Parallel Sorting by Regular Sampling†

Hanmao Shi

Jonathan Schaeffer

Department of Computing Science

University of Alberta

Edmonton, Alberta

Canada T6G 2H1

jonathan@cs.UAlberta.CA

† Funding for this research was provided by the Canadian Natural Sciences and Engineering Resewrch Council, grant number OGP-8173.

Running head: "Parallel Sorting by Regular Sampling"

Contact author:

Jonathan Schaeffer
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1

W: 403-492-3851
H: 403-464-0740
F: 403-492-1071

ABSTRACT

A new parallel sorting algorithm suitable for MIMD multiprocessors is presented. The algorithm reduces memory and bus contention, which many parallel sorting algorithms suffer from, by using a *regular sampling* of the data to ensure good pivot selection. For n data elements to be sorted and p processors, when $n \geq p^3$ the algorithm is shown to be asymptotically optimal. In theory, the algorithm is within a factor of two of achieving ideal load balancing. In practice, there is almost perfect partitioning of work. On a variety of shared and distributed memory machines, the algorithm achieves better than half-linear speedups.

1. Introduction

Sorting is one of most studied problems in computer science because of its theoretical interest and practical importance. With the advent of parallel processing, parallel sorting has become an important area for algorithm research. Although considerable work has been done on the theory of parallel sorting and efficient implementations on SIMD architectures, good parallel performance on a variety of multiprocessor MIMD architectures with a large number of processors remains a challenging problem.

A multitude of innovative parallel sort algorithms have been proposed. Akl provides a good overview of the subject [2]. It is not practical to mention all previous proposed parallel sorts here due to space limitations. Nor is there a necessity to do so, since many of them are based on unrealistic assumptions, which are beyond our interests. Instead, several algorithms that we consider representative of work in this area are discussed. These algorithms are characterized by being realistic and likely to yield good performance in an implementation.

The speedup of a parallel sort achievable on a multiprocessor depends largely on how well the average memory latency and overhead of scheduling and synchronization can be minimized. Based on the general strategies utilized, most parallel sorts suitable for multiprocessor computers can be placed into one of two rough categories: merge-based sorts and partition-based sorts. Merge-based sorts consist of multiple merge stages across processors, and perform well only with a small number of processors. When the number of processors utilized gets large, so does the overhead of scheduling and synchronization, which reduces the speedup. Partition-based sorts consist of two phases: partitioning the data set into smaller subsets such that all elements in one subset are no greater than any element in another, and sorting each subset in parallel. The performance of partition-based sorts primarily depends on how well the data can be evenly partitioned into smaller ordered subsets. Unfortunately, no general, effective method is currently available, and it is an open question of how to achieve linear speedup for parallel sorting on multiprocessors with a large number of processors.

Parallel Quicksort has been a popular choice for research [6, 8, 17]. The basic result is that initial data splitting limits the speedup to a maximum, generally believed to be about 5 or 6, regardless of how many processors are used. A similar effect happens to Evans and Yousif's two-way, merged-based parallel sort [9], as little parallelism can be exploited in the last few phases of merge. Francis and Mathieson have noticed this problem and proposed a parallel Mergesort (PMS) which evenly partitions data to be merged among any number of processors [10]. However, this sorting algorithm is merge-based and, consequently, involves too much data movement, causing frequent memory reference conflicts and serious bus contention. Quinn has implemented a combination of Quicksort and Mergesort, Quickmerge, significantly reducing the amount of data movement [18]. However its execution time is unstable in the sense that the pivots (or dividers) selected are not guaranteed to divide the data to be sorted into ordered subsets reasonably evenly. In the worst case, a single processor may have to perform a Mergesort on nearly all the data in the last phase, which makes linear speedup impossible.

The ability to partition the data evenly into ordered subsets is essential for partition-based sorts. If the distribution statistics of the data are known, it becomes easy to divide the data into equal-sized subsets such that each element in the ith subset is no greater than any element in the $(i+1)$th subset, and then sort each subset in parallel.

Unfortunately, in general, we do not know the data distribution. To overcome this difficulty, Huang and Chow proposed extracting a random sample from the data and use the order information of the sample to help the partitioning, Parallel Sorting by Sampling (PSS) [12]. The effectiveness of sampling depends largely on the distribution of the original data, the choice of a proper sample size, and the way in which the sample is drawn. It appears to be a difficult problem to find pivots that partition the data to be sorted into ordered subsets of equal size without sorting the data first.

This paper describes a new parallel sorting algorithm suitable for a variety of multiprocessor architectures. *Parallel Sorting by Regular Sampling (PSRS)* finds pivots for partitioning the data into ordered subsets of approximately equal size by using a *regular sample* from sorted sublists of the data. It is proven that this form of sampling results in all processors being within a factor of two of achieving ideal load balancing. In practice, this results in a nearly perfect partitioning of the data. On a variety of shared and distributed memory machines, including the Myrias SPS-2, BBN TC2000, a local area network of Sun 3/80 workstations, and iPSC/2-386 and iPSC/860 Hypercubes, the algorithm achieves better than half-linear speedups. For example, on the SPS-2, sorting 8,000,000 data items on 64 processors achieved a 44.4-fold speedup.

2. Parallel Sorting by Regular Sampling

Let the data set to be sorted on a p-processor MIMD multiprocessor be denoted by X, and the size of X be n. Let $X_{i:j}$ be $\{ X_i, X_{i+1}, \ldots X_j \}$, where $0 \le i \le j < n$. For simplicity in the analysis of the algorithm, we assume $p^2 \mid n$ and $X_i \ne X_j$, where $i \ne j$.

PSRS has three phases. The appendix outlines the pseudo-code for the algorithm (more details can be found in [22]). In the first phase, each of p processors sorts a contiguous list of size $w = \frac{n}{p}$ using sequential Quicksort. More precisely, each processor i $(1 \le i \le p)$ sorts a list $X_{(i-1)w \,:\, iw-1}$ with Quicksort. After this phase all processors synchronize and X is now said to be locally ordered. It is convenient to think of X as one contiguous list containing p sorted lists. In practice, depending on the architecture, X may in fact be distributed over the processors.

Define the *regular sample* of the locally ordered X be a set of the following $p(p-1)$ elements:

$$X_{\frac{w}{p}+jw},\;\; X_{\frac{2w}{p}+jw},\;\; \ldots,\;\; X_{\frac{(p-1)w}{p}+jw},\;\; \text{Rwhere}\,P\;\; 0 \le j \le p-1.$$

From each of the p lists, $p-1$ samples are chosen, evenly spaced throughout the list. The regular sample contains the "order information" of the original data set. The pivots, which divides the regular sample into ordered subsets of sample of equal size, will also partition the original data into order subsets of roughly equal size.

In the second phase, the regular sample set, Y, is sorted using sequential Quicksort yielding an ordered list $Y_1, Y_2, \ldots, Y_{p(p-1)}$. Next choose $Y_{\frac{p}{2}}, Y_{p+\frac{p}{2}}, \ldots, Y_{(p-2)p+\frac{p}{2}}$ as the $p-1$ pivots for partitioning X, referred to as $y_1, y_2, \ldots, y_{p-1}$. In other words, the $p(p-1)$ samples are sorted and $p-1$ elements, evenly spaced throughout the sorted list, are chosen to be the pivots.

The partitioning of X is accomplished as follows. Each processor finds where each of the $p-1$ pivots divides its list, using a binary search. More precisely, each processor i $(1 \le i \le p)$ finds the index of the largest element no larger than the j-th pivot, $j = 1, 2, \ldots, p-1$. After doing this, all processors synchronize. At this point each of the p sorted lists of X have been divided into p sorted sublists with the property that every item in every list's i-th sorted sublist is greater than any item in any list's $(i-1)$-th sorted sublist, for $2 \le i \le p$.

In the last phase, each processor i $(1 \le i \le p)$ performs a p-way Mergesort to merge all the i-th sorted sublists of p lists. Note that unlike phase one, in which each processor sorts a contiguous block of keys, in phase three each processor merges p sublists stored in p different areas. Because of the demarcations established in phase two, their merges are completely independent of each other. After this all processors synchronize and X is sorted.

Fig. 1 illustrates how PSRS works for $n = 36$ and $p = 3$. In phase 1, each processor is assigned $w = \frac{n}{p} = 12$ contiguous elements to sort. After sorting its list, each processor selects a regular sample, elements 4 and 8, evenly spaced throughout the 12 elements. In the second phase, the $p(p - 1)$ samples are gathered together and sorted. In the example, this is the list of elements {10, 13, 16, 22, 23, 27}. From this, a regular sample of $p - 1 = 2$ pivots are selected, evenly spaced throughout the list, resulting in the selection of pivots 14 and 23.

Finally, the 3 sorted lists of 12 elements are partitioned into 3 sublists by the pivots. Each processor keeps one sublist and passes the others on to the appropriate processor. In the example, processor 1 keeps the first sublist, {0, 1, 2, 9}, and sends the second, {16, 17}, to processor 2 and the third, {24, 25, 27, 28, 30, 33}, to processor 3. Similarly for processors 2 and 3. Once each processor receives its portions of the data from the other processors, it can merge the results into the final sorted array.

Given the extensive literature on parallel sorting, it is not surprising that PSRS is similar to other proposed sort algorithms. For example, Won and Sahni's balanced bin sort [26] and Abali, Ožguňer and Bataineh's load balanced sort [1] for Hypercube architectures both resemble PSRS. Both differ from PSRS in the second phase of the algorithm. The balanced bin sort merges each processor's sample, keeping the elements in the odd positions and discarding those in the even positions. In the final partition phase, the sort has an upper bound of $\frac{3n}{p}$ elements that must be merged by a single processor (PSRS is shown to have a bound of $\frac{2n}{p}$ in the next section). The load balanced sort iterates in phase 2, repeatedly modifying its selection of pivots until it eventually achieves perfect load balancing. In this paper, it is shown that PSRS achieves almost perfect load balancing in practice without the overhead of finding a perfect partition.

3. Complexity Analysis

In phases one and two of PSRS, all processors have roughly the same amount of work to do. In phase three, it is not obvious how evenly the work is divided because this depends on how well the data has been partitioned. However, it can be shown that there is an upper bound on the amount of work a processor must do.

Theorem 1: In phase 3 of PSRS, each processor merges less than $2w$ elements.

Proof: Consider any processor i, where $1 \leq i \leq p$. There are three cases:

1) $i = 1$. All the data to be merged by processor 1 must be $\leq y_1$. Since there are $p^2 - p - \frac{p}{2}$ elements of the regular sample which are $> y_1$, there are at least $(p^2 - p - \frac{p}{2}) \frac{w}{p}$ elements of X which are $> y_1$. In other words, there are at most $n - (p^2 - p - \frac{p}{2}) \frac{w}{p} = (p + \frac{p}{2}) \frac{w}{p} < 2w$ elements of X which are $\leq y_1$.

2) $i = p$. All the data to be merged by processor p must be $> y_{p-1}$. There are $(p-2)p + \frac{p}{2}$ elements of the regular sample which are $\leq y_{p-1}$. That is, there are at least $(p^2 - 2p + \frac{p}{2})\,\frac{w}{p}$ elements of X which are $\leq y_{p-1}$, or there are at most $n - (p^2 - 2p + \frac{p}{2})\,\frac{w}{p} < 2w$ elements of X which are $> y_{p-1}$.

3) $1 < i < p$. All data to be merged by processor i must be $> y_{i-1}$ and $\leq y_i$. There are $(i-2)p + \frac{p}{2}$ elements of the regular sample which are $\leq y_{i-1}$, implying $lb = ((i-2)p + \frac{p}{2})\frac{w}{p}$ elements of X. On the other hand, there are $(p-i)p - \frac{p}{2}$ elements of the regular sample which are $> y_i$. As well, there are $\frac{w}{p} - 1$ elements that fall between y_i and the next highest element in the regular sample. Thus there are at least $ub = ((p-i)p - \frac{p}{2} + 1)\,\frac{w}{p} - 1$ elements of X which are $> y_i$. Since the size of X is n, there are at most $n - ub - lb = 2w - \frac{w}{p} + 1 <$ $2w$ elements of X for processor i to merge.

In conclusion, no processor merges more than $2w = \frac{2n}{p}$ elements in the last phase of PSRS. If p^2 doesn't divide n evenly, it is easy to prove that no processor merges more than $2\lceil \frac{n}{p} \rceil$ elements.

What is the time complexity of PSRS? The analysis for phase one is easy. The initial Quicksort phase has time complexity $O(w \log w)$, in the average case †, representing

† Quicksort is an $O(w \log w)$ algorithm in practice, but has a worst case that is $O(w^2)$. PSRS can be changed to use an algorithm that has a worst-case performance of $O(w \log w)$, such as Mer-

the time consumed by each processor to sort w data using sequential Quicksort. In phase two, sorting the regular sample using Quicksort requires $O(p^2 \log p^2)$ time. Then each processor performs $p - 1$ binary searches on sorted lists of no greater than w elements. Hence the time complexity of phase two is $O(p^2 \log p^2 + p \log w)$. In phase three, the size of data to be merged by any processor is always less than $2w$. Thus the last phase can finish in no more than $O(2w \log p)$ time.

Summation of the times of all three phases gives a time complexity for PSRS of $O(w \log w + w \log p + p \log w + p^2 \log p^2)$ which is asymptotic to $O(w \log w) = O(\frac{n}{p} \log n)$ when $n \geq p^3$.

The bound on the size of the data to be merged in PSRS is an important difference which other partition-based sorts, such as Quickmerge and PSS, don't have. Theoretically, PSRS is optimal when $n \geq p^3$, regardless of the distribution of the original data. Aside from easy scheduling and few synchronization points, another advantage of PSRS lies in its good per-task locality of reference. In all three phases, each task accesses only a small portion (always less than $2w$) of the data and the accesses are highly localized. This minimizes the amount of paging, hence reducing the average memory latency. The algorithm is especially suitable for distributed memory multiprocessors. If each processor is initially allocated a portion of approximately w elements, then no data transmission is required during the first phase. In the second phase, only $p(p - 1)$ elements need to be collected and $p - 1$ pivots are required to be broadcast to all the processors. In the last phase, each processor has to send $p - 1$ sublists to the other $p - 1$ processors. After a processor receives its sublists, it stores them locally and then works on them

gesort.

totally independent of others. The total number of messages required in this phase is therefore $p\ (p\ -\ 1)$ and the total data traffic is no greater than n.

Two final points need to be addressed. First, the algorithm and its analysis is based on there being no duplicate elements in the list to be sorted. If there are duplicates, then Theorem 1 does not hold. However, even if there are a large number of keys, each having a small number of duplicates, in practice there will be no problem. If there are keys for which there are a large number of duplicates, parallel performance will degrade unless the algorithm is enhanced to cover this case (for example, by adding a secondary key).

Secondly, it is possible to use more than $p\ (p\ -\ 1)$ elements in the regular sample for choosing the pivots. The more elements chosen, the tighter the bound in Theorem 1 can be made. A larger sample implies more overhead in determining the pivots, although, this could also be done in parallel. If n is large enough, the more effective load balancing may offset this cost.

4. Experimental Results

The first version of PSRS was implemented in C to run on a 64-processor Myrias SPS-2 [4]. Parallelism is expressed on the SPS-2 system by a single *pardo* extension to the programming languages Fortran and C. The *pardo* causes independent loop iterations to be executed in parallel as independent tasks. Parallel tasks can be heterogeneous and recursive. Each task, in principle, gets (by demand paging) a virtual copy of its parent's address space, and manages its own portion of the program independent of other concurrent tasks. When the tasks of a loop all complete, the new parent state is formed by *merging* the results computed by all the parallel tasks. Specifically, there are four *merging* rules:

1) no update:If no child task assigns to a variable, then the parent variable is unchanged.

2) one task updates:If exactly one child task assigns to a variable, the variable in the parent task is changed to the assigned variable.

3) several tasks update with the same value:If more than one child task assign to a variable, but the values assigned are identical, then the variable in the parent task is changed to the assigned value.

4) otherwise:Any other update pattern will cause the value of the parent task variable to be unpredictable.

The parent task then resumes just as it would at completion of a corresponding serial *do/for* loop in Fortran/C. The fundamental idea behind this new memory model is to relieve memory contention at the price of memory replication.

The Myrias SPS-2 system provides a transparent control mechanism that automatically schedules parallel tasks on PEs, optimizes the use of hardware resources, and manages all data motion. This, however, has both advantages and disadvantages. The major advantage is that it simplifies programming; a user need not worry about the details of task allocation, data motion, and synchronization. On the other hand, this also makes it impossible for a user to have their own control scheme to execute one's specific program. This is not always desirable since the system's optimizer cannot take advantage of the properties specific to a program.

PSRS was tested using randomly-generated 32-bit integers with various distributions. No tests were made for duplicate elements, of which there were undoubtedly a few. As long as the number of duplicates is small in relation to the total number of elements, this will not significantly affect the performance. The size of the array to be sorted ranged from 0.1 million to 8 million elements. Experiments were done using 2, 4, 8, 16, 32, and 64 processors on a dedicated machine. Each data point presented in this section was obtained as the average of 20 program executions, each on a different set of test data.

An improved version of Quicksort was used as the optimal sequential sort. The improvements consisted of two common modifications. First, the median of the first, middle, and last elements of the subarray were used as the pivot. Second, subarrays with size less than ten integers were sorted using linear insertion sort.

Unfortunately, one PE on the SPS-2 can only sort at most 0.2 million 32-bit integers due to the limitations of main memory. To compute the speedups, we need to know the sequential sorting time of the improved Quicksort on the Myrias SPS-2 when n is larger than 0.2 million. Since the bound of sequential comparison-based sorting is $O(n \log n)$, it is reasonable to assume the time of sorting n integers using the improved Quicksort on one PE of the Myrias SPS-2 as follows:

$$t_{PE}(n) = C\, n \log n$$

where C is a constant independent of size n. Sequential times for lists of more than 0.2 million elements were calculated using the following formula:

$$t_{PE}(n) = \frac{n \log n}{100{,}000 \log 100{,}000} \times t_{PE}(100{,}000)$$

where 0.4 million $\leq n \leq$ 8 million and $t_{PE}(100{,}000)$ = 6.63 seconds, the experimental time for the improved sequential Quicksort to sort 0.1 million integers on one PE of the SPS-2. Note that if one uses this formula to compute $t_{PE}(200{,}000)$, the result is almost a perfect match with the corresponding experimental time.

To obtain more confidence in the speedups, we also took an experimental approach to estimate the sequential sorting time of the improved version of Quicksort on the Myrias SPS-2. Timing experiments were done on a MIPS R1000, with 32 MByte of main memory, that is able to sort 4 million 32-bit integers without any memory shortage. Without memory limitations, we would have the following approximate equation:

$$\frac{t_{PE}(n_1)}{t_{R1000}(n_1)} = \frac{t_{PE}(n_2)}{t_{R1000}(n_2)}$$

where n_1 and n_2 are the sizes of data to be sorted, and $t_{R1000}(n)$ represents the sequential sorting time of the improved Quicksort on the MIPS machine. Using this estimate tends to result in higher sequential times, improving our speedups. The results presented in this section use the first approximation for the sequential running time, preferring to err on the side of being conservative. Sequential times that were obtained by calculation, instead of measurement, are indicated by italics.

Table 1 shows the time to sort using PSRS and Fig. 2 plots the speedups achieved. As the problem size increases, task granularity increases, offsetting the overheads of the algorithm, resulting in better speedups. Sorting 8,000,000 items with 64 processors gave a 44.4-fold speedup †.

For comparison purposes, PMS, Quickmerge and PSS have also been implemented. For brevity, only the PSS results are shown, as they are significantly better than both PMS and Quickmerge. Table 2 shows the time to sort the elements using PSS and Fig. 3

† Note that the original PSRS speedup results are less than those reported here [22]. The original results included the cost for initially distributing the data, and the time taken to merge all the sorted sublists back to one processor. Most parallel sorting results assume the data is initially distributed and that merging the sorted sublists in one processor is not necessary.

plots the speedups achieved. Our implementation of PSS used an initial sample of $16p$ elements randomly spread over the data. The speedup increases with the value of p in general, but quickly tapers off when p is greater than 16. This is mainly because when the number of processors used increases, so does the probability that some processor may have to merge significantly more data than its share. Theoretically it is possible that some processor may have to perform a Mergesort on nearly all the data.

Since the strength of PSRS is the claim that the data is evenly partitioned, it is important to measure how successful it was. The *RDFA* measure is used to quantify this. *RDFA* is the *R*elative *D*eviation of the size of the largest partition *F*rom the *A*verage size of the p partitions, which is defined as follows:

$$RDFA = \frac{m}{w}$$

where m is the size of the largest of the p partitions and $w = \frac{n}{p}$, the average size of the p partitions. Since $m \geq w$, it is always the case that $RDFA \geq 1$. The smaller the *RDFA* is, the more balanced the partitioning is. An *RDFA* of r indicates that largest portion was r times larger than optimal.

The *RDFA* measures for PSRS are shown in Table 3 and PSS in Table 4. For PSS, the data clearly shows that as the number of processors increases so does the *RDFA* metric. Hence, for a large number of processors, the disparity in the amount of work between processors grows, reducing the speedup possible. For PSRS, the results are remarkably consistent, in all cases being within a few percent of perfect partitioning.

From the data obtained, a few observations can be made on the performance of PSRS:

1) As the number of PEs increase, the speedups are good provided that the problem size is large enough to give each PE a sufficient amount of work to do. When both the problem size and number of PEs double, so does the speedup.

2) Although we have used a conservative approach in computing the speedups, the results are still quite good. PSRS has a speedup of 44.4 when sorting 8 million 32-bit integers with 64 PEs. Using the other estimate for sequential time, 49-fold speedups are obtained. The trends suggest that additional processors beyond 64 can be effectively utilized to further reduce sorting times

3) Table 3 shows that PSRS RDFAs are consistently close to 1.0, perfect load balancing. The size of the largest of the *p* partitions is very close to the average, indicating that the regular sample does indeed provide a good representation of the original data. Surprisingly, even when *p* is small, the regular sample is still able to help partition the data evenly. Theoretically, the *RDFA* of PSRS is only guaranteed to be less than two. However, in practice, our experiments show it to be within a few percent of optimal. Contrast that with the PSS results, where the sorting of 8 million integers on 64 processors was a factor of 2.4 off of the optimal partition.

4) PSRS runs poorly on the SPS-2 when the size of the data is not large enough. Two factors contribute to this. The first comes from the SPS-2 architecture. The implementation of PSRS required using three *pardo* statements. The overhead of a *pardo* is not insignificant. For good performance, the problem size must be large enough to provide each individual processor a sufficient amount of computation. The second factor comes from the algorithm itself. The sampling cost in PSRS is basically fixed, regardless of the size of the data. This cost, in proportion to the total

sorting time, increases as the problem size decreases, reducing the speedup.

5) PSRS is guaranteed to work well on any data distribution, as long as the number of duplicates is relatively small. For example, using the IMOX data set from image processing [7], which form a normal distribution, speedups that were almost identical to those reported here were achieved [15].

6) The executions of PSRS on the SPS-2 show that system activities account for the performance losses. Most of the system time is spent in memory replication and merging required by the Myrias SPS-2 memory model. The rest is spent in task creation, scheduling, and synchronization. Since sorting is a data-intensive problem, it is not a "good" problem for the Myrias SPS-2 to solve.

In general, the speedup relationships of the four algorithms implemented are as follows:

$$PMS \leq Quickmerge \leq PSS \leq PSRS \ .$$

The speedups of PMS are found to be rather poor. We can’t confirm Francis and Mathieson’s claim that PMS has linear speedup [10]. These results may be due to the Myrias SPS-2 architecture. However, one should not understate the overhead of scheduling and synchronization during the second phase of the algorithm. The speedups of Quickmerge are much better than those for PMS, but not as good as for PSS. Random sampling seems to be a reasonable approach to parallel sorting, depending on the data distribution.

PSRS has been implemented on the shared memory BBN TC2000 [16]. A 44.5-fold speedup was achieved sorting 8,000,000 elements on 64 processors. The sorting was done in each processor’s local memory, while the global memory was used to communicate data.

5. PSRS on Hypercubes

In this section, PSRS is extended to run on Hypercubes machines. A Hypercube is a multiprocessor in which the (processor) nodes can be imagined to lie at the vertices of a multidimensional cube. Multiprocessors based on this topology require modest interconnections, yet seem rich enough to allow many of the classical interconnections to be easily constructed. The architecture is appealing because of its homogeneity and symmetry properties. Considerable effort has been devoted at achieving good results (for example, [1, 13, 14, 21, 23, 24, 26]).

Assume p is the number of nodes on a Hypercube and that p is a power of 2. The nodes are indexed by a linear sequence ranging from 0 to $p - 1$, with the neighboring nodes differing in exactly one bit position in their binary representations. Assume the data is evenly distributed to all p nodes. The n elements are said to be sorted if all elements are sorted at each node, and there is no element at any node that fits in the range of values of any other node.

The complexity of Hypercube sorting is usually measured in terms of the computing and communication costs. Presently, Johnsson's algorithm, which is an adaptation of Batcher's bitonic sorting algorithm [3], has the best theoretical worst case bounds for Hypercube sorting [13]. His algorithm has $O(w \log w + w \log^2 p)$ computing complexity and $O(w \log^2 p)$ communication complexity, with $w = \frac{n}{p}$.

PSRS appears easily adaptable to Hypercubes. The algorithm requires 4 phases:

1) Each node sorts in parallel its local list of size w with sequential Quicksort. Each

node selects $p - 1$ elements evenly spaced from its local sorted list and sends them to a designated node, say node 0.

2) After it receives all $p\ (p - 1)$ elements, node 0 selects $p - 1$ pivots using the same method described previously. It then broadcasts the chosen pivots to all other nodes. After each node receives the pivots, a binary search is used to partition each local list into p sublists.

3) Each node i $(0 \le i \le p - 1)$ sends its $(j + 1)$-th sublist to node j, $j = 0, 1, ..., p - 1, j \ne i$.

4) Finally, after it receives its $p - 1$ sublists, each node performs a merge on all the received sublists plus the local one unsent. The sort completes after all nodes finish merging.

Note that no global synchronization is necessary between phases. Data transfers are performed by passing messages between nearest neighbors. Thus data which must travel from node *A* to node *B* must cross a sequence of nodes starting at node *A* and ending at node *B*.

The lower bound of the total computing time required for sorting n numbers is $O(p^2 \log p^2 + w \log p + w \log w + p \log w)$, which is asymptotic to $O(\frac{n}{p} \log n)$ when $n \ge p^3$. The computing time alone is not sufficient to distinguish a parallel sort for Hypercubes. To study the communication complexity of this algorithm, the model of Saad and Schultz is used [20]:

a) Moving a vector of length m from one node to a neighbor takes the time $\beta + m\tau$, where β represents the communication startup time and τ, the elemental transfer time.

b) It takes the same time to move the same data from one node to any number of its $\log\ p$ neighbors.

Further details can be found in [20].

The algorithms and estimates for the times of data transfers required in phases 1, 2 and 3 are sketched in Fig. 4 (phase 4 has no communication). For convenience, exponents applied to the binary bits 0 and 1 stand for concatenation.

Phase 1 of the algorithm has $\log\ p$ steps. At the jth step, the algorithm transfers $2^{j-1}\ p$ selected elements between two nodes, since the number of elements collected doubles at every step. Therefore the total time required for gathering p^2 elements, p per node, is

$$\sum_{j=1}^{\log\ p} (\beta + 2^{j-1} p\tau) = \beta \log\ p + (p - 1) p\tau .$$

Phase 2 of the algorithm consists of $\log\ p$ steps, all requiring the same amount of time. Hence the total time for broadcasting $p - 1$ pivots in phase 2 is $(\beta + (p - 1)\tau) \log\ p$.

The data transfer operations in phase 3 are equivalent to transposing a $p \times p$ matrix, if the $(j+1)$th sublist at node i is viewed as an entry (i, j) of the matrix, where $0 \le i, j \le p - 1$. To formulate the algorithm, we denote by $(i)_k$ the bit in position k of the $\log\ p$-bit binary representation of number i, and $\bar{i}^k$ the number whose binary representation differs only in the kth bit from that of i. Let $L_{i,\ j}$ be the $(j+1)$th sublist at node i and $|L_{i,j}|$ be its length, where $0 \le i, j \le p - 1$. $[L_{i,\ j}]$ stands for the $p \times p$ matrix as explained before. Then we have the following relations:

$$0 \le |L_{i,j}| \le w, \quad 0 \le i,\ j \le p - 1, \tag{1}$$

$$\sum_{j=0}^{p-1} |L_{i,j}| = w, \quad 0 \le i \le p - 1, \quad \text{Rand}\, P \tag{2}$$

$$\sum_{i=0}^{p-1} |L_{i,j}| < 2w, \quad 0 \le j \le p - 1. \tag{3}$$

Proving (1) and (2) is trivial. The proof of (3) is the same as in Section 3.

The principle of the algorithm is to exchange data across opposite edges along the $\log p$ dimensions in turn. The first step consists of exchanging the matrices that are in the upper right and lower left positions of the large 2×2 block matrix obtained from $[L_{i,j}]$ by splitting it into four equal parts. Each of the four can again be split into four parts in the same manner.

The next step deals with each of the four parts in a manner similar to what was done with the original matrix, and the total number of the sublists exchanged is still p. To obtain the time bound of each step, the following lemmas are introduced:

Lemma 1: At step k ($1 \le k \le \log p$), all the sublists sent by any node must come from 2^{k-1} different rows and $2^{\log p - k}$ different columns of the original matrix.

Proof: At the first step, all the sublists sent by any node come from one single row and $\frac{p}{2}$ different columns of the original matrix. After each step, the number of different rows of the sublists to be sent doubles, while the number of different columns of the sublists to be sent reduces by half.

Lemma 2: At step k ($1 \le k \le \log p$), the total size of all the sublists sent by any node is no larger than

$$min\left\{2^{k-1}w,\ \ 2^{\log p\ -\ k\ +\ 1}w\right\}.$$

Proof: Straightforward from Lemma 1, Equation (2), and Equation (3).

Hence step k costs less than or equal to

$$2(\beta\ +\ (min\left\{2^{k-1}w,\ 2^{\log p\ -\ k\ +\ 1}w\right\})\tau),$$

and the upper bound on the total communication time of phase 3 is

$$\sum_{k=1}^{\log p} 2(\beta\ +\ (min\left\{2^{k-1}w,\ 2^{\log p\ -\ k\ +\ 1}w\right\})\tau)\ =\ 2\beta\ \log\ p\ +\ \frac{6(\sqrt{p}\ -\ 1)}{p}\,n\tau.$$

Summation over the three phases yields an upper bound of the total communication cost of PSRS as follows:

$$4\beta\ \log\ p\ +\ (\frac{6(\sqrt{p}\ -\ 1)}{p}\,n\ +\ (p\ -\ 1)(p\ +\ \log\ p))\tau,$$

where the first item represents the total startup time, and the latter one stands for the total transmission time. The total startup time is determined only by p and is of a relatively low order. When $n >> \log p$, the total communication cost is asymptotic to $O(\frac{n}{\sqrt{p}})$. The average communication time can be expected to be *much* less than this bound. Note that no assumption is made that allows a node to send (receive) data to (from) its neighbors simultaneously.

Summarizing, the extension of PSRS on a Hypercube computer has $O(w \log w)$ computing complexity and $O(\frac{n}{\sqrt{p}})$ communication complexity, where $n >= p^3$. Compared with Johnsson's algorithm, it has a better computing complexity. As well, the startup time of the PSRS is much less than Johnsson's.

PSRS has been implemented for the Intel iPSC/2-386 and iPSC/860 Hypercubes [16]. On the iPSC/2-386 using 32 processors, sorting 4,000,000 items resulted in a 27.49-fold speedup. The better results here are due to fast communication between processors. The iPSC/860 is similar to the iPSC/2-386 except for the use of a faster processor per node. Hence, for comparably sized problems, one would expect the speedups to be less than for the iPSC/2-386. Using 64 processors, sorting 8,000,000 items gave a 38-fold speedup. Note that results reported here were obtained using the same version of PSRS as used on the Myrias and BBN machines, except for the parallel programming constructs used. No attempt was made to take advantage of the architecture. In particular, the communications patterns did not take advantage of the savings possible by using the architecture's nearest neighbour links. Other parallel Hypercube sorts have reported better results than PSRS (for example, [1, 24]) in part because they exploit the communication structure of the Hypercube. Our PSRS implementation can be similarly extended.

6. PSRS on LANs

The last decade has witnessed the explosive growth of LANs (Local Area Networks). Large files on such a network may be physically distributed over a number of stations. A file X of size n is assumed to be distributed over p stations of a LAN. The stations are logically ordered into a linear sequence with station $i+1$ being immediately to the right of station i, where $1 \leq i \leq p - 1$. We initially assume an equal number of $w = \frac{n}{p}$ elements at each station. The distributed sorting problem is to relocate some of the elements so that each subfile at each station is sorted and each element at station i is less than or equal to any element at station $i+1$.

To simplify the discussion, we refer to an Ethernet-type LAN, where all stations are interconnected by a common bus. Each station on the LAN is able to send a message to any other station(s) across the bus. Since the cost of a distributed sort is dominated by the communication cost, its complexity is usually measured in terms of the required number of messages and the total data traffic.

Several distributed algorithms have been proposed for solving the distributed sorting problem [5, 19, 25]. Wegner's algorithm is based on parallel Quicksort [25]. In each iteration, every partition is divided into two smaller ones. On the average, the algorithm has message complexity $O(p \log p)$ and data traffic complexity $O(n \log p)$. Rotem, Santoro and Sidney's algorithm [19] starts with finding $p - 1$ even partition points by using a distributed version of the kth selection algorithm [11]. After that, each data item is sent directly to its destination station. The algorithm has message complexity $O(p^2 \log n)$ and data traffic complexity $O(n)$.

While both algorithms have relative low orders of complexity compared with other distributed sorting algorithms, they still suffer from drawbacks. Wegner's algorithm requires a large amount of data traffic, while Rotem et al.'s algorithm requires too many messages.

Although PSRS was developed for sorting on a multiprocessor, the algorithm is readily amenable to distributed sorting. As was analyzed in Section 3, the algorithm has $O(n)$ data traffic complexity and $O(p^2)$ message complexity, which is a substantial improvement over the $O(p^2 \log n)$ worst case for Rotem et al.'s algorithm.

PSRS has been implemented on a network of Sun 3/80 workstations connected via

Ethernet. For 500,000 items, a 4-fold speedup with 8 processors has been achieved [15]. These poor results were the result of a system limitation that restricted the amount of data that could be sent between processors at a time.

7. Conclusions

Long memory latency and the overhead of scheduling and synchronization are two critical factors that greatly affect the speedup of a parallel algorithm on present multiprocessor architectures. Parallel Sorting by Regular Sampling is intended to minimize both. It has a high per-task reference locality, yet is simple to schedule and synchronize. Experiments on a variety of architectures make it clear that good speedups for parallel sorting on present multiprocessor architectures is indeed achievable. The results reported here are quite encouraging, considering that sorting is generally believed a hard problem to parallelize. The success of PSRS indicates that good sorting performance for a variety of supercomputers is obtainable.

Acknowledgments

Paul Lu, John Shillington, Xiaobo Li and two anonymous referees supplied us with constructive suggestions for improving the paper. The Myrias Research Corporation provided us with machine time.

References

1. Abali, B., Ožguňer, F. and Bataineh, A., Load Balanced Sort on Hypercube Multiprocessors, *5th Distributed Memory Computing Conference* , 1990, 230-236.
2. Akl, S.G., *Parallel Sorting Algorithms*, Academic Press, Orlando, Florida, 1985.
3. Batcher, K.E., Sorting Networks and Their Applications, *Proceedings of the 1968 Spring Joint Computer Conference 32*, (1968), 307-314, AFIPS Press.
4. Beltrametti, M., Bobey, K., Manson, R., Walker, M. and Wilson, D., PAMS/SPS-2 System Overview, *Supercomputing Symposium*, 1989, 63-71.
5. Cheung, T.Y., An Algorithm with Decentralized Control for Sorting Files in a

Network, *Journal of Parallel and Distributed Computing 7*, (1989), 464-481.

6. Deminet, J., Experience with Multiprocessor Algorithms, *IEEE Transactions on Computers C-31*, (1982), 278-288.
7. Dubes, R.C. and Jain, A.K., Clustering Techniques: The User's Dilemma, *Pattern Recognition 8*, (1976), 247-260.
8. Evans, D.J. and Yousif, N.Y., Analysis of the Performance of the Parallel Quicksort Method, *BIT 25*, (1985), 106-112.
9. Evans, D.J. and Yousif, N.Y., The Parallel Neighbor Sort and Two-way Merge Algorithm, *Parallel Computing 3*, (1986), 85-90.
10. Francis, R.S. and Mathieson, I.D., A Benchmark Parallel Sort for Shared Memory Multiprocessors, *IEEE Transactions on Computers 37*, 12 (1988), 1619-1626.
11. Frederickson, G.N., Tradeoffs for Selection in Distributed Networks, *Proceedings of the 2nd ACM Symposium on the Principles of Distributed Computing*, 1983, 154-160.
12. Huang, J.S. and Chow, Y.C., Parallel Sorting and Data Partitioning by Sampling, *7th International Computer Software and Applications Conference*, 1983, 627-631.
13. Johnsson, S.L., Combining Parallel and Sequential Sorting on a Boolean n-cube, *International Conference on Parallel Processing*, 1984.
14. Li, P.P. and Tung, Y.W., Parallel Sorting on the Symult 2010, *5th Distributed Memory Computing Conference*, 1990, 224-229.
15. Li, X., Lu, P., Schaeffer, J., Shi, H. and Shillington, J., Parallel Sorting by Regular Sampling, Tech. Rep. 91-6 Department of Computing Science, University of Alberta, 1991.
16. Lu, P. and Shillington, J., Sorting on MIMD Architectures, CMPUT 535 report, Department of Computing Science, University of Alberta, 1990.
17. Moller-Nielsen, P. and Staunstrup, J., Problem-heap: A Paradigm for Multiprocessor Algorithms, *Parallel Computers 4*, (1987), 63-74.
18. Quinn, M.J., Parallel Sorting Algorithms for Tightly Coupled Multiprocessors, *Parallel Computing 6*, (1988), 349-357, North-Holland.
19. Rotem, D., Santoro, N. and Sidney, J.B., Distributed Sorting, *IEEE Transactions on Computers 34*, 4 (1985), 372-275.
20. Saad, Y. and Schultz, M.H., Data Communication in Hypercubes, *Journal of Parallel and Distributed Computing 6*, 1 (1989), 115-135.
21. Seidel, S. and Ziegler, L.R., Sorting on Hypercubes, in *Hypercube Multiprocessors*, M.T. Heath (ed.), SIAM, 1987, 285-291.
22. Shi, H., Parallel Sorting on Multiprocessor Computers, M.Sc. thesis, Department of Computing Science, University of Alberta, 1990.
23. Tang, T., Parallel Sorting on the Hypercube Concurrent Processor, *5th Distributed Memory Computing Conference*, 1990, 237-240.
24. Wagar, B., Hyperquicksort: A Fast Sorting Algorithm for Hypercubes, in *Hypercube Multiprocessors*, M.T. Heath (ed.), SIAM, 1987, 292-299.
25. Wegner, L.M., Sorting a Distributed File in a Network, *Computer Networks 8*,

(1984), 451-461.

26. Won, Y. and Sahni, S., A Balanced Bin Sort for Hypercube Multicomputers, *Journal of Supercomputing 2*, (1988), 435-448.

Appendix. The PSRS Algorithm

```
procedure PSRS(array, n, p)
# array[0:n−1]: array to be sorted, n: size of the array, p: number of processors
begin
        # Divide the array into p contiguous lists and sort each in parallel.
        # Select p − 1 elements evenly spaced from each of the p sorted lists
        # as the regular sample and store them in sample[0:p(p − 1)−1].
        size = ⌊(n+p−1)/p⌋
        rsize = ⌊(size+p−1)/p⌋
        for i = 0 to p−1 do in parallel
                start = i×size
                end = (i+1)×size−1 # = start + size − 1
                if end ≥ n then
                        end = n−1
                # Sort subarray array[start:end] with sequential Quicksort.
                Quicksort(array, start, end)
                for j = 1 to p−1 do
                        if j×rsize ≤ end then
                                sample[i×p+j] = array[j×rsize]
                        else    sample[i×p+j] = array[end]
                endfor
        endfor

        # One designated processor sorts the regular sample sample[0 : p(p−1)−1]
        # using sequential Quicksort.  Then it chooses p−1 pivots from the sorted
        # regular sample and stores them in pivots[1:p−1].
        Quicksort(sample, 0, p(p − 1)−1)
        for i =0 to p−2 do
                pivots[i] = sample[i×p+p/2]
        endfor

        # Divide, in parallel, each sorted list i into p sublists:
        # array[subsize[i×(p+1)+j] : subsize[i×(p+1)+j+1]−1],
        #  0 ≤ j ≤ p−1, with the chosen pivots as splitters.
        for i = 0 to p−1 do in parallel
                start = i×size
                end = (i+1)×size−1 # = start + size − 1
                if end ≥ n then
                        end = n−1
                subsize[i×(p+1)] = start
                subsize[i×(p+1)+p] = end+1
                Sublists(array, start, end, subsize, i×(p+1), pivots, 1, p−1)
        endfor

        # In parallel, count the size of each of the p partitions.
        for i = 0 to p−1 do in parallel
```

```
            bucksize[i] = 0
            for j = i to p×(p+1)−1 step p+1 do
                bucksize[i] = bucksize[i] + subsize[j+1] − subsize[j]
            endfor
        endfor

        # In parallel, decide the start-point of each partition in the final array.
        for i = 1 to p−1 do in parallel
            bucksize[i] = bucksize[i] + bucksize[i−1]
        endfor
        bucksize[0] = 0

        # Merge each partition in parallel.
        for i = 0 to p−1 do in parallel
            merge the following sublists with the standard Mergesort:
            array[subsize[i+j×(p+1)] : subsize[i+j×(p+1)+1]−1], 0 ≤ j ≤ p−1
            The merged results are stored in array starting from index bucksize[i]
        endfor
end PSRS


procedure Sublists(array, start, end, subsize, at, pivots, fp, lp)
# Recursively divide array[start:end] into p sublists with pivots[fp:lp] as splitters.
# The final demarcations for the  sublists are stored in subsize starting from index at.
begin
        mid = ⌊(fp+lp)/2⌋
        pv = pivot[mid]
        lb = start
        ub = end
        while lb ≤ ub do
            center = ⌊(lb+ub)/2⌋
            if array[center] > pv then
                ub = center−1
            else    lb = center+1
        endwhile
        subsize[at+mid] = lb
        if fp < mid then
            Sublists(array, start, lb−1, subsize at, pivots, fp, mid−1)
        if mid < lp  then
            Sublists(array, lb, end, subsize, at, pivots, mid+1, lp)
end Sublists
```

FIG. 1. PSRS example.

Sizes	Sorting Times *(in seconds)*						
	1PE	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	6.63	3.86	2.14	1.56	1.06	1.09	1.66
200000	14.00	8.08	4.20	2.74	1.71	1.43	1.83
400000	*29.71*	16.77	8.60	5.58	3.13	2.11	2.20
800000	*62.62*	-	21.51	10.67	5.84	3.50	2.86
1000000	*79.56*	-	29.88	13.13	6.97	4.38	3.12
2000000	*167.10*	-	-	-	13.73	7.87	5.19
4000000	*350.17*	-	-	-	-	14.83	8.70
8000000	*732.28*	-	-	-	-	-	16.49

Table 1. Sorting times of PSRS.

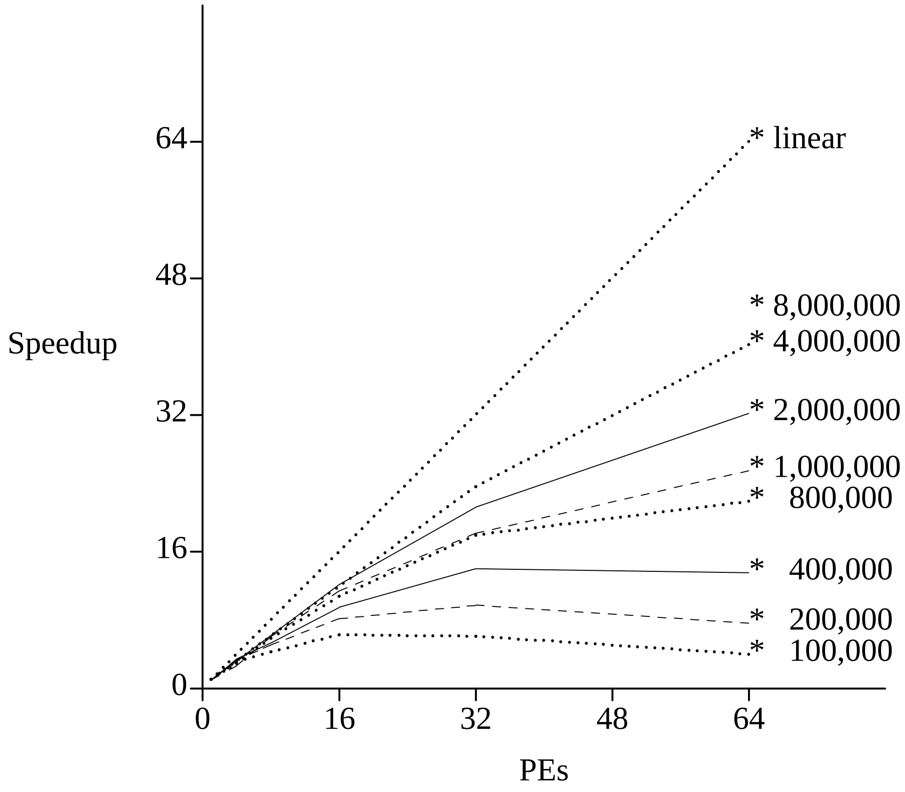

FIG. 2. Speedups of PSRS.

Sizes	Sorting Times *(in seconds)*						
	1PE	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	6.63	4.33	2.44	2.00	1.80	1.97	2.34
200000	14.00	8.87	4.78	3.35	2.60	2.72	3.00
400000	*29.71*	19.34	9.67	6.26	4.33	3.91	4.34
800000	*62.62*	-	24.27	12.14	7.15	6.01	6.57
1000000	*79.56*	-	33.86	16.54	8.75	6.96	7.28
2000000	*167.10*	-	-	-	18.39	11.09	10.95
4000000	*350.17*	-	-	-	-	22.10	15.83
8000000	*732.28*	-	-	-	-	-	

Table 2. Sorting times of PSS.

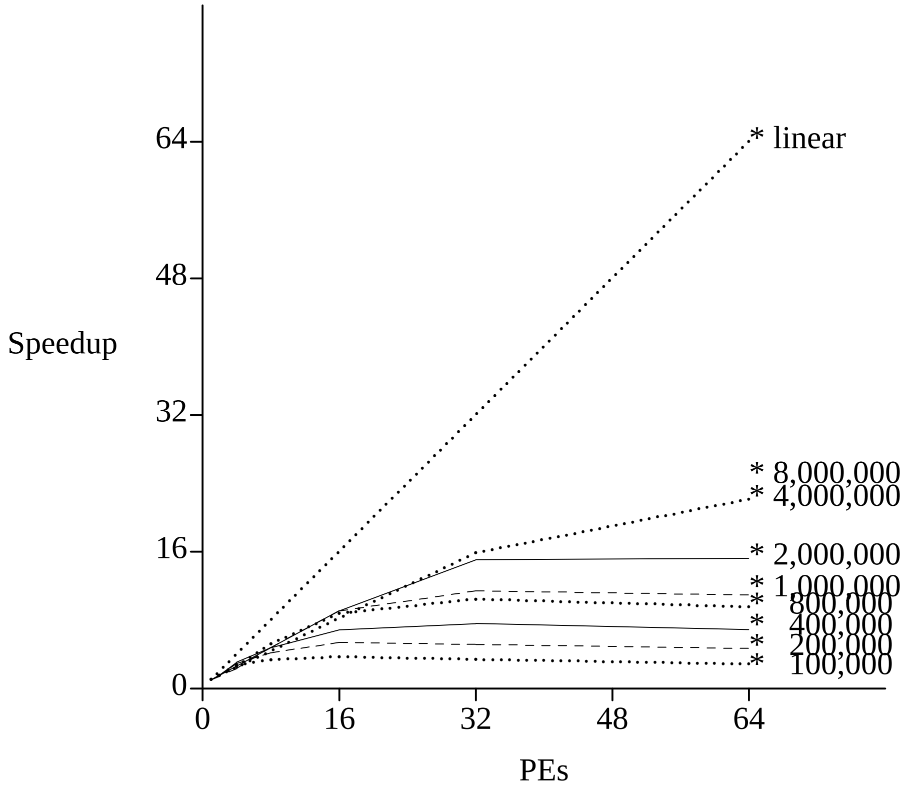

FIG. 3. Speedups of PSS.

Sizes	RDFAs					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	1.001	1.008	1.021	1.030	1.074	-
200000	1.002	1.003	1.012	1.032	1.043	-
400000	1.001	1.002	1.008	1.017	1.044	-
800000	-	1.002	1.005	1.017	1.026	1.062
1000000	-	1.001	1.004	1.010	1.021	1.047
2000000	-	-	-	1.009	1.016	1.045
4000000	-	-	-	-	1.011	1.026
8000000	-	-	-	-	-	1.017

Table 3. RDFAs of PSRS.

Sizes	RDFAs					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	1.342	1.373	2.029	2.096	2.861	-
200000	1.339	1.349	1.722	1.685	2.515	-
400000	1.351	1.403	1.485	2.100	2.425	-
800000	-	2.264	1.441	2.339	2.150	2.001
1000000	-	1.649	2.348	1.996	2.044	2.635
2000000	-	-	-	1.578	2.104	2.447
4000000	-	-	-	-	2.365	2.670
8000000	-	-	-	-	-	2.407

Table 4. RDFAs for PSS.

Data gathering in phase 1:

for $j = \log p, \log p - 1, ..., 1$ **do**
All nodes numbered $0^{\log p - j} 1 a_j$, where a_j is any
$(j - 1)$-bit binary number, send in parallel their respective data
accumulated from the previous steps to nodes $0^{\log p - j + 1} a_j$.

Data broadcasting in phase 2:

for $j = 1, 2, ..., \log p$ **do**
All nodes numbered $0^{\log p - j + 1} a_j$, where a_j is any
$(j - 1)$-bit binary number, send in parallel the received $p - 1$ pivots
to nodes $0^{\log p - j} 1 a_j$.

Data exchange operations in phase 3:

for $k = 1, 2, ..., \log p$ **do**
Each node i, such that $(i)_k = 1$, exchanges with node $\overline{i}^k$
all sublists $L_{i,j}$ and $L_{\overline{i}^k, \overline{j}^k}$,
for all $0 \le j \le p - 1$ such that $(j)_k = 0$.

FIG. 4. PSRS on a Hypercube.

Author biographies:

Hanmao Shi:

Hanmao Shi received his B.Sc. degree from the University of Science and Technology of China in 1988 and his M.Sc. degree in Computer Science from the University of Alberta in 1990. Currently, he is a Ph.D. candidate in Computer Science at the University of Waterloo. His research interests include parallel and distributed computing, database management, and artificial intelligence.

Jonathan Schaeffer:

Jonathan Schaeffer received his B.Sc. degree from the University of Toronto (1979), and M.Math (1980) and Ph.D. (1986) degrees in Computer Science from the University of Waterloo. Currently he is an Associate Professor at the Department of Computing Science, University of Alberta. His research interests include parallel and distributed computing and artificial intelligence.