JOURNAL OF COMPUTER AND SYSTEM SCIENCES 30, 130–145 (1985)

Routing, Merging, and Sorting on Parallel Models of Computation*

A. BORODIN

Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, M5S 1A7

AND

J. E. HOPCROFT

Department of Computer Science, Cornell University, Ithaca, N.Y. 14853

Received June 13, 1983; revised July 15, 1984

A variety of models have been proposed for the study of synchronous parallel computation. These models are reviewed and some prototype problems are studied further. Two classes of models are recognized, fixed connection networks and models based on a shared memory. Routing and sorting are prototype problems for the networks; in particular, they provide the basis for simulating the more powerful shared memory models. It is shown that a simple but important class of deterministic strategies (oblivious routing) is necessarily inefficient with respect to worst case analysis. Routing can be viewed as a special case of sorting, and the existence of an $O(\log n)$ sorting algorithm for some n processor fixed connection network has only recently been established by Ajtai, Komlos, and Szemeredi ("15th ACM Sympos. on Theory of Comput.," Boston, Mass., 1983, pp. 1–9). If the more powerful class of shared memory models is considered then it is possible to simply achieve an $O(\log n \log\log n)$ sort via Valiant's parallel merging algorithm, which it is shown can be implemented on certain models. Within a spectrum of shared memory models, it is shown that $\log\log n$ is asymptotically optimal for n processors to merge two sorted lists containing n elements.

I. INTRODUCTION: WHAT IS A REASONABLE MODEL?

A number of relatively diverse problems are often referred to under the topic of "parallel computation." The viewpoint of this paper is that of a "tightly coupled," synchronized (by a global clock) collection of parallel processors, working together to solve a terminating computational problem. Such *parallel processors* already exist and are used to solve time-consuming problems in a wide variety of areas

* This research was supported in part by ONR contract N00014-76-C-0018 and NSF grant MCS-81-01220.

0022-0000/85 $3.00

including computational physics, weather forecasting, etc. The current state of hardware capabilities will facilitate the use of such parallel processors to many more applications as the speed and the number of processors that can be tightly coupled increases dramatically. (A very good introduction to the future promise of "highly parallel computing" can be found in the January, 1982 issue of *Computer*, published by the IEEE Computer Society.)

Within this viewpoint, Preparata and Viullemin [20] distinguish two broad categories. Namely, we can differentiate between those models that are based on a fixed connection network of processors and those that are based on the existence of global or shared memory. In the former case, we assume that only graph theoretically adjacent processors can communicate in a given step, and we usually assume that the network is reasonably sparse; as examples, consider the shuffle-exchange network (Stone [26]) and its development into the ultracomputer of Schwartz [22], the array or mesh connected processors such as the Illiac IV, the cube-connected cycles of Preparata and Viullemin [20], or the more basic d-dimensional hypercube studied in Valiant and Brebner [29]. As examples of models based on shared memories, there are the PRAC of Lev, Pippenger, and Valiant [18], the PRAM of Fortune and Wyllie [8], the unnamed parallel model of Shiloach and Vishkin [23], and the SIMDAG of Goldschlager [11]. Essentially these models differ in whether or not they allow fetch and write conflicts, and if allowed, how write conflicts are resolved.

From a hardware point of view, fixed connection models seem more reasonable and, indeed, the global memory-processor interconnection would probably be realized in practice by a fixed connection network (see Schwartz [22]). Furthermore, for a number of important problems (e.g., FFT, bitonic merge, etc.) either the shuffle-exchange or the cube-connected cycles provide optimal hosts for well-known algorithms. On the other hand, many problems require only infrequent and irregular processor communication, and in any case the shared memory framework seems to provide a more convenient environment for constructing algorithms. Finally, in defense of the PRAM, it is plausible to assume that some broadcast facilities could be made available.

The problem of sorting, and the related problem of routing are prototype problems, due both to their intrinsic significance and their role in processor communication. Since merging is a (the) key subroutine in many sorting strategies, we are interested in merging and sorting with respect to both the fixed connection and shared memory models. For many fixed connection networks ([20, 26, 29]) the complexity of merging has been resolved by the fundamental $\log n$ algorithms of Batcher (see Knuth [15] for a discussion of odd–even and bitonic merge). The lower bound in this regard is immediate because $\log n$ is the graph theoretic diameter. In this paper, we concentrate on routing in networks and the complexity of merging (with application to sorting) on shared memory machines.

II. Routing in Networks

The problem of routing packets in a network arises in a number of situations. Various routing protocols have been considered. For packet switching networks we are concerned with arbitrary interconnections. For these general networks, protocols such as forward state controllers have been studied. For applications in parallel computer architecture we are concenred with special networks such as a d-dimensional hypercube or a shuffle exchange interconnection. In this setting $O(\log n)$ global or centralized strategies and $O(\log^2 n)$ local or distributed strategies are known.

For our purposes, a *network* is a diagraph $\langle V, E\rangle$, where the set of nodes V are thought of as processors, and $(i, j) \in E$ denotes the ability of processor i to send one *packet* or message to processor j in a given time step. A packet is simply an $\langle$origin, destination$\rangle$ pair or, more generally, $\langle$origin, destination, bookkeeping information$\rangle$.

A set of packets are initially placed on their origins, and they must be *routed* in parallel to their destinations; bookkeeping information can be provided by any processor along the route traversed by the packet. The prototype question in this setting is that of full permutation routing, in which we must design a strategy for delivering n packets in an n node network, when π: {origins} $\rightarrow$ {destinations} is a permutation of the node labels $\{1, 2, ..., n\}$. Other routing problems, in particular partial routing, where π is 1–1 but there may be less than N packets, are also of immediate interest. Unless stated otherwise, a routing strategy will denote a solution to the full permutation routing problem.

There are three positive routing results which provide the motivation for this section.

(1) Batcher's (see Knuth [15]) $O(\log^2 n)$ sorting network algorithm (say, based on the bitonic merge) can be implemented as a routing strategy on various sparse networks, such as the shuffle exchange (Stone [26]), the d-dimensional cube (Valiant and Brebner [29]), or the cube connected cycles (CCC—Preparata and Vuillemin [20]). This constitutes a local or distributed strategy, in that each processor p decides locally on its next action using only the packets at p. Indeed, the algorithm can be implemented so that there is exactly one packet per processor throughout execution. We also note that Batcher's bound is a *worst case* bound, holding for any initial permutation of the packets; in fact, every initial permutation uses the same number of steps.

(2) Valiant and Brebner [29] construct an $O(\log n)$ *Monte Carlo* local routing strategy for the d-cube, where $d = \log n$. In a Monte Carlo strategy, processors can make random choices in deciding where to send a given packet. Valiant's strategy achieves $O(\log n)$ in the sense that for every permutation, with high probability (e.g., $\geqslant 1 - n^{-c}$) all packets will reach that destination in $\leqslant (1 + c)\log n$ steps. Valiant's analysis can also be used to show that for a *random* input placement, the "naive strategy" on the d-cube terminates in $O(\log n)$ steps

with high probability (here, the probability is on the space of possible inputs). The naive strategy is to order the dimensions of the cube, and then proceed to move packets along those dimensions (in order) in which the origin and destination differ. These initial results have been extended to the shuffle exchange model (Aleliunas [1]) and a variant CCC^+ of the CCC model (Upfal [27]), these latter networks having only constant degree as contrasted with the $\log n$ degree of the cube. Following this development, Reif and Valiant [21] are now able to achieve an $O(\log n)$ Monte Carlo *sorting* algorithm for the CCC^+.

(3) The Slepian–Benes–Clos permutation network (see Lev, Pippenger, and Valiant [18]) can be implemented as a global worst case $O(\log n)$ routing scheme (on any of the above-mentioned sparse networks). Here, a pair of processors at a given time step simulate a switch of the permutation network; as such, the actions of any processor depend on the entire permutation.

We note that each of these strategies can be modified to handle partial routing. (This is immediate except for the strategy derived from Batcher's algorithm.) In order to make the distinction between local and global more meaningful we must limit the amount of bookkeeping information (say to $O(\log n)$ bits) since otherwise we can funnel all the input permutation information to a distinguished processor which then broadcasts this information throughout the network. Recently, Ajtai, Komlos, and Szemeredi [2] have constructed an ingenious $O(\log n)$ sorting algorithm for the comparator network model (see Knuth [15]). Since the gates in each stage of the comparator network have constant degree, these gates can be collapsed to form an $O(\log n)$ degree network in our sense. At the moment, it is not clear whether or not this algorithm could be implemented (within the same $O(\log n)$ bound) on any of the specific fixed connection networks mentioned in this paper. Nor is it clear whether the implied constant could be made small enough to make the algorithm "more competitive." Yet, this result stands as a theoretical breakthrough, further challenging us to search for a "simple" and very efficient local routing (or sorting) method for specific networks of interest.

The Relation Between Fixed Connection and Global Memory Models

Before proceeding to discuss routing algorithms for fixed connection networks, we want to briefly relate such parallel machines with parallel models based on a global memory. Indeed, this relation is yet another motivation for the importance of the routing and sorting problems. The importance of the routing problem is also emphasized in the paper of Galil and Paul [10] who consider simulations between various parallel models. We mention only a few global memory models:

(1) PRAC (Lev, Pippenger, and Valiant)—Simultaneous read or write (of the same cell) is not allowed (also called EREW RAM).

(2) PRAM (Fortune and Wyllie)—Simultaneous fetches are allowed but no simultaneous writes (also called CREW RAM).

(3) WRAM—WRAM denotes a variety of models that allow simultaneous

reads and (certain) writes, but differ in how such write conflicts are to be resolved (also called CRCW RAM).

(a) (Shiloach and Vishkin) A simultaneous write is allowed only if all processors are trying to write the same thing, otherwise the computation is not legal.

(b) An arbitrary processor is allowed to write.

(c) (Goldschlager) The lowest numbered processor is allowed to write.

For the purpose of comparison with an n-node fixed connection network, we assume the global memory models have n processors. It is obvious that even the weakest of the above models, the PRAC, can efficiently simulate a fixed connection network by dedicating a memory location for each directed edge of the network. Conversely, Lev, Pippenger, and Valiant [18] observe that a fixed connection network capable of (partial) routing in $r(n)$ time, can simulate one step of a PRAC in time $O(r(n))$, where now n bounds both the number of processors and the number of global memory cells. Furthermore, if we assume that the network is capable of *sorting* in $r(n)$ time, then one can also simulate one step of a PRAM or WRAM in time $O(r(n))$. The idea for this simulation was sketched in a preliminary version of this paper (Borodin and Hopcroft [3]). A more comprehensive approach is given by Vishkin [30].

As can then be expected, sorting also provides a way to simulate a PRAM or WRAM by a PRAC without changing the processor or memory requirements. That is, if the original machine operates in time t using n processors and m global memory cells, then the simulating PRAC operates in time $O(t \cdot r(n))$ using the same number of processors and memory cells. A more straightforward $O(t \cdot \log n)$ time simulation can be achieved using a "tournament" but here we must increase the global memory by an $O(n)$ factor, although the number of processors does not increase.

By these observations, and by using the Ajtai *et al.* [2] or Batcher sort, one sees that all these models have time complexities within an $O(\log n)$ (resp. $O(\log^2 n)$) multiplicative factor. These observations then closely follow the spirit of Cook [5] who distinguishes between fixed connection models (e.g., circuits, alternating Turing machines) and variable connection models (e.g., SIMDAGS) in the context of developing a complexity theory for the parallel computation of Boolean functions. For all the models he considers, one has variable connection machine (time t) $\leqslant$ fixed connection machine (time t^2). We note that Cook [5] only considers models when the number of processors n is at most exponential in t. Without certain restrictions (e.g., see the precise definition of a SIMDAG in Goldschlager [11]), n can be arbitrarily large relative to t.

A Lower Bound for a Special Case

It turns out to be surprisingly difficult to analyze reasonable simple strategies. We are able to show, however, that a very simple class of strategies, including the naive

strategy, cannot work well in the worst case, this being the case for a wide class of networks. Specifically, we study *oblivious strategies*, where the route of any packet is completely determined by the ⟨origin, destination⟩ of the packet. Oblivious strategies are almost, but not quite, local by definition; we might still determine *when* a processor sends a packet along an edge by glocal means. By its very nature, oblivious strategies tend to be very easy to implement. We can further motivate the use of such strategies by calling attention to the processor–memory interconnection network of the NYU-Ultracomputer (see Gottlieb, Lubachevsky, and Rudolpf [12]) which requires an oblivious strategy for the memory arbitration scheme.

THEOREM 1. *In any n-node network having in-degree d, the time required in the worst case by any oblivious routing strategy is* $\Omega(\sqrt{n}/d^{3/2})$.

Proof. For motivation, first consider a more restricted class of protocols, namely those where the next step in the route of a packet depends only on its present location and its final destination. For this class the set of routes for a packet heading for a given destination forms a tree. To see this observe that at any vertex there is a unique edge that the packet will take. Following the sequence of edges from any vertex must lead to the final destination.

Now, for the given definition of oblivious routing, the packets heading for a given destination no longer form a tree. Instead, we have a directed, not necessarily acyclic, graph G with a distinguished vertex v_0 (i.e., the destination). Furthermore G has in-degree d, and there are n distinct (in terms of their origin) but not necessarily disjoint *routes* (=distinguished paths) in G terminating at v_0. Let us call such a graph a (d, n) *destination graph.* If u is the origin of a route that enters v, we will call u *an origin for* v.

The goal is to find a vertex through which many, say t, packets must traverse. Since at most d packets can enter this vertex at a given time, a delay of at least t/d would be necessitated.

If we superimpose the n destination graphs determined by the n possible destinations for packets, each vertex is on n graphs. In order to force a packet headed for a given destination to go through a specific vertex v, we must initially start the packet on a vertex that is an origin of v in the particular destination graph. But there may be only a small number of such origins of v and if many destination graphs have the same set there will not be enough descendents to start each packet at a distinct vertex. Hence we would like to have a vertex v with the property that v has many origins in many destination graphs. To this end, we need

LEMMA 1. *Let* $G=(V, E)$ *be a* (d, n) *destination graph and let* $T(d, k, n)$ *be the minimum number of vertices having* k $(k \geqslant 1)$ *or more origins in G. Then* $T(d, k, n) \geqslant n/(d(k+1)+1)$.

Proof. (The proof we use here is due to Paul Beam.). Let $S=\{v \mid v \in V$ and v has at least k origins$\}$. Let $s=$ the cardinality of S. Now a route must either originate in S or V-S. Hence n is less than or equal to s plus the number of routes originating in V-S. We wish to count the routes originating in V-S as they enter S

for the first time. Since the in-degree of G is d, for any node $v \in S$ there are at most d nodes u such that $(u, v) \in E$. Now any $u \in V\text{-}S$ can contribute at most $k-1$ routes to the count. Hence the number of routes originating in $V\text{-}S$ is at most $s \cdot d \cdot (k-1)$, since s bounds the number of entry points. Hence $n \leqslant s + s \cdot d \cdot (k-1)$ or $s \geqslant n/(d(k-1)+1)$. ∎

We are now ready to complete the proof of Theorem 1. In each destination graph mark those vertices that have at least k origins. Let $k = \sqrt{n/d}$. In the network assign a count to each vertex indicating the number of destination graphs in which the vertex is marked. Since at least n/dk vertices are marked in each graph, the sum of the counts must be at least n^2/dk. Therefore, the average count (over all vertices) is at least $n/dk = \sqrt{n/d}$ which implies there is at least one vertex v which has at least $\sqrt{n/d}$ origins in each of $\sqrt{n/d}$ destination graphs. For each of these destination graphs we can place the corresponding packet at some vertex of the network so that it will pass through vertex v. Thus $\sqrt{n/d}$ packets will go through v. Since v is of degree at most d, routing requires time at least equal to $\sqrt{n/d^3}$. ∎

The question of when there exists a corresponding upper depends on the particular structure of the network. One important parameter in addition to the degree is the diameter of the graph. Clearly if the diameter of the graph is n we cannot hope for a $O(\sqrt{n})$ algorithm. However, even for some $O(\log n)$ diameter graphs with degree 2 we cannot achieve an $O(\sqrt{n})$ algorithm since there may be an isthmus or other type of bottleneck. However, for many structures there are oblivious routing algorithms that achieve or approach this lower bound.

Lang [17] has given an $O(\sqrt{n})$ oblivious routing algorithm for the shuffle exchange network, which is then asymptotically optimal since this network has constant degree. For the $d = \log n$ dimensional hypercube, the "obvious" oblivious algorithm will route packets for any permutation in time $O(\sqrt{n})$. (Since the hypercube has degree $\log n$ this is a factor of $(\log n)^{3/2}$ from optimal.) Namely order the dimensions $x_1, x_2, \ldots, x_{\log n}$. At time 1 transmit any packet whose destination is on the opposite subcube determined by the x_1 dimension. This may result in as many as two packets being on a vertex. Next transmit packets across the x_2 dimension, etc. After $\frac{1}{2}\log n$ dimensions there may be as many as $\sqrt{n}$ packets at a vertex causing a delay of $\sqrt{n}$. In each subsequent dimension the maximum number of packets that can be on a vertex halves, with eventually each packet arriving at its destination. It may be possible to improve this bound to $\sqrt{n}/\log n$ if one partitions the $\sqrt{n}$ packets that could go through a vertex v into $\log n$ groups and routes them by different edges.

Another restriction on routing is minimality. A *minimal routing* scheme forbids transmitting along an edge if it increases the distance of the packet from its destination. Thus every packet must follow a shortest path. For minimal routing schemes it is an interesting open problem whether there is a local (or even a global) routing scheme that is $O(\log^r n)$ for any r. For networks such as the d-cube we know of no monotone scheme better than the oblivious strategy just given.

A primary question is whether there is a local routing algorithm for say the d-cube that is better than $O(\log^2 n)$. It is remarkably difficult to analyze some very simple strategies. In particular, does the following algorithm or some variant of it, route an arbitrary permutation in $O(\log n)$. Consider some vertex. At a given stage as many as d packets, where d is the dimension of the cube, will arrive. As many as possible will be sent closer to their destinations. The remaining packets will be shipped to vertices of distance one greater. Since packets are not allowed to build up at a vertex the intuitive effect is to enlarge bottlenecks to several vertices and hence to allow more packets to get to their destinations in a given time. Although experimentally the algorithm appears promising we have not been able to formally analyze its behavior.

III. Merging and Sorting on Shared Memory Models

A Hierarchy of Models

The shared memory models usually studied all possess a global memory, each *cell* of which can be read or written by any processor. For the purpose of constructing algorithms, one usually assumes a single instruction stream; that is, one program is executed by all processors. However, when the processor number itself is used to control the sequencing of steps, and some ability to synchronize control is introduced, then the effect is that of a multiple instruction stream. The processors are assumed to have some local memory and each processor can execute basic primitive operations such as $\leqslant, =, \neq$ comparisons and integer $+, -, \times, \div$ and $\lfloor\sqrt{\ }\rfloor$ arithmetic operations in a single step. The PRAC, PRAM, and WRAM models have already been noted in Section II.

Other variants are clearly possible. We are concerned with the merging and sorting problems of elements from an arbitrary linear order (i.e., the schematic or structured approach). In this context, a "most powerful" parallel model (analogous to the comparison tree for sequential computation) has been studied by Valiant [28]. The *parallel computation tree* idealizes k-processor parallelism by a 3^k-tree, where each node is labeled by a set of k $\{<, =, >\}$ comparisons and the branches are labeled by each of the 3^k possible outcomes. It should be clear that for the problems of concern, parallel computation trees can simulate any reasonable parallel model, and in particular, can simulate all of the aforementioned shared memory models.

Let M denote any of these models. We will be concerned with $T^M_{\text{merge}}(n, m, p)$ and $T^M_{\text{sort}}(n, p)$, the minimum number of parallel steps to merge two sorted lists of n and m elements (resp. to sort n arbitrary elements) using p processors. Typically, $n = m$ and p is $O(n)$ or $O(n \log^\alpha n)$. Clearly, for any problem we have

$$T^{\text{PRAC}} \geqslant T^{\text{PRAM}} \geqslant T^{\text{WRAM}}.$$

Our main contribution is to establish the following two theorems:

THEOREM 2. *Let M denote the parallel computation tree model. Then $T^M_{\text{merge}}(n, n, cn)$ is $\Omega(\log\log cn - \log\log c)$ and hence $T^M_{\text{merge}}(n, n, n\log^\alpha n)$ is $\Omega(\log\log n)$ for all α. (Haggkvist and Hell [14] have independently proved similar lower bounds for merging on parallel computation trees.)*

THEOREM 3. *$T^{\text{PRAM}}_{\text{merge}}(n, n, n)$ is $O(\log\log n)$.*

We use Valiant's algorithm, which already establishes the upper bound for the parallel comparison tree, but following Valiant [28], Preparata [19], and Shiloach and Vishkin [23], remark that a "processor allocation" problem must be solved to realize Valiant's algorithm on the PRAM model. Hence, the problem of merging is now resolved on all of the above-shared memory models except the PRAC, for which we cannot improve on the $\log n$ upper bound of the Batcher merge. For the PRAC, Snir [24] has recently established an $\Omega(\log n)$ lower bound for searching or insertion (and hence merging), allowing any number of processors. Hence, the asymptotic complexity of parallel merging is now well understood for all the above-mentioned models.

With regard to sorting, we have the following direct corollaries:

COROLLARY 1. *$T^{\text{PRAM}}(n, n)$ is $O(\log n \log\log n)$.*

COROLLARY 2. *$T^{\text{PRAM}}(n, n\log n)$ is $O(\log n)$.*

Clearly, Corollary 1 follows from a standard merge sort, whereas Corollary 2 is a restatement of Preparata's [19] result, which can now be stated for PRAM's using Theorem 2. Corollaries 1 and 2 should be compared with the Shiloach and Vishkin upper bound of $O(\log^2(n/\log(p/n)) + \log n)$ for sorting on their version of a WRAM with p processors. Whether or not an $O(\log n)$ sort can be achieved on any of the RAM models using only $O(n)$ processors would be resolved by the Ajtai, Komlos, and Szemeredi [2] result (assuming their algorithm can be implemented without loss of efficiency on these models). Without their result, Corollary 2 and Preparata's [19] $O(k\log n)$ sort using $n^{1+1/k}$ processors on a PRAC represent the best known $O(\log n)$ sorting algorithms for the RAM models.

With regard to lower bounds for sorting, Haggkvist and Hell [13] prove that in terms of the parallel computation tree, time less than or equal to k implies $\Omega(n^{1+1/k})$ processors are required (and this is essentially sufficient). It follows, that for the tree model and any of the RAM models, $\Omega(\log n/\log\log n)$ is a lower bound for sorting with $O(n\log^\alpha n)$ processors. For $O(n)$ processors, $\Omega(\log n)$ is a trivial lower bound resulting from the sequential lower bound of $\Omega(n\log n)$.

Cook and Dwork [6] show that $\Omega(\log n)$ steps on a PRAM are required for the Boolean function $x_1 \vee x_2 \vee \cdots \vee x_n$, no matter how many processors are available. It follows that $\Omega(\log n)$ steps on a PRAM are required for the MAX function and hence for sorting. However, it is possible to achieve $O(\log n)$ sorting on a WRAM. Indeed, it is possible to achieve $O(1)$ time by using an exponential

number of processors; e.g., use $n \cdot n!$ processors to check each of the possible permutations. A major open problem is to determine the minimal number of processors needed for an $O(1)$ time sort on a WRAM. Stockmeyer and Vishkin [25] have shown how to simulate a t time, p processor WRAM (in particular, the SIMDAG) by an unbounded fan-in AND/OR circuit having depth $O(t)$ and size polynomial (p). By this simulation and some appropriate reducibilities, Stockmeyer and Vishkin [25] are able to use the beautiful lower bound of Furst, Saxe, and Sipser [9] to show that a WRAM cannot sort in $O(1)$ time using only a polynomial (in n) number of processors. (See also Chandra, Stockmeyer, and Vishkin [4].)

A $\Omega(\log \log n)$ Lower Bound for Merging on Valiant's Model

Sequentially merging two lists of length n can be accomplished with $2n$-1 comparisons and this is provably optimal. Since only $2n$-1 comparisons are necessary to merge two such lists, conceivably in a parallel model they could be merged in time $O(1)$ with n processors. However, we shall show that this is not possible. Even allowing $n \log^{\alpha} n$ comparisons per step, a depth of $\log \log n$ is needed. The previously stated Theorem 2 will follow as an immediate corollary of Lemma 3, which we now motivate. Throughout Section III we use integer floor and integer ceiling only when crucial.

Consider the process of merging two sorted lists $a_1,..., a_n$ and $b_1,..., b_n$ with n processors. At the first step at most n comparisons can be made. Partition each list into $2\sqrt{n}$ blocks of length $\frac{1}{2}\sqrt{n}$. Form pairs of blocks, one from each list. There are $4n$ such pairs of blocks. Clearly there must be $3n$ pairs (A_i, B_j) of blocks such that no element from the block A_i is compared with any element from the block B_j. We shall show that we can select $\frac{1}{2}\sqrt{n}$ pairs of blocks

$$(A_{i_1}, B_{j_1}), (A_{i_2}, B_{j_2}),..., (A_{i_{(1/2)\sqrt{n}}}, B_{j_{(1/2)\sqrt{n}}})$$

such that $i_l < i_{l+1}$ and $j_l < j_{l+1}$ for $1 \leqslant l < \frac{1}{2}\sqrt{n}$. If the total order is such that all elements in $A_{i_l} \cup B_{j_l}$ are less than any element in $A_{i_{l+1}} \cup B_{j_{l+1}}$, $1 \leqslant l < \frac{1}{2}\sqrt{n}$, then after the first stage we are faced with $\frac{1}{2}\sqrt{n}$ subproblems each of size $\frac{1}{2}\sqrt{n}$.

At the second stage the n processors are partioned somehow among the $\frac{1}{2}\sqrt{n}$ subproblems. However this is done, at least one half of the subproblems have assigned to them fewer than twice the average available number of processors per subproblem. Thus there are $\frac{1}{4}\sqrt{n}$ subproblems with at most $4\sqrt{n}$ processors per problem. Intuitively this argument suggests that at each stage the size of subproblem goes down by a square root and hence $\log \log n$ time is necessary. These ideas will be made precise in the following lemmas.

In what follows let $G = (A \cup B, E)$ be a bipartite graph with $E \subseteq A \times B$. Further let $A_1, A_2,...,$ and $B_1, B_2,...,$ be fixed orderings of the vertices in A and B, respectively. A matching is said to be *compatible* if for each pair of edges (A_i, B_j) and (A_g, B_h) in the matching $i < g$ if and only if $j < h$.

LEMMA 2. *Let $G = (A \cup B, E)$ be a bipartite graph with $A = A_1, A_2,..., A_{2k}$ and*

$B = B_1, B_2, \ldots, B_{2k}$ and let $E \subseteq A \times B$ have $3k^2$ edges. Then G has a compatible matching of cardinality at least k.

Proof. Partition the edges into $2k$ blocks as follows. For $-k < b < k$ we have a block consisting of edges $\{(A_i, B_{i+b}) \mid 1 \leqslant i \leqslant 2k \text{ and } 1 \leqslant i+b \leqslant 2k\}$. In addition we have one block consisting of all other edges. At most k^2+2 edges fall into the block of other edges. Thus at least $2k^2-k$ edges must be partitioned into $2k-1$ blocks. Hence at least one block must have at least k edges and these edges form a compatible matching.

LEMMA 3. *Let $T(s, c)$ be the time necessary to solve k, $k \geqslant 1$, merging problems of size s with cks processors. Then $T(s, c)$ is $\Omega(\log(\log sc/\log c))$.*

Proof. On the average we can assign cs processors to each problem. At least one half of the problems can have no more than twice this number of processors assigned to them. That is, at least $k/2$ problems have at most $2cs$ processors.

Consider applying $2cs$ processors to a problem of size s. This means that in the first step we can make at most $2cs$ comparisons. Partition the lists into $2\sqrt{2cs}$ blocks each of size $\frac{1}{2}\sqrt{s/2c}$. There are $8cs$ pairs of blocks. Thus there must be $6cs$ pairs of blocks with no comparisons between elements of the blocks in a pair. Construct a bipartite graph whose vertices are the blocks from the two lists with an edge between two blocks if there are no comparisons between elements of the two blocks. Clearly there are $6cs$ edges and thus by the previous lemma there is a compatible match of size at least $\frac{1}{2}\sqrt{2cs}$. This means that there are at least $\frac{1}{2}\sqrt{2cs}$ problems each of size at least $\frac{1}{2}\sqrt{s/2c}$ that we must still solve. Thus $T(s, c) \geqslant 1 + T(\frac{1}{2}\sqrt{(s/2c)}, 4c)$.

We show by induction on s, that

$$T(s, c) \geqslant d \log \frac{\log sc}{\log c}$$

for some sufficiently small d.

$$\begin{aligned} T(s, c) &\geqslant 1 + d \log \frac{\log \frac{1}{2}\sqrt{(s/2c)}\, 4c}{\log 4c} \\ &\geqslant 1 + d \log \left(\frac{\log sc}{4 \log c}\right) \text{ (w.l.o.g. assume } c \geqslant 4) \\ &\geqslant 1 + d \log \frac{\log sc}{\log c}\, d \log 4 \\ &\geqslant d \log \frac{\log sc}{\log c} \end{aligned}$$

provided $d < \frac{1}{2}$. Observe that $\log(\log sc/\log c)$ is $\Omega(\log\log s - \log\log c)$ which matches Valiant's upper bound of $2(\log\log s - \log\log c)$.

An $O(\log \log n)$ Upper Bound for Merging on a PRAM

We recall Valiant's (n, m) merging algorithm which merges X and Y with $\#X = n$, $\#Y = m$, $n \leqslant m$ using $\sqrt{nm}$ processors. Our goal is to implement Valiant's algorithm on a PRAM. However, we shall require $n+m$ processors rather than $\sqrt{nm}$ as in Valiant, because of the data movement which need not be accounted for in the tree model. The algorithm (taken verbatim from Valiant [28]) proceeds as follows:

(a) Mark the elements of X that are subscripted by $i \cdot \lceil \sqrt{n} \rceil$ and those of Y subscripted by $i \cdot \lceil \sqrt{m} \rceil$ for $i = 1, 2, \ldots$. There are at most $\lfloor \sqrt{n} \rfloor$ and $\lfloor \sqrt{m} \rfloor$ of these, respectively. The sublists between successive marked elements and after the last marked element in each list we call *segments.*

(b) Compare each marked element of X with each marked element of Y. This requires no more than $\lfloor \sqrt{nm} \rfloor$ comparisons and can be done in unit time.

(c) The comparisons of (b) will decide for each marked element the segment of the other list into which it needs to be inserted. Now compare each marked element of X with every element of the segment of Y that has thus been found for it. This requires at most

$$\lfloor \sqrt{n} \rfloor \cdot (\lceil \sqrt{m} \rceil - 1) < \lfloor \sqrt{nm} \rfloor$$

comparisons altogether and can also be done in unit time.

On the completion of (a), (b), and (c) we can store each $x_{i\lceil \sqrt{n} \rceil}$ in its appropriate place in the output Z. It then remains to merge the disjoint pairs of sublists $(X_0, Y_0), (X_1, Y_1), \ldots$, where the X_i and Y_i are sequences of X and Y, respectively. Whereas Cauchy's inequality guarantees that there will be enough processors to carry out these independent merges by simultaneous recursive calls of the algorithm, it is not clear how to inform each processor to which (X_i, Y_i) subprogram (and in what capacity) it will be assigned. This is the main concern in what Shiloach and Vishkin [23] refer to as the *processor allocation problem.*

We desire a recursive procedure (in fact, a macro might be more appropriate) MERGE(i, n_i, j, m_j, k) which merges $x_i, x_{i+1}, \ldots, x_{i+n_i-1}$ and $y_j, \ldots, y_{j+m_j-1}$ into $z_{i+j-1}, \ldots, z_{i+j+n_i+m_j-2}$ using at most $n_i + m_j$ processors beginning at processor number p_k. Such a merge will be simultaneously invoked by processors $p_k, p_{k+1}, \ldots, p_{k+n_i+m_j-1}$. The initial call is MERGE$(1, n, 1, m, 1)$. As we enter this subroutine, a processor p_k will know from i, j, n_i, m_j, and k, the (relative) role it plays in parts (a), (b), and (c) of Valiant's algorithm. For example, say $n_i \leqslant m_j$ and let

$$l = k + i' \lceil \sqrt{m_j} \rceil + j', \qquad 0 \leqslant i' \leqslant \lfloor \sqrt{n_i} \rfloor - 1, \qquad 0 \leqslant j' \leqslant \lfloor \sqrt{m_j} \rfloor - 1,$$

then in step (a), processor p_l compares $x_{i'\sqrt{n_i+i}}$ and $y_{j'\sqrt{m_j+j}}$.

We will now indicate how processors reassign themselves before recursively invoking the merge routine. For simplicity, assume that we have just completed steps (a), (b), (c) of MERGE$(1, n, 1, m, 1)$. We can assume that we have deter-

TABLE J

0	j_1	j_2	...	$j_{\sqrt{n}-1}$	m

mined for each i, $0 \leqslant i \leqslant \lfloor \sqrt{n} \rfloor - 1$ that $y_{j_i} < x_{i\lceil \sqrt{n} \rceil} \leqslant y_{j_i+1}$ and that we have constructed a table J accessible by all processors. A given processor p must determine its role in the next iteration of the algorithm. We state the following without proof.

LEMMA 4. *Suppose* $(X_0, Y_0), \ldots, (X_{r-1}, Y_{r-1})$ *have been assigned processors, and* $X_{r-1} = (\ldots, x_{r\sqrt{n}-1})$ *and* $Y_{r-1} = (\ldots, y_{j_i})$. *There exists a function* ϕ *such that no more than* $\phi(r, n, j_r)$ *processors have been assigned. Indeed,* $\phi(r, n, f) = r\sqrt{n} + f$.

The impact of this Lemma is that we can safely assign processors $p_{\phi(r,n,j_r)+1}, \ldots, p_{\phi(r+1,n,j_{r+1})}$ to (X_r, Y_r). It remains for each processor to know to which (X_k, Y_k) it will be assigned. Indeed, once a processor knows to which (X_k, Y_k) it has been assigned, then it can obtain all the information it will need to invoke MERGE from Table J and the ϕ function; namely, X_k starts at $k\lceil \sqrt{n} \rceil + 1$ and has length $\lfloor \sqrt{n} \rfloor - 1$, Y_k starts at y_{i_k} and has length $j_{k+1} - j_k$, and the processors assigned to this task begin at $p_{\phi(k,n,j_k)+1}$.

The actual assignment of a processor to a (X_k, Y_k) subproblem proceeds in two stages (note that we cannot simply do a sequential binary search in J because this would require $\log \sqrt{n}$ steps):

Stage (1) Processors are assigned for those (X_k, Y_k) with $\# Y_k \leqslant m/\sqrt{n}$ (and hence no more than $\sqrt{n} + m/\sqrt{n} = (n+m)/\sqrt{n}$ processors need be assigned to this task since $\# X_i = \sqrt{n} - 1$ for all i.

Stage (2) Processors are assigned to the remaining (X_k, Y_k).

Stage 1. For each k, $0 \leqslant k \leqslant \sqrt{n} - 1$, we assign $(n+m)/\sqrt{n}$ processors to look at both the k and the $k+1$th entry of Table J. If $j_{k+1} - j_k \leqslant m/\sqrt{n}$, then these processors inform (by posting the information in an appropriate place of global memory) processors numbered $\phi(k, n, j_k) + 1, \ldots, \phi(k+1, n, j_{k+1})$ that they are assigned to (X_k, Y_k). We then wait until the completion of Stage 2 before invoking merge on (X_k, Y_k) since all processors are needed for Stage 2.

Stage 2. The processors are divided into $(n+m)/\sqrt{n}$ blocks, each block containing $\sqrt{n}$ processors. Each of the $\sqrt{n}$ processors in a block are trying to determine to which X_k, Y_k these $\sqrt{n}$ processors will be assigned. Let p_{j_1} be the first processor of block 1. The kth processor of block 1 looks at the j_k and j_{k+1} in Table J and determines (via the function ϕ) whether or not processor p_{j_1} would be

assigned to this subproblem. Now each processor p in the 1th block can determine (again via Table J and ϕ) which of the following hold:

(i) p is assigned to (X_k, Y_k), the subproblem assigned to p_{j_1}.

(ii) p is assigned to $(X_{k'}, Y_{k'})$, the subproblem assigned to $p_{j_{l+1}}$.

(iii) p has already been assigned in Stage 1.

We claim that if neither (i) and (ii) hold, then (iii) must hold since clearly less than $\sqrt{n} < (n+m)/\sqrt{n}$ processors have been assigned to the same task as p. ▮

Finally, we note that the base case ($n-1$, m arbitrary), where most of the data movement takes place, is easily performed with $m+1$ processors. This completes the informal description of the algorithm which we claim establishes Theorem 3.

Kruskal [16] has unified and somewhat improved Valiant's merging algorithms to yield the following upper bound: p processors can perform an (n, m) merge in time $(n+m)/p + O(\log\log n) + O(\log(n+m)/p))$. Letting $n=m$, it follows that $n/\log\log n$ processors are sufficient to achieve an (n, n) merge in time $O(\log\log n)$. Clearly, this is now asymptotically optimal since $2n$-1 comparisons are needed sequentially. Assuming this can be implemented on a PRAM, there will be some corresponding improvements in Corollaries 1 and 2. For example, Corollary 1 would become $T(n, n)$ is $O(\log n \log\log n/\log\log\log n)$.

IV. Conclusion

Our underlying goal is to better understand the theoretical capabilities and limitations inherent in large scale, general purpose, parallel computation. To this end, we have considered in this paper two of the major theoretical viewpoints of large scale parallel processing, namely fixed connection networks and shared memory models. We have not considered circuit models or Cook's [5] fundamental HMM model, nor have we dealt with the more specific consideration of various VLSI models. In fact, the HMM model is "conceptually" between the fixed connection and shared memory models, although its power relative to either the d-dimensional hypercube or the PRAC is not precisely known. (On the other hand, an HMM can be viewed as a PRAM with a restricted instruction set.) As a common point of reference for discussing these models, we have considered the routing, merging, and sorting problems. In this regard, the theoretical complexity of oblivious routing and merging has been essentially resolved for a wide spectrum of models. For many models, sorting is also very well understood.

We are, however, still fundamentally intrigued by the worst case analysis of simple routing strategies such as the one suggested in this paper. Other natural classes of routing strategies (e.g., minimal routing) also deserve further consideration. As for sorting, while the Ajtai, Komlos, and Szemeredi [2] result should resolve many questions, we are still concerned with the problem of $O(\log n)$, and in particular

$O(1)$, parallel time sorting on a WRAM. Further progress (beyond [9, 25]) on this problem will almost surely have to await a better understanding of the more basic circuit models. The general question of tradeoffs between parallel time and the number of processors (or "hardware" as referred to by Cook and Dymond [7]) is clearly a major theme in complexity theory.

ACKNOWLEDGMENTS

We are indebted to U. Vishkin for observing that our original claims about merging on a PRAM could not hold because of the amount of data movement required by our formulation of the problem. We thank P. Beam for his clear proof of Lemma 1, and we also thank R. Prager and L. Rudolph for many helpful comments.

REFERENCES

1. R. ALELIUNAS, Randomized parallel communications, *in* "Proc. of ACM Sympos. on Principles of Distributed Computing," Ottawa, Canada, 1982, pp. 60–72.
2. M. AJTAI, J. KOMLOS AND E. SZEMEREDI, An $O(n \log n)$ sorting network, *in* "15th Annual ACM Sympos. on Theory of Comput." Boston, Mass., 1983, pp. 1–9.
3. A. BORODIN AND J. HOPCROFT, Routing, merging, and sorting in parallel models of computation, *in* "Proc. 14th Annual ACM Sympos. on Theory of Comput." San Francisco, Calif., 1982, pp. 338–344.
4. A. K. CHANDRA, L. J. STOCKMEYER, AND U. VISHKIN, Complexity theory for unbounded fan-in parallelism, *in* "Proc. 23rd Annual IEEE Sympos. on Found. of Comput. Sci.," 1982, pp. 1–13.
5. S. A. COOK, Towards a complexity theory of synchronous parallel computation, *Enseign. Math. (2)* **27** (1981), pp. 99–124.
6. S. A. COOK AND C. DWORK, Bounds on the time of parallel RAM's to compute simple functions, *in* "Proc. 14th Annual ACM Sympos. on Theory of Comput.," 1982, pp. 231–233.
7. S. A. COOK AND P. DYMOND, Hardware complexity and parallel computation, *in* "Proc. 21st Annual IEEE Sympos. on Found. of Comput. Sci.," 1980, pp. 360–372.
8. S. FORTUNE AND J. WYLLIE, Parallelism in random access machines, *in* "Proc. 10th Annual ACM Sympos. on Theory of Comput.," San Diego, Calif., 1978, pp. 114–118.
9. M. FURST, J. B. SAXE, AND M. SIPSER, Parity, circuits and the polynomial-time hierarchy, *in* "Proc. 22nd Annual IEEE Sympos. on Found. of Comput. Sci.," 1981, pp. 260–270.
10. Z. GALIL AND W. J. PAUL, An efficient general purpose parallel computer, *in* "Proc. 13th Annual ACM Sympos. on Theory of Comput.," Milwaukee, Wisc., 1981, pp. 247–256.
11. L. GOLDSCHLAGER, A unified approach to models of synchronous parallel machines, *in* "Proc. 10th Annual ACM Sympos. on Theory of Comput.," San Diego, Calif., 1978, pp. 89–94.
12. A. GOTTLIEB, B. D. LUBACHEVSKY, AND L. RUDOLPH, Coordinating large number of processors, *in* "Proc. of Int. Conf. on Parallel Processing," 1981, pp. 341–349.
13. R. HAGGKVIST AND P. HELL, Parallel sorting with constant time for comparisons, *SIAM J. Comput.* **10** (3) (1981), 465–472.
14. R. HAGGKVIST AND P. HELL, "Sorting and Merging in Rounds," Computing Science technical report TR81–9, Simon Fraser University, Burnaby, B.C., Canada, 1981.
15. D. E. KNUTH, "The Art of Computer Programming, Vol. 3: Sorting and Searching," Addison–Wesley, Reading, Mass., 1972.
16. C. P. KRUSKAL, Searching, merging, and sorting in parallel computation, *IEEE Trans. Comput.*, 1983, in press.
17. T. LANG, Interconnections between PE and MBs using the shuffle-exchange, *IEEE Trans. Comput.* **C-25** (1976), 496.

18. G. LEV, N. PIPPENHER AND L. G. VALIANT, A fast parallel algorithm for routing in permutation networks, *IEEE Trans. Comput.* **C-30** (1981), 93–100.
19. F. P. PREPARATA, New parallel-sorting schemes, *IEEE Trans. Comput.* **C-27** (7) (1978), 669–673.
20. F. P. PREPARATA AND J. VIULLEMIN, The cube-connected cycles, *in* "Proc. 20th Sympos. on Found. of Comput. Sci.," 1979, pp. 140–147.
21. J. REIF AND VALIANT, L. G., "A Logarithmic Time Sort for Linear Size Networks," Aiken Computation Laboratory, TR13-82, Harvard University, 1982.
22. J. T. SCHWARTZ, "Ultracomputers," *ACM Trans. Programming Lang. Systems* **2** (1980), 484–521.
23. Y. SHILOACH AND U. VISHKIN, Finding the maximum, merging, and sorting in a parallel computation model, *J. Algorithms* **2** (1) (1981), 88–102.
24. M. SNIR, "On Parallel Searching," Department of Computer Science technical teport TR045, Courant Institute, New York University, June 1982.
25. L. J. STOCKMEYER AND U. VISHKIN, "Simulation of Parallel Random Access Machines by Circuits," RC-9362, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 1982.
26. H. STONE, Parallel processing with the perfect shuffle, *IEEE Trans. Comput.* **C-20** (2) (1971), 153–161.
27. E. UPFAL, "Efficient schemes for parallel communication, *in* "Proc. of ACM Sympos. on Principles of Distributed Computing," Ottawa, Canada, 1982.
28. L. G. VALIANT, Parallelism in comparison problems, *SIAM J. Comput.* **4** (1975), 348–355.
29. L. G. VALIANT AND G. J. BREBNER, Universal schemes for parallel computation, *in* "Proc. 13th Annual ACM Sympos. on Theory of Comput., Milwaukee, Wisconsin, 1981, pp. 263–277.
30. U. VISHKIN, "A Parallel-Design Space Distributed-Implementation Space (PDDI) General Purpose Computer," RC-9541, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 1983.