Efficient Permutation-Based Range-Join Algorithms on N-Dimensional Meshes Using Data-Shifting (Preliminary version)

Shao Dong Chen Hong Shen Rodney Topor

School of Computing and Information Technology
Griffith University
Nathan, Queensland 4111, AUSTRALIA
Email: {schen,hong,rwt}cit.gu.edu.au

Abstract

In this paper, we present two efficient parallel algorithms for computing a non-equijoin, range-join*, of two relations on N-dimensional mesh-connected computers. The proposed algorithms uses the data-shifting approach to effectively permute every sorted subset of relation S to each processor in turn recursively in dimensions from low to high, where it is joined with the local subset of relation R.*

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 for future database systems [5]. 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 [2, 9, 11]. In this research, we generalize the conventional equijoin and band-join operations [6] to *range-join* operations, and design 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 $e_1 \leq |r.A - s.B| \leq e_2$ [12]. 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 for "finding all customers whose account balance differs 100 to 1000 dollars from that of some customer" requires 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 [11]. 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 [6] because the conventional hash functions (e.g. modulo-division, folding, radix-transformation, and mid-square methods) will inherently destroy the ordering of the tuples. In contrast, permutation-based join algorithms, which is an efficient implementation of parallel nested-loops join algorithms, have been shown to be effective for computing range-joins on hypercube computers [12] and torus computers [3]. Moreover, unlike most of hash-based join algorithms which are vulnerable to data skew and will result in an unacceptable performance for extremely 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 [4].

In this paper, we present an effective approach for efficiently permuting all subsets of S. Our *data-shifting* approach permutes the data recursively from lower dimensions to higher dimensions. It can be applied to meshes with different storage capacity which results in two different data permutation join algorithms. The *Basic Data-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 due to the low parallelism. Conversely, the *Buffered 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.

Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS '01)
1530-2075/01 $10.00 © 2001 **IEEE**

2 N-Dimensional Meshes

Meshes are an important class of parallel interconnection networks which have been well studied in the literature [10]. In parallel database design, mesh-connected parallel computers are characterized by the shared-nothing architecture [13]. There are several commercially available mesh-connected computers, such as the recent Intel Paragon XP/S [7] whose processors are connected by meshes instead of hypercubes which were used in the earlier Intel iPSC/860.

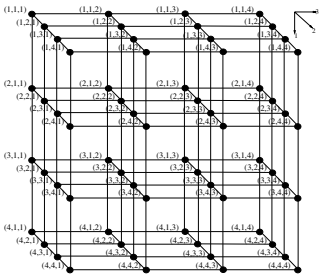

Figure 1. A $4 \times 4 \times 4$ mesh

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 $(j+1)$-th submesh with an edge in dimension N, such that their indices differ by one in the N-th dimension, where $1 \leq j < D_N$.

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 any 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 submesh by M_{k-1}^j, where $1 \leq j \leq D_k$.

Example 1 The 3-D mesh in Figure 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, [], 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 a 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. □

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 [8] when a high level comparison between meshes and hypercubes is presented, but the author of [8] doesn't present these two algorithms in detail and doesn't demonstrate how to implement them. In this paper, we present and analyze new permutation-based range-join algorithms on N-dimensional meshes using data-shifting.

3 Analytical Model and Permutation-Based Join

We adopt the analytical model used in [2] 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 don't consider the join-product skew [15] 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, namely I/O, communication and computational costs. The I/O costs are required to read/write data from/to the disks, while the communication costs are required to transfer data between different processors across an interconnection network. The computational costs are required for the operations which are performed in 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 don't consider the 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 the algorithms are 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;

Permutation-based join 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, permutation-based join computes 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 [4]. Thus, locally sorting the initial subsets in each processor only once 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

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

In addition, 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 for one sorted subset of R and one sorted subset of S is realized by a sequential sort-merge range-join algorithm [1], which is based on the standard sort-merge join algorithm [14] 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 result 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. In the following section we present a permutation algorithm based on data-shifting.

4 Basic Data-Shifting Join

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 like pulsing water through a pipe between its two ends in turn, as suggested in Figure 2. Thus, it consists of two steps, each with $D_1 - 1$

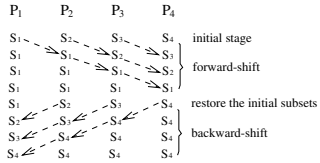

Figure 2. Permuting data in a linear array

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 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 in each processor exactly once.

This data-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:

BASHJ (k, L):

2. All processors permute the subsets simultaneously on all $(k-1)$-D submeshes, each in dimensions from 0 to $k-1$ recursively.
3. Every processor $P_{i_1,\ldots,i_N}$ copies its current subset $S_{i_1,\ldots,i_N}$ to buffered subset $S^k_{i_1,\ldots,i_N}$ temporarily, where $S^k_{i_1,\ldots,i_N}$ is stored in the local memory first until the local memory is exhausted, then is stored in the local disk.
4. 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 $k-1$.
5. Every processor $P_{i_1,\ldots,i_N}$ restores $S^k_{i_1,\ldots,i_N}$ to be $S_{i_1,\ldots,i_N}$.
6. 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 $k-1$.
7. Every processor $P_{i_1,\ldots,i_N}$ restores $S^k_{i_1,\ldots,i_N}$ to be $S_{i_1,\ldots,i_N}$.

Thus, a subset $S_{i_1,\ldots,i_N}$ is backed up once in Step 2 and restored twice in Steps 4 and 6, one for performing backward-shift, and one for permuting in the higher dimension $k+1$. During the i-th iteration of forward-shift, every processor in each M^j_{k-1} $(j = i+1,\ldots,D_k)$ 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 i-th iteration of backward-shift, every processor in M^j_{k-1} $(j = 1\ldots, D_k - i)$ 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.

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 Figure 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 will be grayed. □

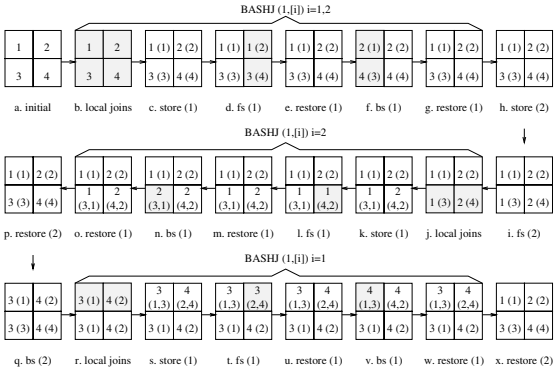

Figure 3. Permuting S on a 2×2 mesh

Analysis

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

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 is $\max\{0, M_f - (k+1)B_S/p\}$ subsets which requires 3 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 relation

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

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 recurrent relation and have the total cost T_{bashj} of the whole BASHJ

algorithm in dimension N as follows:

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

5. Buffered Data-Shifting Join

Description

From the preceding analysis, we know that the parallelism of the previous data-shifting algorithm does not appear to be very attractive. In particular, during the j-th 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 don't 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 j-th iteration of backward-shift, all processors in submeshes $M_{k-1}^{j+1}, \ldots, M_{k-1}^{D_k}$ are idle.

To obtain better parallelism, we propose a Buffered Data-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 the 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 dimension from 0 to $k-1$ as the first step of the BASHJ, but it allows every 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 t-th element, denoted by $Q_{i_1,\ldots,i_N}[t]$, is the t-th subset appended into. With $Q_{i_1,\ldots,i_N}$, $P_{i_1,\ldots,i_N}$ doesn't 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 t-th step for $1 \le t \le D_1 \times \cdots \times D_{k-1}$, the t-th subset in $Q_{i_1,\ldots,i_N}$ is transferred in forward- and backward-shift in turn as in the previous data-shifting algorithm, but in each iteration, the recursive call in the previous algorithm is replaced by a local join operation for the arriving subset and an operation for storing this arriving subset.

Example 3 We consider the same problem in Example 2, and solve it by using the BUSHJ algorithm now, as illustrated in Figure 4. We use the same notation and representations which are used in Example 2. □

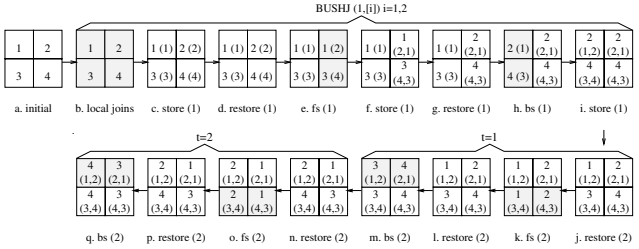

Figure 4. BUSHJ($2, []$): Permuting S on a 2×2 mesh

Analysis

As the BASHJ algorithm, the BUSHJ algorithm can permute every subset of S to every processor exactly once, and hence it is also correct. However, unlike the BASHJ algorithm which requires each processor to store at most $N+1$ subsets during the permutation, the BUSHJ algorithm requires each processor to store all subsets which are initially stored the $(N-1)$-D submesh where the processor is in. That is, each processor needs to store $(D_1 \times \cdots \times D_{N-1})$ subsets during the permutation, and hence more disk I/O operations are required to shuffle the overflow subsets of S in and out of memory several times. On the other hand, the parallelism of the BUSHJ is higher than that of the BASHJ, and it requires fewer data transmissions 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

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

The above equation clearly indicates that we don't need the cost of 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:

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

Note that, although the number of disk I/O operations of the BUSHJ (which is 9) is fewer than that of the BASHJ (which is 12) due to the small D_1 and D_2 in this example, the BUSHJ algorithm normally requires much more disk I/O operations than the BASHJ algorithm, but much fewer parallel data transmissions and local join operations.

6 Concluding Remarks

In this paper, we have presented two parallel algorithms to efficiently compute the range-joins on an N-D mesh. Both algorithms use the data-shifting 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. This approach minimizes the communication costs and can be applied to a system with either large or limited storage capability.

It is worthwhile to note that, as the range-join operation is the generalization of the conventional equi-join and band-join operations, all the 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 applied for solving any problem whose main communication pattern is 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. 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 University, Australia, Aug. 1994.

[2] 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. In *Proc. of the 17th Australia Computer Science Conference*, Christchurch, New Zealand, Jan. 1994.

[3] S. D. Chen, H. Shen, and R. W. Topor. Permutation-based parallel range-join algorithm on N-dimensional torus computers. *Information Processing Letters*, 52(10):35–8, Oct. 1994.

[4] S. D. Chen, H. Shen, and R. W. Topor. Efficient parallel permutation-based range-join algorithms on mesh-connected computers. In *Proc. of the 1995 Asian Computing Science Conference*, pages 225–38, Pathumthani, Thailand, Dec. 1995. Springer-Verlag.

[5] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communication of ACM*, 35(6):85–98, 1992.

[6] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of Non-Equijoin algorithms. In *Proc. of the 17th VLDB*, Barcelona, Spain, Sept. 1991.

[7] Intel Corporation. Intel Corporation literature, Nov. 1991.

[8] H. Jhang. Performance comparison of join on hypercube and mesh. In *1992 ACM Computer Science Conference*, pages 243–50, Kansas City, MO, USA, 1992.

[9] 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). In *Proc. of the 16th VLDB*, pages 210–221, Brisbane, Australia, 1990.

[10] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays Trees Hypercubes*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[11] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *ACM SIGMOD Record*, 18(2):110–121, June 1989.

[12] H. Shen. An improved selection-based parallel range-join algorithm in hypercubes. In *Proc. of the 20th EUROMICRO Conference*, pages 65–72, Liverpool, UK, Sept. 1994.

[13] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

[14] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 2. Computer Science Press, 1989.

[15] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proc. of the 17th VLDB*, Barcelona, Spain, Sept. 1991.