INFORMATION AND CONTROL 56, 174–182 (1983)

Dynamic Parallel Memories

UZI VISHKIN

Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012

AND

AVI WIGDERSON

Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720

Say that a parallel algorithm that uses p processors and N ($>p$) shared memory locations is given. The problem of simulating this algorithm by p processors and only p shared memory locations without increasing the running time by more than a constant factor is considered. A solution for a family of such parallel algorithms is given. The solution utilizes the idea of dynamically changing locations of the addresses of the algorithm throughout the simulation.

1. INTRODUCTION

The current state of technology implies that memories which include many cells must be partitioned into a number of modules each containing many cells; where, only one cell (or a small number of cells) of each module can be accessed at a time. For more on this, see Kuck (1977) and Gottlieb *et al.* (1983). On the other hand, many published parallel algorithms are designed for abstract shared-memory models of parallel computation, where the processors have free access to each cell of the shared memory for both read and write purposes. An obvious difficulty arises when one wants to simulate these algorithms on buildable machines. One approach is to require that designers of algorithms (for abstract shared-memory models of parallel computation) limit, as much as possible, the size of the shared memory that the algorithm must use. This is usually done in favor of more local computations in which each processor accesses its own local memory only. Kuck mentions several papers that practiced this ad hoc approach. Even in cases where such a limitation is possible this approach puts some undesirable additional burden on the designer.

0019-9958/83 $3.00

Let us be a bit more precise. Given a shared memory model of parallel computation D we define $M(D)$ to be the model of computation which is derived from D by partitioning the shared memory of D into modules so that no more than one cell of each module can be accessed at a time. If there are several simultaneous requests for the same common memory location in $M(D)$ they are treated in the same way as in D. If there are several simultaneous requests for different cells of the same module, they are queued and responded one at a time.

The *granulrity problem* is defined as the problem of simulating a cycle of D by $M(D)$. Automatic solutions for the general case where we do not know anything about the cycle to be simulated are discussed in Mehlhorn and Vishkin (1983). They suggest a multi-stage approach for attacking the granularity problem. We mention the two main stages. The first stage designed to keep us "out of trouble," in the average case, utilizes universal hashing in the simulating machine $M(D)$. $M(D)$ itself picks at random a hashfunction from an entire class of hashfunctions before each simulation of an algorithm, instead of a specific hashfunction. This is shown to keep memory contention low. The idea behind the second stage is to keep several copies of each memory address in distinct memory modules. This idea, in conjunction with fast algorithms for picking the "right" copy of each requested address is shown to decrease memory contention for the worst case results of the first stage.

The main result of the present paper is that in a few general cases the idea of dynamically changing locations of addresses among modules throughout the performance of an algorithm provides a solution for the granularity problem in constant time utilizing only as many modules as the number of processors.

2. A RELATION BETWEEN MODELS OF PARALLEL COMPUTATION

The main model of parallel computation that is used in the present paper is the exclusive-read exclusive-write parallel random-access machine (EREW PRAM). It employs p processors (RAMs) $P_1,\ldots,P_p$ that operate synchronously in parallel. Eeach processor has access to both a shared memory of size N and its private local memory. Simultaneous access of more than one processor to the same memory location is not allowed. At each cycle a processor may either perform an operation that relates to its local memory or read from a shared memory address or write into a shared memory address. The convention of not allowing simultaneous access by several processors to the same memory location is used in Lev, Pippenger, and Valiant (1981). This model is a member in a whole family of shared-memory parallel RAM models of computation. We refer the reader to

Stockmeyer and Vishkin (1982) for a formal definition of these models including the list of operations they allow and to Vishkin (1983) for a recent survey of results concerning them.

A second model of computation that we employ is the module parallel machine (MPM). It employs r processors $R_1, R_2, \ldots, R_r$ and is similar to the EREW PRAM with the following exception. The L cells of the shared memory are partitioned among m modules. Only one cell of each module can be accessed at any cycle of the MPM. In both models of computation the program for each processor is located in its local memory.

How do these models relate?

(1) Every algorithm for an EREW PRAM that employs p processors and shared memory of size N can be run on an MPM using p processors and N nonempty modules. This trivial observation follows readily by employing one memory cell at each module of the MPM.

(2) Suppose that we are given an algorithm for the MPM that employs p processors and m shared memory modules; suppose that module i, $1 \leqslant i \leqslant m$, contains N_i cells; suppose that $m \leqslant p$ and the algorithm runs in (at most) T cycles. This algorithm can be simulated by the EREW PRAM in $O(T)$ cycles using p processors, shared memory of size m and the local memory that is used by processor P_i, $1 \leqslant i \leqslant m$ (resp. $m < i \leqslant p$), of the EREW PRAM is greater by N_i than (resp. is the same as) the local memory of processor R_i of the MPM.

The rest of this section is devoted to outline how this is done. Processor P_i is "responsible" for simulating the behavior of processor R_i, for $1 \leqslant i \leqslant p$. In addition, Processor P_i is "responsible" for simulating the behavior of module i, $1 \leqslant i \leqslant m$. For the latter purpose each cell of module i of the MPM is represented by a corresponding cell in the local memory of processor P_i, $1 \leqslant i \leqslant m$.

The simulation proceeds as follows. Each cycle t, $1 \leqslant t \leqslant T$, of the MPM is simulated by three pulses of the EREW PRAM denoted $(t, 1)$, $(t, 2)$, and $(t, 3)$.

Pulse $(t, 1)$:
If R_i performed, at cycle t, an operation that relates to its local memory only
Then P_i does the same with respect to its local memory
Else If R_i performed a read instruction from cell j of shared memory module l
Then P_i writes into shared memory cell l:
"cell j is requested"
Else If R_i wrote some value v into cell j of shared memory module l
Then P_i writes into shared memory cell l:
"write v into cell j"

Pulse $(t, 2)$:
(Only processors P_i, $1 \leqslant i \leqslant m$, are active)
If shared memory cell i contains: "cell j is requested"
Then P_i copies the contents of its local memory address which corresponds to cell j of module i of the MPM into common memory cell i.
Else If shared memory cell i contains:
"write v into cell j"
Then P_i copies v into its local address corresponding to cell j of module i

Pulse $(t, 3)$:
If R_i performed at cycle t a read instruction from a cell of module l
Then P_i reads the contents of shared memory cell l

Proofs of correctness of this simple simulation and our claims regarding time and space complexity are straightforward.

3. Reducing the Size of the Shared Memory

Suppose we are given an algorithm (designed for the EREW PRAM) which employs p processors, uses N shared memory locations and runs in T cycles for some input. Suppose $p \ll N$. *Question.* Is it possible to simulate this algorithm on an EREW PRAM that employs the same number of processors and "significantly" less than N shared memory cells, without increasing the running time "too much?"

The following fact gives some hope: Since there are p processors, no more than p shared-memory addresses may be accessed at the same time.

Before we proceed to our main theorem, we would like to say the following regarding the most general case.

Remark. In general, using a shared memory of size $O(pT)$ should suffice. The reason for this is that we can maintain all shared memory cells which are actually being accessed in the course of the algorithm in 2–3 trees. A processor may initialize only one cell at a time. Therefore, the number of shared memory cells that can be initialized is $O(pT)$. The paper Paul, Vishkin, and Wagener (1983) shows how to perform the search and insertion operations that may be required for the simulation of one cycle of the algorithm in $O(\log pT)$ time of the simulating (EREW PRAM) machine.

Main Theorem. *Let S be a program for an* EREW PRAM *which is designated for some set of inputs I. Suppose S uses p processors, N shared-memory locations, local memories of sizes $m_1, m_2, \ldots, m_p$ of respective processors, and runs in at most T cycles for each input in I. Assume that for*

each cycle t, $1 \leqslant t \leqslant T$, each of the p processors and all inputs in I there is at most one common memory address that can be accessed by this processor at this cycle. Then, a program S' for an EREW PRAM *can be constructed from S such that S' simulates S for each input in I using p processors, only p shared-memory locations, $m_i + \lceil N/p \rceil + O(T)$ $(1 \leqslant i \leqslant p)$ local memory locations of respective processors, and $O(T)$ pulses.*

Before we proceed to the proof we would like to discuss the significance of our theorem. First, observe that the assumptions of the theorem are readily satisfied if the cardinality of I is one. This is simply because an execution of a parallel program on some input x results in at most one common memory access at a time by each processor! *Problem.* Find instances where "common memory access patterns" of a program S, for a set of inputs I, are the same (or about the same) for all the inputs in I.

It turns out that researchers in the field of numerical computations identified the notion of serial straight-line programs, which characterizes many of the known programs for problems in this field. For a definition of serial straight-line programs see Aho, Hopcroft, and Ullman (1974, Sect. 1.5). Serial straight-line programs for inputs of size n do not include branching, loops, or indirect addressing. Therefore, for all inputs of size n and for each time unit of such a program the same registers are being accessed.

Heller (1978) includes references to numerous numerical parallel algorithms. Many of these algorithms satisfy such "uniform" (local and common) memory access pattern property including algorithms for evaluating arithmetic expressions of a given format (see Winograd, 1975), the "naive" matrix multiplication, the "naive" raising of an $n \times n$ matrix to the nth power (in particular, transitive closure; see Savage and Ja'Ja', 1981), and others. So, our theorem is applicable to these programs. Note, however, that for our theorem we may dispense with the uniform local memory access pattern property and ease a little the uniform common memory property; as long as no more than one common memory address can be accessed for each processor and each $1 \leqslant t \leqslant T$.

Proof of the Theorem. Let us call the time units of S cycles and the time units of S' pulses. Assume, w.l.g., that N/p is an integer. Otherwise, we "add" some dummy common memory addresses in order to increase N to the next multiple of p. Let $P_1, P_2, \ldots, P_p$ (resp. $R_1, R_2, \ldots, R_p$) be the processors of the EREW of S (resp. S'). Let x_{k_i}, $1 \leqslant i \leqslant m_k$, be the local registers of processor P_k, $1 \leqslant k \leqslant p$, and w_j, $1 \leqslant j \leqslant N$, be the common memory locations of S. We set x'_{k_i}, $1 \leqslant i \leqslant m_k$, to be local memory locations of processor R_k which correspond, respectively, to local registers x_{k_i}, $1 \leqslant i \leqslant m_i$, of processor P_k, for $1 \leqslant k \leqslant p$. Let u_j, $1 \leqslant j \leqslant p$, be the common memory locations of S'. Set y_{k_j}, $1 \leqslant k \leqslant p$, $1 \leqslant j \leqslant N/p$, to be local registers of processor R_k (in addition to $x'_{k_1}, x'_{k_2}, \ldots$).

Generally speaking, we design S' in such a way that processor R_k simulates the behavior of processor P_k, $1 \leqslant k \leqslant p$; each local memory location x'_{k_j} simulates x_{k_j}; and the locations of the form u_j and y_{k_j} simulate the w_i locations. The additional $O(T)$ local memory locations are required for the code of S' as explained at the end.

By our assumption no more than p of the w_i locations may be accessed at each cycle of S. Denote the w_i locations which may be accessed at cycle t, $1 \leqslant t \leqslant T$, by $v_{t_1}, v_{t_2}, \ldots, v_{t_p}$. (In a cycle where less than p w_i's may be accessed we set some of these u_{t_j}'s to represent w_i's which are not accessed by any processor for any input during this cycle). In any case, the locations $v_{t_1}, v_{t_2}, \ldots, v_{t_p}$ are p distinct common memory locations.

A High-Level Description of S'

The following condition is satisfied just before the simulation of cycle t, $1 \leqslant t \leqslant T$, of S by S' starts:

(*) Each processor R_k, $1 \leqslant k \leqslant p$, keeps the content of exactly one of the variables $v_{t_1}, v_{t_2}, \ldots, v_{t_p}$ in a local memory location of the form y_{k_i}; and no more than N/p of the w_i locations of S are stored in the local memory of each processor R_k, $1 \leqslant k \leqslant p$.

Every cycle t of S is simulated by S' in three pulses:

(1) The *fetch pulse.* Processor R_k which keeps the contents of variable v_{t_j} in its y_{k_i} local variable assigns it into u_j.

(2) The "*real-thing*" *pulse.*

If processor P_k performs in cycle t an instrution which relates
to its local registers only (or remains idle)
Then processor R_k does the same with respect to its
corresponding x'_{k_i} registers
Else (processor P_k performs an instruction of the form:
$x_{k_i} \leftarrow v_{t_j}$-read from memory, or
$v_{t_j} \leftarrow x_{k_i}$-write into common memory)
processor R_k performs the same replacing v_{t_j} by u_j
and x_{k_i} by x'_{k_i}.

(3) The *store pulse.* Processor R_k copies the contents of some (one) u_j into one of its y_{k_i} variables so that condition (*) will hold for every cycle which follows.

The remainder of the proof is devoted to showing that there is a way to partition initially the w_i's among the local memories of the R_k processors, and perform the store pulses of all cycles such that condition (*) is satisfied before the simulation of each cycle. This is done by reducing our problem to an edge coloring problem on a bipartite graph.

Consider an auxiliary digraph G which is defined as follows.

(a) It has $(T + 2N/p) \times p$ vertices. $T \times p$ of these vertices represent the common memory locations v_{t_j}, $1 \leqslant t \leqslant T$, $1 \leqslant j \leqslant p$. N of these vertices, denoted v_{t_j}, $-(N/p) + 1 \leqslant t \leqslant 0$, $1 \leqslant j < p$, represent each of the w_i, $1 \leqslant i \leqslant N$. They are called *input vertices*. The last N vertices denotes v_{t_j}, $T + 1 \leqslant t \leqslant T + N/p$, $1 \leqslant j \leqslant p$, represent also each of the w_i, $1 \leqslant i \leqslant N$. They are called *output vertices*.

(b) There exists an edge of the form $v_{t_j} \to v_{s_l}$ if

(1) both v_{t_j} and v_{s_l} stand for the same w_i; and

(2) $s > t$ and there is no v_{r_h} such that $s > r > t$ and v_{r_h} stands for w_i.

It should be obvious that the out-degree of each v_{t_j}, $-(N/p) + 1 \leqslant t \leqslant T$, $1 \leqslant j \leqslant p$, is one and the out-degree of the other vertices is zero, and the in-degree of each v_{t_j}, $1 \leqslant t \leqslant T + N/p$, $1 \leqslant j \leqslant p$ is one and the in-degree of the other vertices is zero.

Layer t of G (L_t in short) is the set $\{v_{t_j} \mid 1 \leqslant j \leqslant p\}$, $-(N/p) + 1 \leqslant t \leqslant T + N/p$. The correspondence between layers 1, 2,..., T and cycles should be obvious.

Our solution assigns each edge of the form $v_{t_j} \to v_{s_l}$ to a processor R_k. This implies that processor R_k stores the content of u_j into a local variable of the form y_{k_a} at the store pulse of cycle t (if $t \leqslant 0$, then a y_{k_a} variable contains the input value that corresponds to v_{t_j}); later, at the fetch pulse of cycle s processor R_k assigns the content of y_{k_a} into u_l (if $s > T$, then a y_{k_a} variable contains the output value that corresponds to v_{s_l}).

In order to satisfy the $(*)$ condition throughout the simulation it is readily sufficient to do the following. Partition the edges of G into p sets $C_1, C_2,..., C_p$ such that for any two edges e_1 and e_2 of the same set both: (1) tail(e_1) and tail(e_2) belong to different layers, and (2) head(e_1) and head(e_2) belong to different layers. This partitioning enables us to associate each of these sets with a processor which will do the work corresponding to edges of this set.

Still, a further simplification of the problem is possible. Consider another auxiliary graph H; a bipartite undirected graph. Note that H may include parallel edges. Let $\{a_1, a_2,..., a_{T+N/p}\}$ and $\{b_{-(N/p)+1},..., b_1\}$ be the two disjoint sets of vertices of H. The connection to the digraph G becomes clear through the definition of the edges of H. There is a one-to-one correspondence between the edges of G and the edges of H. Let $v_{t_j} \to v_{s_l}$ be an edge of G. Then, the corresponding edge in H is of the form (b_t, a_s). Our edge partitioning problem for G translates into the following edge partitioning problem for the undirected graph H. Partition the edges of H into p sets such that no two edges of the same set share an end point.

This is the well-known edge coloring problem for a bipartite graph. Since the degree of each vertex in H is not greater than p, a known theorem (see Ore, 1967) implies that it is possible to partition the the edges of H into p sets as required.

Algorithms that achieve this partitioning: We refer the reader to Gabow and Kariv (1982) for sequential algorithms and Lev *et al.* (1981) for parallel algorithms.

We would like to ascertain that the proof of the theorem is completed. The set (color) of the edge in H corresponding to an edge of the form $v_{t_j} \to v_{s_l}$, where $-(N/p) + 1 \leqslant t \leqslant 0$, yields a processor R_k. Now, the contents of the w_r that corresponds to this edge is initially in one of its y_{k_a} locations. We need exactly (N/p) y_{k_a} locations, for $1 \leqslant k \leqslant p$, for this initialization. At each fetch pulse of the simulation of a cycle t, $1 \leqslant t \leqslant T$, we "release" one y_{k_l}, $1 \leqslant k \leqslant p$. This released y_{k_l} can be used to store the w_i that has to be stored by processor R_k as a result of the store pulse that follows, $1 \leqslant k \leqslant p$. The introduction of the v_{t_j}'s, $T + 1 \leqslant t \leqslant T + N/p$, gives actually an "equal" partition of the outputted w_i's which was not "promised" in the theorem. They are not necessary for the proof.

For each cycle t, $1 \leqslant t \leqslant T$, the code of S' at each procesor R_k must specify the y_{k_l} to be released and reoccupied, the u_a into which this y_{k_l} is copied in the fetch pulse, the u_b (if any) that may be accessed in the real-thing pulse and the u_c copied into y_{k_l} in the store pulse. Thus, the code of S' is longer by $O(T)$ than the code of S at each local memory.

Extensions

All the results in this paper can be extended in a straightforward manner to more permissive models of parallel computation where simultaneous access of several processors to the same memory location is allowed; in particular, the powerful concurrent-read concurrent-write (CRCW) PRAM allows several processors to read (or write) simultaneously from (into) the same memory location. See Stockmeyer and Vishkin (1982) for more on these models of computation.

Acknowledgment

A referee remarked the need for the additional $O(T)$ local memories in the main theorem. We are grateful for this remark.

Received: July 22, 1983; accepted: October 19, 1983

REFERENCES

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, Mass.

GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLP, L., AND SNIR, M. (1983), The NYU ultracomputer—Designing a MIND shared memory parallel machine, *IEEE Trans. Comput.* **C32** (2), 175–189.

GABOW, H. N., AND KARIV, O. (1982), Algorithms for edge coloring bipartite graphs and multigraphs, *SIAM J. Comput.* **11** (1), 117–129.

HELLER, D. (1978), A survey of parallel algorithms in numerical linear algebra, *SIAM Rev.* **20** (4), 740–777.

KUCK, D. J. (1977), A survey of parallel machine organization and programming, *Comput. Survey* **9** (1), 29–59.

LEV, G., PIPPENGER, N., AND VALIANT, J. G. (1981), A fast parallel algorithm for routing in permutation networks, *IEEE Trans. Comput.* **C30** (2), 93–100.

MEHLHORN, K., AND VISHKIN, U. (1983), Granularity of parallel memories, TR-89, Department of Computer Science, Courant Institute, New York Univ., New York; for an extended abstract see Granularity of shared memory in parallel computation, *in* "Proceedings, 9th Workshop on Graphtheoretic Concepts in Computer Science (WG-83)," Fachbereich Mathematic, Universitat Osnabruck, June, in press.

ORE, O. (1967), "The Four Color Problem," Academic Press, New York.

PAUL, W., VISHKIN, U., AND WAGENER, H. (1983), Parallel dictionaries on 2–3 trees, *in* "Proceedings, 10th ICALP," Lecture Notes in Computer Science, Vol. 154, pp. 597–609, Springer-Verlag, Berlin/New York.

SAVAGE, C., AND JA'JA', J. (1981), Fast, efficient parallel algorithms for some graph problems, *SIAM J. Comput.* **10** (4), 682–691.

STOCKMEYER, L. J., AND VISHKIN, U. (1982), Simulation of parallel random access machines by circuits, RC 9362, IBM T. J. Watson Research Center, Yorktown Heights, New York; *SIAM J. Comput.*, in press.

VISHKIN, U. (1983), Synchronous parallel computation—A survey, TR-71, Department of Computer Science, Courant Institute, New York Univ., New York.

WINOGRAD, S. (1975), On the parallel evaluation of certain arithmetic expressions, *J. Assoc. Comput. Mach.* **22** (4), 477–492.

G
par

INFORMATION AND CONTROL **56**, 183–198 (1983)

Succinct Representations of Graphs

HANA GALPERIN*

Mini Systems Inc., Herzlia, Israel

AND

AVI WIGDERSON*,†

Computer Science Division, University of California, Berkeley, California 94720

For a fixed graph property Q, the complexity of the problem: Given a graph G, does G have property Q? is usually investigated as a function of $|V|$, the number of vertices in G, with the assumption that the input size is polynomial in $|V|$. In this paper the complexity of these problems is investigated when the input graph is given by a succinct representation. By a succinct representation it is meant that the input size is polylog in $|V|$. It is shown that graph problems which are approached this way become intractable. Actually, no "nontrivial" problem could be found which can be solved in polynomial time. The main result is characterizing a large class of graph properties for which the respective "succinct problem" is NP-hard. Trying to locate these problems within the P-Time hierarchy shows that the succinct versions of polynomially equivalent problems may not be polynomially equivalent.

1. INTRODUCTION

The design of efficient algorithms for graph theoretic problems is a major research area in recent years. The word "efficient" generally means that the amount of computing resources is minimized. One of the ways considered frequently is the use of complex data structures in algorithms, while the assumption is made that the input is given by some conventional representation. Traditionally, graphs are represented by either adjacency matrices or adjacency lists with representation size of $O(|V|^2)$ and $O(|E|)$, respectively. For graphs that are relatively small this is perfectly acceptable, but when we deal with graphs that have a huge number of vertices the conventional representations are quite costly. In the areas of architectural design systems

* This research was conducted when the authors were in the EECS Department at Princeton University, Princeton, N. J.

† The author was partially supported by NSF Grant ENG76-16808 and DARPA Contract N0039-82-C-0235.