Order Number 9029990

Design of parallel mergesort and quicksort algorithms

Xiong, Renbing, Ph.D.

City University of New York, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

DESIGN OF PARALLEL MERGESORT AND QUICK SORT ALGORITHMS

BY

RENBING XIONG

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1990

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

April 30, 1990
Date

Theodore Brown
Chair of Examining Committee

April 30, 1990
Date

T C Wesselkamper
Executive Officer

Prof. Michael Anshel

Prof. Theodore Brown

Prof. Stefan Burr

Prof. Carol Tretkoff

Prof. Thomas Wesselkamper

Supervisory Committee

The City University of New York

Acknowledgements

I would like to thank the members of my examining committee, many of whom are my teachers. From Dr. Tretkoff's class I have gotten many ideas about how to design parallel algorithms. From Dr. Anshel's class I got a broad mathematical and logical background. Dr. Anshel and Dr. Burr gave me much good advice. Dr. Wesselkamper provided financial aid through the department, so that I could successfully finish my study. I wish to also thank him for his careful proof reading. He gave me many good suggestions.

Added to these I would also like to thank Dr. Beckman. He provided fellowships and (or) tuition waivers for me through the department every year since I entered this program, and arranged an teaching assistant-ship for me, so that I could concentrate on my study.

I also like to thank Queens College, Computer Science Department, who invited me to the U. S., and brought me back to school after nineteen years of teaching in China.

Dr. Brown served as a my supervisor, helped me a lot, from choosing topics, to English writing. Here is a story. Last Spring, I read Akl's EREW merging algorithm and found that he had already published the work I had gotten some results on. So I wanted to give up. Dr. Brown read both papers carefully, compared my results with Akl's, pointed out the different between these two, and encouraged me to continue. He spent a lot time to help me to overcome difficulties I met when I wrote the paper. I couldn't have done it without his help.

CONTENTS

Design of Parallel Mergesort and Quicksort Algorithms

Renbing Xiong

This thesis is primarily concerned with presenting improved parallel processing versions of the quicksort and mergesort algorithms. In doing so we also touch on the broader issues of devising divide-and-conquer algorithms for parallel machines. These two sorting methods have been chosen not only because of their inherent interest and numerous studies over the years, but also as a paradigm for a larger class of algorithms that is especially suited for parallel processing - divide-and-conquer.

Although there has been interest in parallel computing for some time, recently there has been considerable research activity in the design, analysis, implementation, and the use of parallel algorithms. Whereas a few years ago there were almost no books, new books seems to appear almost monthly. There are books on the development and implementation of parallel programming languages: eg. Brawer[1989] and Babb II [1988]; others concentrate on parallel algorithms: Akl[1989], Quinn[1987], Modi[1988]. There are at least two journals devoted to parallel processing algorithms, and several conferences a year devoted only to parallel processing.

A few years ago parallel computers could be found only in research laboratories. Now they are available commercially. To name a few: the Connection

Machine, Alliant FX/8, BBN Butterfly Parallel Processor, CRAY X-MP, FPS T Serial Parallel Processor, IBM 3090 series, Intel iPSC Concurrent Computer, Loral Dataflo LDF 100, and the Sequent Balance Series. Architecturally these machines vary greatly although they can generally be categorized as MIMD (Multiple Instruction stream - Multiple Data stream), or SIMD (Single Instruction stream - Multiple Data stream). Their architectural memory access connections also vary considerably as well, from a globally shared memory to no shared memory. Below we give more detail on the architectural choices.

It has been found that it is not a straight forward exercise to take a (sequential) program and run it on a parallel machine. Many algorithms that run well on sequential computers are not easily transformed to algorithms that efficiently run on parallel computers. We are interested in examining ways of improving the implementations of programs that result from using the paradigm of divide-and-conquer on MIMD tightly coupled machines.

The scope of this thesis is primarily sorting algorithms, specifically: quicksort and mergesort. Sorting is one of the most common activities performed by computers. It is often said that $25 - 50\%$ of all the work performed by computers consists of sorting data. Many programs such as compilers and editors 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. Some results on parallel sorting are related to a sorting network, such as odd even merge sort and bitonic sort [Batcher 1968], others are for theoretical models of parallel processors with shared

memory [Hirschberg 1978; Preparata 1978]. We present improved version of two sorting algorithms: quicksort and mergesort on an MIMD with shared memory. In doing so we make use of probabilistic analyses, simulations, and complexity arguments.

Chapter 1 introduces the methods used in this paper and some terminology, including a literature review concentrating on the recent results of parallel merging (or mergesort). Chapter 2 presents the design and analysis of a parallel merging algorithm - more median splitting, k-splitting, merging, and sorting algorithms. Chapter 3 starts from a literature review on the recent results of parallel quicksort, followed by the design and analysis of two quicksort algorithms: a parallel quicksort and a probabilistic parallel quicksort. In chapter 4, an annotated bibliography appears.

Chapter 1 Introduction

1.1 Divide-and-Conquer Paradigm

Divide-and-conquer has been a popular paradigm for algorithms for many years. Many algorithms fall within its classification. Examples are: finding the maximum (minimum) of a set of numbers, finding the sum of a set of values, etc. Almost any book on the design and analysis of algorithms includes a section on divide and conquer. Horowitz and Sahni [1978] for example has a well done chapter. What follows is outline of the paradigm. We are interested in improving the efficiency of the algorithms that are of a divide-and-conquer type.

Divide and conquer works by dividing the set of input data items into two or more disjoint subsets with different processes working on each subset. The partial results from these processes are then combined to get a final answer. The paradigm can be thought of as consisting of 1) a splitting step, 2) the divide-and-conquer calls, then 3) the merging or combining of the subproblem results. The algorithm that results is often written recursively. As an example, consider the finding of a maximum of a set of numbers. One way of finding the maximum of a set is by dividing the set into two disjoint subsets S1 and S2 and then finding the maximum of each. If there are more than 2 elements in the resulting subset the algorithm could divide the subset again in two (via a recursive call). The third part of the algorithm is the determination of the global maximum of the subset maximums.

The dynamic nature of divide-and-conquer algorithms makes them especially

interesting for parallel processing. With dynamic (parallel) processes the interactions between subprocesses cannot be predicted, as the processes can spawn new processes, or be killed by other processes. Static parallel processes are interesting in other ways in that a programmer knows in advance how many subprocesses are created and how they interact. The interest then is one of scheduling. There are many papers devoted to implement divide-and-conquer paradigm on a parallel computer. Horowitz and Zorat [1983] design a general algorithm to implement divide-and-conquer algorithms on a parallel computer. Mou and Hudak [1988] give an algebraic model for divide-and-conquer.

1.2 MIMD and SIMD Machines

In an MIMD machine each processor can follow an independent instruction stream. An MIMD machine can consist of p asynchronous processors and p local memories, called a distributed system, or p processors and a global shared memory, called a tightly coupled machine, or other intermediate organizations. We deal here with the tightly coupled machine. In this organization any processor may access any location of the global memory at every computation step with the same access delay.

In a SIMD (single instruction stream and multiple data stream) machine there are N identical processors, each of them possessing its own local memory. Both instructions and data can be stored into the local memory. All processors operate under the control of a single instruction stream issued by a central control unit

(CCU). All the processing elements perform the same function synchronously in a lock-step fashion under the command of the CCU. The processors may communicate with each other via shared memory (SM), called SM- SIMD machine, or a connected network. We deal here with the SM-SIMD machine, which is also known as the Parallel Random-Access Machine (PRAM) model.

Data exchanges among the processors are done via the shared memory. The basic model allows all processors to gain access to the shared memory simultaneously if the memory locations they are trying to read from or write into are different. Algorithms that run on shared-memory (SM) computers can be divided into four subclasses, according to whether more than one processor is required to read from or write to the same memory location simultaneously. The three important ones are:

(1) Exclusive-Read, Exclusive-Write (EREW). Access to memory locations is exclusive, i.e. no two processors are requested to simultaneously to read from or write into the same memory location.

(2) Concurrent-Read, Exclusive-Write (CREW). The algorithm allows requests from multiple processors to read from the same memory location but no two processors are allowed to write simultaneously into the same memory location.

(3) Concurrent-Read, Concurrent-Write (CRCW) SM Computers. Both multiple read and multiple write are allowed.

This last category requires further specifications as to what happens if more than one processor tries to write to one location and their values are different.

Various possibilities have been considered, adding additional complications. One such is a ***random-write model.*** In this model if more than one processor tries to write to the same memory location simultaneously, exactly one processor wins in writing its value; the writing processor is determined randomly, and each processor wishing to write to this location has equal opportunity to succeed.

An instruction that is useful when concurrent reads and concurrent writes are allowed is a Fetch-and-Add ($F\&A$) instruction. A fetch and add ($F\&A$) instruction [Lipovski 1987] permits highly concurrent execution of operating system and applications programs. This instruction has two operands: $F\&A(V, e)$. V is an integer variable and e is an integer expression. This operation returns the (old) value of V and replaces V by the sum of $V + e$. If many fetch and add operations simultaneously address V (i.e. in the same memory cycle) the effect of these operation is exactly what it would be if they occurred in some serial order, i.e. V is modified by the appropriate total increment and each operation yields the intermediate value of V corresponding to its position in this order. Simultaneous memory updates are in fact accomplished in one cycle.

1.3 Run Time Analysis

A measure of performance for an algorithm running on a parallel computer is its speedup, S_p, defined as the ratio of the total execution time, if it were to run on a sequential computer $T(1)$, to the corresponding execution time, $T(p)$, on the

parallel computer for the same problem, i.e.

$$Sp = T(1)/T(p).$$

A second measure is efficiency, E_p, defined as the ratio of the total execution time, if it were to run on a sequential computer $T(1)$, to the product of the number of processors, p, and the corresponding execution time, $T(p)$, on the parallel computer for the same problem, i.e.

$$Ep = T(1)/(p * T(p)) = Sp/p.$$

Of course $T(1) \leq p * T(p)$. That is, $E_p \leq 1$. If $E_p = 1$, we say that algorithm is optimal in the sense that its total cost, (the number of processors it uses multiplied by its parallel running time) equals the lower bound on a sequential machine.

1.4 Divide-and-Conquer Paradigm on an MIMD Machine

Moller-Neilsen and Staustrup [1987] suppose that a control task list, which we call a problem heap, is used to schedule tasks for processors of the MIMD system [Moller 1987]. The problem heap supplies all processes with tasks. In the divide-and-conquer paradigm the task (subtask of the original problem) might be solved immediately if the subtask is simple enough, or it might generate two or more new subtasks which are put then back onto the problem heap. This paradigm splits a task into a number of identical, asynchronous subtasks. The standard way the divide-and-conquer paradigm is run is to start with one task. Consequently all

but one of the p processors are idle to start. Only as more subtasks are generated can more processors start execution. Once the number of tasks is greater than or equal to the number of processors, every processor has gotten a problem to work on, and no processor will be idle. But the initial idleness of processors causes a loss in efficiency and is called a starvation loss.

This starvation is a direct cause of a loss in efficiency of an algorithm as processing power of the MIMD system is lost. In some algorithms the loss caused by the initial tasks is negligible, in others it is not. Quicksort's starvation is considered by some as the bottleneck of the algorithm [Moller-Nielsen and Staustrup, 1987]. Mergesort for a MIMD machine as described by Yousif and Evans[1987] has starvation phase at the end. Their algorithm is the standard one applied to MIMD machines. In this algorithm the number of problems is reduced in each following step until there is only one task at the end. Of course this is true generally in any merging type of algorithm.

Although our broad interest is in how the divide-and-conquer paradigm can be structured in an MIMD machine environment to improve efficiency, we next look at two common sorting algorithms as examples, as they exhibit behavior which illustrates our discussion.

1.5 Sorting Algorithms

Sorting in computer terminology is defined as the process of rearranging a sequence of values in ascending or descending order. This thesis improves two

of the most important sorting algorithms designed for an MIMD machine, all of which are internal sorts (sorting tables small enough to fit entirely in primary memory). Both of the sorting algorithms we consider sort by comparing pairs of keys and are based on the divide-and-conquer paradigm.

The following sections of this thesis reviews the literature, and then using quicksort and mergesort examines the effects of some improvements in efficiency based on reducing starvation.

1.6 Parallel Mergesort

The merging algorithms usually include two phases: the first phase to separate the ordered lists into p disjoint parts, and the second consists of p parallel sequential merges. The merging algorithms of Akl[1987] and that of Valiant[1975] and Kruskal [1983] (Kruskal improves Valiant's algorithm) differ only in their first phase. Since the time complexity of the second phase is $O((m+n)/p)$, and this is optimal as long as the first phase has a smaller complexity, both algorithms are optimal. Consequently it is in their differing first phase that improvements that can be made.

In the first phase, Akl's algorithm makes use of Rodeh's idea of distributed median. The complexity of this part is $O(\lg p \lg(m+n))$ and so his algorithm will be optimal as long as for $\lg p \lg(m+n) \leq (m+n)/p$, i.e. $p \leq (m+n)/(\lg(m+n)\lg p)$, adding more restrict, $p \leq (m+n)/\lg^2(m+n)$. Below we reduce the complexity of the first to $O(\lg(m+n))$ by generalizing the idea of distributed median to that

of distributed k^{th} smallest (largest). These ideas are presented as Lemma 1 and 2 below.

1.6.1 Yousif and Evans's Mergesort

A shared memory MIMD version of a mergesort algorithm has been analyzed by Yousif and Evans [1987]. The algorithm evenly divides the N elements to be sorted into M subsets, and sorts each subset by an efficient sequential algorithm. Then each pair of sorted lists is merged using any free processor and a binary search merge algorithm to form a sorted list of double length. When there are fewer than p (number of processors) tasks left some processors are idle. At the final step all processors are idle but one.

1.6.2 Quinn's Quick/Merge Algorithm

Quinn [1988] designed a p-way quick/merge algorithm. Each of p processor sorts a contiguous set of about size N/p using sequential quicksort. Then $p-1$ evenly-spaced keys from the first sorted list are used as divisors to partition each of the remaining sorted sets into p sublists. Each processor i, $1 \leq i \leq p$, performs a p-way merge of the i^{th} sorted sublists. This version removes the starvation problems of the Yousif and Evans's algorithm. The size of partitions to be assigned to each processor might be very different which can still create substantial starvation at the end.

1.6.3 Shiloach's Parallel Merging

Assume there are two sorted lists, in ascending order, $X = (x_1, ..., x_m), Y = (y_1, ..., y_n), m \leq n$, with the merged list to be stored in $Z = (z_1, ..., z_{m+n})$. Shiloach [1981] designed a merging algorithm in two different cases. Case 1: assume $m = n = p$ (the number of processors). Case 2: $p \leq m \leq n$.

In the case 1, $m = n = p$, processor i by a binary search ($O(\lg n)$ operations) finds the smallest y_j such that $x_i < y_j$ and then performs $z_{i+j-1} \leftarrow x_i$. If there is no such y_j it performs $z_{n+i} \leftarrow x_i$. Processor i then finds the smallest x_j such that $y_i < x_j$ and then sets $z_{i+j-1} \leftarrow y_i$. If there is no such x_j it sets $z_{m+i} \leftarrow y_i$. The time complexity is $2 \lg n$.

In the case 2 $p \leq m \leq n$, the algorithm is as follows:

Stage 1.

1. Choose $p - 1$ evenly-spaced keys x' from X and $p - 1$ evenly spaced keys y' from Y.

2. Merge x' and y' into a vector A of length $2p - 2$. Associate with each element in A its original set (X or Y) membership and its index in it.

Stage 2.

3. Processor i, $2 \leq i \leq p$, checks the origin of the $(2i - 2)^{th}$ elements in A. If it belongs to $X(Y)$ it finds (by a binary search) the smallest element in $Y(X)$ that is greater than it. These two elements provide processor i with a starting point for its merging. (The starting point for processor 1 is x_1 and y_1).

4. Processor i, $2 \leq i \leq p - 1$, merges (by inserting into Z) all the elements

that fall between the $(2i-2)^{th}$ and $2i^{th}$ elements of A. Processor 1 does the same for the elements that are smaller than the second element of A. Processor p merges the elements that are greater than the greatest element of A.

Note that the size of each interval to be merged by each processor does not exceed $2(m+n)/p$. Therefore the time complexity is $2\lg p+\lg m+\lg n+2(m+n)/p$.

1.6.4 Akl's EREW Merging and Sorting

Akl [1987] has given an EREW merging algorithm to merge two sorted lists A and B of size m and n ($m \geq n$). Set N equal to $m+n$. The building block of this algorithm is Rodeh's algorithm [1982] to find the distributive median.

Rodeh [1982] gives an algorithm for finding the distributive median using two communicating processes, pA and pB, having local access to equal sized sets, A and B, respectively. Rodeh's algorithm sorts A and B in separate local memories first. It then determines the distributive median by reducing the set of possibilities through a sequence of steps. At the end of each step, some elements of A are removed to either $A1$ or $A2$ (which are initially considered empty) and some from B are removed to $B1$ or $B2$, thereby reducing the cardinality of A and B, and increasing either that of $A1$ and $B2$ or $A2$ and $B1$. When all the elements of A and B are exhausted, i.e. $A1$ ($B2$) of size x and $A2$ ($B1$) of size $m-x$, the process terminates. The median should be the maximum of $A1$ and $B1$ or the minimum of $A2$ and $B2$.

The definition of distributive median of two sorted lists with unequal sizes is

similar to the equal size one except B1 including $N/2 - x$ smallest of B and $B2$ the rest of B. Let la and ua be the lower and upper bound of indexes of A and lb and ub the same of B.

The algorithm is as follows:

```
procedure Select
{Sorted lists A and B of size m and n}
{ are initially in array A(1,...,n) and B(1,...,m).}
begin
   la := 1
   ua := m
   lb := 1
   ub := n
   ra := m { the number of elements at that time in A}
   rb := n { the same as above in B}
   u := ⌈m/2⌉
   v := ⌈n/2⌉
   while ra > 1 and rb > 1 do
      if a_u ≥ b_v then
         ra := u − la + 1
         rb := ub − v
         ua := u
         lb := v + 1
      else
         ra := ua − u
         rb := v − lb + 1
         la := u + 1
         ub := v
      end if
      u := la + ⌈(ua − la − 1)/2⌉
      v := lb + ⌈(ub − lb − 1)/2⌉
   end while
   X :=< a_{u−1}, a_u, a_{u+1} >
   Y :=< b_{v−1}, b_v, b_{v+1} >
end. { Select }
```

The algorithm returns the pair $(a_x, b_y) \in X \times Y$. It requires $c_1 + c_2 \lg(min(n,m))$ time, which is $O(\lg(m+n))$. Akl's[1] EREW splitting at the beginning assigns single processor to run procedure Select, to split $A \perp B$ into two equal sized pairs,

$A1 \perp B1$ and $A2 \perp B2$, in $O(lg(m+n))$ time. Then assign two processors to run the same procedure: one is assigned to work on $A1 \perp B1$, and the other on $A2 \perp B2$. When the pair of sublists of A and B is split into two equal sized pairs, one processor is assigned to each pair. This process does not terminate until p equal sized pairs are generated, which is $\lg(p)$ iterations. The complexity of his splitting is $O(\lg(p)\lg(m+n))$ and so is optimal if $\lg(p)\lg(m+n) \leq (m+n)/p$, or, by solving for p with $p < m+n$, $p \leq (m+n)/\lg^2(m+n)$.

The sorting algorithm includes two main steps. In the first step, the input set is subdivided into p subsets of size N/p each. Each subset is then sorted sequentially by one of the processors in $O((N/p)\lg(N/p))$ time if an optimal sequential algorithm is used.

In the second step, pairs of sublists are merged simultaneously using the merging algorithm above using all p processors; pairs of resulting sublists are merged in turn, and the process is continued until one sorted list of size N is produced. There are $\lg p$ merging stages; at stage $i, 1 \leq i \leq \lg p, 2^i$ processors are used to merge two sublists of size $2^{i-1}N/p$ each. Hence, this step requires $O((N/p)\lg p + \lg N \lg^2 p)$ time. The total running time of this algorithm is, therefore, $O(N \lg N/p + \lg N \lg^2 p)$. It is optimal for $p \leq N/\lg^2 N$.

1.6.5 Valiant/Kruskal's Merging

Valiant[9] is able to claim a speed of $2\lg\lg n + c$ comparisons to create $A \perp B$ with $p = \lfloor\sqrt{mn}\rfloor$ processors and $1 < n \leq m$. (This algorithm determines indices;

it does not move the values.) A sample from A, say $< a_i^*, i \geq 1 >, a_1^* = a_{\lceil\sqrt{m}\rceil}$ and the rest $\lceil\sqrt{m}\rceil$ apart, and a like sample from B, say $< b_i^* >$ that are $\lceil\sqrt{n}\rceil$ apart are chosen. The p processors are enough to compare every element in one sample sequence with every element from the other sample sequence in one step. These comparisons determine the position of each b_i^* into which it belongs in the A sequence up to the interval $[a_{j-1}^*, a_j^*]$. Since there are $\lceil\sqrt{m}\rceil - 1$ elements between a_{j-1}^* and a_j^* in A, the exact merge location of each b_i^* in A can be determined in parallel in one additional step using $\lfloor\sqrt{n}\rfloor(\lceil\sqrt{m}\rceil - 1)$ comparisons. The location of these b_i^* segment the B sequence into $\lceil\sqrt{n}\rceil$ segments and the location of b_i^* in A segments the A sequence into $\lceil\sqrt{n}\rceil$ segments that can be paired with the B segments. Again there are enough processors to assign to each pair of segments (i.e. the square root of the product of the paired size of the segments) to repeat these steps inductively for each pair in parallel.

When there are more elements to merge than there are processors available (i.e. $1 < p < n \leq m$) Valiant [9] modifies this algorithm. A sample of $p-1$ equally spaced points from each list is chosen. The sampled values' locations up to their interval in the other list can be determined in $O(\lg\lg(p))$ using the algorithm stated above. The exact location in the original list of each sample value can be then found in $O(\lg(n/p) + \lg(m/p))$ by using, for example, binary search (Shiloach[7]). The resulting locations determine $2p-1$ pairs of disjoint sublists to be merged, in which no pair contains more than $(m+n)/p$ elements. As there are twice as many pairs as processors, Valiant [9] argues a (complicated) scheduling of processors can

be used so that the merging time can be $(m+n)/p$ (rather than twice as much).

The resulting total time of this algorithm is $O((m+n)/p+\lg(mn\lg p/p^2)+c)$, based on comparison steps. As two lists can be merged in $O(m+n)$ time with one processor, an algorithm that uses p processors are optimal only if it merges the lists in $O((m+n)/p)$ time. Hence this algorithm is optimal only when the first term dominates the others.

Kruskal [5] has improved Valiant's algorithm. By using a different sampling plan, he is able to reduce the problem of merging two lists of length m and n to the problem of merging a number of pairs of lists where each pair's shorter list has length less than $n^{1/k}$. A sample from A, say $a_i^*, i \geq 1$, $m^{1-1/k}$ apart, and a sample from B, say $b_i^*, i \geq 1$, $n^{1/k}$ apart are chosen. The p processors are enough to compare every element in sample B^* with every element from A^* in one step. These comparisons determine the position of each b_i^* into which it belong in A sequence up to the interval a_{j-1}^*, a_j^*. Since there are $m^{1-1/k}$ elements between a_{j-1}^* and a_j^* in A. To determine the exact location of b_i^* in interval a_{j-1}^*, a_j^* by $m^{1/k}$ processors, choose evenly spaced $m^{1/k}$ elements A_j^{**} from that interval, then compare them with b_j^* in one step. These comparisons determine the position of each b_i^* into which it belong in A up to the interval $[a_{j-1}^{**}, a_j^{**}]$ of size $m^{1-2/k}$. By repeating this step k times on the newly found interval, the exact merge location of b_i^* in A is found. The location of these b_i^* segment the B sequence into $n^{1-1/k}$ segments and the location of b_i^* in A segments the A sequence into $n^{1-1/k}$ segments that can be paired with the B segments. This procedure continues until p pairs of

segments are generated. The time complexity is $k\lceil\frac{\lg\lg n}{\lg k}+1\rceil, k \geq 2$. When $k = 3$, it reaches the minimum, i.e. $1.893 \lg\lg n + 4$.

Chapter 2 Parallel Median Splitting, k-Splitting, Merging, and Sorting

Let $A \perp B$ represent the result of the merging of two non-decreasing sequences $A =< a_1, a_2, ..., a_m >$ and $B =< b_1, b_2, ..., b_n >$. Median splitting divides the two sequences A and B into two parts so that $A1 \perp A2 = A$, $B1 \perp B2 = B$, $|A1| + |B1| = \lceil (m+n)/2 \rceil$, $|A2| + |B2| = \lfloor (m+n)/2 \rfloor$, and all the elements in $A1$ and $B1$ are less than or equal to those in $A2$ and $B2$. We present MIMD algorithms to do median splitting, k-splitting (a generalization of median splitting in which $|A1| + |B1| = k$), and splitting using a vector of k values. This latter algorithm leads to an optimally efficient merging algorithm that is faster than Valiant's when the ratio of the number of values to be sorted to the number of processors is greater than $(\lg p)^{1/3.4}$. It is also superior in that the partitioned sections it creates for parallel merging are all the same size.

Section 2.1 Introduction

We have two non-decreasing sequences $A =< a_1, ..., a_m >$ and $B =< b_1, ..., b_n >$ (i.e. ordered lists) that are to be merged and p MIMD (Multiple Instruction stream, Multiple Data stream) processors available. Denote the merged sequence as $A \perp B$. We are interested in devising algorithms that split the two lists into pairs of disjoint sublists so that the pairs can be efficiently merged in parallel. We restrict our attention to shared memory MIMD (SM-MIMD) machines [1989].

This paradigm is powerful, but care must be taken for memory conflicts in algorithms written under its assumptions. It creates the possibility of synchrony and concurrency problems that have worried designers of multiprogramming operating systems for some time [1988]. It is common to characterize a parallel algorithm for a shared memory machine as an EREW (exclusive read, exclusive write), as an CREW (concurrent read, exclusive write), or a CRCW (concurrent read, concurrent write) algorithm.

The fastest merging algorithm reported to date for a SM-MIMD for $p = \lfloor\sqrt{mn}\rfloor$ is Valiant's [1975] which is $O(\lg\lg(n))$, and which is asymptotically optimal. We make use of his algorithm for $p < \lfloor\sqrt{mn}\rfloor$ (which is also optimal) and we have outlined it before.

Section 2.2 Median and Median Splitting

We make use of algorithms for finding a median of the $A \perp B$ sequence, as a median can then be used to separate the two sequences into equal parts. Firstly, exactly what is meant by a median needs clarification. The classical definition of the median of a distribution is that value that splits the distribution in half, in the sense that the probability is one half that an outcome is less than this value and half more. To be consonant with this definition, the median of finite sample of values is often also defined as the value that divides the sample into two equal parts. Consequently, if the size of the sample, n, is odd, the median is the $(n+1)/2$ ordered value; if it is even, the median is generally defined as the average of the

$n/2$ and $n/2 + 1$ ordered values.

The algorithm below shall partition both A and B into two parts each say $A1$, $A2$, and $B1$, $B2$, respectively so that $A = A1 \perp A2$, $B = B1 \perp B2$, $|A1| + |B1| = \lceil (m+n)/2 \rceil$ and $|A2| + |B2| = \lfloor (m+n)/2 \rfloor$, and all the elements in $A1$ and $B1$ are less than or equal to those in $A2$ and $B2$.

We shall define the ***distributive median*** of $A \perp B$ as the $\lceil (m+n)/2 \rceil$ ordered value. Consequently the distributive median equals the maximum value of $A1 \perp B1$. According to traditional definition, the median is the maximum of the largest element in $A1 \perp B1$ when n is odd (equal to our definition of distributive median) or the average of this value and the minimum of the smallest values in $A2 \perp B2$ when n is even.

The term "distributive median" is borrowed from Rodeh [1982]. His algorithm determines what he also calls the distributive median. Akl's [1987] makes use of Rodeh's distributive median in his merging algorithm, but their definitions have slightly different meanings. We do not dwell here on the differences between definitions since our interest is in dividing the two sequences into two equal parts when n is even or equal but one if n is odd. A generalized version of Rodeh's procedure follows in the paragraph below. The method is based on reducing the set of possibilities that belong to A and B until both are empty. The elements that are removed are put into $A1$, $A2$, $B1$ or $B2$ as appropriate. Assume for the time being that the sizes of both lists are equal to n. We remove this restriction shortly. The simple, but crucial idea used to separate the lists into two parts is that the n

largest elements of $A \perp B$ belong in $A2$ or $B2$ and the n smallest elements belong in $A1$ or $B1$. (Rodeh's [1982] algorithm is based on $m = n$, no equality between keys, and having the x of the algorithm as defined immediately below equal to $\lceil n/2 \rceil$.)

Suppose we compare a_x with b_y for any fixed $x \in (1, 2, ..., n)$ and $y = n+1-x$. Call y the *complementary* index of x since for any x and y, $x+y = n+1$. If $a_x < b_y$ we can place $< a_1, ..., a_x >$ into $A1$ and $< b_y, ..., b_n >$ into $B2$ because we know that for the former sequence, there are at least n elements of $A \perp B$ greater than any of these elements, namely $< a_{x+1}, ..., a_n >$ plus $< b_y, ..., b_n >$ and for the latter sequence that there are at least n elements of $A \perp B$ less than any of these elements, namely $< a_1, ..., a_x >$ plus $< b_1, ..., b_{y-1} >$.

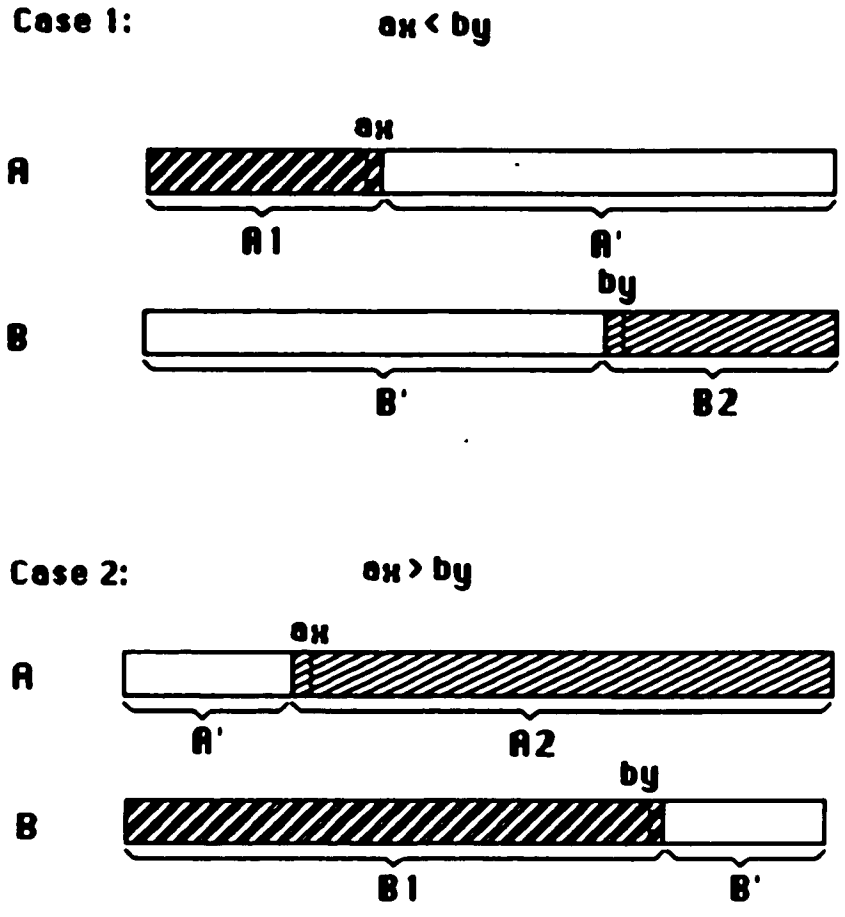
If, on the comparison the inequality is reversed, the elements $< b_1, ..., b_y >$ have at least n elements greater and can be removed to $B1$ and $< a_x, ..., a_n >$ have at least n elements less and can be removed to $A2$, using reasoning similar to that above.

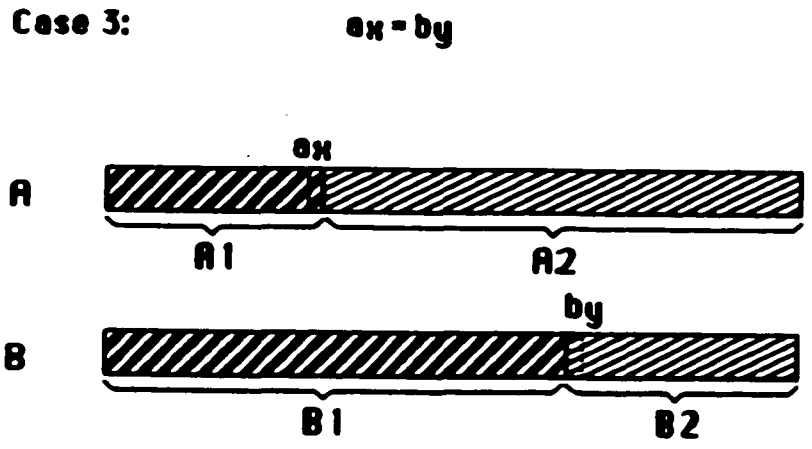
When these values are placed in $A1$, $A2$, $B1$, or $B2$, they are also removed from A or B. As a result when $a_x < b_y$ the sizes of A and B are both reduced by x elements and when $b_y < a_x$ their sizes are reduced by $n + 1 - x$. Importantly, whichever action is the one taken, we have removed the same number of elements from the left side of $A \perp B$ as from the right and we are left with the original problem of dividing $A \perp B$ into two equal parts, but for a smaller sized problem. Notice also that the best a priori choice of x is $\lceil n/2 \rceil$ (as in Rodeh's algorithm) as

then independent of the outcome of the comparison, the size of $A \perp B$, is reduced by at least n. With this choice of x the number of steps to reduce A and B to null sets is equal to at worst $\lfloor \lg(n) \rfloor + 1$.

If on any iteration $a_x = b_y$ then the algorithm can terminate immediately. This is because, for the reasoning identical to that above, we can move $< a_1, ..., a_x >$ into $A1$, $< a_{x+1}, ..., a_n >$ into $A2$, $< b_1, ..., b_{y-1} >$ into $B1$, and $< b_y, ..., b_n >$ into $B2$ immediately. (Note that a_x and b_y could as well be put into $A2$ and $B1$, so multiple solutions exist.) Figure 1 shows these three cases schematically. The part of A (B) that are to be kept for the next iteration is denoted by A' (B').

Figure 1



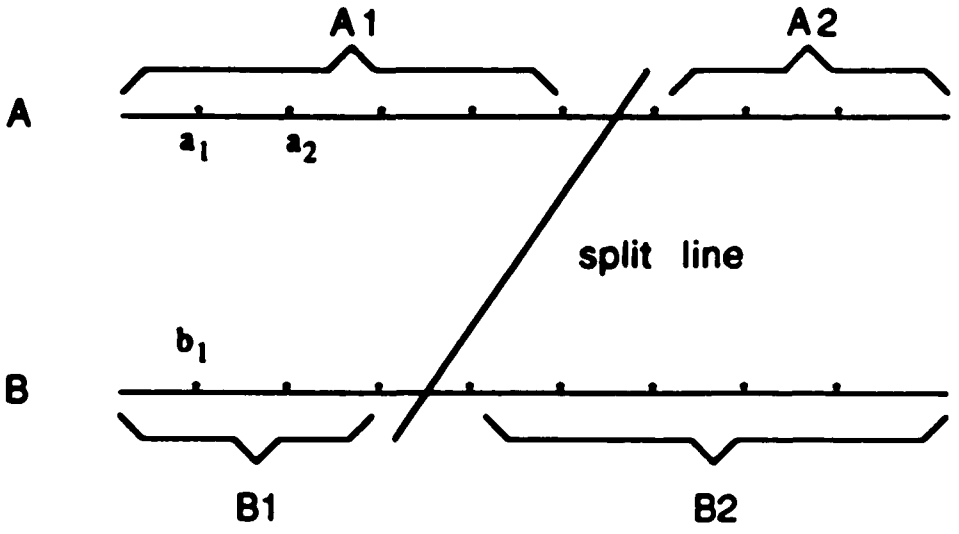


The procedure Mediansplit defined below incorporated these ideas. It calls upon the procedure Reduce to reduce the size of A and B. Rather than move the elements between sets (eg, from A into $A1$ or $A2$), the indices la, lb, ua, ub are used to represent the movement of the data between sets. Values to the left of la (lb) are to be considered in the $A1$ ($B1$) set, and to the right of ua (ub) are in the $A2$ ($B2$) set. Values from la through ua inclusive are in A and likewise for B.

Mediansplit expects that it is passed la and lb having values equal to the low index of A and B, respectively, and ua and ub the high indices, respectively. The Reducetoequal procedure is needed when $m \neq n$ and is explained below. When A (and B) are null the **while** loop is exited. The procedure Reduce implements the three cases stated above by modifying the appropriate indices: la, lb, ua or ub. When Mediansplit is exited: ua equals $la + 1$ and ub equals $lb + 1$; ua points to the lowest index of $A2$, la points to the highest index of $A1$, and likewise for lb and ub. One of the two indices, la and lb is the index of median of $A \perp B$.

If we envision the sequence A directly above sequence B, and envision a line intersecting A between la and ua and intersecting B between lb and ub, then the values to the left of this line belong to $A1$ or $B1$ and those to the right to $A2$ or $B2$. See figure 2. Call this line a *splitline*. Since la (lb) and ua (ub) have redundant in' nation we define $pa = la + \epsilon$, and $pb = lb + \epsilon, 0 < \epsilon < 1$. Then we can say that the intersection of the cutline with the A sequence occurs at pa and its intersection with B at pb. Call the pair (pa, pb) a *split pair*. Clearly $la = \lfloor pa \rfloor$, $ua = \lceil pa \rceil$ and similarly for pb.

Figure 2




```
procedure Mediansplit (la, ua, lb, ub)
begin
   Reducetoequal(la, ua, lb, ub) { see below }
   while  la ≤ ua do
      Reduce(la, ua, lb, ub)
   end {while}
   la ↔ ua
```

```
    lb ↔ ub
end. { Mediansplit }

procedure Reduce (la, ua, lb, ub)
begin
    d := ⌊(ua − la)/2⌋
    x := la + d
    y := ub − d
    if a_x < b_y then
        la := x + 1
        ub := y − 1
    else if a_x > b_y then
        ua := x − 1
        lb := y + 1
    else {a_x = b_y}
        la := x + 1
        ua := x
        lb := y
        ub := y − 1
    end if
end. {Reduce}
```

The Mediansplit procedure either removes one more than half the remaining elements from consideration from each list on each iteration, or (on equality) it terminates immediately. Hence it requires $\lfloor \lg(n) \rfloor + 1$ comparisons in the worst case.

We are now in a position to assume that the sizes of A and B are not necessarily the same. With $|A| = m, |B| = n$, without loss of generality assume that $n \leq m$.

There are $m - \lfloor (m+n)/2 \rfloor$ elements of A that can be removed immediately to $A1$ and $m - \lceil (m+n)/2 \rceil$ elements to $A2$, reducing the problem to one with both sequences equal to n. This is true because the elements $A_l =< a_1, ..., a_{m-\lfloor (m+n)/2 \rfloor} >$ of A must belong to $A1$ since there are $\lfloor (m+n)/2 \rfloor$ elements

that are greater than or equal to any of those in A_l namely, $a_{m-\lfloor(m+n)/2\rfloor+1}, ..., a_m$. Likewise, the elements $A_r =< a_{\lceil(m+n)/2\rceil+1}, ..., a_m >$ of A must belong to $A2$ since the $\lceil(m+n)/2\rceil$ elements $a_1, ..., a_{\lceil(m+n)/2\rceil}$ are less than or equal to those in A_r. Removing these elements from A leaves it with n elements, and since the same number of elements were removed from both sides of the cutpoint if $m+n$ is even, and the same number but one were removed from both sides if $m+n$ is odd, we are left with the basic problem. Procedure Reducetoequal below, already invoked immediately above the while loop of the Mediansplit procedure, incorporates these ideas.

```
procedure Reducetoequal (la, ua, lb, ub)
begin
   if m > n then
      ua := la + ⌈(m + n)/2⌉ − 1
      la := la + m − ⌊(m + n)/2⌋
   else if m < n then
      ub := lb + ⌈(m + n)/2⌉ − 1
      lb := lb + n − ⌊(m + n)/2⌋
   end if
end. { Reducetoequal }
```

Therefore without loss of generality we can suppose that $|A| = |B| = n$. There is another way of explaining the way median splitting works that is enlightening. See figure 1. The element and those to the left of (those smaller than) the smaller of a_x and b_y, are added to $A1$ or $B1$ as appropriate, and the element and those to the right of (those larger than) the larger of the two are added to $A2$ or $B2$, as appropriate. The values in between are kept in A and B for a next comparison and are denoted by A' and B' respectively.

Suppose we know the results of more than one complementary comparison.

For example suppose we do three comparisons as shown in figure 3a. In this figure, the original sequence A is represented by the top horizontal line and the lower horizontal line represents the B sequence. The three complementary comparisons are represented by the lines that intersect both sequences. The sense of the resulting inequality of these comparisons can be represented by the inequality symbols that are at the ends of these lines. For instance if suppose $a_{x_3} > b_{y_3}$ then a ">" symbol could be added to the figure at the intersection of this line with A and a "<" symbol at its intersection with B. See figure 3b. As a result of this comparison, the part of A to the right of the > symbol (and a_{x_3}) be removed to $A2$ and the part of B to the left of the , (and b_{y_3}) be removed to $B1$. Now imagine the results of all three comparisons are known and are as shown in figure 3c. Independent of the order that these comparisons have been made the resulting set compositions are as shown. The part of the original A (B) sequence that becomes A' (B') is that part between the innermost < and > symbols.

Figure 3a

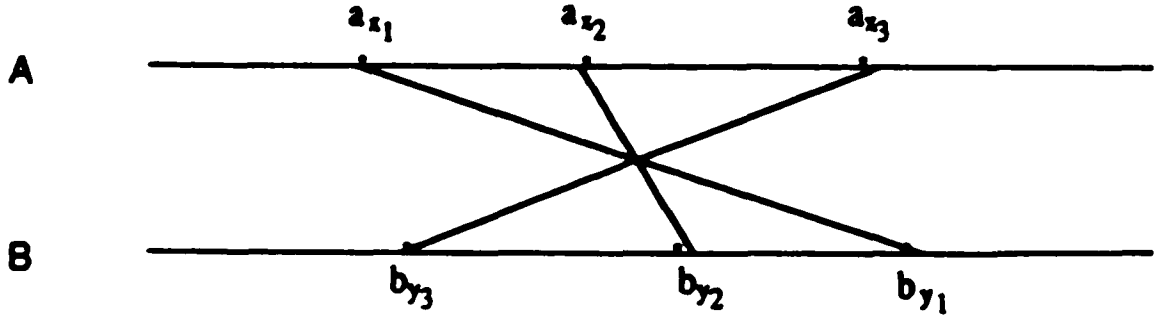


Figure 3b

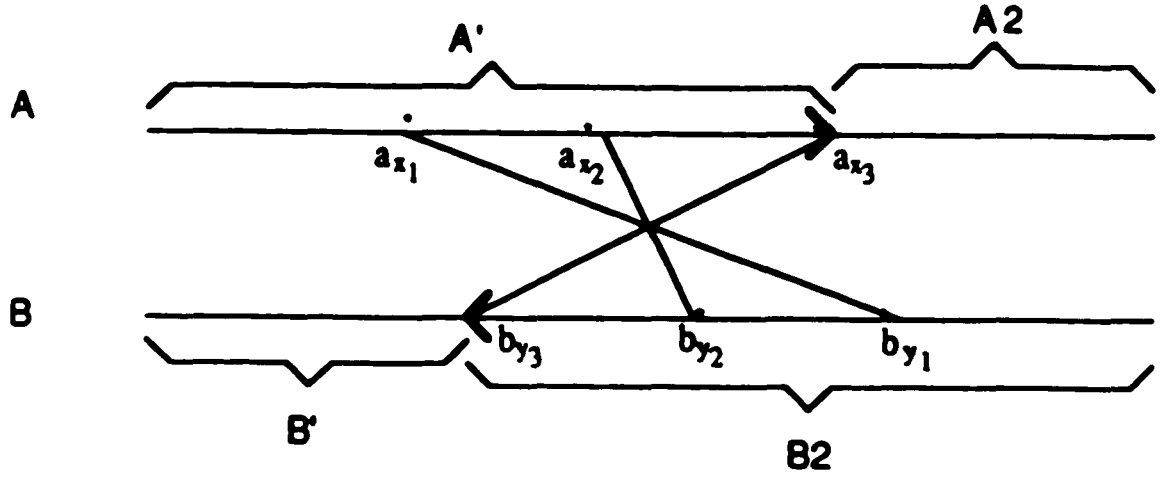
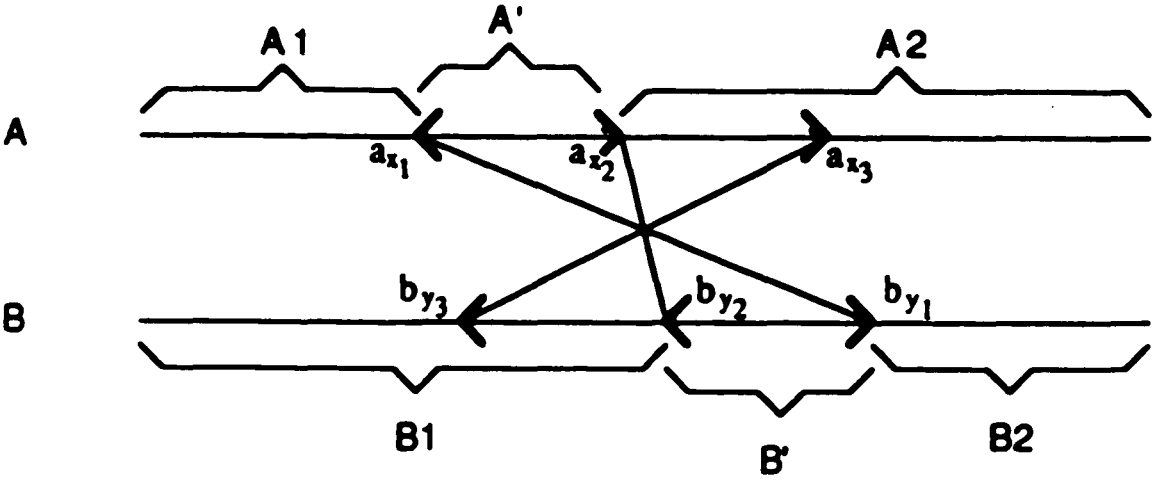


Figure 3c



The advantage of this view is that we can replace the sequential viewpoint of the Mediansplit procedure to a view of the process as a parallel one. In this view we do multiple comparisons in parallel. The part of A and B that must be kept for an additional iteration is that part that is between the comparison reversals.

Suppose t complementary comparisons are made. Let the comparisons be made at points $x_1, x_2, ..., x_t$ ($x_i < x_j$, if $i < j$) in combination with their comple-

mentary points denoted by $y_1, y_2, ..., y_t$ where $x_i + y_i = n + 1$:

$$(1) \quad a_{x_1} : b_{y_1}, a_{x_2} : b_{y_2}, ..., a_{x_j} : b_{y_j}, ..., a_{x_t} : b_{y_t}.$$

Lemma 1. If we make the t comparisons of (1), then there is at most one index at which the inequality reverses.

Proof. The lemma is trivially true since both sequences $< a_1, ..., a_n >$ and $< b_1, ..., b_n >$ are by definition monotonically non-decreasing. Thus the subsequence $< a_{x_1}, ..., a_{x_t} >$ is monotonically non-decreasing and the subsequence $< b_{y_1}, ..., b_{y_t} >$ is monotonically non-increasing. Once they cross they cannot cross again. ∎

The following theorem is an immediate consequence of this lemma.

Theorem 1. Denote by $A'(B')$ the elements left in A (B) after the t comparisons of (1). The following divisions of A and B result:

if $a_{x_j} = b_{y_j}$, for at least one j, then
a):

$A1 =< a_1, ..., a_{x_j} >$,
$A2 =< a_{x_j+1}, ..., a_n >$,
$B1 =< b_1, ..., b_{y_j-1} >$,
$B2 =< b_{y_j}, ..., b_n >$,
$A' = \phi$,
$B' = \phi$; or,

if $a_{x_1} < b_{y_1}$ and $a_{x_t} < b_{y_t}$ then
b):

$A1 =< a_1, ..., a_{x_t} >$,
$A2 = \phi$,
$B1 = \phi$,
$B2 =< b_{y_t}, ..., b_n >$,
$A' =< a_{x_t+1}, ..., a_n >$,
$B' =< b_1, ..., b_{y_t-1} >$; or,

if $a_{x_1} < b_{y_1}$ and there exists a j such that $a_{x_{j-1}} < b_{y_{j-1}}$ and $a_{x_j} > b_{y_j}$ then

c):

$A1 =< a_1, ..., a_{x_{j-1}} >,$
$A2 =< a_{x_j}, ..., a_n >,$
$B1 =< b_1, ..., b_{y_j} >,$
$B2 =< b_{y_{j-1}}, ..., b_n >,$
$A' =< a_{x_{j-1}+1}, ..., a_{x_j-1} >,$
$B' =< b_{y_j+1}, ..., b_{y_{j-1}-1} >;$

if instead $a_{x_1} > b_{y_1}$ then

d):

$A1 = \phi,$
$A2 =< a_{x_1}, ..., a_n >,$
$B1 =< b_1, ..., b_{y_1} >,$
$B2 = \phi,$
$A' =< a_1, ..., a_{x_1-1} >,$
$B' =< b_{y_1+1}, ..., b_n > .$

Notice that in case a) we have completed the median split of $A \perp B$.

Corollary 1a. If $x_1 = 1$ then in case d) we have completed the median split or if $x_t = n$ then in case b) we have completed the median split.

The following two algorithms accomplish the divisions of A and B of Theorem 1; they determine if the j of the above theorem exists and its location. We present two different versions. The first takes advantage of the possibility that $a_{x_i} = b_{y_i}$ by exiting a loop early, but it is a CRCW algorithm; the other does not try to make use of this knowledge and is an EREW algorithm. Both assume that $x_i < x_{i+1}$; that is the x_i's are unique. An array of boolean variable's, b[], is used locally to determine sign changes.

procedure MSP(la, ua, lb, ub, p){ a CRCW version}
$b[1, ..., p+1] = true : boolean$
$c : integer$
$exit = false : boolean$
begin
Reducetoequal(la, ua, lb, ub)
while $la \leq ua$ **do**

```
      {one possible choice of an x_i and y_i sequence appears below}
            for i := 1 to p do in parallel
                x_i := la + i * ⌈(ua - la + 1)/(p + 1)⌉
                y_i := la + ub - x_i
            end for

      { from Theorem 1}
            for i := 1 to p do in parallel {one time step}
                if a_{x_i} > b_{y_i} then
                    if i = 1 then
                        ua := x_1 - 1
                        lb := y_1 + 1
                        exit:=true
                    end if
1:              else if a_{x_i} < b_{y_i} then
                    if i = p then
                        la := x_p + 1
                        ub := y_p - 1
                        exit:=true
                    else
                        b[i]:=false
                    end if
2:              else {a_{x_i} = b_{y_i}}
      { concurrent writing may occur in this next statement}
3:                  c:=i
                end if
            end for

4:          barrier

            if not exit then
                for i := 1 to p do in parallel {one time step}
5:                  if c=i then
                        la := x_i + 1
                        ua := x_i
                        lb := y_i
6:                      ub := y_i - 1
                    end if
                    if (not b[i]) and b[i + 1] then
                        la := x_i + 1
                        ua := x_{i+1} - 1
                        lb := y_{i+1} + 1
                        ub := y_i - 1
                    end if
```

```
        end for
      end if
    end while
    la ⇔ ua
    lb ⇔ ub
end. {MSP}
```

In this algorithm as in all the algorithms that appear below we assume that a processor knows its own index. The processor with the same index value as the for index in a construction "for...in parallel" in meant to execute the code within the loop. The "barrier" statement (statement labeled 4:) forces synchronization; it causes the processes to wait at this statement until all have arrived.

If the x_i's (and y_i's) are unique, the only place that a concurrent read can occur in this algorithm is in the "if c=i..." statement. Because $a_{x_1} > b_{y_1}$ and $a_{x_p} < b_{y_p}$ cannot occur simultaneously, no concurrent write to variable "exit" can occur. Interestingly, the algorithm correctly works no matter which i is the last one stored in c. To make this procedure into an EREW version we need to remove the **else** clause that tests for equality (statements labeled 2: and 3:) and also the **if** statement (statements 5: through 6:) and replace the test condition $a_{x_i} < b_{y_i}$ (statement 1:) by $a_{x_i} \leq b_{y_i}$. These changes no longer allow exiting the loop when $a_{x_i} = b_{y_i}$ as the CRCW algorithm allows.

Corollary 1b. If $x_i = la + i\lceil(ua - la + 1)/(p+1)\rceil, i = 1, 2, ..., p$, then using the MSP procedure we can determine the median-split of $A \perp B$ in $\lfloor \lg(n)/\lg(p+1) \rfloor + 1$ comparison steps.

Proof: After the first parallel comparison step we have reduced the sizes of A and B to at most $n/(p+1)$ each. The while loop is not exited until A (and B)

is reduced to an empty set. Let z be the number of repetitions of the loop, then

$$n/(p+1)^z < 1$$

holds. Solving of z gives $z = \lfloor \lg(n)/\lg(p+1) \rfloor + 1$. ∎

Corollary 1c. MSP requires one comparison step if $p \geq n$.

If $p \geq n$, then set $x_i = i$ for $i = 1, 2, \ldots$. Each element of A and B would then be compared to its complement in one parallel step. Therefore after one iteration of the while loop in MSP, $ua < la$, $ub < lb$; and $A1 = < a_1, \ldots, a_{ua} >$, $A2 = < a_{la}, \ldots, a_n >$, $B1 = < b_1, \ldots, b_{ub} >$, and $B2 = < b_{lb}, \ldots, b_n >$. ∎

Section 2.3 k-Splitting

The definition of a median-split is easily generalized to a *k-split*. Define a *k-split* as partitioning $A \perp B$ so that $|A1| + |B1| = k$ and $|A2| + |B2| = m + n - k$; and define $X_{[k]}$ as a set of the k smallest values of a set X. Interestingly a k-split problem can be made equivalent to a median split problem as shown below.

Theorem 2. Assume without loss of generality that $|A| = m \geq |B| = n$. A k-split of $A \perp B$, $1 \leq k \leq m + n$, is equivalent to a median split of

a) $A_{[k]} \perp B_{[k]}$ if $1 \leq k \leq n$, or

b) $(A_{[k]} - A_{[k-n]}) \perp B$ if $n < k \leq m$, or

c) $(A - A_{[k-n]}) \perp (B - B_{[k-m]})$ if $m < k \leq m + n$.

Proof. A k-split of $A \perp B$ can be thought of in terms of determining the k-th smallest of $A \perp B$ in the following sense: If $A \perp B$ is k-split then $|A1| + |B1| = k$ and hence the k-th smallest item (and those that are less) is to be part of $A1 \perp B1$. Consequently the sets $A2$ and $B2$ are where we put the non-candidate elements

for the k-th smallest. Clearly the elements $\{a_i, i > k\}$ and $\{b_j, j > k\}$ cannot be candidates for the k-th smallest and consequently they can immediately be removed from A and B and put into $A2$ and $B2$, respectively. Doing so leaves us with a). The problem remaining is a median split of $A_{[k]} \perp B_{[k]}$.

Likewise when $n < k \leq m$ (case b) we can remove from A and add to $A2$ the elements $< a_i, i > k >$ for the same reason. Furthermore, the first $k - n$ elements of A, $< a_i, i \leq k - n >$ must belong to $A1$ as they could belong to $A2$ only if there are k elements of $A \perp B$ less than or equal to each of them. There cannot be, for at most there are the n elements from B.

If $m < k \leq m + n$ as in case c) the arguments of case b) used in removing $A_{[k-n]}$ from consideration from set for A hold for B as well, and consequently the first k-m elements of B must be part of $B1$. ∎

The ideas of theorem 2 are incorporated into the procedure Adjustbound below. Also below procedure ksplit, that does k-splitting, first calls this procedure before invoking Mediansplit.

```
procedure Adjustbound (la, ua, lb, ub, k)
begin
  if k ≤ min(m, n) then
    ua := la + k − 1
    ub := lb + k − 1
  else if min(m, n) < k ≤ max(m, n) then
    if  n < m then
      la := la + k − n
      ua := la + k − 1
    else
      lb := lb + k − m
      ub := lb + k − 1
    end if
  else {k > max(m, n)}
```

```
      la := la + k − n
      lb := lb + k − m
    end if
end. {Adjustbound}

procedure ksplit (la, ua, lb, ub, k)
{ ksplit will split A ⊥ B}
{ so that |A1| + |B1| = k, |A2| + |B2| = m + n − k}
begin
   Adjustbound(la,ua,lb,ub,k)
   Mediansplit(la,ua,lb,ub)
end {ksplit}
```

We can make this algorithm into a parallel version by replacing the call to Mediansplit with a call to MSP. This gives us a new degree of freedom in that we can assign some of the p processors to do the k-split. We call this procedure KSP and write it below for reference.

```
procedure KSP (la, ua, lb, ub, k, p)
begin
{1 ≤ p ≤ min(k, m + n + 1 − k, n, m)}
   Adjustbound(la, ua, lb, ub, k)
   MSP(la, ua, lb, ub, p)
end. {KSP}
```

Corollary 2. By using KSP a k-split of $A \perp B$ can be accomplished in single comparison step with $\min(k, m+n-k, n, m)$ processors for $1 \leq k \leq m+n$.

Consequently if, as we have been assuming $n \leq m$, then KSP can k-split $A \perp B$ in single comparison step needing no more than k processors if $k \leq n$; or needing no more than n processors if $n < k \leq m$; or needing no more than $m+n-k$ processors if $m < k \leq m+n$. The following symmetry exists: we require the same number of processors to calculate the i^{th} largest value ($= m+n+1-i$ smallest) as the i^{th} smallest.

Section 2.4 Multiple Split Points

In this section we use the results of the previous sections to help determine multiple split pairs (or equivalently multiple split lines) simultaneously. The problem addressed here is to simultaneous split $A \perp B$ according to a splitting vector $K = (k_1, k_2, ..., k_t)$, where $k_i < k_{i+1}$, $i = 1, ..., t-1$. Doing so results in sectioning $A \perp B$ into $t+1$ sections. If we denote each section by A_i and B_i then $\sum_{j=1}^{i} |A_j| + |B_j| = k_i, i = 1, ..., t$, $\cup_{i=1}^{t+1} A_i = A$, $\cup_{i=1}^{t+1} B_i = B$. The first procedure presented below is a straightforward generalization of the Mediansplit procedure (remember it uses a single processor to find a single split pair). This algorithm assigns one processor to each k_i so it requires that $t = p$. It is a CREW algorithm.

```
procedure Kpartition { parallel p splitting}
begin
   la := lb := 1
   ua := m; ub := n
   for  i := 1 to p do in parallel
      ksplit(la, ua, lb, ub, k_i)
   end for
end. {Kpartition}
```

As we have p calls to ksplit running in parallel, the complexity of this algorithm is equal to the complexity of ksplit which is $\lfloor \lg(\min(m,n)) \rfloor + 1$.

We now present an algorithm that determines $t = p/2^q$ sections, for a fixed integer q, $0 \leq q \leq \lfloor \lg(p) - 1 \rfloor$, so that all sections are of equal size (except possibly the last), i.e. $|A_i| + |B_i| = \lceil (m+n)/t \rceil, i = 1, ..., t$. The algorithm is a CREW algorithm. For the sake of efficiency in explanation (but not necessarily) assume that p, m, and n are powers of two.

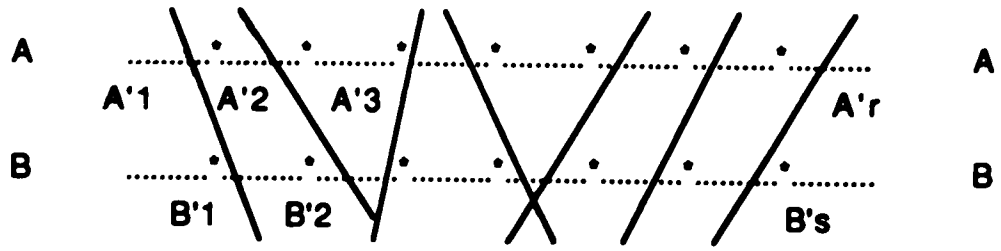
Algorithm A Evenly split $A \perp B$ into t sections.

Step 1.

Create A^*, a subsequence of A, that are the $i\lceil(m+n)/t\rceil, i = 1,2,...$ elements from the original sequence. Likewise create B^* from B. We do not need a_m or b_n to be an element of A^* or B^*; therefore remove them if they have been included. (They are be by our assumption that p, m, and n are powers of two.) Consequently $|A^*| + |B^*| = t - 2$.

The A^* subsequence segments the A sequence. Call these segments A'_j, $j \geq 1$ and have the elements of A'_j be the elements of A between a^*_{j-1} and a^*_j (see figure 4). Likewise denote by B'_j, $j \geq 1$ the segments of B. $|A'_j| \leq \lceil(m+n)/t\rceil - 1$ $j \geq 1$; likewise for $|B'_j|$, $j \geq 1$.

Figure 4



{ Basically, steps 2 through 5 are designed to find the split lines of $A^* \perp B^*$ for $K = (1,3,5,...,t-3)$. Note that $|K| = t/2-1$. If there are enough processors available this can be accomplished in one comparison step (step 2), otherwise we resort to recursively reducing the problem (steps

3 through 5). We can represent the result of this partitioning pictorially as in figure 4. The stars in the figure represent the starred sample. The lines crossing both sequences denote the split lines. Splitting using the odd starred points results in there being two starred values between each split line except the ends, which have one. We say that a splitting line *intersects* segment A'_j (or B'_j) if it between a'_{j-1} and a'_j.}

{ Consider the total number of comparisons needed to calculate the split points of K in the worst case. Since when k is in the range $n \leq k \leq m$, corollary 2 says that the number of comparisons needed is equal to $\min(n, m+n-k) = n$, the worst case number of comparisons occurs when $m = n$, for k in this range. Assume this is the case. Since $(1+3+...+(2x-1)) = x^2$, and by the symmetry discussed following corollary 2, the worst case requires $2(1+3+...+(t/2-2)) = (t-2)^2/8$ (for $t/2$ odd) or $2(1+3+...+(t/2-3))+(t/2-1) = (t-2)^2/8+1/2$ (for t/2 even). Both sums are bounded by $t^2/8$ for $t > 1$.}

Step 2.

If $t > 2\lceil\sqrt{2p}\rceil$ then continue with step 3, otherwise, use Kpartition on $A^* \perp B^*$ with $K = (1,3,5,...,t-3)$. This step determines $t/2-1$ split lines of $A^* \perp B^*$ in single step. Continue with **step 6**.

Step 3.

Create A^{**} and B^{**} as subsequences of A^* and B^* that are the $i\lceil(t-$

$2)/(2\sqrt{2p})]$, $i = 1, 2, \ldots$ elements from the A^* and B^* respectively sequences. Remove the last elements of A^* and B^* from A^{**} and B^{**} if included.

{Consequently $|A^{**}| + |B^{**}| = 2\lfloor\sqrt{2p}\rfloor - 2$.}

Step 4.

Use KSP on A^{**} and B^{**} for each element of K, $K = (1, 3, \ldots, 2\lfloor\sqrt{2p}\rfloor - 3)$.

{There are enough processors available to do this in parallel in one step. (See discussion above Step 2).}

Step 5.

The A^{**} subsequence divides the A^* sequence into segments; call the segments $A_i'^*, i = 0, 1, \ldots$. Likewise define $B_i'^*$. Since $|K| = \sqrt{2p} - 1$, the odd split points determine $\sqrt{2p} - 1$ split lines. Think of these split lines as intersecting segments of the A^* and B^* sequences.

In parallel, do a median split of the $\sqrt{2p} - 1$ pair of segments, $A_i'^*$ and $B_j'^*$, that each split line intersects.

{Since the size of $A_i'^* \perp B_j'^*$ is bounded by $\lceil(t-2)/\sqrt{2p}\rceil - 1$ we can accomplish this median split using MSP on each in one step, by assigning $\sqrt{2p}/2$ processors for each median split. (see corollary 1c.)}

{At this point we have split $A^* \perp B^*$ into $\lfloor\sqrt{2p}\rfloor$ equal sized segment, each of size $\lceil(t-2)/\sqrt{2p}\rceil$ but we need to split it into t segments. We

now treat each segment as an independent problem, assigning $p/\sqrt{2p} = \lfloor\sqrt{p/2}\rfloor$ processors to each, and recursively repeat steps 4 and 5 on each until each segment is of size two.}

{Steps 3 through 5 contain two comparison steps. If we take this path after k iterations we have reduced the interval size by $(2p)^{1-(1/2)^k}$. Thus the complexity, in the number of comparisons, of these steps is $2(\lg\lg p - \lg(q+1))$. }

Step 6.

{ In either step 2 or in steps 3 through 5 we have determined the location of the odd split lines of $A^* \perp B^*$. That is, we have determined the sequence $A^* \perp B^*$ up to a partial order.}

{ Step 6 does a simple comparison of the starred values between the split lines. This step fully sorts the starred values and can be thought of, in effect, obtaining $t-3$ split lines or split pairs.}

Using processor i determine the order of two elements of $A^* \perp B^*$ immediately to the left of the $i+1-st$ splitting line, $i \geq 1$.

Step 7.

{ At this point we have $t-3$ splitting lines. As in **step 5**, think of the split lines of $A^* \perp B^*$ intersecting A and B within specific segments. This step positions the split lines of $A^* \perp B^*$ to their accurate positions in $A \perp B$. }

In parallel do a median split of the $t-3$ segment pairs of A and B that each of the $t-3$ splitting lines intersect. At the same time do a median split of A'_1 with B'_1 and of the last two segments of A and B. These median split lines divide $A \perp B$ into t equal sized sections all, except possibly the last, each equal to $\lceil (m+n)/t \rceil$.

{In this step we are executing $t-1$ processes in parallel and have $p \geq t-1$ processors available. Thus using MSP this step can be executed in $\lfloor \lg((m+n)/t) / \lg(\lfloor p/(t-1) \rfloor + 1) \rfloor + 1$ comparisons by assigning $\lfloor p/(t-1) \rfloor$ processors to each process (see corollary 1b). }

{The idea of recursively reducing the problem using a sample of $\sqrt{p}$ elements comes from Valiant [1975].}

Theorem 3. The number of comparisons necessary if Algorithm A partitions $A \perp B$ into t equal sized segments is

$\lg((m+n)/t) / \lg(\lfloor p/(t-1) \rfloor + 1) + 2(\lg\lg p - \lg(q+1)) + 2$, if $t > 2\sqrt{2p}$, or

$\lg((m+n)/t) / \lg(\lfloor p/(t-1) \rfloor + 1) + 3$, if $t \leq 2\sqrt{2p}$.

Corollary 3. If $p = n = m$, then $A \perp B$ can be partitioned into t parts in $2(\lg\lg(p) - \lg(q+1)) + 1$ comparisons.

If $p = n = m$ then steps 1 and 6 of Algorithm A are unnecessary and $A^* = A$ and $B^* = B$.

Section 2.5 Applications of K-splitting to Merging and Sorting

Although k-splitting can be useful in its own right this paper is predicated on the need to create $A \perp B$. Certainly two natural applications of k-splitting are

merging and merge sorting. Previous work on parallel merging includes Akl [1987] and the previously mentioned Valiant's algorithm [1975] with Kruskal's [1983] improvement. Akl's[1987] algorithm, like Valiant/Kruskal's, partitions $A \perp B$ into disjoint sections. Akl does this by making use of Rodeh's idea of a distributed median [6]. The distributive median is used to first divide $A \perp B$ into two disjoint parts; the distributive median of both parts are then found and this process repeated until these are p sections (p is a power of two). The time complexity of this algorithm is $(m+n)/p + c_1 \lg(p) \lg(m+n) + c_2$ $c_1 \geq 1$. Valiant's algorithm (with or without Kruskal's improvement) is faster, but Akl's algorithm is a EREW algorithm.

We can use Algorithm A to section $A \perp B$ as well. By first using $t = p/2$ in Algorithm A to get $p/2$ disjoint sections, two processors can then be used for the actual merging of each pair of sections (see e.g. Knuth [1973]). Adding the merging time to time given by Theorem 3, the time complexity is then seen to be $(m+n)/p + \lg((m+n)/p)/\lg 3 + 2\lg\lg(p) + 1$. The second term is lower by a factor of $1/\lg 3 \approx 0.63$ than the Valiant/Kruskal's algorithm and the coefficient in front of the $\lg\lg$ term is 0.11 greater. If we let $r = (m+n)/p$, the ratio of the number of values over the number of processors, then it can be shown that Algorithm A is faster than the Valiant/Kruskal version if

$$p < 2^{r^x}, \quad \text{where } x = (\lg 3 - 1)/(2\lg 3 - 3) \approx 3.4.$$

Hence, if we need to merge twice as many numbers as processors, Algorithm A is faster when the number of processors is less than 512; if the ratio is three to one

then Algorithm A is faster if $p < 4.1 \times 10^{12}$! Algorithm A also has the advantage over the Valiant/Kruskal algorithm in that it creates sections that are of equal size.

The key reason Algorithm A is faster than Valiant's is that by finding the odd splitting points and hence only determining the partial order of the elements in the main loop (steps 3- 5), we are able to reduce the number of comparisons in the loop by about half. Had we used a $K = (1, 2, ...)$, then the number of comparisons would be the same as a Valiant's and the speed of the algorithm would have been essentially the same.

To sort $N > p$ values first divide the values into p parts. In parallel, use an efficient sequential sort on each of the p parts, one part on each processor. This requires $\lceil N/p \rceil \lg \lceil N/p \rceil$ comparison steps at most. Repeatedly, two-way merge each pair of the resulting sequences, first using two processors and Algorithm A for each pair, then four processors for each pair, then eight, etc, each time using Algorithm A, until sorted. This requires $\lceil \lg p \rceil$ steps. Consequently, from Theorem 3, sorting can be accomplished using Algorithm A in $\lceil N/p \rceil \lg N + 0.63 \lceil \lg p \rceil \lg(N/p) + 2 \lceil \lg p \rceil (\lg \lg(p) + 1)$, which is optimally efficient as long as the first term is larger than the others. This is true for $p \leq N / \lg \lg N$. If $p = N$ then as a consequence of corollary 3, we can sort in $O(\lg(N) \lg \lg(N))$ which is not optimal. Recently Cole [1988] has devised an optimal algorithm for this case.

Section 2.6 EREW splitting

Now we present two EREW algorithms to determine p equal sized split pairs (eg $|A_i| + |B_i| = (m+n)/p$, $i = 1, ..., p$.

Akl's[1987] splitting algorithm, an EREW type, is described in section 1.7.4. Its complexity is $O(\lg(p)\lg(m+n))$.

We present two algorithms. One for $p = (m+n)/2$, and one for $p < (m+n)/2$. For the sake of efficiency in explanation we also assume each of m, n, p to be a power of two. The first requires half as many processors as values to be split.

Case 1. $p = (m+n)/2$.

The algorithm first assigns p processors to run MSP to split $A \perp B$ into two equal sized pairs, $A1 \perp B1$ and $A2 \perp B2$. There are sufficient processors available so that this can be accomplished in single comparison step; see Corollary 1c. Now assign $p/2$ processors to each pair (child) to run MSP, again in one comparison step resulting in four equal sized pairs, etc. The process can be likened to a tree structure. It terminates when p equal sized pairs are generated, which take $lg(p)$ steps.

It is not difficulty to see that, during each step, the sublists on which processors are working to determine the new partition are all disjoint, and MSP can be an EREW algorithm no memory conflicts occur.

If $|B| = n \leq |A| = m$, and $p = (m+n)/2$. We store the splitlines into array A as storing nodes of a search tree. First store the p-splitting line into a_1, then store the $p/2^{th}$ and $3p/2^{th}$ splitlines into a_2 and a_3 respectively, and so on. Procedure PMSP is an implementation of the EREW splitting, which is coded as follows:

```
procedure PMSP (la, ua, lb, ub, p)
{la, ua, lb, ub are the bounds of input data}
La, Ua, Lb, Ub: array[1..p] of integer
begin
{ use parameter la, ua, lb, ub as initial lower or upper indices }
{ of p- splitting line }
  cobegin
     La[1] := la
     Lb[1] := lb
     Ua[1] := ua
     Ub[1] := ub
  coend
  for i := 1 to lg p - 1 do
     for j := 2^(i-1) to 2^i - 1 do in parallel
{ pass the parents' indices to their children }
        cobegin
           La[2j] := La[j]
           Lb[2j] := Lb[j]
           Ua[2j + 1] := Ua[j]
           Ub[2j + 1] := Ub[j]
        coend
        MSP(La[j], Ua[j], Lb[j], Ub[j], p/2^(i-1))
{ pass the parents' indices to their children }
        cobegin
           Ua[2j] := La[j]
           Ub[2j] := Lb[j]
           La[2j + 1] := Ua[j]
           Lb[2j + 1] := Ub[j]
        coend
     end for
  end for
  for j := p/2 to p - 1 do in parallel
     MSP(La[j], Ua[j], Lb[j], Ub[j], 2)
  end for
end { PMSP }
```

The procedure that follows is another version. In this version each child procedure can locate its nearest parents.

```
procedure EREWPMSP (la, ua, lb, ub, p)
{la, ua, lb, ub are the bounds of input data}
La, Ua, Lb, Ub: array[0..p] of integer
begin
```

```
{ use parameter la, ua, lb, ub as initial lower or upper indices }
{ of p^th splitting line }
  cobegin
      La[p/2] := la
      Lb[p/2] := lb
      Ua[p/2] := ua
      Ub[p/2] := ub
  coend
  for i := 1 to lg p − 1 do
      for j := 1 to 2^(i−1) do in parallel
          k_j := p/2^i(2j − 1) {k_j and q_j are local variables }
          q_j := p/2^(i+1)
{ pass the parents' indices to their children }
          cobegin
              La[k_j − q_j] := La[k_j]
              Lb[k_j − q_j] := Lb[k_j]
              Ua[k_j + q_j] := Ua[k_j]
              Ub[k_j + q_j] := Ub[k_j]
          coend
          MSP(La[k_j], Ua[k_j], Lb[k_j], Ub[k_j], p/2^(i−1))
{ pass the parents' indices to their children }
          cobegin
              Ua[k_j − q_j] := La[k_j]
              Ub[k_j − q_j] := Lb[k_j]
              La[k_j + q_j] := Ua[k_j]
              Lb[k_j + q_j] := Ub[k_j]
          coend
      end for
  end for
  for j := 1 to p/2 do in parallel
      k_j := 2j − 1
      MSP(La[k_j], Ua[k_j], Lb[k_j], Ub[k_j], 2)
  end for
end { EREWPMSP }
```

Case 2. $p < (m+n)/2$

If $p < (m+n)/2, n \leq m$, we consider multiple splitting $A \perp B$ at k_1^{th} and k_2^{th} ordered values ($k_1 < k_2$). $A1', A2', B1', B2'$ are used to represent the four subsets $A1, A2, B1$, and $B2$ which are defined by k_1- splitting. Likewise $A1'', A2'', B1'', B2''$ for k_2- splitting. Any $x \in A1'$, which is less than at least

$m+n-k_1$ elements of $A \perp B$, so that it less than $m+n-k_2 (m+n-k_2 < m+n+k_1)$ elements of $A \perp B$, which leads $x \in A1''$, we have $A1' \subseteq A1''$. By the similar argument, $A2'' \subseteq A2'$, $B1' \subseteq B1''$, and $B2'' \subseteq B2'$.

If k_1-splitting of $A \perp B$ is equivalent to median splitting (la', ua') and (lb', ub'), where la' and lb' are lower indices of sublists of A and B and ua' and ub' are upper indices of them and k_2-splitting $A \perp B$ equivalent to median splitting (la'', ua''), and (lb'', ub''), where la'', lb'', ua'', ub'' have similar meaning. Assume sublists (la', ua') and (lb', ub') have equal sizes. So do (la'', ua'') and (lb'', ub'').

Lemma 2. A k_1-splitting of $A \perp B$ is equivalent to a median splitting of (la', ua'') and $(lb' + (ua' - ua''), ub')$ if $ua'' < ua'$.

proof. Because of $A2'' \subseteq A2'$, every $a_x \in A2''$, $(x \geq ua'')$, can be removed to $A2'$, so ua' can be updated to ua''. The elements at the left most of sublist (lb'', ub'') with size $ua' - ua''$ cab be removed to $B1'$ since if they were in $B2'$, $B2'$ would overflow. ∎

Lemma 3. A k_2-splitting of $A \perp B$ is equivalent to a median splitting of (la', ua'') and $(lb'', ub'' - (la' - la''))$ if $la' > la''$.

proof. Because of $A1' \subseteq A1''$, every $a_x \in A1'$,$(x \leq la')$, can be removed to $A1''$, so la'' can be updated to la'. The elements at the right most of sublist (lb'', ub'') with size $la' - la''$ can be removed to $B2''$ because they have no way to be in $B1''$. ∎

Likewise for list B.

Lemma 4. A k_1-splitting of $A \perp B$ is equivalent to a median splitting of $(la' +$

$(ub' - ub''), ua')$ and (lb', ub'') if $ub'' < ub'$.

Lemma 5. A k_2-splitting of $A \perp B$ is equivalent to a median splitting of $(la'', ua'' - (lb'' - lb'))$ and (lb', ub'') if $lb' > lb''$.

To split $A \perp B$ into p equal sized segments, we create A^*, a subsequence of A, that are the $i\lceil(m+n)/(2p)\rceil^{th}, i = 1, 2, \ldots$ elements from original sequence. Likewise create B^* from B. We do not need either a_m or b_n to be an element of A^* or B^*; therefore remove them if they have been included. (They will be included by our assumption that p, m, and n are power of two.) Add two dummy records: $d_1 < min(a_1, b_1)$ and $d_2 > max(a_m, b_n)$. Consequently $|A^*| + |B^*| = 2p$.

Run EREWPMSP on A^* and B^* using p processors to split $A^* \perp B^*$ into p equal sized segments in $\lg(p)$ steps, i.e. each segment has two elements except the ends which have single element with one dummy, and there is a splitline between two consecutive segments.

The A^* subsequence divides the A sequence into segments; call the segments $A'_i, i = 0, 1, \ldots$. Similarly define B'_i. Think of these splitlines as intersecting segments of the A and B sequences. In parallel, do a median split of the pair of segments, A'_i and B'_j, that each split line intersects. If two or more splitlines intersect the same segment say A'_i (or B'_j). In parallel, doing median splitting of the pairs of segments of A and B sequences (as that in Algorithm A) may have concurrent reading. To avoid that, we schedule these $p-1$ median splitting jobs into a pipeline. First work on the $p/2^{th}$ splitline, when the sizes of its A'_i and B'_j are reduced to half of themselves, start working on the $(p/4)^{th}$ and $(3p/4)^{th}$ splitlines,

then work next on $(p/8)^{th}$, $(3p/8)^{th}$, $(5p/8)^{th}$, and $(7p/8)^{th}$, and so on,.... The job is scheduled like a balanced binary tree. Between the consecutive levels, there is one comparison time delay. If two or more splitlines in different levels of the tree intersect the same segment A'_i (or B'_j), they may read the same value from A'_i (or B'_j) but at different time by this scheduling method, so there is no concurrent read between levels.

Let us consider two splitlines k_1 and k_2, $(k_1 < k_2)$, which intersect the same segment in A (or B), are scheduled at the same level, By the scheduling method at least one splitline k, $k_1 < k < k_2$, at the higher level of the tree, intersects the same segment. Assume k^{th} splitline is intersecting (la, ua) and (lb, ub), k_1^{th} splitline intersecting (la_1, ua_1) and (lb_1, ub_1), and k_2^{th} splitline intersecting (la_2, ua_2) and $lb_2, ub_2)$. Without loss of generality assume $la_1 = la_2$ and $ua_1 = ub_2$, and $(la, ua) \subset (la_1, ua_1)$. If $la > la_2$, the bounds of k'_2s segments can be reduced to (la, ua_2) and $(lb_2, ub_2 - (la - la_2))$ by lemma 3, and if $ua < ua_1$, the bounds of k'_1s segments can be reduced to (la_1, ua) and $(lb_1 + (ua_1 - ua), ub_1)$ by lemma 2. One or two possible conditions above is true, i.e. k's bound reduces the bound for at least one child.

Lemma 6. If $x(x \geq 2$ integer) splitlines intersect the same segment in A (or B) at the same level, at least $x - 1$ pairs of bounds intersecting these splitlines can be reduced by their parents.

Proof. The x splitlines at least have $x - 1$ parents, which intersect the same segment. One parent may reduce the bounds of one or two children by the

discussion above. So $x-1$ parents reduce the bounds of at least $x-1$ children. ∎

Note that at most one out of x pair of bounds, which intersect the same segment at the same level, has to be reduced by reading data from the segment, any others do not. Therefore there is no concurrent reading from the common segment if use current lower and upper bounds of ancestors to reduce the bounds of their children.

The code of this EREW algorithm is as follows:

```
procedure EREWsplit(m, n)
La[], Ua[], Lb[], Ub[] : array of integer
flag[]: array of boolean
begin
        cobegin
            q := ⌈2pm/(m + n)⌉ − 1
            r := ⌈2pn/(m + n)⌉ − 1
        coend
{ set up x_i and y_i sequence and two dummies for A* and B* }
        cobegin
            for i := 1 to q do in parallel
               x_i := ⌈(m + n)/(2p)⌉ * i
            end for
            for i := 1 to r do in parallel
               y_i := ⌈(m + n)/(2p)⌉ * i
            end for
{ add two dummy elements }
            a_{x_0} := min(a_1, b_1) − 1
            b_{y_{r+1}} := max(a_m, b_n) + 1
        coend
{ run EREWPMSP on A* and B*}
{ set up la = 0, ua = q, lb = 1, ub = r + 1}
st1.    EREPWMSP(la, ua, lb, ub, p)
{ initially La[0] = Lb[0] = La[p] = Lb[p] = 1, }
{Ua[0] = Ua[p] = m, Ub[0] = Ub[p] = n}
        for i := 1 to p − 1 do in parallel
            La[i] := La[i] × ⌈(m + n)/(2p)⌉ + 1
            Ua[i] := Ua[i] × ⌈(m + n)/(2p)⌉ − 1
            Lb[i] := Lb[i] × ⌈(m + n)/(2p)⌉ + 1
            Ub[i] := Ub[i] × ⌈(m + n)/(2p)⌉ − 1
```

```
{flag[] keep flags }
            flag_i := true
        end for
{ assume p = 2^h - 1}
st2.    for i := 1 to lg(m + n) - 1 do
st3.        for s := 1 to min(i, h) do in parallel
st4.            for j:=1 to 2^(s-1) do in parallel
{flag[] controls to set up k_j, nl_j, nr_j only once }
                    if flag_i then
{ set k_j^th ordered value of A ⊥ B as splitting point }
                        k_j := 2^(h-s)(2j - 1)
{ set nl_j to be the nearest splitting point started to the left of k_j}
                        nl_j := k_j - 2^(h-s)
{ set nr_j to be the nearest splitting point started to the right of k_j}
                        nr_j := k_j + 2^(h-s)
                        flag_i := false
                    end if
                    d_j := ⌊(Ua[j] - La[j])/2⌋
st5.                if La[nl_j] - La[j] ≥ d_j then
                        La[j] := La[j] + d_j
                        Ub[j] := Ub[j] - d_j
                    else if Ua[j] - Ua[nr_j] ≥ d_j then
                        Ua[j] := Ua[j] - d_j
                        Lb[j] := Lb[j] + d_j
                    else
{ may access elements at indices La[j] + d_j, and Ub[j] - d_j}
                        reducing(La[j], Ua[j], Lb[j], Ub[j])
                    end if
                end for {j}
            end for {s}
        end for {i}
end. { EREWsplit}
```

Let us analyze this algorithm statement by statement. There are two cobegin - coend blocks, before the statement st1., which set up at most $2p$ evenly spaced elements from A or B as x_i and y_i, which takes single step. st1. is a procedure call, which takes $\lg(p)$ time to split the sequence $A^* \perp B^*$ at all its odd points. Thus there are $p-1$ pairs of A'_i and B'_j with size $(m+n)/p$ each. The "for" loop, before st2., takes O(1) time. st2., st3., and st4., three nested for loops, implement the

pipeline scheduling strategy. When i equals lg(p) in st2., all jobs are scheduled, the splitting terminate after lg($(m+n)/(2p)$) additional steps. Therefore the total steps of the three nested "for" loops are lg($m+n$) $-$ 1. Within the "for" loop, there are two "if" statement, which take constant time. Thus the time complexity of this algorithm is $O(\lg(m+n))$.

Theorem 4. The number of comparisons necessary if Algorithm EREWsplit partitions $A \perp B$ into p equal sized segments is

$$O(\lg(m+n)).$$

Section 2.7 EREW Merging and Sorting

When two sorted lists are split into p disjoint sections, assign one processor to merge one pair. The time complexity for merging is $O((m+n)/p + \lg(m+n))$, which is faster than Akl's algorithm by a factor of lg(p).

To sort $N > p$ values, first divide the values into N/p parts. In parallel use an efficient sequential sort on each of the p parts. This takes $O(N/p \lg(N/p))$ time. Repeatedly merge each of the resulting sequences, first with two processors, then four processors, then eight, etc, each time using the EREWsplit, until sorted. This requires $\lfloor \lg p \rfloor$ steps. Consequently, from Theorem 4 sorting can be accomplishing using Algorithm EREWsplit in $O(N \lg N/p + \lg p \lg N)$ time, which is optimal efficient as long as the first term is larger than the second. This is true for $p \leq N/\lg(N)$.

Chapter 3 An Improved Parallel Quicksort Algorithm on an MIMD Machine

Section 3.1 Introduction

This chapter deals with improvements possible when quicksort is made into a parallel algorithm. The input is N keys in an array $A = (a_1, ..., a_N)$. We assume that the keys are pairwise distinct. This is not a real restriction since if we are given N arbitrary keys $a_1, ..., a_N$ we can replace each a_i by the tuple (a_i, i) and define an order of the tuples by $(a_i, i) < (a_j, j)$ iff $a_i < a_j$ or $a_i = a_j$ and $i < j$.

Because it has excellent average-case behavior, quicksort is a commonly used sorting algorithm on serial computers. Given a random permutation of N keys, quicksort can be expected to sort the N keys in about $2N \lg N$ comparisons [Knuth, 1973]. Quicksort is generally written as a divide-and-conquer recursive algorithm. As such it splits an unsorted set of values into two disjoint subsets (subarrays) by creating one subset with values greater than a chosen pivot value and the other with values less than the pivot. Its sorting of each subset is accomplished by independently using recursive calls to quicksort on each subset.

A shared memory MIMD version of a quicksort algorithm was suggested by Sedgwick [1978], implemented by Deminet [1982] and Quinn [1988], and analyzed by Evans and Dunbar [1982], and by Evans and Yousif [1985]. A task queue is used to store the tuple of the upper and lower bound indices of subarrays that are unsorted. Any processor that is unoccupied checks the task queue to see if

it is empty. If it is not, the processor locks the queue, pops a tuple of indices of a task from the queue, and unlocks the queue. Locking is necessary to prevent concurrent problems. The processor then splits the subset, using the splitting algorithm described above, into two smaller subsets, containing keys less than or greater than a chosen pivot value. The processor then locks the task queue, pushes the indices of one of unsorted subarrays onto the queue, and then unlocks the queue. The processor repeats the splitting process on the other unsorted subarray.

This implementation of quicksort exhibits a power loss due to what we have called starvation. At first only the single tuple of the array boundaries are in the stack. Since the first step of the quicksort consists of splitting the keys into two subsets by one processor, during this period one processor only is working. As this splitting requires an inspection of all keys, the time to split is proportional to the number of elements in the set N. Once this is done, only two processors can work on splitting the resulting two subsets into four, etc. The total loss of efficiency is $(p-1)N + (p-2)N/2 + ... + (p/2)2N/p = \sum_{j=0}^{k-1}(p-2^j)N/2^j = 2N(p-1-k)$ (supposing $p = 2^k$).

Quinn [1988] implemented this algorithm with two modifications to improve the running time. One modification used the median of the first, middle, and last elements of the subarray as the pivot value. The other used a linear insertion sorting algorithm to sort a subarray that has fewer than ten elements. These are not really new ideas, as both modifications are standardly used for sequential

algorithms (Sedgwick [1978]). Quinn points out that even with these modifications the parallel quicksort algorithm efficiency is rather mediocre. He states that the most important factor is the low amount of parallelism that occurs early in the algorithm's execution (i.e., starvation).

Mattel and Gusfield[1989] have given a version of parallel quicksort algorithm which sorts N elements using N processors in $O(\lg N)$ time in a CRCW *random-write* PRAM model. No starvation occurs early in the algorithm's execution. Each processor has one element of the set to be sorted. The values are be sorted in an array $A[1,...,2^N-1]$. All processors try to write their value to $A[1]$. Each processor decides if their value is larger or smaller than $A[1]$. Because of the random write model one random processor succeeds. Assume x_i is written into $A[1]$. Elements less than x_i are called a "*small*", and elements bigger than x_i are called a "*large*". All processors with small values try to write to $A[2]$ and all processors with large values try to write to $A[3]$. The remaining (active) processors now are partitioned into four groups: those small in both iterations, those small in the first but large in the second, those large in the first and small in the second, and those large in both. In the third iteration each of these groups of processor write to locations $A[4]$ through $A[7]$, in the order above. This procedure continues until all n values have been written into the array A.

After writing all the values into the array A, the array contains the sorted list of values. To see how this, think of the array A as a representation of a binary search tree. Consider $A[1]$ the root, $A[2]$ the root of the left subtree, and $A[3]$ the

root of the right subtree. In general, $A[2i]$ is the left child of $A[i]$, $A[2i+1]$ is the right child of $A[i]$, and $A[\lfloor i/2 \rfloor]$ is the parent of $A[i]$. From this binary search tree we obtain a sorted list by doing a preordered traversal.

On average the time complexity to write the values into A is $O(\lg N)$, the average depth of the binary search tree. The algorithm has a small constants associated with its running time. The disadvantage of this algorithm is that it uses a more powerful model, CRCW, rather than EREW or CREW, and has a larger space requirement than the other sorting algorithm. It needs $2^N - 1$ space in the worst case, and N^3 space in the average case.

Mattel and Gusfield [1989] give two techniques to reduce the space requirement. One of these techniques is to be more careful in choosing the pivot element which divides the current list into large and small elements. They suggest using the median of three: choose three random elements, and use the median of these as the pivot element. Simulations they did show that the median of three (or five) rule reduce the space bound to below N^2. The other technique is to combine the median of three with an attempt to allocate less space for empty subtrees. To do this they build the tree for $\lfloor \lg N \rfloor$ iterations. Then they collect all the subtrees which still have at least one element and assign them to the next m locations of A, where m is the number of such subtrees. After that they continuously assign space to their children as before, until the total space used is $2N$. They collect the nonempty subtrees and reassign them to the next consecutive locations of A. They continue splitting these subtrees until total space used reach $3N$, at which

point they collect the nonempty subtrees and reassign location. This continues until the entire tree is built. The process of collecting the nonempty subtrees and reassigning addresses in A can be done in $O(\lg N)$. Their simulations suggest that it uses linear space. On problems of up to 8 million elements at most $6N$ locations in A were needed to complete the computation.

Heidelberger, Norton, Robinson, and Watson [1987] give a parallel quicksort algorithm which sorts N elements using N processors in $O(\lg N)$ time on a CRCW PRAM machine. They cleverly make use of the Fetch and Add ($F\&A$) instruction to remove the bottlenecks.

They partition an array A into two subarrays which are the left and right portions of A by copying the values into another array B. The partitioning is done as one parallel loop. Let I, L, and R be shared integer variables. I (initialized to 1) is used to index A. l (initialized to 1) and r (initialized to N) are used to point to positions from the left and right of B; $B(l)$ is the location into which an element less than pivot V is stored and $B(r)$ is the location into which an element greater than or equal to V is stored. t_I is a local variable. $F\&A$ distribution makes the partitioning algorithm simple:

```
for I := 1 to N do in parallel
    t_I := A(I)
    if (t_I < V) then
        B(F&A(L, 1)) := t_I
    else
        B(F&A(R, -1)) := t_I
end for
```

They also give two improved algorithms: one that is in-place for $p \ll N$,

the other that reduces the number of $F\&A$ operations. The in-place partitioning removes the need for dual copies by adding a small constant-time procedure to clean up at the end. The variables I ,L, and R have the same initial values and meaning as that above. l and r are local integers into which the results of Fetch-and-Add on L and R are stored. Let $N = R + 1 - L$, and let $Swap$ be a local boolean variable indicating whether or not a processor has fetched an element from the left which is greater than V and needs to be swapped with an element fetched from the right. Swap is initialized to be false. Through the use of the parallel for loop each processor does a $F\&A(I, 1)$ to determine if any more elements need to be examined. If the value returned is less than or equal to N it does either step 1 to 2 below and then gets another value of I.

1. If $Swap = False$, then the processor allocates an element from the left by executing $l = F\&A(L, 1)$. If $a[l] \geq V$, it sets $Swap = True$.

2. If $Swap = True$, then the processor allocates an element from the right by executing $r = F\&A(R, -1)$. If $a[r] < V$, then $a[l]$ and $a[r]$ are interchanged and $Swap$ is set back to $False$.

At the end of the parallel for loop, $L = R - 1$. However, L may not be the number of elements less than V since a processor may have fetched an element $a[l] \geq V$ from the left but was unable to find an element from the right with which to swap $a[l]$. This is the case if $Swap = True$. To clean up needs a procedure which at most performs P Fetch-and-Add.

Reischuk [1985] has described a quicksort-like parallel algorithm which runs

in $O(\lg N)$ expected time, using N processors on a parallel decision tree model (Valiant [1975]). In a parallel decision tree model, the only work charged is for the comparison of pairs of elements.

The first part of his algorithm makes use of a probabilistic parallel selection algorithm. Suppose we wish to select the l^{th} ordered value from set Y. If the size of Y, m, is not greater than $\sqrt{2N}$, then it is possible to compare every pair of elements in parallel in Y in one step determining the l^{th} ordered value. If $m > \sqrt{2N}$ then choose a sample S of size $s = \sqrt{m}$ and let $f = f(m) = m^{7/16}$. In parallel do a pairwise comparison of every element in S. Define two integers $t_1 = \max(0, \lfloor l(s+1)/(m+1) - f \rfloor)$ and $t_2 = min(s+1, \lceil l(s+1)/(m+1) + f \rceil)$. Let $s_i, i = 1,2$ denote the element of rank t_i in S. Compare in parallel every element of Y with s_1, and compare in parallel every element of Y with s_2. Split Y into three parts:

$$A = \{y \in Y | y < s_1\}$$

$$B = \{y \in Y | s_1 \leq y \leq s_2\}$$

$$C = \{y \in Y | s_2 < y\}.$$

If $|A| < l$ and $|A| + |B| \geq l$ then as next step select the $(l - |A|)^{th}$ value from set B, if $|A| \geq l$ then select the l^{th} value from A, and if $|A| + |B| < l$ select $l - |A| - |B|$ from C recursively.

Reischuk [1985] proves that for any k, $1 \leq k \leq N$, this probabilistic parallel decision tree algorithm selects the k^{th} smallest element of a set of N elements in $O(1)$ time with probability greater than $1 - exp(-N^{3/16}/4 + O(\lg N))$. As N increases this probability goes to one. A probabilistic selection algorithm running

in constant time implies a $O(\lg n)$ probabilistic sorting algorithm.

We now look into modifications of the parallel quicksort algorithm that reduce this starvation and so improve the speedup. The general idea is to have each of the p processors in parallel perform pivoting using a single pivot value. In standard quicksort, the pivoting step involves data movement, i.e., values less than the pivot are moved to the low indices and those greater than the pivot are moved to the higher indices. Trying to do this in parallel causes concurrency problems from the movement of data required by these parallel writes. Each write could require a lock out to protect the integrity of the data base. Below we give a solution to the problem that avoids this difficulty.

The quicksort partitioning algorithm that we give in section 3.2.1 is faster than the standard parallel version of quicksort. A standard version has a loss in efficiency of about $C \approx 2Np$ (see section 3.1), which we remove in our algorithm. The total cost to sort N elements for a standard parallel quicksort is $O(N \lg N + Np)$. Consequently this algorithm is optimal only if $p \leq \lg N$. The algorithm of section 3.2 is optimal if $p \leq N/(lgN)$. The algorithm has some disadvantages however. In the worst case a doubling the space is required. We show in section 3.2.2 that by using the Fetch-and-Add instruction. In section 3.3 we present a probabilistic quicksort partitioning, which reduces the space needed, and reduces the coefficient in the time complexity.

Section 3.2 Quicksort Partitioning Algorithms

Section 3.2.1 Algorithm A

We consider a shared memory MIMD machine with p independent processors and N keys stored in an array $< a_1, ..., a_N >$. We give an outline here of the partitioning algorithm. The algorithm moves the values in the array so that values less than the pivot value are to its left in the array, values greater are to the right and the pivot is in its correct place: the same result as if the partitioning were done sequentially. When called recursively, partitioning uses three variables, l and r, the bounds of a subset of A, $a_l, ..., a_r$, and a subset of processors of size $q \leq p$. Initially we call partitioning$(1, N, p)$.

procedure partitioning(l, r, q)

If $q = 1$ then partition using the sequential quicksort partition,

else $\{ q > 1 \}$ do steps 1) to 3).

1). Distribute the pivot V (stored in a_l), the left bound l, the right bound r, and the number of processors assigned q to each processor. Consider the input set (starting from a_{l+1}), so that elements whose indices modulo q are equal are in the same subset $(S_i, i = 1, ..., q)$. Each processor partitions one subset, therefore the q processors can simultaneously work without starvation or concurrency problems. Each processor runs a sequential quicksort partition using the same pivot V splitting each subset S_i into two parts. The one on the left side contains the keys that are less than the pivot V, and the one on the right side, keys greater than it. Call the index of the right-most element less than V, t_i, and left-most element greater than V, s_i (note that $s_i - t_i = q$). If $a_i < V$ for all i in a subset, the boundary

s_i can be defined as equal to $t_i + q$, or if $a_i > V$ for all i in the subset, we define $t_i = s_i - q$.

For example, suppose there are 17 elements to be partitioned in A, four processors are available, and the elements in the array A are as follows (pivot V is in a_1):

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}
$A = ($	13	12	1	23	25	18	9	7	16	14	22	10	0	15	20	17	5).

The processors are assigned values:

p_1	12				18				14				15			
p_2		1				9				22				20		
p_3			23				7				10				17	
p_4				25				16				0				5

$V = 13$. After the splitting step, subsets are reordered as follows:

	12'				14"				18				15			
		1				9'				22"				20		
			10				7'				23"				17	
				5				0'				16"				25

and $t_1 = 2$, $s_1 = 6$, t_2=7, $s_2 = 11$, $t_3 = 8$, $s_3 = 12$, $t_4 = 9$, and $s_4 = 13$. In this diagram, an apostrophe presents the location of t_i, a quote represents the location of s_i, and a single vertical line presents the location of the pivot in sequential quicksort.

2). Define the $smin = min(s_1, ..., s_q)$, $tmax = max(t_1, ..., t_q)$, and $s = tmax -$

smin. As long as $tmax > smin > 1$ we partition A into as many as three subsets: $[a_l, ..., a_{smin-1}]$, $[a_{smin}, ..., a_{tmax}]$, $[a_{tmax+1}, ..., a_r]$. The one on the left side of *smin* has values that are less than the pivot, the one on the right side of *tmax* has values that are greater than the pivot, and the elements in the interval $[smin, tmax]$, if there are any such elements, are mixed - there are some values greater than the pivot and some less. A swapping procedure is needed to partition the mixed part. The swapping procedure below first calculates the exact location of pivot V so that wrong -sided elements can be discovered and can be swapped.

The procedure swapping is as follows:

procedure Swapping

a). The q processors calculate the exact location of the pivot S in array A, where $S = l + \sum_{i=1}^{q} max(0, \lceil (t_i - l)/q \rceil)$, in $\lceil \lg q \rceil$ time. Then each of the processors compares its s_i and t_i with S.

Each processor has to move elements either to the right of V (eg $t_i > S$) or to the left of V (e.g. $s_i < S$), not both. Call a processor a *"right"* or a *"left"* processor.

b). Two auxiliary arrays are used to associate swapping the wrong-sided elements; a "*right array*", or a "*left array*". Two pointer sets, L_i and R_i $1 \leq i \leq q$, are used to point to positions into which the next wrong side element from set S_i moves. Each of the processors calculates two partial sums, one of them is $L_i = 1 + \sum_{j=1}^{i-1} max(0, \lceil \frac{S+1-s_j}{q} \rceil)$, and the other $R_i = 1 + \sum_{j=1}^{i-1} max(0, \lceil \frac{t_j - S}{q} \rceil)$, for $1 \leq i \leq q$, which takes $2 \lg q$ time in parallel. The right (left) processors

remove their wrong-sized elements into right (left) array using its pointer R_i (L_i) simultaneously; after a barrier, they get the same number of left (right) elements from the left array (right array) by using the same pointer R_i (L_i) in parallel in at most $2M/q$ time; after a barrier, processor p_1 swaps pivot V in a_l with a_S to move the pivot into its exact location.

3). We now call the partitioning procedure recursively on the two partitions created, proportioning the number of available processors. Let $M_1 = S - l, q_1 = M_1 q/M$. Call the procedures Partitioning($l, S-1, q_1$), and Partitioning($S+1, r, q-q_1$) recursively.

Section 3.2.2 Analysis of the Time Complexity

We analyze this algorithm by comparing the cost of the parallel version with the cost of sequential quicksort. The calls to the partitioning algorithm may be thought of as a "recursive" tree. The root of this tree is the initial call. The embedded calls create left and right children. We define the *cost*, C, as the product of T (the time spent for a job) and p (the number of processors assigned for the job), i.e. $C = T \times p$. Thus if we know the total cost for a job is C, and if the number of processors used stays fixed, the time complexity is C/p.

In step one, distributing the pivot, the left and right bounds l and r, and, q, the number of processors assigned to each processor takes $4 \lg q$ time, the cost being $4q \lg q$. Then in parallel each of the processors partitions the elements into their subset of array A, which costs $M + q$. In step 2, the calculation of S (the

exact location of V), the partial sum of $L_i = 1 + \sum_{j=1}^{i-1} max(0, \lceil \frac{S+1-s_j}{q} \rceil)$, and the partial sum of $R_i = 1 + \sum_{j=1}^{i-1} max(0, \lceil \frac{t_j-S}{q} \rceil)$, for $1 \leq i \leq q$, takes $3 \lg q$ time in parallel; the cost is $3q \lg q$, the comparison of S with s_i and t_i takes two time and costs $2q$. The swapping procedure at most costs $2M + 3q$. So on the whole in one partitioning step the cost is $3M + 7q \lg q + 6q$.

Lemma 3.1 explode the relations between M, size of a subset of A, and q, the number of processors assigned for a partitioning, if N, the size of input set and p, the number of processors available have $p \leq N/(\lg N)$ or $p \leq N^\gamma, \gamma < 1$.

Lemma 3.1.

$$\text{If } p \leq N/(\lg N), \text{then } q \leq M/(\lg M), \tag{3.1}$$

$$\text{if } p \leq N^\gamma, \gamma < 1, \text{then } q \leq M^\gamma, \tag{3.2}$$

at every level of this recursive parallel quicksort algorithm.

Proof. We prove these by induction. For level one, $q = p$ and $M = N$, the Lemma is true obviously. Suppose they are true for level $i - 1$ (we add subscripts to q and M to distinguish them at different levels); that is

$$q_{i-1} \leq M_{i-1}/(\lg M_{i-1})$$

(for eq(3.1)) and

$$q_{i-1} < M_{i-1}^\gamma$$

(for eq(3.2)). From step 3) of the algorithm

$$q_i = q_{i-1} M_i / M_{i-1} \tag{3.3}$$

and $M_i \leq M_{i-1}$, so $\lg M_i \leq \lg M_{i-1}$; therefore

$$q_i \leq (\frac{M_{i-1}}{\lg M_{i-1}})M_i/M_{i-1}$$
$$\leq M_i/(\lg M_{i-1}) \leq M_i/(\lg M_i).$$

The last step follows by the induction assumption. Hence the eq(3.1) follows.

To prove eq(3.2), we start from (3.3), on substitution from the induction assumption, hence

$$g_i \leq M_{i_1}^{\gamma} M_i/M_{i-1} = M_i/M_{i-1}^{1-\gamma} \leq M_i^{\gamma}.$$

Lemma 3.2 If $p \leq N/(\lg N)$, the cost for this parallel quicksort partitioning algorithm at any level is less than $16M$.

Proof. In any level, the cost is $3M + 7q\lg q + 6q$. Since $q \leq M/(\lg M)$ by Lemma 3.1, which implies $\lg q \leq \lg M - \lg\lg M$, therefore

$$q\lg q \leq (M/(\lg M)) \times (\lg M - \lg\lg M) \leq M,$$

for $M \geq 2$, and $7q\lg q + 6q \leq 13M$. Thus the cost in any level of the partitioning is less than $3M + 13M = 16M$. ∎

Since the cost of a sequential partition is M, the cost of our algorithm is 16 times greater.

Knuth [1973] analyzes the complexity of sequential quicksort in the average case. He uses C_N to denote the cost for sorting a set of N elements, and the cost is

$$C_N \approx 2(N+1)\lg(\frac{N+1}{K+2}), \quad \text{for} \quad N > K,$$

where K specifies the threshold value between straight insertion and partitioning. Since the cost needed for our algorithm is not more than 16 times as much as that for a sequential quicksort in any level, the total cost for sorting N elements by our algorithm is $16C_N$, i.e.

$$\begin{aligned} C_N &\approx 32(N+1)\lg(\frac{N+1}{K+2}), \quad \text{for} \quad N > K \\ &= O(N \lg N). \end{aligned} \tag{3.4}$$

Since there are p processors available, the work can be done in $O(N \lg N/p)$ time in parallel.

Theorem 3.1 It takes $O(N \lg N/p)$ time to sort N elements by p ($p \leq N/\lg N$) processors on a EREW SM-MIMD model by the parallel quicksort partitioning.

In addition, for $p \leq N/\lg N$, this algorithm is optimal, in view of the lower bound $\Omega(N \lg N)$ on sequential sort.

Section 3.2.3 Improving Algorithm A by Using *F&A* Operation

If a Fetch-and-Add instruction is available, the algorithm can be more efficient. To distribute a variable X to the processors, each processor can call $x = F\&A(X, 0)$, where x is variable local to the processor. Thus to distribute the pivot V, the left bound l, right bound r, and number of processors assigned q to each processor, call Fetch-and-Add replacing X by V, l, r, or q, and replacing x by different local variables. This only takes four time units. In step 2, to calculate the

exact location S of pivot V, each processor issues $S = F\&A(S, max(0, \lceil (t_i - l)/q \rceil))$ (S initialized to l). To distribute the calculated S, calls to $F\&A(S, 0)$ (in previous algorithm, we can calculate and distribute S in one procedure). Next compare S with its t_i and s_i to decide if it is a left or a right processor. These steps require four time units. To do the actual swapping of the mixed values use L and R, two shared integers (both initialized to 1), to allocate places in left array or right array. To get the pointer L_i or R_i, Each processor issues $L_i = F\&A(L, max(0, \lceil \frac{S+1-s_i}{q} \rceil))$, and $R_i = F\&A(R, max(0, \lceil \frac{t_i - S}{q} \rceil))$, which takes two time units. Therefore ten additional time units are required. Equivalently the cost is $10q$. In addition, splitting costs $M + q$, and swapping costs at most $2M + 3q$.

Thus the total cost is at most $3M + 14q$, which has an upper bound of $17M$, if we assume $q \leq M$. This is equivalent to 17 times the cost of a sequential quicksort partitioning. To sort the N elements the cost will be at most 17 times cost that spent by sequential quicksort, i.e.

$$C_N \approx 34(N+1)\lg(\frac{N+1}{K+2}), \quad \text{for} \quad N > M$$

$$= O(N \lg N).$$

Thus the time complexity is $O(N \lg N/p)$, which is optimal if $p \leq N$.

Section 3.3 A Probabilistic Quicksort Partitioning

Section 3.3.1 Probabilistic Partitioning Algorithm, Algorithm B

We give here an outline of the partitioning algorithm. The algorithm par-

titions in the same way as if done by a sequential quicksort that uses a median of a sample to choose a pivot. When called recursively, partitioning uses three variables, l and r the bounds of a subset of A, $a_l, ..., a_r$, of size $M \leq N$, and a subset of processors of size $q \leq p$. Initially we call partitioning(l, r, p).

procedure partitioning(l, r, q)

If $q = 1$ then partition using the sequential quicksort partition,

else $\{ q > 1 \}$ do steps 1) to 4).

1). Choose a small sample of size $m = \sqrt{M} = o(M)$ from the subset of A, $a_l, ..., a_r$. Find the median of this sample, which is used as the pivot V for partitioning. If $q \geq \sqrt{M}$ call Reischuk's probabilistic parallel selection (see section 3.1) which executes in constant time and costs $O(q)$; if $q < \sqrt{M}$, this task can be done by one processor in $O(m)$, and cost $o(M)$.

2). Distribute the pivot V (stored in a_l) to each processor. Consider the input set (starting from a_{l+1}), so that elements whose indices modulo q are equal are in the same subset $(S_i, i = 1, ..., q)$. Each processor partitions one subset, therefore the q processors can simultaneously work without starvation or concurrency problems. Each processor runs a sequential quicksort partition using the same pivot V. Each subset S_i can be considered as being split into two parts. The one on the left side contains the keys that are less than the pivot V, and the other on the right side keys greater than it. t_i and s_i $(s_i - t_i = q)$ are defined as before. (If $a_i < V$ for all i in a subset, the boundary s_i can be defined as equal to $t_i + q$, or if $a_i > V$ for all i in the subset, we define $t_i = s_i - q$.)

3). Define the *smin* and *tmax* as before, and $s = tmax - smin$. As long as $tmax > smin > 1$ we have partitioned A into three subsets. Obviously the one on the left side of *smin* has values that are less than the pivot, the one on the right side of *tmax* has values that are greater than the pivot, and the elements in the interval $[smin, tmax]$, if there is any, are mixed - there are some values greater than the pivot and some less. A swapping procedure is needed to partition the mixed part. The swapping procedure below first calculates the exact location of pivot V so that wrong -sided elements can be discovered and can be swapped.

4). We now call the partitioning procedure recursively on the two partitions created, proportioning the number of available processors. Let $M_1 = S - l, q_1 = M_1 q/M$. Call procedure partitioning$(l, S-1, q_1)$, and partitioning$(S+1, r, q-q_1)$ recursively.

The procedure swapping is as follows:

procedure Swapping

a). The q processors calculate the exact location S of the pivot in array A, where $S = l + \sum_{i=1}^{q} max(0, \lceil (t_i - l)/q \rceil)$, in $\lceil \lg q \rceil$ time. Then each of the processors compares its s_i and t_i with S.

Each processor has to move elements either to the right of V (eg $t_i > S$) or to the left of V (eg $s_i < S$), not both. Call a processor a *"right"* or a *"left"* processor.

b). Two auxiliary arrays are used to associate swapping the wrong-sided elements; a "*right array*", or a "*left array*". Two pointer sets, L_i and R_i $1 \leq i \leq q$,

are used to point to positions into which the next wrong sided element from set S_i moves. Each of the processors calculates two partial sums, one of them is $L_i = 1 + \sum_{j=1}^{i-1} max(0, \lceil \frac{S+1-a_j}{q} \rceil)$, and the other $R_i = 1 + \sum_{j=1}^{i-1} max(0, \lceil \frac{t_j-S}{q} \rceil)$, for $1 \leq i \leq q$, which takes $2 \lg q$ time in parallel. The right (left) processors remove their wrong-sized elements into right (left) array using its pointer R_i (L_i) simultaneously; after a barrier, they get the same amount of left (right) elements from the left array (right array) by using the same pointer R_i (L_i) in parallel in at most $2M/q$ time; after a barrier, processor p_1 swaps pivot V in a_l with a_S to move pivot into its exact location.

Section 3.3.2 Analysis of the time Complexity

3.3.2.1 Probability and Statistics Background

There are three statistical distributions important to this algorithm.

1). The **hypergeometric distribution** arises when there is a finite population of items of two types and sampling is done without replacement. (Guenther 1984)

Let there be N items in the set, k of which have some identifiable characteristic and $N-k$ which do not have this characteristic. Choose a sample of size n and let X be the number of items in the sample which have the identifiable characteristic. A drawing in which an item that has the characteristic are called a *success*, otherwise the drawing are called a *failure*. It is well known that if all possible draws are

equally likely then the probability of obtaining x successes in a sample of size n is

$$P_r(X = x) = h(x; N, k, n) = \frac{\binom{k}{x}\binom{N-k}{n-x}}{\binom{N}{n}}, \qquad x = 0, ..., k. \tag{3.5}$$

X is a hypergeometric random variable, and (3.5) is the hypergeometric density function. Symmetry directly shows that $h(x; N, k, n) = h(x; N, n, k)$. We use capital letters to denote cumulative distributions. Thus H denotes the cumulative distribution of h:

$$H(r) = \sum_{x=0}^{r} h(x; N, k, n). \tag{3.6}$$

The mean and variance of a hypergeometric distribution are

$$E(X) = np, \tag{3.7}$$

$$Var(X) = \sigma^2 = \frac{np(1-p)(N-n)}{N-1}, \tag{3.8}$$

where $p = k/N$. [Guenther, 1984]

When N is large, the hypergeometric distribution (1) can be approximated by a Normal distribution.

$$\begin{aligned} h(x; N, k, n) &\approx \phi((x - np)/\sigma)/\sigma \qquad \textit{or} \\ \sigma h(x; N, k, n) &\approx \phi((x - np)/\sigma). \end{aligned} \tag{3.9}$$

where ϕ denotes the density function of a Normal distribution.

2). The second distribution of interest is so closely related to the hypergeometric, that is called the **inverse hypergeometric distribution.**

For this distribution to occur we should have a situation identical to 1 above, except instead of a prespecified sample size, we sample until we obtain a predetermined number of successes; the random variable is the sample size that is needed.

Now let X be the number of drawings required to obtain exactly c successes. The form of this distribution is

$$h^*(x) = \frac{\binom{x-1}{c-1}\binom{N-x}{k-c}}{\binom{N}{k}}, \tag{3.10}$$

for $c \leq x \leq N - k + c$, and

$$E(X) = c\frac{N+1}{k+1} \tag{3.11}$$

[Guenther, 1984].

Use the notation

$$H^*(r) = \sum_{x=c}^{r} h^*(x), \tag{3.12}$$

equation (3.10) can be derived directly by observing that

$P_r(X = x)$ =P_r(c-1 successes are obtained in the first $x - 1$ draws)

$\times P_r$(a success occurs on the x^{th} draw, given that $c - 1$ successes have already been obtained).

This is

$$p_r(X = x) = \frac{\binom{k}{c-1}\binom{N-k}{x-c}}{\binom{N}{x-1}} \times \frac{k-c+1}{N-x+1} = h^*(x).$$

Assume $k = \sqrt{N}$. Then for any integer $f > 0$ Reischuk [1985] has shown that for the inverse hypergeometric the tails can be bounded thusly:

$$1 - H^*(\frac{N+1}{k+1}(c+f) - 1) \leq exp(-\frac{f^2}{4(k+1)} + O(\lg N)),$$

$$H^*(\frac{N+1}{k+1}(c-f)) \leq exp(-\frac{f^2}{4(k+1)} + O(\lg N)). \tag{3.13}$$

3). The third distribution of interest is the Sample Range. Let $X_1, ..., X_n$ be a random sample of size n from a normal distribution. Let $\phi(x)$ denotes the density function, and $\Phi(x)$ the CDF of Normal distribution. If these X's are arranged in ascending order of magnitude and written $X_{(1)} \leq X_{(2)} \leq ... \leq X_{(n)}$, the range W of a sample of size n is $X_{(n)} - X_{(1)}$. For a standard normal parent, the probability density function $g(w;n)$ and the cumulative density function $G(w;n)$ are given by David [1970] and Harter [1969a] as

$$g(w;n) = n(n-1)\int_{-\infty}^{\infty}[\Phi(x+w) - \Phi(x)]^{n-2}\phi(x)\phi(x+w)dx, w > 0,$$

$$G(w;n) = n\int_{-\infty}^{\infty}[\Phi(x+w) - \Phi(x)]^{n-1}\phi(x)dx,$$

and an approximation to the CDF $G(w;n)$ is

$$G(2y;n) \approx (2\Phi(y)-1)^n + 2na(2\Phi(y)-1)^{n-1} \times exp(-(1/2)y^2(1-a^2))(1-\Phi(ay)), \quad (3.14)$$

where $a^{-2} = 1 + (n-1)\phi(y)/(\Phi(y) - 1/2)$. (Cadwell 1954)

3.3.2.2 The Time Complexity of the Parallel Quicksort Algorithm

In step one of the partitioning algorithm, a sample of size $m = \sqrt{M}$ is drawn from the population, and the pivot V is the median of the sample. Let X be the rank of V in the population M. If $X = j$, then $\lfloor m/2 \rfloor$ elements which are less than V in the sample need to be chosen from the $j-1$ elements, that are less than V in the population, and the other $\lceil m/2 \rceil - 1$ elements in the sample come from the

other $M-j$ elements of the population. The probability of $X=j$, given V is the median of the sample of size m, has an inverse hypergeometric distribution (see eq(3.10)):

$$Pr(X=j)=h^*(j)=\frac{\binom{j-1}{\lfloor m/2\rfloor}\binom{M-j}{\lceil m/2\rceil-1}}{\binom{M}{m}}, \tag{3.15}$$

where $\lfloor m/2\rfloor+1\le j\le M-\lceil m/2\rceil+1$. The mean pivot location is $E(X)=(M+1)/2$. Let D denote the interval around the median $[E(X)-f\frac{M+1}{m+1}, E(X)+f\frac{M+1}{m+1}]$, where $f\equiv f(M)>0$ is an increasing function of M. From Reischuk's bounds (3.13) by replacing N by M, k by m, and $c(N+1)/(k+1)$ by $E(X)$, we get

$$1-H^*(E(X)+f\frac{M+1}{m+1}-1)\le exp(-\frac{f^2}{4(m+1)}+O(\lg M)),$$

$$H^*(E(X)-f\frac{M+1}{m+1})\le exp(-\frac{f^2}{4(m+1)}+O(\lg M)).$$

Adding these two inequalities and replacing $H^*(E(X)-f\frac{M+1}{m+1})-H^*(E(X)+f\frac{M+1}{m+1})$ by $Pr(X\in D)$, we get

$$Pr(X\in D)\ge 1-2exp(-f^2/(4(m+1))+O(\lg M)). \tag{3.16}$$

We choose $f(M)=M^{1/4+\alpha}$, where $\alpha>0$. Since $|D|=2f\frac{M+1}{m+1}$, the size of D, is bounded by $2M^{3/4+\alpha}$ $(o(M))$ with high probability by (3.16).

Let k_i be the number of elements in subset S_i not greater than the pivot V. If the rank of V is j, then $(k_1,k_2,...,k_q)$ satisfy a multivariate hypergeometric distribution. Namely for $0\le k_i\le n_i$, where n_i represents the size of S_i and $\sum k_i=j$, holds

$$Pr(k_1,k_2,...,k_q|X=j)=\frac{\binom{n_1}{k_1}\binom{n_2}{k_2}...\binom{k_q}{k_q}}{\binom{M}{j}}. \tag{3.17}$$

Let $p_i(k_i)$ be the marginal probability given $X = j$. Lemma 3.3 below shows that $p(k_i)$ is independent of i, i.e. the $k_i's$ are identically distributed random variables, although they are not statistically independent.

Lemma 3.3

$$p(k_i) = \frac{\binom{n_i}{k_i}\binom{M-n_i}{j-k_i}}{\binom{M}{j}}. \tag{3.18}$$

Proof. For $1 \leq i \leq q$, the marginal distribution is

$$p_i(k_i) = \sum_{k_1+\ldots+k_{i-1}+k_{i+1}+\ldots+k_q=j-k_i} \frac{\binom{n_1}{k_1}\binom{n_2}{k_2}\ldots\binom{n_q}{k_q}}{\binom{M}{j}},$$

reordering the binomial coefficients and multiplying both denominator and numerator by $\binom{M-n_i}{j-k_i}$,

$$\begin{aligned} p_i(k_i) &= \sum_{k_1+\ldots+k_m+\ldots+k_q=j-k_i} \frac{\binom{n_1}{k_1}\ldots\binom{n_{i-1}}{k_{i-1}}\binom{n_{i+1}}{k_{i+1}}\ldots\binom{n_q}{k_q}}{\binom{M-n_i}{j-k_i}} \times \frac{\binom{M-n_i}{j-k_i}\binom{n_i}{k_i}}{\binom{M}{j}} \\ &= \frac{\binom{M-n_i}{j-k_i}\binom{n_i}{k_i}}{\binom{M}{j}} = h(k_i; M, j, n_i), \end{aligned}$$

which follows since the last term in the sum is independent of the indexes and can be taken out of the summation and since the remaining terms sum to one. ∎

Let $n_i = M/q$ for all i then it follows from (3.7) and (3.8) that $E(k_i) = j/q$ and $Var(k_i) = \sigma^2 = (M/q)(j/M)(1 - j/M) \times (M(q-1)/q)/(M-1)$. Replace $(j/M)(1-j/M)$ by its maximum, 1/4, which occurs when $j = M/2$ for $1 \leq j \leq M$, and replace $(M(q-1)/q)/(M-1)$ by an upper bound of 1 if $M \geq q$. This results in the upper bound of σ^2 of $M/(4q)$, denoted by $\bar{\sigma}^2$.

Let $y_i = (k_i - j/q)/\sigma$. Assume we have q independent random variables y_i,

$y_1, ..., y_q$. Then the lower bound to the CDF of the range $W = y_{(q)} - y_{(1)}$ is shown below:

Lemma 3.4

$$G(w;q) \geq 1 - (4q/w) \times 1/\sqrt{2\pi} \times exp(-w^2/8) \tag{3.19}$$

Proof. For any normal random variable y,

$$\begin{aligned}\Phi(y) &= 1 - 1/\sqrt{2\pi} \int_y^\infty exp(-t^2/2)dt \\ &= 1 - 1/\sqrt{2\pi} \int_y^\infty (1/t)exp(-t^2/2)d(t^2/2) \\ &\geq 1 - \frac{1}{y\sqrt{2\pi}} \int_y^\infty exp(-t^2/2)d(t^2/2),\end{aligned}$$

we have

$$\Phi(y) \geq 1 - (1/y)\phi(y). \tag{3.20}$$

From (3.14) and (3.9),

$$\begin{aligned}G(w;q) &\approx (2\Phi(w/2) - 1)^q + 2qa(2\Phi(w/2) - 1)^{q-1} \\ &\quad \times exp(-(1/2)(w/2)^2(1 - a^2))(1 - \Phi(aw/2)) \\ &\geq (2\Phi(w/2) - 1)^q\end{aligned}$$

where $a^{-2} = 1 + (q-1)\phi(w/2)/(\Phi(w/2) - 1/2)$, which follows from omitting the non-negative terms when $w \geq 0$. We have, on substitution from (3.20),

$$\begin{aligned}G(w;q) &\geq (1 - (4/w)\phi(w/2))^q \\ &\geq 1 - (4q/w)\phi(w/2),\end{aligned}$$

which follows from a Taylor series expansion about the point 0. ∎

Let w be defined as $w = M^\beta$, then $G(M^\beta;q) \geq 1 - (4q/M^\beta)\phi(M^\beta/2) \geq 1 - o(1/M)$, for any $\beta > 0$. In our algorithm, k_i $i = 1, ..., q$ is hypergeometrically

distributed and can be approximately by a normal distribution for M large enough. The size of mixed part, s, can be denoted by W, the range of k_i, times q, i.e. $s = qW$. If denote s/q by s^*, when $s \leq q\bar{\sigma}M^{\beta}$,

$$Pr(s^* \leq \bar{\sigma}M^{\beta}) \geq G(M^{\beta}, q) \geq 1 - o(1/M). \tag{3.21}$$

Since we choose q based on M, i.e. $q = M^{\gamma}, 0 < \gamma < 1$, we can choose $\beta > 0$ to make $\beta + \gamma/2 < 1/2$ and $q\bar{\sigma}M^{\beta} = \sqrt{q}M^{1/2+\beta}/2 = o(M)$. Consequently the size of mixed part, is bounded by $o(M)$ with high probability. Thus this algorithm needs $o(M)$ more extra space in each level, and swaps $o(M)$ size od elements as extra work if compare with a sequential quicksort.

We consider the complexity of the algorithm. On the average, this probabilistic parallel quicksort with divide-and-conquer strategy sorts N items in time $O(N \lg N/p)$ for $p \leq N^{\gamma}$ any $0 < \gamma < 1$ on an MIMD machine with shared memory.

To see this, let us analyze that partitioning step by step. In the step one, a small sample of size $m = \sqrt{M}$ is chosen and a median of the sample is generated which costs $o(M)$ or $O(q)$. Since $p \leq N^{\gamma}$, from eq(3.2) we have $q \leq M^{\gamma}$, for $0 < \gamma < 1$, in any level. Therefore $O(q) \leq O(M^{\gamma}) = o(M)$, which leads the cost in step one is $o(M)$. In step 2, distributing the pivot V, the bound l and r, the number of processors assigned q to each processor takes $4 \lg q$ time, and the cost is $4q \lg q$. Then each of the processors examines every element in the subset in parallel, which costs $M + q$. In step 3, calculate S (the exact location of V), the partial sums of $L_i = 1 + \sum_{j=1}^{i-1} max(0, \lceil \frac{S+1-s_j}{q} \rceil)$, and $R_i = 1 + \sum_{j=1}^{i-1} max(0, \lceil \frac{t_j - S}{q} \rceil)$, which takes $3 \lg q$ time in parallel. The cost is $3q \lg q$, and the comparison of S with s_i

and t_i costs $2q$. To see the complexity of swapping, we can choose the threshold value as $q\bar{\sigma}M^{\beta}$, $\beta > 0$.

If $s > q\bar{\sigma}M^{\beta}$, the swapping assigns at most M/q elements to each processor, therefore the expense, $C(s > q\bar{\sigma}M^{\beta})$, is in $2M+3q$; otherwise since $q\bar{\sigma}M^{\beta} = o(M)$, for $q \leq M^{\gamma}, \gamma < 1$, and some $\beta > 0$ $(\beta+\gamma/2 < 1/2)$; Thus the cost, $C(s \leq q\bar{\sigma}M^{\beta})$, to swap the wrong sides elements, is $O(q\bar{\sigma}M^{\beta}) = o(M)$. We have, on substitution from (3.21), the average cost, $\bar{C}$, of this step is bounded by

$$\bar{C} = Pr(s \leq q\bar{\sigma}M^{\beta}) \times C(s \leq q\bar{\sigma}M^{\beta}) + Pr(s > q\bar{\sigma}M^{\beta}) \times C(s > q\bar{\sigma}M^{\beta})$$

$$= (1 - o(1/M)) \times o(M) + o(1/M) \times (2M + 3q) = o(M).$$

Thus the expense of the partitioning(l, r, q) is in $o(M) + 4q \lg q + M + q + 3q \lg q + 2q + o(M)$ time, that is equivalent to in $O(M)$ time with a small constant, for $p < N^{\gamma}$, $\gamma < 1$.

From (3.16), we can see the sample sort ensure the location j of the pivot V is within interval D of size $o(M)$, with high probability. It leads the sizes of two parts partitioned by V are approximately equal. Thus the total cost required to sort a set of size N is close to $N \lg N$ with high probability, which leads to the time complexity of our probabilistic quicksort algorithm, $O(N \lg N/p$, for $p < N^{\gamma}, \gamma < 1$ with probability $1 - o(1/N)$.

Theorem 3.2 For an input set of size N and P processors there are a probabilistic parallel quicksort algorithm that sorts the N elements in $O(N \lg N/P)$ time for any input on an MIMD machine with a shared memory which is optimal efficient algorithm, for $P = N^{\gamma}$, $0 < \gamma < 1$, in the view of the lower bound $\Omega(N \lg N)$ on

the sequential sort.

Section 3.3.3 Revisiting Algorithm A

Note that the size of mixed part, s, bounded by $o(M)$ is independent of the way to choose a pivot in section 3.3.2. Thus the analysis above can be used to analyze the average cost of the Algorithm A at every level.

In comparison of the average cost of the Algorithm A to the Algorithm B, Algorithm A costs $o(M)$ less than that of Algorithm B by without choosing median from a sample, which leads to the average cost for Algorithm A at any level of recursive call is bounded by $7q \lg q + M + 3q + o(M)$, i.e. in $O(M)$ time with a small constant, for $p < N^{\gamma}, \gamma < 1$.

The total cost to sort a set of size N is about $2N \lg N$; therefore the time complexity of Algorithm A on average case is $O((N \lg N)/p)$ with small constant, for $p < N^{\gamma}, \gamma < 1$.

Section 3.4 An in-place swapping procedure

We can have an in-place swapping procedure to clear up the "wrong- sided" elements.

Assume S (the exact location of Pivot V) has been calculated, and each of the processors has compared S with its s_i and t_i, so that the right or left processor

is defined. For distributing the same number of elements to be swapped to each processor, q processors calculate the total number of wrong-sided elements, U, by $U = \sum_{i=1}^{q}(max(0, \lceil \frac{S+1-s_i}{q} \rceil) + max(0, \lceil \frac{t_i-S}{q} \rceil))$, which takes $\lg q$ time. We represent U by $2tq - d$, where integers $t, d \geq 0$, and $d < 2q$, about $2t$ elements are assigned to each processor: t wrong-sided elements from one side and the same from the other side.

We maintain two stacks, A and B, to hold the pointers which point to the locations where the wrong-sided elements are stored. First of all, each of the right processors locks the stack B, pushes its t_j into B, unlocks it, and goes to an idle queue. Each of the left processors looks into stack B to find a job t_j. The processor, having gotten a job t_j, runs a getjob procedure in constant time as follows:

```
procedure getjob(s_i, t_j)
{u_i, u_j, u, and w are local variables. w is initialized to t}
begin
   u_i := ⌈(S + 1 − s_i)/q⌉
   u_j := ⌈(t_j − S)/q⌉
   u := min(u_i, u_j)
   if u > w then
      push s_i + w × q to stack A
      push t_j − w × q to stack B
      w := 0
   else if u_i ≤ u_j then
      push t_j − u_i × q to stack B
      w := w − u_i
   else
      push s_i + u_j × q to stack A
      w := w − u_j
   end if
end { getjob}.
```

The processor, having found the jobs, swaps the wrong-sided elements. When

the swapping terminates, the process goes back to the idle queue.

A processor in the idle queue goes to stack A recursively to find a task s_i. When a task is found, it looks into stack B to find a job t_j. The processor, having found both s_i and t_j, runs the procedure getjob (above) until its w gets down to 0 or no any task that is in the stacks. Then one processor swaps a_l with a_S to make the pivot V at its exact location. When every processor is back to the idle queue, it reaches the end of swapping.

Let us consider the time complexity of this swapping. The calculation of U takes $O(\lg q)$ time, to access the two job stacks for putting or getting jobs which are critical sections needs $O(q)$ time and the remaining part is swapped, which takes $O(M/q)$ time at most. The time spent is $O(\lg q + q + M/q)$.

Chapter 4 Summary, Conclusion, and Future Work

We have shown the design of two merging (or mergesort) algorithms. One merges two non-decreasing sequences A and B with size m and n respectively in $(m+n)/p + \lg((m+n)/p)/\lg 3 + 2\lg\lg p + 1$ time. This is faster than Valiant [1975]/Kruskal[1983]'s algorithm when the ratio of $m+n$, the number of the elements to be merged, over p, the number of processors available is greater than $(\lg p)^{1/3.4}$. It is optimal efficient algorithm when $p < (m+n)/\lg\lg(m+n)$. The other is an EREW algorithm which merges A and B in $O((m+n)/p + \lg(m+n))$ time. It removes the starvation, that occurs in Akl's algorithm. It is faster than Akl[1987]'s merging by a factor of $\lg p$.

We also have shown the design of several versions of a parallel quicksort algorithm which sorts N elements by p processors ($p \leq N\lg N$) in $O(N\lg N/p)$ time on a EREW SM-MIMD machine. It removes the starvation, which occurs at the beginning of the Deminet[1982], Quinn[1988], Evans and Dunbar[1982], and Evans and Yousif[1985]'s parallel quicksort algorithms. Our algorithm is optimal for $p \leq N/\lg N$; theirs is optimal for $p \leq \lg N$.

This thesis has demonstrated that merging, mergesort, and quicksort can be efficiently parallelized on a SM MIMD or SIMD machine by removing starvation. Future work can include examining other algorithms that suffer starvation. We believe these too can be improved. Also divide-and-conquer algorithms on computation models other than those studied here can be the subject of further research, for instance distributed memory systems.

Annotated Bibliography

1. Akl, S. G," An Optimal Algorithm for Parallel Selection", *Info. Proc. Let.* pp47-50, 19(1984).
   There are n items in array S, and n^{1-x} $(x > 0)$ processors which share common memory. Distribute n^x elements to each processor. n^{1-x} processors work simultaneously finding the median values of their own subarrays. Find the median, denoted by M, of medians. Use M as pivoting value to separate S into three parts: S_1 to contain the keys less than M, S_2 to contain the keys equal to M, and S_3 to contain the keys greater than M. If $k \leq |s_1|$ split S_1 as above, else if $k > |s_1| + |s_2|$, split S_3; otherwise return (k^{th} smallest number equals to the pivot value). The time complexity is $O(n^x)$. The total cost is O(n) which is optimal.

2. Akl, S. G, *Parallel Sorting Algorithms*, Academic Press, New York, 1985.
   Well written. An early book.

3. Akl, S. G, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, New Jersey, 1989.

4. Akl, S. G,"Adaptive and Optimal Parallel Algorithms for Enumerating permutations and Combinations", *The Computer Journal*, vol. 30, No.5, pp433-436, 1987.

5. Atallah, M.J., R. Cole, and M. Goodrich, "Cascading Divide-And-Conquer: A Technique for Designing Parallel Algorithms", *SIAM J. Comput.*, Vol.18, No.3, pp499-532, June 1989.

6. Baase, S, *Computer Algorithms : Introduction to Design and Algorithms*, Addison-Wesley, Reading Mass.,1988.

7. Babb II, R. G., *Programming Parallel Processors*, Addison Wesley, Publishing NY, 1988.

8. Bitton, D, Dewitt, D. J, Hsiao, D. K, and Menon, J," A Taxonomy of Parallel Sorting", *Computing Surveys*, vol. 16, NO. 3, pp287-318, Sept. 1984.

9. Borodin, A and Hopcroft, J. E," Routing, Merging, and Sorting on Parallel Models of Computation", *J. of Computer and System Sciences*, vol.30, pp130-145,1985.
   A variety of models are proposed for the study of synchronous parallel computation. Two classes of models are recognized, fixed connection networks and models based on a shared memory. Routing can be viewed as

a special case of sorting, and the existence of an $O(\log(n))$ sorting algorithm for an n processor fixed connection network. If the more powerful class of shared memory models is considered then it is possible to achieve an $O(\log(n)\log\log(n))$ sort.

10. Brawer, S, *Parallel Programming*, Academic Press, New York, 1989.

11. Bui, T. D and Thanh, M,"Significant Improvements to the Ford-Johnson Algorithm for Sorting",*Bit*, vol.25,pp70- 75,1985.

12. Carnevali, P," Timing Results of Some Internal Sorting Algorithms on the IBM 3090", *Parallel Computing*, 6,pp115-117, 1988.

13. Chin, F. and Ting, H. F.," An Improved Algorithm for Finding the Median Distributively", *Algorithmica*, 2, pp235- 249, 1987.

    Given two processes, each having a total ordered set of n elements, we present a distributed algorithm for finding median of these 2n elements using no more than $\log(n) + O(\sqrt{\log(n)})$ messages, but if the elements are distinct, only $\log(n)+O(1)$ messages will be required. The communication complexity of this algorithm is better than the previously known result (see Rodeh (1982)) which takes $2\log(n)$ messages.

14. Cole, R,"Slowing Down Sorting Networks to Obtain Faster Sorting Algorithms", *J. of the ACM*, vol.34, No.1, pp200-208, Jan. 1987.

    This paper provides a general method that trims a factor of $O(\log(n))$ time (or more) for many applications that can make use of this technique.

15. Cole, R and Yap, C. K," A Parallel Median Algorithm", *Information Processing Letters*, vol. 20, pp137- 139, 1985.

    A deterministic algorithm got finding the k^{th} smallest item in a set of n items, running in $O((\log\log(n))^2)$ parallel time on $O(n)$ processors in Valiant's comparison model.

16. Cole, R. J," An Optimally Efficient Selection Algorithm", *Information Processing Letters*, vol. 26,pp295- 299, Jan. 1988.

    An optimally efficient parallel algorithm for selection on the EREW PRAM is given. It requires a linear number of operations and $O(\lg n \lg n)$ time. A modification of the algorithm runs on the CRCW PRAM. It requires a linear number of operations and $O(\lg n \lg n/\lg\lg n)$ time.

17. Cole, R," Parallel Merge Sort", *SIAM J. Comput.*, vol 17, No. 4 pp770-785, Aug. 1988.

A parallel implementation of merge sort on a CREW PRAM that uses n processors and $O(\lg n)$ time is presented. This is a natural tree-based merge sort. To obtain an $O(\log(n))$ time sorting algorithm, the merge at the different levels of the tree is pipelined.

18. Cooper, J. and Akl, S. G," Efficient Selection on a Binary Tree", *Info. Proc. Let.* pp123-126, 23(1986).

    There are n elements and $2N-1$ processors structured in a binary tree with N leaves. n/N elements are assigned to each leave. k^{th} smallest element is to be selected.

    In i^{th} cycle the leaves send i^{th} significant bits to their parents who add the values of bits together, then send the sum up, and the parents do the same procedure until reaching the root. This takes ln N steps. At the root, compare the k value with n-sum. If $k \leq n - sum$, switch out all of elements whose i^{th} bits have 1 value, otherwise switch out all of elements with 0 value in i^{th} bits and assign k-sum to k. So the time complexity is $O(b\log(N))$ for adding the sum in the tree. For each processor the time for adding i^{th} bits b times is $O(bn/N)$. So the time complexity is $O(bn/N)$. The cost is $O(bn)$ which is optimal.

19. Dam, W.B.V, Frank, J.B.G, and Kan, A.H.G,"The Asymptotic Behavior of a Distributive Sorting Method", *Computing*, vol.31, pp287-303, 1983.

    In the distributive sorting method of Dobosiewicz, both the interval between the minimum and the median of the numbers to be sorted and the interval between the median and maximum are partitioned into n/2 subintervals of equal length; the procedure is then applied recursively on each subinterval containing more than three numbers. This paper refines and extends previous analyses of this method, e.g., by establishing its asymptotic linear behavior under various probabilistic assumptions.

20. Dromey, R. G," An Algorithm for The Selection Problem", *Soft.-Pract. and Exper.*, vol.16(11), pp981-986, Nov.1986.

    This algorithm improves the Hoare's Selection Algorithm. The point is to stop the comparison - exchange when $i(j)$ passes k and use $X(k)$ value as pivoting value X. If $i > k$, assign j to r, if $j > k$, assign i to l, then start again.

21. Evans, D. J. and Dunbar, R. C.," The Parallel Quicksort Algorithm Part 1-Run Time Analysis", *Intern. J. Comp. Math.*, vol 12, pp19-55,1982.

    In this paper a general purpose sorting algorithm is produced which is suitable for execution on an MIMD computer. The algorithm which is based on quicksort does not require a fixed number of processors but may theoretically use as many processors as are available. The analysis of the algorithm reveals that there is a maximum number of processors

that can be used for a particular size of set S_n. Binary tree structure is used to partition the data. This algorithm assigns the parts of data to each available processor which splits the part of data into two subsets by a pivot using median of three simultaneously, i.e. one subset contains the elements less than the pivot, the other greater than it, and assign the pivot at correct location. For each subsets of data repeat the same procedure or change to insertion sort for small size data.

Average time spent for sorting n elements is

$$\bar{t} = 264/7(n+1)H_{n+1} - 150 + (n+1)\{16432/245+ 2140/7/(m+2) - 264/7 \times H_{m+2} - 456/7/(m+2)H_{m+1}+ 192/35m - 2544/(7m(m+1)(M+2))\}.$$

22. Evans, D. J and Dunbar, R. C," The Parallel Quicksort Algorithm Part 2-Simulation", *Inter. J. Computer Math.*, vol.12, pp125-133, 1982.

    This paper is the second of a two part series in which a general purpose sorting algorithm ie. Quicksort is adapted for execution on an MIMD computer system. In this part, the parallel algorithm derived in Part 1 was simulated and qualitative agreement with the results from the run-time analysis was obtained.

23. Evans, D. J. and Yousif, N. Y,"Analysis of the performance of the Parallel Quicksort Method", *Bit*, 25, pp106-112, 1985.

    This is an analysis of performance of a parallel quicksort algorithm. They use a MIMD machine. Each processor works on a subset of data whenever the partition is done by quicksort. At the start only one processor is active, other processors are idle, then two work simultaneously, etc. They ran the program on a NEPTUNE system with different number of processors and different M values (M is number of element included in the subset to start using insertion sort instead of quicksort).

24. Floyd, R. W. and Rivest, R. L," Expected Time Bounds for Selection", *CACM*, vol.18, No.3, pp165-173, Mar. 1975.

    A selection algorithm is presented which is shown to be very efficient on the average, both theoretically and practically. The number of comparisons used to select the i^{th} smallest of n numbers is shown to be $n + \min(i, n-i) + O(n)$.

25. Frazer, W. D. and Mckellar, A. C.," Samplesort: A Sampling Approach to Minimal Storage Tree Sorting", *JACM*, vol.17, No.3, pp496-507, 1970.

26. Frederickson, G. N,"Upper Bounds for Time-Space Trade-offs in Sorting and Selection", *J. of Computer and System Sciences*, vol.34,pp19-26,1987.

27. Frank, W. B. V. D and Kan, A. H. G. R," The Asymptotic Behavior of a Distributive Sorting Method", *Computing*, 31,pp287-303, 1983.

28. Gibbons, A and W. Rytter, *Efficient Parallel Algorithms*, Cambridge, New York, 1989.

29. Gupta, P and Bhattacharjee, G.P," A Parallel Selection Algorithm", *Bit*, vol. 24, pp274-287, 1984.

    The problem of selecting the k^{th} largest element of $\{x_i + y_j | x_i \in X$ and $y_j \in Y \vee i, j\}$ where X and Y are two arrays of n elements each, is considered. An algorithm requiring $O(\log k * \log n)$ units of time on a shared memory model of a parallel computer having $O(n^{1+1/\beta})$ processors is presented where β is a pre- assigned constant lying between 1 and 2.

30. Handley, C. C," An in Situ Distributive Sort", *Inf. Pro. let.*, 23, pp265-270, 1986.

31. Heide, F. M. A. D and Wigderson, A,"The Complexity of Parallel Sorting", *SIAM, J. Comput.*, vol.16,No1,pp100- 107,1987.

32. Heidelberger, P, A. Norton, J. T. Robinson, "Parallel Quicksort using Fetch-and- Add", *IBM Research Center*, RC 12576, Mar 1987.

33. Hillis, W. D and Steele Jr, G. L," Data Parallel Algorithms", *CACM*, vol.29, No.12, pp1170-1183, Dec. 1986.

34. Hirschberg, D. S," Fast Parallel Sorting Algorithms", *CACM*, vol.21, No.8, pp657-661, Aug. 1978.

    A parallel bucket-sort algorithm is presented that requires $O(\log(n))$ time and n processors.

35. Hoare, C. A. R," Communicating Sequential Processes", *CACM*, Vol.21, No.8, pp666-677, Aug. 1978.

36. Horowitz, E. and Zorat, A," Divide-and-Conquer for Parallel Processing", *IEEE Trans. Comput.*, vol c-32, No.6, pp582-585, June 1983.

    A clearly written overview article worthy of reading. The paper's main contribution is that a realistic model for divide and conquer based algorithms has been postulated. The divide and conquer computer is discussed. The efficiency of some algorithms is analyzed, taking into account

all relevant parameters of the model. The parallel version of divide and conquer algorithm is described below: Given a problem, divide it into k subproblems. Then make k divide and conquer concurrent recursive calls. When the k subproblems are solved, combine these solutions into a solution for the original problem.

37. Huang, B. C and Knuth, D. E," A One-way, Stackless Quicksort Algorithm", *Bit*, 26,pp127-130, 1986.

This note describes a sorting technique that is similar to the well-known "quicksort" method, but it is unidirectional and avoids recursion. The new approach, which assumes that the keys to be sorted are positive numbers, leads to a much shorter program.

38. Hyslop, G. A, and Lamagna, E. A," Performance of Distributive Partitioned Sort in A Demand Paging Environment", *Inf. Proc. Letters*, 25,pp61-64, 1987.

The performances of Distributive Partitioned Sort (DPS) and Quicksort are compared empirically in a demand paging environment. It is found that DPS requires an amount of real memory equal to approximately 40 to 50% in its image size in order to run faster than quicksort. The performance of DPS deteriorates rapidly in smaller partitions due to excessive page faulting, while that of quicksort remains fairly constant.

39. Incerpi, J and Sedgewick, R,"Improved Upper Bounds on Shellsort", *J. of Computer and System Sciences*, Vol.31, pp210-224, 1985.

The running time of Shellsort, with the number of passes restricted to $O(\log(n))$, was thought for some time to be $O(N^{3/2})$, due to general results of Pratt. Sedgewick recently gave an $O(N^{4/3})$ bound. This paper gives a approach to achieve $O(N^{1+\epsilon/\sqrt{\log N}})$, for any $\epsilon > 0$.

It uses a base sequence a_1, a_2, a_3,..., of relatively prime integers to construct a sequence

$$\begin{array}{llll} a_1 & a_1a_2 & a_1a_2a_3 & a_1a_2a_3a_4 \quad \ldots \\ & a_1a_3 & a_1a_2a_4 & a_1a_2a_3a_5 \quad \ldots \\ & & a_1a_3a_4 & a_1a_2a_4a_5 \quad \ldots \\ & & & a_1a_3a_4a_5 \quad \ldots \\ & & & \ldots \end{array}$$

The c^{th} column in the table is formed by starting with $\Pi_{1 \le i \le c} a_i$, then multiplying each element in the previous column by a_{c+1}.

40. Incerpi, J and Sedgewick, R," Practical Variations of Shellsort", *Inf. Proc. Lett.*, 26, pp37-43, Sept. 1987.

They examine variants of shellsort that perform limited work per pass, i.e., not necessarily sorting the subfiles. One variant developed is shown

to perform very well empirically and has potential as a practical sorting algorithm and as a possible sorting network.

41. Jajodia, S, Liu, J, and NG, P. A," A Scheme of Parallel Processing for MIMD Systems", *IEEE Trans. on Software Eng.*, vol SE-9, No.4, pp436-445, July 1983.

42. Janus, P. J. and Lamagna, E. A,"An Adaptive Method for Unknown Distributions in Distributive Partitioned Sorting", *IEEE Trans on Comp.*, Vol c-34, 4, pp367-372, April 1985.

    An adaptation of DPS, which estimates the cumulative distribution function of the input data from a randomly selected sample, is developed and tested. The method runs only 2-4 percent slower than DPS in the uniform case, but outperforms DPS by 12-13 percent on exponentially distributed data for sufficiently large files. This algorithm is described below:

    First divide the range between the maximum and minimum into m equal length cells, and distribute the s samples into these cells. Then find the cumulative probabilities $p_1, ..., p_m$ of the m cells. Fit a line between each adjacent pair of cumulative probabilities, using $p_0 = 0$ and $p_m = 1$. Save the slope and y- intercept of each line. This yields an estimate of the CDF. Distribute each of the n items to be sorted by first determining to which sampling cell it belongs, say k, and then use k^{th} line equation to find the DPS bucket into which the item falls.

43. Knuth, D. E, *The Art of Computer Programming, vol.3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

44. Kruskal, C. P," Searching, Merging, and Sorting in Parallel Computation", *IEEE Trans. on Comp.*, vol. c-32, no.10, pp942-946, Oct. 1983.

45. Kumar, M. and Hirschberg, D. S, " An Efficient Implementation of Batcher's Odd-Even merge Algorithm and Its Application in Parallel Sorting Schemes", *IEEE Trans. on Comp.*, vol. c-32,no.3, pp254-264, Mar.1983.

    A efficient implementation of Batcher's Odd-Even merge algorithm including row merge algorithm, horizontal merge algorithm, and vertical merge algorithm on a mesh connected processor array and a main algorithm- S algorithm in $n \times n$ mesh connected processor array.

46. Kwan, S. C and Baer, J. L,"The I/O Performance of Multiway Mergesort and Tag Sort",*IEEE Trans. on Comp*, vol. c-34, No4, pp383-386, April 1985.

47. Lai, T. H and Sprague, A," A Note on Anomalies in Parallel Branch-and-Bound Algorithms with One-to One Bounding Functions", *Inf. Proc. Lett.*, vol.23,pp119-122, 1986.

48. Linial, N,"The Information-Theoretic Bound is Good for Merging", *SIAM J. Comput.*, vol.13, No.4, pp795-801, Nov. 1984.

49. Lipovski, G. J. and Malek, M, *Parallel Computing: Theory and Comparisons*, Wiley-Interscience, New York, 1987.

50. Martel, C.U. and D. Gusfield, "A Fast Parallel Quicksort Algorithm ", *Inf. Proc. Lett.*, Vol.30, No.2, pp97- 102, Jan. 1989.

51. Mikkilineni, K.P and Su, S.Y.W,"An Evaluation of Sorting Algorithms for Common-Bus Local Networks", *J. of Parallel and Distributed Comput.*, vol.5, pp59-81, 1988.

52. Megiddo, N," Applying Parallel Computation Algorithms in the Design of Serial Algorithms", *JACM*, vol.30, No.4, pp852-865, Oct. 1983.

53. Merritt, S. M," An Inverted Taxonomy of Sorting Algorithms", *CACM*, vol 28, No.1, pp96-99, Jan. 1985.

    An alternative taxonomy (to that of Knuth and others) of sorting algorithms is proposed. It emerges naturally out of a top-down approach to the derivation of sorting algorithms. Work done in automatic program synthesis has produced interesting results about sorting algorithms that suggest this approach.

54. Modi, J and Prager, L,"Implementation of Bubble Sort and the Odd-even Transposition Sort on a Rack of Transputers", *Parallel Comput.*, 4,pp345-348, 1987.

    In this paper they discuss implementation of bubble sort and its variant the odd-even transposition sort in a network of transputers, using the OCCAM language.

55. Moller-Nielsen, P and Staustrup, J," Problem-heap: A Paradigm for Multiprocessor Algorithms", *Parallel Comput.*, 4,pp63-74, 1987.

    The problem-heap paradigm has evolved through four years of experiments with Multi-Maren multiprocessor. Problem-heap algorithms have been formulated for a number of different tasks such as numerical problems, sorting, searching and optimization. Although these tasks are very different, the analyses of the running times of all the problem- heap algorithms are very similar. The problem-heap paradigm is illustrated by algorithms which have been implemented and analyzed using the Multi-Maren multiprocessor.

56. Mou, Z and P Hudak" An Algebraic Model for Divide- and-Conquer and Its

Parallelism", *J. Supercomp*, vol.2, pp257-278, 1988.

57. Motoki, T," A Note on Upper Bounds for the Selection Problem", *Inf. Proc. Lett.* vol.15, No.5, pp214-219, Dec.1982.

58. Nassimi, D and Sahni, S," Bitonic Sort on a Mesh- Connected Parallel Computer", *IEEE Trans. on Comp.*, C- 27, no.1,pp2-7, Jan. 1979.

    A efficient implementation of Bitonic Sort algorithm including row merge algorithm, two column merge algorithm, vertical merge algorithm, horizontal merge algorithm, and an application in parallel sorting main algorithm in $n \times n$ mesh connected processor array.

59. Negri, M. and Pelagatti, G," Join During Merge: An Improved Sort Based Algorithm", *Inf. Pro. Lett.*, 21, pp11- 16, 1985.

60. Noga, M.T. and Allison, D.C.S,"Sorting in Linear Expected Time",*Bit*, vol.25, pp451-465,1985.

    A sorting algorithm, Double Distributive Partitioning, is introduced and compared against Sedgwick's quicksort. It is shown that the DDP algorithm runs in $O(n)$ time for many distributions of keys. Furthermore, the combined number of comparisons, additions, and assignments required to sort by the new method on these distributions is less than quicksort. The DDP algorithm is similar to an adaptive method for unknown distributions in DPS by Janus and Lamagna (1985). (See Noga(1987) for a parallel version of this algorithm)

61. Noga, M. T," Sorting in Parallel by Double Distributive Partitioning ", *Bit* ,vol.27, pp340-348, 1987.

    A parallel version of the double distributive partitioning sorting algorithm of Noga and Allison (1985) is described, and two versions of this algorithm are compared. A "reclaimer" version, in which child processes completely build their page tables before starting useful work, exhibits $O(n/p)$ expected-case time complexity for a wide class of distributions. But the worst case performance is $O(n^2)$.

62. Owens, R.M and Ja'ja' J,"Parallel Sorting with Serial Memories", *IEEE Trans. on Comp.*, C-34, No4,pp379- 383, 1985.

63. Perrott, R. H, *Parallel Programming*, Addison- Wesley, New York, 1987.

64. Pippenger, N," Sorting and Selecting in Rounds", *SIAM J. Comput.*, Vol.16, No.6, pp1032-1038, Dec. 1987.

65. Preparata, F. P," New Parallel-Sorting Schemes", *IEEE Trans. on Comp.*, vol c-27, pp669-673, July 1978.

66. Quinn, M. J, *Designing Efficient Algorithms for Parallel Computers*, McGraw -Hill, New York, 1987.

67. Quinn, M. J," Parallel Sorting Algorithms for Tightly Coupled Multiprocessors", *Parallel Comput.*, 6, pp349-357, 1988.

    Three parallel sorting algorithms suitable for implementation on tightly coupled multiprocessors are presented and their performance on the Denelcor HEP are compared. Two of the algorithms parallel Shellsort and quickmerge are new. Shellsort is amenable to parallelization; however, since Shellsort has a higher complexity than quicksort, parallel Shellsort is inferior to parallel quicksort. A second new parallel algorithm, called quickmerge, is based upon both quicksort and mergesort. The implementation of quickmerge achieves significantly higher speedup than the implementation of parallel quicksort.

68. Reischuk, R.," Probabilistic Parallel Algorithms for Sorting and Selection", *SIAM J. Comput.*, vol.14, No.2, pp396-409, May 1985.

    Probabilistic parallel algorithms are described to sort n keys and to select the k-smallest element among them. For each problem they construct a probabilistic parallel decision tree. The tree selection finishes with high probability in constant time and the sorting tree in time $O(\log(n))$.

69. Rodeh, M," Finding the Median Distributively", *J Comp. and System Sci.* 24, pp162-166 (1982).

    To find the median of numbers which are in two equal size bags using two processors. The median of each bag is first found individually, then communicate this value between the two processors. The size of data can be reduced to half because half of data is smaller (greater) than the median. Do the program recursively, until the bags are empty.

    The time and space complexity are linear while the communication complexity is $2 \log n$, which is lower bound.

70. Ronsch, W and Strauss, H," Timing Results of Some Internal Sorting Algorithms on Vector Computers", *Parallel Comput.*, 4, pp49-61,1987.

    Seven internal methods for sorting a set $(a_1, a_2, ..., a_n)$ of real numbers into non-descending order are compared with regard to their performance on the vector computers CRAY-1S, CRAY-1M, CRAY X-MP,AMDAHL 1100, AMDAHL 1200 and the AMDAHL 470/V7. The algorithms considered are: Bubble sort, odd- even transposition sort, Batcher's parallel merge-exchange sort, heapsort, quicksort, vector quicksort and diamond

sort. Moreover, certain variants of some of these algorithms are also considered. The suitability of the algorithms with respect to vector machine implementation is discussed and the FORTRAN Cray codes for Batcher's parallel merge-exchange sort as well as Diamond sort are given.

71. Rotem, D, Santoro, N, and Sidney, J. B," Distributed Sorting", *IEEE Trans. on Comp.*, vol. c-34,No.4, pp372-376, April 1985.

72. Santoro, N, Scheutzow, M, and Sidney J.B,"On the Expected Complexity of Distributed Selection", *J. of Parallel and Distributed Comput.*, vol.5, pp194-203, 1988.

    In this paper, the expected communication complexity of the distributed selection problem is investigated; i.e. the problem of selecting the k^{th} smallest element in a file of N records distributed among d sites of a communication network. Letting $\delta = Min(k, n+1-k)$, it is shown that $O(d(\log(\delta)+\log d)$ and $O(\log(\delta)+\log d)$ communication actively, improving the existing bounds. These results do not rely on any assumption on the distribution of the file elements among the network sites.

73. Salehmohamed, M., Luk, W. S., and Peters, J. G," Performance Evaluation of LAN Sorting Algorithms", *ACM Proceeding*, x/87/0005, pp226-233, 1987.

    They adapt several parallel sorting algorithms and distributed sorting algorithms for implementation on an Ethernet network with diskless Sun workstations.

74. Sedgewick, R,"A New Upper Bound for Shellsort", *J. of Algorithms*, vol.7, pp159 -173, 1986.

    A direct relationship between Shellsort and the classical "problem of Frobenius" from additive number theory is used to derive a sequence of $O(ln(n))$ increments for Shellsort for which the worst case running time is $O(N^{4/3})$ for increments

    1, 8, 23, 77, 281, 1073, 4193, 16577,...,$4^{j+1} + 3 \times 2^j + 1$,... or
    1, 5, 65, 377, 1769,...,$2 \times 4^j - 9 \times 2^j + 9$,....

75. Shiloach, Y," Finding the Maximum, Merging, and Sorting in a Parallel Computation Model", *J. of Algorithms*, Vol.2,pp88-102, 1981.

    A merging algorithm for the case m=n=p (X size m, Y size n, and p processors) is given. The complexity is $O(\log(n)+1)$.

    (1) Processor i finds by a binary search (O(log(n)) the smallest y_j such that $x_i < y_j$ and then performs $z_{i+j-1} \leftarrow x_i$. If there is no such y_j it performs $z_{n+i} \leftarrow x_i$.

    (2) Processor i finds the smallest x_j such that $y_i < x_j$ and then sets $z_{i+j-1} \leftarrow y_i$. If there is no such x_j it sets $z_{m+i} \leftarrow y_i$.

76. Shrira, F, Francez, N, and Rodeh, M," Distributed k- Selection: From a Sequential to a Distributed Algorithm", *ACM Proceeding*, 5/83 /008, pp143-153, 1983.

A methodology for transforming sequential recursive algorithms to distributive ones is suggested. The assumption is that the program segments between recursive calls have a distributive implementation. The methodology is applied to two k-selection algorithms and yields new distributed k-selection algorithms. Some complexity issues of the resulting algorithms are discussed.

77. Tarjan, R. E," Amortized Computational Complexity", *SIAM J. Algorithm Disc. Math.*, vol.6, No.2, pp306-318, April 1985.

78. Tang, C. Y. and Lee, R. C. T," Optimal Speeding Up of Parallel Algorithms Based upon the Divide-and-Conquer Strategy", *Inf. Sci.*, 32, pp173-186, 1984.

How to design parallel algorithms based upon the divide-and-conquer strategy is discussed. If the data does not split too thoroughly, a more cost-effective parallel algorithm may be obtained.

79. Valiant, L. G. " Parallelism in Comparison Problems", *SIAM J. Comput.*, Vol.4 No3, pp348-355, Sept.1975.

A merging algorithm is given for two sorted lists, A and B, of length n,m ($n \leq m$ respectively using $k = \lfloor\sqrt{mn}\rfloor$ processors. First of all get sublists indexed $x_i, y_i, i = 1, 2, ...,$ of the two lists, where $x_i = i \times \lceil\sqrt{n}\rceil$ and $y_i = i \times \lceil\sqrt{m}\rceil$. The x_i sequence separate A into $\lfloor\sqrt{n}\rfloor$ equal sized segments $A_1, ..., A_{\lfloor\sqrt{n}\rfloor}$. Compare $a_{x_i} (i = 1, ..., \lfloor\sqrt{n}\rfloor)$ with every $b_{y_j}, j = 1, ..., \lfloor\sqrt{m}\rfloor$ to find out a interval of B a_{x_i} should belong in. Then compare a_{x_i} with every element in that interval of B to find out the exact location of a_{x_i} in B. Use a_{x_i} to separate Y into $\lfloor\sqrt{n}\rfloor$ sublists $B_1, ..., B_{\lfloor\sqrt{n}\rfloor}$. Finally do the same procedure on (A_i, B_i), i=1,2..., recursively. The time complexity is $MERGE^k(n, m) \leq 2 \log\log n + c, k = \lfloor\sqrt{mn}\rfloor$ and $1 \leq n \leq m$.

80. Wainwright, R. L," A Class of Sorting Algorithms Based on Quicksort", *CACM*, vol. 28, NO..4, pp396-402, April 1985.

Bsort, a variation of quicksort, combines the interchange technique used in bubble sort with the quicksort algorithm to improve the average behavior of quicksort and eliminate the worst case situation of $O(n^2)$ comparisons for sorted or nearly sorted lists. Bsort works best for nearly sorted lists or nearly sorted in reverse.

81. Walsh, T. R," How Evenly Should One Divide to Conquer Quickly?", *Inf. Pro. Lett.*, 19, pp203-208, 1984.

82. Warshauer, M. L,"Conway's Parallel Sorting Algorithm", *J. of Algorithm*, vol.7, pp270-276, 1986.

A parallel processor composed of (N- 1) finite state machines which is used to sort N keys was analyzed. In one cycle, comparisons and exchanges are made between pairs of adjacent keys. The keys is sorted after at most (2N-3) cycles.

83. Wegner, L. M," Sorting a Distributed File in a Network", *Comp. Net.*, 8, pp451-461, 1984.

This paper presents a new algorithm based on quicksort for sorting in place a distributed file in a message switching network.

84. Wegner,L. M," A Generalized, One-Way, Stackless Quicksort ", *Bit*, vol 27, pp44-48, 1987.

This note generalizes the one-way, stackless quicksort of Huang and Knuth to work for any type of sort key. It thus proves that quicksort can run with minimal space in $O(n \log(n))$ average time.

85. Winslow, L. E. and Chow, Y. C.," The Analysis and Design of Some New Sorting Machines", *IEEE Trans. on Comp.*, vol. c-32, No.7, pp677-683, July 1983.

86. Yousif, N. Y. and Evans, D. J.," Parallel Distributive Partitioned Sorting Methods", *Inter. J. Computer Math*, vol. 15, pp231-254, 1984.

In this paper, the distributive partitioned sorting method is developed into parallel forms suitable for use on a parallel computer. The algorithms are analyzed and implemented on the Loughborough University NEPTUNE parallel system and shown to be competitive with a parallel quicksort algorithm.

They examine two versions of the DPS algorithm. The version one includes three parts: a part to find the minimum and Maximum elements of the N elements, a part to distribute the N elements into a number of buckets, and the final part is to sort these buckets by using the Quicksort. The version two includes two main parts, one part for the distribution and the sorting, the other part is to merge the subsets obtained from the first part.

87. Yousif, N. Y. and Evans, D. J., " The Parallel Odd- Even Merge Algorithm", *Intern. J. Comp. Math.*, vol.18,pp265- 273,1986.

This paper describes the implementation of the odd-even merge algorithm on a parallel MIMD computer and discusses its computational complexity. This algorithm is described below: After sorting M subsets

by the neighbor sort, the subsets become sorted within themselves but not amongst each other. Therefore, the merge split algorithm is carried out to complete the solution of the problem. The odd-even merge (Akl calls it merge split) algorithm merges these M subsets of N/M each in at most M steps, where the parallelism is introduced within each step. On average (that is worst case!), (2N/M-1) comparisons are required to merge two subsets of size (N/M) each. The total comparisons C of the sequential implementation of the merge algorithm on average case is

$$C = N(M-1) - 1/2 \times M(M-1).$$

When $M/2 < p$, the total complexity of all the merge steps becomes:

$$C_p \leq NM/P * (M-1)/(M-2) - M^2/(2p) * (M-1)/(M-2) + 2.$$

When M/2=p, it becomes:

$$C_p \leq N((2Mp - M - 2p)/(2p)/(p-1)) \\ -M^2/4/p + -M/2(M/2-1)/(p-1) + 2.$$

When $M/2 > p$, it is

$$C_p \leq N/p(M-1) - M^2/(2p) + M/(2p) + 2.$$

88. Yousif, N.Y. and Evans, D.J,"Merging by the Parallel Binary Search Algorithm", *Inter. J. Comp. Math.*, Vol.22, pp239-248, 1987.

A parallel binary search sorting algorithm is presented and its complexity is analyzed. Using a binary search the algorithm merges M sorted subsets. The subsets are sorted independently of each other by some suitable sequential algorithm. (They suggest bubble sort!) The merges are 2-way. Hence $\log(M)$ merge steps are necessary. The 2-way binary search merging is explained as follows:

Assume $A = (a_1, ..., a_m)$ and $B = (b_1, ..., b_n)$ are two sorted lists. First assign n to i and then does binary search of a_i in B to find the smallest j, such that $a_i < b_j$. Then move those b's whose ranks are greater than j and also a_i to a destination array, decrease i by 1, assign j to n, and repeat the previous procedure again until all the elements in A are inserted into B.

89. Zaks, S,"Optimal Distributed Algorithms for Sorting and Ranking", *IEEE Trans. on Comp.*, vol c-34, No4,pp376-379, 1985.