ROUTING, MERGING AND SORTING ON PARALLEL MODELS OF COMPUTATION

Extended Abstract

A. Borodin
University of Toronto
J.E. Hopcroft
Cornell University

A variety of models have been proposed for the study of synchronous parallel computation. We review these models and study further some prototype problems. We distinguish two classes of models, fixed connection networks and models based on a shared memory. Routing is the prototype problem for the networks. In particular, routing provides the basis for simulating the more powerful shared memory models. We show 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 a deterministic O(logn) routing or sorting algorithm for an n processor fixed connection network remains open. However, if we consider the more powerful class of shared memory models, we are "almost" able to achieve such an efficient sort via Valiant's parallel merging algorithm. Within a spectrum of models, we show that log log n - log log r is asymptotically optimal for rn processors to merge two sorted lists of 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 including computational physics, weather forecasting, etc. The current state of hardware capabilities will facilitate the use of such parallel processors to many more applications.

This research was supported in part by ONR contract N00014-76-C-0018.

Within this viewpoint, Preparata and Vuillemin [79] 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 [71]) and its development into the Ultracomputer of Schwartz [80], the array or mesh connected processors such as the Illiac IV, the cube-connected cycles of Preparata and Vuillemin [79], or the more basic n-dimensional hypercube studied in Valiant and Brebner [81]. As examples of models based on shared memories, there are the PRAC of Lev, Pippenger and Valiant [81], the PRAM of Fortune and Wyllie [78], the unnamed parallel model of Shiloach and Vishkin [80], and the SIMDAG of Goldschlager [78]. 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 [80]). 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 braodcast 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 a fixed connection network such as the n-dimensional cube, the complexity of merging has been resolved by the fundamental log n algorithms of Batcher (see Knuth [72] 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. For applications in parallel computer architecture we are concerned with special networks such as a d-dimensional hypercube or a shuffle exchange interconnection. In this setting O(log n) global strategies are known as well as $O(\log^2 n)$ local strategies.

For our purposes, a network is a diagraph <V,E> 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 <origin,destination> pair or, more generally, <origin,destination,bookkeeping information>.

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: {origin} → {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.

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

1. Batcher's (see Knuth [72]) $O(\log^2 N)$ 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 [71]), the n-dimensional cube (Valiant and Brebner [81]), or the cube connected cycles (Preparata and Vuillemin [79]). This constitutes a local or distributed strategy, in that each processor p decides locally on its next action using only the packets at p. We also note that Batcher's bound is a worse case bound, holding for any and indeed all initial permutations of the packets.

2. Valiant and Brebner [81] construct an O(log N) Monte Carlo local routing strategy for the n-cube. 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. $\geq 1-N^c$) all packets will reach that destination in $\leq(1+c)$ log N steps. Valiant's analysis can also be used to show that for a random input placement, the "naive strategy" on the n-cube terminates in O(log N) steps with high probability (here, the probability is on the space of possible inputs).

3. The Slepian-Benes-Clos permutation network (see Lev, Pippenger and Valiant [81]) 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.) The obvious question remains as to whether or not there exists a local, worst case, O(log N) routing strategy.

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 problem. The importance of the routing problem is also emphasized in the paper of Galil and Paul [81] 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 same cell) is not allowed.

2. PRAM (Fortune and Wyllie) - Simultaneous fetches are allowed but no simultaneous writes.

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.

   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 and m-edge fixed connection network, we assume the global memory models have n processors and m memory cells. It is then 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, letting n=m, Lev, Pippenger and Valiant [81] 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)). With some extra care, one can also simulate one step of a PRAM or WRAM in time O(r(n)). The idea is roughly as follows:

a) Sort the read requests into consecutive locations; that is, sort pairs <request for memory located in i, by processor j> by i and then j. We note that other authors (see Schwartz [80] for the Ultracomputer) have indicated that sorting can often be used to implement partial routing which is all that is required here. The omitted details are not difficult.

b) Fan in all requests for a location i to a specified processor. This is done in such a way that a given processor only has to remember to which neighbors the requested information must be returned. This step will also be depend on the network and will take either $O(\log n)$ or $O(\log^2 n)$ time.

c) In the specified processors do a memory fetch (i.e. this is another partial sort).

d) The requested information (the memory contents) is distributed back to requesting processors by the specified processor.

A Routing Lower Bound for a Special Case

It turns out to be surprisingly difficult to analyze reasonably 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 global means. We can motivate the use of oblivious strategies by calling attention to the processor-memory interconnection network of the NYU-Ultracomputer (see Gottlieb, Lubachevsky and Rudolph [82]), which requires an oblivious strategy for their memory arbitration scheme.

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

We first prove the lower bound for 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.

If we superimpose the n trees determined by the n possible destinations for packets, each vertex is on n trees. This suggests that we might be able to route n packets through a vertex. Since at most d packets can leave the vertex at any given time this would imply a delay of n/d. The difficulty is that in order to force a packet headed for a given destination to go through vertex v we must initially start the packet on a vertex that is a descendent of v in the particular destination tree. But there may be only a small number of such descendents of v and if many trees have the same set there will not be enough descendents to start each packet at a distinct vertex. This motivates the following technical lemma (proven by induction on n):

Lemma Let $T_{d,k}(n)$ be the minimum number of vertices with k, $k \geq 2$, or more descendents in any n vertex tree with maximum degree d, $d \geq 2$. Then

$$T_{d,k}(n) \geq \frac{n-k+1}{1+(d-1)(k-1)} .$$

In each destination tree mark those vertices that have at least k descendents. Let $k = \sqrt{n/d}$. In the network assign a count to each vertex indicating the number of destination trees in which the vertex is marked. Since at least n/dk vertices are marked in each tree, 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_0 which has at least $\sqrt{n/d}$ descendents in each of $\sqrt{n/d}$ descendent trees. For each of these descendent trees we can place the corresponding packet at some vertex of the network so that it will pass through vertex v_0. Thus $\sqrt{n/d}$ packets will go through v_0. Since v_0 is of degree at most d, it requires time at least equal to $\sqrt{n/d^3}$. In particular, any routing procedure for an n-cube where the route of a packet depends only on the destination requires time at least $\sqrt{n}/(\log n)^{3/2}$.

The above proof can be modified to apply to oblivious routing schemes. Consider a single destination. We no longer have a tree since the route of a packet depends on the source as well as the destination. We modify the lemma to say that there must be at least $T_{d,k}(n)$ vertices having at least k paths through them. The inductive hypothesis is that in any directed graph with maximum fan-in d, with n loop-free directed paths to a designated vertex, at least $T_{d,k}(n)$ vertices must be on at least k paths.

An Oblivious Routing Algorithm For An n-cube

The question remaining is how tight is the lower bound. The answer depends on the actual 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 are close to this lower bound. We exhibit one for the hypercube. Lang [76] has previously given an $O(\sqrt{n})$ oblivious routing algorithm for the shuffle-exchange network.

Future Work

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 regular networks such as the n-cube we know of no minimal scheme better than $O(\sqrt{n/\log n})$.

The primary question is whether there is a local routing algorithm for say an n-cube that is

better than $O(\log^2 n)$. 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 effect is to enlarge bottlenecks to several vertices and hence to allow more packets to get to their destinations in a given time. Although the algorithm appears promising we have not been able to formally analyze its behaviour.

We note that the above strategy avoids queues. It is also an interesting question to study the class of strategies which do not use queues (like Batcher).

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 $\leq,=,\neq$ comparisons and integer $+,-,\times,\div$ and $\lfloor\sqrt{\ }\rfloor$ arithmetic operations in a single step. The following models have already been noted in part II: PRAC, PRAM, WRAM.

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. The parallel computation tree idealizes k-processor parallelism by a 3^k-tree where each node is labelled by a set of k $\{<,=,>\}$ comparisons and the branches are labelled 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 $T^M_{merge}(n,m,p)$ and $T^M_{sort}(n,p)$, the minimum number of parallel steps to merge two sorted lists of n and m elements (respectively, to sort n arbitrary elements) using p processors. Typically, n=m, and p=O(n) or $O(n \log^\alpha n)$. Clearly, for any problem we have

$$T^{PRAC} \geq T^{PRAM} \geq T^{WRAM} \geq T^{\text{parallel computation tree}}$$

Our main contribution is to establish the following two theorems:

Theorem 2 Let M denote the parallel computation tree model. Then $T^M_{merge}(n,n,n \log^\alpha n) = \Omega(\log \log n)$ for all α.

Theorem 3 $T^{PRAM}_{merge}(n,n,n) = O(\log \log n)$.

We use Valiant's algorithm, which already establishes the bound for the parallel comparison tree, but following Valiant [75], Preparata [78] and Shiloach and Vishkin [80], 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 by the above results on all of the above shared memory models except the PRAC (for which we have the log n upper bound of the Batcher merge). For the PRAC, it is not difficult to show that $\Omega(\sqrt{\log n})$ is a lower bound for insertion (and hence merging); recently, Snir [82] has announced an $\Omega(\log n)$ lower bound, thus resolving the problem.

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

Corollary 1 $T^{PRAM}(n,n) = O(\log n \log \log n)$.

Corollary 2 $T^{PRAM}(n,n \log n) = O(\log n)$.

Clearly, Corollary 1 follows from a standard merge sort, whereas Corollary 2 is a restatement of Preparata's [78] 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 $O(\log^2 \frac{n}{\log(p/n)} + \log n)$ for sorting on their version of a WRAM with p processors. With regard to lower bounds for sorting, Haagvist and Hell [81] 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. Cook and Dwork [82] show that $\Omega(\log n)$ steps on a PRAM are required for the Boolean OR no matter how many processors are available. It follows that $\Omega(\log n)$ steps on a PRAM are required for the MAX and sorting. For O(n) processors, $\Omega(\log n)$ is a trivial lower bound resulting from the sequential lower bound of $\Omega(n \log n)$. Among the open questions for parallel sorting are the following: the number of processors for O(log n) time sorting on a PRAC (Preparata [78] achieves O(k log n) time with $n^{1+1/k}$ processors); whether it is possible to sort in time O(log n) with only n processors on a PRAM or WRAM; whether it is possible to sort in time O(1) on a WRAM using less than an exponential in n number of processors. Recently, Stockmeyer and Vishkin [82] have shown how to simulate a WRAM (in particular, the SIMDAG) by an unbounded fan-in AND/OR circuit with only a constant delay factor. By this simulation and some appropriate reducibilities, Stockmeyer and Vishkin are able to use the beautiful lower bound of Furst, Saxe and Sipser [81] to show that a WRAM cannot sort in O(1) time using only a polynomial in n number of processors.

An $\Omega(\log \log n)$ Lower Bound for Merging on Valiant's Model - Sketch of Proof

Since only 2n-1 comparisons are necessary to sequentially merge two n 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.

Consider the process of merging two sorted lists $a_1,\ldots,a_n$ and $b_1,\ldots,b_n$ 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

$$\{(A_{i_\ell},B_{i_\ell})\}$$

such that $i_\ell<i_{\ell+1}$ and $j_\ell<j_{\ell+1}$ for $1\le\ell<\frac{1}{2}\sqrt{n}$. If the total order is such that all elements in $A_{i_\ell}\cup B_{j_\ell}$ are less than any element in $A_{i_{\ell+1}}\cup B_{j_{\ell+1}}$, $1\le\ell<\frac{1}{2}\sqrt{n}$, 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 are made precise.

In what follows let $G = (A\cup B,E)$ be a bipartite graph with $E\subset A\times B$. Further let $A_1,A_2,\ldots$ and $B_1,B_2,\ldots$ 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 Let $G = (A\cup B,E)$ be a bipartite graph with $A = A_1,A_2,\ldots,A_{2k}$ and $B = B_1,B_2,\ldots,B_{2k}$ and let $E\subset A\times B$ have $3k^2$ edges. Then G has a compatible matching of cardinality at least k.

Theorem 2' Let T(s,c) be the time necessary to solve k, $k\ge1$, problems of size s with cks processors. Then T(s,c) is $\Omega(\log \frac{\log sc}{\log c})$.

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. 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 $T(s,c) \le 1+T(\frac{1}{2}\sqrt{s/2c},4c)$. We show by induction on s, that $T(s,c) \ge d\log \frac{\log sc}{\log c}$ for some sufficiently small d. Observe that $\log \frac{\log sc}{\log c}$ 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 - Sketch of Algorithm

We assume the reader is familiar with Valiant's (n,m) merging algorithm which merges X and Y with #X=n, #Y=m, $n\le m$ using n+m processors. Our goal is to implement Valiant's algorithm on a PRAM.

On the completion of stages (a), (b) and (c) of Valiant's algorithm we can store each $x_{i\lceil\sqrt{n}\rceil}$ in its appropriate place in the output Z. It then remains to merge the $\lfloor\sqrt{n}\rfloor$ 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 there will clearly 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 [80] refer to as the processor allocation problem.

We desire a recursive procedure MERGE(i,n_i,j,m_i,k) which merges $x_i,x_{i+1},\ldots,x_{i+n_i-1}$ and $y_j,\ldots,y_{j+m_i-1}$ into $z_{i+j-1},\ldots,z_{i+j+n_i+m_i-2}$ using at most n_i+m_i processors beginning at processor number p_k. Such a merge will be simultaneously invoked by processors $p_k,p_{k+1},\ldots,p_{k+n_i+m_i-1}$. The initial call is MERGE(1,n,1,m,1).

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 determined for each i, $0\le i\le\lfloor\sqrt{n}\rfloor-1$ that $y_{j_i}<x_{i\lceil\sqrt{n}\rceil}\le y_{j_i+1}$ and that we have constructed a table J

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

accessible by all processors. A given processor p must determine its role in the next iteration of the algorithm.

Lemma 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 $Y_{r-1} = (\ldots,y_f)$. There exists a function ϕ such that no more than $\phi(r,n,f)$ processors have been assigned. Indeed, $\phi(r,n,f) = r\sqrt{n}+f$.

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 $\#Y_k \le m/\sqrt{n}$ (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, $0\le k\le\sqrt{n}-1$, we assign $(n+m)/\sqrt{n}$ processors to look at both the k and the $k+1^{st}$ entry of the table J. If $j_{k+1}-j_k\le m/\sqrt{n}$, 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_ℓ} be the first processor of block ℓ. The k^{th} processor of block ℓ looks at the j_k and j_{k+1} in table J and determines (via the function ϕ) whether or not processor p_{j_ℓ} would be assigned to this subproblem. Now each processor p in the ℓ^{th} 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_ℓ}.

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

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 $(n+m)/\sqrt{n}$ 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.

Kruskal [82] has recently unified (and improved) Valiant's merging algorithms to yield the following upper bound: p processors can do an (n,m) merge in time $\frac{n+m}{p}$ + 1.893 log log n + $O(\log(\frac{n+m}{p}))$. 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. The claim is that this improvement can also be implemented on a PRAM. Finally then, this permits corresponding improvements in Corollaries 1 and 2 (e.g. T(n,n) = O(log n log log n/log log log n). □

Acknowledgement: 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 in our formulation of the problem. We also thank L. Rudolph for many helpful comments.

References

Cook, S. and C. Dwork, Bounds on the Time for Parallel RAM's to Compute Simple Functions, Proc. of the 14th Annual ACM Symposium on Theory of Computation, May 1982.

Fortune, S. and J. Wyllie, Parallelism in Random Access Machines, Proceedings 10th Annual ACM Symposium on Theory of Computing, San Diego, California, 1978, 114-118.

Furst, M., J.B. Saxe, and M. Sipser, Parity, Circuits and the Polynomial Time Hierarchy, Proc. of 22nd Annual Symposium on Foundations of Computer Science, Oct. 1981, 260-270.

Galil, Z. and W.J. Paul, An Efficient General Purpose Parallel Computer, Proceedings 13th Annual ACM Symposium on Theory of Computing, Milwaukee, Wisconsin, 1981, 247-256.

Goldschlager, L., A Unified Approach to Models of Synchronous Parallel Machines, Proceedings 10th Annual ACM Symposium on Theory of Computing, San Diego, California, 1978, 89-94.

Gottlieb, A., B.D. Lubachevsky, and L. Rudolph, Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors, to appear in ACM TOPLAS, 1982.

Haagvist, R. and P. Hell, Parallel Sorting with Constant Time for Comparisons, SIAM J. on Computing 10:3, 1981, 465-472.

Knuth, D.E., The Art of Computer Programming, vol.3, Sorting and Searching, Addison-Wesley, Reading, Massachusetts, 1972.

Kruskal, C., Personal Communication, 1982.

Lang, T., Interconnections between PE and MBs using the Shuffle-Exchange, IEEE Transactions on Computers, vol. C25, 1976, 496.

Lev, G., N. Pippenger and L.G. Valiant, A Fast Parallel Algorithm for Routing in Permutation Networks, IEEE Transactions on Computers, 1981.

Preparata, F.P., New Parallel-Sorting Schemes, IEEE Transactions on Computers, C27:7, 1978, 669-673.

Preparata, F.P., and J. Vuillemin, The Cube-Connected Cycles, Proceedings 20th Symposium on Foundations of Computer Science, 1979, 140-147.

Schwartz, J.T., Ultracomputers, ACM TOPLAS 2, 1980, 484-521.

Shiloach, Y., and U. Vishkin, Finding the Maximum, Merging and Sorting in a Parallel Computation Model, Department of Computer Science, Technion Israel, TR173, March 1980.

Snir, M., On Parallel Search (Extended Abstract), Preprint-Courant Institute, 1982.

Stockmeyer, L., and Vishkin, U., Simulation of Parallel Random Access Machines by Circuits, IBM Yorktown Heights, Preprint, 1982.

Stone, H., Parallel Processing with the Perfect Shuffle, IEEE Transactions on Computers, C20:2, 1971, 153-161.

Valiant, L.G., Parallelism in Comparison Problems, SIAM J. on Computing 4, 1975, 348-355.

Valiant, L.G., and G.J. Brebner, Universal Schemes for Parallel Computation, Proceedings 13th Annual ACM Symposium on Theory of Computation, Milwaukee, Wisconsin, 1981, 263-277.