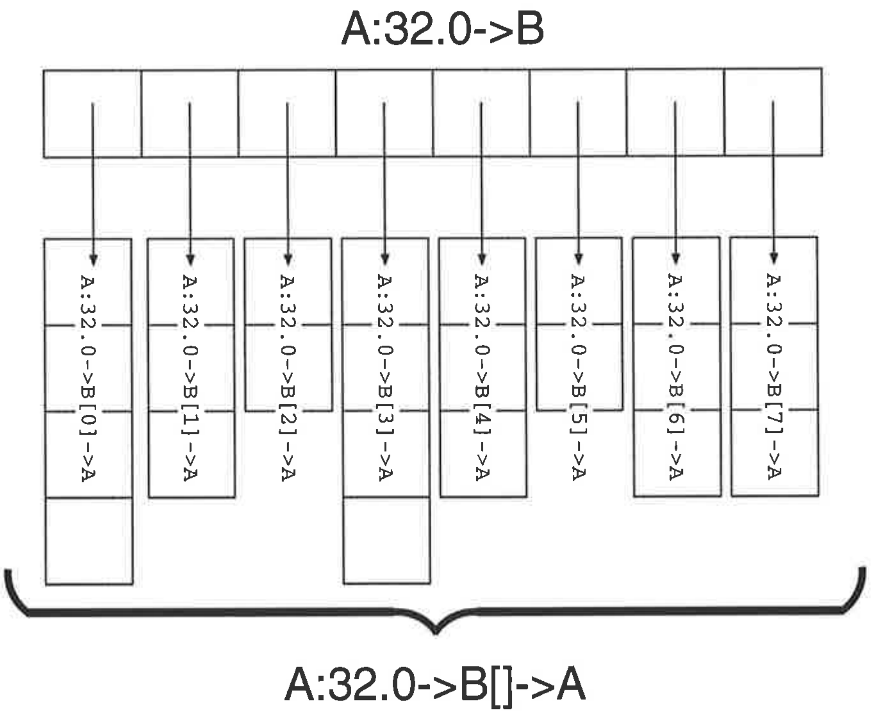


Figure 68. Naming a Nested Structure in PFN

We choose to solve the problem of determining the number of sub-vector partitionings within our possibility set implicitly rather than explicitly. We introduce a simple convention to inherit this information from an earlier (unrelated) declaration:

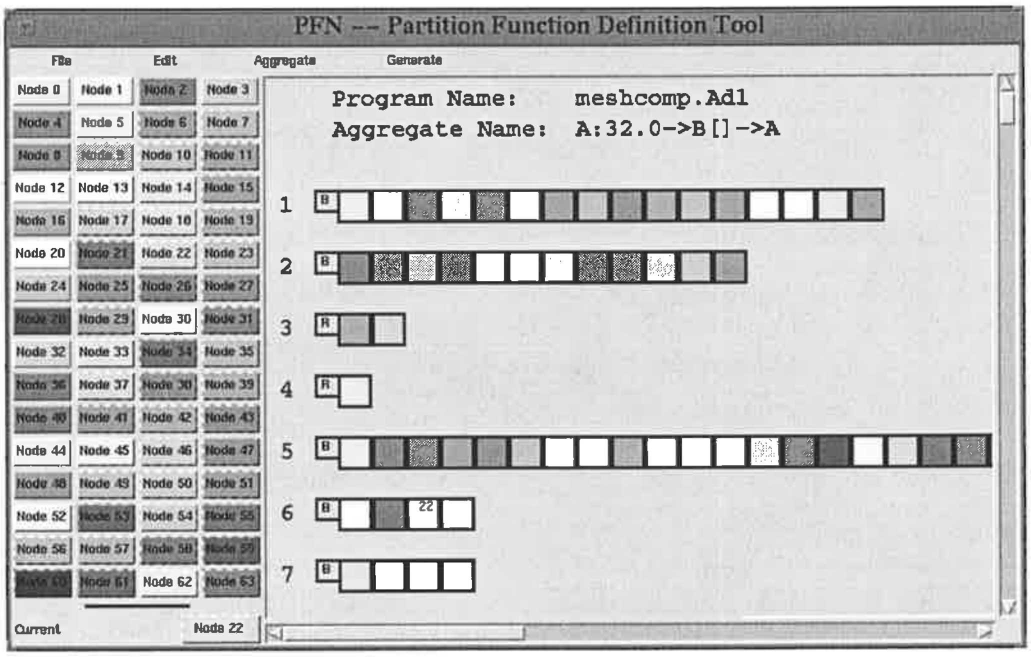
Convention 1:

By convention, the number of inner-vector partitioning possibilities shall equal the number of partitioning elements in the nest's outer (abstract) vector

That is, if we have previously defined a partitioning for the outer vector of the nest in which its abstract form was decomposed into n partitioning elements, we implicitly decide that this same number, n, should be used as the number of inner-vector partitioning possibilities for the scheme. There is no reason such a relationship must exist, our decision to adopt such a convention is merely to reduce the necessity of declaring the number explicitly. That is, it is merely a specificational convenience which permits a conceptually simple abstraction for the user to interact with — he or she can think of the partitioning scheme under construction as a nest of abstract vectors, the outer abstract vector (divided into n partitioning elements) containing n abstract forms for inner vectors.

Once we have decided the number of inner-vector partitioning possibilities, the next task is to define each of them. The PFN tool, as we have already seen, is capable of specifying the complete partitioning of a single vector. It would be feasible, once the number of inner partitionings had been decided upon, for us to simply allow the user the opportunity of dragging representations of each individually upon the PFN canvas and partitioning these representations in isolation. While this would be a workable system, it denies the relationship between partitionings within the set of inner vector possibilities. Specifically, a user constructing such a set, may wish to do so with constant consultation to previous element definitions from the same set.

To avoid the need for constantly switching between isolated representations of these related abstract vectors, we choose to extend the functionality of the PFN canvas to allow for *multiple* abstract vectors to be simultaneously represented and interacted with.



Program Name: meshcomp.Adl
Aggregate Name: A:32.0->B[]->A

Figure 69. Using PFN to Specify the Set of Inner Partitionings in a Nest

Returning to the example of partitioning the vector nest `A:32.0->B`, we see that our outer vector partitioning (as shown in Figure 67) divided the abstract form into sixteen partitioning elements. Thus, by the convention described above, the tool would implicitly afford us 16 inner vector partitioning possibilities. Furthermore, PFN will allow us the opportunity of partitioning these 16 abstract vectors simultaneously upon the same canvas. Figure 69 shows how the tool might appear during the specification of such a set of inner vector possibilities. This display would have been arrived at by the user first defining the division of the outer vector, then selecting the child set `A:32.0->B[]->A` from the aggregate selection pop-up.

As discussed previously, the PFN tool allows for the elements to be *selected* by being middle-button dragged across with the mouse. In the instance where several abstract vectors are simultaneously upon the canvas, the tool supports the selection of elements from any number of different vectors. Figure 70 shows one such multi-vector selection: a subset of the elements from the first and second canvas vector have been selected, as has the sole partitioning element comprising the fourth vector. This is indicated by the rectangles representing these selected elements being drawn with thick borders. PFN considers a multi-vector selection as a set of disjoint single-vector selections. Thus, a rotation operation applied to such a selection would cause each set of elements corresponding to a single canvas vector to exchange colours. In the example shown in the figure, this would mean that the selected elements of vector 1 would cyclically swap colours amongst themselves completely independently to the selected elements of vector 2 who collectively are performing the same operation. Colours are never transmitted from one vector to another during a rotation.

Figure 70 shows that this model of multi-vector disjointedness also prevails within the clipboard (shown as a separate window at the bottom of the diagram). The clipboard view window shows the effects of applying the copy operation (from the Edit pull-down) to the three-vector selection prevailing on the canvas. The end result of the operation is the creation of three disjoint *clipboard lines*, each a vessel for the colour assignments copied from one of the vector selections.

Pasting a multi-line selection from the clipboard into a canvas selection also preserves the disjoint nature of the various clipboard lines. Elements from the first clipboard line are copied into the partitioning elements of the first (i.e., top-most) selected vector according to the normal rules of pasting (c.f. Section A.2.2). Next, those from the second clipboard line overwrite elements of the second vector with selections, and so on. If the canvas selection spans fewer vectors than there are clipboard lines, the surplus lines are not pasted to the canvas. Alternatively, running out of clipboard lines during a paste causes the operation to cease with no modification of some selected vectors.

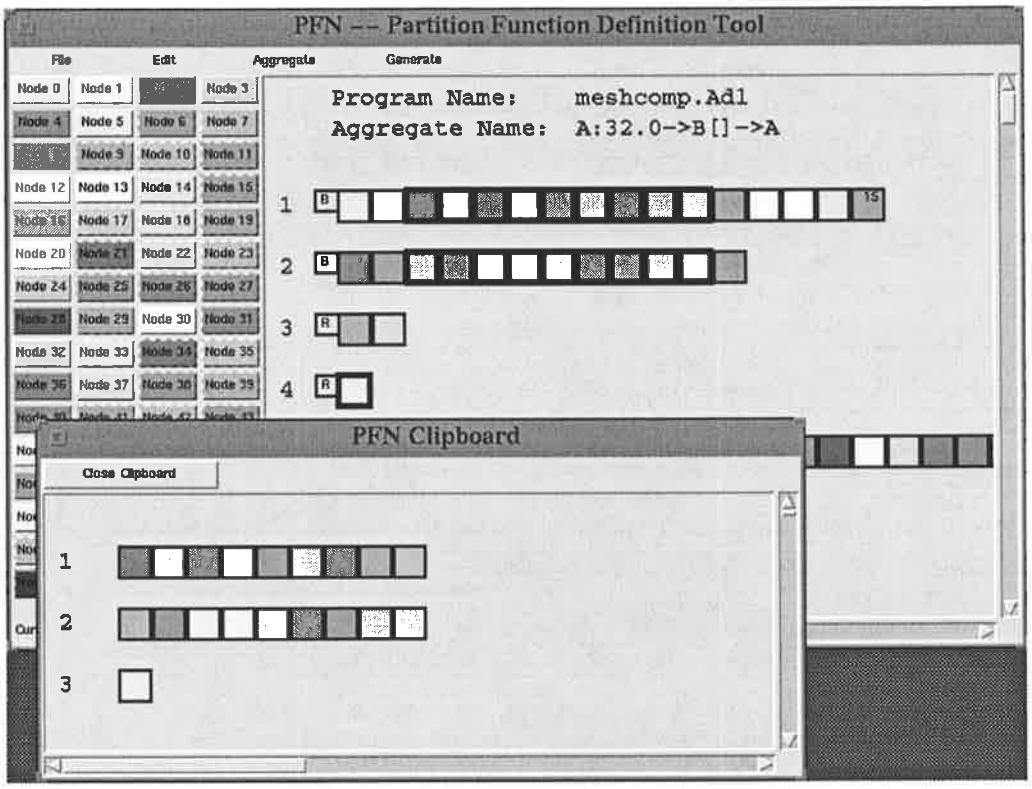

PFN -- Partition Function Definition Tool
File
Edit
Aggregate
Generate
Program Name: meshcomp.Adl
Aggregate Name: A:32.0->B[]->A
PFN Clipboard
Close Clipboard

Figure 70. Selected Elements from Several Vectors and the PFN Clipboard

Figure 71 shows the effects of pasting the selection from the clipboard as shown in the previous figure into a selection consisting of three vectors each with four selected elements. Since the first two clipboard lines are longer than the corresponding vector selections being pasted to, certain of the elements are not written. Conversely, the third clipboard line consists of only a single element, thus at the end of the paste operation, three of the final vector's selected elements remain unassociated.

Defining a Strategy Function

The final aspect of defining a partitioning scheme is the specification of a strategy function. Recall that the purpose of this function is to map to each concrete inner vector exactly one of the possibilities from the set of inner partitionings defined for the scheme. Again we choose to adopt a definition by convention to lessen the specificational complexity faced by users building partitioning schemes.

Consider a vector nest with outer vector V whose abstract form is divided into

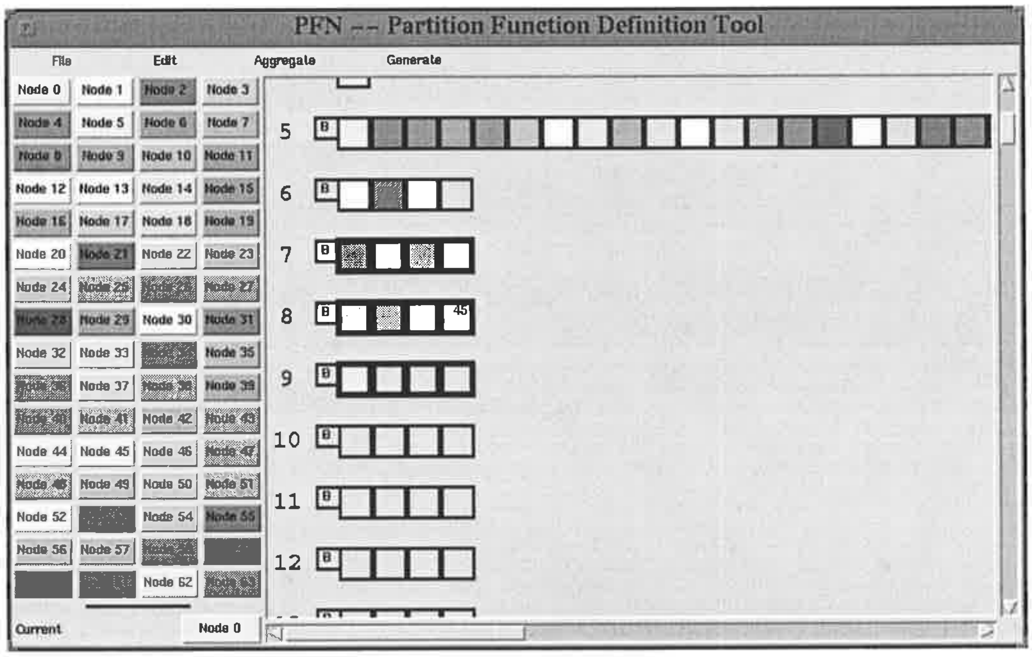


Figure 71. Pasting Multi-Vector Clipboard into a Selection

n partitioning elements. By Convention 1, this implies that the partitioning scheme for this nest has exactly n inner partitioning possibilities. Furthermore, this outer abstract form has a repetition pattern V_p' which maps indices of the concrete outer vector into exactly n sets. That is, given an index i (i.e., an inner vector), $V_p'(i)$ returns a number in the range $0 \ldots n-1$. This gives us the basis for a second definition by convention.

Convention 2:

By convention, the strategy function for a partitioning scheme is defined to be equivalent to the function defined by the repetition pattern of the outer vector

To illustrate this Convention at work, consider a vector nest where the outer vector has already been defined as being divided into 16 elements (and hence there are exactly 16 sub-vector partitioning possibilities). In the case where the outer vector has the **Block** repetition pattern associated with it, the strategy function for the partitioning scheme may be defined as: for all sub-vectors corresponding to indices of the outer vector which fall into block $j (0 \leq j \leq 15)$, choose possibility j. Alternatively, if the outer vector was defined to have the **Repeat** repetition pattern, its strategy function would be: for all sub-vectors corresponding to index j of the outer vector, choose possibility j mod 16.

A.3 Generating Partitioning Functions

Once a programmer has interacted with the PFN tool to define each aggregate within the program as a set of assigned partitioning elements, total partitioning of the program's data has been specified in terms of an abstract model. This section describes how from such a high-level specification of partitioning, the PFN tool is able to synthesize lower-level concrete forms. Specifically, we will consider the task of generating executable forms of the type required by the Adl runtime system. That is, we will consider the generation of partitioning functions, relative location functions and partitioning scheme implementations.

In the current PFN prototype, the process of generation is initiated when the user selects the "Generate C Source" option from the menu bar. The first action instigated by this user command is an analysis of the abstract vectors present within the current program. At this stage all that is considered is whether any elements from such vectors are yet to be allocated to a processing node (i.e., has yet to have its box coloured). The presence of any such elements means that the program's data layout is not fully specified, and hence concrete generation cannot commence. If the tool detects any unallocated elements, an error is produced at this stage and the synthesis of concrete form abandoned.

In the case that the specification is complete, PFN proceeds to generate C code forms of the partitioning function and relative location function for every vector partitioning within the program. Furthermore it creates C versions of any strategy functions required for partitioning schemes within the program. The remainder of this Section describes the techniques by which such forms are generated, detailing the mapping from abstract decomposition to C code in terms of a series of optimized, provably correct code templates.

A.3.1 Generating Partitioning Functions by Simple Template

The process of generating executable forms that are efficient and logically consistent is handled by a small set of *code templates*. Each template consists of a short segment of C code within which appear a number of *place-holders*, symbols that are replaced with actual concrete names or numbers when the template is instantiated by the generation routine. Each template embodies an aspect of the abstract model's semantics, representing it in terms of a low-level executable form.

Figures 72 and 73 show templates which implement a simple partitioning function (and its associated relative location function) for vectors partitioned with PFN. Figure 72's template is instantiated for each abstract vector form which was tagged with the **Block** repetition pattern by a user of the tool. The other figure's template handles the **Repeat** repetition pattern case.

```
#define BLOCK(els,div) (els%div?1+els/div:els/div)

host_id pf_name (int i, int len)
{
   int block_size;

   block_size = BLOCK (len, n_els);
   switch (i/block_size) {
   case 0:  return (node_mapping_for_element_0);
   case 1:  return (node_mapping_for_element_1);
   ⋮
   case n_els−1:  return (node_mapping_for_final_element);
   }
}

int rl_name (int i, int len)
{
   int block_size;

   block_size = BLOCK (len, n_els);
   return (i%block_size);
}
```

Figure 72. Code Templates for **Block** Partitioning Functions

In each of these illustrated templates, the symbol *n_els* is used to refer to the number of partitioning elements the programmer chose to divide the abstract vector into (i.e., the number of boxes which represented the vector upon the canvas). The symbol *node_mapping_for_element_j* is used to stand for the identity of the processing

```
host_id pf_name (int i, int len)
{
   switch (i%n_els) {
   case 0:  return (node_mapping_for_element_0);
   case 1:  return (node_mapping_for_element_1);
   :
   case n_els−1:  return (node_mapping_for_final_element);
   }
}

int rl_name (int i, int len)
{
   return (i/n_els);
}
```

Figure 73. Code Templates for **Repeat** Partitioning Functions

node to which partitioning element j was assigned during the PFN session (i.e., the colour of box j). In the C code generated by the tool, these placeholder symbols are replaced by concrete information; that is, a numeric constant.

Note that the functions generated by this process of template instantiation are computationally inexpensive. The most expensive case, that of a **Block** repeated vector of many abstract elements, still only causes at most 3 – 4 arithmetic operations and *n_els* integer comparisons. Since *n_els* is clearly a static constant, this is still an $O(1)$ operation.

The logical consistency of the `(pf,rl)` pairs generated by the tool can easily be proven correct. We consider only one such proof, that demonstrating `pf-rl` correspondence for the **Block** repeated template, here. A similar proof by construction is available for the **Repeat** pattern template.

Proof:
the `rl` *function shown in Figure 72 is a relative location function which corresponds to the partitioning function* `pf` *shown above it, and is thus pairable with it.*

We assume that a concrete vector V of length V_{len} has (at runtime) been annotated with a `(pf,rl)` pair generated from the template in Figure 72. Inspecting the `pf` function it is clear that for a given fixed length, it will always generate a constant `block_size`. If we consider the values generated by the expression `(i/block_size)` for different values of **`i`**, remembering that this division is an integer one (i.e., the mathematical operation *div*), we see that $\forall i \leq$**`block_size`** -1, the expression evaluates to 0. Generally $\forall i, n.$**`block_size`** $\leq i \leq (n+1).$**`block_size`** -1, the expression evaluates to n. We may safely assume that each of these blocks of n consecutive indices is mapped by **`pf`** to a unique processing node $\Pi_j, 0 \leq j < n\_els$. If we consider an arbitrary node from this set, two possibilities arise: either the node has been assigned exactly **`block_size`** elements, with indices $i : x.$**`block_size`** $\leq i \leq (x+1).$**`block_size`** -1 for some x, or the node has been assigned the last block consisting of the indices $i : (n\_els - 1).$**`block_size`** $\leq i < V_{len}$. We consider these two cases individually.

In the first case, the storage semantics of the Adl system enforce that a single block B with **`block_size`** slots be allocated, and that index $x.$**`block_size`** fill the zero slot, $x.$**`block_size`** fill the number one slot, and so on up to the final slot which contains index $(x+1).$**`block_size`** -1. We note that one representation of this mapping from index to slot is $i \rightarrow i \bmod$ **`block_size`**.

In the case where the chosen node is the bearer of the final block, a region of memory with $V_{len} - (n\_els - 1).$**`block_size`** $+ 1$ slots is allocated. The index $(n\_els - 1).$**`block_size`** is mapped into the zero slot, its successor into the next slot, and so on. The last element of the entire vector would be placed in slot number $V_{len} - (n\_els - 1).$**`block_size`**. Once again, for all indices i stored on this processing node, this slot mapping may be described by $i \rightarrow i \bmod$ **`block_size`**.

Thus we have the situation that for any node which holds indices from the vector V, the mapping function from index to slot may be described by the same function. Therefore, this function $i \rightarrow i \bmod$ **`block_size`** is universally valid across the vector and is hence the relative location function pairable with the partitioning function **`pf`**. □

A.3.2 Generating Partitioning Functions by Global Template

It is important to note that the templates shown in Figures 72 and 73 make an implicit assumption that each of the mappings for the abstract vector under consideration is to a unique processing node – in the case where two or more mappings within a vector are to the same processing node, an incorrect relative location function will be generated. To illustrate this limitation, consider the template-based generation for the abstract vector shown in Figure 74. This abstract vector is perfectly allowable under PFN's model of specification, yet an application of the appropriate (**Block**) templates described previously produces C code (shown in Figure 75) which implements an inconsistent view of vector storage.

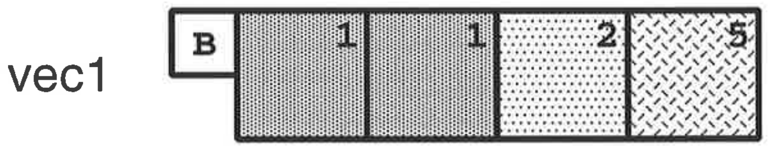


Figure 74. An Abstract Vector with Multiple Mappings to Node 1

To illustrate this point, consider the partitioning function `pf_vec1` shown in the figure: the function defines a division of a concrete vector into four equal blocks of B indices, two of which are to be stored within the memory of node 1. Recalling the storage model described in Section A.1), we know that the indices of the first of these blocks will occupy slots 0 to $B-1$ in node 1's memory. The indices from the second block (indices B through $2B-1$) will be stored within slots B through $2B-1$ accordingly. Now, for a relative location function `rl_vec` to correspond to this storage layout, it must provide the following sets of mappings:

$$\texttt{rl\_vec}\ 0 \mapsto 0, \quad \texttt{rl\_vec}\ 1 \mapsto 1, \ldots, \texttt{rl\_vec}\ (B-1) \mapsto (B-1)$$

$$\texttt{rl\_vec}\ B \mapsto B, \quad \texttt{rl\_vec}\ (B+1) \mapsto (B+1), \ldots, \texttt{rl\_vec}\ (2B-1) \mapsto (2B-1)$$

It is clear that the relative location function defined in Figure 74 does not adhere to the second set of mappings. Specifically, the template-generated function maps index index B to slot 0, index $B+1$ to slot 1 and so on. That is, there is a semantic discrepancy between the generated partitioning function and relative location function.

It is clear that such errors arise because of the limited knowledge used by the simple templates in their construction of relative location functions. Specifically, the difficulties are due to a lack of global analysis of the entire set of the abstract vector's colour assignments during the synthesis.

To address this limitation of the simple template approach, PFN provides a second set of generation templates which may be used in situations where an abstract vector

```
#define BLOCK(els,div) (els%div?1+els/div:els/div)

host_id pf_vec1 (int i, int len)
{
   int block_size;

   block_size = BLOCK (len, 4);
   switch (i/block_size) {
   case 0:  return (1);
   case 1:  return (1);
   case 2:  return (2);
   case 3:  return (5);
   }
}

int rl_vec1 (int i, int len)
{
   int block_size;

   block_size = BLOCK (len, 4);
   return (i%block_size);
}
```

Figure 75. Template Generated Code for Multiple Mapping Example

has been specified by a user to contain multiple mappings to the same node. This alternate approach generates functions which are less efficient than those supplied by the simple templates, but are guaranteed to generate a (`pf,rl`) pair that is consistent for any PFN specification. To limit the efficiency losses introduced by such functions, the tool only makes use of this approach in situations where generality is required — that is, where possible the simple templates are still used.

The global analytic approach to function synthesis is based on a simple extension to the existing template approach in which complex relationships introduced by the storage model (through multiple mappings) may be expressed. Partitioning functions are treated as before, the instantiations of simple templates. The generation of the corresponding relative location functions, however, is handled in a manner which incorporates a global picture of the abstract vector (rather than the per-element consideration made previously) to capture the full storage-model repercussions of the partitioning.

Figures 76 and 77 show the template forms for generating relative location functions in the **block** and **repeat** cases respectively. Central to these descriptions are the functions **colour_count** and **colour_total** which are the sources of global knowledge concerning the coloured PFN vector under consideration. The first of these functions accepts an index into the abstract vector, and returns a count of the number of indices which have identical colour to this specified partitioning element and which also precede it in the ordering of indices. Thus for the example abstract vector shown in Figure 74, **colour_count** (0) would return 0 (there are no indices preceding index 0), while **colour_count** (1) would evaluate to 1 indicating one predecessor of identical colouring.

The function **colour_total**, used in the definition of the relative location function for a multiply-mapped instance of a **repeat** abstract vector, simply accepts an index into that vector and returns the total number of elements which have the same colour. For the example vector, **colour_total** (0) would return 2 as would **colour_total**(1); calling the function with any other indices from the vector would evaluate to 1.

```
int rl_name (int i, int len)
{
   int block_size;

   block_size = BLOCK (len, n_els);
   switch (i/block_size);
   {
     case 0:  return (0 % block_size);
     case 1:  return (1 % block_size + colour_count (1) * block_size);
     case 2:  return (2 % block_size + colour_count (2) * block_size);
   :
     case n_els-1:  return ((n_els-1) % block_size +
                          colour_count (n_els-1) *block_size);
   }
}
```

Figure 76. Global Code Template for **Block** Relative Location Functions

It is obvious that relative location functions generated by this approach are more computationally costly (potentially involving numerous comparisons, then up to three mathematical operations) than those synthesized by instantiating the simple

```
int rl_name (int i, int len)
{
   switch (i%n_els)
   {
   case 0:  return (0/n_els);
   case 1:  return (1/n_els * colour_total(1) + colour_count(1));
   case 2:  return (2/n_els * colour_total(2) + colour_count(2));
   ⋮
   case n_els−1:  return (i/(n_els−1) * colour_total(n_els−1) +
                              colour_count(n_els−1));
   }
}
```

Figure 77. Global Code Template for **Repeat** Relative Location Functions

templates discussed previously. They do, however, offer guaranteed correspondence with the associated partitioning functions and are computationally inexpensive as a model for the complex situation (uneven distribution of partitioning elements) under consideration.

A.3.3 Generating Partitioning Schemes

The task of generating code implementing a partitioning scheme for a vector of vectors may be viewed as a simple extension of the basic case presented above. Of the three scheme components, two — the outer vector partitioning and the set of inner partitioning possibilities — may be considered as merely a collection of simple partitionings. Thus code may be generated for each by application of the templates shown previously.

Generating the strategy function component of the scheme, however, requires an extension of the presented technique. The templates that are required to implement schemes with both **Block** and **Repeat** patterned outer vectors can be easily defined as variants of the corresponding partitioning function templates. Rather than having these templates return a host identifier indicating which host owns the element, we modify them to return a `(pf,rl)` pair indicating which of the possible inner-vector partitionings this index of the outer vector should receive.

Figure 78 shows the strategy function for a scheme whose outer vector was given the **Block** repetition pattern. In this template the following symbols are used:

- *n_els* refers to the number of partitioning elements into which the outer vector was divided,
- *length_of_outer_v* denotes the length of the outer vector,
- *pf_and_rl_functions_for_option_j* refers to the pair of partitioning and relative location functions generated for one of the inner-partitioning possibilities.

```
#define BLOCK(els,div) (els%div?1+els/div:els/div)
typedef host_id (*pfunc) (int,int);
typedef int (*rlfunc) (int,int);

(pfunc,rlfunc) strat_name (int i)
{
   int block_size;

   block_size = BLOCK (length_of_outer_v, n_els);
   switch (i/block_size) {
   case 0:  return (pf_and_rl_functions_for_option_0);
   case 1:  return (pf_and_rl_functions_for_option_1);
   ⋮
   case n_els−1:  return (pf_and_rl_functions_for_final_option);
   }
}
```

Figure 78. Code Templates for Partitioning Scheme Strategy Function (**Block** Outer)

A.3.4 Extending to the Arbitrary Nesting Case

In the case of an partitioning scheme for an aggregate nest deeper than two, a slightly different form of the partitioning scheme template is required. Rather than returning the `(pf,rl)` pair of functions for each of the possible inner-vector choices, these templates return a (different) partitioning scheme for each inner-vector possibility. This generalizes the generation of partitioning scheme code to an arbitrary depth.

A.3.5 Extended Code Generation Example

Consider the Adl program shown in Figure 79, a simple program which takes a vector of vectors and reverses the ordering of the inner vectors within the outer. For example, given the input `[[1,2],[3],[4,5]]` the program generates `[[4,5],[3],[1,2]]`.

```
main inp:  vof vof real :=
  let
    len := #inp;
    f x := inp!(len-x-1)
  in
    map f (iota len)
  endlet
```

Figure 79. A Simple Adl Program

This program has two partitioning introducing elements: the call to the vector-creating function `iota`, and the nested vector created at the time the program reads its input.

Using PFN, we consider the partitioning of this program's aggregates in an abstract sense. First we consider an abstract entity `iota_instance` which will represent the vector created by the call to `iota`. Bringing this abstract form onto the canvas, we divide it into four partitioning elements and attach the **Repeat** repetition pattern with it. We now colour the boxes of the abstract form, assigning element 0 to reside upon processing node 3, element 1 to node 6, element 2 to node 9 and element 3 to node 12. Once these four node associations are made, this abstract vector (and hence all concrete instantiations of it) are fully partitioning specified.

We next consider the input to the program. Since this is a vector nest, we will effectively be generating a partitioning scheme rather than a simple partitioning function. We consider an abstract entity called `input_outer` — on the PFN canvas we give this abstract vector the **Block** pattern and divide it into only two partitioning elements. The first is allocated to node 17, the second to node 23. The fact that we have divided this abstract form into two elements means that, by convention, we will be considering the definition of exactly two inner-vector partitioning possibilities. This set can be collectively named `input_outer[]->inner`, and can be brought *en masse* onto the canvas. Once we have done this, we choose to divide the first possibility into four partitioning elements and grant it the **Block** pattern;

the second we divide into only a single partitioning element and give a **Repeat** pattern. By gesture, we colour the first abstract vector's representation as follows: element 0 → node 7, element 1 → node 8, element 2 → node 9, element 3 → node 10. The second inner possibility's sole partitioning element is mapped onto node 0.

With the partitioning specification thus completed, the user clicks initiates the generation process. The C code file that results contains the definitions for four partitioning functions (and their relative location functions) plus the definition of a strategy function for the partitioning scheme.

```
host_id pf_iota_instance (int i, int len)
{
   switch (i%4) {
   case 0:  return (3);
   case 1:  return (6);
   case 2:  return (9);
   case 3:  return (12);
   }
}

int rl_iota_instance (int i, int len)
{
   return (i/4);
}
```

Figure 80. Code Implementing the Partitioning of the `iota` Instance

Figure 80 shows the code which pertains to the `iota` instance. Figure 81 details the generated code which describes the partitioning of the outer vector of the input nest. Note that the nodes named within this partitioning function are not intended to convey memory spaces which will hold numeric elements of the input vector. Rather they are memories which will store vector descriptors for the inner-vectors of the input.

The partitioning function declarations for the two inner-vector partitioning possibilities for the input are shown in Figure 82.

Figure 83 implements the strategy function for the input vector's partitioning scheme. Recall that the outer vector has the **Block** repetition pattern, hence we use the strategy function template appropriate to that pattern.

```
#define BLOCK(els,div) (els%div?1+els/div:els/div)

host_id pf_input_outer (int i, int len)
{
   int block_size;

   block_size = BLOCK (len, 2);
   switch (i/block_size) {
   case 0:  return (17);
   case 1:  return (23);
   }
}

int rl_input_outer (int i, int len)
{
   int block_size;

   block_size = BLOCK (len, 2);
   return (i%block_size);
}
```

Figure 81. Code Implementing the Partitioning of Outer Vector of Input Nest

Now consider the situation where all of these concrete partitioning codes have been compiled and linked with the implementation of the program in question. The program is then invoked and passed the input `[[1,2],[3],[4,5]]`.

As the program begins its execution it first must build the distributed concrete vectors which hold its input. The task of constructing the outer vector of the nest begins first: the length of that vector is now known to be 3, which when divided into two blocks gives a decomposition into the block `(index0, index1)` and the block `(index 2)`. Thus, the outer vector will be divided so as two of the inner vector's descriptors will reside on processing node 17 with the final one upon node 23.

The next stage of input construction concerns building the three inner vectors of the nest. Before we can proceed with this, however, we must determine which partitioning functions (and relative location functions) are associated with each. Executing the input partitioning scheme's strategy function for each of the indices $0, 1$ and 2 we learn that the first two of the inner vectors are to receive the partitioning function `pf_input_inner_1` while the third is to receive the function

```
#define BLOCK(els,div) (els%div?1+els/div:els/div)

host_id pf_input_inner_1 (int i, int len)
{
   int block_size;

   block_size = BLOCK (len, 4);
   switch (i/block_size) {
   case 0:  return (7);
   case 1:  return (8);
   case 2:  return (9);
   case 3:  return (10);
   }
}

int rl_input_inner_1 (int i, int len)
{
   int block_size;

   block_size = BLOCK (len, 4);
   return (i%block_size);
}

host_id pf_input_inner_2 (int i, int len)
{
   switch (i%1) {
   case 0:  return (0);
   }
}

int rl_input_inner_2 (int i, int len)
{
   return (i/1);
}
```

Figure 82. Code Implementing the Partitioning Possibilities for Inner Vectors of Input Nest

```
#define BLOCK(els,div) (els%div?1+els/div:els/div)
typedef host_id (*pfunc) (int,int);
typedef int (*rlfunc) (int,int);

(pfunc,rlfunc) strat_input_outer (int i)
{
   int block_size;

   block_size = BLOCK (input_outer.len, 2);
   switch (i/block_size) {
   case 0:  return ((pf_input_inner_1,rl_input_inner_1));
   case 1:  return ((pf_input_inner_2,rl_input_inner_2));
   }
}
```

Figure 83. Code Implementing Strategy Function for Input Nest

`pf_input_inner_2`.

With this knowledge we may now consider the process of allocating these three inner vectors. Considering the first inner vector, we can see that (with a block size of 1) the first vector index (data item = 1 will be stored on node 7; the second vector index (data item = 2) will reside on node 8.

Considering the second inner input vector, it will again be divided into blocks. Since the concrete vector has only one element, the block size will be one. Thus the vector's only element (data item = 3) will be mapped to node 7.

Coming to the final inner vector, this time we are to distribute the data according to `pf_input_inner_2`. Thus, both indices of this vector (data items = 4 and 5) are mapped to node 0.

With the construction of the input vector (and its partitioning among the machine), execution can begin. First we calculate the length of the input vector, which is 3. We then call `iota` with this length as argument. The semantics of this call are to generate a new vector of length 3 with the partitioning function `pf_iota_instance`. The values stored within this vector will be the numbers 0 (in index 0), 1 (in index 1) and 2 (in index 2). Considering the partitioning function in question, index 0 will be stored on node 3, index 1 on node 6 and index 2 of node 9.

The call to the first order operator `map` also generates a new vector (which happens to be the program's output). Note, however, that the programmer does not specify the partitioning of this aggregate. Instead, the partitioning function for this vector is defined to be identical to that of the input to map. That is, in this case, the result of map is partitioned according to `pf_iota_instance`. Since it also is of length 3, its indices (which store vector descriptors for inner vectors) are stored on nodes $3, 6$ and 9.

A.4 Related and Future Work

We have described a graphical interactive environment, PFN, which allows for the high-level specification of data layout for a nested data parallel program. Such a tool is essential to any language system which allows for data to be distributed between memory-spaces in complex or arbitrary patterns. Little research has addressed the needs of such systems in the context of programming languages, primarily because language implementations which support such generality in data placement are rare. In the context of complex decomposition of finite element meshes, however, similar tools have previously been proposed [29].

The novelty of our approach lies in its support for the partitioning specification of structured aggregates whose extents are likely unknown until the time they are instantiated. While trivial data layout specifications may be made for such structures (e.g., map all levels of the aggregate identically), these are unlikely to be generally efficient. The *partitioning schemes* we introduce provide for a flexible and expressive means of defining layouts for structures.

PFN greatly simplifies the specification of partitioning for both the simple (non-structured aggregate) case and this more complex situation. It supplies an abstract representation of each aggregate within a program and allows these entities to be partitioned into collections of *partitioning elements*. By gesture, the user may assign such elements to processing nodes of the target machine, thus defining an abstract decomposition of the aggregate across the memory spaces of that machine. Once such a specification is complete (i.e., every element of every aggregate has been assigned to a processing node), the tool can automatically transform this high-level specification into a concrete executable form by applying a set of proven and efficient C code templates.

Experience with using the PFN in benchmarking the Adl implementation has shown it to provide an enormous simplification over the hand-coding of complex partitioning details. The tool is, however, quite simplistic in the facilities it presents users for defining allocations of data. Many possibilities exist for future expansion of this functionality (e.g., allowing for the user to specify that two abstract aggregates will always be partitioned identically). The ultimate goal of research into this area of visual specification of data partitioning is for users of computational systems, such as ours, which support arbitrary data decomposition to be presented with an environment which abstracts the complex problem of data specification to the point where it becomes a series of simple and intuitive decisions. In PFN, we believe, we have gone a long way towards meeting this ideal.

Appendix B

Source Code for Evaluated Programs

This appendix contains the full source for each of the Adl, NESL and CM Fortran programs whose performance is benchmarked in Chapter 8. Descriptions of the language features which appear in the Adl codes may be found in Section 7.1. An account of NESL's features is available in [18, 21]; CM Fortran details can be found in various Thinking Machines manuals [127, 130, 128, 129].

B.1 Flat Data-Parallel Codes

B.1.1 Flat `map` Program

Adl Source

```
add1 n := n+1;
main vec: vof real := map (add1, vec)
?
```

NESL Source

```
function main (v) = { z+1 : z in v }
```

CM Fortran Source (Aggregate Level +)

```
      program forces
      integer, parameter :: LEN=1000000
```

```
C       declare a source and destination vector of length 1000000
        real src (LEN), dest (LEN)

C       declare that both vectors should be spread across CM nodes
CMF$    layout src (:news), dest (:news)

C       set each element in src to value 1
        src = 1

C       in parallel, add one to every element of src
        dest = src + 1
        end
```

CM Fortran Source (`forall`)

```
        program forces
        integer, parameter :: LEN=1000000
        integer i

C       declare a source and destination vector of length 1000000
        real src (LEN), dest (LEN)

C       declare that both vectors should be spread across CM nodes
CMF$    layout src (:news), dest (:news)

C       set each element in src to value 1
        src = 1

C       in parallel, add one to every element of src
        forall (i=1:LEN) dest(i) = src(i) + 1
        end
```

B.1.2 Flat scan Program

Adl Source

```
add (x,y) := x+y;
main vec: vof real := scan (add,0.0,vec)
?
```

NESL Source

```
function main (v) = +-scan (v)
```

CM Fortran Source

```
      program scan
      INCLUDE '/usr/include/cm/CMF_defs.h'
      integer, parameter :: LEN=1000000

C     declare a source and destination vector of length 1000000
      real src (LEN), dest (LEN)

C     declare that both vectors should be spread across CM nodes
CMF$  layout src (:news), dest (:news)

C     set each element in src to value 1
      src = 1

C     perform parallel prefix add across vector src
      call CMF_SCAN_add (dest,src,CMF_NULL,1,CMF_UPWARD,
     +                   CMF_INCLUSIVE,CMF_NONE,.TRUE.)
      end
```

B.1.3 Irregular Vector Index Program

Adl Source

```
% program inputs: A ----- source vector (contains values)
%                 perm -- permutation vector (contains indices)
main (A:vof real, perm:vof int) :=
 let
   deref x := A!x  % returns the x'th element of A
 in
   map (deref,perm) % apply fn deref to all elements in perm vector;
                    % each such element is an index into A
 endlet

?
```

NESL Source (map-index method)

```
  % in parallel use element of perm vector as an index into %
  % source vector A                                        %
function neslperm1 (A,perm) = { A[i]: i in perm };
```

NESL Source (inbuilt Perm method)

```
  % use NESL's inbuilt perm operator to perform parallel   %
  % dereference of A by each element of perm               %
function neslperm2 (A,perm) = A -> perm;
```

CM Fortran Source

```
       program perm
       integer, parameter :: LEN1=100
       integer, parameter :: LEN2=50

C      declare source, destination and permutation vectors
       real a (LEN1), dest (LEN2)
       integer perm (LEN2)

C      set values for the vectors
       a = [ 0.0, 1.0, 2.0, ... ]
       perm = [ 39, 94, 57, ... ]
```

```
C       use CMF's vector-valued dereference to specify a
C       parallel permutation
        dest = a(perm)

        end
```

B.2 Nested Data-Parallel Codes

B.2.1 Nested map-scan Program

Adl Source

```
add (x,y) := x+y;            % binary add function
acc v := scan (add,0.0,v); % use fn add in a parallel prefix operation

  % perform the function acc for every inner vector of the nest
main vvec: vof vof real := map (acc,vvec)
?
```

NESL Source

```
  % for each v in vv, perform a parallel prefix add operation %
function main (vv) =
  {+-scan (v) : v in vv}
```

CM Fortran Source (Outer Serialization)

```
      program forces
      INCLUDE '/usr/include/cm/CMF_defs.h'
      integer, parameter :: OLEN=200
      integer, parameter :: ILEN=50

C     declare source and destination vectors
      real src (OLEN,ILEN), dest (OLEN,ILEN)
      integer i

C     set all values in src to 1.0
      src = 1.0

C     serially loop through the outer dimension, performing a
C     parallel prefix operation on each column of src during
C     each iteration
      do i=1,OLEN
        call CMF_SCAN_add (dest(1,:),src(1,:),CMF_NULL,1,CMF_UPWARD,
     +                    CMF_INCLUSIVE,CMF_NONE,.TRUE.)
      enddo

      end
```

CM Fortran Source (Inner Serialization)

```
      program forces
      INCLUDE '/usr/include/cm/CMF_defs.h'
      integer, parameter :: OLEN=20
      integer, parameter :: ILEN=5

C     declare source and destination vectors
      real src (OLEN,ILEN), dest (OLEN,ILEN)
      integer i

C     set all values in src to 1.0
      src = 1.0

C     copy first row of src to first row of dest
      dest (:,1) = src (:,1)

C     serially iterating down rows of src (inner dimension),
C     perform a parallel addition across columns at each
C     iteration. We take the previous column from dest
C     and accumulate it with the current column from src
      do i=1,ILEN-1
        dest (:,i+1) = dest (:,i) + src (:,i+1)
      enddo
      end
```

B.2.2 `meshcomp`: A Simple Finite Element Mesh-Based Computation

Adl Source

```
% Program Input: vector of nodes + vector of elements
%                each node is a four-tuple --
%                  *  x position of node in plane (real)
%                  *  y position of node in plane (real)
%                  *  type of node (0 = non-boundary,
%                         others denote different boundary conds)
%                  *  a node number (0..num_nodes-1)
%                each element is a two-tuple --
%                  *  a vector of length 3, defining which
%                     three nodes bound the triangular element
%                  *  an element number (0..num_elts-1)

% functions to select an element from a two-tuple
first (x,y) := x;
second (x,y) := y;

% functions to select fields from a node
xof (x,y,i,n) := x;
yof (x,y,i,n) := y;
iof (x,y,i,n) := i;
nof (x,y,i,n) := n;

% useful numeric functions
sqrt x := x^0.5;
fabs x := if x >= 0.0 then x else -x endif;

% function to compute the straight-line distance between two nodes
dist (p1,p2) :=
  let
    x1 := xof (p1);
    y1 := yof (p1);
    x2 := xof (p2);
    y2 := yof (p2)
  in
    sqrt ((y2-y1)^2 + (x2-x1)^2)
  endlet;
```

```
% function to pairwise add two vectors of equal length
add_vec (v1,v2) :=
  let
     g x := v1!x + v2!x
  in
     map (g,iota #v1)
  endlet;

% entry point
main mesh:(vof (real,real,int,int), vof (vof int,int)) :=
  let
    nodes := first (mesh);
    elements := second (mesh);

    % function to compute the area contained within a
    % triangular element
    area el :=
     let
       conn := first (el);
       p1 := conn!0;
       p2 := conn!1;
       p3 := conn!2;
       n1 := nodes!p1;
       n2 := nodes!p2;
       n3 := nodes!p3;
       a := dist (n1,n2);
       b := dist (n1,n3);
       c := dist (n2,n3);
       s := (a+b+c)/2
     in
       sqrt (s*(s-a)*(s-b)*(s-c))
     endlet;
```

```
% function to compute the stiffness matrix for the
% laplace iteration across the irregular mesh
compute_stiff (nodes,elements) :=
  let
    % compute the stiffness associated with a single element
    f el :=
     let
       a := area (el);
       conn := first (el);

       % compute the stiffness contributed by one edge of the
       % triangular element
       g ind :=
        let
          n := conn!ind;
          n1 := conn!((ind+1) mod 3);
          n2 := conn!((ind+2) mod 3);
          node := nodes!n;
          node1:= nodes!n1;
          node2:= nodes!n2;
          aa := xof (node);
          bb := yof (node);
          dx1 := xof (node1) - aa;
          dx2 := xof (node2) - aa;
          dy1 := yof (node1) - bb;
          dy2 := yof (node2) - bb
        in
          (dx1*dx2 + dy1*dy2)/a
        endlet;

        t := iota 3
    in
       map (g,t)  % perform g for each of 0,1 and 2 in //
     endlet
  in
    map (f,elements) % perform f for each element in //
  endlet;

  % compute the stiffness matrix: *** PHASE ONE ***
  stiff := compute_stiff (mesh);
```

```
% define a zero vector of equal length to node vector
zero x := 0.0;
zerovec := map (zero,iota #nodes);

% compute the diagonal matrix contribution (a vector of
% length = # nodes) for one element
diag_per el :=
  let
    iter := iota #nodes;
    conn := first (el);
    id := second (el);
    n1 := conn!0;
    n2 := conn!1;
    n3 := conn!2;

    f n := if n = n1 then
             stiff!id!1 + stiff!id!2
           else if n = n2 then
             stiff!id!0 + stiff!id!2
           else if n = n3 then
             stiff!id!0 + stiff!id!1
           else
               0.0
           endif
           endif
           endif
  in
    map (f,iter)  % apply f in // across 0, 1,.., num_nodes-1
  endlet;

% compute a vector containing the diagonal matrix contributions
% for every element (ie a vector of vectors): *** PHASE TWO ***
diag_tmp := map (diag_per,elements);

% now collapse this vector by summing across the inner dimension;
% ie by adding all the inner vectors together: *** PHASE THREE ***
diag := reduce (add_vec,zerovec,diag_tmp);
error_thresh := 0.001;

% expression to count the number of nodes which are NOT
% on the boundary: *** PHASE FOUR ***
```

```
non_bdy_nodes :=
  let
    plus (x,y) := x+y;
    f n :=
      let
        bd_code := iof (n)
      in
        if bd_code > 0 then 1
        else                0
        endif
      endlet;

    non_bdy_flags := map (f,nodes)
  in
    reduce (plus,0,non_bdy_flags)
  endlet;

% set an initial value phi (which represents the fluid density
% at the given point) for every node to begin the Laplace iteration
% *** PHASE FIVE ***
 begin_phi :=
   let
     bdy_cond n :=
      let
        bd_code := iof (n);
        ypos := yof (n)
      in
        % the nodes on the boundary (ie bd_code > 1)
        % get a special value, others get 0.0
        if bd_code = 1      then 1.0
        else if bd_code = 2 then -1.0
        else if bd_code = 3 then 0.02 * ypos
        else                     0.0
        endif
        endif
        endif
      endlet
   in
     map (bdy_cond,nodes)
   endlet;
```

```
% define a termination function for the iteration
not_done (phi,err) := err >= error_thresh;

% the body of the Laplace iteration: *** PHASE SIX ***
jacobi (phi,err) :=
  let
    % function to compute the effects on one element due to
    % each node (returns a vector of length # nodes)
    phi_per el :=
      let
        iter := iota #nodes;
        conn := first (el);
        id := second (el);
        n1 := conn!0;
        n2 := conn!1;
        n3 := conn!2;

        f n := if n = n1 then
                 phi!n3 * stiff!id!1 +
                 phi!n2 * stiff!id!2
               else if n = n2 then
                 phi!n1 * stiff!id!2 +
                 phi!n3 * stiff!id!0
               else if n = n3 then
                 phi!n1 * stiff!id!1 +
                 phi!n2 * stiff!id!0
               else
                 0.0
               endif
               endif
               endif
      in
        map (f,iter) % apply f in // for 0,1,...,num_nodes-1
      endlet;
```

```
% function to calculate the new value for each node by
% first computing the effect contributed by each node upon
% each element (a vector of vectors), then collapsing the
% inner dimension by vector addition
phinew :=
  let
    phi_tmp := map (phi_per,elements)
  in
    reduce (add_vec,zerovec,phi_tmp)
  endlet;

% perform one step of the iteration, computing
% the change in value for a NON-BOUNDARY node
% (the others are fixed during the computation)
one_step n :=
  let
    my_idx := nof (n)
  in
    if iof(n) = 0 then
      phinew!my_idx / diag!my_idx - phi!my_idx
    else
      0.0
    endif
  endlet;

% for all nodes in //, compute the change in value
delta_phi := map (one_step,nodes);

% we now know how much the value of each node changes by,
% so we can compute the new value phi_prime for each node
phi_prime :=
   let
     increment n :=
       let
         myidx := nof (n)
       in
         phi!myidx + delta_phi!myidx
       endlet
   in
     map (increment,nodes)
   endlet;
```

```
          % also using the vector of node-deltas, we sum the absolute
          % change in values to give an error estimate
          err_red (a,b) := fabs (a) + fabs (b);
          new_err := reduce (err_red,0.0,delta_phi)
        in
          (phi_prime, new_err / non_bdy_nodes)  % return the vector of new values
                                                % plus an error estimate per node
        endlet
  in
    while (jacobi,not_done,(begin_phi,error_thresh))  % the iteration
  endlet
?
```

NESL Source

```
% Program Input: vector of nodes + vector of elements              %
%                each node is a four-element record --             %
%                  * x position of node in plane (real)            %
%                  * y position of node in plane (real)            %
%                  * type of node (0 = non-boundary,               %
%                        others denote different boundary conds) %
%                  * a node number (0..num_nodes-1)                %
%                each element is a two-element record --           %
%                  * a vector of length 3, defining which          %
%                    three nodes bound the triangular element      %
%                  * an element number (0..num_elts-1)

datatype node_r (float,float,int,int);
datatype element_r ([int],int);

% functions to select fields from a node %
function xof (z) =
  let node_r(x,y,i,n) = z
  in x;
function yof (z) =
  let node_r(x,y,i,n) = z
  in y;
function iof (z) =
  let node_r(x,y,i,n) = z
  in i;
function nof (z) =
  let node_r(x,y,i,n) = z
  in n;

% functions to select fields from an element %
function connof (z) =
  let element_r(c,n) = z
  in c;
function elidof (z) =
  let element_r(c,n) = z
  in n;

% function to compute the straight-line distance between two nodes %
function distance (p1,p2) =
  let
    x1 = xof (p1);
    y1 = yof (p1);
    x2 = xof (p2);
    y2 = yof (p2);
  in
    sqrt ((y2-y1)*(y2-y1) + (x2-x1)*(x2-x1));

% define a simple mesh %
nodes = [node_r(0.000000, 0.000000,1,0), node_r(0.500000, 1.000000,0,1),
         node_r(1.000000, 2.000000,3,2), node_r(1.500000, 1.000000,2,3),
         node_r(2.000000, 0.000000,0,4), node_r(1.000000, 0.000000,0,5)];

elements = [element_r([0,1,5],0),  element_r([1,2,3],1),
            element_r([1,3,5],2),  element_r([3,4,5],3)];

% function to compute the area contained within a
% triangular element
function area_tri (nodes,element) =
  let
    conn = connof (element);
    p1 = conn[0];
    p2 = conn[1];
    p3 = conn[2];
    n1 = nodes[p1];
    n2 = nodes[p2];
    n3 = nodes[p3];
    a = distance (n1,n2);
    b = distance (n1,n3);
    c = distance (n2,n3);
    s = (a+b+c)/2.0;
  in
    sqrt (s*(s-a)*(s-b)*(s-c));

% compute the stiffness contributed by one edge of the triangular element %
function compute_stiff_g (nodes,a,conn,i) =
  let
    n = conn[i];
    n1 = conn[rem(i+1,3)];
    n2 = conn[rem(i+2,3)];
    node = nodes[n];
    node1 = nodes[n1];
    node2 = nodes[n2];
    aa = xof (node);
    bb = yof (node);
    dx1 = xof (node1) - aa;
    dx2 = xof (node2) - aa;
    dy1 = yof (node1) - bb;
    dy2 = yof (node2) - bb;
  in
    (dx1*dx2 + dy1*dy2)/a;

% compute the stiffness associated with a single element %
function compute_stiff_f (nodes,el) =
  let
    a = area_tri (nodes,el);
    conn = connof (el);
  in
    {compute_stiff_g (nodes,a,conn,z): z in [0,1,2]};

function diag_per_f (n1,n2,n3,id,stiff,i) =
  if i == n1 then
    stiff[id][1] + stiff[id][2]
  else if i == n2 then
    stiff[id][0] + stiff[id][2]
  else if i == n3 then
    stiff[id][0] + stiff[id][1]
  else
    0.0;

% compute the diagonal matrix contribution (a vector of  %
% length = # nodes) for one element                      %
function diag_per (stiff,el) =
  let
    iter = [0:#nodes];
    conn = connof (el);
    id = elidof (el);
    n1 = conn[0];
    n2 = conn[1];
    n3 = conn[2];
  in
    {diag_per_f (n1,n2,n3,id,stiff,z): z in iter};
```

```
error_thresh = 0.001;

% translates boundary codes into initial values %
function bdy_cond (n) =
  let
    bd_code = iof(n);
  in
    if bd_code == 1      then 1.0
    else if bd_code == 2 then -1.0
    else if bd_code == 3 then 0.02*yof(n)
    else                      0.0;

function phi_per_f (phi,stiff,n1,n2,n3,id,n) =
  if n == n1 then
     phi[n3]*stiff[id][1] + phi[n2]*stiff[id][2]
  else if n == n2 then
     phi[n1]*stiff[id][2] + phi[n3]*stiff[id][0]
  else if n == n3 then
     phi[n1]*stiff[id][1] + phi[n2]*stiff[id][0]
  else
     0.0;

% function to compute the effects on one element due to  %
% each node (returns a vector of length # nodes)          %
function phi_per (phi,stiff,el) =
  let
    iter = [0:#nodes];
    conn = connof (el);
    id = elidof (el);
    n1 = conn[0];
    n2 = conn[1];
    n3 = conn[2];
  in
    {phi_per_f (phi,stiff,n1,n2,n3,id,z): z in iter};

% perform one step of the iteration, computing  the change in value     %
% for a NON-BOUNDARY node (the others are fixed during the computation) %
function one_step (diag,phinew,phi,n) =
  let
    my_idx = nof (n);
  in
    if iof(n) == 0 then
      phinew[my_idx] / diag[my_idx] - phi[my_idx]
    else
      0.0;

% do one iteration of the Laplace %
function jacobi (nodes,elements,stiff,diag,non_bdy_nodes,phi) =
  let
    phi_tmp = flatten ({phi_per(phi,stiff,z): z in elements});
    phinew = {sum(phi_tmp->[i:3*#elements+i:3]): i in [0:#nodes]};
    delta_phi = {one_step(diag,phinew,phi,z): z in nodes};
    phi_prime = {phi[nof(z)]+delta_phi[nof(z)]: z in nodes};
    new_err = sum ({abs(z): z in delta_phi});
  in
    (phi_prime,new_err/float(non_bdy_nodes));

% a recursive formulation of the iterative loop %
function jac_iter (nodes,elements,stiff,diag,non_bdy_nodes,phi,error,i) =
  if error >= error_thresh then
    let
      (new_phi,new_err) = jacobi (nodes,elements,stiff,diag,non_bdy_nodes,phi)
    in
      jac_iter (nodes,elements,stiff,diag,non_bdy_nodes,new_phi,new_err,i+1)
  else
    (phi,i);

% entry point
function main (ns,es) =
  let
    % compute the stiffness matrix: *** PHASE ONE ***                  %
    stiff = {compute_stiff_f (ns,z): z in es};

    % compute a vector containing the diagonal matrix contributions   %
    % for every element (ie a vector of vectors): *** PHASE TWO ***   %
    diag_tmp = flatten ({diag_per (stiff,z): z in es});

    % now collapse this vector by summing across the inner dimension;%
    % ie adding all the inner vectors together: *** PHASE THREE ***   %
    diag = {sum(diag_tmp->[z:3*#es+z:3]): z in [0:#nodes]};
  % expression to count the number of nodes which are NOT on the   %
  % boundary: *** PHASE FOUR ***                                   %
  non_bdy_nodes = count ({iof(z) == 0: z in ns});

  % set an initial value phi (which represents the fluid density   %
  % at given point) for every node to begin the Laplace iteration  %
  begin_phi = {bdy_cond(z): z in ns};
in
  % the body of the Laplace iteration: *** PHASE SIX ***           %
  jac_iter (ns,es,stiff,diag,non_bdy_nodes,begin_phi,error_thresh,0);
```

CM Fortran Source

```
C       The irregular mesh is composed of a set of nodes and a set of
C       triangular elements which join those nodes. This program
C       represents nodes by way of three vectors:
C           nodex  ... holds the x position of each node
C           nodey  ... holds the y position of each node
C           nodei  ... holds a code which identifies the node type
C                      (0=non-boundary, other values denote boundary types)
C       Elements are represented by a two-dimensional matrix. Each
C       column of the matrix (all length 3) denotes the node numbers
C       which represent the boundary nodes of this element

        program meshcomp
        integer, parameter :: NNODES=6,NELEMNTS=4
        real nodex (NNODES), nodey (NNODES)
        integer nodei (NNODES), noden (NNODES)
        integer elmtc (NELEMNTS,3), elmtid (NELEMNTS)

C       values computed during the iteration
        real stiff (NELEMNTS,3)
        real diag (NNODES)
        real phi (NNODES), phinew (NNODES), dest_addr (NNODES)
        real delta_phi (NNODES)
        integer tt (NELEMNTS,3)
        integer tmp1 (NELEMNTS,3), tmp2 (NELEMNTS,3)
        real n1 (NELEMNTS,3), n2 (NELEMNTS,3)
        real s1 (NELEMNTS,3), s2 (NELEMNTS,3)
        real error_thresh, error
        logical non_bdy_mask (NNODES)
        integer non_bdy_nodes
        real lens (NELEMNTS,3)
        real ess (NELEMNTS), area (NELEMNTS)
        integer en0 (NELEMNTS), en1 (NELEMNTS), en2 (NELEMNTS)
        real dx1 (NELEMNTS), dx2 (NELEMNTS), dy1 (NELEMNTS), dy2 (NELEMNTS)


C       define a simple grid
        nodex = [0.0,0.5,1.0,1.5,2.0,1.0]
        nodey = [0.0,1.0,2.0,1.0,0.0,0.0]
        nodei = [1,0,3,2,0,0]
        elmtc(1,:) = [1,2,6]
        elmtc(2,:) = [2,3,4]
        elmtc(3,:) = [2,4,6]
        elmtc(4,:) = [4,5,6]

        diag = 0.0

C       BEGIN PHASE ONE
C       For each element compute the lengths of the three
C       edges which define its triangular space. Note this
C       is a serialized NDP computation.
        forall (i=1:NELEMNTS) lens(i,1) =
     +        ((nodey(elmtc(i,2))-nodey(elmtc(i,1)))**2 +
     +         (nodex(elmtc(i,2))-nodex(elmtc(i,1)))**2)**0.5
        forall (i=1:NELEMNTS) lens(i,2) =
     +        ((nodey(elmtc(i,3))-nodey(elmtc(i,1)))**2 +
     +         (nodex(elmtc(i,3))-nodex(elmtc(i,1)))**2)**0.5
        forall (i=1:NELEMNTS) lens(i,3) =
     +        ((nodey(elmtc(i,3))-nodey(elmtc(i,2)))**2 +
     +         (nodex(elmtc(i,3))-nodex(elmtc(i,2)))**2)**0.5

C       compute the area of each triangular element
        ess = SUM (lens,2)/2
        area = (ess*(ess-lens(:,1))*(ess-lens(:,2))*(ess-lens(:,3)))**0.5
        tt = elmtc

C       loop A:
        do i=1,3
C       compute the stiffness matrix
          en0 = tt(:,i)
          tt = CSHIFT (tt,2,1)
          en1 = tt(:,i)
          tt = CSHIFT (tt,2,1)
          en2 = tt(:,i)
          dx1 = nodex(en1) - nodex(en0)
          dx2 = nodex(en2) - nodex(en0)
          dy1 = nodey(en1) - nodey(en0)
          dy2 = nodey(en2) - nodey(en0)
          stiff (:,i) = (dx1*dx2 + dy1*dy2)/area
```

```
C       END PHASE ONE
C       PHASE TWO: implicitly computed
C       BEGIN PHASE THREE

C       Using the stiffness values, compute the diagonal matrix.
C       Note that this is a serialized NDP computation.
          do j=1,NNODES
            diag (j) = diag (j) +
     +                          sum (stiff(:,i), mask=(en1(:) .eq. j))
            diag (j) = diag (j) +
     +                          sum (stiff(:,i), mask=(en2(:) .eq. j))
          enddo

C       END PHASE THREE
        enddo
C       end loop A

C       BEGIN PHASE FOUR
        error_thresh = 0.001

C       determine which nodes are not on the boundary and count them
        non_bdy_mask = nodei .EQ. 0
        non_bdy_nodes = count (non_bdy_mask)

C       END PHASE FOUR
C       BEGIN PHASE FIVE

C       set the initial phi value (the fluid density) for each node by
C       inspecting the boundary codes
        phi = 0.0
        where (nodei .eq. 1)
           phi = 1.0
        end where
        where (nodei .eq. 2)
           phi = -1.0
        end where
        where (nodei .eq. 3)
           phi = 0.02*nodey
        end where

C       END PHASE FIVE
```

```
C       BEGIN PHASE SIX

        error = error_thresh

C       the iterative body of the Laplace
        do while (error >= error_thresh)
          phinew = 0.0

C         compute the phi change contributed by each element
          tmp1 = CSHIFT (elmtc,2,1)
          tmp2 = CSHIFT (elmtc,2,2)
          forall (i=1:NELEMNTS) n1(i,:) = phi(tmp1(i,:))
          forall (i=1:NELEMNTS) n2(i,:) = phi(tmp2(i,:))
          forall (i=1:3) s1(:,i) = n1 (:,i) * stiff (:,i)
          forall (i=1:3) s2(:,i) = n2 (:,i) * stiff (:,i)

C         sum and apply the per-element changes to produce a
C         vector of overall phi changes
          do j=1,3
            do i=1,NNODES
              phinew(i) = phinew(i) +
     +                        sum (s1(:,j), mask=tmp2(:,j) .eq. i)
              phinew(i) = phinew(i) +
     +                        sum (s2(:,j), mask=tmp1(:,j) .eq. i)
            end do
          end do

C         use these changes to update the phi value of each node
          delta_phi = 0.0
          where (non_bdy_mask)
            delta_phi = phinew/diag - phi
          end where
          phi = phi + delta_phi

C         compute the error value by summing the absolute change for
C         all nodes
          error = SUM (abs(delta_phi))/non_bdy_nodes

        end do

C       END PHASE SIX
```

B.2.3 forces: A Molecular Chemistry Kernel

Adl Source

```
% Input data: a vector of five-tuples, each representing an atom.
%             the elements of the tuples are as follows:
%              *   tuple element 1: index number of atom (0,..,#atoms-1)
%              *   tuple element 2: charge of atom
%              *   tuple element 3: x position of atom
%              *   tuple element 4: y position
%              *   tuple element 5: list of elements to which this
%                                   element is bonded

% for two particles (with charge q1 and q2), positioned at (x1,y1) and
% (x2,y2), compute the force of attraction or repulsion by Coulomb's law
force (q1,x1,y1,q2,x2,y2) :=
 let
  dx := x2-x1;
  dy := y2-y1;
  k := (q1*q2) / (dx^2 + dy^2)
 in
  (k*dx,k*dy)
 endlet;

% entry point
main irreg: vof (int,real,real,real,vof int) :=
  let
    % function to compute the force upon one atom 'me'
    f me :=
      let
        % functions to select a field from an atom tuple
        whoami (i,q,x,y,v) := i;
        charge (i,q,x,y,v) := q;
        xpos   (i,q,x,y,v) := x;
        ypos   (i,q,x,y,v) := y;
        edges  (i,q,x,y,v) := v;

        % compute the force due to each bond of the atom 'me'
        % return value is a pair (force in x-axis, force in y-axis)
        g z:int :=
         let
           other := irreg!z
```

```
          in
            force (charge(me),xpos(me),ypos(me),
                   charge(other),xpos(other),ypos(other))
          endlet;

         % create a vector of all bonded forces applied to the
         % atom 'me'
         t := map (g,edges(me));

         % define a function which adds force-pairs
         h (x,y) :=
          let
           first (i,j) := i;
           second (i,j) := j
          in
           (first (x) + first (y), second (x) + second (y))
          endlet
      in
        % combine all bonded forces for atom 'me' by accumulating
        % them with the function h
        reduce (h,(0.0,0.0),t)
      endlet
   in
     map (f,irreg)  % compute forces for each atom in //
  endlet
?
```

NESL Source

```
% Input data: a vector of five-field-records, each representing an atom.  %
%             the elements of the tuples are as follows:                  %
%              *   record element 1: index number of atom (0,..,#atoms-1) %
%              *   record element 2: charge of atom                       %
%              *   record element 3: x position of atom                   %
%              *   record element 4: y position                           %
%              *   record element 5: list of elements to which this       %
%                                    element is bonded                    %

datatype atom_r (int,float,float,float,[int]);

% for two particles (with charge q1 and q2), positioned at (x1,y1) and   %
% (x2,y2), compute the force of attraction or repulsion by Coulomb's law %
function force (q1,x1,y1,q2,x2,y2) =
let
  dx = x2-x1;
  dy = y2-y1;
  k = (q1*q2)/(dx*dx + dy*dy)
in
  (k*dx,k*dy);

% functions to select a field from an atom tuple %

function whoami (at) =
  let atom_r(i,q,x,y,v) = at
  in i;
function charge (at) =
  let atom_r(i,q,x,y,v) = at
  in q;
function xpos (at) =
  let atom_r(i,q,x,y,v) = at
  in x;
function whoami (at) =
  let atom_r(i,q,x,y,v) = at
  in i;
function ypos (at) =
  let atom_r(i,q,x,y,v) = at
  in y;
function edges (at) =
  let atom_r(i,q,x,y,v) = at
  in v;

% compute the force due to each bond of the atom 'me'       %
% return value is a pair (force in x-axis, force in y-axis) %
function main_f_g (z,me,irreg) =
  let
    other = irreg[z];
  in
    force (charge(me),xpos(me),ypos(me),
           charge(other),xpos(other),ypos(other));

% function to compute the force upon one atom 'me' %
function main_f (me,irreg) =
  let
    tee = {main_f_g(z,me,irreg): z in edges(me)};
    % convert the vector of pairs into a pair of vectors
    (tx,ty) = unzip (tee);
  in
    % combine all bonded-forces for the atom using sum
    % on the x-dimension force vector tx and the y-dimension
    % force vector ty
    (sum(tx),sum(ty));

% compute forces for each atom in // %
function main (irreg) = {main_f(z,irreg): z in irreg};
```

CM Fortran Source

```
C       This program computes the bonded forces on atoms in a molecule.
C       Each atom has propeties of x-position, y-position, charge. It
C       can be connected to any number of other atoms. We represent
C       the molecule as follows:
C          the vector charges holds the charge for each atom
C          the vector xpos holds the x-coordinate of the atom
C          the vector ypos holds the y-coordinate of the atom
C          the vector bonds is a logically segmented vector which
C             holds the identies of atoms which are bonded to.
C             The segmentation is defined in the vector starts
C             which lists, for each atom, which index of the
C             vector bonds holds its first bond. Subsequent
C             bonds for the same vector are stored in adjacent
C             positions

        program forces
        integer, parameter :: NATOMS=16, TOTALCONS=54
        real charges(NATOMS), xpos(NATOMS), ypos(NATOMS)
        integer starts(NATOMS+1), bonds(TOTALCONS)
        integer i,j

C       temporary vectors used during computation
        real xforper (TOTALCONS), yforper (TOTALCONS)
        real dx (TOTALCONS), dy (TOTALCONS)
        real xfortot (NATOMS), yfortot (NATOMS)
```

```
C       ensure that all vectors are laid out across the CM nodes
CMF$    layout charges (:news), xpos (:news), ypos (:news)
CMF$    layout starts (:news), bonds (:news), dx (:news), dy (:news)
CMF$    layout xforper (:news), yforper (:news)

C       define a simple molecule
        charges = 1.0
        xpos = [0.0,0.8,-0.4,1.0,-1.0,2.0,0.0,-0.1,0.2,-0.1,0.2,
     +          3.3,-0.1,-1.1,0.0,1.0]
        ypos = [0.0,1.2,1.2,1.0,-0.2,0.0,-1.0,-0.1,0.6,-11.0,0.0,
     +         -0.2,-0.1,1.1,2.1,1.1]
        starts = [1,5,9,15,17,20,24,25,28,33,40,43,46,48,51,52,55]
        bonds = [2,5,7,10,1,3,9,10,2,4,6,8,9,10,3,5,1,4,6,3,5,9,10,1,
     +           3,9,10,2,3,6,8,10,1,2,3,6,8,9,14,13,15,16,13,15,16,
     +           11,12,10,12,16,11,11,12,14]

C       serially iterate across all atoms in the molecule
        do i=1,NATOMS

C         for each bond of the current atom, calculate the
C         distance to the atom at the other end
          forall (j=starts(i):starts(i+1)-1)
     +      dx (j) = xpos(bonds(j))-xpos(i)
          forall (j=starts(i):starts(i+1)-1)
     +      dy (j) = ypos(bonds(j))-ypos(i)

C         compute the force upon this atom due to each bond
          forall (j=starts(i):starts(i+1)-1)
     +      xforper (j) = ((charges (bonds(j))*charges(i))/
     +                    (dx(j)**2 + dy(j)**2)) * dx(j)
          forall (j=starts(i):starts(i+1)-1)
     +      yforper (j) = ((charges (bonds(j))*charges(i))/
     +                    (dx(j)**2 + dy(j)**2)) * dy(j)

C         sum all the per-bond forces together to produce a resultant force
          xfortot (i) = sum (xforper, mask= (i .ge. starts(i)) .and.
     +                                      (i .lt. starts(i+1)))
          yfortot (i) = sum (yforper, mask= (i .ge. starts(i)) .and.
     +                                      (i .lt. starts(i+1)))
        end do
        end
```

Bibliography

[1] David Abramson and Gregory Egan. Design of a high performance data-flow multiprocessor. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 121–142. Prentice-Hall, Inc., 1991.

[2] David Abramson and Gregory K. Egan. The RMIT dataflow computer: a hybrid architecture. *The Computer Journal*, June 1990.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986. The "Dragon Book".

[4] Brad Alexander. Adl review. Technical Report 94-10, University of Adelaide, May 1993.

[5] Brad Alexander. *Mapping a Functional Language to a Data-Parallel Model of Computation.* PhD thesis, University of Adelaide, (in preparation).

[6] Brad Alexander, Dean Engelhardt, and Andrew Wendelborn. An overview of the Adl project. In *Proceedings of the Conference on High Performance Functional Computing*, April 1995.

[7] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, 1989.

[8] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *1990 International Conference on Supercomputing*, June 1990.

[9] Boon S. Ang, Derek Chiou, Larry Rudolph, and Arvind. Message passing support on StarT-Voyager. Technical Report CSG Memo 387, MIT Laboratory for Computer Science, July 1996.

[10] Murali Annavaram. Blocking versus non-blocking: Issues and tradeoffs in multithreaded code execution. Master's thesis, Colorado State University, 1996.

[11] Arvind and Robert A. Iannucci. Two fundamental issues in multiprocessing. Technical Report CSG 226-5, MIT, July 1986.

[12] Arvind and R.S Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In *PARLE: Parallel Architectures and Languages Europe (vol II)*, June 1987.

[13] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[14] Timothy M. Axelrod. Effects of synchronous barriers on multiprocessor performance. *Parallel Computing*, 3:129–140, 1986.

[15] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Stotnick, and Richard A. Stoakes. The ILLIAC IV computer. *IEEE Transactions on Computers*, 17(8), 1968.

[16] Tom Blank. The MasPar MP-1 architecture. In *COMPCON Proceedings*, pages 20–24, 1990.

[17] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT press, Cambridge Massachusetts, 1990.

[18] Guy E. Blelloch. NESL: A nested data-parallel language (version 3.0). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, 1995.

[19] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California*, pages 102–111, 19–22 May 1993.

[20] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstien, Marco Zahga, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel

language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[21] Guy E. Blelloch, Jonathan C. Hardwick, and Marco Zagha. NESL user's manual (for NESL version 3.0). Technical Report CMU-CS-95-169, School of Computer Science, Carnegie Mellon University, 1995.

[22] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[23] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), Santa Barbara*, pages 207–216, July 1995.

[24] Wim Böhm, David C. Cann, Rod R. Oldehoeft, and John T. Feo. Sisal 2.0 reference manual. Technical Report CS-91-118, Computer Science Department, Colorado State University, November 1991.

[25] Denis Caromel. Service, asynchrony, and wait-by-necessity. *Journal of Object-Oriented Programming*, pages 12–22, November/December 1989.

[26] Siddhartha Chatterjee. Programming models, compilers, and algorithms for irregular data-parallel computations. *International Journal of High Speed Computing*, 6(2):183–222, 1994.

[27] Derek Chiou, Boon S. Ang, Robert Grenier, Arvind, James C. Hoe, Michael J. Beckerle, James E. Hicks, and Andy Boughton. StarT-NG: Delivering seamless parallel computing. In *Proceedings of Euro-Par95*, 1995. Also available as MIT Laboratory for Computer Science CSG Memo 371.

[28] Nikos Chrisochoides, Elias Houstis, and John Rice. Mapping algorithms and software environment for data parallel PDE iterative solvers. *Journal of Parallel and Distributed Computing*, 21:75–95, 1994.

[29] N.P. Chrisochoides, C.E. Houstis, E.N. Houstis, P.N. Papachiou, S.K. Kortesis, and J.R. Rice. DOMAIN DECOMPOSER: A software tool for

mapping PDE computations to parallel architectures. In R. Glowinski, Y.A. Kuznetsov, G. Meurant, J Periaux, and O. Widlund, editors, *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 341–357, Philadelphia, 1991. SIAM.

[30] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, 1993.

[31] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. *Sigplan Notices*, 26(4):164–175, April 1991.

[32] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. Technical report, University of California, Berkeley, 1993.

[33] Dean Engelhardt. Evaluation of characteristics of three programming paradigms supported by existing CM-5 software. Technical report, Department of Computer Science, University of Adelaide, February 1993.

[34] Dean Engelhardt and Andrew Wendelborn. Progress in implementing Adl on the CM-5 supercomputer. Technical report, University of Adelaide, August 1994.

[35] Dean Engelhardt and Andrew Wendelborn. A partitioning-independent paradigm for nested data parallelism. *International Journal of Parallel Programming*, 24(4):291–318, August 1996. Also appears in proceedings of PACT95.

[36] Dean Engelhardt and Andrew Wendelborn. Visualizing communications patterns in nested data-parallel computations. Technical Report 96-02, University of Adelaide, January 1996.

[37] Dean Engelhardt and Andrew Wendelborn. A multi-threaded implementation of nested data-parallelism. In *Proceedings of the 20th Australasian Computer Science Conference (ACSC '97), Sydney*, pages 346–355, February 1997.

[38] Dean Engelhardt and Andrew Wendelborn. PFN: An interactive visual tool for specifying data-layout in distributed nested data-parallel computations. Technical Report (in preparation), Department of Computer Science, University of Adelaide, 1997.

[39] Paraskevas Evripidou and Jean-Luc Gaudiot. A decoupled data-driven architecture with vectors and macro actors. In H. Burkhart, editor, *Proceedings of the Joint International Conference on Vector and Parallel Processing (CONPAR90 — VAPP IV), Zurich*, number 457 in Lecture Notes in Computer Science, pages 39–50. Springer-Verlag, September 1990.

[40] Paraskevas Evripidou and Jean-Luc Gaudiot. The USC decoupled multilevel data-flow execution model. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 347–380. Prentice-Hall, Inc., 1991.

[41] John T. Feo, David C. Cann, and Rod R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, December 1990.

[42] Alice E. Fischer and Frances S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice Hall, 1993.

[43] Jean-Luc Gaudiot and Chung-Ta Cheng. A scalable cache design for I-structures in multithreaded architectures. In *Proceedings of the 1996 International Conference on Parallel Processing*, 1996.

[44] Seth C. Goldstein. The implementation of a threaded abstract machine. Technical Report UCB/CSD 94-818, University of California, Berkeley, May 1994.

[45] Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. In *Proceedings of the 26th Annual Symposium on the Foundations of Computer Science*, pages 241–249, October 1985.

[46] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[47] Manish Gupta and Prithviraj Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In *Proceedings of the ACM International Conference on Supercomputing*, July 1993.

[48] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1991.

[49] Michael T. Heath. Recent developments and case studies in performance visualization using ParaGraph. In G. Haring and G. Kotsis, editors, *Performance Measurement and Visualization of Parallel Systems*, pages 175–200. Elsevier Science Publishers, 1993.

[50] Michael T. Heath and J.A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–40, September 1991.

[51] Michael T. Heath and Jennifer E. Finger. *ParaGraph: A Tool for Visualizing Performance of Parallel Programs.* Oak Ridge National Laboratory, April 1995.

[52] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[53] W.Daniel Hillis and Guy L. Steele. *The Connection Machine Lisp Manual.* Thinking Machines Corp., Cambridge, MA.

[54] J. E. Hoch. Compile-time partitioning of a non-strict language into sequential threads. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 180–189, 1991.

[55] R.W. Hockney and C.R. Jesshope. *Parallel Computers 2.* Adam Hilger, 1988.

[56] Paul D. Hovland and Lionel M. Ni. A model for automatic data partitioning. Technical Report MSU-CPS-ACS-73, Michigan State University, October 1992.

[57] HPF Committee. High Performance Fortran terms and concepts. *Fortran Forum*, 12(4), December 1993.

[58] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[59] ICL. DAP:Fortran language reference manual. ICL Technical Publication 6918, ICL, 1979.

[60] K.E. Iverson. *A Programming Language.* Wiley, 1962.

[61] Stephen Jenks and Jean-Luc Gaudiot. Nomadic threads: A migrating multithreaded approach to remote memory access in multiprocessors. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT96)*, pages 2–11. IEEE Computer Society Press, October 1996.

[62] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, January 1996.

[63] Harry F. Jordan. A special purpose architecture for finite element analysis. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 263–266, 1978.

[64] L.V. Kale and W. Shu. The Chare kernel base language: Preliminary performance results. In *Proceedings of the International Conference on Parallel Processing*, pages 118–121, August 1989.

[65] Ken Kennedy and Ulrich Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive programming environment. Technical Report CRPC-TR91205, Rice University, April 1991.

[66] Ulrich Kremer. Automatic data layout for distributed-memory machines. Technical Report CRPC-TR93299-S, Rice University, February 1993.

[67] Boontee Kruatrachue and Ted Lewis. Grain size determination for parallel processing. *IEEE Software*, pages 23–32, January 1988.

[68] R.E. Ladner and M.J. Fisher. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[69] B. Lee and A.R. Hurson. Dataflow architectures and multithreading. *IEEE Computer*, pages 27 – 39, August 1994.

[70] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. In *Symposium on Parallel and Distributed Algorithms '92*, June 1992.

[71] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–900, October 1985.

[72] Jingke Li and Maria Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–375, July 1991.

[73] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, October 1990.

[74] Wen-Yen Lin and Jean-Luc Gaudiot. I-structure software cache — a split-phase transaction runtime cache system. In *Proceedings of the 1996 Parallel Architectures and Compilation Tecniques*, 1996.

[75] D. B. Loveman. High performance Fortran. *IEEE parallel and distributed technology: systems and applications*, 1(1):25–42, February 1993.

[76] B.D. Lubachevsky and A.G. Greenberg. Simple, efficient aysnchronous parallel prefix algorithms. In *Proceedings of the International Conference on Parallel Processing*, pages 66-69, August 1987.

[77] MasPar Computer Corporation, Sunnyvale, CA. *MP-1 Programming Manuals*, January 1990. Technical Reports PN 9305-0000 01/90 MPPE, PN 9302-0100 01/90 MPL, PN 9302-0000 01/90 MPL Ref.

[78] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL: Streams and Iteration in a Single Assignment Language: reference manual version 1.2.* Lawrence Livermore National Laboratory, March 1985.

[79] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 364–384. MIT Press, 1991.

[80] Message Passing Interface Forum. MPI: a message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[81] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford University Press, 1990.

[82] Michael L. Nelson. Concurrency and object-oriented programming. *ACM SIGPLAN Notices*, 26(10):63–73, October 1991.

[83] A. Nicolau and H. Wang. Optimal schedules for parallel prefix computation with bounded resources. In *Third ACM SIGPLAN Symposium on Principles and Pratics of Parallel Programming*, pages 1–10, April 1991.

[84] Rishiyur S. Nikhil. Id (version 90.1) reference manual. Technical Report CSG Memo 284-2, MIT Lab for Computer Science, July 1991.

[85] Rishiyur S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 390–405, August 1993.

[86] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, May 1989.

[87] Rishiyur S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 156–167, 1992.

[88] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. *Computer Architecture News*, 21(2):224–235, May 1993.

[89] Michael O'Boyle. A data partitioning algorithm for distributed memory compilation. Technical Report UMCS-93-7-1, Department of Computer Science, University of Manchester, 1993.

[90] R.R. Oldehoeft and S.J. Allen. Adaptive exact-fit storage management. *Communications of the ACM*, 28(5):506–511, May 1985.

[91] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1993.

[92] Daniel J. Palermo and Prithviraj Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[93] R. H. Perrott. *Parallel Programming*. Addison-Wesley, Menlo Park, 1987.

[94] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[95] Ravi Ponnusamy, Rajeev Thakur, Alok Choudhary, Kishore Velamakanni, Zeki Bozkus, and Geoffrey Fox. Experimental performance evaluation of the CM-5. *Journal of Parallel and Distributed Computing*, 19:192–202, 1993.

[96] Jan F. Prins and Daniel W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 119–128, May 1993.

[97] S.F. Reddaway. DAP — a distributed array processor. In *Proceedings of the First Annual Symposium on Computer Architecture (IEEE/ACM), Florida*, 1973.

[98] Paul Roe. Adl: The Adelaide language. Technical report, University of Adelaide, July 1992.

[99] Lucas Roh, Walid A. Najjar, Bhanu Shankar, and Wim Böhm. Generation, optimization and evaluation of multi-threaded code. *Journal of Parallel and Distributed Computing*, 32(2), February 1996.

[100] Lucas J. Roh. *Code Generations, Evaluations and Optimizations in Multithreaded Executions.* PhD thesis, Colorado State University, 1995.

[101] J. Rose. C*: a C++-like language for data-parallel computation. In *Proceedings of the 1986 USENIX Workshop on C++*, Santa Fe, December 1987.

[102] John R. Rose and Jr. Guy L. Steele. C*: an extended C language for data parallel programming. In *Second International Conference on Supercomputing, Proceedings Supercomputing '87, Industrial Supercomputer Applications and Computations*, volume II, pages 2–16. International Supercomputing Institute, Inc., 1987.

[103] Gary W. Sabot. *The Paralation Model: Architecture Independent Parallel Programming.* The MIT press, Cambridge Massachusetts, 1990.

[104] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Y. Kodama, and Toshitsugu Yuba. An architecture of a dataflow single chip processor. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 46–53, May 1989.

[105] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing, Cambridge, Massachusetts, 1989.

[106] Klaus Erik Schauser, David E. Culler, and Thorsten von Eicken. Compiler-controlled multithreading for lenient parallel languages. In I. Hughes, editor, *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 50–72, August 1991.

[107] Willi Schönuer. *Scientific Computing on Vector Computers*, volume 2 of *Special Topics in Supercomputing.* North-Holland, 1987.

[108] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, 1986.

[109] Bhanu Shankar. *The Spectrum of Thread Implementations on Hybrid Multithreaded Architectures.* PhD thesis, Colorado State University, 1995.

[110] John Sharley. Visualization of the performance of nested data-parallelism. Master's thesis, Department of Computer Science, University of Adelaide, November 1996.

[111] Wei Shu. *Chare Kernel and its Implementation on Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, January 1990.

[112] Wei Shu and L.V. Kale. A dynamic scheduling strategy for parallel computations. In *Proceedings of Supercomputing '89*, pages 389–398, November 1989.

[113] Wei Shu and L.V. Kale. Chare kernel — a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11(3):198–211, March 1991.

[114] Wei Shu and Min-You Wu. CAB: supporting irregular problems with runtime scheduling. Technical report, State University of New York at Buffalo, 1996.

[115] Jay Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[116] Stephen K. Skedzielewski. *Parallel Functional Languages and Compilers*, chapter 4, pages 105–157. Addison-Wesley, 1991.

[117] B.J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE (Real Time Signal Processing)*, (298):241–248, 1981.

[118] Justin R. Smith. *The Design and Analysis of Parallel Algorithms.* Oxford University Press, December 1992.

[119] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5. *Computer Architecture News*, 21(2):302–313, May 1993.

[120] K. Stevens. CFD — a FORTRAN-like language for the ILLIAC IV. *SIGPLAN Notices*, 10, 1975.

[121] Thinking Machines Corporation, Cambridge, Massachusetts. *Introduction to Data Level Parallelism*, April 1986.

[122] Thinking Machines Corporation, Cambridge, Massachusetts. *Connection Machine Model CM-2 Technical Summary*, 1990.

[123] Thinking Machines Corporation, Cambridge, Massachusetts. *Getting Started in *Lisp*, 1991.

[124] Thinking Machines Corporation, Cambridge, Massachusetts. *CM5 Technical Summary*, January 1992.

[125] Thinking Machines Corporation, Cambridge, Massachusetts. *CM-5 C* Performance Guide (version 7.1)*, August 1993.

[126] Thinking Machines Corporation, Cambridge, Massachusetts. *CMMD Reference Manual (version 3.0)*, May 1993.

[127] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Language Reference Manual (version 2.2)*, October 1994.

[128] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Programming Guide (version 2.2)*, October 1994.

[129] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran User's Guide*, October 1994.

[130] Thinking Machines Corporation, Cambridge, Massachusetts. *Getting Started in CM Fortran*, October 1994.

[131] Simon Thompson. *Miranda: The Craft of Functional Programming.* Addison-Wesley, 1995.

[132] Arthur Trew and Greg Wilson, editors. *Past, Present, Parallel.* Springer-Verlag, 1991.

[133] Lewis W. Tucker and George G. Robertson. Architecture and applications of the Connection Machine. *Computer*, 21(8):26–38, August 1988.

[134] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[135] Reinhard von Hanxleden. Compiler support for machine independent parallelization of irregular problems. Technical Report CRPC-TR92301-S, Rice University, November 1992.

[136] Brent B. Welch. *Practical Programming in Tcl and Tk.* Prentice Hall, 1995.

[137] S. Wholey and Guy L. Steele Jr. Connection Machine Lisp: A dialect of Common Lisp for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing*, 1987.

[138] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, 1987.

[139] N. Yoo. Generic multithreaded machine (GMT) simulator. Technical report, Department of Electrical Engineering, University of Southern California, 1993.

[140] Toshitsugu Yuba, Toshino Shimada, Yoshinori Yamaguchi, Kei Hiraki, and Shuichi Sakai. Dataflow computer development in Japan. In *Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, volume 18, pages 140–147, September 1990.

[141] M. E. Zosel. High performance Fortran: an overview. In IEEE, editor, *Digest of papers: Compcon spring '93, San Francisco*, pages 132–136. IEEE Computer Society Press, February 1993.