Parallel Sorting by Overpartitioning

Hui Li and Kenneth C. Sevcik

Technical Report CSRI-295
February 1994

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Parallel Sorting by Overpartitioning *

Hui Li and Kenneth C. Sevcik

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A4
Email: {*hui*|*kcs*}@*csri.toronto.edu*

Abstract

A new approach to parallel sorting called *Parallel Sorting by OverPartitioning* (PSOP) is presented. The approach limits the communication cost by moving each element between processors at most once, and leads to good load balancing with high probability. The PSOP framework can be applied to both comparison and non-comparison sorts. Implementations on the KSR1 and Hector shared memory multiprocessors show that PSOP achieves nearly linear speedup and outperforms alternative approaches. An analytical model for PSOP has been developed that predicts the performance within 10% accuracy.
Key Words: Parallel Sorting, Load Balance, Overpartitioning, Oversampling, COMA and NUMA Multiprocessors.

1 Introduction

Sorting is an important symbolic application [4], and parallel algorithms for sorting have been studied intensively ever since Batcher [5] proposed the bitonic-sorting network. Although the theoretical community has devoted considerable effort to the design of parallel sorting algorithms [2, 3, 18, 27, 10, 26, 16, 23], empirical studies [12, 11, 7, 34, 25, 29, 35] started appearing only in the late 1980's when multiprocessors became widely available.

In a typical parallel sorting algorithm, each processor contains a portion of the list of n elements to be sorted. In a merge-based approach, each processor sorts the portion it holds initially by some sequential sorting algorithm, and then all the sorted portions are exchanged among all the processors and merged. Alternatively, in a quicksort-based approach, the unsorted list is partitioned into a number of progressively smaller sublists defined by selected pivots. Then, the processors employ a sequential sorting algorithm to sort the sublists for which they are responsible. The merging of the sorted portions of different processors in the merge-based approach and the partitioning into sublists in the quicksort-based approach can be achieved either in one step or many steps.[1] Existing parallel sorting algorithms can therefore be classified as *single-step* or *multi-step* (as in Table 1).

*This research has been supported by research grants from the Natural Sciences and Engineering Research Council of Canada and from the Information Technology Research Centre of Ontario.

[1] A single-step algorithm will merge the portions in all the processors or partition the sublists and assign to all the processors in one step.

	Single-step	Multi-step
merge-based	quickmerge [25] PSRS [30, 21]	Batcher's bitonic sort [5] parallel merge sort [9, 10] smoothsort [24] Nassimi and Sahni's sort [22] column sort [18] Tridgell and Brent's sort [32] snakesort [6]
quicksort-based	parallel quicksort [7, 20, 13] parallel sample sort [15, 8]	flashsort [27] B-flashsort [14] hyperquicksort [34, 25] Kalé and Krishnan sort [17]

Table 1: Classification of parallel sorting algorithms

The performance of a parallel sorting algorithm is determined by the communication cost and the degree of load balancing. Single-step algorithms incur low communication cost, because they move each element at most once. However, single-step algorithms often have poor load balancing because it is difficult to derive nearly equal sized sublists in the absence of global information and single-step algorithms cannot refine the sublists once they are derived. On the other hand, multi-step algorithms typically incur higher communication cost because elements may be moved several times between processors.

A single-step parallel sort can perform well if an effective way to balance the load can be found. In this paper, we propose a new approach to single-step parallel sorting, *Parallel Sorting by OverPartitioning* (PSOP). PSOP provides good load balancing with high probability. On p processors, PSOP uses $pk-1$ pivots to divide the work into k times as many sublists as there are processors. These sublists are ordered in decreasing size, and dynamically scheduled to be sorted by a "work crew" model. The parameter k is called the *overpartitioning ratio.* For sufficiently large k, the maximum size of any sublist is small enough to ensure good load balancing. In our experiments, loads on different processors differ by less than 5%. Since PSOP is a single-step approach, it involves moving each element between processors at most once.

We describe a canonical single-step parallel sort in Section 2. We then present the PSOP approach in Section 3 and show that it guarantees good load balancing with high probability. We demonstrate that the approach is general by using it with quicksort and radix sort in Section 4. The resulting algorithms are called *Parallel Quicksort by OverPartitioning* (PQOP) and *Parallel Radix sort by OverPartitioning* (PROP) respectively. We have implemented these algorithms on two shared memory multiprocessors, namely the Kendall Square Research KSR1 [28], and the University of Toronto Hector system [33]. The experiments in Section 5 show that these algorithms achieve nearly linear speedup and perform better than (our implementation of) *Parallel Sorting by Regular Sampling* (PSRS) [30, 21], *Parallel Sample Sort* (PSS) [15, 8], and Parallel Radix Sort [31] on these machines. The performance obtained is very close to that predicted by the analytical model developed in Section 4. The execution times of these algorithms on the KSR1 are comparable to and sometimes significantly better than other algorithms on various multiprocessors including the iPSC/860 and CM-5.

2 The General Structure of a Single-step Algorithm

A canonical single-step parallel sort on p processors to sort a given list ℓ of unsorted n keys typically involves four phases:

1. Processor i has a sorted or an unsorted portion ℓ_i of ℓ;
2. Some $q-1$ pivots are selected from ℓ (in "some" way), and are sorted (if necessary);
3. On each processor i, ℓ_i is divided into pieces $\langle \ell_{i1}, \ldots, \ell_{iq} \rangle$ using the $q-1$ pivots, so that ℓ_{ij} is the set of elements on processor i that are between pivots $j-i$ and j. A sublist S_j is then the union of ℓ_{ij}'s with i ranging over all processors.
4. Process "some" sublists in each processor. If sorting was not done in Phase 1, each processor sorts the sublists assigned to it. Otherwise, the processor simply merges the pieces of each of the sublists assigned to it.

2.1 Load Balance Metrics

Since the sizes of sublists S_j may vary, the load balancing in Phase 4 is critical to the performance of single-step algorithms. Two metrics can be used to measure the load balancing: (i) the sublist expansion [8] and (ii) the load expansion. The *sublist expansion* is defined as the ratio of the maximum sublist size to the mean sublist size:

$$\frac{\max_{j=1,q} |S_j|}{\sum_{j=1}^{q} |S_j|/q}$$

where $|S_j|$ is the size of the sublist S_j. The *load expansion* is defined as the ratio of the time spent in sorting sublists on the most heavily loaded processor to the mean time spent over all processors. If $f(|S_j|)$ is the cost function of sorting S_j, then the load expansion is

$$\frac{\max_{i=1,p}[\sum_{S_j \text{ sorted by } i} f(|S_j|)]}{\sum_{j=1}^{q} f(|S_j|)/p}$$

2.2 Examples of Single-step Sorting

We can obtain different flavors of single-step algorithms based on whether the ℓ_i's are sorted in Phase 1, and the way the pivots are picked in Phase 2.

- *Parallel Sorting by Regular Sampling* (PSRS) [30, 21] sorts the portions in Phase 1 and uses *regular sampling* to select pivots. It picks $p-1$ pivots by first choosing p equally spaced candidates from each portion, and then selecting $p-1$ equally spaced pivots from the sorted p^2 candidates. The regular sampling guarantees to balance the load among processors within a factor of two of optimal in theory, and within a few percent of optimal in practice.
- *Parallel Sample Sort* (PSS) [15, 8] does not sort the portions in Phase 1 and uses *oversampling* to select pivots. It picks $p-1$ pivots by randomly choosing ps candidates from the entire list ℓ, where s is the *oversampling ratio*, and then selecting $p-1$ equally spaced pivots from the sorted candidates. A larger oversampling ratio results in better load balancing, but increases the cost of selecting pivots.

3 The Parallel Sorting by Overpartitioning (PSOP) Approach

In this section, we present the PSOP approach on scalable shared memory multiprocessors. In a multiprocessor of this type, each processor typically has one memory module directly connected to it, but all processors can access all memory modules. The processors and memory modules may be connected in a manner so that the access times to various memory locations fall into several classes over a range of magnitudes.

3.1 Overview of the PSOP Approach

The PSOP approach follows the four phase canonical form of single-step sort. These four phases are described in detail as follows.

1. Initially, processor i has ℓ_i, a portion of size n/p of the unsorted list ℓ.

2. *Selecting Pivots.* A sample of pks candidates are randomly picked from the list, where s is the oversampling ratio, and k is the overpartitioning ratio. Each processor picks sk candidates and passes them to a designated processor. These candidates are sorted, and then $pk-1$ pivots are selected by taking $s^{th}, 2s^{th}, \ldots, (pk-1)^{th}$ candidates from the sample. The selected pivots, $d_1, d_2, \ldots, d_{pk-1}$ are made available to all the processors.

3. *Partitioning.* Since the pivots have been sorted, each processor performs binary partitioning on its local portion. Processor i first uses the middle pivot, $d_{(pk-1)/2}$, to partition ℓ_i into ℓ_i^{left} and ℓ_i^{right}, then recursively uses the left pivots to partition ℓ_i^{left} and the right pivots to partition ℓ_i^{right} until all ℓ_{ij} are identified. A sublist S_j is the union of ℓ_{ij} with i ranging over all processors:

   $$S_j = \cup_{i=1}^{p} \ell_{ij}$$

4. *Building a task queue and sorting sublists.* Let $T(S_j)$ denote the task of sorting S_j. The size of each sublist can be computed :

   $$|S_j| = \sum_{i=1}^{p} |\ell_{ij}|$$

   Also, the starting position of sublist S_j in the final sorted array can be calculated:

   $$\sigma_j = 1 + \sum_{h=1}^{j-1} |S_h|$$

   A task queue is built with the tasks ordered from largest sublists size to smallest. Each processor repeatedly takes one task, $T(S_j)$, at a time from the queue. It processes the task by (a) copying the p parts of the sublist, $\langle \ell_{1j}, \ldots, \ell_{pj} \rangle$, into the final array at positions σ_j to $\sigma_j + |S_j| - 1$, and (b) applying a sequential sort to the elements in that range. This process continues until the task queue is empty.

These four phases define a general framework for parallel sorting algorithms with overpartitioning, where any sequential sorting algorithm can be used by each processor in the last phase. Figure 1 illustrates the PSOP approach.

Since the tasks in the queue are in decreasing order of sublist size, in Phase 4 every processor gets larger tasks at the beginning, and smaller tasks towards the end. Overpartitioning leads to a sufficient number of tasks of varying size such that PSOP achieves good load balancing.

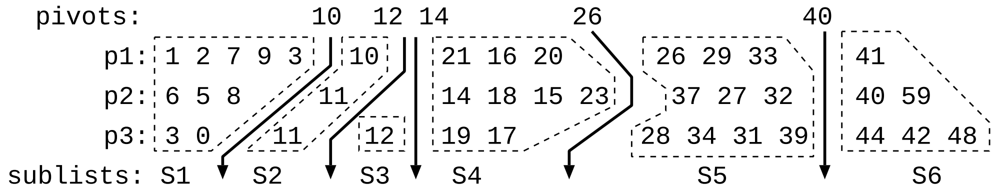
Phase 1. Initially, a list to be sorted is distributed among processors ($p = 3, n = 39$).

p_1 :	10	21	1	2	26	29	7	33	9	20	3	16	41
p_2 :	6	40	59	5	14	37	18	8	27	15	32	23	11
p_3 :	11	44	12	3	19	28	34	31	42	17	48	0	9

Phase 2. Randomly pick up psk candidates ($s = 1, k = 2$), sort them, and select $(pk - 1)$ from them as pivots.

psk candidates :	10	26	40	14	44	12
sorted :	10	12	14	26	40	44
$(pk - 1)$ pivots :	10	12	14	26	40	

Phase 3. Each processor uses the pivots to partition its own portion locally.



Phase 4. Compute the sublist sizes and their final positions in the sorted list and build a task queue. When a processor is ready, it gets a task from the queue and sorts the sublist of the task until the queue is empty.

sublists :	S_1	S_2	S_3	S_4	S_5	S_6
sizes :	10	3	1	9	10	6
final positions :	0	10	13	14	23	33

task queue : $T(S_1), T(S_5), T(S_4), T(S_6), T(S_2), T(S_3)$

Figure 1: An example using PSOP

3.2 Load Balancing

There are two parameters in PSOP that influence the degree of load balancing, namely, the oversampling ratio and the overpartitioning ratio. The oversampling ratio governs the variance of sublists sizes. A larger oversampling ratio produces sublists with a smaller variability in size. The overpartitioning ratio determines the total number of sublists and the average sublist size. A larger overpartitioning ratio gives more sublists but with smaller average and maximum sublist sizes, performing better load balancing in Phase 4. However, the time spent in Phase 1 increases with the oversampling ratio and the overpartitioning ratio. Therefore, it is necessary to choose the appropriate values for the two parameters in order to achieve good performance.

3.2.1 Maximum Sublist Size

Blelloch et al. [8] have shown that the sublist sizes can be reduced by increasing the oversampling ratio. They have proved that for $(p-1)$ pivots and an oversampling ratio s, the probability that the sublist expansion is greater than some factor $\alpha > 1$ is

$$Pr[\max_{i=1,p} |S_i| > \alpha(n/p)] \leq ne^{-(1-1/\alpha)^2 \alpha s/2}$$

The following theorem establishes a relationship between the maximum sublist size and the overpartitioning ratio as follows:

Theorem: *For an unsorted list of size n, $(pk-1)$ pivots (with $k \geq 2$) partition the list into pk sublists such that the size of the maximum sublist is less than or equal to n/p with probability at least*

$$1 - 2p(1 - \frac{1}{2p})^{pk}$$

.

Proof: The final sorted list can be viewed as $2p$ segments of size $\frac{n}{2p}$. If every segment contains at least one pivot, then $\max_{j=1,q} |S_j| \leq \frac{n}{p}$, no matter where the pivots are located within their respective segments.

Consider one of the segments. Since the pivots are chosen randomly, the probability that a specific pivot is not in the segment is $(1-\frac{1}{2p})$. Since the $q = pk-1$ pivots are selected independently, the probability that none of the pivots are in the segment is

$$(1 - \frac{1}{2p})^q$$

Therefore, even assuming mutual exclusion, the probability of at least one of the $2p$ segments not containing any pivots cannot exceed

$$2p(1 - \frac{1}{2p})^q$$

In other words, every segment contains at least one pivot with probability at least

$$1 - 2p(1 - \frac{1}{2p})^q$$

□

Figure 2 shows the lower bounds on the probabilities that the maximum sublist size is less than n/p for overpartitioning ratios of $k = (q+1)/p = 2\log_2(p)$ and $k = 3\log_2(p)$. When k grows in proportion to $\log_2(p)$ the lower bound on probability approaches one as p increases.

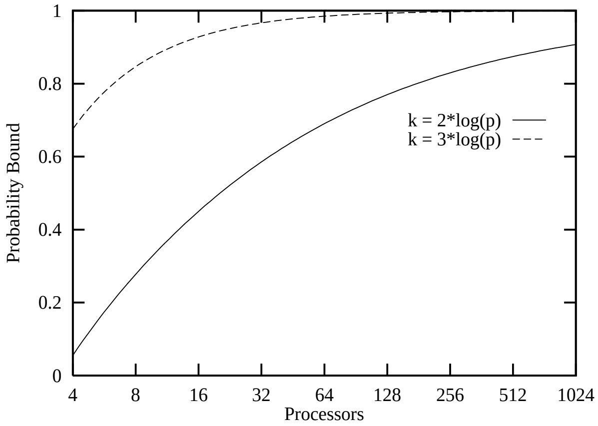


Figure 2: Lower bounds on Prob[$\max_i |S_i| \leq n/p$] as a function of the number of processors for two choices of overpartitioning functions

3.2.2 Tradeoff between s and k

To illustrate the effects of the overpartitioning ratio k and the oversampling ratio s on load balancing, we performed 100 tests for each combination of the parameters p, s, and k. For each test, we randomly generated one million ($n = 2^{20}$) 32-bit integers and used the combination of parameters to perform the partitioning. These integers are uniformly distributed in $[0, 2^{31}]$. The results of this experiment are shown in Figures 3, 4, and 5.

Oversampling. Figure 3 shows the effect of oversampling on the sublist sizes.

- Sublist expansion decreases as s increases. However, average sublist expansion for $p \geq 64$ is still around 1.3 even when s is as high as 128. Thus, oversampling can reduce load imbalance only to a certain degree.
- Sublist expansion increases as the number of processors increases.
- Small values of s make sublist expansions sufficiently small for our purposes. The sublist expansions are less than 3 for $s = 4$ for as many as 128 processors.

Overpartitioning. The effect of overpartitioning on load balancing is shown in Figures 4 where the oversampling ratio s was set to 3 so that the sublist expansion is sufficiently small.

- In Figure 4 (a), load expansion decreases dramatically as k increases. For p from 4 to 128, the average value of load expansion is only about 1.05 when k is as small as 5.
- Figure 4 (b) shows that average load expansion also depends on the number of processors p. When $k = \log_2(p)$, the load expansions decreases slightly with p. This is consistent with the Theorem and suggests that $k = \log_2(p)$ is suitable for balancing the load.

Overpartitioning with Oversampling. The effect of oversampling in overpartitioned sorting is shown in Figure 5, where the overpartitioning ratio is set to $\log_2(p)$. Without oversampling (i.e., $s = 1$), load expansions can be as high as 1.3. A larger oversampling ratio reduces the sublist expansion, and thus is expected to reduce the load expansion as well. However, when the oversampling ratio is higher than 4, the load expansion slightly increases because larger oversampling ratios generates nearly equal size sublists, making dynamic load balancing difficult.

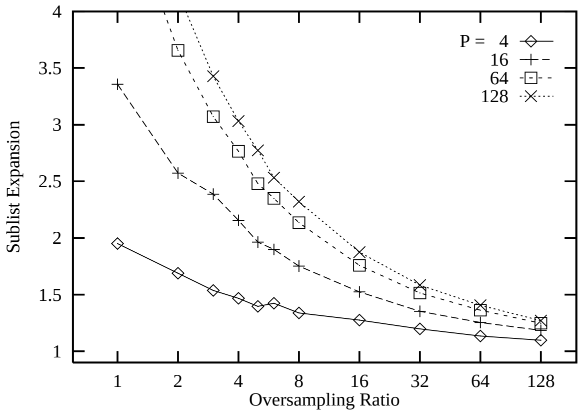


Figure 3: Ratio of maximum sublist size to mean sublist size as a function of the oversampling ratio for various number of processors.

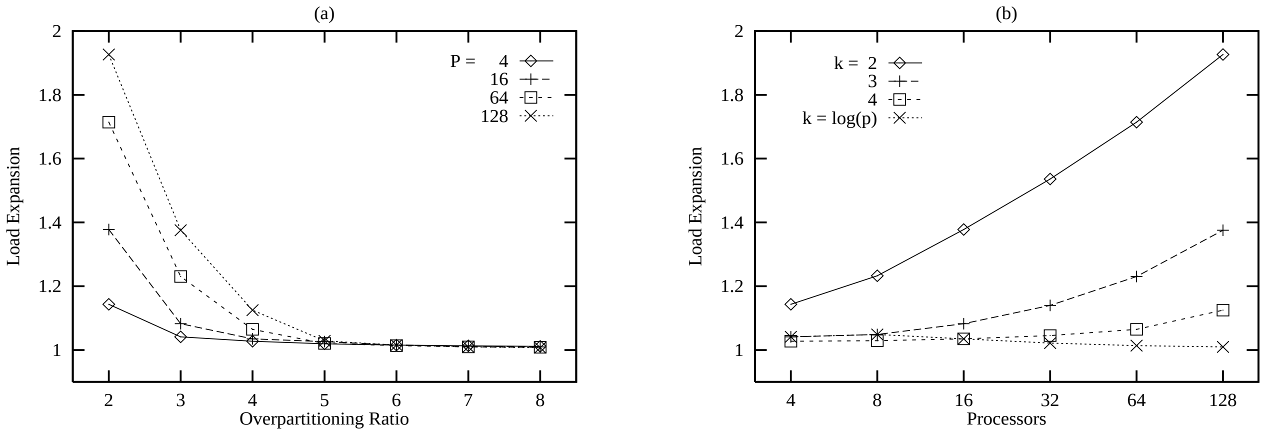


Figure 4: Ratio of maximum load to mean load: (a) as a function of overpartitioning ratios and (b) as a function of number of processors

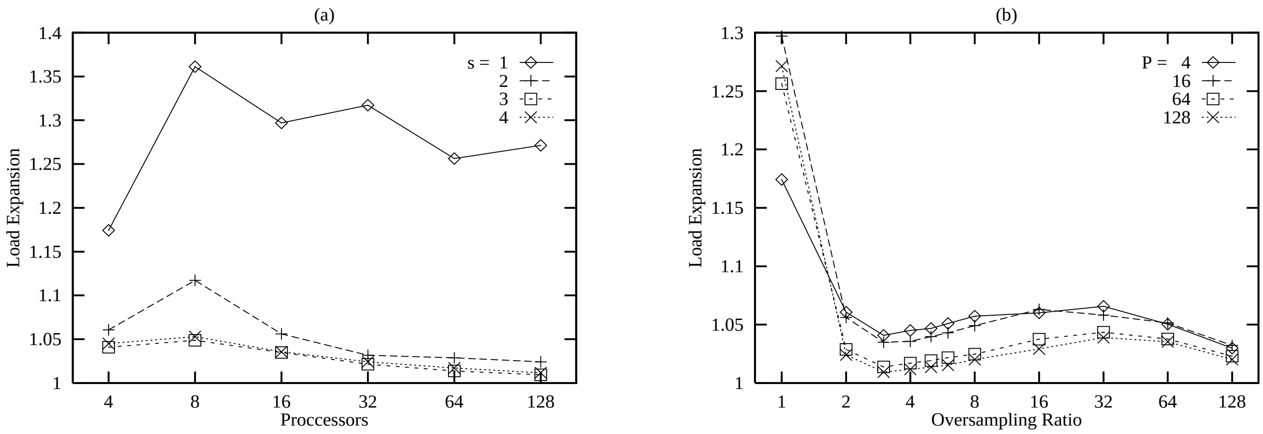


Figure 5: Oversampling in overpartitioning

	Meaning
C	constant factor in quicksort
Q	unit cost in quicksort
R_b	unit cost in b-bit radix sort
W	number of integers per cache line
t_r	mean time to access remote memory

Table 2: Parameters for analysis of PQOP and PROP

4 PSOP with Two Sequential Sorts

In this section, we specify and analyze two versions of PSOP that use a comparison based sort (quicksort) and a non-comparison based sort (radix sort) respectively. The resulting algorithms are called *Parallel Quicksort by Overpartitioning (PQOP)* and *Parallel Radix sort by Overpartitioning (PROP)* respectively. Experimental results with implementations of these are presented in Section 5.

Table 2 lists the parameters involved in the complexity analysis of these algorithms. In shared memory multiprocessors with non-uniform memory access time, remote memory accesses must be distinguished from local memory accesses in doing performance analysis. The value of Q is the ratio of the total execution time for quicksort to the number of comparisons (in CPU cycles). The value of t_r, the mean time to access remote memory, depends not only on how many processors are allocated, but also on which of the processors are used.

4.1 Parallel Quicksort by Overpartitioning (PQOP)

In PQOP, we use pivoting just as in sequential quicksort and binary partitioning in Phases 2 and 3 of PSOP. The last phase of PSOP uses a sequential quicksort to sort the sublists.

Selecting Pivots. Each processor randomly selects sk candidates and puts them into an array, requiring $t_r sk/W$ for remote memory access. One of the processors then sorts the candidates by quicksort and determines the $(pk-1)$ pivots.[2] It takes $t_r psk/W$ in accessing the psk candidates from other processors and $CQpsk\log(psk)$ time in sorting them. The time for selecting pivots is therefore:

$$T_{pivot} = CQpsk\log_2(psk) + t_r\frac{(p+1)sk}{W}$$

Partitioning. All processors except one need to read the $pk-1$ pivots remotely in about $(t_r pk/W)$ time. Since a recursive binary partitioning is used, the depth of the recursion is $\log(pk)$, resulting in the partitioning time $Q\frac{n}{p}\log(pk)$. So the time for this phase is:

$$T_{part} = t_r\frac{pk}{W} + Q\frac{n}{p}\log_2(pk)$$

Building the Task Queue. Since each sublist is formed by the union of the corresponding parts on all processors, computing the size of a sublist requires p remote memory accesses and p

[2]In our current implementation, one processor is used to do this sorting, but this phase could be parallelized when p is large.

operations. Each processor needs to compute k sublist sizes independently in time Qpk with remote memory accesses in time $t_r pk/W$. The task queue is built by sorting the tasks in decreasing sublist sizes.[2] The time to construct the task queue is therefore:

$$T_{queue} = t_r \frac{(p+1)k}{W} + Qpk + CQpk\log_2(pk)$$

Scheduling Local Sorting. Each processor gets a task from the task queue and copies the sublist of the task into the destination array at its final location. Thus, the total time for remote memory copying on all the processors is $t_r n/W$. To find the total time of sorting all the sublists, denoted by T^{all}_{sort}, we observe that the partitioning in Phase 2 and the sorting in Phase 4 perform the work equivalent to quicksort, thus

$$T^{all}_{sort} = CQn\log_2(n) - Qn\log_2(pk)$$

By choosing appropriate values for k and s to achieve good load balancing, the time in this phase on every processor should be:

$$T_{sched} = (t_r \frac{n}{W} + T^{all}_{sort})/p$$

Therefore, the total execution time of PQOP is:

$$T_{PQOP} = T_{pivot} + T_{queue} + T_{part} + T_{sched}$$

or

$$\begin{aligned} T_{PQOP} \leq\ & CQn\log_2(n)/p + CQpsk\log_2(psk) + \\ & CQpk\log_2(pk) + 2t_r psk/W + t_r n/(pW) \end{aligned}$$

4.2 Parallel Radix Sort by Overpartitioning (PROP)

We assume that the list to be sorted contains 32-bit positive integers. In PROP, we use a radix partitioning in Phase 3 and radix sort in Phase 4. Phase 2 is not needed because the radix partitioning is based on the first H-bits.

Radix Partitioning. Each processor performs radix partitioning independently on a portion of the list as follows:

1. Build a histogram of the first H-bits of the locally held elements;
2. Pass over the keys and move each one to its appropriate positions in a second array as determined by the histogram counts within the processor.

The time to do radix partitioning is therefore:

$$T_{part} = R_H \frac{n}{p}$$

Building a Task Queue. Each value of the first H-bits defines a sublist that corresponds to a task, so the total number of tasks is 2^{H-1}. The size of the sublist is the number of occurrences of each histogram entry over all the processors. These tasks are sorted by quicksort in decreasing order of the sublist size in time $CQ2^{H-1}\log_2(2^{H-1})$, or $CQ(H-1)2^{H-1}$. So, the complexity of this phase is similar to that in PQOP.

$$T_{queue} = t_r \frac{(p+1)k}{W} + Qpk + CQ(H-1)2^{H-1}$$

	t_l	t_r^1	t_r^2
Hector	10	19	29
KSR1	18	175	530

Table 3: Memory access times (in CPU cycle)

Scheduling Local Sorting. This component is similar to that in PQOP except that radix sort is applied to sort each sublist. Since all the numbers in the same sublist have the same value in their first H-bits, radix sort is applied only to the last $(32 - H)$ bits. If each pass in the radix sort performs on B bits, then the number of passes should be $\lceil \frac{32-H}{B} \rceil$. The time for this phase is thus:

$$T_{sched} = t_r \frac{n}{pW} + R_B \lceil \frac{32 - H}{B} \rceil \frac{n}{p}$$

Therefore, the total time for PROP is:

$$T_{PROP} = T_{part} + T_{queue} + T_{sched}$$

or

$$\begin{aligned} T_{PROP} \quad \leq \quad & R_B \frac{n}{p} + R_B \lceil \frac{32 - H}{B} \rceil \frac{n}{p} + \\ & CQ(H - 1)2^{H-1} + 2t_r pk/W + t_r n/(pW) \end{aligned}$$

Since radix sort has a random access pattern, sorting a sublist of size larger than the cache size has a low cache hit ratio, thus degrading the performance. We thus choose a value for H such that most sublists have size smaller than the cache size.

5 Experimental Results

The PQOP and PROP algorithms have been implemented on both the KSR1 [28] and Hector [33] multiprocessors. The measured timing does not include the times for initializing the list and confirming the correct order of the final list. The cost of handling page faults is not included in the timing because the current KSR OS does not handle page faults efficiently. We believe that this cost will be negligible with the future version of the KSR OS.

5.1 The KSR1 and Hector Multiprocessors

Both the KSR1 and Hector have a hierarchical ring interconnection network. The KSR1 [28] is a scalable Cache-Only-Memory Architecture (COMA) system composed of a hierarchy of rings. The lowest level, ring:0, consists of 32 processor cells and two cells for routing to the higher level ring - ring:1. The system we used consists of 64 cells on two ring:0's connected by a ring:1. Each processor has a 256K data subcache, a 256K instruction subcache, and a 32M cache memory. The subcache and cache memory on the KSR1 are similar to cache and local memory on other multiprocessors (e.g., Stanford DASH [19], MIT Alewife [1], and University of Toronto Hector). Memory access times are 18 cycles for a subcache line of 64-bytes on local cache, 175 cycles for a subpage of 128-bytes on ring:0, and 530 cycles on ring:1.

n	100K	256K	512K	800K	1M	8M
KSR1	2.05	5.60	12.00	19.51	25.46	234.26
Hector	2.70	7.34	16.40	25.67		

Table 4: Sequential execution times

	KSR1	Hector
C	1.32	1.32
Q	18.06 (cycles)	18.78 cycles
W	32	4
t_r	depends on p	depends on p

Table 5: Parameter values

Hector [33] is a scalable NUMA shared memory multiprocessor system consisting of sets of processor-memory pairs connected by buses to form a station, and several stations connected by local rings, and several local rings connected by a global ring. Hector provides a single global physical address space; each memory module contains one portion of the global memory. On the current prototype of 16 processors, the memory access time for a 16-byte cache line is 10 cycles for the local memory, 19 cycles on the bus, and 29 cycles on the local ring.

Table 3 shows the memory access times in CPU cycles on the KSR1 and Hector.

5.2 Performance of PQOP

Speedup. The speedups are calculated based on the execution times of sequential quicksort given in Table 4. When a data size is too large to fit in the local memory, we use $CQn\log_2(n)$ to estimate the sequential execution time.

Figure 6 (a) shows speedups on the KSR1. The speedup for sorting 100 K integers gradually stops increasing, resulting in an execution time of 69 milliseconds and 30-fold speedup on 64 processors. When a larger number of elements are sorted, the speedups become close to linear. On 64 processors, sorting one million integers takes only 457 milliseconds with a speedup of 56. The speedup curve of sorting 800 K integers on Hector is shown in Figure 6 (b).

The complexity analysis of Section 4 is used for predicting the speedup and execution time. Table 5 shows the values of the parameters where Q is empirically determined, and t_r depends on how many processors and which processors are used. The predicted speedups shown match the measured ones within 10% accuracy in Figures 6.

Since the experimental results on Hector are similar to those on the KSR1, we present only the experimental results on the KSR1 in the rest of the paper.

The effect of oversampling. We measured the time for selecting pivots and the time for the load imbalance in the last phase on the KSR1 when the oversampling ratio s varied from 1 to 8 for sorting one million integers on 50 processors with the overpartitioning ratio $k = 5$ (i.e, k=$\log_2(p)$). Figure 7 shows these times as a percentage of the minimum total execution time (which is achieved when $s = 3$ and $k = 6$). The overhead of selecting pivots increases linearly with s, but load imbalance is minimized around $s = 5$. The minimum total overhead is observed to be when

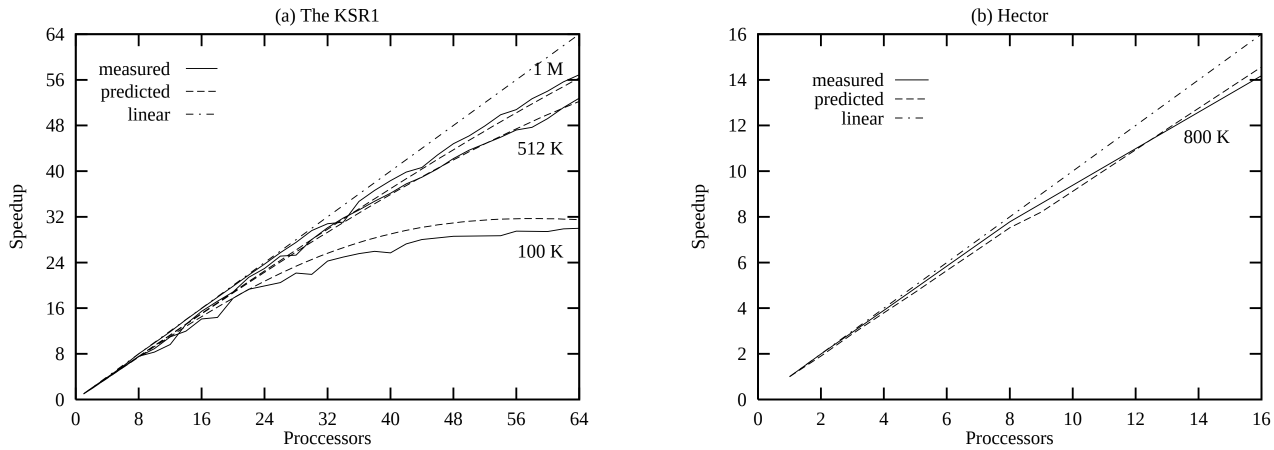


Figure 6: Speedup of PQOP

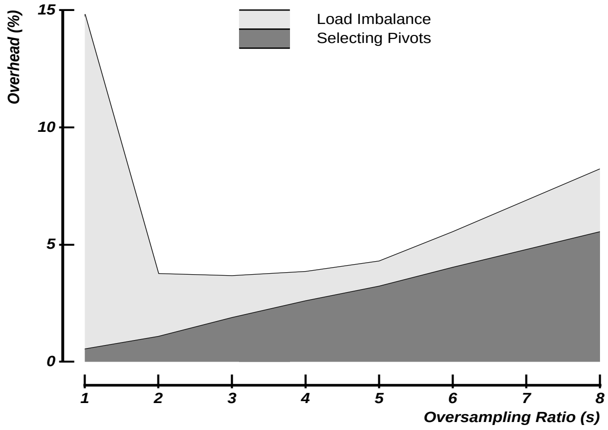


Figure 7: The effect of oversampling

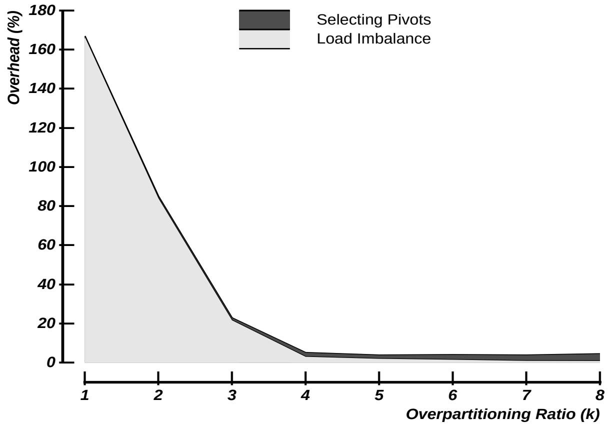


Figure 8: The effect of overpartitioning

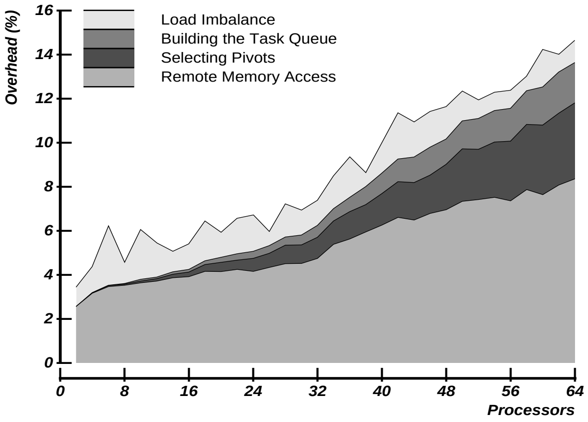


Figure 9: Overhead analysis of PQOP for the KSR1

$s = 3$.

The effect of overpartitioning. Again on the KSR1, we varied overpartitioning ratio k for sorting one million integers on 50 processors with $s = 3$. Figure 8 gives the time for selecting pivots and the time for the load imbalance in the last phase as a percentage of the execution time for the case $s = 3$ and $k = 5$. Without overpartitioning ($k = 1$), the load imbalance is as high as 170%. With increasing k, the load imbalance decreases dramatically, but the overhead for selecting pivots increases slowly. The total overhead of selecting pivots and load imbalance remains less than 5% when k is at least 4.

Overhead analysis. Figure 9 gives the ratio of the overhead in each phase to the total execution time for one million integers where the overpartitioning ratio is $k = \log_2(p)$ and the oversampling ratio is $s = 3$.

- Load imbalance is about 3% and decreases slightly with the number of processors. This indicates that overpartitioning ratio of $\log_2(p)$ leads to good load balancing.
- Times for selecting pivots and building the task queue grow with the number of processors because their time complexities depend on p.
- The total number of remote memory accesses is $\frac{n}{W}$ in the worst case and $\frac{n}{W}\frac{p-1}{p}$ on average.

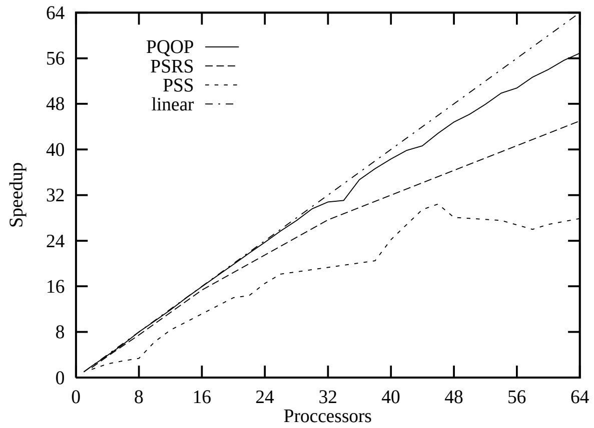


Figure 10: Speedups for sorting one million integers.

Algorithms	Machine	Processors	Time (Sec.)
PQOP	KSR1	64	3.78
PQOP	KSR1	32	7.49
PSRS [21]	TC-2000	64	4.29
PSRS [21]	iPSC/860	64	2.46
K&K [17]	iPSC/860	64	3.87
K&K [17]	nCUBE/2	64	12.30
T&B [32]	CM-5	32	5.48
T&B [32]	AP-1000	32	7.62

Table 6: Execution time for sorting 8 million integers.

Thus, as the number of processors increases, the fraction of the time involving remote memory accesses increases. When $p > 32$, some portion of remote memory accesses are across ring:0 with a latency is 530 cycles on the KSR1 shown in Table 3. The number of accesses across ring:0 also depends on the number of processors: $\frac{n}{W}\frac{p-32}{p}$. Thus the time for remote memory increases more quickly when $p > 32$.

Comparison. Two other parallel sorting algorithms PSS and PSRS were also implemented on the KSR1 in order to compare their performance with PQOP. It has been shown that PSS performs well on the SIMD machine CM-2 [8], and PSRS achieves good performance on a variety of MIMD machines [21]. Figure 10 shows the speedup curves for the three algorithms for sorting one million integers. Although PSS has low communication overhead, it does not balance load well even with an oversampling ratio as large as 32. This results in relatively poor speedups. PSRS minimizes the communication cost and partitions the list into sublists evenly. The speedup of PSRS is much higher than that of PSS, but degrades with the number of processors because:

- the pivoting cost increases with the number of processors; and
- merging used in the last phase of PSRS is not as efficient as binary partitioning based on the data we collected from the KSR1 hardware monitor. With more processors, the time required

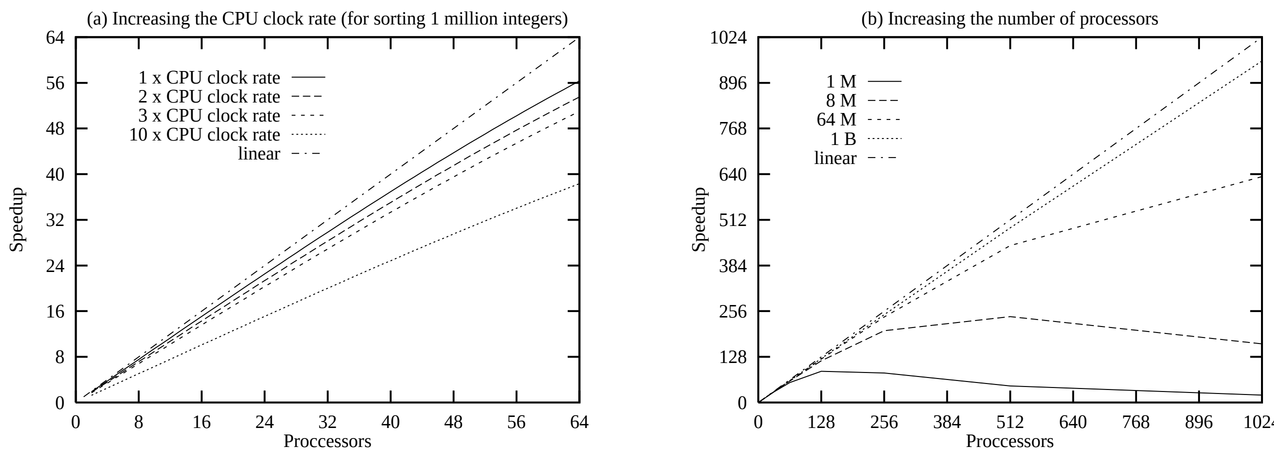


Figure 11: Performance Impact

for merging increases in percentage of the total execution time and thus accounts for a major portion of the total execution time.

PQOP outperforms both PSS and PSRS on the KSR1.

Table 6 gives the execution times and speedups of sorting 8 million integers by various algorithms across several machines. With the consideration of the CPU clock rates (20, 25, 32, to 50 among the KSR1, AP1000, CM-5, and iPSC/860 respectively), the performance of PQOP on the KSR1 is competitive.

Impact of number of processors and their speed. Since the complexity analysis of PQOP accurately predicts performance, we use it to illustrate the performance impact when the CPU clock rate and the number of processors increase. Figure 11 (a) predicts the speedups for sorting one million integers when the CPU clock rate increases. Figure 11 (b) shows that with the scaling of the system, the problem size needs to be scaled up as well to retain good speedup.

5.3 Performance of PROP

Speedup. Figure 12 shows the speedup curves for Parallel Radix Sort and PROP on the KSR1 in sorting one million randomly generated integers. PROP on one processor takes only 11.25 seconds, much faster than the sequential execution of radix sort (19.22 seconds). This difference results from the higher cache hit ratio in PROP due to overpartitioning. Thus, the speedups in Figure 12 are relative to the execution time of PROP on one processor.

PROP achieves nearly linear speedups because it balances the load well by overpartitioning and minimizes the remote memory accesses. Parallel Radix Sort is a multi-step algorithm. It permutes the keys based on the value of the R bits in each step and moves from the least significant to the most significant with each pass. Therefore, the number of remote memory accesses in each step is close to the total number of remote memory accesses in PROP. In our implementation, R was set to 11, and thus the number of steps was 3. The speedup of Parallel Radix Sort is sub-linear because of the high remote memory access overhead and the low cache hit ratio.

Overhead Analysis. Since PROP does not involve pivoting, the main overhead is in the cost of remote memory access. Figure 13 shows that the overhead of remote memory accesses increases

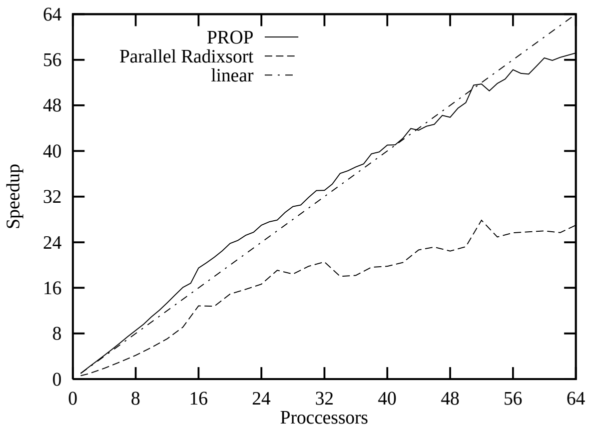


Figure 12: Speedups of Parallel Radix Sort and PROP for sorting one million 32-bit integers

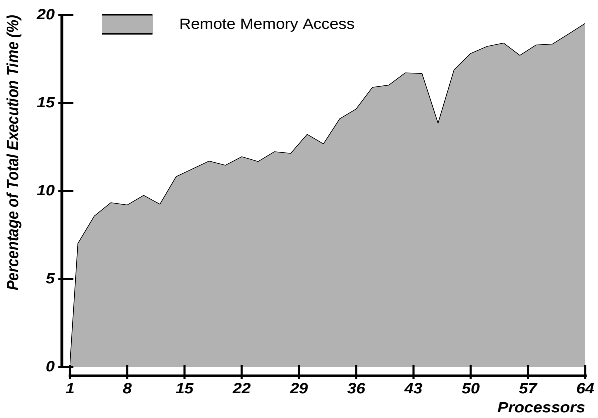


Figure 13: Overhead of Remote Memory Accesses

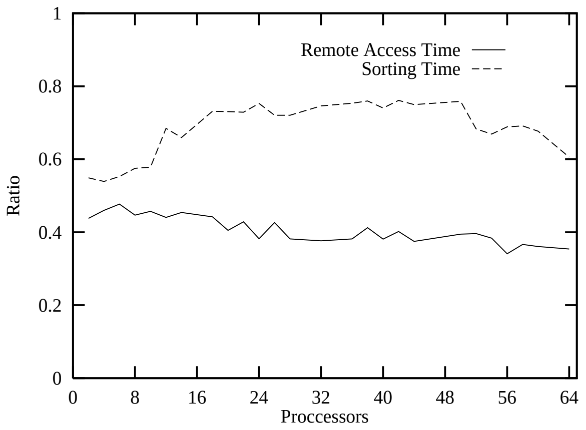


Figure 14: Time Ratio (PROP/Parallel Radix Sort)

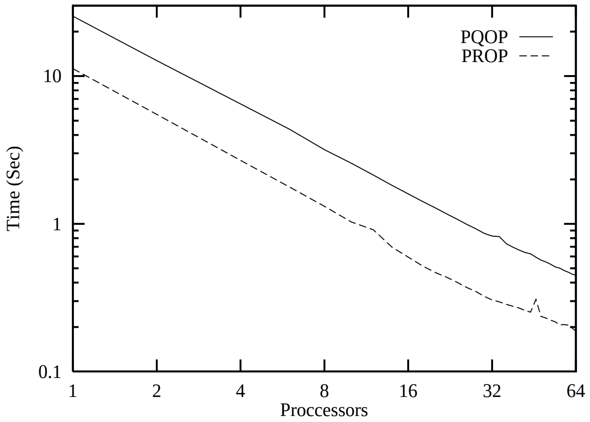


Figure 15: Comparison of PQOP and PROP

Machine	KSR1	CM-5
Algorithm	PROP	Parallel Radix Sort [31]
Time (sec)	10.02	16.7

Table 7: Execution times for sorting 64 million 32-bit integers on 64 processors.

with the number of processors up to about 20% on 64 processors. However, the speedup is not reduced substantially, because of the increase in the cache hit ratio.

Figure 14 gives the ratios of sorting time and remote memory access time of PROP to those of Parallel Radix Sort. The sorting time in PROP includes the time for the radix partitioning in Phase 3 and the time for sequential sorting in Phase 4. PROP reduces the sorting time to 55–75% of that in Parallel Radix Sort. Since PROP is a single-step sort, its remote memory access time is only 35–45% of that of Parallel Radix Sort.

Comparison. Figure 15 gives the execution times of PQOP and PROP for sorting one million integers. Since radix sort outperforms quicksort on randomly generated 32-bit integers with uniform distribution, the execution time in PROP is about 40% of that in PQOP.

Table 7 shows that PROP on the KSR1 outperforms Parallel Radix Sorting on the CM-5 with 64 processors for sorting 64 million uniformly distributed 32-bit integers.

6 Conclusions

Load imbalance and communication cost are two critical factors in determining the performance of a parallel sorting algorithm on multiprocessors. PSOP, the overpartitioning approach to parallel sorting presented in this paper, partitions the sublists in a single-step and thus limits the communication cost. Overpartitioning is a very effective technique and leads to good load balancing with high probability. PQOP and PROP are versions of PSOP based on quicksort (a comparison sort) and radix sort (a non-comparison sort) respectively. Our implementations of PQOP and PROP on the KSR1 and Hector shared memory multiprocessors show that:

- PSOP achieves nearly linear speedup and the performance obtained is very close to that predicted by the analytical model.
- PSOP leads to good load balancing. Loads on the processors differ by less than 5%.
- PSOP is shown to outperform single-step parallel comparison sorts PSRS and PSS and multi-step Parallel Radix Sort.
- The execution time on the KSR1 is comparable to, and sometimes better than parallel sorting implemented on several other multiprocessors including the iPSC/860 and CM-5.

Currently, we are applying PSOP to other sorting algorithms. The extension of the analytical model to predict performance of PSOP on machines with several levels of memory hierarchy is straightforward. The overpartitioning approach can also be used in message passing multiprocessors, such as the Intel Paragon, or the Cray T3D. We plan to implement PQOP and PROP on some message passing machines and compare the results against alternative algorithms.

Acknowledgements: Our thanks to Dattatraya H. Kulkarni at University of Toronto, Stephen Breit at the Kendall Square Research, and Jonathan Schaeffer at University of Alberta for their excellent comments on earlier drafts of this paper, and to University of Washington for providing us access to the KSR1.

References

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Architectures.* Kluwer Academic Publishers, 1991.

[2] M. Ajtai, J. Komolos, and E. Szemeredi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[3] S. G. Akl. *Parallel Sorting Algorithms.* Academic Press, Toronto, 1985.

[4] D. H. Bailey, E. Barszcz, L. Dagum, and H. Simon. NAS parallel benchmark results. *IEEE Parallel and Distributed Technology*, 1(1):43–52, February 1993.

[5] K. Batcher. Sorting networks and their applications. In *Proc. of the AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[6] D. T. Blackston and A. Ranade. Snakesort: A family of simple optimal randomized sorting algorithms. In *Proc. of 22nd International Conference on Parallel Processing*, pages III–201–III–204, August 1993.

[7] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

[8] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. of Symposium on Parallel Algorithms and Architectures*, pages 3–16, Hilton Head, SC., July 1991.

[9] A. Borodin and J. Hopcroft. Routing, merging and sorting on parallel models of computation. *J. Computer Systems and Science*, 30:130–145, 1985.

[10] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[11] E. Felten, S. Karlin, and S. Otto. Sorting on a hypercube. Technical report, Hm 244, Caltech/JPL, 1986.

[12] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Volumne I, General Techniques aand Regular Problems.* Prentice Hall, 1988.

[13] R. S. Francis and L. J. H. Pannan. A parallel partition for enhanced parallel quicksort. *Parallel Computing*, 18:543–550, 1992.

[14] W. L. Hightower, J. F. Prins, and J. H. Reif. Implementation of randomized sorting on large parallel machines. In *Proc. of Symposium on Parallel Algorithms and Architectures*, pages 158–167, San Diego, CA., July 1992.

[15] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *Proc. of the IEEE Computer Society's 7th International Computer Software and Applications Conference*, pages 627–631, 1983.

[16] C. Kaklamanis, D. Kraizanc, L. Narayanan, and T. Tsantilas. Randomized sorting and selection on mesh-connected processor arrays. In *Proc. of Symposium on Parallel Algorithms and Architectures*, pages 17–28, Hilton Head, SC., July 1991.

[17] L. V. Kalé and S. Krishnan. A comparison based parallel sorting algorithm. In *Proc. of 22nd International Conference on Parallel Processing*, pages III–196–III–200, August 1993.

[18] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C–34(4):344–354, April 1985.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[20] P. P. Li and Y.-W. Tung. Parallel sorting on Symult 2010. In *Proc. of 5th Distributed Memory Computing Conference*, pages 224–229, Charleston, SC., April 1990.

[21] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):543–550, October 1993.

[22] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generlized connection network. *Journal of the ACM*, 29(3):642–667, July 1982.

[23] M.H. Nodine and J.S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. of Symposium on Parallel Algorithms and Architectures*, Velen, Germany, July 1993.

[24] C. G. Plaxton. Efficient computation on sparse interconnection networks. Technical Report STAN-CS-89-1283, Stanford University, Department of Computer Scienece, Stanford, CA, September 1989.

[25] M. J. Quinn. Analysis and benchmarking of two parallel sorting algorithms: hyperquicksort and quickmerge. *BIT*, 29(2):239–250, 1989.

[26] S. Rajasekaran and J. H. Reif. Optimal and sublogrithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, June 1989.

[27] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[28] Kendall Squre Research. *KSR1 Principles of Operation.* Waltham, MA, 1991.

[29] S. R. Seidel and W. L. George. Binsorting on hypercubes with d-port communication. pages 1455–1461, 1988.

[30] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:362–372, 1992.

[31] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. In *Proc. of Supercomputing*, pages 307–314, November 1992.

[32] A. Tridgell and R. P. Brent. An implementation of a general-purpose parallel sorting algorithm. Technical Report TR-CS-93-01, Computer Scienece laboratory, Australian National University, Australia, February 1993.

[33] Z. G. Vranesic, M. Stumm, D. M. Lewis, and R. White. Hector: A hierarchically structured shared memory multiprocessor. *IEEE Computer*, 24(1):72–79, January 1991.

[34] B. A. Wagar. *Practical Sorting Algorithms for Hypercube Computers.* PhD thesis, Depatrment of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, July 1990.

[35] Y. Won and S. Sahni. A balanced bin sort for hypercube multicomputers. *Journal of Supercomputing*, 2:435–448, 1988.