University of Alberta

PARALLEL SORTING ON MULTIPROCESSOR COMPUTERS

by

Shi, Hanmao

SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIRMENTS FOR THE DEGREE OF
Master of Science

Department of Computing Science

Edmonton, Alberta
FALL 1990

ISBN 0-315-64974-7

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: **Shi, Hanmao**

TITLE OF THESIS: **Parallel Sorting on Multiprocessor Computers**

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: **Master of Science**

YEAR THIS DEGREE GRANTED: **Fall 1990**

(Signed) [illegible]

Permanent Address:
Department of Computer Science
University of Science and Technology of China
Hefei, Anhui, P.R. China

Date: [illegible]

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Parallel Sorting on Multiprocessor Computers** submitted by **Shi, Hanmao** in partial fulfillment of the requirements for the degree of **Master of Science.**

[illegible]
(Supervisor)

[illegible]

[illegible]

[illegible]

Date: Aug 21/90

To My Parents

ABSTRACT

Sorting in computer terminology is defined as the process of rearranging a sequence of values in ascending or descending order. With the advent of parallel processing, parallel sorting has been an active area of research. Many parallel sorts which are theoretically optimal have been proposed. Unfortunately, their experimental results have been rather bleak.

In this thesis, the performance problem of parallel sorting on multiprocessor computers is studied. Based on the general strategies utilized, many parallel sorts suitable for multiprocessors can be placed into one of two rough categories: merge-based sorts and partition-based sorts. Merge-based sorts consist of multiple stages of merge, and are generally believed to perform well only with a small number of processors. Partition-based sorts consist of two phases: partitioning the data into smaller subsets and then sorting each in parallel. The key point of this sort method lies in developing an efficient, well-balanced partitioning scheme. Previous partition-based sorts are unstable in the sense that one of subsets may contain significantly more elements than its average share. A new algorithm called PSRS (*Parallel Sorting by Regular Sampling*) is developed to solve this problem. A proof is presented which shows PSRS is theoretically optimal when $n \geq p^3$. The algorithm has been implemented on the 64-processor Myrias SPS-2 and it shows half linear speedup. The significance of the result is that it demonstrates the first successful linear speedup parallel sort workable on parallel multiprocessors with a large number of processors. Extensions of PSRS on hypercube computers and LANs are also addressed.

Acknowledgements

I am indebted for advice and guidance to my supervisor Dr. Jonathan Schaeffer. His knowledge and experience have guided me throughout this research.

I would like to thank the members of my examining committee: Dr. W. Joerg, W. Dobosiewicz, and B. Joe, for their valuable comments and suggestions and for their time spent in reading this thesis.

I would also like to express my appreciation to the Department of Computing Science for technical and financial support.

Finally, deepest thanks to my parents, to whom I owe most.

Table of Contents

List of Tables

List of Figures

Chapter 1
Overview of Parallel Sorting

1.1. Introduction

Sorting in computer terminology is defined as the process of rearranging a sequence of values in ascending or descending order. Computer programs, such as compilers, editors, or database systems, often choose to sort tables and lists of symbols stored in memory in order to enhance the speed and simplicity of algorithms used to access them. Because of both their practical importance and theoretical interest, algorithms for sorting data stored in random access memory (*internal sorting*) have been the focus of extensive research. Early serial sorting algorithms were investigated [Kn73, Me84]. Many efficient serial algorithms[1] are known which can sort n values in $O(n \log n)$ comparisons[2], the theoretical lower bound for this problem. When n is large, this represents a significant amount of CPU time. Many applications, such as database enquiry evaluations, need to sort large lists frequently. IBM estimates that about 25% of total computing time is spent on sorting in commercial computing centers [Me84]. When a quick system response is critical (e.g. on-line enquiry), serial sorting can be undesirable.

The advent of parallel processing opens a new area of algorithmic research. Parallel processing makes it possible to perform more than a single comparison during each time unit; parallel processing also makes it possible to move multiple data items simultane-

[1] Strictly speaking, sorting algorithms can be divided into two large groups: those that are comparison-based and those that are not. The algorithms in the first group only make use of the fact that the universe is linearly ordered; while the algorithms in the second group depend on specific characteristics of keys in some restricted domains. Unless stated otherwise, we are referring to comparison-based algorithms.

[2] Unless stated otherwise, in this thesis, n stands for the size of the data to be sorted and p for the number of processors. All the logarithms use base two.

ously. By exploiting both kinds of parallelism, parallel algorithms for sorting can, in principle, sort n values in less than $O(n \log n)$ time units. In the last two decades, research on parallel sorting has evolved from the early sorting networks to (theoretical) shared memory models of parallel computation. Much of the work has been largely concerned with purely theoretical issues [BDHM84, Ri86]. Recently, with multiprocessor computers increasingly available, more realistic parallel sorts for the realizable architecture are becoming many researchers' interests [De82, EvYo85, FrMa88, Ri86].

1.2. Sorting Networks

It is somewhat surprising that initially the simple hardware problem of designing a multiple-input, multiple-output switching network was a prime motivation for the development of parallel sorting algorithms. Since a sorting network with n input lines can order any permutation of $(1, 2, ..., n)$, it can be used as a multiple-input, multiple-output switching network. One of the earliest and most important results is due to Batcher, who presented two fundamental parallel merging networks: the odd-even network and the bitonic network. Both sorting networks require $O(n \log^2 n)$ comparators to sort n values in $O(\log^2 n)$ time [Ba68]. As shown in Figure 1.1, a comparator is a module that receives two numbers on its two input lines A, B, and outputs the minimum on its output line L and the maximum on its output line H.

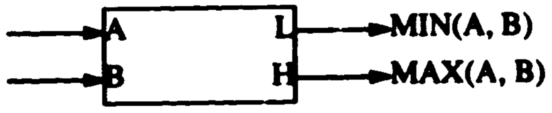

Figure 1.1 A comparison-exchange module

Since then, a wide range of network topologies have been proposed, and their ability to support fast sorting algorithms has been extensively investigated [Fe81, Pe77, Si79]. Until recently, however, the best-known performance remained an $O(\log^2 n)$ sorting time with $O(n \log^2 n)$ comparators. Despite the advances of VLSI technology, $O(n \log^2 n)$ comparators still represent a significant cost, especially when n is large. Another disadvantage is that a sorting network requires all the input data to be available simultaneously, which is far from easy to meet in practice. For these reasons, special purpose sorting networks have rarely been used in practice. A recent theoretical result may renew the interest in network sorting algorithms. Ajtai showed a network of $O(n \log n)$ comparators that can sort n values in $O(\log n)$ comparisons [Aj83, GiRy88]. Unfortunately his algorithm is unsuitable for implementation.

Sorting networks are characterized by their property of nonadaptability. They perform the same sequence of comparisons, regardless of the results of intermediate comparisons. Because of this, networking algorithms are conveniently implemented on SIMD (*Single Instruction stream, Multiple Data streams*) machines. For instance, Thompson and Kung adapted the bitonic sorting scheme to a mesh-connected processor, with three alternative indexing rules: the row-major rules, the snakelike row-major rules and the shuffled row-major rules [ThKu77]. The lower bound of sorting on a SIMD machine is dominated by the number of parallel routine steps and is often referred as the *distance-bound*. In the case of a $n \times n$ mesh-connected processor, the *distance-bound* is $O(n)$, since it may have to change the elements from two opposite corners of the mesh-connected processor. This *distance-bound* effectively imposes a bound of $O(n)$ sorting time on the mesh processor.

In summary, special purpose sorting networks are not cost-effective in practice. Most issues addressed are largely of theoretically interest. It is beyond the scope of this paper to investigate all these sorting networks in details.

1.3. Parallel Sorting on Shared Memory Models of Parallel Computation

After the time bound of $O(\log^2 n)$ was achieved with the network sorting algorithms, researchers eagerly attempted to improve it with new models of parallel computation. One of the most used models of parallel computation is the PRAM (*Parallel Random Access Model*) [Qu87]. The PRAM consists of a set of processors and a common random access memory, where each processor is fully programmable and can read or write to the common random access memory at every step of the computation. Different processors may read the same location at the same time, but writing to the same location is disallowed. The model essentially neglects any hardware constraints which a specific multiprocessor architecture would impose. In particular, it assumes a conflict-free communication channel of virtually unlimited bandwidth between the processors and the common random access memory. With this model, Cook *et al.* formally proved $O(\log n)$ to be the theoretical lower bound of sorting n values in parallel [CRR86]. Based on the same or similar models, various "fast" parallel sorting algorithms have been proposed [Co86, Hi78, Pr78, Va75]. One of the fastest parallel sorts achieves a complexity of $(C \log n)/a + O(\log n)$ using n^{1+a} processors [Pr78]. These algorithms generally use ***enumeration*** to compute the rank of each element. Sorting is performed by computing in parallel the rank of each element, and routing the elements to the location specified by their ranks. Unfortunately, the PRAM and its associated models are too powerful to be implemented with short-term or foreseeable technology. Hence sorting on PRAM remains only of pure theoretical interest.

1.4. Sorting on Multiprocessor Computers

The rapid advances of VLSI technology have made it possible to design and fabricate single-chip processors that have transistor complexity and performance comparable to CPUs found in minicomputers and mainframes [He84]. The cost of hardware continues to drop. As pipe-lined vector processing computers approach the limits imposed by technology and physics, many computer manufacturers, and the scientific computing community in general, have begun to realize that further substantial gains in processing speed could be achieved by linking a number of processors together. Thus, the computing power of many individual processors could be harnessed into a single parallel multiprocessor computer or, simply, multiprocessor. Each processor could work in parallel on independent subcomputations, or tasks, contributing to the fast solution of a large problem. Two types of multiprocessors can be identified with respect to memory organization: shared memory multiprocessors and distributed memory multiprocessors. In the former case, each processor accesses the shared main memory through buses or a network, possibly via its own small private cache. In the latter case, the processor plus local memory forms a processing element (PE), and all the PEs communicate through buses or a network. Shared memory multiprocessors have the significant advantage of fast communication through shared memory, but have also a limitation on the number of processors (some researchers suggest that 16 might be the maximum) due to the fact that the communication bandwidth between the processors and the shared memory can easily become saturated. Currently, the most commercially successful supercomputers and mini-supercomputers remain of this type. Notable examples are the Cray series, and the Alliant FX series [DoDu89]. On the other hand, distributed memory multiprocessors are easier to build and more important, *scalable* [Sm87]. No limitation exists on the number of processors. Examples are the Hypercube [KaKa89], the BBN, the Myrias SPS-2

[KVW87], *etc.* However, the recent explosive growth of research on distributed systems may result in a blurring of this simple distinction [BaTa88, LaEl90]. In the remainder of this thesis, unless explicitly stated otherwise, by multiprocessor we mean either a shared memory or distributed multiprocessor.

The speedup of a program on a multiprocessor can be defined for a given number of processors p as the ratio of time elapsed when executing the program on a single processor to the execution time when there are p processors available. Ideally a multiprocessor computer with p identical processors working concurrently on a single problem is p times faster than a single processor. In other words, the maximal speedup achievable in solving a problem on a multiprocessor with p identical processors is p. Unfortunately, the speedup is much less in practice. It has been reported that the effective performance of a current supercomputer ranges between only 5% and 25% of its peak performance [Hw87].

The multiprocessor approach introduces three new requirements that are not encountered in the uniprocessor environment. First, each problem to be solved must be partitioned into smaller tasks. Second, each task must be scheduled for execution on one or more processors. Third, synchronization of control and data flow must be performed during execution to ensure that cooperating tasks are executed in the correct sequence. In addition, the memory latency has been further aggravated by the addition of a network between processors and memories. As a result, the behavior of a parallel program exerts great effect on the efficiency of the program executing on such architectures [EgKa88]. Roughly speaking, if tasks are relatively independent of each other and possess a good per-task reference locality, the program will likely yield good performance.

Various sorting algorithms suitable for parallel multiprocessors have been proposed. Unfortunately, their experimental results give a rather pessimistic picture. Some of them show a limited speedup, generally believed to be 5 or 6, regardless of the number of processors available [De82]. Others perform well only with a very small number of processors [FrMa88, Qu88].

1.5. The Problem

Although more realistic, as compared to the PRAM, sorting networks are not yet considered cost-effective in practice. The remaining avenue of approach to efficient sorting is to find fast parallel sorts suitable for multiprocessors. The speedup of a parallel sort achievable on a multiprocessor depends largely on how well we can minimize the average memory latency and the overhead of scheduling and synchronization. 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, and perform well only with a small number of processors. When the number of processors utilized gets larger, so does the overhead of scheduling and synchronization, which reduces the speedup. The performance degrades when the overhead of scheduling and synchronization prevails over the benefits obtained from parallelism. On the other hand, 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 we can quickly partition the data evenly into smaller ordered subsets. Unfortunately, to the author's knowledge, no 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.

1.6. Thesis Organization

Chapter 2 reviews previous work on multiprocessor parallel sorting. Five algorithms are examined that are considered representative. Some of them represent "state-of-the-art" parallel sorts suitable for multiprocessors. The topics of discussion of each algorithm include the algorithm description, its time complexity, and performance analysis. Chapter 3 is wholly devoted to our new proposed parallel sort PSRS (*Parallel Sorting by Regular Sampling*). A theoretical proof is presented which shows PSRS is asymptotically optimal when $n \geq p^3$. Chapter 4 describes our experiments on the 64-PE Myrias SPS-2, a parallel multiprocessor computer recently developed by the *Myrias Research Corporation* based in Edmonton. PSRS and three of the five algorithms discussed in Chapter 2 are implemented on the SPS-2. The performance results show PSRS has the best speedups among the four. On the average, PSRS achieves the speedup of roughly $\frac{p}{2}$ when p processors are utilized. The experiments demonstrate the first successful linear speedup parallel sort workable for parallel multiprocessors with many processors. One of interesting point is that PSRS spends 4.05 seconds to sort one million 32-bit integers with 64 PEs. It has been reported that a Cray X-MP, which is one of the most commercially successful supercomputers and costs about 7 times more than a 64-PE Myrias SPS-2, spends 4.96 seconds sorting the same number of integers[3]. Chapter 5 discusses the extensions of PSRS on hypercube computers and LANs. Some analytic results are presented. Finally, Chapter 6 concludes the thesis.

[3] Based on personal correspondence.

Chapter 2
Previous Studies

2.1. Introduction

This chapter reviews the previous work on multiprocessor parallel sorting. It is not practical to include all previous proposed parallel sorts here due to space limitations. Neither is there a necessity to do so, since many of them are based on unrealistic assumptions, which are beyond our interests. Our focus will be on those which are *suitable* for multiprocessors. By *suitable* we mean those algorithms that are realistic and likely to yield good performance in an implementation. Instead of saying a few words about each algorithm, we have chosen to look at five algorithms that we consider representative. The algorithms examined here are parallel Quicksort, parallel two-way merge sort, parallel merge sort, Quickmerge, and parallel sorting by sampling. The first two are straightforward extensions of their respective serial sorts, while the latter three represent "state-of-the-art" parallel sorts for multiprocessors. They are all easily implementable on general purpose multiprocessors. Algorithms which depend on specific multiprocessor architectures are not discussed here.

The topics of discussion for each algorithm include the algorithm description, its time complexity, and performance analysis. For practical purpose, we always assume $p < n$. Our emphasis is on their speedups in practice, not merely in theory.

2.2. Parallel Quicksort (PQ)

The Quicksort algorithm is considered to be the most efficient general-purpose sorting algorithm, as both empirical and analytic studies have shown [Ho61, Kn73, Lo74]. It is a comparative, interchange algorithm based on recursively partitioning the original data set into smaller subsets. The nature of this algorithm appears suitable to be implemented on a parallel multiprocessor. PQ is a parallel version of Quicksort. It has been implemented by several authors [De82, EvYo85, MoSt87]. However, their basic results are similar.

2.2.1. Algorithm Description

All the data to be sorted are stored in an array. A single global stack stores the indices of subarrays that are still unsorted. A number of processors work independently. Whenever a processor is without work, it checks to see if the stack is nonempty. If it is, the processor locks the stack, pops a pair of indices of an unsorted subarray off the stack, and then unlocks the stack. If the unsorted subarray is smaller than a predefined threshold, the processor sorts it using a simple insertion method. Otherwise the processor partitions its subarray, using the median of the first, middle, and last element of the subarray as a splitter, into two smaller subarrays, containing less than or greater than the splitter. After the partitioning step, the processor locks the global stack, pushes the indices of one unsorted subarray onto the stack, unlocks the stack, and repeats the partitioning process on the other unsorted subarray. When the stack becomes empty, the array is sorted.

2.2.2. Time Complexity and Performance Analysis

Two phases may be distinguished in PQ. In the first phase, the number of processors is greater than the number of available subarrays to be sorted. Thus some processors remain idle during that phase. In the second phase there are enough subarrays so that all processors can keep busy. Because of the first phase, the speedup of the algorithm can't be linear. The best speedup theoretically achievable with this algorithm is $O(p \,/\, (1 + (2p - 2 - \log p) \,/\, \log n))$ [De82]. No synchronization overhead is assumed in this complexity. The speedup should grow as more processors are used. However, experiments show only a limited speedup, generally believed to be 5 or 6 [FrMa88], can be achieved. The most important factor that constrains the speedup is the low amount of parallelism early in the algorithm's execution. The initial partitioning step, during which the array is partitioned into two subarrays, is performed by a single processor; the other processors remain idle. After the array has been divided into two subarrays, two processors can each partition one of these subarrays, but again the rest of the processors have nothing to do. This is not an insignificant amount of time to wait. The partitioning of the original unsorted array forms a significant sequential component that puts a ceiling on the maximum speedup achievable, regardless of the number of processors available. The second factor, which limits the speedup when more processors are used, is the contention of processors for the shared global stack containing indices of unsorted subarrays. Many processors may happen to require access to the global stack simultaneously, yet only one of them is allowed to access it at a time, others have to wait for another chance after the stack is released. If the number of processors is large, accessing the stack inevitably becomes a bottleneck. Consequently, more processors may mean a lower speedup. There is an optimal number of processors for sorting a certain sized problem [De82].

2.3. Parallel Two-way Merge Sort (PTMS)

Like PQ, PTMS [EvYo86] is a straightforward extension of the sequential two-way merge sort. Two-way merge sort is well understood in the sequential model of computation. It consists of multiple stages. During each stage, pairs of sorted subarrays produced in the previous stage are merged into longer ones. PTMS makes use of the simple fact that pairs are independent of each other, and therefore can be merged in parallel.

2.3.1. Algorithm Description

The algorithm has two phases. In the first phase, each of p processors sorts a contiguous subarray of approximately $\frac{n}{p}$ elements of the array using sequential Quicksort. After this phase, all processors synchronize, and the array can now be seen as a set of p sorted subarrays. Phase two consists of $\log p$ merge stages, assuming p is a power of 2. During each stage i, where $1 \le i \le \log p$, $\frac{p}{2^{i-1}}$ sorted subarrays are merged pairwise into $\frac{p}{2^i}$ longer sorted subarrays in parallel using $\frac{p}{2^i}$ processors, one per pair. At the end of each stage, all processors synchronize. For example, Figure 2.1 illustrates how eight processors would sort an array. At the beginning, each of the processors sorts a contiguous portion of the array in parallel with sequential Quicksort. The merge phase consists of three stages. In the first stage, eight sorted subarrays are merged pairwise in parallel into four longer ones with four processors, one per pair. In the second stage, four sorted subarrays are merged pairwise into two sorted subarrays with two processors in parallel. Finally, two sorted subarrays are merged by one processor.

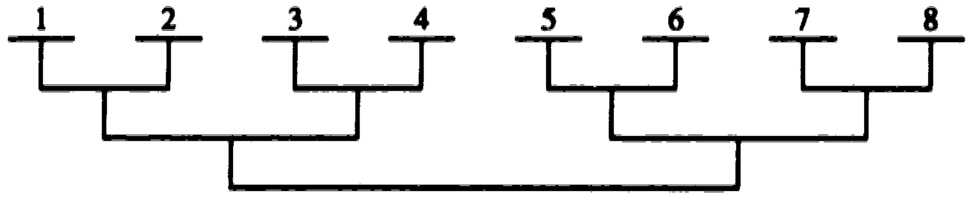

Figure 2.1 Parallel two-way merge

2.3.2. Time Complexity and Performance Analysis

The maximal speedup theoretical achievable is $O(p\ /\ (1 + (2p\ - 2 - \log p)\ /\ \log n))$ [EvYo86]. The low parallelism late in the algorithm's execution is the major factor that limits the speedup. In the first merge stage, only half of the processors are busy. In the second merge stage, only one fourth of the processors are busy, and so on. In the last merge stage, a single processor has to perform a merge of two sorted subarrays of total size n, which puts a ceiling of $O(n)$ on the sorting time regardless of the value of p. Experiments show the algorithm works well with a few number of processors. When p is larger, it shows an effect similar to that seen with PQ. This is because when p gets larger, so does the overhead of scheduling and synchronization, which reduces the speedup. The performance degrades when the total overhead prevails over the benefits obtained from parallelism.

2.4. Parallel Merge Sort (PMS)

It becomes clear that better speedups can't be achieved with merge-based sorts if the performance problem inherent in the process of merging two sorted subarrays remains unsolved. With this in mind, Francis and Mathieson proposed a *parallel merge*, which eliminates the $O(n)$ lower bound by first dividing the data to be merged evenly among the processors, and then merging them in parallel [FrMa88]. Based on this *parallel merge*, PMS was proposed by the same authors.

2.4.1. Algorithm Description

The algorithm is basically similar to PTMS. It has also two distinct phases. The first phase is the same as that of PTMS, with each of p processors sorting a contiguous subarray of no more than $\lceil \frac{n}{p} \rceil$ elements of the array with Quicksort. Phase two also consists of $\log p$ merge stages, assuming p is power of 2. During each stage i, where $1 \le i \le \log p$, $\frac{p}{2^{i-1}}$ sorted subarrays produced in the previous stage are merged pairwise into $\frac{p}{2^i}$ sorted subarrays using p processors, 2^i per pair. At the end of each merge stage, all processors synchronize. The difference from PTMS lies in the process of merging two ordered subarrays into a longer one. The *parallel merge* allows each pair of sorted subarrays to be merged by a group of processors instead of a single processor. Specifically, at stage i, p processors are divided into $\frac{p}{2^i}$ groups of a size. Each group works independently on one pair of previously produced sorted subarrays. First, each of $2^i - 1$ processors of a group works in parallel to find a pair of boundaries with a modified binary search. These pairs of boundaries divide the two sorted subarrays to be merged evenly into 2^i pairs of sorted sublists with the following property: each element of one pair is no greater than any element of another. After that, all the 2^i processors synchronize, and the two subarrays can be seen as 2^i independent pairs of sorted sublists. With the demarcations established before, 2^i pairs of sublists are then merged in parallel by the same group of 2^i processors, one per processor. Figure 2.2 illustrates how two sorted subarrays are merged by two processors in parallel. First, the two subarrays are divided evenly into two pairs of sublists such that each element of one pair is no greater then any of the other. This can be done with a modified binary search on the two subarrays. The dashed lines in the figure show the demarcations for the established partitioning. After that, each processor performs a merge on one of the two pairs simultaneously.

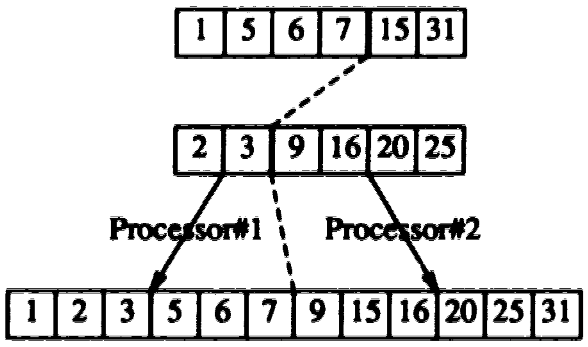

Figure 2.2 Example of parallel merge

2.4.2. Time Complexity and Performance Analysis

The time complexity of this algorithm is $O(\frac{n}{p}\log n) + O(\frac{n}{p}\log p)$ [FrMa88]. This complexity assumes no synchronization overhead. Francis' implementation on the Sequence Balance 2100, a bus-connected, shared memory multiprocessor, shows a near-linear speedup when the number of processors utilized is not large. Unfortunately, he didn't count the time taken to create tasks[4], leaving his linear speedup arguable. In gen-

[4] Based on personal correspondence with the authors of the algorithm.

eral, the algorithm works well with a small number of processors. It is, however, not suitable for a large number of processors because the overhead of the synchronization and scheduling required can make the parallelism completely worthless. Furthermore, the algorithm is not well suitable for distributed multiprocessors because of the intensive data motion involved in the process of the merging.

2.5. The Quickmerge Algorithm

PQ is known to inefficiently use processors early in its execution, while merge-based parallel sorts suffer from the high overhead of scheduling and synchronization as well as the intensive data motion during its merging process. Quickmerge [Qu88], a hybrid of mergesort and Quicksort as its name suggests, is intended to take advantage of both the algorithms to overcome their respective deficiencies.

2.5.1. Algorithm Description

The Quickmerge algorithm has three phases. In the first phase, each of p processors sorts a contiguous subarray of approximately $\frac{n}{p}$ elements of the array using sequential Quicksort. After this phase all processors synchronize. The array can now be seen as a set of p sorted subarrays of size $\frac{n}{p}$.

In the first sorted list of $\frac{n}{p}$ elements, $p-1$ evenly spaced elements are used as boundaries to partition each of the remaining sorted lists into p sublists. The second phase accomplishes the partitioning as follows. Each processor i, where $1 \le i \le p-1$, finds, for lists 2 through p in parallel, the index of the largest element no larger than the element located at index $\lfloor \frac{i \times n}{p^2} \rfloor$ in the first sorted list. After this phase all processors synchronize. At this point each of the sorted lists has been divided into p sorted sublists with the property that each element in every list's ith sorted sublist is greater than any element in any list's $(i-1)$th sorted sublist, for $2 \le i \le p$.

In the third phase, each processor i, where $1 \le i \le p$, performs a merge sort on every list's ith sorted sublists. Because of the demarcations established in phase two, these merges are completely independent of each other, and are therefore done simultaneously. After this phase all processors synchronize and the array is sorted. Figure 2.3 illustrates how a list of twenty-four elements are sorted by three processors using Quickmerge. After the first phase, the array consists of three sorted lists of size eight. During the second phase, each sorted list is divided into three sublists. In the last phase, each processor performs a merge on its own set of sublists. Note that every processor knows where to start storing its merged results in the final array.

2.5.2. Time Complexity and Performance Analysis

The time complexity of this algorithm depends largely on how evenly the data can be partitioned in the second phase. In the best case, each processor performs a merge sort on approximately $\frac{n}{p}$ elements. The best time complexity is therefore $O(p + p \log\frac{n}{p} + \frac{n}{p}\log n) = O(\frac{n}{p}\log n)$, for $n \ge p^2$. In the worst case, a single processor may have to perform a merge sort on $O(n)$ elements in phase three. Hence the worst time complexity of Quickmerge is $O(p + n \log p + (\frac{n}{p} + p)\log\frac{n}{p})$.

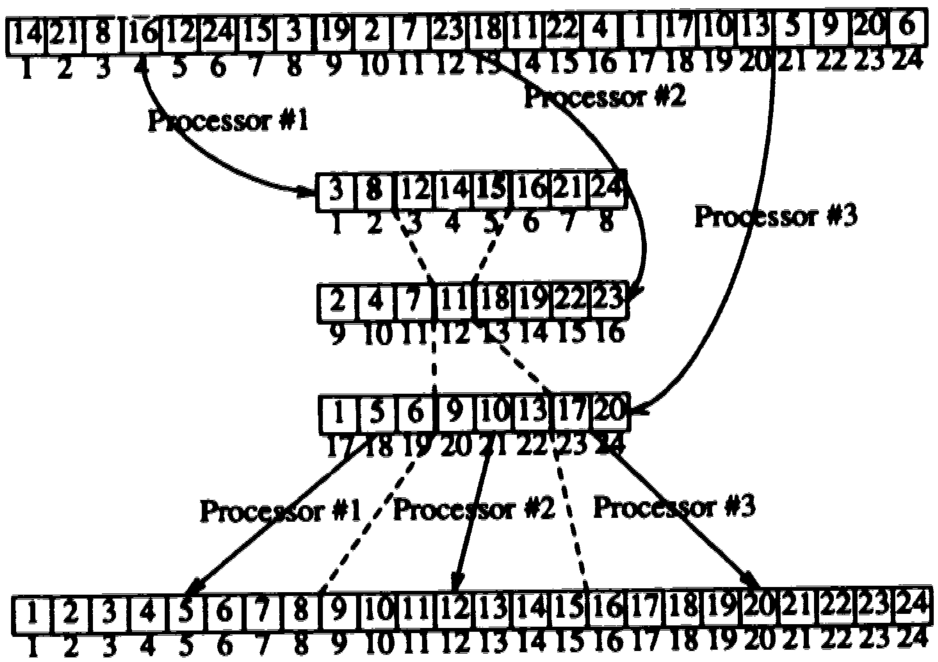

Figure 2.3 Example of Quickmerge

Quinn's experiments show a better execution time than PQ [Qu88]. Like PMS, the algorithm is suitable only for multiprocessors with a small number of processors. As the number of processors increases, so does the probability that one of the processors will have to merge significantly more than its share of elements in the third phase of the algorithm. This reduces the speedup, since the final synchronization can't complete until the last processor finishes merging.

2.6. Parallel Sorting by Sampling (PSS)

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, we can easily divide the data into p equal-sized subsets such that each element in the ith subset is no greater than any element in the $(i+1)$th subset, where $i = 1, 2, \ldots, p-1$, and then sort each subset in parallel. Unfortunately, in general, we have no such luck. To overcome this difficulty, we may draw a random sample from the data and use the order information of the sample to help the partitioning [HuCh83]. 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.

2.6.1. Algorithm Description

The algorithm is basically similar to Quickmerge, except for the additional work of selecting $p-1$ elements as boundaries to form p partitions. We sometimes call such elements *pivots*. PSS has three distinct phases. In the first phase, $p-1$ unique pivots are selected from the data. The pivots are selected by drawing a small random sample from the data, sorting it, and then selecting $p-1$ elements evenly spaced from the sorted sample. All are done in serial by one processor.

The second phase is equivalent to both phase one and two of Quickmerge together. Each processor i, where $1 \le i \le p$, first sorts a contiguous subarray of approximately $\frac{n}{p}$ elements of the array using serial Quicksort, then finds the index of the largest element no larger than each of the jth pivots, $j = 1, 2, \ldots, p-1$. After this phase all processors synchronize. At this point the data to be sorted can be seen as set of p sorted lists, each having been divided into p sorted sublists with the property that each element in every list's ith sorted sublist is greater than any element in any list's $(i-1)$th sorted sublist, for

$2 \le i \le p$.

The last phase is exactly the same as that of Quickmerge. After this phase all processors synchronize and the array is sorted.

2.6.2. Time Complexity and Performance Analysis

It is not an easy task to give a theoretical analysis on the speedup of the sort. Generally speaking, the performance depends to a large degree on the distribution of the original data and the choice of a proper sample size. If the data to be sorted has a uniform or near uniform distribution, a small sample is almost sufficient to achieve an even partitioning. On the other hand, if the data has unusual ordering characteristics, a larger sample might be necessary. As the cost of sampling increases with the size of the sample, the speedup decreases. To make matters worse, a larger sample does not necessarily result in an even partitioning. It is possible that in the worst case one processor may still have to perform a merge sort on nearly all the data in the last phase of the algorithm.

2.7. Summary

We have examined five parallel sorting algorithms suitable for parallel multiprocessors. Quicksort has been parallelized by several authors. 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 used. A similar effect happens to Evan's PTMS, as little parallelism can be exploited in the last few phases of merge. Francis noticed this problem and proposed a *parallel merge* which eliminates the $O(n)$ lower bound by first partitioning the data to be merged evenly among the processors. However, his sorting algorithm works well only with a small number of processors. The Quickmerge algorithm is a combination of Quicksort and mergesort. Similarly, the sort is unsuitable for a large number of processors as the performance depends on how evenly the data can be partitioned. When the number of processors increases, so does the probability that one of the processors will have to merge significantly more than its share of elements in the third phase of the algorithm. The performance of PSS depends on the distribution of the original data and the choice of a proper sample size. Theoretically, it is always possible that in the worst case one processor may have to perform a merge sort on nearly all the data.

In conclusion, we can distinguish two kinds of sort based on their general strategies: merge-based sorts and partition-based sorts. Merge-based sorts consist of multiple merge stages, and performs well only with a small number of processors. They are not suitable for distributed memory multiprocessors. PTMS and PMS belong to this category. On the other hand, 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. Examples are PQ, Quickmerge, and PSS. Partition-based sorts encounter the difficulty of achieving a balanced or even partitioning. To solve this problem, we propose a new partitioning scheme called ***regular sampling***, which is described in the next chapter.

Chapter 3
Parallel Sorting by Regular Sampling (PSRS)

3.1. Introduction

This chapter describes in detail our new proposed algorithm called PSRS (*Parallel Sorting by Regular Sampling*). A proof is presented which shows PSRS is asymptotically optimal when $n \geq p^3$. The advantages and disadvantages of the algorithm are also addressed.

In the review chapter we summarized that parallel sorts suitable for multiprocessors fall into two rough categories: merge-based sorts and partition-based sorts. Merge-based sorts consist of multiple merge stages and are generally believed to perform well only with a small number of processors. Partition-based sorts consist of two phases:

(1) partition the data set into p subsets such that each element in the ith subset is no greater than any element in the $(i+1)$th subset, where $i = 1, 2, ..., p-1$;

(2) sort each independent subset in parallel.

The key point of this sort method lies in developing an efficient, well-balanced partitioning scheme. A recursive binary partitioning scheme has been tried in the PQ algorithm (Section 2.2). The problem lies in the first few partitioning steps, which include significant sequential components that impose an $O(n)$ lower bound on the sorting time, regardless of the number of processors available. It becomes clear that the sort would probably run much faster if a way could be found to do a p'ary partitioning at the top level instead of using the standard binary partitioning recursively.

An arbitrary p'ary partitioning will not make much sense as one of p partitions may contain significantly more elements than its average share. In such a case, one of p

processors has to spend a lot more time than others. The speedup reduces since the sort can't complete until the last processor finishes sorting. In order for a partition-based sort to perform well, it is critical that the partitioning is well-balanced. It seems difficult since, in general, we have no knowledge of the distribution of the data before we start sorting.

To overcome this difficulty, we may draw a (random) sample from the data. Intuitively, the sample provides a representation of the original data. In other words, the elements of a sample bear some order information of the original data. Therefore, the order statistics of a sample can be used to help partitioning (see Section 2.5). The effectiveness of random sampling depends on how well the sample drawn represents the original data. This in turn depends on the distribution of the original data, the choice of a proper sample size, and some other factors.

Since our final objective is to get the data ordered, we may let each processor sort in parallel a contiguous list of approximately $\frac{n}{p}$ elements at the beginning, as in Quickmerge. After this the data can be seen as a set of sorted lists. Because the data are now locally ordered, intuitively, a set of elements evenly spaced from all the sorted lists should make a good representation of the original data. Their order statistics can therefore be used to help partition the data, hopefully, into well-balanced smaller subsets. We refer to the process of selecting a set of elements evenly spaced from all the sorted lists of the array to be representatives of the original data as *regular sampling*, as opposed to random sampling used by PSS in Section 2.5.

3.2. Parallel Sorting by Regular Sampling

Let the data { x_1, x_2, ..., x_n } to be sorted on a p-processor multiprocessor be denoted by X and the size of X by n; let $X_{i:j}$ be { $x_i, x_{i+1}, \dots x_j$ }, where $1 \le i \le j \le n$. X is a subset of a linearly ordered domain. We assume $n \ge p^2$, p^2 divides n, and $x_i \ne x_j$, where $1 \le i < j \le n$. The first assumption is based on our observation that in practice parallel sorting makes sense only when n is large compared with the number of processors p. The last two assumptions are only to simplify the discussions which follow and the modifications will be trivial if these two assumptions are not strictly met.

PSRS has three phases. In the first phase, each of p processors sorts in parallel a contiguous list of size $\frac{n}{p}$ using sequential Quicksort. Specifically, each processor i, where $1 \le i \le p$, sorts a list $X_{(i-1)\times\frac{n}{p}+1\,:\,i\times\frac{n}{p}}$ with sequential Quicksort. After this phase all processors synchronize and X is now said to be locally ordered.

We define the *regular sample* of X as a set of p^2 elements evenly spaced from all the p sorted lists, p per list, as follows:

$$x_{1+j\times\frac{n}{p}},\ x_{1+\frac{n}{p^2}+j\times\frac{n}{p}},\ \dots,\ x_{1+(p-1)\times\frac{n}{p^2}+j\times\frac{n}{p}},\quad \text{for } j = 0, 1, \dots, p-1.$$

In other words, from the p sorted lists, we extract p elements evenly spaced throughout each of the lists. Intuitively, as X is now locally ordered, these p^2 elements make a good representation of the original X. Their order information is useful to help partition X evenly.

In phase two, the *regular sample* is sorted at first using the sequential Quicksort and the sorted results are as follows:

$$y_1 < y_2 < \cdots < y_{p^2},$$

where y_k is the kth smallest element in the *regular sample*. Because the *regular sample* contains the p low ends of all the p lists, we exclude the first p smallest elements of the sorted *regular sample* and choose $p - 1$ elements,

$$y_{p+\lfloor \frac{p}{2} \rfloor}, y_{2p+\lfloor \frac{p}{2} \rfloor}, \ldots, y_{(p-1)p+\lfloor \frac{p}{2} \rfloor},$$

evenly spaced from the remainder of the *regular sample* as $p-1$ pivots, which are then used as boundaries to form p partitions of X. The partitioning is accomplished as follows. Each processor i, where $1 \le i \le p$, finds, for lists 1 through p in parallel, the index of the largest element no larger than each of the jth pivots, $j = 1, \ldots, p - 1$. After this phase all the processors synchronize. At this point each of the p sorted lists of X has been divided into p sorted sublists with the property that each element in every list's ith sorted sublist is greater that any element in any list's $(i-1)$th sorted sublist, for $2 \le i \le p$.

In the last phase, each processor i, where $1 \le i \le p$, performs a two-way mergesort to merge every list's ith sorted sublists. 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, and can therefore be done simultaneously. After this all processors synchronize, and X is sorted.

Figure 3.1 illustrates a three-processor sort of an array of thirty-six elements using PSRS. After the initial Quicksort phase, the array consists of three sorted lists of length twelve. The *regular sample* of the array consists of nine evenly spaced elements, three per list. In the figure they are in boldface. In the second phase, the *regular sample* is sorted first and two elements, 11 and 23, of it are chosen as boundaries to form three partitions of the array. A binary search is then used to divide each sorted list into three sublists. In the final phase each processor performs a merge on its own set of sublists. The dashed lines connecting the sublists in the figure demarcate the elements to be merged in the final phase. Note that every processor knows where to begin storing its merged list, since the sizes of all the three partitions are known.

3.3. Time Complexity and Performance Analysis

The initial Quicksort phase has time complexity $O(\frac{n}{p}\log\frac{n}{p} + p)$. The first term represents the time consumed by each processor to sort $\frac{n}{p}$ elements using sequential Quicksort and the final term represents the time needed to synchronize p processors. In phase two, sorting the *regular sample* using sequential Quicksort needs $O(p^2 \log p^2)$ time. Then each processor must perform $p - 1$ binary searches on a sorted list of size no greater than $\frac{n}{p}$. Hence the time complexity of phase two is $O(p + p^2 \log p^2 + p \log\frac{n}{p})$. In phase three, we shall prove that no processor merges more than $\frac{2n}{p}$ elements. Thus the last phase can finish in time no more than $O(p + \frac{2n}{p} \log p)$. Upon summation over the times of all the three phases, we get the time complexity of PSRS as follows: $O(p + p^2 \log p^2 + \frac{n}{p} \log p + \frac{n}{p} \log n)$, which is asymptotic to $O(\frac{n}{p}\log n)$ when $n \geq p^3$.

Now we turn to prove that the size of the data to be merged by any processor in the last phase of the algorithm is less than $\frac{2n}{p}$.

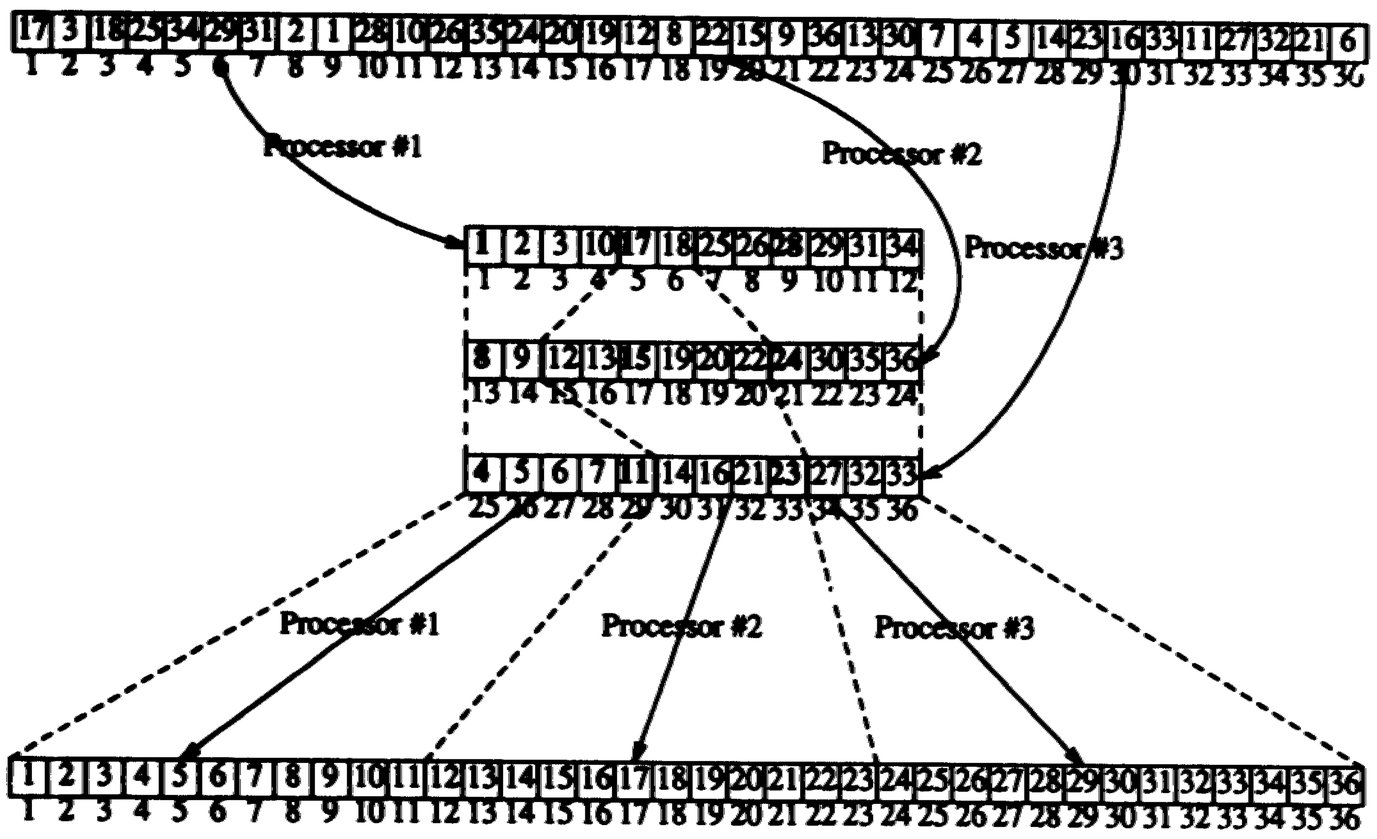

Figure 3.1 Example of PSRS.

Proof:

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

Case 1: $i = 1$.

All the data to be merged by processor 1 must be less than or equal to $y_{p+\lfloor \frac{p}{2} \rfloor}$. Since there are $(p^2 - p - \lfloor \frac{p}{2} \rfloor)$ elements of the *regular sample* which are greater than $y_{p+\lfloor \frac{p}{2} \rfloor}$, there are at least $(p^2 - p - \lfloor \frac{p}{2} \rfloor) \times \frac{n}{p^2}$ elements of X which are greater than $y_{p+\lfloor \frac{p}{2} \rfloor}$. In other words, there are at most $n - (p^2 - p - \lfloor \frac{p}{2} \rfloor) \times \frac{n}{p^2} = (p + \lfloor \frac{p}{2} \rfloor) \times \frac{n}{p^2} < 2 \times \frac{n}{p}$ elements of X which are less than or equal to $y_{p+\lfloor \frac{p}{2} \rfloor}$.

Case 2: $i = p$

All the data to be merged by processor p must be greater than $y_{(p-1)\times p+\lfloor \frac{p}{2} \rfloor}$. There are $(p-1) \times p + \lfloor \frac{p}{2} \rfloor$ elements of the *regular sample* which are less than or equal to $y_{(p-1)\times p+\lfloor \frac{p}{2} \rfloor}$. That is, there are at least $(p^2 - 2\times p + \lfloor \frac{p}{2} \rfloor) \times \frac{n}{p^2} + p$ elements of X which are less than or equal to $y_{(p-1)\times p+\lfloor \frac{p}{2} \rfloor}$, or there are at most $(2\times p - \lfloor \frac{p}{2} \rfloor) \times \frac{n}{p^2} - p \; < \; 2 \times \frac{n}{p}$ elements of X which are greater than $y_{(p-1)\times p+\lfloor \frac{p}{2} \rfloor}$.

Case 3: $1 < i < p$

All data to be merged by processor i must be greater than $y_{(i-1)\times p+\lfloor \frac{p}{2} \rfloor}$ and less than or equal to $y_{i\times p+\lfloor \frac{p}{2} \rfloor}$. There are at least $((i-2)\times p + \lfloor \frac{p}{2} \rfloor) \times \frac{n}{p^2} + p$ elements of X which are less than equal to $y_{(i-1)\times p+\lfloor\frac{p}{2}\rfloor}$. On the other hand, there are at least $((p-i)\times p - \lfloor\frac{p}{2}\rfloor)\times\frac{n}{p^2}$ elements of X which are greater than $y_{i\times p+\lfloor\frac{p}{2}\rfloor}$. Since the size of X is n, there are at most $2p\times\frac{n}{p^2}-p = 2\times\frac{n}{p}-p < 2\times\frac{n}{p}$ elements of X for processor i to merge.

In conclusion, no processor merges more than $\frac{2n}{p}$ elements in the last phase of PSRS.[5]

This bound on the size of the data to be merged in PSRS makes a big 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 (never exceeding $\frac{2n}{p}$) 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 PE is initially allocated a portion of approximately $\frac{n}{p}$ elements, then no data transmission is required during the first phase. In the second phase, only p^2 elements need to be collected and $p-1$ pivots are required to be broadcast to all the PEs. In the last phase, each PE has to send $p-1$ sublists to the other $p-1$ PEs. After a PE receives its sublists, it stores them locally and then works on them, totally independent of others. The total number of messages required in this phase is therefore $(p-1)p$ and the total data traffic is no greater than n. To sum up, PSRS has $O(p^2)$ message complexity and $O(n)$ data traffic complexity,

[5] If p^2 doesn't divide n, it is easy to prove that no processor merges more than $2\lceil\frac{n}{p}\rceil$ elements.

where $p \ll n$.

The disadvantage of PSRS lies in the fact that PSRS is asymptotically optimal when $n \geq p^3$. This means the number of the data must be large compared with the number of processors in order for PSRS to yield good performance. In practice, however, it is still quite reasonable to assume $n \geq p^3$

The above analysis is backed by the experiments on the Myrias SPS-2, which are described in the next chapter.

Chapter 4
Experiments on the Myrias SPS-2

4.1. Introduction to the Myrias SPS-2

The rapid advances of VLSI technology have made it possible to design and fabricate single-chip processors that have transistor complexity and performance comparable to CPU's found in minicomputers and mainframes [He84]. As the cost of hardware continues to drop, many computer manufacturers, and the scientific computing community in general, have begun to realize that further substantial gains in processing speed could be achieved by linking a number of cheaper processors together. The computing power of many individual processors can then be harnessed into a single, parallel multiprocessor computer.

The *Myrias Research Corporation* has taken this approach in the construction of a parallel computing system that allows a large number of heterogeneous tasks to be executed simultaneously. Their final aim is to develop a parallel processing system that has four main features [KVW87]:

scalability:	no *a priori* limit on performance should be imposed,
programmability:	the computer should be easy to use,
reliability:	failures should not interrupt the execution of user programs, and
stability:	user software should remain unchanged while the underlying hardware co-evolves with technology.

Unlike many of today's other parallel multiprocessor computers, the Myrias SPS-2 system architecture was derived from a high-level language extension (PARDO) and its new memory model. The basic idea is that correct results can be obtained by executing independent iterations of a looping construct simultaneously.

4.2. PARDO Extension and the Myrias Memory Model

Access to parallelism within the Myrias SPS-2 system is provided by a single PARDO extension to the high-level programming languages, *Fortran* and *C*. This PARDO extension 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 the current address space, and manages its own portion of the program, totally oblivious to 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* loop in *Fortran* (*for* loop in *C*) (see Figure 4.1). The fundamental idea behind this new memory model is to relieve memory contention at the price of memory replication [BKTJ89].

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.

4.3. Present Hardware Configurations

The Myrias SPS-2 system has a hierarchical architecture consisting of many processing elements (PEs). Presently one PE contains a Motorola 68020 CPU, 68851 MMU, 68882 FPU, 4 MBytes of memory, and a custom ASIC (*Application Specific Integrated Circuits*) that provides fast communication links between PEs and various supporting functions. Four PEs share a single bus on a multiple-processing element board with 16 MBytes of memory. Up to 16 multiple-processing boards share two 33 Mbytes/sec backplane buses in a card cage (Figure 4.2). Multiple cages can further be interconnected to build larger configurations. There is virtually no restriction on the size of a configuration one can build. Our experiments were done on a 64-PE Myrias SPS-2.

4.4. General Programming Considerations

At present the Myrias SPS-2 supports two high-level languages: *MPF* (*Myrias Parallel Fortran*) and *MPC* (*Myrias Parallel C*). The languages are standard, apart from the PARDO extension discussed in Section 4.2. The operation system available is a modified version of AT&T Bell Laboratories' UNIX.†

† Registered trademark of AT&T in the USA and other countries.

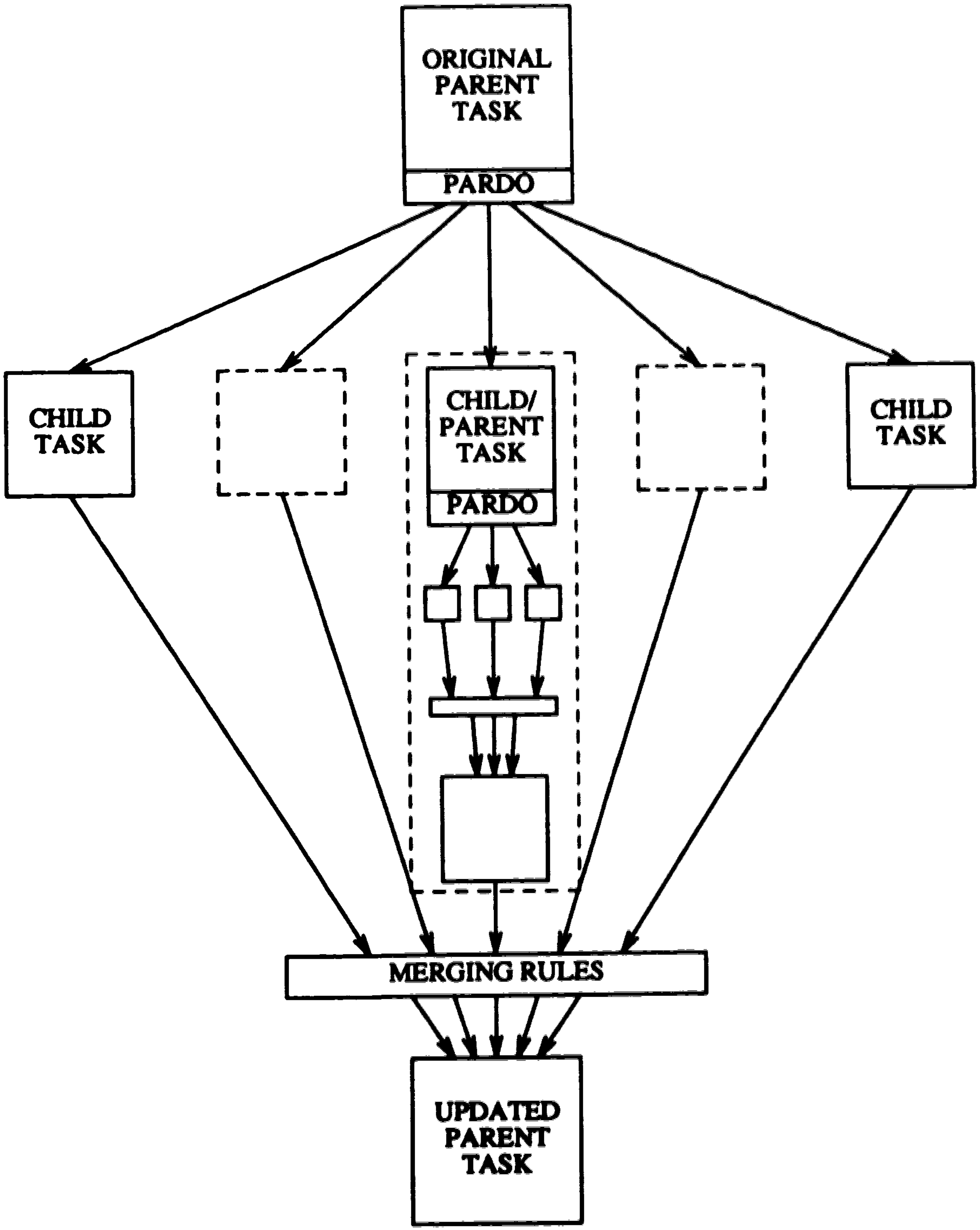

Figure 4.1 Fire up and merge parallel tasks [KVW87]

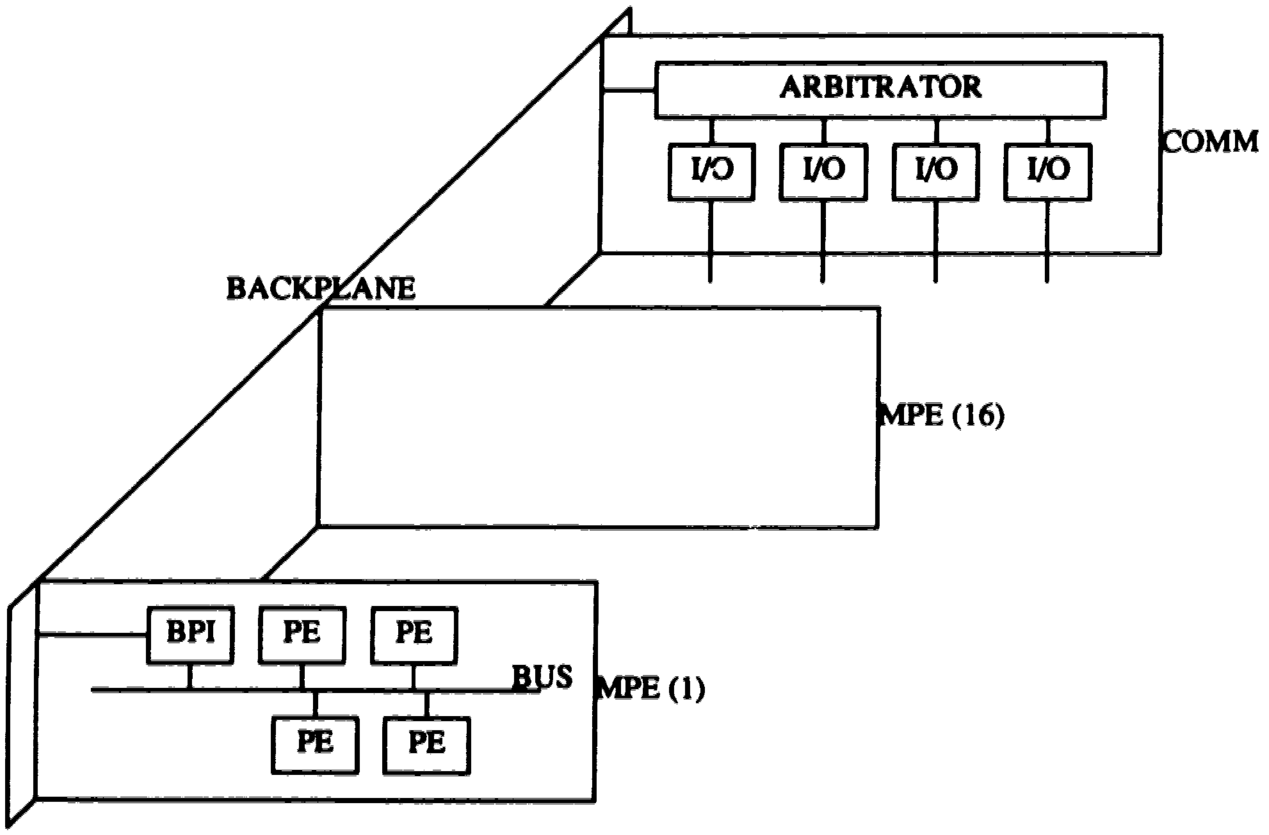

Figure 4.2 Component scheme of a Myrias card cage [KVW87]

Programming on the Myrias SPS-2 is much easier compared with many other multiprocessor computers, as users don't have to be aware of the underlying hardware configuration unless performance issues are concerned. PARDO is indeed a very powerful and flexible language facility. There are no restrictions on the number of "iterations" in a PARDO, nor on the depth of nesting of PARDO and recursive subroutines, nor on the requirement that parallel tasks should be homogeneous. Parallelism in the majority of problems can be naturally expressed with PARDO.

The simplest way to provide an adequate supply of tasks is to execute every piece of code in parallel and to make each PARDO contain as many child tasks as possible. Unfortunately such an approach may not always yield programs with good performance on the Myrias SPS-2. Rather, parallelism should not be overstated. Each PARDO requires some system management work, which includes child task creation and synchronization, and a certain amount of memory replication and merging. The Myrias SPS-2 has no central physical memory, therefore it has the requirement of reference locality similar to those of other virtual memory machines with caches. In order to make a PARDO worthwhile, each child task must perform an adequate amount of computing while at the same time read and write as few locations of different pages as possible. For this reason, it is often wise to put PARDOs on the outermost reaches of a program, so as to maximize the ratio of computing to data motion cost.

4.5. Notes on the Implementation

We have implemented PSRS and three parallel sorts discussed in Chapter 2 (i.e. PMS, Quickmerge, and PSS) in *MPC* on the 64-PE SPS-2. PQ and PTMS discussed in Chapter 2 were not implemented simply because they are both $O(n)$ bound, and are therefore unlikely to show better performance. The pseudo-codes of these four algorithms can be found in Appendix A1. The Myrias SPS-2 provides a transparent control mechanism that automatically schedules parallel tasks, optimizes resource usage, and manages data motion. This, however, has both advantages and disadvantages. The major advantage is that it simplifies programming. A user should not worry about the details of task allocation and data motion. One the other hand, this also makes it impossible for a user to have his or her own control schemes to execute one's specific program. It is not always pleasant since the system's general optimizer can't take advantage of the properties specific to a program.

4.6. Performance Results and Analysis

For each of the four algorithms implemented, we present a table of sorting times, and a table and a graph of speedups (see in the end of this chapter). The size of test data ranges from 0.1 million to 10 million integers. The numbers of processors utilized are 2, 4, 8, 16, 32, and 64. Each data item in the tables of sorting times was collected as the average of five executions, each on different test data of a given size. Specifically, the test data were generated in the following way: each processor i, where $1 \leq i \leq p$, generates in parallel a portion (approximately $\frac{n}{p}$) of the test data by the standard library routine *rand()* initialized with random seed $i \times seed$. *Rand()* here is a multiplicative congruential random number generator which returns successive pseudo-random 32-bit integers in the range from 0 to $2^{31} - 1$ with period 2^{32}. Different test data are generated through changing the value of $seed$.

An improved version of Quicksort has been used as the optimal sequential sort. More precisely, we made two common modifications. First, we used the median of the first, middle, and last elements of the subarray as the splitter. Second, we sorted subarrays with size less than ten integers using linear insertion sort. The same version was also used in the implementation of all the four parallel sorts. Unfortunately, one PE on the SPS-2 can only sort at most 0.2 million 32-bit integers due to the limited space of its 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 theoretical lower bound of sequential 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:

$$t_{PE}(n) = C\, n \log n \qquad (1)$$

where C is a constant *independent* of size n. The sequential times with a size larger than 0.2 million in all the tables of sorting times except the last one (in *italics*) are calculated according to the following formula, a simple derivation from (1):

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

where 0.4 million $\le n \le$ 10 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.

Table 4.1 shows the sorting times of PMS. Table 4.2 and Figure 4.3 show the speedups of PMS on the 64-PE Myrias SPS-2. There is almost no speed improvement at all compared with the sequential Quicksort. Some speedups are even less than one. The results are quite disappointing. The problem is found to be largely due to the poor performance of the second phase of the algorithm. Phase two of PMS has $\log p$ merge stages. Stage i, where $1 \le i \le \log p$, has $\frac{p}{2^i}$ independent parallel tasks. Each parallel task has two nested PARDO statements, each involving a large amount of memory replications and merging required by the Myrias memory model. The high overhead of the PARDO statements plus the relatively small computational granularity makes the parallelism nearly worthless. To get a sense how much a PARDO statement overhead would be, it is found that the time of executing the PARDO statement with 32 "iterations" of an empty loop body, which requires no memory replication and merging at all, is approximately 0.108 seconds, a significant amount of CPU time.

Table 4.3 shows the sorting times of Quickmerge. The speedups of Quickmerge can be found in Table 4.4 and Figure 4.4. The results are much better than PMS', especially when p is less than 16. However, when p increases from 16 to 32, there is almost no improvement in speedup. When more processors are added, the speedup declines. This can be explained by Table 4.5. *RDFA* in the table stands for *Relative Deviation* of the size of the largest one of the p partitions *From* the *Average* size of the p partitions, which is defined as follows:

$$RDFA = \frac{m - \frac{n}{p}}{\frac{n}{p}}$$

where m is the size of the largest one of the p partitions and $\frac{n}{p}$ the average size of the p partitions. Since $m \geq \frac{n}{p}$, it is always the case that $RDFA \geq 0$. The smaller the $RDFA$ is, the more balanced the partitioning is. Each data item in Table 4.5 is actually the average of the five executions. Table 4.5 shows that $RDFA$ increases with the number of the processors used. In other words, the larger the number of processors used is, the more imbalanced the partitioning becomes. This reduces the speedup since the final synchronization can't complete until the last processor finishes merging. The reasons behind this effect have already been explained in Section 2.4.2.

Table 4.7 and Figure 4.5 show the speedups of PSS. The sorting times of PSS are given in Table 4.6. Random sampling seems to be an effective approach to parallel sorting. An initial sample of $16 \times p$ elements randomly spread over the data seems to be large enough to provide a good representation of the data. The speedup increases with the value of p in general, yet much slower when p is larger than 24. This is mainly because when the number of processors used increases, so does the probability that some processor may have to merge significant more data than its share. This coincides with the values of $RDFA$ found in Table 4.8. Theoretically it is possible that some processor may have to perform a merge sort on nearly all the data.

Table 4.9 shows the sorting times of PSRS. The speedups of PSRS can be found in Table 4.10 and Figure 4.6. To get 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. We have a uniprocessor computer *Menaik*, which has a 32 MByte main memory and is able to sort 4 million 32-bit integers without any memory shortage. The machine is of RISC (*Reduced Instruction-Set Computer*) architecture and runs much faster than a single PE on the SPS-2. Without memory limitations, we would have the following approximate equation:

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

where n_1 and n_2 are the sizes of data to be sorted, $t_{Menaik}(n)$ represents the sequential sorting time of the improved Quicksort on *Menaik*.

The sequential sorting times in Table 4.12 (in *italics*) are calculated based on the above assumption, as are the speedups in Table 4.13 and Figure 4.7. Table 4.13 and Figure 4.7 give better speedups than Table 4.10 and Figure 4.6, especially when n is large. This is understandable. Because of memory limitations, when more data are being sorted, there is a greater chance that a page will stay outside of the main memory, and therefore more paging will result. The overhead of paging affects the time of sorting, *especially* when n is large compared with the size of the main memory. Due to the slower sequential time when n is larger, the resulting speedups in Table 4.13 and Figure 4.7 overstate to some extent the actual performance of the parallel sort[6].

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

[6] It should be pointed out that some researchers use the slower sequential time when n is large to calculate the speedup. Their results are inflationary.

(a) Both Figure 4.6 and Figure 4.7 show the same trend: with the increase of the number of PEs, the speedup increases are nearly linear, provided that the problem size is large enough to give each PE a sufficient amount of work to do. *When both the sizes of data and the number of PEs double, so does the speedup. The graphs demonstrate the first successful linear speedup parallel sort workable for parallel multiprocessors with a large number of processors.*

(b) Although we have used a conservative approach in computing the speedups in Table 4.10 and Figure 4.6, the results are still quite impressive. PSRS has a speedup of 31.73 when sorting 8 million 32-bit integers with 64 PEs. The trends of both the graphs suggest that better speedup can be expected if more than 64 PEs are utilized. PSRS sorts one million 32-bit integers in only 4.05 seconds using 64 PEs. It has been reported that a Cray X-MP, which is one of the most commercially successful supercomputers and costs almost 7 times more than a present 64-PE Myrias SPS-2, spends 4.96 seconds in sorting the same number of integers. On the average, PSRS achieves a speedup of roughly $\frac{p}{2}$ when p PEs are utilized.

(c) Table 4.11 shows PSRS is extremely stable. The size of the largest one of the p partitions is very close to the average, which indicates that the *regular sample* indeed provides a very good representation of the original data. Surprisingly, even when p is very small, the small *regular sample* is still able to help partition the data evenly. Theoretically, the *RDFA* of PSRS is always less than one.

(d) PSRS runs poorly on the SPS-2 when the size of the data is not large enough. Two factors may contribute to this. The first factor comes from the Myrias SPS-2. PSRS' implementation on the SPS-2 requires the use of at least three PARDO statements. The overhead of each PARDO statement is not an insignificant time. In order to gain some speedup, 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. The weight of this cost in the total sorting time increases with the decrease of the size of data, which reduces the speedup.

(e) The executions of PSRS on the Myrias SPS-2 show that more than half of its sorting time is due to the system activities. This explains why we get nearly 50% efficiency. Most of the system time is spent in memory replication and merging required by the Myrias SPS-2 new memory model. The rest is spent in task creation, scheduling, and synchronization. As sorting is in fact a data-intensive problem, it is not a "good" problem for the Myrias SPS-2 to solve. Nonetheless, 50% efficiency is still quite impressive, considering the fact that sorting is generally believed to be a hard problem to be parallelized.

4.7. Summary

In this chapter, we present the performance results of the four algorithms on the 64-PE Myrias SPS-2. The speedups obtained are highly reliable because the sequential time when n is large is extrapolated according to the sequential lower bound. In general, the speedup relationships of the four algorithms 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 [FrMa88]. It may largely be due to the Myrias SPS-2 architecture. One, however, should not understate the overhead of scheduling and synchronization during the second phase of the algorithm. The speedups of Quickmerge are much better than PMS', but not as good as PSS'. Random sampling seems to be an effective approach to parallel sorting. Since our test data are not "general", we are not quite sure if it works equally well on data from real-life applications.

The speedup of PSRS is the best among the four. The significance of the result is that it demonstrates the first successful parallel sort workable for multiprocessors with many processors.

Sizes	**Sorting Times** *(in seconds)*						
	1PEs	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	6.63	6.01	5.70	6.65	6.98	7.46	-
200000	14.00	12.95	11.26	13.16	13.10	13.12	-
400000	*29.71*	29.38	24.49	28.79	26.86	27.14	-
800000	*62.62*	-	62.71	61.76	59.03	56.14	55.85
1000000	*79.56*	-	-	81.71	75.04	69.12	69.43
2000000	*167.10*	-	-	-	162.13	162.43	155.73
4000000	*350.17*	-	-	-	-	321.14	333.32
8000000	*732.28*	-	-	-	-	-	641.32
10000000	*928.20*	-	-	-	-	-	812.57

Table 4.1 Sorting times of PMS

Sizes	**Speedups**					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	1.10	1.16	1.00	0.95	0.89	-
200000	1.08	1.24	1.06	1.07	1.07	-
400000	1.01	1.21	1.03	1.11	1.09	-
800000	-	1.00	1.01	1.06	1.16	1.12
1000000	-	-	0.97	1.06	1.15	1.15
2000000	-	-	-	1.03	1.03	1.07
4000000	-	-	-	-	1.09	1.05
8000000	-	-	-	-	-	1.14
10000000	-	-	-	-	-	1.14

Table 4.2 Speedups of PMS

Sizes	**Sorting Times** *(in seconds)*						
	1PEs	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	6.63	5.54	3.27	3.24	3.42	4.25	-
200000	14.00	11.29	6.18	4.93	4.37	5.59	-
400000	*29.71*	23.74	12.34	8.67	6.34	7.89	-
800000	*62.62*	-	26.27	16.73	10.33	10.26	19.12
1000000	*79.56*	-	40.68	20.89	12.46	11.35	19.98
2000000	*167.10*	-	-	-	24.93	18.64	25.39
4000000	*350.17*	-	-	-	-	33.63	34.69
8000000	*732.28*	-	-	-	-	-	75.72
10000000	*928.20*	-	-	-	-	-	92.56

Table 4.3 Sorting times of Quickmerge

Sizes	**Speedups**					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	1.20	2.03	2.05	1.94	1.56	-
200000	1.24	2.27	2.84	3.20	2.50	-
400000	1.25	2.41	3.43	4.69	3.77	-
800000	-	2.38	3.74	6.06	6.10	3.28
1000000	-	1.96	3.81	6.39	7.01	3.98
2000000	-	-	-	6.70	8.96	6.58
4000000	-	-	-	-	10.41	10.09
8000000	-	-	-	-	-	9.67
10000000	-	-	-	-	-	10.03

Table 4.4 Speedups of Quickmerge

Sizes	RDFAs					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	0.183	0.745	2.97	11.80	31.000	-
200000	0.092	0.369	1.501	5.584	23.493	-
400000	0.462	0.183	0.731	3.004	11.722	-
800000	-	0.096	0.369	1.493	5.943	23.170
1000000	-	0.078	0.296	1.195	4.773	18.761
2000000	-	-	-	0.603	2.397	9.410
4000000	-	-	-	-	1.207	4.757
8000000	-	-	-	-	-	2.371
10000000	-	-	-	-	-	1.995

Table 4.5 RDFAs of Quickmerge

Sizes	Sorting Times *(in seconds)*						
	1PEs	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	6.63	5.46	3.16	2.54	2.18	2.32	-
200000	14.00	11.36	6.15	4.61	3.30	3.21	-
400000	*29.71*	23.87	12.54	8.35	5.59	4.84	-
800000	*62.62*	-	27.50	16.67	9.61	7.61	8.17
1000000	*79.56*	-	39.61	21.16	11.81	8.90	9.00
2000000	*167.10*	-	-	-	24.03	14.96	13.53
4000000	*350.17*	-	-	-	-	29.66	21.22
8000000	*732.28*	-	-	-	-	-	38.29
1000000	*928.20*	-	-	-	-	-	48.62

Table 4.6 Sorting times of PSS

Sizes	Speedups					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	1.21	2.10	2.61	3.04	2.86	-
200000	1.23	2.28	3.04	4.24	4.36	-
400000	1.24	2.37	3.56	5.31	6.14	-
800000	-	2.28	3.76	6.52	8.23	7.66
1000000	-	2.01	3.76	6.74	8.94	8.84
2000000	-	-	-	6.95	11.17	12.35
4000000	-	-	-	-	11.80	16.50
8000000	-	-	-	-	-	19.12
1000000	-	-	-	-	-	19.09

Table 4.7 Speedups of PSS

Sizes	RDFAs					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	0.342	0.373	1.029	1.096	1.861	-
200000	0.339	0.349	0.722	0.685	1.515	-
400000	0.351	0.403	0.485	1.100	1.425	-
800000	-	1.264	0.441	1.339	1.150	1.001
1000000	-	0.649	1.348	0.996	1.044	1.635
2000000	-	-	-	0.578	1.104	1.447
4000000	-	-	-	-	1.365	1.670
8000000	-	-	-	-	-	1.407
1000000	-	-	-	-	-	1.560

Table 4.8 RDFAs of PSS

Sizes	Sorting Times *(in seconds)*						
	1PEs	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	6.63	5.41	2.96	2.10	1.45	1.42	-
200000	14.00	11.18	5.82	3.82	2.25	1.89	-
400000	*29.71*	23.35	11.97	7.48	4.10	2.71	-
800000	*62.62*	-	25.46	15.28	7.93	4.65	3.61
1000000	*79.56*	-	38.24	19.31	9.83	5.68	4.05
2000000	*167.10*	-	-	-	20.21	10.67	6.58
4000000	*350.17*	-	-	-	-	21.64	11.84
8000000	*732.28*	-	-	-	-	-	23.08
10000000	*928.20*	-	-	-	-	-	29.62

Table 4.9 Sorting times of PSRS

Sizes	Speedups					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	1.23	2.24	3.16	4.57	4.67	-
200000	1.25	2.40	3.66	6.22	7.41	-
400000	1.27	2.48	3.97	7.45	10.96	-
800000	-	2.45	4.10	7.90	13.47	17.35
1000000	-	2.08	4.12	8.09	14.01	19.64
2000000	-	-	-	8.27	15.66	25.40
4000000	-	-	-	-	16.18	29.58
8000000	-	-	-	-	-	31.73
10000000	-	-	-	-	-	31.33

Table 4.10 Speedups of PSRS

Sizes	RDFAs					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	0.001	0.008	0.021	0.030	0.074	-
200000	0.002	0.003	0.012	0.032	0.043	-
400000	0.001	0.002	0.008	0.017	0.044	-
800000	-	0.002	0.005	0.017	0.026	0.062
1000000	-	0.001	0.004	0.010	0.021	0.047
2000000	-	-	-	0.009	0.016	0.045
4000000	-	-	-	-	0.011	0.026
8000000	-	-	-	-	-	0.017
10000000	-	-	-	-	-	0.014

Table 4.11 RDFAs of PSRS

Sizes	**Sorting Times** *(in seconds)*						
	1PEs	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	6.63	5.41	2.96	2.10	1.45	1.42	-
200000	14.00	11.18	5.82	3.82	2.25	1.89	-
400000	*31.22*	23.35	11.97	7.48	4.10	2.71	-
800000	*67.17*	-	25.46	15.28	7.93	4.65	3.61
1000000	*85.97*	-	38.24	19.31	9.83	5.68	4.05
2000000	*181.45*	-	-	-	20.21	10.67	6.58
4000000	*422.46*	-	-	-	-	21.64	11.84

Table 4.12 Sorting times of PSRS (E)

Sizes	**Speedups**					
	2PEs	4PEs	8PEs	16PEs	32PEs	64PEs
100000	1.23	2.24	3.16	4.57	4.67	-
200000	1.25	2.40	3.66	6.22	7.41	-
400000	1.33	2.60	4.17	7.61	11.52	-
800000	-	2.64	4.40	8.47	14.45	18.61
1000000	-	2.25	4.45	8.75	15.14	21.23
2000000	-	-	-	8.98	17.01	27.58
4000000	-	-	-	-	19.52	35.68

Table 4.13 Speedups of PSRS (E)

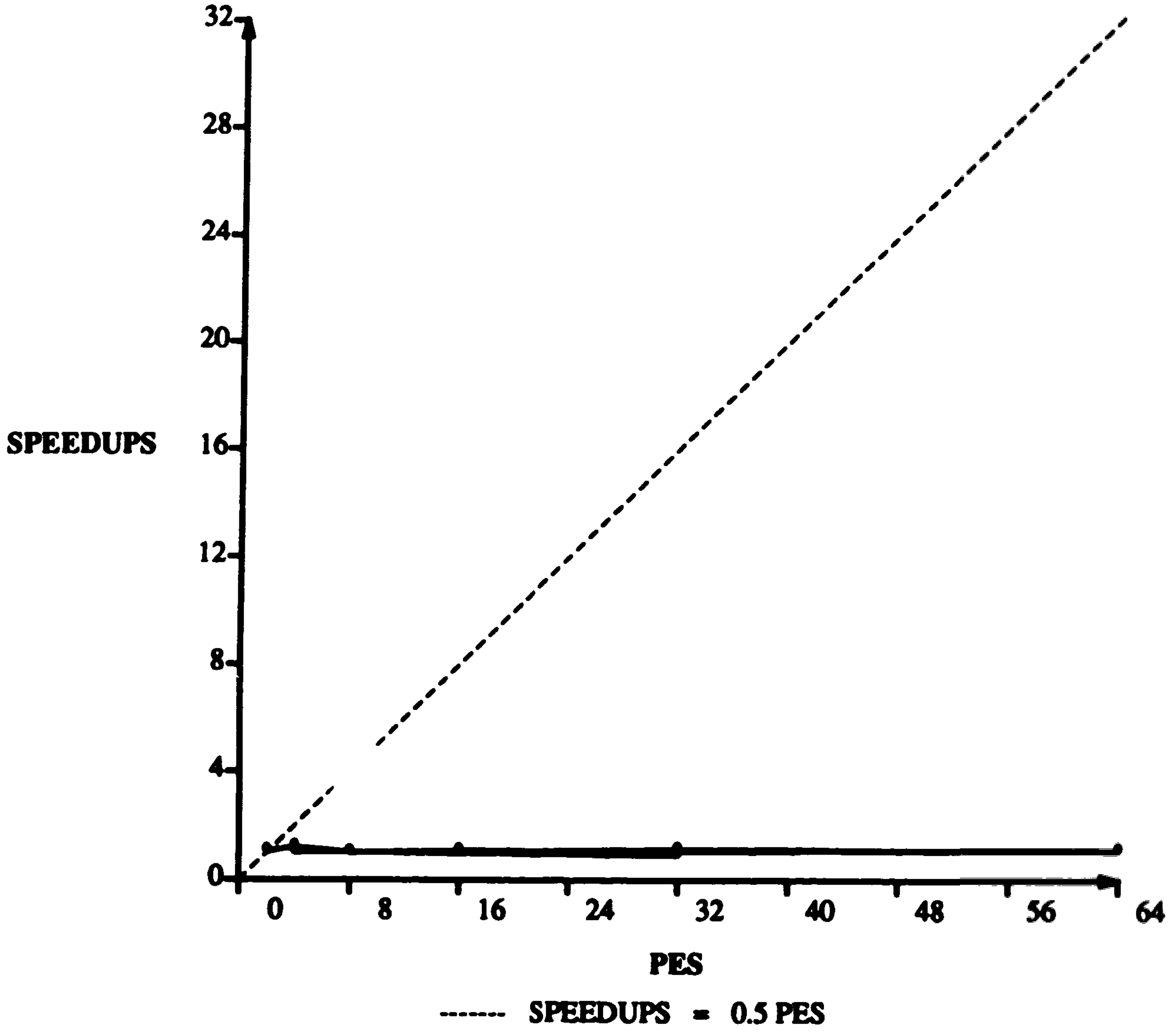

Figure 4.3 Speedups of PMS

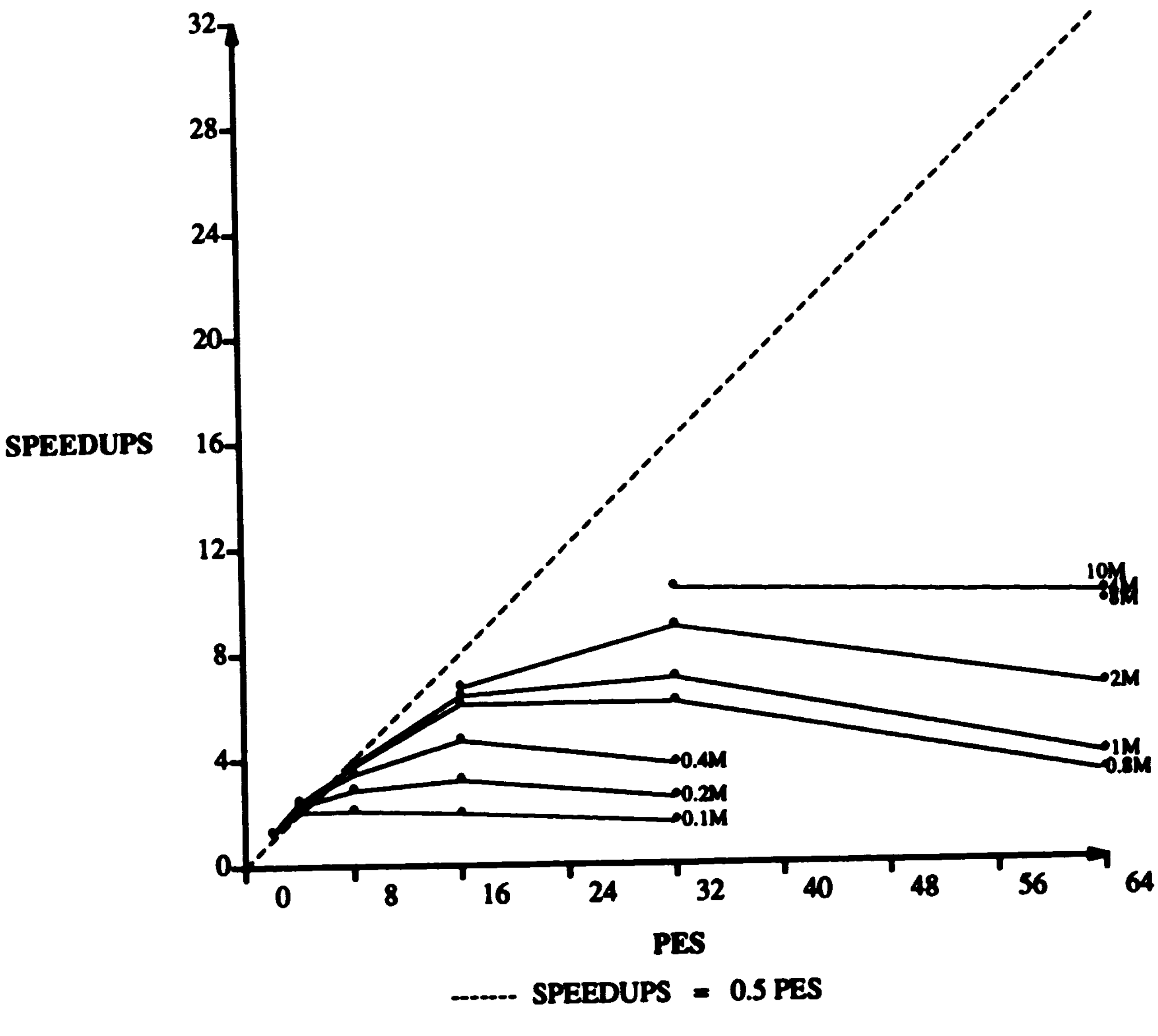

Figure 4.4 Speedups of Quickmerge

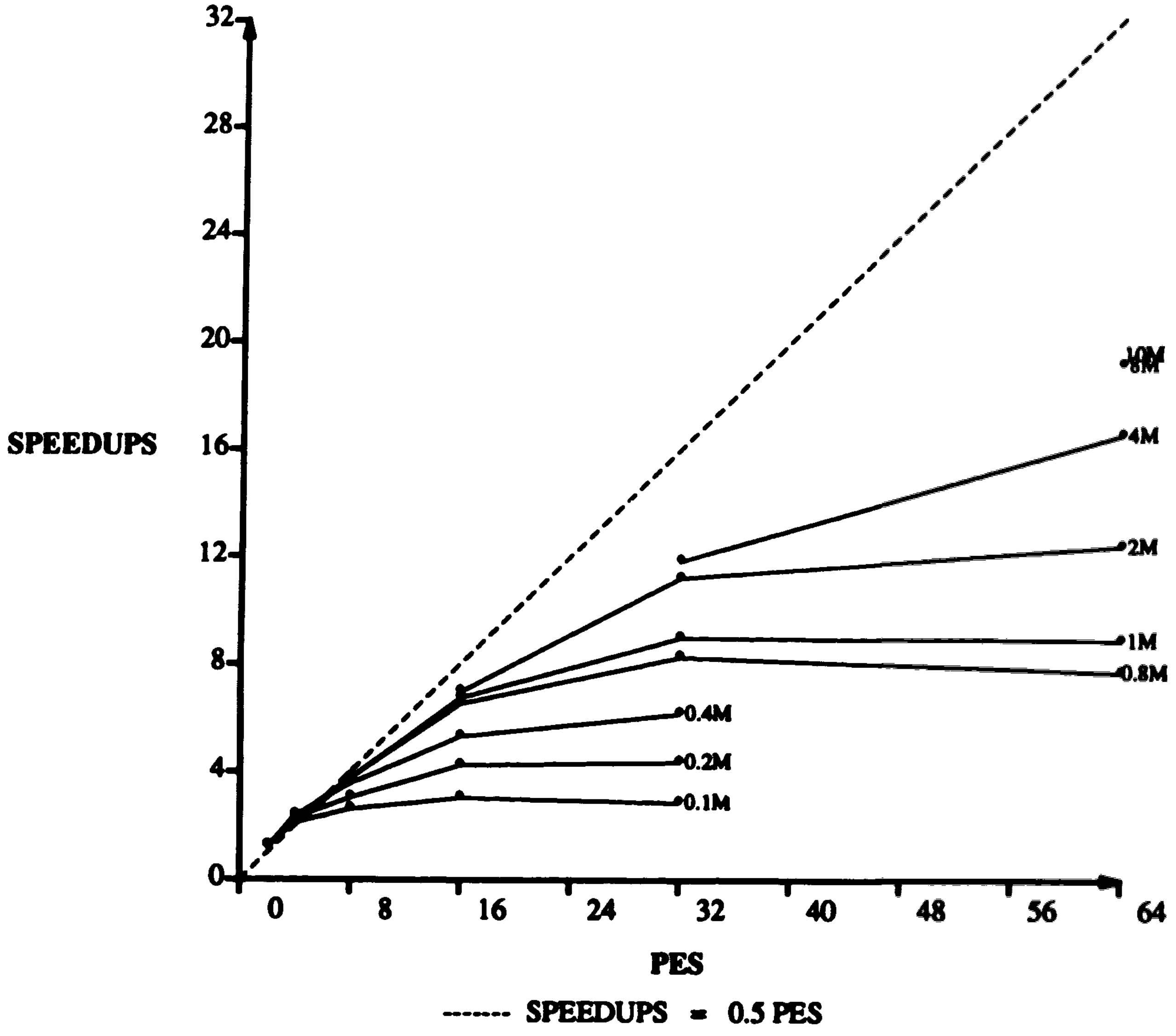

Figure 4.5 Speedups of PSS

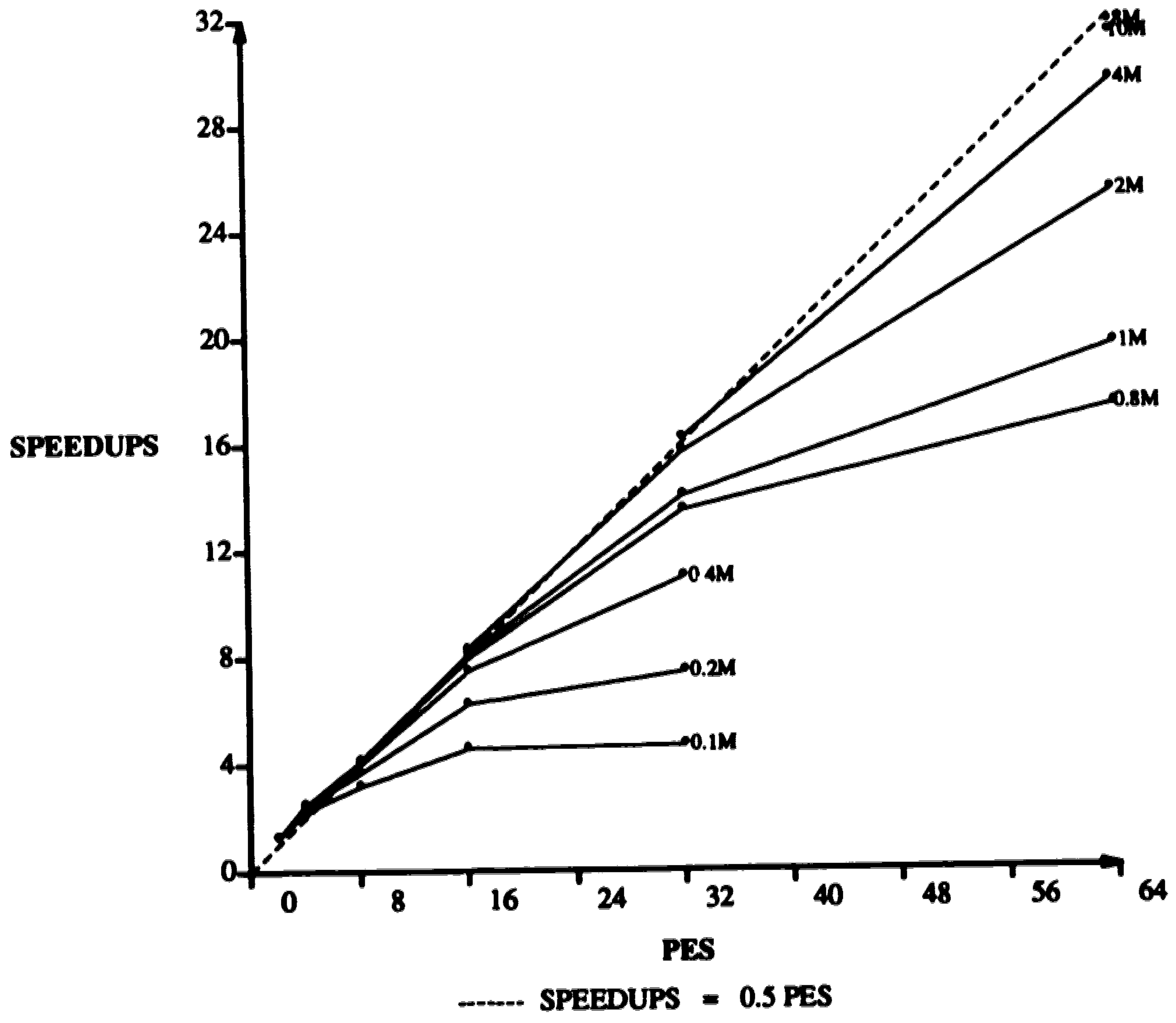

Figure 4.6 Speedups of PSRS

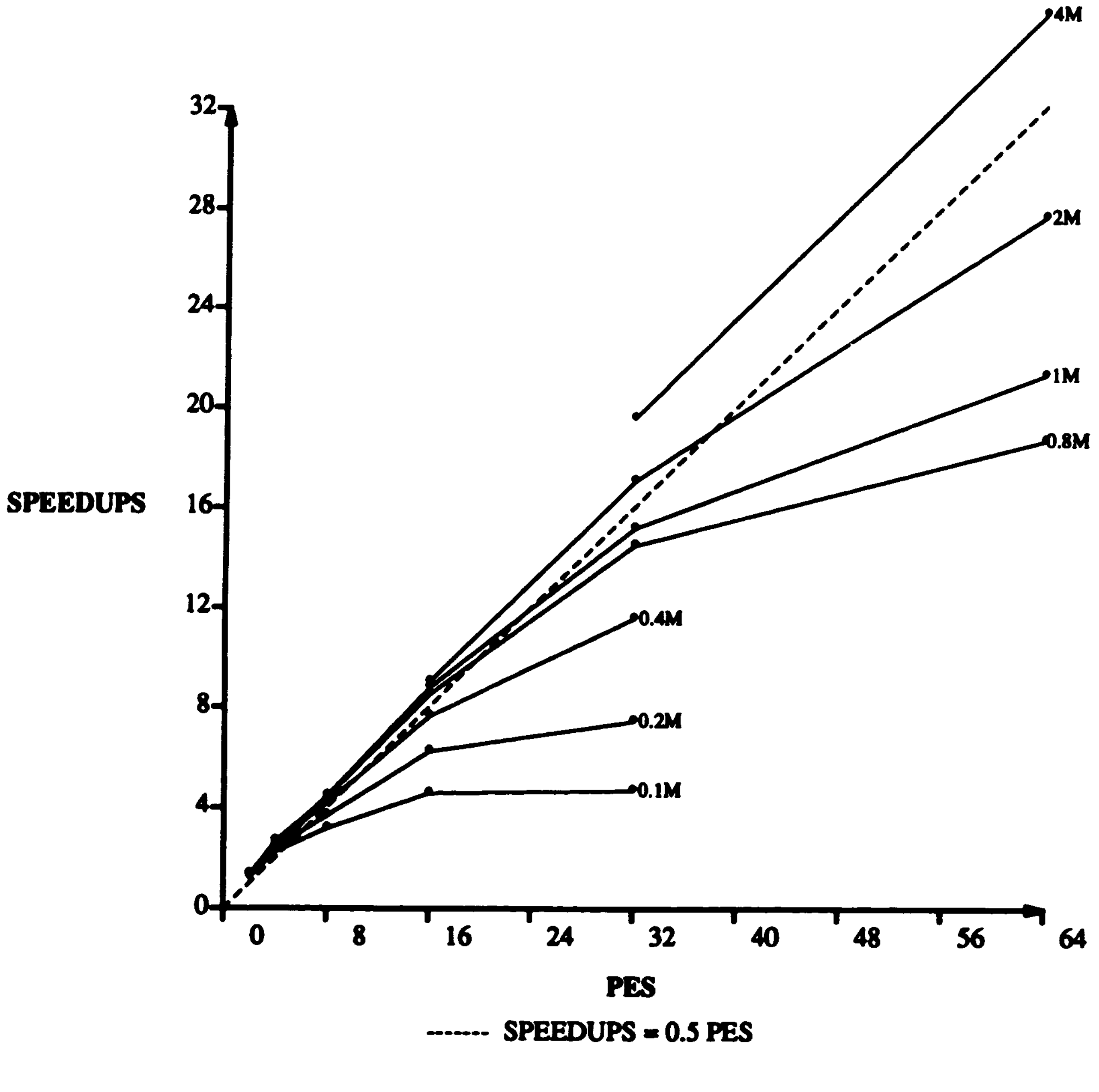

Figure 4.7 Speedups of PSRS (E)

Chapter 5
Extensions of PSRS on Hypercubes and LANs

PSRS has been implemented on the Myrias SPS-2, whose memory model is quite different from those of other multiprocessors. This, however, does not mean the algorithm is suitable only for this architecture. Rather, the essence of the algorithm makes it equally applicable to many other architectures. Two extensions at least can be made to the work presented.

5.1. Extension of PSRS on Hypercube Computers

A hypercube computer [KaKa89] is a multiprocessor in which the (processor) nodes can be imagined to lie at the vertices of a multi-dimensional cube.[7] Multiprocessors based on this topology require modest interconnections, yet seem rich enough to allow many of the classical interconnections to be easily constructed. For this reason, hypercube computers have emerged as the most promising multiprocessors of its kind. Another appealing feature of hypercubes is their homogeneity and symmetry properties. Despite of these facts, sorting on hypercube computers seems a challenge problem [Wa87, SeZi87, Jo84].

Assume p is the number of nodes of a hypercube and n is the number of data to be sorted, where p is a power of 2 and p^2 divides n. 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. We also assume the data are initially distributed evenly at all the p nodes. The n elements are said to be sorted if all elements are sorted at each node, and each element at one node is no less (greater) than any element at another

[7] The author assumes the reader has some knowledge of hypercube computers.

node.

For a sort for hypercubes, its complexity is usually measured in terms of computing cost as well as communication cost. Presently, Johnsson's algorithm [Jo84], which is an adaptation of Batcher's bitonic sorting algorithm [Ba68], is considered the fastest worst case algorithm for hypercubes in theory. His algorithm achieves $O(\frac{n}{p}\log\frac{n}{p} + \frac{n}{p}\log^2 p)$ computing complexity and $O(\frac{n}{p}\log^2 p)$ communication complexity.

PSRS appears easily adaptable to hypercubes. Here is a hypercube version of PSRS.

Phase 1. Each node sorts in parallel its local list of size $\frac{n}{p}$ with sequential Quicksort. After that, each one sends p elements evenly spaced from its local sorted list to one designated node, say node 0.

Phase 2. After it receive all p^2 elements, node 0 first selects $p - 1$ pivots with the same method as described in Section 3.2, then broadcasts the chosen pivots to all other nodes. After each node receives $p - 1$ pivots, a binary search is used to partition each local list into p sublists.

Phase 3. Each node i, $0 \le i \le p - 1$, sends each of its (j+1)th sublists to nodes j, $j = 0, 1, \cdots p - 1, j \ne i$.

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

Note that no 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 therefore $O(p^2 \log p^2 + \frac{n}{p} \log p + \frac{n}{p} \log n)$, which is asymptotic to $O(\frac{n}{p} \log n)$ when $n \geq p^3$. The computing time alone is not sufficient to distinguish a parallel sort for hypercubes. In order to study the communication complexity of this algorithm we use the following model [SaSc89]:

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

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

The algorithms and estimates for the times of data transfers required in phase 1, phase 2, and phase 3 are sketched below. Exponents applied to the binary bits 0 and 1 stand for concatenation. Other details can be referred to [SaSc89].

Algorithm for data gathering in phase 1:

For $j = \log p, \log p - 1, \ldots, 1$ 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 $0^{\log p - j + 1} a_j$.

The algorithm consists of a total of $\log p$ steps. At the jth step, the algorithm transfers $p\,2^{j-1}$ 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

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

Algorithm for data broadcasting in phase 2:

For $j = \log p, \log p - 1, \ldots, 1$ do:

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

The above algorithm consists of a total of $\log p$ steps all requiring the same amount of time. Hence the total time of broadcasting $p - 1$ pivots in phases 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 we see the $(j+1)$th sublist at node i 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 $\overline{i}^k$ 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 equation and inequalities:

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

$$\sum_{j=0}^{p-1} |L_{i,j}| = \frac{n}{p}, \quad 0 \le i \le p - 1 \tag{2}$$

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

Equation (2) and inequality (1) are trivial. The proof of inequality of (3) can be found in Section 3.3.

Algorithm for data exchange operations in phase 3:

For $k = 1, 2, \ldots, \log p$ 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 $(j)_k = 0$.

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 in the similar way as with the original matrix, and the total number of the sublists exchanged is still p. To obtain the time bound of each step, we introduce the following lemmas:

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

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

Proof: Straightforward from *Lemma 1*, equation (2), and inequality (3).

Hence step k costs less than or equal to

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

The upper bound of the total communication time of phase 3 is

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

Upon summation over three phases, we get the 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 start-up time, while the latter one stands for the total transmission time. The total start-up time is determined only by p and has a relatively low order. When $n \gg p$, 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 here no assumption is made to allow a node to send (receive) data to (from) its neighbors simultaneously.

Summarizing, the extension of PSRS on a hypercube computer has $O(\frac{n}{p}\log n)$ computing complexity and $O(\frac{n}{\sqrt{p}})$ communication complexity, where $n \gg p$. Compared with Johnsson's algorithm, it has a better computing complexity. Besides, the start-up time of the PSRS is much less compared with Johnsson's.

5.2. Extension of PSRS on LANs

The last decade has witnessed the explosive growth of LANs (*Local Area Networks*) [Ta88][8]. Large files on such a network may be physically distributed over a number of stations. The problem of sorting a distributed file is to relocate some of the data items so

[8] Again, the author assumes the reader has some knowledge of LANs.

that each data item of a subfile at one station will be less (or greater) than any data item of a subfile at another. More precisely, the problem can be stated as follows:

A file X of size n is 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 $\frac{n}{p}$ 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 [Ch89, Ro85, We84]. Two of them are worthy to be mentioned. One is attributed to Wegner [We84]. His algorithm is based on the Quicksort approach. In each iteration, every partition is divided further into two smaller ones. On the average, the algorithm has message complexity $O(p \log p)$ and data traffic complexity $O(n \log p)$.

Another one is attributed to Rotem [Ro85]. His algorith.n starts with finding $p-1$ even partition points by applying a distributed algorithm for the kth selection [Fr83] in a straightforward manner. 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 great amount of data traffic, while Rotem's algorithm requires too many messages.

Although PSRS is developed for sorting on a multiprocessor, the algorithm can be applied to distributed sorting in a straightforward manner. As has been analyzed in Section 3.2, the algorithm has $O(n)$ data traffic complexity and $O(p^2)$ message complexity, which is a substantial improvement of the $O(p^2 \log n)$ worst case of Rotem's algorithm.

Chapter 6
Conclusions

6.1. Thesis Summary

During the last two decades parallel sorting has been an area of active research. Much of the work has been concerned with theoretical issues. In this thesis, the performance problem of parallel sorting on multiprocessor computers is studied. The speedup of a parallel sort achievable on a multiprocessor depends largely on how well we can minimize the average memory latency and the overhead of scheduling and synchronization. Many parallel sorts suitable for multiprocessors fall into one of two rough categories: merge-based sorts and partition-based sorts. Merge-based sorts consist of multiple stages of merge, and are generally believed to perform well only with a small number of processors. Partition-based sorts consist of two phases: partitioning the data into smaller subsets and then sorting each in parallel. The performance of merge-based sorts depends largely on how well the data are partitioned. Previous partition-based sorts are unstable in the sense that one processor may have to sort significantly more elements than its average share. This reduces the speedup since the final synchronization can't complete until the last processor finishes sorting. PSRS is proposed by the author to solve this problem. The basic idea is to make use of the order information of a small set of elements evenly spaced from the locally ordered data to help partitioning. Theoretically PSRS is optimal when $n \geq p^3$. In particular, we show that each processor has to access only a small portion (less than $\frac{2n}{p}$) of the data and the accesses are highly localized. In our experiments on the 64-PE Myrias SPS-2, it defeats the best previous proposed parallel sort known to the author. On the average, PSRS achieves a speedup of nearly $\frac{p}{2}$ when

p PEs are utilized. The speedups obtained are highly reliable since the sequential time when n is large is computed according to the sequential sorting lower bound $O(n \log n)$. The experiments demonstrate the first successful parallel sort workable on multiprocessors with many processors. The basic idea developed is equally applicable to hypercube computers and LANs.

6.2. General Conclusions

Parallel processing has emerged as an active field of research and development by computer professionals. Various classes of parallel and vector supercomputers have appeared in the past two decades. However, the claimed performance may not be delivered as promised by the vendors. Usually the effective performance of a current supercomputer ranges between only 5% and 25% of its peak performance. Such a pessimistic show of the delivered performance motivates many of us to search for better algorithms, languages, software and hardware techniques to yield higher performance.

Sorting is perhaps one of most studied problems in computing science. Many parallel algorithms which are theoretically optimal have been proposed. Despite this, their experimental results have been rather bleak.

Parallelism should not be overstated on the present multiprocessor architectures. Long memory latency and the overhead of scheduling and synchronization are two critical factors that greatly affect the speedup of a parallel algorithm on such architectures. PSRS is intended to minimize both. It has a high per-task reference locality, yet is very simple to schedule and synchronize. Our experiments on the 64-PE Myrias SPS-2 make it clear that linear speedup for parallel sorting on the present multiprocessor architecture is indeed achievable. The obtained 50% efficiency is also quite encouraging, considering that sorting is generally believed a hard problem to be parallelized. The success of PSRS indicates that better performance for the current supercomputers are obtainable.

References

[Aj83] M. Ajtai, "An O(*n* log *n*) Sorting Network," *Proceedings of the 15th Annual ACM Symposium on Theory of Computing* (Boston, Apr. 25-27), ACM, New York, pp. 1-9, 1983.

[Ba68] K. E. Batcher, "Sorting Networks and Their Applications," *Proceedings of the 1968 Spring Joint Computer Conference*, Vol. 32. AFIPS Press, Reston, Va., pp. 307-314, 1968.

[BaTa88] H.E. Bal and A.S. Tanenbaum, "Distributed Programming with Shared Data," *IEEE Conf. on Computer Languages*, IEEE, pp. 82-91, 1988.

[BDHM84] D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon, "A Taxonomy of Parallel Sorting," *ACM Computing Surveys*, Vol. 16, No. 3, pp. 287-318, Sept. 1984.

[BKTJ89] H.E. Bal, M.F. Kaashoek, A.S. Tanenbaum, and J. Jansen, "Replication Techniques for Speeding up Parallel Applications on Distributed Systems," Report IR-202, Dept. of Mathematics and Computer Science, Vrije, Oct. 1989.

[CRR86] S. Cook, C. Reischuk and R. Reischuk, "Upper and Lower Bounds For Parallel Random Access Machines Without Simultaneous Writes," *SIAM J. Computers*, Vol. 15, No. 1, pp. 87-97, Feb. 1987.

[Ch89] T. Y. Cheung, "An Algorithm with Decentralized Control for Sorting Files in a Network," *Journal of Parallel and Distributed Computing*, Vol. 7, pp. 464-481, 1989.

[Co86] R. Cole, "Parallel Merge Sort," *FOCS*, 1986.

[De82] J. Deminet, "Experience with Multiprocessor Algorithms," *IEEE Trans. on Computers*, Vol. C-31, pp. 278-288, Apr. 1982.

[DoDu89] J. J. Dongarra and I. S. Duff, "Advanced Architecture Computers," Report CS-89-90, Department of Computing Science, University of Tennessee, Tennessee, 1989.

[EgKa88] S. J. Egger and R. H. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," Report No. UCB/CSD 88/475, University of California, 1988.

[EvYo85] D. J. Evans and N. Y. Yousif, "Analysis of the Performance of the Parallel Quicksort Method," *BIT 25*, pp. 106-112, 1985.

[EvYo86] D. J. Evans and N. Y. Yousif, "The Parallel Neighbor Sort and Two-way Merge Algorithm," *Parallel Computing*, Vol. 3, pp. 85-90, 1986.

[Fe81] T.Y. Feng, "A Survey of Interconnection Networks," *Computer 14*, Dec. 1981.

[Fr83] G. N. Frederickson, "Tradeoffs for Selection in Distributed Networks," *Proc. 2nd ACM Symposium on the Principles of Distributed Computing*, Montreal, Quebec, Canada, pp. 154-160 , 1983.

[FrMa88] R. S. Francis and I. D. Mathieson, "A Benchmark Parallel Sort for Shared Memory Multiprocessors," *IEEE Trans. on Computers*, Vol. 37, No. 12, pp. 1619-1626, Dec. 1988.

[GiRy88] A. Gibbons and W. Rytter, "Efficent Parallel Algorithms," Cambridge University Press, Cambridge, 1988.

[He84] J. L. Hennessy, "VLSI Processor Architecture," *IEEE Trans. on Computers*, Vol. C-33, No. 12, pp. 1221-1246, Dec. 1984.

[Hi78] D. S. Hirschberg, "Fast Parallel Sorting Algorithms," *Comm. ACM 21*, pp. 657-666, Aug. 1978.

[Ho61] C.A.R. Hoare, "Partition: Algorithm 63; Quicksort: Algorithm 64; and Find: Algorithm 65," *Comm. ACM*, Vol. 4, No. 7, pp. 321-322, 1961.

[HuCh83] J. S. Huang and Y. C. Chow, "Parallel Sorting and Data Partitioning by Sampling," *COMPSC 83*, pp. 627-631, 1983.

[Hw87] K. Hwang, "Advanced Parallel Processing with Supercomputer Architectures," *Proceeding of the IEEE*, Vol. 75, No. 10, pp. 1348 1377, Oct. 1987.

[Jo84] S. L. Johnsson, "Combining Parallel and Sequential Sorting on a Boolean n-cube", *ICPP*, 1984.

[KaKa89] S. P. Kartashev and S. I. Kartashev, "Supercomputing Systems: Architectures, Design, and Performance," Van Nostrand Reinhold Inc., New York, 1989.

[Kn73] D. E. Knuth, "The Art of Computer Programming: Vol. 3, Sorting and Searching," Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

[KVW87] A.M. Kobos, R.E. VanKooten, and M.A. Walker, "The Myrias Parallel Computer System," *Algorithms and Applications on Vector and Parallel Computers*, Th. J. Decter and H.A. van der Vorst (Ed.), pp. 103-127, Elsevier Science Publishers B.V. (North-Holland), 1987.

[LaEl90] R. P. LaRowe Jr. and C. S. Ellis, "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," Report No. CS-90-10, Duke University, 1990.

[Lo74] R. Loeser, "Some Performance Tests of Quicksort and Descendants," *Comm.*

ACM, Vol. 17, No. 3, pp. 143-152, 1974.

[Me84] K. Mehlorn, "Sorting and Searching," Springer-Verlag, Berlin, 1984.

[MoSt87] P. Moller-Nielsen and J. Staunstrup, "Problem-heap: A Paradigm for Multiprocessor Algorithms," *Parallel Computers*, Vol. 4, pp. 63-74, Feb. 1987.

[Pe77] M. C. Pease, "The Indirect Binary *n*-Cube Microprocessor Array," *IEEE Trans. on Computers*, Vol. 31, No. 2, Feb. 1977.

[Pr78] F. P. Preparata, "New Parallel Sorting Schemes," *IEEE Trans. on Computers*, Vol. 27, pp. 669-673, 1978.

[Qu87] M. J. Quinn, "Designing Efficient Algorithms for Parallel Computers," McGraw-Hill Book Company, New York, 1987.

[Qu88] M. J. Quinn, "Parallel Sorting Algorithms for Tightly Coupled Multiprocessor," *Parallel Computing 6*, North-Holland, pp. 349-357, 1988.

[Ri86] D. Richards, "Parallel Sorting -- A Bibliography," *ACM SIGACT News Summer*, pp. 24-28, 1986.

[Ro85] D. Rotem, N. Santoro, and J. B. Sidney, "Distributed Sorting," *IEEE Trans. on Computers*, Vol. 34, No. 4, pp. 372-275, 1985.

[SaSc89] Y. Saad and M. H. Schultz, "Data Communication in Hypercubes," *Journal of Parallel and Distributed Computing*, Vol. 6, No. 1, pp. 115-135, 1989.

[SeZi87] S. Seidel and L. R. Ziegler, "Sorting on Hypercubes," *Hypercube Multiprocessors*, M. T. Heath (Ed.), *SIAM*, pp. 285-291, 1987.

[Si79] H. J. Siegel, "Interconnection Networks for SIMD Machines," *IEEE Computers 12*, Jun. 1979.

[Sm87] B. J. Smith, "Shared Memory, Vectors, Message Passing, and Scalability," *Lecture Notes in Computer Science 295*, G. Goos (Ed.), pp. 29-34, Springer-

Verlag, Berlin, 1987.

[Ta88] A. S. Tanenbaum, "Computer Networks," Prentice-Hall, Inc., New Jersey, 1988.

[ThKu77] C. D. Thompson and H. T. Kung, "Sorting on A Mesh-connected Parallel Computer," *Comm. ACM 20*, 4, pp. 263-271, 1977.

[Va75] L. G. Valliant, "Parallelism in Comparison Problems," *SIAM J. Computers 3*, 4, Sept. 1975.

[Wa87] B. Wagar, "Hyperquicksort: A Fast Sorting Algorithm for Hypercubes," *Hypercube Multiprocessors*, M. T. Heath (Ed.), *SIAM*, pp. 292-299, 1987.

[We84] L. M. Wegner, "Sorting a Distributed File in A Network," *Computer Networks* 8, pp. 451-461,)84.

Appendix A1

Pseudo-codes of PMS, Quickmerge, PSS, and PSRS

A1.1 Parallel Merge Sort

```
procedure pms($array$, $n$, $p$)
//$array$[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
    //Store the start-points and end-points of all the lists in $bk1$ and $bk2$ respectively
    $gn = \lfloor (n+p-1)/p \rfloor$
    $ggn = \lfloor (gn+p-1)/p \rfloor$
    for $i = 0$ to $p-1$ do in parallel
        $start = i \times gn$
        $end = (i+1) \times gn - 1$
        if $end \geq n$ then $end = n-1$
        $bk1[i] = start$
        $bk2[i] = end$
        //Sort subarray $array[start:end]$ with sequential Quicksort
        quicksort($array$, $start$, $end$)
    endfor
    $k = p$
    //Pairs of sorted subarrays are merged into longer ones
    while $k > 1$ do
        if $k$ mod $2 \neq 0$ then
            $bk1[k] = 0$
            $bk2[k] = -1$
            $k = k+1$
        endif
        for $i = 1$ to $k$ step 2 do in parallel
            parmerge($array$, $bk1[i-1]$, $bk2[i-1]$, $bk1[i]$, $bk2[i]$, $array1$, $bk1[i-1]$, $\lfloor 2 \times p/k \rfloor$)
            $bk1[(i-1)/2] = bk1[i-1]$
            $bk2[(i-1)/2] = bk2[i]$
        endfor
        $k = k/2$
        if $k$ mod $2 \neq 0$ then
            $bk1[k] = 0$
            $bk2[k] = -1$
            $k = k+1$
        endif
        for $i = 1$ to $k$ step 2 do in parallel
            parmerge($array1$, $bk1[i-1]$, $bk2[i-1]$, $bk1[i]$, $bk2[i]$, $array$, $bk1[i-1]$, $\lfloor 2 \times p/k \rfloor$)
            $bk1[(i-1)/2] = bk1[i-1]$
            $bk2[(i-1)/2] = bk2[i]$
        endfor
        $k = k/2$
    endwhile
end
```

```
procedure parmerge($array1, from1, to1, from2, to2, array2, at, np$)
//Merge two sorted lists, $array1[from1:to1]$ and $array1[from2:to2]$, with $np$ processors in parallel
//The merged results are stored in $array2$ starting from index $at$
begin
    //Divide the two lists into $np$ pairs of sublists:
    //$array1[pbk1[i]:pbk1[i+1]-1]$ & $array1[pbk2[i]:pbk2[i+1]-1], 0 \le i \le np-1$
    $avl = \lfloor (to1-from1+from2-to2+2)/np \rfloor$
    for $i = 0$ to $np-1$ do in parallel
        $lb = \max\left\{from1, from1+i \times avl-(to2-from2+1)\right\}$
        $ub = \min\left\{to1+1, from1+i \times avl\right\}$
        $length = i \times avl + from1 + from2$
        while $lb+1 < ub$ do
            $k = (lb+ub)/2$
            if $array[k] \le array[length-k]$
                then $lb = k$
                else $ub = k$
            endif
        endwhile
        if $array1[lb] \le array1[length-ub]$
            then $pbk1[i] = ub$
            else $pbk1[i] = lb$
        endif
        $pbk2[i] = length - pbk1[i]$
    endfor
    $pbk1[np] = to1+1$
    $pbk2[np] = to2+1$
    //Merge each pair in parallel
    for $i = 0$ to $np-1$ do in parallel
        merge two sublists, $array1[pbk1[i]:pbk1[i+1]-1]$ and $array1[pbk2[i]:pbk2[i+1]-1]$. The
        merged results are stored in $array2$ starting from index $at+pbk1[i]-from1+pbk2[i]-from2$
    endfor
end
```

A1.2 Quickmerge

```
procedure qm($array$, $n$, $p$)
//$array[0{:}n-1]$: 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
    $gn = \lfloor (n+p-1)/p \rfloor$
    $ggn = \lfloor (gn+p-1)/p \rfloor$
    for $i = 0$ to $p-1$ do in parallel
        $start = i \times gn$
        $end = (i+1) \times gn - 1$
        if $end \geq n$ then $end = n-1$
        //Sor subarray $array[start{:}end]$ with sequential Quicksort
        quicksort($array$, $start$, $end$)
    endfor
    //Choose $p-1$ elements evenly spaced from the first sorted list as pivots
    //Store them in $pivots[1{:}p-1]$
    for $i = 1$ to $p-1$ do
        $pivots[i] = array[i \times ggn]$
    endfor
    //Divide in parallel each sorted list $i$ into $p$ sublists:
    //$array[subsize[i \times (p+1)+j] : subsize[i \times (p+1)+j+1]-1]$,
    //$0 \leq j \leq p-1$ with the chosen pivots as splitters
    for $i = 0$ to $p-1$ do in parallel $start = i \times gn$
        $end = (i+1) \times gn - 1$
        if $end \geq n$ then $end = n-1$
        $subsize[i \times (p+1)] = start$
        $subsize[i \times (p+1)+p] = end+1$
        sublists($array$, $start$, $end$, $subsize$, $i \times (p+1)$, $pivots$, 1, $p-1$)
    endfor
    //Count the size of each of the $p$ partitions
    for $i = 0$ to $p-1$ do
        $bucksize[i] = 0$
        for $j = i$ to $p \times (p+1)-1$ step $p+1$ do
            $bucksize[i] = bucksize[i] + subsize[j+1] - subsize[j]$
        endfor
    endfor
    //Decide the start-point of each partition in the final array
    for $i = 1$ to $p-1$ do
        $bucksize[i] = bucksize[i] + bucksize[i-1]$
    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 two-way mergesort $0 \leq j \leq p-1$:
```

```
        $array[subsize[i+j \times (p+1)] : subsize[i+j \times (p+1)+1]-1]$. The merged results are stored in array
        starting from index bucksize[i]
    endfor
end
```

A1.3 Parallel Sorting by Sampling

```
procedure pss(array, n, p)
//array[0:n-1]: array to be sorted, n: size of the array, p: number of processors
begin
    //Draw a random sample of size $16 \times p$ from array and store them in subsize[0: $16 \times p-1$]
    for $i = 0$ to $16 \times p-1$ do
        $subsize[i] = array[rand() \bmod n]$
    endfor
    //Sort the sample with sequential Quicksort
    quicksort(subsize, 0, $16 \times p-1$)
    //Choose $p-1$ elements evenly spaced from the sorted sample as pivots
    //Store them in pivots[1:$p-1$]
    $pivots[i] = subsize[i \times 16]$
    //Divide the array into p contiguous lists and sort each in parallel
    //Divide in parallel each sorted list i into p sublists:
    //array[subsize[$i \times (p+1)+j+1$] : subsize[$i \times (p+1)+j$]-1],
    // $0 \le j \le p-1$ with the chosen pivots as splitters
    $gn = \lfloor (n+p-1)/p \rfloor$
    $ggn = \lfloor (gn+p-1)/p \rfloor$
    for $i = 0$ to $p-1$ do in parallel
        $start = i \times gn$
        $end = (i+1) \times gn - 1$
        if $end \ge n$ then $end = n-1$
        //Sort subarray array[start:end] with sequential Quicksort
        quicksort(array, start, end)
        $subsize[i \times (p+1)] = start$
        $subsize[i \times (p+1)+p] = end+1$
        sublists(array, start, end, subsize, $i \times (p+1)$, pivots, 1, $p-1$)
    endfor
    //Count the size of each of the p partitions
    for $i = 0$ to $p-1$ do
        $bucksize[i] = 0$
        for $j = i$ to $p \times (p+1)-1$ step $p+1$ do
            $bucksize[i] = bucksize[i]+subsize[j+1]-subsize[j]$
        endfor
    endfor
    //Decide the start-point of each partition in the final array
    for $i = 1$ to $p-1$ do
        $bucksize[i] = bucksize[i]+bucksize[i-1]$
    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 two-way mergesort:
        array[subsize[$i+j\times(p+1)$] : subsize[$i+j\times(p+1)+1$]$-1$], $0 \le j \le p-1$. The merged results are
        stored in array starting from index bucksize[$i$]
    endfor
end
```

A1.4 Parallel Sorting by Regular Sampling

```
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 $p$ elements evenly spaced from each of the $p$ sorted lists
    //as the regular sample and store them in subsize[$0:p^2-1$]
    $gn = \lfloor (n+p-1)/p \rfloor$
    $ggn = \lfloor (gn+p-1)/p \rfloor$
    for $i = 0$ to $p-1$ do in parallel
        $start = i \times gn$
        $end = (i+1) \times gn - 1$
        if $end \ge n$ then $end = n-1$
        //Sort subarray array[start:end] with sequential Quicksort
        quicksort(array, start, end)
        for $j = 0$ to $p-1$ do
            if $j \times ggn \le end$
                then subsize[$i \times p + j$] = array[$j$/(muggn]
                else subsize[$i \times p + j$] = array[end]
            endif
        endfor
    endfor
    //Sort the regular sample subsize[$0 : p^2-1$]
    //Choose $p-1$ pivots from the sorted regular sample with sequential Quicksort
    //Store them in pivots[$1:p-1$]
    quicksort(subsize, 0, $p^2-1$)
    for $i=1$ to $p-1$ do
        pivots[$i$] = subsize[$i \times p + \lfloor p/2 \rfloor$]
    endfor
    //Divide in parallel each sorted list $i$ into $p$ sublists:
    //array[subsize[$i\times(p+1)+j$] : subsize[$i\times(p+1)+j+1$]$-1$],
    //$0 \le j \le p-1$ with the chosen pivots as splitters
    for $i = 0$ to $p-1$ do in parallel
        $start = i \times gn$
```

```
            end = $(i+1) \times gn - 1$
            if end $\geq n$ then end = $n-1$
            subsize $[i \times (p+1)]$ = start
            subsize $[i \times (p+1)+p]$ = end $+1$
            sublists(array, start, end, subsize, $i \times (p+1)$, pivots, 1, $p-1$)
        endfor
        //Count the size of each of the $p$ partitions
        for $i = 0$ to $p-1$ do
            bucksize $[i] = 0$
            for $j = i$ to $p \times (p+1)-1$ step $p+1$ do
                bucksize $[i]$ = bucksize $[i]$+subsize $[j+1]$−subsize $[j]$
            endfor
        endfor
        //Decide the start-point of each partition in the final array
        for $i = 1$ to $p-1$ do
            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 two-way mergesort:
            array [subsize $[i+j \times (p+1)]$ : subsize $[i+j \times (p+1)+1]-1$], $0 \leq j \leq p-1$. The merged results are
            stored in array starting from index bucksize $[i]$
        endfor
end
```

```
procedure sublists(array, start, end, subsize, at, pivots, fp, lp)
//This procedure is called by qm, pss, and psrs
//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 = $\lfloor (fp+lp)/2 \rfloor$
    pv = pivot [mid]
    lb = start
    ub = end
    while lb $\leq$ ub do
        center = $\lfloor (lb+ub)/2 \rfloor$
        if array [center] > pv
            then ub = center $-1$
            else lb = center $+1$
        endif
    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
```