Permutation-Based Range-Join Algorithms on *N*-Dimensional Meshes

Shao Dong Chen, Hong Shen, and Rodney Topor, *Member*, *IEEE Computer Society*

Abstract—In this paper, we present four efficient parallel algorithms for computing a nonequijoin, called *range-join*, of two relations on N-dimensional mesh-connected computers. Range-joins of relations R and S are an important generalization of conventional equijoins and band-joins and are solved by permutation-based approaches in all proposed algorithms. In general, after sorting all subsets of both relations, the proposed algorithms permute every sorted subset of relation S to each processor in turn, where it is joined with the local subset of relation R. To permute the subsets of S efficiently, we propose two data permutation approaches, namely, the *shifting* approach which permutes the data recursively from lower dimensions to higher dimensions and the *Hamiltonian-cycle* approach which first constructs a Hamiltonian cycle on the mesh and then permutes the data along this cycle by repeatedly transferring data from each processor to its successor. We apply the shifting approach to meshes with different storage capacities which results in two different join algorithms. The *Basic Shifting Join* (BASHJ) algorithm can minimize the number of subsets stored temporarily at a processor, but requires a large number of data transmissions, while the *Buffering Shifting Join* (BUSHJ) algorithm can achieve a high parallelism and minimize the number of data transmissions, but requires a large number of subsets stored at each processor. For constructing a Hamiltonian cycle on a mesh, we propose two different methods which also result in two different join algorithms. The *Recursive Hamiltonian-Cycle Join* (REHCJ) algorithm uses a single processor to construct a Hamiltonian cycle recursively, while the *Parallel Hamiltonian-Cycle Join* (PAHCJ) algorithm uses all processors to construct a Hamiltonian cycle in parallel. We analyze and compare these algorithms. The results shows that both Hamiltonian cycle algorithms require less storage and local join operations than the shifting algorithms, but more data movement steps.

Index Terms—Analysis of algorithms, data permutation, N-dimensional meshes, relational databases, parallel processing, performance, range-join operations.

1 Introduction

With the increases in database size and query complexity, highly parallel database systems supported by general-purpose parallel architectures have become the trend of future database systems [6]. As an important and time-consuming operation in relational database systems, *join* has attracted a significant amount of research effort for designing efficient parallel algorithms [3], [10], [12]. In this paper, we address the problem of range-join that generalizes the conventional equijoin and band-join operations [7] and develop efficient parallel range-join algorithms on N-dimensional mesh-connected computers.

For two given constants e_1 and e_2 with $0 \leq e_1 \leq e_2$, we define the *range-join* of two relations R (*inner relation*) and S (*outer relation*) on attribute A from R and B from S, denoted by $R \bowtie_{e_1}^{e_2} S$, to be the relation T obtained by concatenating all tuples r in R and s in S such that $e_1 \leq |r.A - s.B| \leq e_2$ [13]. Range-join is an important operation in relational database systems and appears frequently in practice, especially in the queries requiring joins over continuous real-world domains such as time and distance. For example, a query of "finding all pairs of customers of two stores whose account balance differs from 100 to 1,000 dollars" for consumption pattern analysis across different stores requires a range-join. In an online stock trading system, all buy and sell orders are stored in two separate tables. A query of "finding all pairs of buy and sell orders for the same share whose prices differ between one to three dollars" is frequently needed in an online stock trading system. Clearly this query also requires performing a range-join. Moreover, as a generalization of band-join operations, the range-join algorithms can be directly used to compute band-joins as well as equijoins.

It has been shown that the hash-based join algorithms are superior to other algorithms for equi-join operations [12]. However, as the join condition of the range-joins involves range comparisons rather than equalities, hash-based join algorithms are unsuitable for range-join operations [7] because the conventional hash functions (e.g., modulo-division, folding, radix-transformation, and midsquare methods) will inherently destroy the ordering property of tuples. In contrast, permutation-based join algorithms, which are an efficient implementation of parallel nested-loops join algorithms, have been shown to be effective for computing range-joins on hypercube computers [13] and torus computers [4]. Moreover, unlike most hash-based join algorithms which are vulnerable to data skew and will result in an unacceptable performance for extremely

- *S.D. Chen is with Hutchison Telecommunications Limited, Hunghom, Kowloon, Hong Kong. E-mail: stephen.chen@post.com.*
- *H. Shen is with the Graduate School of Information Science, Japan Advanced Institute of Science and Technology, Tatsunokuchi, Ishikawa 923-1292, Japan. E-mail: shen@jaist.ac.jp.*
- *R. Topor is with the School of Computing and Information Technology, Griffith University, Nathan, Queensland 4111, Australia. E-mail: rwt@cit.gu.edu.au.*

Manuscript received 14 Sept. 2000; revised 1 June 2001; accepted 14 Sept. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 112868.

1045-9219/02/$17.00 © 2002 IEEE

skewed data, the permutation-based join algorithms are immune to any data skew.

In general, with the assumption that each relation is distributed evenly across all processors in the mesh initially, permutation-based algorithms sort the two local subsets of both relations in each processor, then permute every subset of S to every processor in turn, where it is joined with the local subset of R at that processor. The local range-join operation in each processor for two sorted subsets is implemented by a sequential sort-merge algorithm presented in [5].

While efficient algorithms for range-join have been developed on hypercube, toruses, and 2-dimensional mesh [15], [13], [4], [5], they are unknown on N-dimensional meshes due to the topological sophistication and architectural challenges of N-dimensional meshes. More complex multi-relation range-join based operations such as mutual range-join and chain range-join have also been implemented efficiently on hypercube [16], [14]. Because of the architectural incompatibility between N-dimensional meshes of dimension (side) size greater than 2 and hypercube (binary cube), 2-dimensional mesh, or N-dimensional toruses, all previous results for range-join on hypercube, 2-dimensional mesh, and toruses are neither applicable nor extendible directly to N-dimensional meshes.

In this paper, we develop a set of efficient parallel algorithms for range-join on N-dimensional meshes using permutation-based approaches. We present two new approaches for efficiently permuting all subsets of S in N-dimensional meshes which are the key techniques to be employed in our range-join algorithms. Our first approach, namely *shifting*, permutes the data recursively from lower dimensions to higher dimensions; while the second approach, namely *Hamiltonian-cycle*, first constructs a Hamiltonian cycle on the mesh and then permutes the data along this Hamiltonian cycle by repeatedly transferring data from each processor to its successor.

The shifting approach can be applied to meshes with different storage capacities which results in two different data permutation join algorithms. The *Basic Shifting Join* (BASHJ) algorithm can minimize the number of *buffered* subsets which are needed to be stored temporarily at a processor during the permutation, but it requires a large number of data transmissions, local join operations, and disk I/O operations due to the low parallelism. Conversely, the *Buffering Shifting Join* (BUSHJ) algorithm can achieve a high parallelism and minimize the number of data transmissions, but it needs to store a large number of buffered subsets in each processor.

By using two different methods to construct a Hamiltonian cycle on a mesh, the Hamiltonian-cycle approach also results in two different join algorithms. The *Recursive Hamiltonian-Cycle Join* (REHCJ) algorithm uses a single processor to construct a Hamiltonian cycle in a recursive fashion, and then broadcasts the resulting Hamiltonian cycle to all other processors so that they know their successors in the cycle; while the *Parallel Hamiltonian-Cycle Join* (PAHCJ) algorithm uses all processors to construct a Hamiltonian cycle in parallel so that all processors can know their successors simultaneously without the broadcast required in the REHCJ.

We present an analytical model and use this model to analyze these four algorithms. The result shows that both data permutation approaches outperform each other for different configurations, mainly depending on the costs of local join operations and data transmissions. In general, the Hamiltonian-cycle algorithms require fewer local join operations and less storage than the shifting algorithms, but need more data transmissions than BUSHJ. For the shifting approach, BUSHJ always outperforms BASHJ except that it has higher storage requirements. For the Hamiltonian-cycle approach, the PAHCJ is clearly more efficient than REHCJ due to the parallel construction of Hamiltonian cycles. We include REHCJ in this paper because the PAHCJ is based on the REHCJ, and understanding REHCJ can help to understand the PAHCJ which is more complicated.

The remainder of this paper is organized as follows: In Section 2, we study N-D meshes and discuss their properties, and present the permutation-based join algorithms in general. In Sections 4, 5, 6, and 7, we present and analyze four mesh join algorithms—BASHJ, BUSHJ, REHCJ, and PAHCJ, respectively. Finally, we conclude the paper by comparing these algorithms in Section 9.

2 *N*-Dimensional Meshes

Meshes are an important class of parallel interconnection networks which have been well studied in the literature [11]. In parallel database design, mesh-connected parallel computers are characterized by the shared-nothing architecture [17]. There are several commercially available mesh-connected computers, such as the recent Intel Paragon XP/S [8] whose processors are connected by meshes instead of hypercubes which were used in the earlier Intel iPSC/860.

A large number of parallel algorithms have been designed for meshes, including sorting, routing, and searching. However, very little research has been done on the design of join algorithms on meshes. Simple nested-loops and sort-merge algorithms on meshes are briefly mentioned in [9] when a high-level comparison between meshes and hypercubes is presented, but the author of [9] does not present these two algorithms in detail and does not demonstrate how to implement them. In this paper, we first study the N-dimensional meshes in detail, then present and analyze four new data permutation join algorithms for them.

An N-dimensional mesh (or "N-D mesh" for simplicity) is an interconnection network which connects $p = D_1 \times \cdots \times D_N$ processors. D_j $(j = 1, \ldots, N)$ is called the *degree* in dimension j, and it may be different for different dimensions. Each processor in the computer has its own memory and disk, and does not share any memory or disk with others. The index of a processor corresponds to an N-vector $(i_1, \ldots, i_N)$, where $i_j = 1, \ldots, D_j$ and $j = 1, \ldots, N$. Two processors are linked by a bidirectional communication link if their indices differ by one in precisely one coordinate. A $4 \times 4 \times 4$ mesh is shown in Fig. 1, where the axes are shown in the right-top corner. Note that, unlike toruses, there are *no* wraparound links at the boundary processors

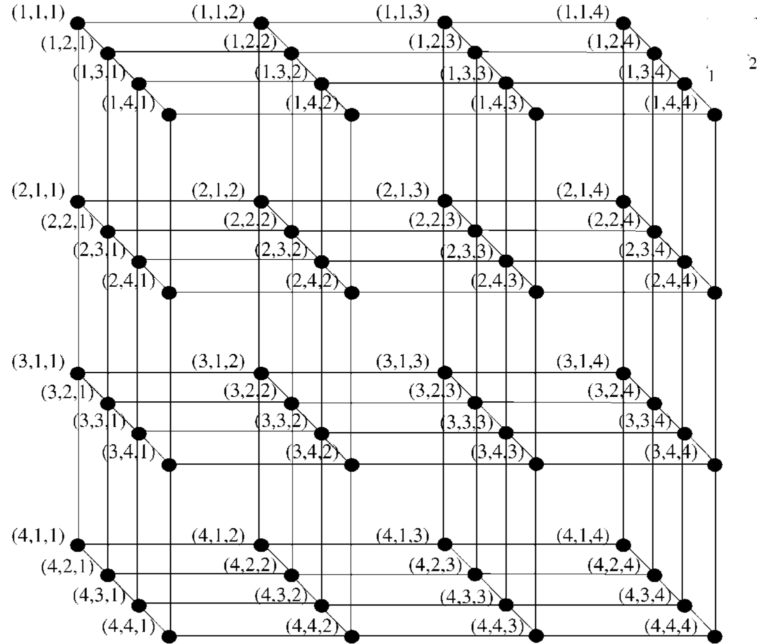


Fig. 1. A $4 \times 4 \times 4$ mesh.

in meshes. This constraint makes it more difficult and challenging to design algorithms on meshes than on toruses.

In this paper, we assume that the mesh-connected computers operate the SPMD (Single Program, Multiple Data) model [1] as many current supercomputers do, in which all processors at each step execute different instructions of the same program (which contains many instructions) at the same time, each on a different datum. It is also possible that only a subset of processors execute a program while the others are idle. Each processor can transfer data from its memory to one of its neighbor's memory along the communication link between them. Parallel data transmissions can only be carried out in the same dimension and in the same direction. Other data transmissions in different dimensions or in different directions are delayed until the current transmissions are completed and the system (or the algorithms) schedules them to start.

An N-D mesh has a simple recursive structure: It can be constructed from D_N different $(N-1)$-D submeshes by simply connecting each processor in the *j*th $(N-1)$-D submesh to the corresponding processor in the $(j+1)\text{th}$ submesh with an edge in dimension N, such that their indices differ by one in the *N*th dimension, where a *j*th $(N-1)$-D submesh is a $(N-1)$-D mesh in which all processors' *N*th coordinate is *j*, and $1 \leq j < D_N$.

We denote an N-D mesh by M and one of its k-D submeshes by M_k which has $(D_1 \times \cdots \times D_k)$ processors, where $0 \leq k \leq N$. For any M_k, there is a fixed list $L = [i_{k+1}, \ldots, i_N]$ which determines the indices of M_k in the higher dimensions $k+1$ to N, where $1 \leq i_j \leq D_j$ for $k+1 \leq j \leq N$. We call L the *determinant* of M_k. The determinant of M_k has $N-k$ elements. When L is empty, M_N is M, and when L has N elements, M_0 contains only one processor. For a k-D submesh M_k, we further denote its *j*th $(k-1)$-D-D submesh by M_{k-1}^j, where $1 \leq j \leq D_k$.

Example 1. The 3-D mesh in Fig. 1 can be denoted by M_3 or simply by M. It has $4 \times 4 \times 4 = 64$ processors and four 2-D submeshes M_2^1, M_2^2, M_2^3, and M_2^4, each being a grid. The determinant of M is empty, denoted by [], while the determinant of each M_2^i is $[i]$ $(i = 1, \ldots, 4)$.

Each 2-D submesh M_2^i contains four 1-D submeshes M_1^1, M_1^2, M_1^3, and M_1^4, each being a linear array. The determinant of its *j*th 1-D submesh is $j : [i] = [j, i]$ $(j = 1, \ldots, 4)$, where operation ":" means to prefix an element l into list L. For example, the determinant of the second 1-D submesh of the 2-D submesh M_2^3 is $[2, 3]$.

Similarly, each 1-D submesh M_1^j also contains four 0-D submeshes M_0^1, M_0^2, M_0^3, and M_0^4, each being a single processor. The determinant of its *t*th 0-D submesh is $t : [j, i] = [t, j, i]$ $(t = 1, \ldots, 4)$. At this stage, the determinant of a 0-D submesh is the same as the index of its single processor.

3 Analytical Model and Permutation-Based Join Algorithms

In this section, after introducing the analytical model, we present the the permutation-based join algorithms in general.

We mainly adopt the analytical model used in [3] and, hence, assume that both relations are initially distributed evenly across all processors in the computer whose total available memory is larger than the size of inner relation R. Data are accessed and transferred in blocks. To simplify the analysis, we do not consider the join-product skew [19] in the data, and assume that the processing time of a join operation depends only on the number of tuples processed.

When analyzing the algorithms, we consider three major costs associated with the join operations: I/O, communication, and computation costs. The I/O cost is required to read/write data from/to the disks, while the

communication cost is required to transfer data between different processors across an interconnection network. The computation cost is required for the operations which are performed in the main memory. There are many different in-memory operations and it is difficult, if not impossible, to consider all of them. Thus, we focus on only three main in-memory operations: comparison, hashing, and probing operations. We do not consider other in-memory operations in the algorithm analysis, such as moving a tuple whose cost is insignificant and negligible, and concatenating two tuples whose cost has been included in the output cost since the number of concatenation operations is proportional to the number of resulting tuples generated. The notations used to describe and analyze our algorithms are listed as follows:

- $|R|, |S|$: number of tuples in relations R and S;
- B_R, B_S: number of blocks of relations R and S ($B_R \leq B_S$);
- $R_{i_1,\ldots,i_N}, S_{i_1,\ldots,i_N}$: the subsets of R and S in processor $P_{i_1,\ldots,i_N}$;
- JS: join selective factor, defined by $\frac{|R \bowtie S|}{|R|\cdot|S|}$;
- p: number of processors;
- M: number of blocks of available memory in a processor, ($B_R \leq M \cdot p$);
- T_{io}: time for reading/writing one block of data from/to the disk;
- T_t: time for transferring a block of data between two neighboring processors;
- T_c: time for comparing two values in memory;

The permutation-based join algorithms consists of the following two phases:

1. **Sorting Local Subsets:** Every processor simultaneously reads its initial subset of relation S, sorts it on the join attribute sequentially, and then applies the same process to relation R.
2. **Permute and Join:** Every processor simultaneously computes the local range-join for its two local subsets of R and S, and then repeatedly reads the current subset of S from a neighbor and performs a local range-join operation on this arriving subset, until all subsets of S have visited each processor exactly once.

Clearly, the permutation-based algorithms compute the whole join by computing totally p^2 subjoins independently, that is,

$$R \bowtie S = \bigcup_{i=0}^{p-1} \bigcup_{j=0}^{p-1} (R_i \bowtie S_j).$$

The purpose of sorting subsets in the first phase is to make the local range-join operations more efficient. When two operand subsets are stored, we can perform these local range-join operations by using our sequential algorithm which has been shown to be more efficient than other possible algorithms for computing range-joins [5]. Thus, locally sorting the initial subsets in each processor can benefit all p^2 subsequent subjoin operations and, hence, the redundant CPU processing required in the previous nested-loops algorithm can be reduced significantly.

The first phase could be implemented by the following statements:

for all processors $P_{i_1,\ldots,i_N}$ **do in parallel**
  Read $S_{i_1,\ldots,i_N}$ from disk to memory;
  Sort $S_{i_1,\ldots,i_N}$ using a sequential external sorting algorithm;
  Write sorted $S_{i_1,\ldots,i_N}$ back to disk;
  Read $R_{i_1,\ldots,i_N}$ from disk to memory;
  Sort $R_{i_1,\ldots,i_N}$ using a sequential internal sorting algorithm;
  Read sorted $S_{i_1,\ldots,i_N}$ from disk to memory

The total cost $T_{ini}(R, S, p)$ of phase 1 is

$$T_{ini}(R, S, p) = T_{io} \times \frac{B_R + B_S + 2B_S \log_M B_S/p}{p} + T_c \times \frac{B_R \log B_R/p + B_S \log B_S/p}{p}. \quad (1)$$

For simplicity, we will use "subset(s)" to mean "the subset(s) of relation S" hereafter when no confusion could occur.

In the second phase, the local range-join operation on one sorted subset of R and one sorted subset of S is realized by a sequential sort-merge range-join algorithm [2], which is based on the standard sort-merge join algorithm [18] for equi-join, with additional backup to inspect previously considered tuples: For each tuple s, it first joins every tuple r such that $r.A + e_1 \leq s.B \leq r.A + e_2$, and then joins every tuple r such that $r.A - e_2 \leq s.B \leq r.A - e_1$. The resulting tuples are stored in the local disk of each processor as they are produced, one block at a time. The running time of this algorithm is denoted by $T_{lj}(R/p, S/p)$. If another sequential local range-join algorithm is used, $T_{lj}(R/p, S/p)$ is simply replaced by that algorithm's running time.

Thus, the remaining problem in the second phase is how to efficiently permute the subsets of S to all processors. Despite the simplicity of the problem, the task of exploring efficient data permutation approaches for an N-D mesh is not an easy task. We devote the following four sections to complete this task, each presenting a parallel algorithm.

4 Basic Shifting Join

4.1 Description

We start with a simple algorithm for permuting (and joining) the subsets of S on an 1-D mesh—a linear array with D_1 processors. This algorithm works in the way similar to pulsing water through a pipe between its two ends in turn, as suggested in Fig. 2. Thus, it consists of two steps, each with $D_1 - 1$ iterations:

Forward Shift. Each processor P_j ($j = 2, \ldots, D_1$) repeatedly reads a subset from its left neighbor P_{j-1}, and performs a join on this newly arrived subset.

Backward Shift. Each processor P_j ($j = 1, \ldots, D_1 - 1$) repeatedly reads a subset from its right neighbor P_{j+1}, and performs a join on this newly arrived subset.

Since every processor replaces its current subset with the subset read from its neighbor, its current subset is S_1 after forward-shift. Thus, we must be able to restore their original subsets to perform backward-shift. To do so, every processor makes a temporary copy of its original subset

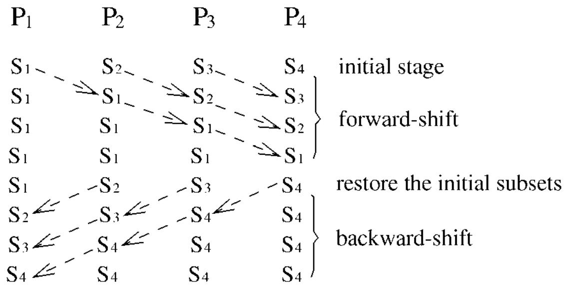


Fig. 2. Permuting data in a linear array.

before forward-shift, and restores the original subset back from this copy after forward-shift. This temporary subset is called a *buffered* subset. The correctness of the algorithm is obvious since every subset is visited and joined at each processor exactly once.

This shifting algorithm for linear arrays can be generalized for higher-dimensional meshes based on their recursive structure, and works in a recursive fashion: When permuting the subsets on a k-D submesh M_k with determinant L, if $k = 0$, the single processor in M_k performs a local join operation on its current subset; if $k > 0$, the processors in M_k execute the following six steps:

1. All processors permute the subsets simultaneously on all $(k-1)$-D submeshes, each in dimensions from 0 to $k-1$ recursively.
2. Each processor $P_{i_1,\dots,i_N}$ copies its current subset $S_{i_1,\dots,i_N}$ to buffered subset $S^k_{i_1,\dots,i_N}$ temporarily, where $S^k_{i_1,\dots,i_N}$ is stored in the local memory first until the local memory is exhausted, then is stored in the local disk.
3. All processors perform a forward-shift in dimension k with $D_k - 1$ iterations, where the local join operation in the algorithm for linear arrays is replaced by a recursive permutation in the lower dimensions from 0 to $k-1$.
4. Each processor $P_{i_1,\dots,i_N}$ restores $S^k_{i_1,\dots,i_N}$ to be $S_{i_1,\dots,i_N}$.
5. All processors perform a backward-shift in dimension k with $D_k - 1$ iterations, where the local join operation in the algorithm for linear arrays is replaced by a recursive permutation in the lower dimensions from 0 to $k-1$.
6. Each processor $P_{i_1,\dots,i_N}$ restores $S^k_{i_1,\dots,i_N}$ to be $S_{i_1,\dots,i_N}$.

Thus, a subset $S_{i_1,\dots,i_N}$ is backed up once in Step 2 and restored twice in Steps 4 and 6, one for performing backward-shift, and the other for permuting in the higher dimension $k+1$. During the ith iteration of forward-shift, every processor in each M^j_{k-1} $(j = i+1, \dots, D_k)$ reads the subset of its neighbor in M^{j-1}_{k-1} along the edge in dimension k to replace its own one, and then permutes the (new) subset on M^j_{k-1} in dimensions from 0 to $k-1$ recursively. Similarly, during the ith iteration of backward-shift, every processor in M^j_{k-1} $(j = 1, D_k - i)$ reads the subset of its neighbor in M^{j+1}_{k-1} along the edge in dimension k, and then permutes the (new) subset on M^j_{k-1} in dimensions from 0 to $k-1$ recursively. Thus, the basic shift join (BASHJ) algorithm for N-D meshes can be summarized as follows, where the step indices correspond to the above:

Algorithm BASHJ (k, L)
 Input: The dimension number k of current submesh M_k, and its determinant L (other parameters such as R, S, D_j, e_1, and e_2 are global variables and, hence, are not included here).
 Output: A range-join of the subsets of R and S in M_k.

```
begin
  if k = 0 then
      Processor P_{i1,...,iN} in M_k do
      Seq-RJoin (R_{i1,...,iN}, S_{i1,...,iN}, e1, e2)
  else
    1: for j := 1 to D_k do in parallel   BASHJ (k-1, j : L);
    2: for all processors P_{i1,...,iN} in M_k do in parallel
         S^k_{i1,...,iN} := S_{i1,...,iN};
    3: for i := 2 to D_k do     {Shift forwards}
          for j := i to D_k do in parallel
              for all processors P_{i1,...,j,i(k+1),...,iN} in M^j_{k-1}
              do in parallel
                  S_{i1,...,j,i(k+1),...,iN} := S_{i1,...,j-1,i(k+1),...,iN};
              BASHJ (k-1, j : L)
          end for
    4: for all processors P_{i1,...,iN} in M_k do in parallel
         S_{i1,...,iN} := S^k_{i1,...,iN};
    5: for i := D_k - 1 downto 1 do     {Shift backwards}
          for j := 1 to i do in parallel
              for all processors P_{i1,...,j,i(k+1),...,iN} in M^j_{k-1}
              do in parallel
                  S_{i1,...,j,i(k+1),...,iN} := S_{i1,...,j+1,i(k+1),...,iN};
              BASHJ~(k-1, j : L)
          end for
    6: for all processors P_{i1,...,iN} in M_k do in parallel
         S_{i1,...,iN} := S^k_{i1,...,iN}
  end if
end.
```

The algorithm starts execution by a call BASHJ $(N, [\,])$.

Example 2. Consider a simplified example in which we permute (and join) four subsets of S denoted by integers 1, 2, 3, and 4 on a 2×2 mesh. In dimension k, let $store(k)$, $restore(k)$, $fs(k)$, and $bs(k)$ denote operations of storing and restoring a subset, and *one-step* shifting a subset forwards and backwards, respectively. The whole process is illustrated in Fig. 3, where the rectangles represent the processors and the integers inside them represent the subsets of S. When a processor performs a local join operation, its corresponding rectangle is gray-colored.

Initially in Fig. 3a, processors $P_{1,1}$, $P_{2,1}$, $P_{1,2}$, and $P_{2,2}$ have subsets 1, 2, 3, and 4, respectively, in their local memory. We issue procedure call BASHJ $(2, [\,])$ to start the permutation. As $k = 2 > 0$, two recursive calls, BASHJ $(1, [1])$ and BASHJ $(1, [2])$, are issued in Step 1 simultaneously. These two recursive calls are performed individually and in parallel, one in row 1 and the other in row 2, as illustrated in Fig. 3b, Fig. 3c, Fig. 3d, Fig. 3e, Fig. 3f, and Fig. 3g.

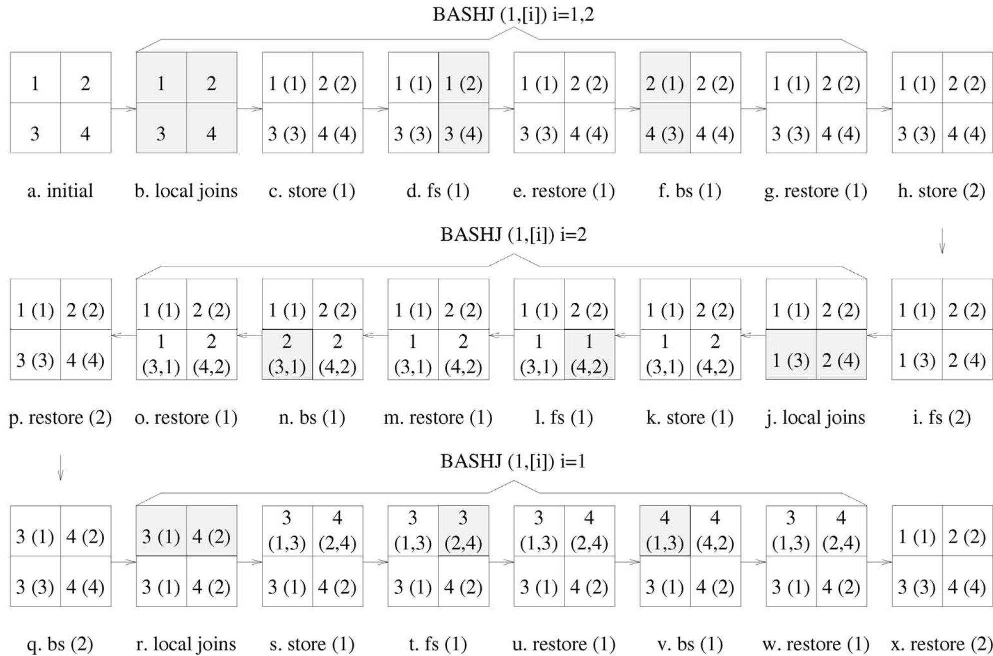


Fig. 3. BASHJ$(2, [])$: Permuting S on a 2×2 mesh.

In Fig. 3b, when further permutation in the dimension 0 occurs, four processors $P_{i,j}$ $(i, j = 1, 2)$ perform recursive procedure calls $BASHJ(0, [i, j])$ and, hence, join their current subsets simultaneously. Then, they store their current subsets of dimension 1 to the temporary copies which are shown in the parentheses in Fig. 3c, and then restore them back in Fig. 3e and Fig. 3g. In Fig. 3d and Fig. 3f, the subsets are transferred between a pair of processors within the same row in dimension 1, forwards and then backwards. After each parallel data transmission, local join operations are performed in the processors which receive subsets.

When these two parallel recursive calls terminate at Fig. 3g, we continue the algorithm in Step 2 where the processors copy their current subsets of dimension 2, as shown in Fig. 3h. Following this in Step 3, we shift the subsets one-step downwards in dimension 2 as shown in Fig. 3i, and recursively call BASHJ$(1, [2])$ in Fig. 3j, Fig. 3k, Fig. 3l, Fig. 3m, Fig. 3n, and Fig. 3o. Similarly in Step 5, we shift the subsets one-step upwards in dimension 2 as shown in Fig. 3q, and recursively call BASH$(1, [1])$ in Fig. 3r, Fig. 3s, Fig. 33t, Fig. 3u, Fig. 3v, and Fig. 3w. We also restore the subsets of dimension 2 in Steps 4 and 6, as shown in Fig. 3p and Fig. 3x. Note that, in the two recursive calls after $fs(2)$ and $bs(2)$, some processors keep two temporary copies in the parentheses, the first one for dimension 2 and the second one for dimension 1.

4.2 Analysis

It is not difficult to verify that every subset of S visits every different k-D submesh exactly once for $0 \leq k \leq N$ and, hence, the algorithm can correctly compute $R \bowtie_{e_1}^{e_2} S$.

During the permutation on a k-D submesh, since each processor keeps one extract temporary copy of its current subset of S for each dimension i $(i = 1, \ldots, k)$, it needs to keep at most $k+1$ subsets of S including the current one in memory. Remember that each processor has M blocks of memory in total, and it already uses M_R/p blocks for the local subset of R and needs to reserve one block for the resulting tuples. Thus, the available free memory for the subsets of S is $M_f = M - B_R/p - 1$, and there are $\max\{0, M_f - (k+1)B_S/p\}$ subsets which require three disk I/O operations: one for storing them to disk (Step 2) and the other two for restoring them back to memory (Steps 4 and 6). Thus, this disk I/O cost is

$$3T_{io} \cdot \max\{0, M_f - (k+1)B_S/p\}.$$

There are $2D_k - 2$ iterations in forward- and backward-shift in Steps 3 and 5, each consisting of one parallel data transmission (which requires $T_t \cdot B_S/p$ time) and one recursive call. With another recursive call in Step 1, the BASHJ algorithm has totally $2D_k - 1$ recursive calls. Hence, the running time $T(k)$ for algorithm *BASHJ* on a k-D submesh is given in the recurrence

$$T(k) = \begin{cases} T(k-1)\times(2D_k-1)+T_t\times(D_k-1)\frac{2B_S}{p}+T_{io}\times 3\max\{0, M_f-\frac{(k+1)B_S}{p}\}, & k > 0 \\ T_{lj}(R/p, S/p), & k = 0. \end{cases}$$

To further simplify the analysis, we assume that $M_f = B_S/p$, that is, only one subset of S—the current one—can fit in the memory. We then resolve the above recurrence and obtain the total cost T_{bashj} of the whole BASHJ algorithm in dimension N as follows:

$$
\begin{aligned}
T_{bashj}(R,S) =& T(N)\\
=& T_t \times \frac{B_S}{p}\cdot\left(\prod_{i=1}^{N}(2D_i-1)-1\right)+T_{lj}(R/p,S/p)\\
&\times \prod_{i=1}^{N}(2D_i-1)+3T_{io}\\
&\times \frac{B_S}{p}\left(1+\sum_{i=1}^{N-1}\prod_{j=i+1}^{N}(2D_j-1)\right).
\end{aligned}
\tag{2}
$$

Example 3. To compute the cost occurring in Example 2 in which the BASHJ algorithm permutes S on a 2×2 mesh, we apply (2), and obtain that the algorithm requires eight parallel data transmissions, nine parallel local join operations, and 12 disk I/O operations, as indicated in Fig. 3.

5 Buffering Shifting Join

5.1 Description

From the preceding analysis, we know that the parallelism of the previous shifting algorithm does not appear to be very attractive. In particular, during the jth iteration of forward-shift, all processors in the $(k-1)$-D submeshes $M_{k-1}^1,\ldots,M_{k-1}^j$ are idle because they do not receive any (new) subsets from their neighbors in other $(k-1)$-D submeshes and, hence, they cannot perform any local join operation at all. Similarly, during the jth iteration of backward-shift, all processors in submeshes $M_{k-1}^{j+1},\ldots,M_{k-1}^{D_k}$ are idle. Obviously, the larger the cost each iteration requires, the longer idle period these processors have. From the previous algorithm description, we know that the cost for each iteration of both forward- and backward-shifts is dominated by the cost for the recursive call which permutes the arriving subsets on some *active* $(k-1)$-D submeshes. Moreover, in those *active* submeshes performing the recursive calls, their processors will also become idle during the recursive calls.

To obtain better parallelism, we propose a Buffering Shifting Join (BUSHJ) algorithm that eliminates the recursive calls inside forward- and backward-shifts by allowing the processors to keep every arriving subset. In particular, if $k=0$, processor $P_{i_1,\ldots,i_N}$ in M_k performs a local join operation on its subset as in BASHJ, but also stores this subset into a sequence $Q_{i_1,\ldots,i_N}$ of subsets of S. Initially, $Q_{i_1,\ldots,i_N}$ is empty.

If $k>0$, the BUSHJ permutes the subsets on all $(k-1)$-D submeshes recursively in dimensions from 0 to $k-1$ as the first step of BASHJ, but it allows each processor $P_{i_1,\ldots,i_N}$ to store every arriving subset to $Q_{i_1,\ldots,i_N}$ during the permutation. Hence, when this step terminates, $P_{i_1,\ldots,i_N}$ has stored all $(D_1\times\cdots\times D_{k-1})$ subsets in its $(k-1)$-D submesh into $Q_{i_1,\ldots,i_N}$, whose tth element, denoted by $Q_{i_1,\ldots,i_N}[t]$, is the tth subset appended into. With $Q_{i_1,\ldots,i_N}$, $P_{i_1,\ldots,i_N}$ does not need to make a temporary copy of current subset as in the second step of BASHJ. The BUSHJ algorithm now starts a loop with $(D_1\times\cdots\times D_{k-1})$ steps. During the tth step for $1\le t\le D_1\times\cdots\times D_{k-1}$, the tth subset in $Q_{i_1,\ldots,i_N}$ is transferred in forward- and backward-shift in turn as in the previous shifting algorithm, but in each iteration, the recursive call in the previous algorithm here is replaced by a local join operation for the arriving subset and an operation for storing this arriving subset. Thus, the BUSHJ algorithm for N-D meshes can be summarized as the following recursive algorithm.

Algorithm BUSHJ (k, L)}
Input: The current dimension number k of submesh M_k, and its determinant L.
Output: A range-join of the subsets of R and S in M_k.

```
begin
  if k = 0 then
      Processor P_{i1,...,iN} in M_k do
          Seq-RJoin (R_{i1,...,iN}, S_{i1,...,iN}, e1, e2);
          Q_{i1,...,iN} := S_{i1,...,iN}
  else
   1: for j := 1 to D_k do in parallel     BUSHJ (k - 1, j : L);
    for t := 1 to D_1 × ··· × D_{k-1} do
      2: for all processors P_{i1,...,iN} in M_k do in parallel
           S_{i1,...,iN} := Q_{i1,...,iN}[t];
      3: for i := 2 to D_k do      {Forward-shift}
           for j := i to D_k do in parallel
              for all processors P_{i1,...,j,ik+1,...,iN} in M^j_{k-1}
              do in parallel
                 S_{i1,...,j,ik+1,...,iN} := S_{i1,...,j-1,ik+1,...,iN};
                 Seq-RJoin (R_{i1,...,iN}, S_{i1,...,iN}, e1, e2);
                 if k < N then
           Q_{i1,...,j,ik+1,...,iN} := Q_{i1,...,j,ik+1,...,iN} ∪ S_{i1,...,j,ik+1,...,iN}
    end for
      4: for all processors P_{i1,...,iN} in M_k do in parallel
         S_{i1,...,iN} := Q_{i1,...,iN}[t];
      5: for i := D_k - 1 downto 1 do      {Backward-shift}
           for j := 1 to i do in parallel
              for all processors P_{i1,...,j,ik+1,...,iN} in M^j_{k-1}
              do in parallel
              S_{i1,...,j,ik+1,...,iN} := S_{i1,...,j+1,ik+1,...,iN};
              Seq-RJoin (R_{i1,...,iN}, S_{i1,...,iN}, e1, e2);
              if k < N then
           Q_{i1,...,j,ik+1,...,iN} := Q_{i1,...,j,ik+1,...,iN} ∪ S_{i1,...,j,ik+1,...,iN}
        end for
    end for
  end if
end.
```

As before, the algorithm starts execution by a call BUSHJ $(N,[])$.

As mentioned above, the purpose of storing subsets into $Q_{i_1,\ldots,i_N}$ is to permute them in higher dimensions. Thus, when $k=N$, the processors do not need to store the arriving subsets anymore because no more further permutation will be performed and, hence, the two simple conditions inside Steps 3 and 5 are required.

Example 4. We consider the same problem in Example 2, and solve it by using the BUSHJ algorithm now, as illustrated in Fig. 4. We use the same notation and representations which are used in Example 2.

As we can see, there is only one recursive call in dimension 1 instead of three in the previous example. When the recursive call terminates, every processor stores all subsets which are initially stored in its row.

Then in dimension 2, the algorithm executes a loop with two iterations to permute the subsets along the columns.

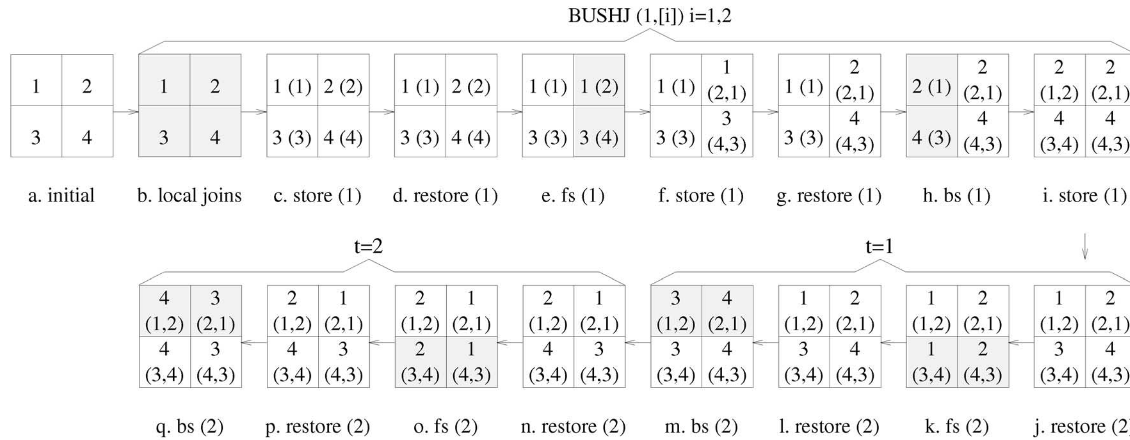


Fig. 4. BUSHJ($2, []$): Permuting S on a 2×2 mesh.

Each iteration has one forward-shift and one backward-shift as before, but the recursive calls in the previous example are replaced by local join operations here.

5.2 Analysis

As BASHJ, BUSHJ permutes every subset of S to every processor exactly once and, hence, it is also correct. However, unlike BASHJ which requires each processor to store at most $(N+1)$ subsets during the permutation, the BUSHJ requires each processor to store at most all subsets which are initially stored the $(N+1)$-D submesh where the processor is in. That is, each processor needs to store at most $(D_1 \times \cdots \times D_{N-1})$ subsets of S during the permutation. Hence, BUSHJ has a higher storage requirement than BASHJ. On the other hand, the parallelism of BUSHJ is higher than that of BASHJ, and as a result, it requires fewer data transmissions, parallel disk I/O operations, and local join operations.

To simplify the analysis, we also make the assumption that only one subset of S can fit in the free memory at a time ($M_f = B_S/p$). Thus, both Steps 2 and 4 require one disk I/O operation. Steps 3 and 5 are two loops with $D_k - 1$ iterations, each consisting of one parallel data transmission (which requires $T_t \cdot B_S/p$ time), a local join operation (which requires $T_{lj}(R/p, S/p)$ time), and a disk I/O operation (which requires $T_{io} \cdot B_S/p$ time). Moreover, Steps 2 to 5 are repeated $D_1 \times \cdots D_{k-1}$ times, each for a subset in $Q_{i_1,\ldots,i_N}$. Hence, the running time $T(k)$ for the BUSHJ algorithm on a k-D submesh is given in the recurrence relation

$$T(k) = \begin{cases} T(k-1)+\prod_{i=1}^{k-1} D_i\times\Big(T_{io}\times\frac{2B_S}{p}+T_t\times(D_k-1)\frac{2B_S}{p}+T_{lj}(R/p,S/p)\times(2D_k-2)\Big), & k=N \\ T(k-1)+\prod_{i=1}^{k-1} D_i\times\Big(T_{io}\times D_k\frac{2B_S}{p}+T_t\times(D_k-1)\frac{2B_S}{p}+T_{lj}(R/p,S/p)\times(2D_k-2)\Big), & 0<k<N \\ T_{lj}(R/p,S/p)+T_{io}\times\frac{B_S}{p}, & k=0. \end{cases}$$

The above equation clearly indicates that we do not need the cost for copying $D_1 \times \cdots \times D_{N-1}$ subsets to disk when permuting in dimension N. We solve the above recurrent relation and have the total cost T_{bushj} of the whole BUSHJ algorithm on an N-D mesh as follows:

$$\begin{aligned} T_{bushj}(R,S) &= T(N) \\ &= T_t \times \frac{2B_S}{p}\sum_{i=1}^{N}\left((D_i-1)\prod_{j=1}^{i-1}D_j\right) + T_{lj}(R/p,S/p) \\ &\quad \times \left(1+2\sum_{i=1}^{N}\left((D_i-1)\prod_{j=1}^{i-1}D_j\right)\right) \\ &\quad + T_{io}\times\frac{B_S}{p}\left(1+\sum_{i=1}^{N-1}\left(2D_i\prod_{j=1}^{i-1}D_j\right)+2\prod_{j=1}^{N-1}D_j\right). \end{aligned} \quad (3)$$

Example 5. We apply (3) to compute the cost in Example 4 in which we use the BUSHJ algorithm on a 2×2 mesh. Then, we know that the algorithm requires six parallel data transmissions, seven parallel local join operations, and nine disk I/O operations, as indicated in Fig. 4.

6 Recursive Hamiltonian-Cycle Join

6.1 Overview of Hamiltonian-Cycle Approach

Both BASHJ and BUSHJ algorithms use the shifting approach to permute the data on an N-D mesh. They can work well if the cost T_t of transferring a block of data between two neighboring processors is high and the cost T_{lj} of a local join operation is low. Otherwise, their performance will become less attractive because they suffer the following two major problems:

Asynchronization of Local Join Operations. When some processors are performing the local join operations, others are idle because no new subset arrives.

High Storage Requirement. The BASHJ and BUSHJ algorithms require each processor to store $N+1$ and $D_1 \times \cdots \times D_{N-1}$ subsets of S, respectively. Additional disk I/O operations are needed to shuffle some subsets between the memory and disk, as we have seen in their analysis.

In the remaining of this paper, we address the above two problems of the shifting approach, and propose a different data permutation approach called *Hamiltonian-Cycle* approach. This new approach requires a minimum number of local join operations and two subsets of S stored in some corner processors during the permutation. It consists of two main steps:

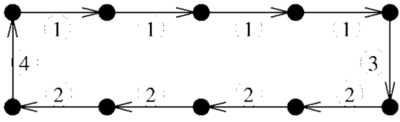


Fig. 5. Permuting data along a Hamiltonian cycle.

1. Construct a Hamiltonian cycle on the given mesh and make every processor know its own successor in the cycle.
2. Repeatedly transfer a subset of S from each processor to its successor in the Hamiltonian cycle until each subset visits (and is joined at) every processor exactly once.

A Hamiltonian cycle is a cycle which connects every processor *exactly once*. Hence, every processor has only one distant successor and it knows the index of its successor after Step 1. As suggested in Fig. 5, all subsets of S are then transferred along the cycle of length p in Step 2, which can be implemented by the following statements:

All processors simultaneously perform a local join operation;
for $i := 1$ **to** $p - 1$ **do**
Each processor transfers its current subset of S to its successor;
All processors simultaneously perform a local join operation
end for

Clearly, Step 2 requires p parallel local join operations in total, each for every processor to join one different subset of S.

Recall that, in SIMD and SPMD computers, parallel data transmissions can only be carried out in the same dimension and in the same direction. Other data transmissions in different dimensions or in different directions are delayed until the current transmissions are completed and the system (or the algorithms) schedules them to start. For example, data transmissions in Fig. 5 follow the order indicated by the numbers associated with the arrows. In an N-D mesh, there are N different dimensions, each with two different directions (forward and backward). Thus, each iteration in Step 2 takes $2N$ different concurrent data transmissions for transferring every subset of S from a processor to its successor.

For the same reason of the delay of data transmissions in different dimensions and directions, some corner processors (such as the left-top and right-bottom corner processors in Fig. 5) in every dimension need to store one additional subsets of S temporarily in each iteration of Step 2. This is required when their predecessors have already transferred subsets to them but they have not yet transferred their current subsets to their successors. With the same assumption that only one subset of S can fit in memory at a time, each iteration of Step 2 requires two disk I/O operations to shuffle one subset in and out of memory once. Therefore, the total cost of Step 2 is

$$T_{perm} = T_t \times 2N(p-1)\frac{B_S}{p} + T_{lj}(R/p, S/p) \times p + T_{io} \times 2(p-1)\frac{B_S}{p}. \tag{4}$$

It is obvious that every subset of S visits and is joined at each processor exactly once along the Hamiltonian cycle in Step 2 and, hence, the Hamiltonian-cycle approach is correct if a Hamiltonian cycle can be constructed correctly on the given mesh.

Thus, the remaining problem is how to construct a Hamiltonian cycle on the given mesh. Two different methods for this problem are proposed in the following and they result in two different join algorithms, namely *Recursive Hamiltonian-Cycle Join* (REHCJ) and *Parallel Hamiltonian-Cycle Join* (PAHCJ). In what follows, we first give the important definitions which are used in both algorithms, and then prove that a mesh with even number of processors has Hamiltonian cycles, and finally present and analyze these algorithms in this and the next sections, respectively.

6.2 Definitions

An N-D mesh M is called *even-sized* for $N \geq 2$ if the number of processors is even (i.e., at least one dimension degree D_i is even) and *odd-sized* otherwise. For a 2-D odd-sized mesh, we have proved that it is not Hamiltonian (i.e., no Hamiltonian cycle can be constructed on it), and have presented a method to construct a partial Hamiltonian cycle excluding only a corner processor [5]. Both of the proof and construction method can be extended for any N-D odd-sized mesh and, hence, in what follows, we concentrate only on the even-sized meshes. Moreover, without loss of generality, we assume that in an even-sized mesh M, the dimension degree D_2 of dimension 2 is always even.

In M, let M_k $(0 \leq k \leq N)$ be a k-D submesh with determinant L. For M_k, we define three kinds of processors which are important for the Hamiltonian cycle construction on M:

- A_k: a processor in M_k with index
$$(\overbrace{1,\ldots,1}^{N-k}, \overbrace{i_{k+1}\ldots,i_N}^{=L});$$
- B_k: a processor in M_k with index
$$(\overbrace{1,\ldots,1}^{N-k-1}, 2, \overbrace{i_{k+1}\ldots,i_N}^{=L});$$
- Z_k: a processor in M_k with index
$$(\overbrace{1,\ldots,1}^{N-k-2}, 2, 1, \overbrace{i_{k+1}\ldots,i_N}^{=L})$$
if $k \geq 2$ or with index
$$(D_1, \overbrace{i_{k+1}\ldots,i_N}^{=L})$$
otherwise.

As shown in the above definitions, the indices of A_k, B_k and Z_k in dimensions from $k+1$ to N are given by determinant L of M_k. When $k = 0$, M_0 has only one processor which is A_0, and it does not have B_0 and Z_0. When $k = 1$ and $D_2 = 2$, B_1 and Z_1 coincide. When $k \geq 2$, A_k is adjacent to B_k and Z_k. For a given even-sized mesh, both REHCJ and PAHCJ algorithms construct the same Hamiltonian cycle in which A_N, B_N, and Z_N are the *first*, *second*, and the *last* processors respectively, and we define the direction of this Hamiltonian cycle to be *forward*.

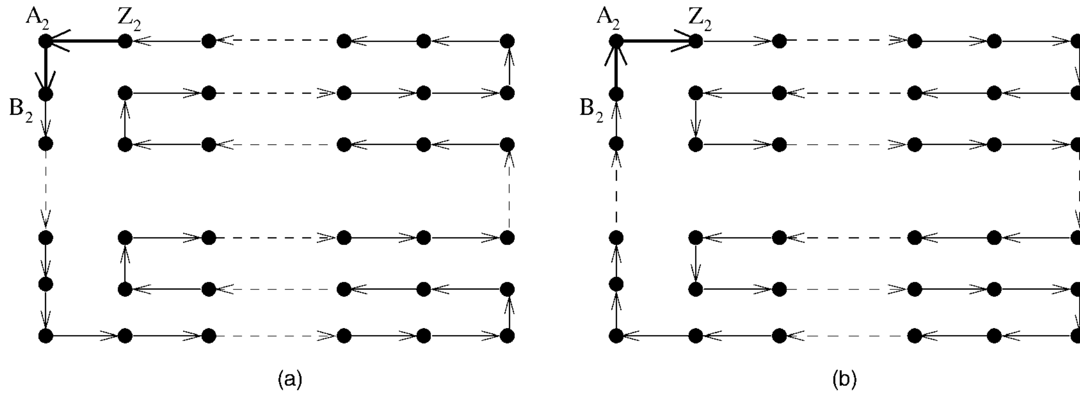


Fig. 6. A Hamiltonian cycle on a 2-D even-sized mesh in two opposite directions.

We call A_k, B_k, and Z_k the A, B, and Z processors in M_k, respectively. If M_k is the jth submesh of a $(k+1)$-D submesh M_{k+1}, we further denote its A, B, and Z processors by A_k^j, B_k^j, and Z_k^j, respectively. According to the above definitions, A_{k+1}, B_{k+1} and Z_{k+1} in M_{k+1} for $k \geq 2$ can be obtained recursively as follows:

$$\begin{aligned} A_{k+1} &= A_k^1 \\ B_{k+1} &= A_k^2 \\ Z_{k+1} &= B_k^1 \end{aligned} \qquad \text{(Def-1)}$$

Thus, every processor $P_{i_1,\dots,i_N}$ in the mesh M acts as a *kind* processor in one or more dimensions, where kind is either A, B, and Z. The highest dimension pd among these dimensions of $P_{i_1,\dots,i_N}$ is called the *prime dimension* of $P_{i_1,\dots,i_N}$, and the corresponding kind pk is called the *prime kind*. In the PAHCJ algorithm, every processor uses its pd and pk to compute their successors.

Example 6. From the 3-D mesh M shown in Fig. 1, its A, B, and C processors are $A_3 = P_{1,1,1}$, $B_3 = P_{1,1,2}$, and $Z_3 = P_{1,2,1}$, respectively. For each of its jth 2-D submesh $(j = 1, \dots, 4)$, the A, B, and Z processors are $A_2^j = P_{1,1,j}$, $B_2^j = P_{1,2,j}$, and $Z_2^j = P_{2,1,j}$, respectively.

Similarly, for each of the ith 1-D submesh of any its jth 2-D submesh $(i = 1, \dots, 4)$, the A, B, and Z processors are $A_1^i = P_{1,i,j}$, $B_1^i = P_{2,i,j}$, and $Z_1^i = P_{4,i,j}$, respectively. Note that the Z processors in dimension 1 is different from those in higher dimensions. In dimension 0, every processor is the A processor of its M_0 which has neither B nor Z processor.

Processor $P_{1,1,1}$ is an A processor in dimensions from 1 to 3, and processor $P_{1,1,2}$ is the B processor in dimension 3 and an A processor in dimensions 1 and 2. Both of their prime dimensions are 3. Processor $P_{3,3,3}$ is only an A processor in dimension 0 and, hence, its prime dimension is 0.

We denote an edge *from* processor u *to* v by (u, v) and prove that every even-sized mesh contains a Hamiltonian cycle in the following theorem.

Theorem 1. *Every N-D even-sized mesh is Hamiltonian for* $N \geq 2$.

Proof. Proof by induction on N. When $N = 2$, we can always construct a Hamiltonian cycle (shown in Fig. 6a) containing two edges (A_2, B_2) and (Z_2, A_2), or another Hamiltonian cycle in reverse direction (shown in Fig. 6b) containing two edges (A_2, Z_2) and (B_2, A_2).

Assume that the theorem is true for $N - 1 \geq 2$. An N-D mesh M has D_N different $(N-1)$-D submeshes. By induction, we can construct a Hamiltonian cycle on every $M_{N-1}^j (j = 1, \dots D_N)$ in two different directions according to the parity of j: If j is even, the Hamiltonian cycle contains two edges (A_{N-1}^j, B_{N-1}^j) and (Z_{N-1}^j, A_{N-1}^j); otherwise, the direction of Hamiltonian cycle is backward and the Hamiltonian cycle contains two edges (A_{N-1}^j, Z_{N-1}^j) and (B_{N-1}^j, A_{N-1}^j). The whole Hamiltonian cycle on M can be constructed by performing the following two steps:

1. For all odd j $(1 \leq j < D_N)$, remove two edges (A_{N-1}^j, Z_{N-1}^j) and $(Z_{N-1}^{j+1}, A_{N-1}^{j+1})$, and add two new edges $(A_{N-1}^j, A_{N-1}^{j+1})$ and $(Z_{N-1}^{j+1}, Z_{N-1}^j)$;
2. For all even j $(1 \leq j < D_N)$, remove two edges (A_{N-1}^j, B_{N-1}^j) and $(B_{N-1}^{j+1}, A_{N-1}^{j+1})$, and add two new edges $(A_{N-1}^j, A_{N-1}^{j+1})$ and $(B_{N-1}^{j+1}, B_{N-1}^j)$. □

Example 7. Fig. 7 show how to construct a Hamiltonian cycle on the $4 \times 4 \times 4$ mesh given in Fig. 1 from four Hamiltonian cycles on its four 2-D submeshes, where the dotted-lines, gray-lines, and bold-lines are the edges removed, added and connecting A, B, and Z processors, respectively. Note that, A_2^1, A_2^2, and B_2^1 become A_3, B_3 and Z_3, respectively, after the Hamiltonian cycle on the 3-D mesh M is constructed. These changes follow the formula in (Def-1).

6.3 Description

An algorithm for constructing a Hamiltonian cycle can be derived straightforwardly from the above proof: Starting with Hamiltonian cycles on every 2-D submesh as shown in Fig. 6, the algorithm repeatedly executes the above two steps to construct Hamiltonian cycles for the submeshes, one dimension higher each time, until the whole cycle is constructed on M.

However, we can observe that, in the above way, the removal of two edges connecting processors A_{k-1}^j, B_{k-1}^j, and Z_{k-1}^j in a Hamiltonian *cycle* on each M_{k-1}^j yields only a *partial Hamiltonian path* connecting all processors in the submesh except A_{k-1}^j, and the two end points of this path are B_{k-1}^j and Z_{k-1}^j. Thus, it would be more desirable to directly construct such a partial Hamiltonian path rather

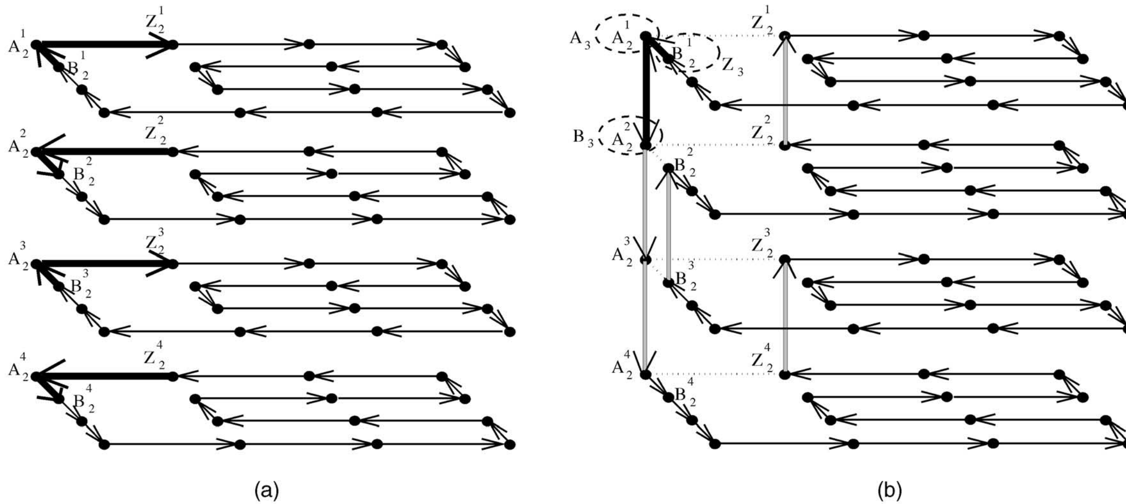


Fig. 7. Constructing a Hamiltonian cycle on a $4 \times 4 \times 4$ 3-D even-sized mesh: (a) Hamiltonian cycles on four 2-D submeshes. (b) Hamiltonian cycle on M.

than to construct a Hamiltonian cycle first and then to remove its two edges. Moreover, by directly constructing a partial Hamiltonian path, we can reduce the dimension for the base case of the above algorithm from 2 to 0 and, hence, simplify the description of the algorithm. We now describe the detailed algorithm for constructing the same Hamiltonian cycle on an even-sized mesh.

Let $\mathcal{P}_k$ denote the *forward* partial Hamilton path on M_k which starts at processor B_k and ends at Z_k, and $\overline{\mathcal{P}}_k$ the *backward* path (i.e., starts at B_k and ends at Z_k). Both $\mathcal{P}_k$ and $\overline{\mathcal{P}}_k$ contain each processor in M_k exactly once, but exclude A_k. On a 0-D submesh, both $\mathcal{P}_0$ and $\overline{\mathcal{P}}_0$ are the paths containing only a processor without any edge. Now, we have the following recurrent definition of $\mathcal{P}_k$ for M_k, where $1 \le k \le N$.

$$\mathcal{P}_k = \begin{cases} (A_{k-1}^2, A_{k-1}^3), (A_{k-1}^3, A_{k-1}^4), \\ \ldots, (A_{k-1}^{D_k-1}, A_{k-1}^{D_k}), (A_{k-1}^{D_K}, B_{k-1}^{D_k}), \mathcal{P}_{k-1}^{D_k}, \\ (Z_{k-1}^{D_K}, Z_{k-1}^{D_k-1}), \overline{\mathcal{P}}_{k-1}^{D_k-1}, (B_{k-1}^{D_k-1}, B_{k-1}^{D_k-2}), \\ \ldots, (Z_{k-1}^2, Z_{k-1}^1), \overline{\mathcal{P}}_{k-1}^1 & \text{if } D_k \text{ is even} \\ (A_{k-1}^2, A_{k-1}^3), (A_{k-1}^3, A_{k-1}^4), \\ \ldots, (A_{k-1}^{D_k-1}, A_{k-1}^{D_k}), (A_{k-1}^{D_K}, Z_{k-1}^{D_k}), \overline{\mathcal{P}}_{k-1}^{D_k}, \\ (B_{k-1}^{D_K}, B_{k-1}^{D_k-1}), \mathcal{P}_{k-1}^{D_k-1}, (Z_{k-1}^{D_k-1}, Z_{k-1}^{D_k-2}), \\ \ldots, (Z_{k-1}^2, Z_{k-1}^1), \overline{\mathcal{P}}_{k-1}^1 & \text{if } D_k \text{ is odd.} \end{cases} \quad \text{(Def-2)}$$

Although the above definition looks complicated, the construction of $\mathcal{P}_k$ is simple. It also works in a recursive fashion like the construction of a Hamiltonian cycle on M_k given in the proof of Theorem 1, with the difference that it constructs a partial Hamiltonian path on each M_{k-1}^j instead of a Hamiltonian cycle. The direction of the partial Hamiltonian path on M_{k-1}^j depends on the parity of j: If j is even, a forward path $\mathcal{P}_{k-1}^j$ is constructed; otherwise, a backward path $\overline{\mathcal{P}}_{k-1}^j$ is constructed. An additional edge is used to connect the end processor of the path on M_{k-1}^j to the beginning processor of the path on M_{k-1}^{j-1}. Also depending on the parity of j, this edge is either $(Z_{k-1}^j, Z_{k-1}^{j-1})$ (between $\mathcal{P}_{k-1}^j$ and $\overline{\mathcal{P}}_{k-1}^{j-1}$) or $(B_{k-1}^j, B_{k-1}^{j-1})$ (between $\overline{\mathcal{P}}_{k-1}^j$ and $\mathcal{P}_{k-1}^{j-1}$) as indicated in the above definition. Moreover, the construction $\mathcal{P}_k$ builds a path from A_{k-1}^2 to $A_{k-1}^{D_K}$, and then adds an edge from $A_{k-1}^{D_k}$ to the beginning processor of the path on $M_{k-1}^{D_k}$, which is either $B_{k-1}^{D_k}$ or $Z_{k-1}^{D_k}$, depending on the parity of D_k.

Let *Forward-HP* (k, L) and *Backward-HP* (k, L) be two mutual recursive functions which returns a forward and backward partial Hamiltonian path on a k-D submesh with determinant L, respectively. According to the above definition of $\mathcal{P}_k$, function Forward-HP (k, L) can be summarized as follows, where the paths are represented as sequences of edges and the operation $\oplus$ means the concatenation of two paths:

function Forward-HP (k, L)
  Input: The current dimension number k of submesh M_k, and its determinant L
  Output: The forward partial Hamiltonian path $\mathcal{P}_k$ on M_k
begin
  if $k > 0$ **then**
    $\mathcal{P}_k = \langle\ \rangle$;   {Initialize}
    for $j := 2$ **to** $D_k - 1$ **do** $\mathcal{P}_k := \mathcal{P}_k \oplus \langle (A_{k-1}^j, A_{k-1}^{j+1}) \rangle$;

    if D_k is even **then** $\mathcal{P}_k := \mathcal{P}_k \oplus \langle (A_{k-1}^{D_k}, B_{k-1}^{D_k}) \rangle$
    else $\mathcal{P}_k := \mathcal{P}_k \oplus \langle (A_{k-1}^{D_k}, Z_{k-1}^{D_k}) \rangle$;
    for $j := D_k$ **downto** 2 **do**
      if j is even **then** $\mathcal{P}_k := \mathcal{P}_k \oplus$ Forward-HP $(k-1, L : j) \oplus \langle (Z_{k-1}^j, Z_{k-1}^{j-1}) \rangle$
      else $\mathcal{P}_k := \mathcal{P}_k \oplus$ Backward-HP $(k-1, L : j) \oplus \langle (B_{k-1}^j, B_{k-1}^{j-1}) \rangle$;
    $\mathcal{P}_k := \mathcal{P}_k \oplus$ Backward-HP $(k-1, L : 1)$;
    Return $\mathcal{P}_k$
  end if
end

In correspondence to $\mathcal{P}_k$, we have the following definition of $\overline{\mathcal{P}}_k$ for $1 \le k \le N$:

$$\overline{\mathcal{P}}_k = \begin{cases} \mathcal{P}^1_{k-1}, (Z^1_{k-1}, Z^2_{k-1}), \overline{\mathcal{P}}^2_{k-1}, (B^2_{k-1}, B^3_{k-1}), \\ \ldots, (Z^{D_k-1}_{k-1}, Z^{D_k}_{k-1}), \overline{\mathcal{P}}^{D_k}_{k-1}, \\ (B^{D_k}_{k-1}, A^{D_k}_{k-1}), (A^{D_k}_{k-1}, A^{D_k-1}_{k-1}), \\ \ldots, (A^4_{k-1}, A^3_{k-1}), (A^3_{k-1}, A^2_{k-1}) & \text{if } D_k \text{ is even} \\ \mathcal{P}^1_{k-1}, (Z^1_{k-1}, Z^2_{k-1}), \overline{\mathcal{P}}^2_{k-1}, (B^2_{k-1}, B^3_{k-1}), \\ \ldots, (B^{D_k-1}_{k-1}, B^{D_k}_{k-1}), \mathcal{P}^{D_k}_{k-1}, \\ (Z^{D_k}_{k-1}, A^{D_k}_{k-1}), (A^{D_k}_{k-1}, A^{D_k-1}_{k-1}), \\ \ldots, (A^4_{k-1}, A^3_{k-1}), (A^3_{k-1}, A^2_{k-1}) & \text{if } D_k \text{ is odd.} \end{cases} \quad \text{(Def-3)}$$

This leads to the following function Backward-HP:

function Backward-HP (k, L)
  Input: The current dimension number k of submesh M_k, and its determinant L
  Output: The backward partial Hamiltonian path $\overline{\mathcal{P}}_k$ on M_k
begin
  if $k > 0$ **then**
    $\overline{\mathcal{P}}_k = \langle\ \rangle$; {Initialize}
    for $j := 1$ **to** $D_k - 1$ **do**
      if j is odd **then** $\overline{\mathcal{P}}_k := \overline{\mathcal{P}}_k \oplus$ Forward-HP $(k-1, L : j) \oplus \langle (Z^j_{k-1}, Z^{j+1}_{k-1}) \rangle$
      else $\overline{\mathcal{P}}_k := \overline{\mathcal{P}}_k \oplus$ Backward-HP $(k-1, L : j) \oplus \langle (B^j_{k-1}, B^{j+1}_{k-1}) \rangle$;
    if D_k is odd **then** $\overline{\mathcal{P}}_k := \overline{\mathcal{P}}_k \oplus$ Forward-HP $(k-1, L : D_k) \oplus \langle (Z^{D_k}_{k-1}, A^{D_k}_{k-1}) \rangle$
    else $\overline{\mathcal{P}}_k := \overline{\mathcal{P}}_k \oplus$ Backward-HP $(k-1, L : D_k) \oplus \langle (B^{D_k}_{k-1}, A^{D_k}_{k-1}) \rangle$;
    for $j = D_k - 1$ **downto** 2 **do**
    $\overline{\mathcal{P}}_k := \overline{\mathcal{P}}_k \oplus \langle (A^{j+1}_{k-1}, A^j_{k-1}) \rangle$;
    Return $\overline{\mathcal{P}}_k$
  end if
end

By using the above two mutual recursive functions, the REHCJ algorithm uses a single processor $P_{0,\ldots,0}$ to construct the Hamiltonian cycle on mesh M in the following statement:

$$\mathcal{C}_N := \langle (A_N, B_N) \rangle \oplus \text{ Forward-HP}(N, [\,]) \oplus \langle (Z_N, A_N) \rangle.$$

That is, $P_{0,\ldots,0}$ first connects edge (A_N, B_N), then constructs a forward partial Hamiltonian path $\mathcal{P}_N$ starting at B_N and ending at Z_N, and finally connects edge (Z_N, A_N) to complete the cycle. After having constructed the cycle, $P_{0,\ldots,0}$ broadcasts the cycle to all other processors by using an SIMD/SPMD mesh broadcast algorithm [11], so that every processor knows its own successor in the cycle and hence is able to perform the permutation phase described in the previous subsection.

6.4 Analysis

As indicated in the above two functions, $P_{0,\ldots,0}$ constructs the whole Hamiltonian cycle by including every node (processor) in M exactly once and, hence, the total in-memory computation cost is $T_c \cdot p$ with the assumption that the operation of including an edge into a list requires the same cost as the operation of comparing two numbers. The broadcast of the resulting cycle from $P_{0,\ldots,0}$ requires $\sum_{i=1}^N (D_i - 1)$ parallel data transmissions [11]. We assume that the data size for storing the cycle is less than one block and, hence, the cost of this broadcast operation is $T_t \cdot \sum_{i=1}^N (D_i - 1)$. Thus, the total cost T_{rehcj} of the REHCJ algorithm is the sum of the costs for constructions and broadcasting, as well as T_{perm} of permutation, that is

$$\begin{aligned} T_{rehcj}(R,S) &= T_c \times p + T_t \times \sum_{i=1}^N (D_i - 1) + T_{perm} \\ &= T_t \times \left(2N(p-1)\frac{B_S}{p} + (p - N) \right) \\ &\quad + T_{lj}(R/p, S/p) \times p \\ &\quad + T_{io} \times 2(p-1)\frac{B_S}{p} + T_c \times p. \end{aligned} \tag{5}$$

7 Parallel Hamiltonian-Cycle Join

7.1 Description

The above REHCJ algorithm achieves a high performance on permuting the data after the Hamiltonian cycle is constructed, but its parallelism of constructing the cycle can be further improved. In the REHCJ algorithm, the Hamiltonian cycle is constructed sequentially by a single processor $P_{0,\ldots,0}$ and, in the SIMD/SPMD computers, all other processors are idle during the construction. In addition, the resulting Hamiltonian cycle is broadcasted to other processors and, hence, additional data transmissions are required as indicated in (5).

In this section, we improve the REHCJ algorithm by developing a new method to construct the same Hamiltonian cycle for a given even-sized mesh, which results in another algorithm called *Parallel Hamiltonian-Cycle Join* (PAHCJ) algorithm. Instead of using a single processor to construct the cycle sequentially, the PAHCJ algorithm uses all processors to find their own successors in the Hamiltonian cycle simultaneously. That is, in the PAHCJ algorithm all processors obtain the indices of their successors independently without a preprocessing operation of constructing a complete cycle which requires additional data transmissions.

The PAHCJ algorithm is based on the REHCJ algorithm and uses its two mutually recursive definitions (Def-2) and (Def-3) of forward and backward partial Hamiltonian paths. In the PAHCJ algorithm, each processor $P_{i_1,\ldots,i_N}$ obtains its successor's index according to its own index in the following two steps:

Step 1. Determine its prime dimension pd and prime kind pk, where pk is either A, B, or Z, and pd is the highest dimension in which the processor $P_{i_1,\ldots,i_N}$ acts as an A, B, or Z processor.

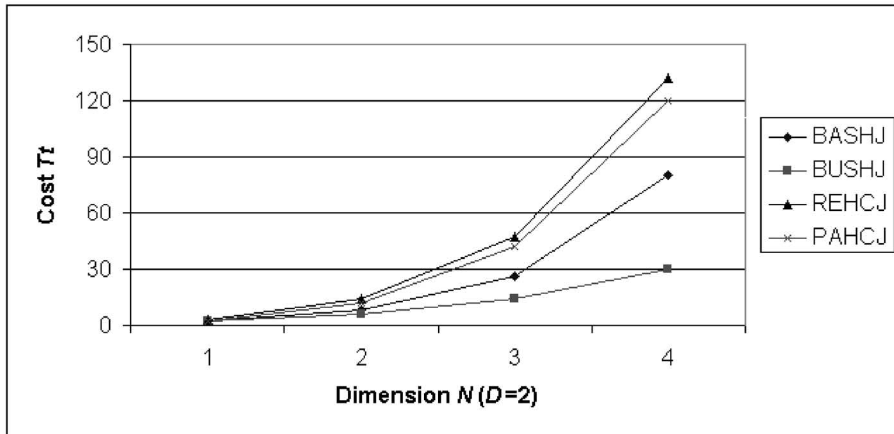
Step 2. Increment or decrement one of its coordinates of its index by 1 to obtain its successor's index.

TABLE 1
Performance Comparison among the Four Algorithms

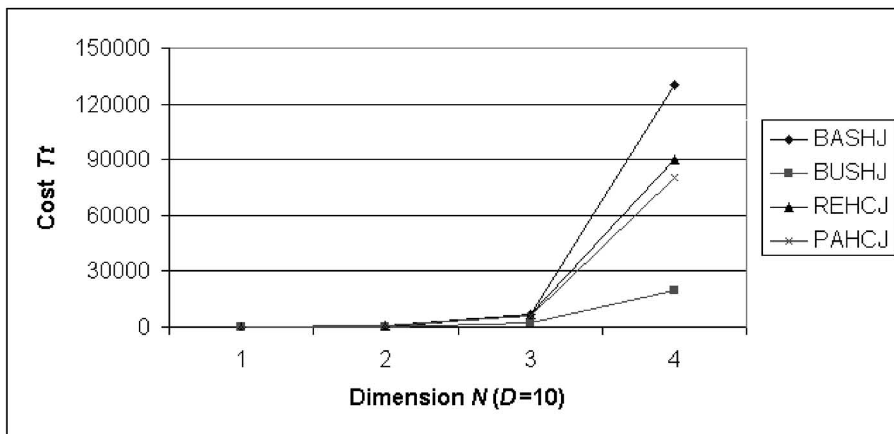
Costs	BASHJ	BUSHJ	REHCJ	PAHCJ
T_t	$\prod_{i=1}^{N}(2D_i-1)-1$	$2\sum_{i=1}^{N}((D_i-1)\prod_{j=1}^{i-1}D_j)$	$2N(p-1)+$ $(p-N)/\frac{B_S}{p}$	$2N(p-1)$
T_{lj}	$\prod_{i=1}^{N}(2D_i-1)$	$2\sum_{i=1}^{N}((D_i-1)\prod_{j=1}^{i-1}D_j)$ $+1$	p	p
T_{io}	$3\sum_{i=1}^{N-1}\prod_{j=i+1}^{N}(2D_j-1)$ $+3$	$\sum_{i=1}^{N-1}(2D_i\prod_{j=1}^{i-1}D_j)$ $+2\prod_{j=1}^{N-1}D_j+1$	$2(p-1)$	$2(p-1)$
T_c	0	0	p	$N+4$
$\|Subset(S)\|$	$N+1$	$\prod_{i=1}^{N-1}D_i$	2	2

TABLE 2
Values of System Parameters

System Parameter	Value	System Parameter	Value
T_t	4 msec	T_{io}	25 msec
T_c	3 μsec	T_{lj}	250 msec

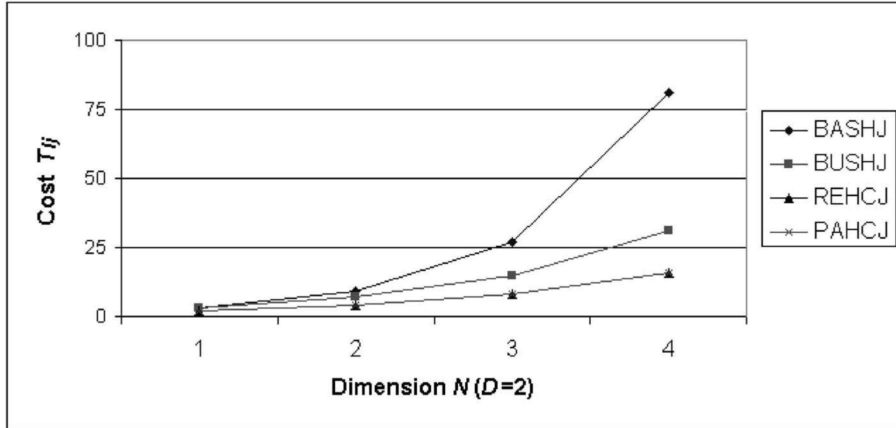


(a)

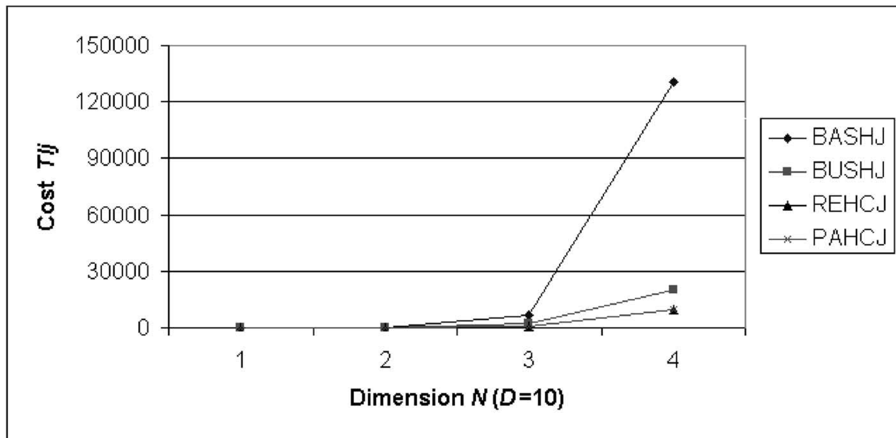


(b)

Fig. 8. Cost comparison of data transfer. (a) $D=2$ and (b) $D=10$.



(a)



(b)

Fig. 9. Cost comparison of local join operations.

In Step 1, $P_{i_1,\ldots,i_N}$ determines its pd and pk by inspecting the coordinates of its index from dimension N to 0, as described in the following procedure *find-prime*:

procedure find-prime $((i_1,\ldots,i_N), \mathbf{var} pd, \mathbf{var} pk)$
  Input: The index $(i_1,\ldots,i_N)$ of the processor
  Output: The prime dimension pd and prime kind pk of the processor
begin
  $pd := N$; $pk := undefined$;
  while $pk = undefined$ **do**
1: **if** $pd = 0$ **then** $pk := A$
2: **elseif** $pd = 1$ **and** $i_1 = D_1$ **then** $pk := Z$
3: **elseif** $(i_1, i_2, \ldots, i_{pd-2}) = (1, 1, \ldots, 1)$ **and** $i_{pd-1} = 1$ **and** $i_{pd} = 1$ **then** $pk := A$
4: **elseif** $(i_1, i_2, \ldots, i_{pd-2}) = (1, 1, \ldots, 1)$ **and** $i_{pd-1} = 1$ and $i_{pd} = 2$ **then** $pk := B$
5: **elseif** $(i_1, i_2, \ldots, i_{pd-2}) = (1, 1, \ldots, 1)$ **and** $i_{pd-1} = 2$ **and** $i_{pd} = 1$ **then** $pk := Z$
  else $pd := pd - 1$;
  end while
end.

In the above procedure, conditions 1 and 2 cope with the special cases in dimensions 0 and 1, and conditions 3-5 correspond to the definitions of A, B, and Z, respectively.

After having determined its pd and pk in Step 1, processor $P_{i_1,\ldots,i_N}$ can obtain its successor's index in Step 2 by simply incrementing or decrementing one of its coordinates. This is because, the successor of every processor in the Hamiltonian cycle of a mesh must be one of its neighbors whose indices differ by 1 from its index in exactly one coordinate. When $pd = N$, the successor of $P_{i_1,\ldots,i_N}$ is

- the B processor in dimension N whose index can be obtained by incrementing its i_{pd} by 1, if $pk = A$;
- the A processor in dimension $N-1$ whose index can be obtained by incrementing its i_{pd} by 1, if $pk = B$; or
- the A processor in dimension N whose index can be obtained by decrementing its i_{pd-1}, if $pk = Z$.

This is because the A, B, and Z processors in dimension N are connected in the fixed pattern $A_N \to B_N$ and $Z_N \to A_N$.

When $pd < N$ and pk is A, $P_{i_1,\ldots,i_N}$ needs to first determine the direction dir_{pd+1} of the partial Hamiltonian path on its $(pd+1)$-D submesh by computing the parities of its coordinates in dimensions from $pd+2$ to N as follows:

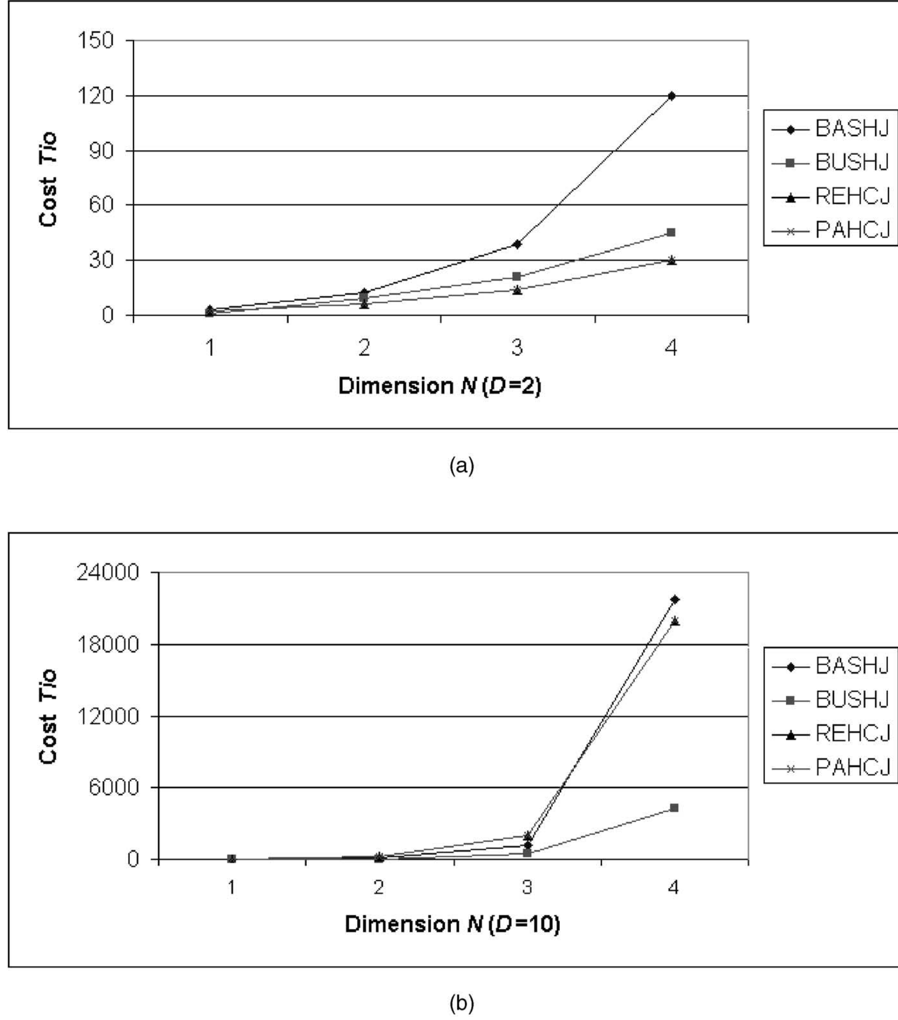


Fig. 10. Cost comparison of disk I/O operations.

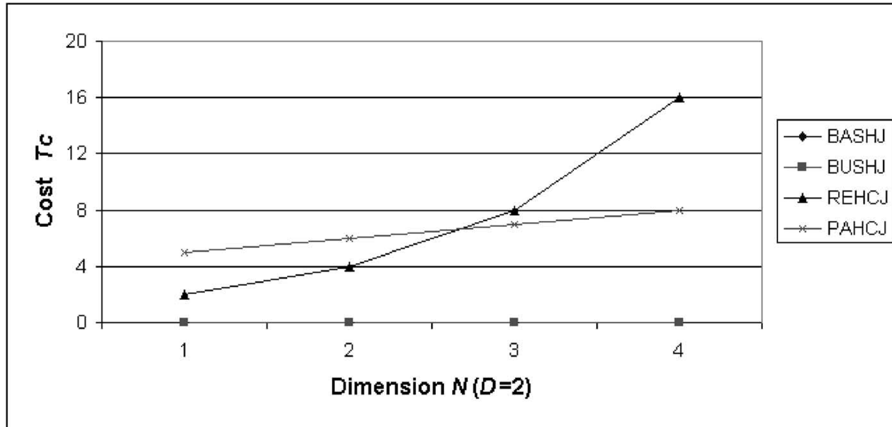
$$dir_{pd+1} = \begin{cases} \text{forward} & \text{if } \left(\sum_{j=pd+2}^{N} i_j\right) \text{ is even} \\ \text{backward} & \text{if } \left(\sum_{j=pd+2}^{N} i_j\right) \text{ is odd.} \end{cases} \quad (\text{Def}-4)$$

This method of computing dir_{pd+1} is based on the construction of the Hamiltonian cycle: The whole Hamiltonian cycle constructed by both REHCJ and PAHCJ algorithms is always *forward* and, hence, the partial Hamiltonian path on the *N*-D mesh is also *forward*. For a *k*-D submesh M_k^j which is the *j*th submesh of a $(k+1)$-D submesh M_{k+1} for $0 \leq k < N$, according to the recursive definitions (Def-2) and (Def-3), we know that the direction of the partial Hamiltonian path on M_k^j is the *same* as that of the partial Hamiltonian path on M_{k+1} if j is even or is the *reverse* if j is odd. Thus, by simply computing the parities of its coordinates, every processor can determine the direction of the partial Hamiltonian path on its *k*-D submesh for any k where $0 \leq k < N$. After determining dir_{pd+1}, $P_{i_1,\ldots,i_N}$ then obtains the index of its successor as follows:

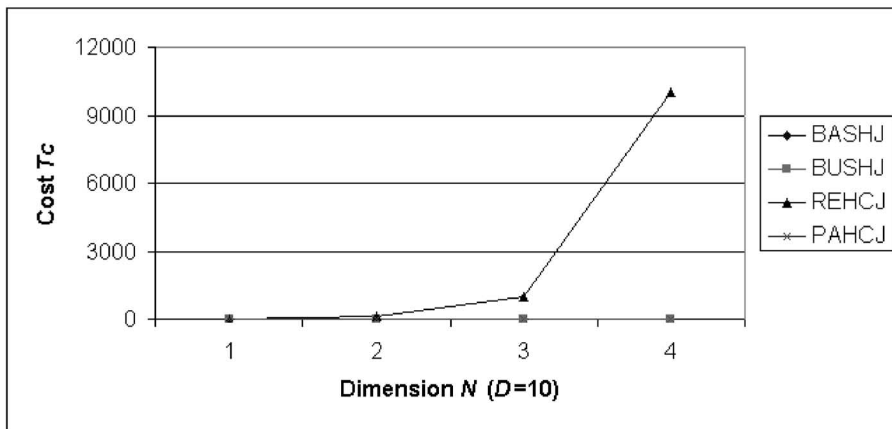
- If dir_{pd+1} is forward, according to (Def-2), the successor of $P_{i_1,\ldots,i_N}$ is either
  - the A processor in an adjacent *pd*-D submesh, whose index can be obtained by incrementing its i_{pd+1} by 1 if its $i_{pd+1} < D_{pd+1}$;
  - the B processor in its *pd*-D submesh, whose index can be obtained by incrementing its i_{pd} by 1 if its $i_{pd+1} = D_{pd+1}$ and D_{pd+1} is even; or
  - the Z processor in its *pd*-D submesh, whose index can be obtained by incrementing its i_{pd-1} by 1 if its $i_{pd+1} = D_{pd+1}$ and D_{pd+1} is odd.
- If dir_{pd+1} is backward, according to (Def-3), the successor of $P_{i_1,\ldots,i_N}$ is the A processor in an adjacent *pd*-D submesh, whose index can be obtained by decrementing its i_{pd+1} by 1.

For a given processor, let $INC\,(k)$ and $DEC\,(k)$ be two functions, which increment or decrement its *k*th coordinate respectively and return the resulting index. Hence, the operations for obtaining the index of the successor of an A processor can be presented by the following procedure:

function A-Successor $((i_1,\ldots,i_N), pd)$
  Input: The index $(i_1,\ldots,i_N)$ of a processor whose prime kind is A in dimension pd
  Output: The index of the successor of the processor
begin
  if $pd = N$ **then return** INC(pd)
  elseif dir_{pd+1} is forward **then**



(a)



(b)

Fig. 11. Cost comparison of memory computation.

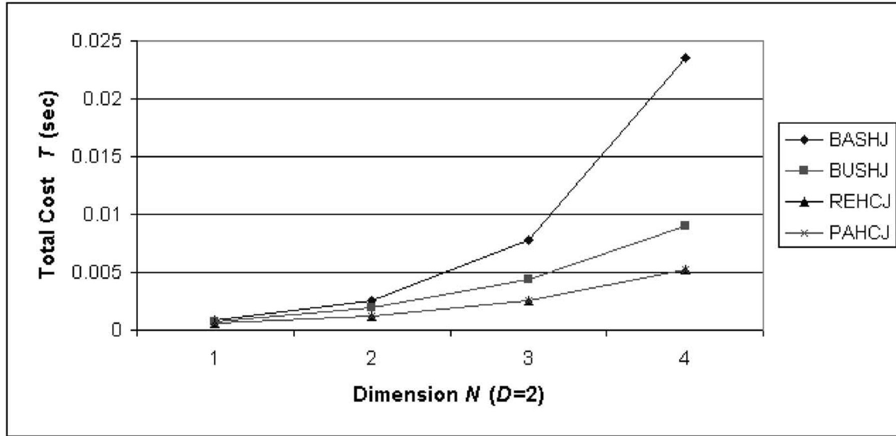
if $(i_{pd+1} = D_{pd+1})$ **and** (D_{pd+1} is even) **then**
  return INC(pd)
elseif $(i_{pd+1} = D_{pd+1})$ **and** (D_{pd+1} is odd) **then**
  return INC($pd - 1$)
else return INC($pd + 1$)
else return DEC($pd + 1$)
end if
end

When $pd < N$ and pk is either B or Z, $P_{i_1,\ldots,i_N}$ also first determines the direction of a partial Hamiltonian path like an A processor, but the dimension of its submesh, where the direction dir_{pd} of partial Hamiltonian path is determined, is pd instead of $pd + 1$. It then obtains the index of its successor according to (Def-2) and (Def-3). Thus, we have the following two functions which return the indices of the successors of B and Z processors respectively:

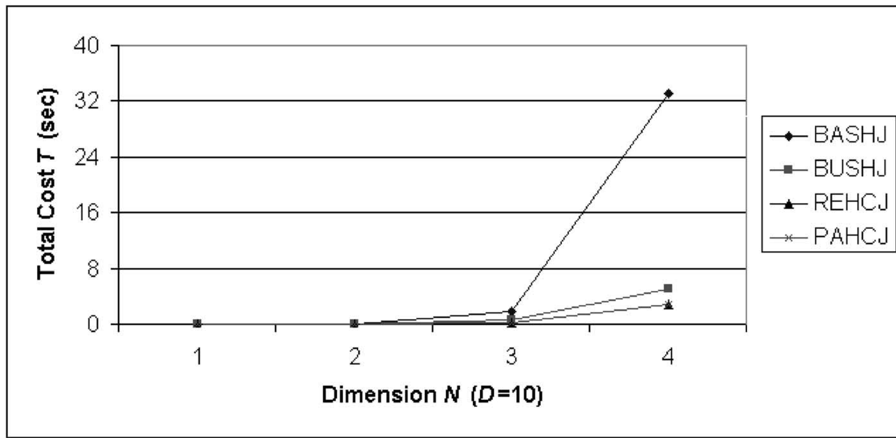
function B-Successor $((i_1, \ldots, i_N), pd)$
Input: The index $(i_1, \ldots, i_N)$ of a B processor, and its prime dimension pd and prime kind pk
Output: The index of successor of the processor
begin
if $pd = N$ **then**
  if $D_N > 2$ **then return** INC(pd) **else return** INC($pd - 1$)
elseif dir_{pd+1} is forward **then**
  if $D_{pd} > 2$ **then return** INC($pd - 1$) **else return** INC(pd)
else return DEC($pd + 1$)
endif
end

function Z-Successor $((i_1, \ldots, i_N), pd)$
Input: The index $(i_1, \ldots, i_N)$ of a Z processor and its prime dimension pd and prime kind pk
Output: The index of successor of the processor
begin
if $pd = N$ **then return** DEC($pd - 1$)
elseif dir_{pd} is forward **then return** INC($pd + 1$)
else
  if $pd = 1$ **then return** DEC(pd)
  elseif $D_{pd} = 2$ **then return** INC(pd) **else** **return**INC($pd - 1$)
endif
end

In summary, every processor in the even-sized mesh applies the following function *Successor* to obtain the index of its successor simultaneously:



(a)



(b)

Fig. 12. Total cost comparison.

function Successor $((i_1, \ldots, i_N))$
  Input: The index $(i_1, \ldots, i_N)$ of a processor
  Output: The index of successor of the processor
begin
  find-prime $((i_1, \ldots, i_N), pd, pk)$;
  if $pk = A$ **then return** *A*-Successor $((i_1, \ldots, i_N), pd)$
  else if $pk = B$ **then return** *B*-Successor $((i_1, \ldots, i_N), pd)$
  else return *Z*-Successor $((i_1, \ldots, i_N), pd)$
end

Example 8. In Fig. 7, processor $P_{1,1,3}$ first determines its $pd = 2$ and $pk = A$, then finds that its dir_3 is forward, and finally obtains its successor's index $(1, 1, 4)$. Processor $P_{1,2,3}$ first determines its $pd = 2$ and $pk = B$, then finds that its dir_2 is backward, and finally obtains its successor's index $(1, 2, 2)$. Processor $P_{2,1,3}$ first determines its $pd = 2$ and $pk = Z$, then finds that its dir_2 is backward, and finally obtains its successor's index $(1, 3, 3)$.

7.2 Analysis

Each processor needs at most N comparisons to determine its pd and pk in function find-prime, and at most four comparisons to obtain its successor's index without any data transmissions. Thus, the total cost T_{pahcj} of the PAHCJ algorithm is the sum of the costs for finding the successor of each processor and for the permutation, that is

$$\begin{aligned} T_{pahcj}(R, S) &= T_c \times (N + 4) + T_{perm} \\ &= T_t \times 2N(p-1)\frac{B_S}{p} + T_{lj}(R/p, S/p) \times p \\ &\quad + T_{io} \times 2(p-1)\frac{B_S}{p} + T_c \times (N + 4). \end{aligned} \tag{6}$$

Therefore, the cost T_{pahcj} of the PAHCJ algorithm is less than the cost T_{rehcj} of the REHCJ algorithm due to the improved efficiency for construction of the Hamiltonian cycle.

8 Evaluation Results

The detailed analysis for each of the four algorithms has been performed in previous sections. We summarize the results in Table 1, where $|Subset(S)|$ is the maximum number of S subsets to be stored in a processor.

In this section, we examine some evaluation results of the four algorithms and compare the algorithms against four major cost factors listed in Table 1, namely, T_t, T_{io}, T_c, and T_{lj}. To validate our analytical comparison, we use the same

values of system parameters that have used in [4], [5], as summarized in Table 2.

To simplify the comparisons, we assume that mesh M has the same degree on all dimensions, that is, $D_1 = D_2 = \ldots = D_N = D$. We further assume that each processor has the fixed number of R and S tuples regardless the number of processors p in M.

8.1 Cost Comparison of Data Transfer T_t

Fig. 8 shows the data transfer costs of all four algorithms, where $D = 2$ and $D = 10$, respectively. It is clear from the figure that, the BUSHJ algorithm always requires the least data transfer costs, while BASHJ require more T_t when p increases.

8.2 Cost Comparison of Local Join Operations T_{lj}

Fig. 9 shows that the BASHJ algorithm requires most local join operations. The two permutation algorithms outperforms the two shift algorithms because they require much fewer local join operations.

8.3 Cost Comparison of Disk I/O Operations T_{io}

Fig. 10 shows that the BASHJ algorithm requires most disk I/O operations. The permutation algorithms requires fewer disk I/O operations when D is small.

8.4 Cost Comparison of Memory Computations T_c

Fig. 11 shows that REHCJ requires more local computation cost for constructing a Hamiltonian cycle than PAHCJ.

8.5 Total Cost Comparison

Fig. 12 shows that the total execution execution time is greatly influenced by the local join operations, and both permutation-based algorithms require fewer local join operations than others.

9 Concluding Remarks

In this paper, we have presented four parallel algorithms to efficiently compute the range-joins operation on an N-D mesh. All algorithms use the permutation-based approach in which all the subsets of both relations are sorted and each subset of S is then permuted to every processor in turn, where it is joined with the local subset of R at that processor. Two data permutation approaches—the shifting and Hamiltonian-cycle approaches—have been developed, each being used by two join algorithms. The shifting approach minimizes the communication costs and can be applied to a system with either large or limited storage capability, while the Hamiltonian cycle approach minimizes the cost for the local join operations as well as the storage requirement.

Several conclusions can be drawn. First, BUSHJ achieves better parallelism performance than BASHJ by storing a large number of buffered subsets of S in each processor, as trade-offs between time and storage. Depending on which resource is more valuable, one approach or the other should be used. Second, the performance of the PAHCJ algorithm is obviously better than the REHCJ algorithm because it constructs the Hamiltonian cycle in parallel and is hence more efficient. Our goal in presenting the REHCJ algorithm is to provide a starting point to understand the more complicated PAHCJ algorithm. Third, both Hamiltonian-cycle algorithms require less storage and fewer local join operations but more data movements than the BUSHJ. Thus, we recommend the use of the shifting algorithms only if the communication time for transferring one data block between two neighbor processors is greater than the time for a local range-join operation. Otherwise, the Hamiltonian-cycle algorithms should be used in order to achieve the best performance.

It is worthwhile to note that, as the range-join operation is the generalization of the conventional equi-join and band-join operations, all four proposed range-join algorithms can be used to compute equi-join and band-join operations. More importantly, all proposed algorithms are general methods for data permutation on an N-D mesh and can be employed for solving any problems that involve data permutation.

Future research tasks are to implement the proposed algorithm on a suitable parallel machine for further performance evaluation, and to develop efficient parallel algorithms on other parallel computer architectures and for other database operations.

References

[1] S.G. Akl, *The Design and Analysis of Parallel Algorithms.* Orlando, Fl.: Academic Press, 1989.

[2] S.D. Chen, H. Shen, and R.W. Topor, "Efficient Parallel Permutation-Based Range-Join Algorithms on Mesh-Connected Computers," Technical Report CIT-94-19, CIT, Griffith Univ., Australia, Aug. 1994.

[3] S.D. Chen, H. Shen, and R.W. Topor, "An Improved Hash-Based Join Algorithm in the Presence of Double Skew on a Hypercube Computer," *Proc. 17th Australian Computer Science Conf.,* Jan. 1994.

[4] S.D. Chen, H. Shen, and R.W. Topor, "Permutation-Based Parallel Range-Join Algorithm on N-Dimensional Torus Computers," *Information Processing Letters,* vol. 52, no. 10, pp. 35-38, Oct. 1994.

[5] S.D. Chen, H. Shen, and R.W. Topor, "Efficient Parallel Permutation-Based Range-Join Algorithms on Mesh-Connected Computers," *Proc. 1995 Asian Computing Science Conf.,* pp. 225-238, Dec. 1995.

[6] D.J. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," *Comm. ACM,* vol. 35, no. 6, pp. 85-98, 1992.

[7] D.J. DeWitt, J.F. Naughton, and D.A. Schneider, "An Evaluation of Non-Equijoin Algorithms," *Proc. 17th Conf. Very Large Databases (VLDB),* pp. 443-452, Sept. 1991.

[8] Intel Corporation. *Intel Corporation Literature.* Nov. 1991.

[9] H. Jhang, "Performance Comparison of Join on Hypercube and Mesh" *Proc. 1992 ACM Computer Science Conf.,* pp. 243-250, 1992.

[10] M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash: A New Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)," *Proc. 16th Conf. Very Large Databases (VLDB),* pp. 210-221, 1990.

[11] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays Trees Hyper-Cubes.* San Mateo, Calif.: Morgan Kaufmann, 1992.

[12] D. Schneider and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," *ACM SIGMOD Record,* vol. 18, no. 2, pp. 110-121, June 1989.

[13] H. Shen, "An Improved Selection-Based Parallel Range-Join Algorithm in Hypercubes," *Proc. 20th EUROMICRO Conf.,* pp. 65-72, Sept. 1994.

[14] H. Shen, "Efficient Parallel k-Set Chain Range-Join in Hypercubes." *Computer J.,* vol. 38, no. 3, pp. 217-225, 1995.

[15] H. Shen, "An Efficient Permutation-Based Parallel Algorithm for Range-Join in Hypercubes," *Parallel Computing,* vol. 21, pp. 303-313, 1995.

[16] H. Shen, “Parallel k-Set Mutual Range-Join in Hypercubes,” *Microprocessing and Microprogramming,* vol. 41, no. 7, pp. 443-448, 1995.
[17] M. Stonebraker, “The Case for Shared Nothing,” *Database Eng.,* vol. 9, no. 1, pp. 4-9, 1986.
[18] J.D. Ullman, *Principles of Database and Knowledge Base Systems,* volume 2. Computer Science Press, 1989.
[19] C.B. Walton, A.G. Dale, and R.M. Jenevein, “A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins,” *Proc. 17th Conf. Very Large Databases (VLDB),* pp. 537-48, Sept. 1991.

Shao Dong Chen received the BSc degree from Shengzheng University, China, BInf (Hons) and the PhD degree from Griffith University, Australia. He was an associate lecturer at Queensland University of Technology and is currently working at Hutchison Telecom HK Ltd. as a senior manager responsible for managing all 3G IT projects. His research interest is in parallel and distributed computing.

Hong Shen received the BEng degree from Beijing University of Science and Technology, the MEng degree from the University of Science and Technology of China, and the PhLic and PhD degrees from Abo Academi University, Finland, all in computer science. He is a professor in the Graduate School of Information Science, Japan Advanced Institute of Science and Technology. Previously, he was a professor of computer science at Griffith University, Australia. He has published more than 130 technical papers on algorithms, parallel and distributed computing, interconnection networks, parallel databases and data mining, multimedia systems, and networking. He has served as an editor for *Parallel and Distributed Computing Practice*, as an associate editor for the *International Journal of Parallel and Distributed Systems and Networks*, an editorial board member for the *Journal of Parallel Algorithms and Applications*, the *International Journal of Computer Mathematics*, and the *Journal of Supercomputing*, and chair/committee member of various international conferences.

Rodney Topor received the BSc degree in mathematics from Monash University, Australia, and the PhD degree in artificail intelligence from Edinburgh University, United Kingdom. He has served on the faculties of Monash University and the University of Molbourne. He is currently a professor of computer science at Griffith University, Brisbane, Australia, where he served as Head of School from 1994 to 1997. He has been a sabbatical visitor at Imperial College, London, the University of Bristol, Standford University, and INRIA, Rocqencourt. He is on the editorial board of the Journal of Intelligent Information Systems. His research interests include declarative interfaces to databases, parallel algorithms for database query evaluation, and data mining. He has more than 50 papers in these and other areas. He is a member of the IEEE Computer Society.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.