Parallel Algorithms for Database Operations and a Database Operation for Parallel Algorithms

Rajeev Raman* Uzi Vishkin†

Abstract

This paper establishes some significant links between two areas: (i) relational parallel database systems; and (ii) the design and analysis of parallel algorithms.

The paper begins with a fundamental but very simple observation: implementing a JOIN *operation in the context of relational parallel database systems is at least as expensive as implementing an arbitrary PRAM computation. Thus, the efficiency with which a given parallel computer can support a parallel relational database where* JOINs *are fairly frequent is strongly related to the efficiency with which that computer can support the PRAM as one of its programmer's models.*

The main technical contribution is an efficient parallel algorithm for the JOIN *operation on a model where, in order to use the available bandwidth effectively, communication has to be performed in large blocks.* [1]

1 Introduction

A key performance bottleneck for various database applications on serial computers has been high latency and low bandwidth while accessing slow memories, such as disks. The advent of multiprocessors, specifically those based on inexpensive processors with local memories ("shared-nothing" architectures), has mitigated the above bottleneck by providing increased bandwidth between processing elements and memories. This has already led [6] to pronounce parallel database systems a success. In this paper we examine some consequences of this development for the parallel processing community at large. One obvious consequence is the need to develop parallel algorithmic methods in order to further improve the performance of parallel database systems on available shared-nothing architectures; in Sections 2 and 3 we take a step in this direction.

More significantly, the emergence of parallel machines supporting (parallel) databases signals the entry of a new "superpower" into the field of parallel computing—the multibillion-dollar-per-year database industry. This industry will clearly be motivated to further improve parallel computers so that they can more effectively support parallel database systems, which may have a profound effect on the field of parallel computing in general. Our discussion below leads to the conclusion that *communication-intensiveness* (sometimes called communication *throughput* or *bandwidth*) is a likely bottleneck for implementing database systems on parallel computers, much in the same way that it is a bottleneck for implementing a PRAM programmer's model on such computers. Having the database industry join the coalition of parties interested in more communication-intensive parallel computers may provide the necessary impetus for obtaining a further increase in the communication-intensiveness of parallel computers.

In Section 1.1, we establish a very simple, yet fundamental, connection between two research activities which so far have been considered rather separate:

- Relational databases, and in particular the JOIN operation, and their implementation on parallel computer systems. For more on this issue see [6].
- Emulating the PRAM programmer's model for general-purpose parallel computing on parallel computing systems, i.e., implementing a "virtual PRAM" on such systems. For more on the role that the PRAM could play as a programmer's model for parallel computing, see the position paper [11].

*Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK. Part of this work was done while the author was at UMIACS. Partially supported by NSF grant CCR-9111348.

†University of Maryland Institute for Advanced Computer Studies (UMIACS) and Electrical Engineering Department, College Park, MD 20742, USA, and Tel Aviv University, Tel Aviv, Israel. Partially supported by NSF grant CCR-9111348.

[1] To be presented at the 9th IEEE-IPPS, April 1995.

Why is this connection surprising?

According to [6], the success of parallel database systems refutes a 1983 thesis, due to Boral and DeWitt, which predicted the demise of (special-purpose) database machines. However, the strong ties between parallel database systems and general-purpose parallel computing, as argued in the current paper, imply that a parallel machine that efficiently supports parallel database systems is actually likely to support general-purpose parallel computation, and not just parallel database computation, and therefore may not be considered a special-purpose machine at all. In other words, it is a serious possibility that the thesis of Boral and DeWitt may not be refutable by shared-nothing parallel database machines. (Note that following the concrete observation in the subsection below, we explain why improved efficiency of the JOIN operation, as well as efficient support of PRAM algorithms, both depend on improved communication throughput of parallel machines.)

Why is the connection actually not too surprising?

While hash partitioning of data appears to be a mainstream technique for performing JOINs and other database operations on shared-nothing architectures, hash partitioning of memory addresses among the local memories of parallel processors is an integral part of *general* methods for emulating a PRAM programmer's model on similar shared-nothing architectures, as suggested in [9]. Also, it is well known that relational databases can be thought of as associative memories.

We directly proceed to making the concrete technical observation that efficiently performing a JOIN operation is at least as hard as efficiently emulating a general PRAM programmer's model, in the next subsection, and then overview the main technical contributions of the paper.

1.1 JOIN versus PRAMs

The JOIN operation: Given k sets (domains) $D_1, \ldots, D_k$ a *relation* is a subset of their Cartesian product $D_1 \times \ldots \times D_k$. Each member of a relation is called a *tuple* and is denoted as $x = (x_1, \ldots, x_k)$, with $x_i \in D_i$ for $i = 1, \ldots, k$. The sets D_i are called the *attributes* of the relation. Given two relations $R \subseteq D_1 \times \ldots \times D_k$ and $S \subseteq D_k \times \ldots \times D_{k+l}$, the JOIN of R and S, denoted by R JOIN S is a maximal set of tuples $t \in D_1 \times \ldots \times D_{k+l}$ such that $(t_1, \ldots, t_k) \in R$ and $(t_k, \ldots, t_{k+l}) \in S$. Thus the JOIN is defined for a pair of relations that have a common attribute, called the *join attribute* and consists of all pairs of tuples, one from each of the relations, which have the same value of the join attribute. The output is in the form of a relation obtained by concatenating the tuples of each pair in the JOIN and deleting one of the two copies of the join attribute.

Remark: What we have defined above is more properly called the *natural* join of R and S. In the more general definition of a JOIN, two tuples can be said to match if their join attribute values satisfy any given binary predicate and not necessarily equality.

The PRAM model: In the PRAM programmer's model, p synchronous processors have access for both read and write purposes to a shared memory. Say that they can read simultaneously from the same memory location but only one processor can write simultaneously into one. (We omit here discussion of those PRAM variants where simultaneous writes into the same memory location are allowed; our results would extend into such variants as well though.) The efficiency of a PRAM algorithm is measured in terms of its *work,* or the total number of operations it performs, and in terms of its parallel running time. Given two PRAM algorithms, the one that performs less work is considered better, and if they perform the same amount of work, then the one with lower running time is considered better. A PRAM algorithm for a problem is called *work-optimal* if its work is of the same order of magnitude as the sequential running time of the best algorithm for that problem.

The PRAM-by-JOIN emulation lemma: A parallel relational database system that efficiently (i.e., work-optimally) supports JOINs and updates to relations can also work-optimally emulate the execution of a PRAM algorithm.

Proof of the PRAM-by-JOIN emulation lemma: The contents of the memory of a PRAM can be viewed as a relation M, consisting of the tuples $(i, contents(i))$ for every memory location i. Consider a PRAM step where each of the p processors is trying to read a shared memory location, specifically, suppose that processor i attempts to read location $loc(i)$, for $i = 1, \ldots, p$. Such a step can be simulated by a JOIN operation as follows: first we produce a new relation consisting of the p tuples $(1, loc(1)), (2, loc(2)), \ldots, (p, loc(p))$. The JOIN of this relation with the relation M consists of the tuples $(1, loc(1), contents(loc(1))), \ldots, (p, loc(p), contents(loc(p)))$, thus completing the read.

Remark: The statement of the PRAM-by-JOIN emulation lemma and its proof above are a bit informal. To be precise: (i) We can assume that M is available for preprocessing, prior to knowing the actual memory requests; similarly, one of the two relations for the JOIN operation through which the PRAM memory access is resolved, is available for preprocessing prior to knowing the second relation. (ii) We should state how to deal with other PRAM operations than just reads; it is not difficult to observe that, in fact, the other operations (such as writes, or computation with respect to local variables) are easier to emulate.

1.2 Overview of contributions and their significance

Above, we observed that a JOIN operation is at least as hard as emulating an arbitrarily complex PRAM algorithm. In particular this leads to the following conclusion: *truly efficient support of the* JOIN *operation must be done by computers which are not less communication-intensive than computers that would support PRAM algorithms.* This conclusion is interesting since it adds new urgency to the call for increasing the communication intensiveness of parallel machines, this time for efficiently supporting JOIN, a key parallel database operation. Previously, a call for increase in communication-intensiveness of parallel systems came in the context of ease of parallel programming, as claimed in [2], and for solving efficiently several elementary non-numeric problems in parallel, as proven in [8].

The main technical contribution of this paper is derived in Section 3, where an algorithm for the JOIN operation is given and analyzed. We have not encountered a similar mathematical analysis of a JOIN algorithm in the literature; in addition, our algorithm has some new features that distinguish it from existing JOIN algorithms. The JOIN algorithm is developed on a new parallel computation model that suitably captures what appear to be the two most important constraints in today's parallel computers: (i) blocking (i.e., communication should be in large blocks in order to maximize utilization of bandwidth) and (ii) limited bandwidth. This model is obtained by suitably modifying the PRAM model, and can be viewed as a refinement of previously defined models. The Block PRAM model of [1] models blocking but assumes unlimited bandwidth; the LogP model [4] models bandwidth limitations but does not model blocking. For suitable choices of the parameters of our model we obtain either the Block PRAM or a PRAM-based alternative to the LogP model. The model is meant to provide a framework for the algorithms, which approximates some realistic situation; it is not claimed to a single "right" model.

In Section 2 we give algorithms on this model for several basic problems such as $h-h$ routing, prefix sums and integer sorting. The problem of $h{-}h$ routing deserves special attention due to its relevance to the problem of parallel hash JOINs (indeed existing JOIN algorithms implicitly use a solution to this problem). Here, a processor has up to h items, each of which is labelled with the name of some other processor to which it must be sent. The labelling is such that no more than h items are destined for any processor. A good algorithm for this problem would make use of the blocking facility in order to communicate items with a common destination together in a single step. We give an algorithm that uses close to an optimal number of messages even when the number of items to be sent is quite small (this is explained in greater detail later).

In the context of parallel JOIN algorithms we give a new solution to the problem of *skew*. It has been noted that using hashing as a partitioning method inevitably leads to load imbalance when the input has degeneracies (i.e. when many tuples have the same value of the join attribute). Also, since the JOIN of two relations may vary in size from zero to a number that is quadratic in the number of tuples in the input, it is desirable that a parallel JOIN algorithm should be *output-sensitive*, that is, the work done by the algorithm should be proportional to the size of the output (plus the size of the input). This requirement leads to the idea that the input should be partitioned in such a way that each processor computes an equal number of output tuples. It appears difficult to simultaneously avoid skew and to guarantee output-sensitivity using hashing alone. We therefore suggest an approach that uses hashing to compute *signatures* of join attribute values and then give a parallel-radix-sort-based approach to parallel JOINs that leads to an output-sensitive algorithm.

2 Model and Subroutines

2.1 Model

The model is a block-memory PRAM. There are p processors each with (unlimited) local memory. The shared memory consists of m blocks each consisting of b locations. We consider time as being globally divided into series of b steps each. In each series a processor may either compute using data in local memory or else read or write (but not both) part or all of a single block in shared memory. A processor may not access elements of more than one block of shared memory in each series. We intend that access to blocks should be considered EREW, i.e., only one processor may access a block at any time. The actual assumptions are described below, but before proceeding any further, we provide some motivation.

The model describes a situation where the data lies in the local memory of processors and is brought into shared memory only where the need arises to send it to other processors. The organization of shared memory is meant to model the two most important communication constraints: limited bandwidth and blocking. We will assume that $m \leq p$ and take $\lambda = \frac{p}{m}$. Firstly, since in one time step it is possible to perform p computation steps summed over all processors, but only m locations can be read from or written to memory, λ is the "communication gap", or the relative cost of communication versus computation. Note that $\lambda \geq 1$ by assumption, i.e., communication is at least as expensive as computation, and in fact we will assume normally that $\lambda \gg 1$, and that λ is an integer. Secondly, even a communication rate of m words per step can only be achieved if each processor that accesses shared memory reads b "useful" words in each series of b steps. This is similar to the Block PRAM [1].

Access to memory and other assumptions: For ease of exposition we will assume a CRCW access to memory, allowing both concurrent reads and concurrent writes. This is of course open to abuse, but we point out that in all our algorithms there is little or no memory contention at the block level. Our algorithms work equally well under the assumption that concurrent block accesses are serialized in some way.

For simplicity, we assume that the original input distribution will not prevent any of the results stated in this paper from holding. While it should be clear to the reader that this assumption may not always be valid, we still make it for convenience of presentation. To avoid misunderstanding: where not specified, our results hold subject to having a "reasonable" initial layout of data.

2.2 Prefix Sums

Now we show how to solve multiple independent prefix summation problems simultaneously in parallel. The input consists of (say) k lists of $n > p$ numbers each. The objective is to compute independently the prefix sums of each of these k lists. In the final output each processor has n/p consecutive prefix sums from each of the k lists.

The rationale for considering multiple prefix summation problems is simply that it takes as long (b time steps) to communicate one word as it does to communicate b words. Since (prefix) summation involves communication of greatly summarized information (e.g. the sum of several numbers) the available bandwidth is much better utilized by considering independent prefix summation problems. Furthermore, the ability to solve simultaneous prefix sums will be critical to the efficiency of the integer sorting algorithm described later.

Lemma 1 *We can compute b independent prefix sums, consisting of $n \geq p$ numbers each in $O(\frac{nb}{p} + b(\lambda + \log m))$ steps.*

Proof. We describe how to solve b independent summation problems, rather than prefix summation problems; extending the solution to the prefix summation case is quite standard.

First each processor computes the sum of n/p consecutive numbers from each of the b problems. This takes $O(nb/p)$ time and leaves each processor with b partial sums which have to be combined independently.

For this purpose, the processors are first divided into m groups of size $\lambda = p/m$. Each group is assigned one memory cell. Using this memory cell the processors within the group combine their partial sums in $O(b\lambda)$ time, by using this shared memory cell in a round-robin manner in order to communicate their numbers to a designated processor within the group.

Now we have m designated processors, each holding the b partial sums from its group. In order to combine these partial sums we use the standard PRAM summation tree algorithm, which can be executed in $O(b \log m)$ time on this model. The total execution time is therefore $O(\frac{nb}{p} + b(\lambda + \log m))$.

Since the input consists of nb numbers, optimal speedup is obtained when $nb/p \geq b(\lambda + \log m)$, or when $n \geq p(\lambda + \log m)$. ∎

2.3 $h - h$ routing

An instance of the $h - h$ routing problem is when each processor has up to h words, each of which is destined for another processor, such that each processor has at most h words destined for it. The objective is to move each word to its destination. Since (on average) communicating one word, per processor, takes λ time units, an optimal running time for this problem is $\Theta(h\lambda)$, and in order to do this it is necessary to block together words with a common destination.

By way of example, suppose that there are $p = 1000$ processors and that $b = 80$. Then suppose each processor has $h = 80,000$ words that it wants to transmit such that 80 words are destined for each of the other processors. Then it is possible to get by with each processor sending only 1000 messages of 80 words each. In general, so long as $h \geq b \cdot p$, one can get by with h/b messages rather than h. Now suppose that h is somewhat smaller, say $h = 8000$, while p and b remain the same. Now each processor has only 8 words to be sent to each other processor. At first glance it still appears that 1000 8-word messages will need to be sent; however we show if we send each word along a 3-hop route via intermediate destinations, we can get by with each processor sending only 300 messages. Specifically we show that so long as $h \geq b \cdot \sqrt{p}$ it is possible to get by with $3h/b$ messages rather than h. This result can be generalized to smaller powers of p.

We distinguish between *average-case* algorithms and *randomized* algorithms for this problem. The former work only under assumptions about the input distribution, whereas the latter work well on every input with high probability. A fundamental observation made by [10] is that one can obtain a good, and somewhat counter-intuitive, randomized algorithm from a good average-case algorithm by a two-phase routing scheme: first we route every word to a randomly chosen intermediate processor and then from the randomly chosen intermediate processor to the correct final destination. The good average-case algorithm can be used in each phase to good effect, since both phases are now truly random instances of the problem, and in fact the analysis of the second phase is symmetric to that of the first. The considerable price paid for this improvement however is that the routing takes twice as long. We should point out that several other papers have utilized this idea to come up with randomized routing algorithms. In particular, Lemma 2 is similar in spirit to an algorithm given by [7].

In what follows, an event which occurs with probability of the form $\geq 1 - 1/p^{\alpha}$ for some constant $\alpha > 0$, is referred to as occurring *with high probability,* and abbreviated as whp.

Lemma 2 *Any $h - h$ routing problem can be solved in $O(h\lambda)$ time whp provided $h \geq p(b + \log p)$.*

Proof. By the above discussion, we will describe only the first phase where each word is routed to an intermediate destination chosen uniformly and independently at random. Since the expected number of words sent from one processor to another is $h/p \geq \log p$, using standard methods it is possible to show that simultaneously for every pair of processors i, j, the number of words starting at i that are to be sent to j is $\Theta(h/p) = \Omega(b)$ whp. This corresponds to the first case of our example and the routing can be done using a straightforward algorithm (details omitted). ∎

This algorithm works only when n (the total number of words) is $\geq bp^2$. We now give a recursive algorithm that works for a larger range of values:

Lemma 3 *If $h \geq k(b + \log p)$, for some integer k any $h - h$ routing problem can be solved in $O(h\lambda \frac{\log p}{\log k})$ time whp.*

Proof. Without loss of generality we assume that k divides p. As before, it suffices to show how to route each word to an intermediate destination chosen uniformly and independently at random.

We divide the processors: (i) into p/k consecutively-numbered *groups* of size k, and also (ii) into k consecutively-numbered *segments* of size p/k. We will begin by routing words inside groups, in the following manner: each word starting in a particular group, whose random intermediate destination is in the j-th segment, is routed to the j-th processor in that group. By the random choice of destination each intra-group routing problem is a $h - h$ routing problem whp with $h = \Theta(k(b + \log p))$. By the above Lemma we can solve each such problem in $O(h\lambda)$ time whp.

Now all the words whose intermediate destination is in the i-th segment are located among the i-th processors of the different groups, for all i. We now proceed to move words to their correct segments. This is done by transferring *en bloc* all the words in the i-th processor of the j-th group to the j-th processor of the i-th segment. Since each transfer will involve the movement of $h = \Theta(b)$ pieces of data, this transfer can be done using the obvious algorithm in $O(h\lambda)$ steps.

Now it merely remains to recurse on each segment. There is a small subtlety — that the destination in the segment indeed should be chosen uniformly at random from among the destinations in the segment, but this is easily seen to be true by considering the initial random choice of intermediate destination as first choosing a segment at random and then choosing a processor within the segment at random. It is not difficult to show that the running time of the entire algorithm is essentially given by (i.e. probabilistic failures can be ignored):

$$T(p,h) = \begin{cases} T(p/k, h) + O(h\lambda) & \text{if } h < p(b + c\log p) \\ O(h\lambda) & \text{otherwise} \end{cases}$$

Solving this recurrence gives the claimed result. ∎

Specializing this result to the case when $k = p^\epsilon$ for some constant $\epsilon > 0$ we see that the $h-h$ routing problem can be solved in optimal $O(h\lambda)$ time when $n = ph = p^{1+\epsilon}(b + \log p)$ as claimed.

2.4 Stable Integer Sorting

Given are n keys $x_1, ..., x_n$ taken from a total ordered domain. In case two keys x_i and x_j are equal we extend the order relation to be $x_i < x_j$ if $i < j$. The stable integer sorting problem is to sort the n keys according to this extended order relation. We also require here that n/p keys end up at each processor.

Lemma 4 *If $n \geq (b + \log p)p^{1+\epsilon}$ then n integers in the range $1..n$ can be stably sorted in $O(n\lambda/p)$ time whp (and hence with $O(n\lambda)$ operations whp).*

Proof. The algorithm proceeds by $O(1)$ passes of radix sort with radix $r \approx n/p$. Each pass is an instance of the Cole-Vishkin prefix sums sort [3, Section 2.3] and can be implemented in our model with $O(n\lambda)$ work using Lemma 1 and Lemma 3. The details are omitted. ▮

3 An Output-sensitive Parallel JOIN Algorithm

In this section we will give an *output-sensitive* algorithm for the JOIN of two n-tuple relations. An output-sensitive algorithm performs work proportional to $n + S$, where S is the number of tuples in the resulting JOIN. We assume that the relations to be JOINed are already partitioned with n/p tuples per processor respectively. The usual methodology for performing a JOIN is to bring together all tuples from either relation that have the same value of the join attribute into the local memory of the same processor, in such a way that each processor has roughly equal numbers of input tuples from each relation.

Setting aside for the moment the problem that this may be impossible for degenerate inputs (the problem of *skew*), we note that in order to obtain an output-sensitive algorithm it is necessary to partition the tuples so that equal numbers of *output* tuples appear at each processor (i.e. load balancing should be done in terms of the output rather than the input). We illustrate this with an example. Supposing we have two relations with 10^7 tuples each and 100 processors, and furthermore that we have succeeded in partitioning the input tuples so that each processor has 10^5 tuples from each relation, such that all tuples with the same value of the join attribute are at the same processor. Even with this optimal partitioning, it is possible that all the processors except one may have no matches in their input tuples, while the remaining one may have the entire Cartesian product of its tuples in the JOIN, which is of size $10^5 \times 10^5 = 10^{10}$. In effect, the entire output will have to be computed by this processor, yielding virtually no speedup.

It appears that hashing alone as a partitioning method can neither deal effectively with skew, nor can it ensure an output-sensitive distribution of input tuples. The method we outline limits the use of hashing to computing "signatures", whereby each join attribute value is replaced by a small integer. Formally, for each join attribute value x in the input, we compute a *signature* which is a function $h(x)$ satisfying the following properties:

(a) The range of the function h should be the integers from $1..m$, where m is "small" relative to n, i.e., bounded by a polynomial in n.

(b) There should be *no collisions*, i.e., for all join attribute values x, y in the input, $h(x) = h(y) \Leftrightarrow x = y$.

Note that (b) implies that replacing the join attribute values by their signatures and performing the JOIN still leads to a correct output. We then suggest that the partitioning be done using a radix-sort-based method. Since m is small, radix sort will require only $O(n\lambda)$ operations. Note that if the join attribute values are already small integers then we can in fact dispense with hashing completely. The theorem we actually prove is as follows:

Theorem 5 *A JOIN of two n-tuple relations can be performed in $O(\lambda(n+S))$ operations whp (and $O(\lambda(n+S)/p)$ time whp) using p processors, where S is the number of tuples in the resulting JOIN, provided $n \geq (b + \log p)p^{1+\epsilon}$ for some fixed $\epsilon > 0$.*

Proof. We first describe how to compute the signatures. It suffices to choose h randomly from a 2-universal family of hash functions [5, pp. 229–231] mapping attribute values to the range $1..m$, for $m = n^3$. It is then immediate that the probability of there being no collisions is $1 - 1/n$, and we therefore assume that no collisions occur. We then replace the values of the join attribute by their signatures and sort the tuples of both relations with respect to their signatures using three passes of the algorithm of Lemma 4 using $O(n\lambda)$ work. (Using k-universal hash functions for some $k > 2$ would allow us to hash into a smaller range, and thereby reduce the number of passes of radix sort needed, at the expense of extra computation.)

Let A and B be the two relations and let $S = |A \text{ JOIN } B|$. We now compute the value S. For $i = 1, \ldots, m$, let A_i be the equivalence class of tuples in A, the signature of whose join attributes have value i, and define B_i analogously. Let $a_i = |A_i|$ and $b_i = |B_i|$. If the signatures satisfy property (b) above then clearly $S = \sum_{i=1}^{m} a_i b_i$. It is easy to compute S after sorting the signatures. We then repartition the data so that each processor computes $O(S/p)$ of the output tuples. This may involve replicating some tuples in one of the relations.

For the sake of exposition we describe an algorithm that is perhaps more complex to implement than necessary. The algorithm runs in two phases. In the first phase the output tuples corresponding to "small" equivalence classes, namely those where $a_i b_i$ is smaller than S/p. The small equivalence classes can clearly be partitioned so that all the tuples of each small equivalence class appear at the same processor and no processor has more than $O(S/p)$ tuples appearing at it; the details are left for the full paper.

In second phase deals with the "large" equivalence classes, namely those where $a_i b_i$ is larger than S/p. We partition the output tuples for each such equivalence class over two or more processors such that $\Theta(S/p)$ of the output tuples for this class appear at each processor. Let i be some equivalence class and assume without loss of generality that $a_i > b_i$ for this class. We partition the tuples of A_i over $l_i = \frac{a_i b_i}{(S/p)}$ processors and broadcast the tuples of B_i to each of these l_i processors. We only sketch here the intuition for proving that the cost of making duplicates is not too high. It is, quite simply, that the cost of sending b_i tuples of B_i to each processor is $O(\lambda b_i + b)$ time steps (since each communication step effectively costs λ computation steps, unless $b_i < b$, in which case the total cost of the communication is b steps). However, since there is at least one (distinct) tuple of A_i at the receiving processor, this processor will have at least b_i distinct tuples of the output. Thus there are $\Omega(b_i + S/p)$ tuples of the output at each processor and the cost of the broadcast can be "charged" to the output tuples at λ time steps per output tuple, and so the total operation cost of the broadcasting is bounded by $O(\lambda \sum_{i=1}^{p}(b_i + S/p)) = O(\lambda(n + S))$.

The total operation cost of the algorithm is therefore bounded by $O(\lambda(n + S))$. ∎

Acknowledgment Helpful comments by Michael Franklin are gratefully acknowledged.

References

[1] A. Aggarwal, A. K. Chandra and M. Snir. On communication latency in PRAM computations. In *1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 11–21, 1989.

[2] G. E. Blelloch, B. M. Maggs and G. L. Miller. The hidden cost of low bandwidth communication. In *Developing a Computer Science Agenda for High-Performance Computing*, U. Vishkin (editor), ACM Press, 1994, 158 pages.

[3] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, **70** (1986), pp, 32–53.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian and T. von Eicken. LogP: towards a realistic model of parallel computation. TR UCB/CS/92 71, University of California, Berkeley, 1992.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[6] D. DeWitt and J. Gray. Parallel database systems: the future of high-performance data base systems. *Comm. of the ACM*, 35:85–98, 1992.

[7] J. F. J*á*J*á* and K. W. Ryu. The block distributed memory model for shared memory multiprocessors. In *Proc. 8th IPPS*, 1994, 752–756.

[8] Y. Mansour, N. Nisan and U. Vishkin. Trade-offs between communication throughput and parallel time. In *Proc. 26th Annual ACM Symp. on Theory of Computing*, 1994, 372–381.

[9] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339-374, 1984.

[10] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proc. 13th ACM STOC*, pp. 263–277, (1981).

[11] U. Vishkin. A case for the PRAM as a standard programmer's model. In *Proc. Workshop on Parallel Architectures and Their Efficient Use: State of the Art and Perspectives*, First Heinz-Nixdorf Symposium, Paderborn, Germany, November 11–13, 1992, Lecture Notes in Computer Science, Volume 678, 1993, 11–19.