Theoretical Computer Science 32 (1984) 157–172
North-Holland

A PARALLEL-DESIGN DISTRIBUTED-IMPLEMENTATION (PDDI) GENERAL-PURPOSE COMPUTER

Uzi VISHKIN*

Department of Computer Science, Courant Institute, New York University, New York, NY 10012, U.S.A.

Communicated by M. Nivat
Received July 1983

Abstract. A scheme of an efficient general-purpose parallel computer is introduced. Its design space (i.e., the model for which parallel programs are written), is a permissive parallel RAM model of computation. The implementation space is presented as a scheme of a synchronous distributed machine which is not more involved than a sorting network followed by a merging network. An efficient translation from the design space into the implementation space is given. Suppose for some t and x there is a parallel algorithm in the design space which has depth (i.e., parallel time) $O(t/p)$ using p processors for all $p \leq x$. This translates to an algorithm in the implementation space with depth $O(t/s)$ for all $s \leq t/l$ where l depends on the choice of the sorting and merging networks, s is the number of 'powerful' processors used (processors not in the sorting or merging networks) and $f(s, m)$ auxiliary processors, where m is the size of the common memory in the design space. For a specific choice, $l = \log^2 s + \log m$ and $f(s, m) = O(s \log^2 s + m \log m)$, comparing favorably with alternative known solutions. Since many parallel algorithms are designed for a wide range of processors our solution pays the fine for implementation where it hurts least.

1. Introduction

This paper is motivated by the fact that the tremendous potential power of microstructure technology can be realized only if we find effective parallel architectures and algorithms for utilizing large numbers of small but powerful processors. On one hand, synchronous shared memory models of parallel computation have been shown to be a very effective framework for designing algorithms for many problems. On the other hand, physical limitations of currently available technologies suggest one, but only one, basic constraint: in a machine built as an assemblage of a large number of processing elements, each processor can be connected only to a fixed number of other processors, and this in a fixed pattern.

In order to support the claim that such models of parallel computation are effective we mention a few salient algorithms that can be implemented in them. (Most of these algorithms were designed for such models.) Finding the maximum among n elements

* Present address: Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel.

Part of the research reported here was performed during the author's visit at IBM Thomas J. Watson Research Center from September 1981 to August 1982.

[23]. Merging [5, 23]. Sorting ([2, 5, 12, 20, 23] and more). Computing convex hulls in two dimensions [19]. Computing connected components of undirected graphs [6, 13, 24, 30, 33]. Computing biconnected components of an undirected graph [28]. Algorithms on trees [17, 28]. Data structures [21]. Finding max-flow in a network [25]. Numerous numerical algorithms (for a survey see [11]).

We suggest a solution for the following problem.

Problem. Design an efficient general-purpose parallel computer that satisfies three requirements:

(1) The design space (i.e., the model of computation for which programs are written) is a *permissive* synchronous shared memory model of parallel computation. In particular, the Fetch-and-* Parallel RAM ($F\&*$ PRAM).

It is slightly more permissive than the concurrent-read concurrent-write parallel RAM (CRCW PRAM). See [26] for a formal definition of the CRCW PRAM. The CRCW PRAM consists of a sequence $P_1, P_2, \ldots, P_p$ of RAM's operating synchronously in parallel. Each individual RAM is similar to a standard uniprocessor model as defined in [1, Chapter 1]. In particular, each RAM is assumed to have its own local random-access memory and has instructions for typical arithmetic and boolean operations and for reading from and writing into its local memory. The RAM's also have access to a shared memory of size m, and each RAM has instructions for reading from and writing into the common memory using one of its private registers to specify the common memory address. Several processors may read simultaneously from the same memory location. If more than one processor attempts to write into the same location in common memory at the same time, the lowest numbered processor succeeds. Let us go back to the $F\&*$ PRAM.

Let A be a common memory address, e_i be a local register of processor P_i and $*$ be an associative and commutative operation. Define the Fetch-and-* ($F\&*$) instruction as follows. (It is similar to [9].) If processor P_i performs an $F\&*(A, e_i)$ and no other processor performs at the same time an instruction that relates to address A, then a local register of P_i is assigned with the contents of A and A is assigned with $A*e_i$. Suppose that several processors perform simultaneously $F\&*$ (for the same $*$ operation) instructions that relate to A. The result is defined to be as if they performed these instructions serially in some order.

We assume that no processor is seeking access to address A with another type of instruction or with an $F\&*$ instruction for another $*$ operation. If this happens the algorithm is considered illegal. Alternatively, some default results can be imagined.) The $F\&*$ PRAM is a CRCW PRAM that allows these $F\&*$ instructions for some set of $*$ operations. Each instruction takes one time unit (uniform cost criterion). Both the program and the input are located in the common memory.

(2) The implementation space (i.e., the model of computation in which the machine is specified) is a synchronous distributed model of parallel computation, where each processor is connected in a fixed pattern to a small number of others.

(3) There is an efficient automatic procedure that translates every algorithm for the design space into the implementation space.

This presentation of the problem explains why we call our solution a parallel-design distributed-implementation (PDDI) computer.

This problem lies in the heart of the theory of parallel computation. An efficient solution of the problem plays also a central role in the theory of distributed computation, since it leads to a utilization of distributed machines visualizing a mathematically appealing and effective design space. In general, it seems unlikely that programmers will be able to write efficient algorithms directly for fixed pattern distributed machines even if the fixed pattern changes from one algorithm to the other as implied by the general-purpose distributed computer of Galil and Paul [10]. (The term 'distributed' in the present paper corresponds to 'parallel' in [10].) This explains why our problem is more general than theirs: their design space is a synchronous distributed model of computation; [16] implies that there exists a simple translation of a program in their design space into our design space in constant time using the same order of the number of processors. Thus, our simulation can be utilized to solve the problem of simulating every special-purpose synchronous distributed machine on our PDDI machine. A simulation that solves this problem is the main contribution of [10]. The worst-case time analysis of their solution is the same as ours (without the improvement due to the efficient version of Section 6) for comparable cases. However, our solution allows more general patterns of communication for the design space and, therefore, equips the designer with more powerful design tools. For example, information which is known to one processor only (it appears in its local memory) may become known to any subset of the processors in constant time through the common memory by utilizing both the common memory and simultaneous reads from the same common memory location. While a time lower bound of the order of the logarithm of the number of processors can be readily established for instances of this problem in a synchronous distributed model where the degree of each node is bounded by a constant, due to fan-in considerations.

Our solution compares favorably with the 'naive' solution for the main problem. By the naive solution we mean the following: There are (1) p (balanced) binary trees each having m leaves, called 'processor-trees', and (2) m (balanced) binary trees each having p leaves, called 'memory-trees'; each of the leaves of a processor-tree is shared with a leaf of a distinct memory-tree. Each processor- (resp. memory-) tree corresponds to one of the $F\&*$ PRAM p processors (resp. m common memory locations). The communication between a processor and a common memory location is simulated in the obvious way via their shared leaf. Extending it for a translation of $F\&*$ PRAM algorithms by this synchronous distributed machine is straightforward. The naive solution multiplies time requirements by $O(\log m + \log p)$ and processor requirements by $O(m)$. Use of pipelining in a way similar to our solution can further improve this solution. The main disadvantage of this approach is the relatively large number ($O(pm)$) of 'auxiliary' processors required. This inefficiency is due to the fact that each leaf is dedicated to simulate communication between a certain processor and a certain common memory location regardless of the need for such communication in the time unit being simulated. Our solution provides for a dynamic assignment of auxiliary processors for this purpose, thereby substantially reducing

the number of auxiliary processors. Applications of this technique can be found in [7], [15] (for related simulation problems) and [27] (for sorting). This technique is sometimes called 'Orthogonal Trees'.

Simulations of tightly coupled parallel computation models by a distributed model of computation is also studied in a few other papers. Each of these works either solves another problem than ours or does not provide for a worst-case efficient solution. Lev, Pippenger and Valiant [16] mention simulations of the exclusive-read exclusive-write (EREW) PRAM model, where concurrent access of more than one processor to the same common memory location is forbidden. Borodin and Hopcroft [5] outline another solution for our problem for the case $p = m$. We refer to their simulation later in the paper. Vishkin [29] presents a solution for an easier problem. The implementation space is an EREW RAM and not a distributed machine.

The comprehensive paper [22] describes the 'Paracomputer', a model of parallel computation very similar to our CRCW PRAM and proposes the former as a model suitable for studying theoretical aspects of parallel computation. Various Paracomputer algorithms are implemented in the 'Ultracomputer' (a perfect shuffle interconnection machine). The paper [9] suggests to replace the CRCW-PRAM-Paracomputer by a Fetch-and-Add-PRAM-Paracomputer and the Perfect-Shuffle-Ultracomputer by another interconnection network. The automatic procedure for the simulation of the Paracomputer by the Ultracomputer which is suggested is claimed to satisfy a good average-case criterion. No claims are made regarding worst-case criteria that this simulation satisfies. Note finally that the subsection on 'alignment networks' by Kuck [14] contains a survey of known interconnection networks for processors and memories.

The general design of the machine is given in the next section. It is followed by a few details that prepare the reader for the simulation. Section 3 gives an important part of the simulation. Its correctness is proven in Section 4. Other parts of the simulation are discussed in Section 5. An efficient version of our simulation that utilizes pipelining and gives our result an edge over previously suggested simulations appears in Section 6. Section 7 includes a few concluding remarks.

2. Preliminaries

In outlining the solution, there was an effort to describe it in the most general form leaving as much freedom to the reader as possible for filling in details that might have a few alternatives. More specifically, an implementation scheme is given that reduces the simulation problem into the problems of designing networks for sorting and merging. These problems are probably among the first to be considered in any new technology. Thompson [27] describes, for example, thirteen (!) algorithms for sorting in a model of VLSI. Let us describe the machine.

The synchronous distributed computer (*SDC*): The machine has a sequence of RAM's $S_1, S_2, \ldots, S_n$ to be called *super-processors*. Each individual super-processor

has many properties in common with the processors of the $F\&*$ PRAM. Actually, the description of the processors of the parallel computation model, up to the point where we start discussing their access to the common memory (that does not exist here), is similar for the super-processors. Our model employs also two families of fairly degenerate processors: (1) A sequence of processors $R_1, R_2, \ldots, R_n$ called *comparator-processors*; and (2) A third sequence of processors $M_1, M_2, \ldots, M_m$ called *memory-processors.*

Each of the comparator-processors has instructions for checking the predicates $=$ and $<$. Each of the comparator- or memory-processors has a small local memory; it can read from and write into its local memory; it can perform only the $*$ operations that we wish to include in the $F\&*$ instruction of our $F\&*$ PRAM design space. It has a program which is independent of the algorithm being implemented. This program is located in its local memory.

All processors operate synchronously in parallel. They can be thought of as nodes (vertices) that are connected by lines (edges) forming a *graph of communication.* Fig. 1 illustrates the general structure of the SDC graph of communication; super-processors are represented by circles and memory-processors by triangles. Comparator-processors only fill the area referred to as sorting and merging networks. Each processor has an additional instruction that serves as the main communication tool. The information to be communicated is loaded into a communication register which corresponds to the adjacent line on which this information is to be transmitted. The processor on the other side of the line may read this register. The degree of each vertex of our graph of communication does not exceed 4. Every instruction takes one time unit (uniform cost criterion).

Note that we do not specify the sorting and merging network. Our results and analysis hold for any such networks. For instance, we may use Batcher's networks.

Our goal is to implement the $F\&*$ PRAM into the SDC.

We define a 1–1 correspondence between the p processors of the $F\&*$ PRAM model and the p super-processors of the SDC. Each super-processor S_i is 'responsible' to simulate the behavior of its corresponding P_i.

Another 1–1 correspondence is defined between the m addresses of the common memory and the memory-processors. Each memory-processor M_i is responsible to store and simulate the updates of the content of (common) memory address i. The simulation of access of processors to the common memory is done by the super-processors and the memory-processors through a network of nodes and lines. All the nodes in this network are comparator processors.

To distinguish between time units of the algorithm and its implementation we refer to the former as *pulses.* We assume that the pulses can be partitioned into three sets: *reading pulses*, $F\&*$ *pulses*, and *writing pulses.*

Remark. The variant of simultaneous access to the same (common) memory location for a mixed objective (e.g., two or more of reading, $F\&*$ and writing) can be avoided without changing the running time of the algorithm by an order of magnitude. Break

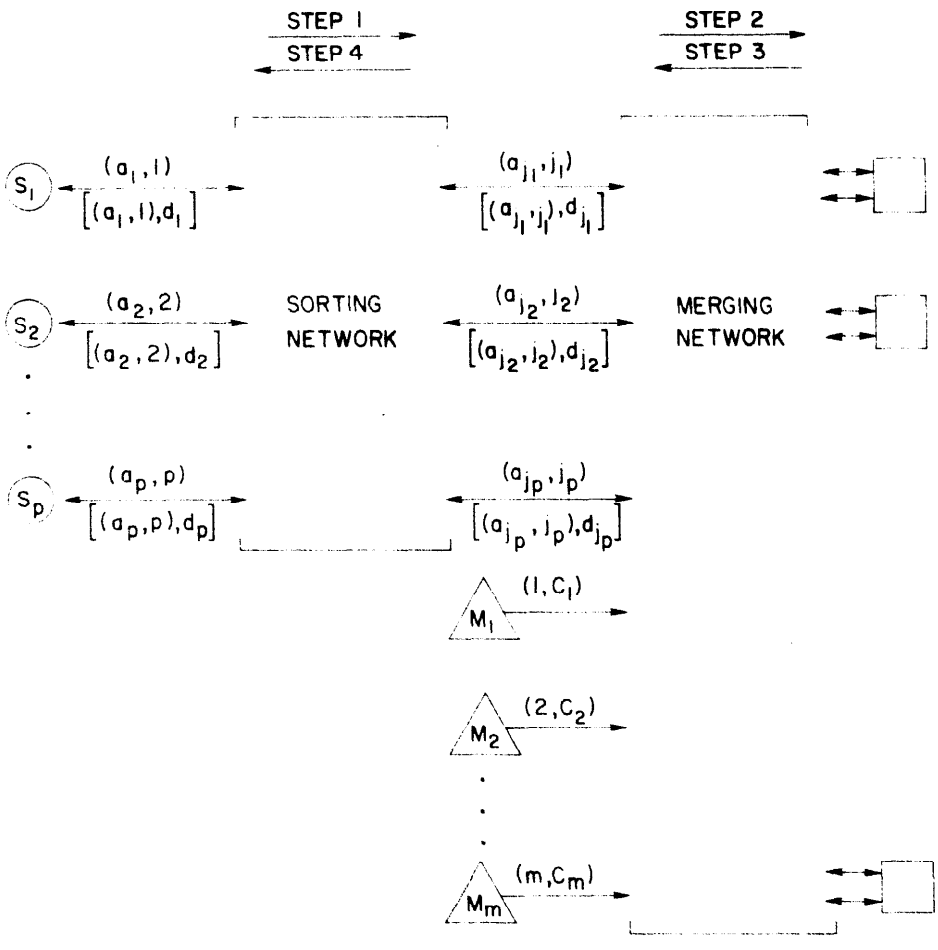

Fig. 1. The network of the implementation. By $\xleftarrow[[(a_1,1),d_1]]{(a_1,1)}$ we mean that $(a_1, 1)$ is transmitted on this line from left to right while $[(a_1, 1), d_1]$ is transmitted from right to left.

each time unit into three. In the first third part of the reading is performed, in the second the $F\&*$ and in the third the writing.

Remark. Let Seq(n) be the lowest worst-case upper bound on the running time of a sequential algorithm for a certain problem of input size n. Obviously, the best upper bound on the parallel running time achievable, without improving the sequential result, for an algorithm using p processors in the $F\&*$ PRAM is of the form O(Seq(n)/p). An algorithm that achieves this running time is said to have 'optimal speed-up', or more simply to be 'optimal'. Upper bounds on the worst-case resource requirements of algorithms which are designed for parallel PRAM's are usually presented as: Depth O(y) for z processors and m common memory locations (y, z, m may be functions of the input parameters). An equivalent formulation of such a result is: Depth O(($y*z$)/p) for all $p \leq z$ processors and m common memory locations for the same y, z and m. We use mostly the second formulation.

3. Simulation of a reading pulse

Say that processor P_i, $1 \leq i \leq p$, wishes to read from address a_i of the common memory. Then S_i (the corresponding super-processor) communicates this by writing into its communication register. In case P_i does not wish to read at the pulse, a_i is assigned fictitiously a nonexistent address. (This nonexistent address is the default content of P_i's communication register.) So, if P_i wishes to perform any instruction other than having access to the common memory then S_i (the corresponding superprocessor) can do it in one time unit.

The simulation of reading from the common memory involves four steps.

Step 1. Apply a sorting network (e.g., Batcher's [3] network) to sort the pairs $(a_1, 1), (a_2, 2), \ldots, (a_p, p)$ in the lexicographic order. Namely, $(a_i, i) < (a_j, j)$ iff $a_i < a_j$ or $a_i = a_j$ and $i < j$. Denote the output of the sorting procedure by (a_{j_1}, j_1), $(a_{j_2}, j_2), \ldots, (a_{j_p}, j_p)$.

Processors that wish to read from the same address are represented in this output by consecutive serial numbers, and sorted according to their original serial numbers. As we mentioned earlier, comparator-processors take the place of comparator modules in the sorting network (and in the merging network of the next step). In addition to their functioning as such, they keep for each input the line on which it arrived. It is used in Step 4 for returning each (a_i, i) along the same path it arrived on.

Remark. A similar application of sorting appears also in [5] and [29]. However, the contribution of this paper is in the next step where we show how to use merging networks for the purpose of fanning out the contents of memory cells to processors that seek to read them. The use of merging networks enables us to use pipelining (see Section 6) in the efficient version of our simulation, which improves the time complexity of [5]. Also, the number of auxiliary processors seems to be smaller than theirs. It should be noted that they do not specify the number of processors. The above applies for comparable cases only, since [5] presented their solution only for the case $p = m$. The merging networks are used in a nonstandard way since we are interested in the intermediate computations of the network rather than in the merging itself. In a companion paper [31] we coin the name '*lucid-box composition technique*' (as opposed to black-box compositions) for this wider notion for using effectively intermediate results of existing (or supposedly existing) procedures for the purpose of designing new ones. This paper includes more examples where this technique is useful.

Step 2. Apply a merging network (e.g., Batcher's network) to merge the output of the sorting network $(a_{j_1}, j_1), (a_{j_2}, j_2), \ldots, (a_{j_p}, j_p)$, and the (sorted) list of addresses of the common memory denoted (w.l.g.) by $1, 2, \ldots, m$. For the merging, we define $(a_i, i) < j$ iff $a_i \leq j$. The comparator-processors of the merging network keep for each input pair (a_i, i) the line on which it arrived.

To each memory address i we attach its content c_i that moves together with i in the merging network. To each pair (a_j, j) we attach a variable d_j. Upon entering the merging network at the beginning of Step 2, d_j is 'undefined' for each j, $1 \leq j \leq p$.

Whenever a memory address j meets a pair (a_i, i), such that $a_i = j$, in a comparator-processor of the network we copy the content of this address into d_i. Whenever two pairs (a_i, i) and (a_j, j) such that $a_i = a_j$ meet at a comparator-processor and one of them, say (a_i, i), found already in a previous time unit its d_i ($d_i \neq$ 'undefined') we copy this value into d_j.

Fig. 2 illustrates Step 2.

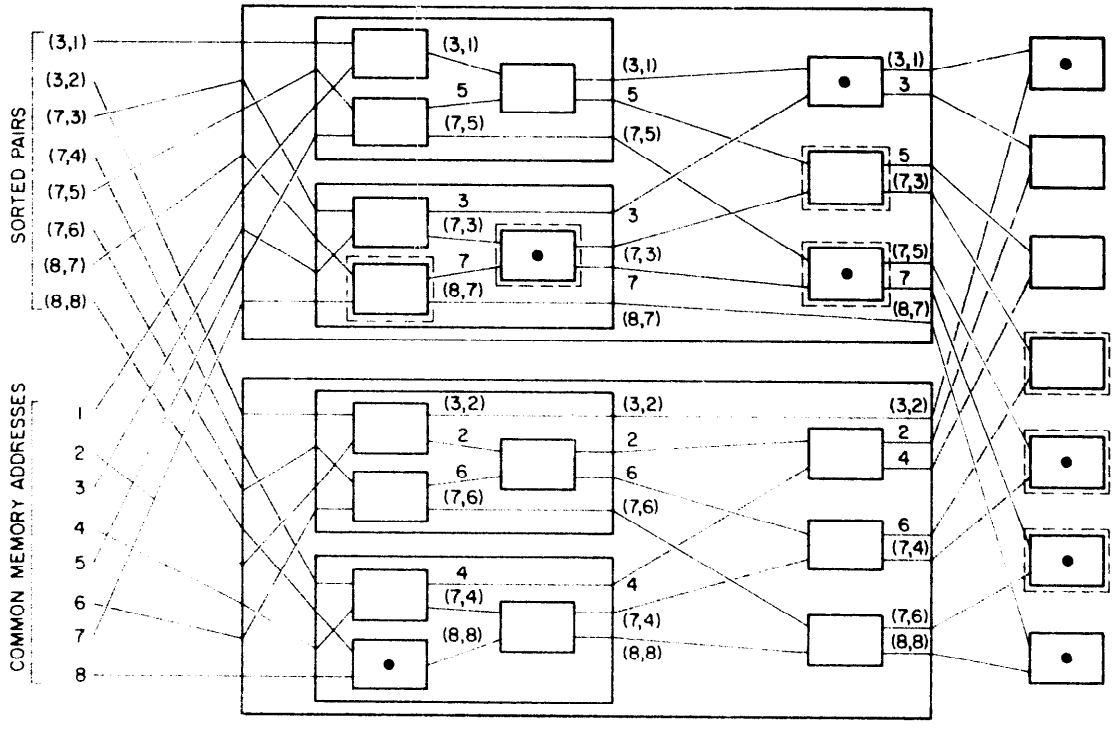

Fig. 2. Example of Step 2. $p = m = 8$, $a_1 = a_2 = 3$, $a_3 = a_4 = a_5 = a_6 = 7$, $a_7 = a_8 = 8$. Dotted boxes represent comparator-processors in which content of some common memory address is copied in Step 2. Dashed boxes represent comparator-processors which are member of V_5, in the proof of Theorem 1.

Actually, we eliminate the output lines of the merging network, since we are not interested (as explained later) in the merging itself; so the last layer of the Batcher merging network, for instance, will be the last 'layer' of our implementation network. The comparator-processor of this last layer, together with the first and last comparator-processor of the first layer of the Batcher's merging network are referred to as *terminals* as there are output lines on the network that emanate from them. It should be clear how to generalize the definition of terminals to general merging networks. In the next section we prove that upon finishing Step 2, when $[(a_j, j), d_j]$ is transmitted to a corresponding terminal comparator-processor $d_j = c_{a_j}$, namely, d_j equals the content of memory address a_j which is exactly what super-processor S_j wishes to read.

Step 3. Each $[(a_i, i), d_i]$ is returned through the merging network, along the paths it traversed during Step 2, to its 'output processor' of the sorting network.

Step 4. It is further returned to its S_i.

4. Correctness proof of the reading pulse simulation

To remind the reader, $[(a_j, j), d_j]$ corresponds to a message originated at super-processor j. This super-processor wishes to read from address a_j. We claim that upon arriving at its corresponding terminal comparator-processor, at Step 2, the variable d_j, of this message, is assigned with c_{a_j}, the content of address a_j. Sending the message $[(a_j, j), d_j]$ back to super-processor j in a later step will achieve the goal of bringing c_{a_j} to j's 'knowledge'.

Let $P_{i_1}, P_{i_2}, \ldots, P_{i_k}$ $(i_1 < i_2 < \cdots < i_k)$ be all the processors that wish to read from the common memory address j ($a_{i_l} = j$ for $1 \leq l \leq k$) in the current pulse.

Let us define recursively the set V_j that contains comparator-processors and memory processor j (vertices) as its elements:

(1) $j \in V_j$;

(2) $v \in V_j$, if for some $w \in V_j$, there is a line directed from w to v such that the message (j, c_j) or a message $[(a_{i_l}, i_l), d_{i_l}]$, $(1 \leq l \leq k)$, was transmitted at Step 2 along this line.

See, for example, the set V_7 in Fig. 2.

Remark. In general merging networks it is possible (apparently due to redundancy) that the set V_j induces a connected graph which is not a tree. It is easy to see that in Batcher's network it induces a tree.

Theorem 1. *Let v be a terminal comparator-processor which corresponds to a message $[(a_{i_l}, i_l), d_{i_l}]$ $(1 \leq l \leq k)$. (Namely, this message is received at v but not transmitted further by Step 2 of the simulation.) Then $v \in V_j$. (Remember $a_{i_l} = j$.)*

Remark. Theorem 1 readily implies the correctness of the reading pulse simulation. This is because every comparator-processor v in V_j 'gets' the value of c_j through a line that implied its membership in V_j (see the recursive definition of V_j) and therefore each d_{i_l} is assigned by c_j until the message $[(a_{i_l}, i_l), d_{i_l}]$ arrives to its corresponding terminal comparator-processor.

Proof of Theorem 1. The only fact required for the proof is that the merging network is correct. The output of our merging includes a successive list of $k+1$ elements sorted in the following order:

$$[(a_{i_1}, i_1)d_{i_1}], [(a_{i_2}, i_2), d_{i_2}], \ldots, [(a_{i_k}, i_k), d_{i_k}], (j, c_j).$$

We assume that common memory addresses are limited to integers, and so are the a_i's. (There could be a problem when a_i corresponds to a nonexistent address.) Imagine that instead of (j, c_j) we take $(j-\frac{1}{2}, c_j)$ leaving all the other inputs as they were. In that case, the same list of $k+1$ elements is placed in the same block of the output but in a different order. The new order is

$$(j-\tfrac{1}{2}, c_j), [(a_{i_1}, i_1), d_{i_1}], \ldots .$$

Remark. Although we do not need the output lines of the merging network for the simulation, we use them here for the sake of argument.

Claim. (1) *Every line which is traversed by one of our $k+1$ messages in the first application of the network is traversed by one of the corresponding $k+1$ messages in the second application and vice versa.*

(2) *Each message other than these $k+1$ messages traverses exactly the same lines in both applications.*

Proof of the claim. If a certain entering line to a comparator-processor transmits one of the $k+1$ messages in each application (not necessarily the same) and the other entering line to this comparator-processor transmits one of the other messages in both applications, the result of the comparison will be the same. Thus, corresponding inputs will be sent through the same emanating lines. The case where both inputs belong, or do not belong, to the $k+1$ messages in both applications is easy because no 'harm' can be done. □

Proof of Theorem 1 (*continued*). Let us look at one of the $[(a_{i_l}, i_l), d_{i_l}]$ messages. We wish to prove that its corresponding terminal comparator-processor belongs to V_l. Let us trace this $[(a_{i_l}, i_l), d_{i_l}]$ message in both applications of the merging network described above. We are going to have two paths that start at the same input. Denote by v_1 the last comparator-processor that belongs to both paths before they split for the first time. There is such v_1 because the paths do not end at the same output line. Proving that $v_1 \in V_l$ would imply that the corresponding terminal comparator-processor to our message $[(a_{i_l}, i_l), d_{i_l}]$ belongs to V_l. Just apply the definition of V_l to the path traversed by the message from v_1 to this terminal comparator-processor to see that. So, it remains to show that $v_1 \in V_l$.

Since one of the v_1's entering lines input the message $[(a_{i_l}, i_l), d_{i_l})]$ in both applications the messages input on the other entering line could not be the same. Let v_2 be the comparator-processor on the other side of this line. As v_2 had a different output from one application of the network to the other, one of its entering lines had to transmit different inputs. Let v_3 be the comparator-processor on the other side of such a line, and so on. Since our merging network is finite and acyclic we arrive eventually at memory processor j. This is because it is 'responsible' for the only input line to the merging network which inputs different data in both applications.

Since j belongs to V_l so does v_1. □

5. Simulation of F&* and writing pulses

Consider a general merging network, where V_j does not necessarily induce a tree. (Recall the remark that precedes Theorem 1.) Following the ideas of the fan-out method of Step 2 in the reading pulse simulation, we can identify very easily comparator-processors whose in-degree in the graph induced by V_j are 2. Eliminating one of the entering edges in each of them would reduce the induced graph to a tree. Denote this tree by V_j'. Appendix A shows how to utilize a binary-tree synchronous distributed machine for computation of an F&* instruction. The F&* instructions that relate to the same shared memory location are entered into the leaves of the tree. Then we climb up the tree from the leaves to the root. Later we return to the leaves. The main idea here is that the V_j' trees can serve as these trees. Let us look at Fig. 1 and try to imagine the direction in which the data moves in the SDC machine. There will be six steps instead of four in the reading pulse simulation. In the first and second steps the information moves from left to right in the sorting and merging networks, respectively. The V_j' trees, which are subgraphs of the merging network, lie on their side, the roots on the left of the leaves. Climbing up the trees corresponds to moving data from right to left in the merging network. This is done in Step 3. In Step 4 we climb down the tree and, therefore, move from left to right in the merging network. In Steps 5 and 6 the data are moved from right to left in the merging and sorting networks, respectively, thereby transmitting the required 'partial sums' to the processors. (The sums were transmitted to the appropriate common memory locations in Step 3.)

No new ideas are required for simulation of the writing pulse. It is left as an exercise for the interested reader.

Complexity. Using Batcher's merging and sorting networks; the sorting network requires $O(\log^2 p)$ layers, each has $O(p)$ comparator-processors. The merging network has $O(\log(p+m))$ layers, $O(m+p)$ comparator-processor in each, plus m memory processors. So, to implement one pulse of the algorithm in the F&* PRAM into the SDC machine, we need $O(\log^2 p + \log(m+p))$ time.

In the next section we show how this can be improved.

6. Efficient simulation

Let us have a second look at a pulse of an algorithm of the F&* PRAM. Say that our parallel computation model employs p processors and uses common memory of size m as before. So far, we have presented a simulation of this algorithm in an SDC machine which uses p super-processors, m memory-processors and $f(p, m)$ comparator-processors where f is a function of p and m that depends on the sorting and merging networks that we utilize. Let $l(p, m)$ be the longest directed path starting at a super processor, or at a memory processor in the combined (both merging and sorting) network of the implementation.

In this section the number of the super-processors is denoted by s. It is smaller than p, the number of processors of the parallel computation model. While m, the number of memory-processors, is the same as the number of common memory addresses, as before.

Super-processor S_i, $1 \leq i \leq s$, will be 'responsible', during this section, to simulate the behavior of processors P_i, P_{i+s}, P_{i+2s}, ... in each pulse of the algorithm which is formulated in the parallel computation model. Therefore, we sometimes call the processors of the $F\&*$ PRAM design space *virtual processors.*

We do it by pipelining. In the first time unit of the pulse simulation, we start a process similar to the simulation of an algorithm that is given for $P_1, \ldots, P_s$ only and m common memory addresses by $S_1, \ldots, S_s$. We call this process the *first cycle* of the pulse simulation. After a constant number of time units we start a second cycle similar to the pulse simulation of an algorithm that is given for $P_{s+1}, \ldots, P_{2s}$ and m common memory addresses by $S_1, \ldots, S_s$, and so on. Simulation of reading pulses does not require more than that. Simulation of a $F\&*$ pulse involves the following additional observations.

Let y denote both the common memory location that a certain $F\&*$ instruction relates to, and the memory processor that simulates this memory address. No confusion will arise. We use the V'_y trees which are formed during each cycle in a form similar to the previous section. Due to a small change, however, these trees compute the $F\&*$ instruction relative to the present pulse (rather than to the present cycle). Remember that memory-processor y is the root of all these V'_y trees. Recall, from the way the previous section invokes the appendix, that the computation of the $F\&*$ instruction involves computation of the A numbers (moving up the V'_y tree or, equivalently, moving right to left in the merging network) followed by computation of the B numbers (moving down V'_y). Here, we start by computing similarly the A numbers. Following this, memory-processor y has its A number in V'_y of the current cycle; this is the '*-sum' of the contents of all virtual processors that participate in the $F\&*$ instruction and are simulated by the cycle. (This A number corresponds to $A(\alpha, 1)$ of Appendix A. Denote it by $A(y)$ and the corresponding B number by $B(y)$.) Let us call the time immediately after the computation of $B(y) \leftarrow B(y) * A(y)$ by memory-processor y and before the next cycle arrives at y the *midpoint* of the cycle. It should be obvious that at this time memory-processor y is 'ready' for the cycle that follows and the computations for this cycle will be with respect to the pulse being simulated. This is so since the only interaction between cycles is at the memory processors.

The degree of each node in the graph of communication of our SDC machine is actually bounded by sixteen (physically it is bounded by four) in the simulation that involves pipelining. This is because:

(1) Each line can be used simultaneously in both directions by two different cycles of the same pulse.

(2) Two cycles of a simultation of a $F\&*$ pulse may simultaneously use the same (merging network) line in the same direction.

If we wish that a processor of the SDC machine is not concerned with more than one of its lines at a time we can do it by partitioning the lines of the network into seventeen sets (the maximum degree of a node plus one, by Vizing's Edge Coloring Theorem [4]) in such a way that no two lines of the same set share a node.

So it takes $O(p/s + l(s, m))$ time units to simulate one pulse of the algorithm. If $p/s \geq l(s, m)$, then the number of time units is simply $O(p/s)$. So, to sum up, we have the following theorem.

Theorem 2. *Given an algorithm with depth* $O(t/p)$ *for all* $p \leq x$ *in a F&** PRAM *with p processors and m common memory addresses where t, x and m are some numbers. We can simulate it in an* SDC *machine with s super-processors, m memory-processors and* $f(s, m)$ *comparator-processors in depth* $O(t/s)$ *for* $s \leq x/l(s, m)$.

One possible configuration (mentioned above) results in $O(s(\log s)^2 + m \log m)$ degenerate processors and $l(s, m) = (\log s)^2 + \log m$ where m is the size of the common memory. A second possible configuration, which uses the recently suggested sorting network of [2] enables us to replace the $(\log s)^2$ term by $\log s$ in the last two formulae. However, the constants involved in the second configuration are substantially larger.

7. Conclusion

A consequence of the use of pipelining in our efficient simulation where each super-processor simulates the behavior of several processors is the incentive to design optimal (or close to optimal) algorithms for a *much wider range of processors* (*virtual processors*) *than actually available* and to use the extensive 'library' of such known algorithms.

The result of this paper follows [9, 26, 29, 32] in supporting a more permissive design space as long as the fine for realizing it in feasible implementation spaces does not increase. For example, consider a CREW (concurrent-read, exclusive-write) PRAM (similar to [8]). It is a CRCW PRAM that does not allow simultaneous access to the same memory location by several processors for write purposes. One could expect that implementing algorithms given in a CREW PRAM into a synchronous distributed model, where each node has a small degree, will require less time or less processors than doing the same for algorithms given in a CRCW PRAM and that this 'ratio' will be even smaller for implementing the *F&** PRAM. By 'less' we mean less by more than a constant factor. As far as I know, every efficient solution for the first problem has corresponding solution for the second problem that preserves both the time and the number of processors, thus supporting the permissive models of computation. Moreover, in [32] it is *proven* that this situation holds for any 'reasonable' simulation of the CREW PRAM into the SDC.

Consider the case where the m common memory addresses are partitioned among N memory-processors where $N \leq m$ and only one address of each memory-processor may be accessed at a time. Let us overview an adaptation of the PDDI to this case which seems to approximate better current technological limitations (according to [14] and [9]). In addition to the sorting and merging network, our revised PDDI includes a balanced binary tree having p (the number of super-processors) leaves. Each leaf is connected to an output line of the sorting network (which is also an input line of the merging network). Inputs for the sorting network are triples of the form (m_i, a_i, i) where m_i is a memory-processor number and a_i is an address at its local memory (i is a super-processor number as before). The binary tree is used to queue up requests for distinct addresses of the same memory-processor by attaching them serial numbers. The simple details are left to the reader. These requests are pipelined into the merging network accordingly. Note that there is an unavoidable bottleneck due to the fact that only one address of each memory-processor can be referenced at a time. We do not elaborate on further details regarding this extension of our solution since no new ideas are involved. Finally, we would like to mention that Mehlhorn and Vishkin [18] suggested recently a few strategies to control these bottlenecks.

Appendix A. The $F\&*$ implementation tree

The following simple binary tree synchronous distributed machine provides for the implementation of the $F\&*$ PRAM. It is similar to the Fetch-and-Add implementation of [9]. Let n be a positive integer. For simplicity we assume that n is a power of 2. Let $\alpha = \log n$ (all logarithms in this paper are to the base of 2). Let T be a complete binary tree with n leaves. A processor is associated with every node in the tree. It is represented by a combination $[h, j]$, h being its height in the tree and j its serial number among the other nodes at the same height (see Fig. 3). Assume $i_1, i_2, \ldots, i_k$ are k numbers where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ and k is some integer.

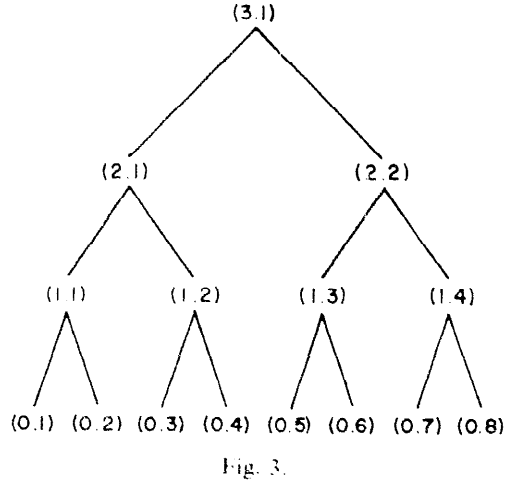

Fig. 3.

There are k numbers $b_{i_1}, b_{i_2}, \ldots, b_{i_k}$ which are associated with leaves $[0, i_1]$, $[0, i_2), \ldots, [0, i_k]$, and a number c associated with the root of the tree. As before, $*$ is any associative and commutative binary operation.

A neutral element for the $*$ operation is denoted by 0: for instance, the neutral element for the $+$ operation is the number zero. We use 0 to denote the number zero as well. No confusion will arise.

Every node in the tree is associated with two numbers, $A(h, j)$ and $B(h, j)$. The A numbers satisfy initially

$$A(i,j) = \begin{cases} b_j & \text{if } i = 0 \text{ and } j = i_l \text{ for some } l,\ 1 \leq l \leq k, \\ 0 & \text{otherwise.} \end{cases}$$

The B number of the root satisfies initially $B(\alpha, 1) = c$. Our goal is that the B numbers will satisfy eventually

$$B(0, i_j) = c * A(0, i_1) * A(0, i_2) * \cdots * A(0, i_{j-1}) = c * b_{i_1} * b_{i_2} * \cdots * b_{i_{j-1}}$$

$$\text{for } 1 \leq j \leq k,$$

and $B(\alpha, 1) = c * b_{i_1} * b_{i_2} * \cdots * b_{i_k}$.

The following (distributed) computation is performed 'up the tree' (from the leaves to the root).

$$A(h, j) \leftarrow A(h-1, 2j-1) * A(h-1, 2j) \quad \text{for } 0 < h \leq \alpha.$$

Immediately after this computation reaches the root we perform the following computation down the tree.

$$B(h, j) \leftarrow \begin{cases} B(h+1, \frac{1}{2}j) * A(h, j-1) & \text{if } j \text{ is even,} \\ B(h+1, \frac{1}{2}(j+1)) & \text{otherwise.} \end{cases}$$

Right after the computation leaves the root node its processor performs $B(\alpha, 1) \leftarrow B(\alpha, 1) * A(\alpha, 1)$.

Note that the computation time is proportional to the height of the tree, i.e., $O(\log n)$.

The V_j' trees which are obtained by the simulation are binary but not complete. Adapting the computation described above to these trees is very simple. Observe that node-processors which are not on a shortest path from a root to an active leaf (a leaf of the form $[0, i_l]$ for some l, $1 \leq l \leq k$) do not participate in the computation. A node-processor of the V_j' tree that has one son is treated as if it is a left son.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[2] M. Ajtai, J. Komlos and E. Szemeredi, An $O(n \log n)$ sorting network, *Proc. 15 ACM Symp. on Theory of Computing* (1983) 1–9.

[3] K.E. Batcher, Sorting networks and their applications, *Proc. AFIPS Spring Joint Computer Conference* **32** (1968) 338–344.

[4] C. Berge, *Graphs and Hypergraphs* (North-Holland, Amsterdam, 1973).
[5] A. Borodin and J.E. Hopcroft, Routing, merging and sorting on parallel models of computation, *Proc. 14 ACM Symp. on Theory of Computing* (1982) 338–344.
[6] F.Y. Chin, J. Lam and I. Chen, Efficient parallel algorithms for some graph problems, *Comm. ACM* **25** (9) (1982) 659–665.
[7] D.M. Eckstein, Simultaneous memory access, TR-79-6, Computer Science Dept., Iowa State University, Ames, 1979.
[8] S. Fortune and J. Wyllie, Parallelism in random access machines, *Proc. 10th ACM Symp. on Theory of Computing* (1978) pp. 114–118.
[9] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir, The NYU ultracomputer—designing an MIMD shared memory parallel machine, *IEEE Trans. Comput.* **C-32** (2) (1983) 175–189.
[10] Z. Galil and W.J. Paul, An efficient general purpose parallel computer, *J. ACM* **30** (2) (1983) 360–387.
[11] D. Heller, A survey of parallel algorithms in numerical linear algebra, *SIAM Rev.* **20** (4) (1978) 740–777.
[12] D.S. Hirschberg, Fast parallel sorting algorithms, *Comm. ACM* **21** (8) (1978) 657–661.
[13] D.S. Hirschberg, A.K. Chandra and D.V. Sarwate, Computing connected components on parallel computers, *Comm. ACM* **22** (8) (1979) 461–464.
[14] D.J. Kuck, A survey of parallel machine organization and programming, *Comput. Surveys* **9** (1) (1977) 29–59.
[15] G. Lev, Size bounds and parallel algorithms for networks, Ph.D. Thesis, Rept. CST-8-80, Dept. of Computer Science, University of Edinburgh, 1980.
[16] G. Lev, N. Pippenger and J.G. Valient, A fast parallel algorithm for routing in permuting networks, *IEEE Trans. Comput.* **C-30** (2) (1981) 93–100.
[17] N. Megiddo, Applying parallel computation algorithms in the design of serial algorithms, *Proc. 22nd IEEE Symp. on Foundations of Computer Science* (1981) 399–408.
[18] K. Mehlhorn and U. Vishkin, Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories, *Acta Informatica*, to appear.
[19] D. Nath, S.N. Maheshwari and P.C.P. Bhatt, Parallel algorithms for the convex hull problem in two dimensions, *Proc. CONPAR 81*, Lecture Notes in Computer Science **111** (Springer, Berlin, 1981) 358–372.
[20] F.P. Preparata, New parallel-sorting schemes, *IEEE Trans. Comput.* **C-27** (1978) 669–673.
[21] W. Paul, U. Vishkin and H. Wagener, Parallel computation on 2–3-trees, TR-70, Dept. of Computer Science, Courant Institute, New York Univ., NY, 1983; in: *Proc. 10th ICALP*, Lecture Notes in Computer Science **154** (Springer, Berlin, 1983) 597–609.
[22] J.T. Schwartz, Ultracomputers, *ACM Trans. Programming Languages and 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] Y. Shiloach and U. Vishkin, An O(log n) parallel connectivity algorithm, *J. Algorithms* **3** (1) (1982) 57–67.
[25] Y. Shiloach and U. Vishkin, An O(n**2 log n) parallel max-flow algorithm, *J. Algorithms* **3** (2) (1982) 128–146.
[26] L.J. Stockmeyer and U. Vishkin, Simulation of parallel random access machines by circuits, *SIAM J. Comput.* **13** (2) (1984) 409–422.
[27] C.D. Thompson, The VLSI complexity of sorting, *IEEE Trans. Comput.* **C-32** (12) (1983).
[28] R.E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.*, to appear.
[29] U. Vishkin, Implementation of simultaneous memory address access in models that forbid it, *J. Algorithms* **4** (1) (1983) 45–50.
[30] U. Vishkin, An optimal parallel connectivity algorithm, RC 9149, IBM J.J. Watson Research Center, Yorktown Heights, NY, 1981; *Discrete Appl. Math.*, to appear.
[31] U. Vishkin, Lucid-boxes vs. black-boxes, Preprint, 1983.
[32] U. Vishkin, On choice of a model of parallel computation, TR-61, Dept. of Computer Science, Courant Institute, New York Univ., 1983; *J. Comput. Systems Sci.*, to appear.
[33] J.C. Wyllie, The complexity of parallel computation, TR 79-387, Dept. of Computer Science, Cornell University, Ithaca, NY, 1979.