Discussion Of Parallel Algorithms

Abha Moitra†
S. Sitharama Iyengar‡

TR 86-759
June 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

† Department of Computer Science, Cornell University, Ithaca, NY 14853
‡ Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803.

DISCUSSION OF PARALLEL ALGORITHMS

Abha Moitra
Department of Computer Science
Upson Hall
Cornell University
Ithaca, New York 14853

S. Sitharama Iyengar
Department of Computer Science
Coates Hall
Louisiana State University
Baton Rouge, LA 70803

Abstract

In recent years we have witnessed a tremendous surge in the availability of very fast and inexpensive hardware. However, our ability to design fast and cheap hardware far outstrips our ability to utilize them effectively in solving large problems fast. Hence there is a continuing interest in the study and development of parallel algorithms. In this paper we present a survey of deterministic parallel algorithms for a class of computational problems. Both graph-theoretic and non graph-theoretic problems are considered and the parallel algorithms presented are motivated by identifying some common paradigms.

TABLE OF CONTENTS

1. Introduction

In recent years we have witnessed a tremendous surge in the availability of very fast and inexpensive hardware. These have been made possible partly by the use of faster circuit technologies and smaller feature sizes, partly by novel architectural features such as pipelining, vector processing, cache memories, systolic arrays, etc., and partly by using novel interconnections between processors and memories such as Hypercube, Omega network, Orthogonal Tree network and others.

However, our ability to design fast and cheap hardware far outstrips our ability to utilize them effectively in solving large problems fast. This is due mainly to the fact that a large problem may not be easy to decompose into smaller problems which could be solved in parallel, on account of data dependencies between the subproblems. The intrinsic parallelism of a problem can be defined as the product of the time required by the fastest parallel algorithm for solving it, and the number of processors required by that algorithm, divided by the time required by the best sequential algorithm. For a problem with high internal data dependency, the intrinsic parallelism would be very low.

Because of these data dependencies, processes which are solving related subproblems would need to communicate with each other. This communication could be via shared memory (with or without concurrent access), via a data bus or via an interconnection network. Any specific scheme incurs an overhead in terms of time lost to contention and/or latency, and also to hardware costs which do not reduce dramatically with improved technology, like that of the underlying circuits. Further, a communication scheme which is good for one class of problems may not be good for another class, which only adds to our difficulty in finding a parallel algorithm.

Memory contention can slow down the execution of a parallel algorithm, if the various processing elements must access the same variable at the same time; some systems must be devised so that only one processor can access a given variable at any one time (for example, LOCK and UNLOCK on MIMD machines). Also, if the number of logical processors is larger than the number of physical processors in the machine, some sort of scheduling must be done to determine where the extra processes will eventually be handled. The scheduling cost is the resource allocated to do this scheduling. For efficient scheduling, the extra logical processes should be saved until a processor is available, and the internal state of the logical processes should be monitored.

One factor which can vary greatly between machines and influences execution speed is the cost of creating a new process. For example, if a single process created all the other processes, then the cost of constructing p processes is $O(p)$. On the other hand, if new processes are created dynamically by other existing processes, then in the best case the construction cost of p processes is $O(\log p)$ if process creation corresponds to a binary tree configuration.

Examination of the above factors suggests two paradigms which can help improve parallel algorithms. The first is to design an algorithm with a large grain size and a smaller number of processes (grain size refers to the amount of processing done by an individual process). The second suggests that the optimal number of processes is a compromise between the gain provided by each process and the time lost to synchronization.

1.1 Models Of Computation

The design of parallel algorithms becomes an important issue as numerous parallel architectures are developed. In fact, a considerable number of very different architectures for parallel computing are in existence - ranging from special purpose array processors, to tree machines, to loosely coupled networks of processors. Since the design and performance of a parallel algorithm depends very much on the architecture of the parallel machine, it is necessary to keep the architecture in mind when designing parallel algorithms. There is however no universal method for designing parallel algorithms. One approach to constructing parallel algorithms is to recognise parallelism in the existing sequential algorithms. This approach has been studied by different researchers (Keller 1973, Lee et al. 1985, Moitra, Iyengar 1986, Nicolau 1985, Shrira et al. 1983, Strom, Yemini 1985). We will therefore first examine briefly some parallel machine models of computations that have been proposed.

1.1.1 SIMD Machine

SIMD (Single Instruction Multiple Data) machine consists of an array of identical processors. All processors simultaneously execute the same instruction supplied by the control unit possibly on different data items. The execution of the instructions is synchronous in the sense that each processor executing the instruction in parallel must be allowed to finish before the next instruction is taken up for execution.

Different models are obtained depending on whether the two types of conflicts are allowed or not. In SIMD-SM model reading and writing conflicts are not allowed; in SIMD-SM-R simultaneous reading (but not writing) is allowed. In SIMD-SM-RW, simultaneous reading and writing is allowed; in case a number of processor are attempting to write to the same location different assumptions can be made about which process succeeds in writing. In the equality-conflict-resolution, simultaneous writing is permitted only when all processors attempt to store the same value. In priority-conflict-resolution, all the processors are linearly ordered according to some priority and the processor with the highest priority attempting to write succeeds. In arbitrary-conflict-resolution, any arbitrary processor attempting to write succeeds.

1.1.2 MIMD Machine

MIMD stands for Multiple Instruction Multiple Data and have the following characteristics. They consist of *processing elements* (*PEs*) which may be individually indexed. Each PE is capable of performing the standard arithmetic and logical operations. In addition each PE knows its index and has some local memory.

The PEs operate asynchronously under the control of individual instruction streams. Different PEs can execute different instructions at any time. During the computation PEs communicate results to each other. In many MIMD models the time required to communicate data from PE to PE often dominates the overall complexity of the algorithm. Several interprocessor communication models for MIMD computers have been proposed in the literature. The communication overhead of an algorithm varies from one communication model to the other, however the shared memory model (SM) is a well studied model. This model has no communication delay. In a shared memory computer there is a large common memory that is shared by all PEs. It is assumed that any PE can access any word in common memory in $O(1)$ time. When two or more PEs read (write to) from the same location simultaneously, we say that a read (write) conflict has occurred.

In the MIMD-TC model, the processors are attached tightly through a central switching mechanism to reach the global memory. Stone (1980) states that architectures based on the MIMD-TC model have a limited number of processors because of the cost involved in the switching process. An example of this architecture is C.mmp which has sixteen processors.

1.1.3 Tree Machine

The tree machine of Bentley, Kung (1979) consists of $O(n)$ processors. These processors are of three types. Processor type 1 is capable of broadcasting data they receive and the processor type 2 can combine their inputs in an elementary way. Processor type 3 stores program and data of size $O(n)$.

1.1.4 Special Purpose Parallel Machines

The advent of parallel processing (specifically VLSI) has led to the development of a number of special purpose parallel machines to support directory structures. Leiserson (1979), Bentley, Kung (1979), Ottaman et al. (1982) and Atallah, Kosaraju (1985) proposed pipelined architectures based on a balanced binary tree. $O(n)$ PEs are used in order to support a search tree containing up to n elements. The machines vary in their wiring complexity and variety of of dictionary operations that they support. Generally all of them can perform the basic dictionary operations in $O(\log n)$ time. The complexity of the dictionary operation varies on the input pipeline intervals. Atallah, Kosaraju (1985) machine provide $O(\log n)$ performance with a pipeline interval of $O(1)$ for a wide range of dictionary operations (n is the actual number of elements stored in the $O(n)$ PEs machine). The above designs maintain the dictionary elements in some sorted order. Somani, Agarwal (1984) propose a binary tree machine with $O(n)$ PEs that does not require any sorted order for the dictionary elements. Their design supports all dictionary operations and provides an $O(\log n)$ time performance with constant pipeline interval. Tanaka et al. (1980) propose a pipelined architecture for maintaining search tree of n elements with only $O(\log n)$ PEs. Fisher (1984) developed an architecture based on the Trie structure. In this design the number of PEs is proportional to the length of the maximum key and this scheme is advantageous when the dictionary keys are long. These pipelined architectures achieve only an $O(\log n)$ throughput improvement over the serial balanced tree algorithms. When the number of records in the dictionary becomes greater than n these designs will not function i.e., the hardware is tailored to the maximal possible number of elements in the tables. While the $O(n)$ PEs architectures can handle efficiently operations beyond basic dictionary functions they have no advantage over the $O(\log n)$ designs when only the basic operations are considered.

1.2 Techniques For Exploiting Parallelism In Algorithms

In most sequential algorithms, some degree of parallelism exists; it is the task of the programmer to exploit it in the most efficient way. Since it is often the case that only one parallelization technique can be applied to an algorithm at a time, a careful examination of the algorithm must be made before a technique can be chosen. The speed up ratio, s, used for comparing the sequential algorithm with the parallel algorithm, is the time required for the sequential execution divided by the time required for parallel execution. In the ideal case, s would be equal to the number of processors used in the parallel execution.

Data partitioning is one technique which can be applied to many algorithms. Data can be partitioned in two ways : statically and dynamically. In static partitioning, the data to be processed is partitioned into p processes, where p is the number of processes. Each processor handles one group of data, and each processor shares the same code. This type of partitioning is efficiently implemented on a SIMD machine. Dynamic partitioning allows data to be subdivided between active processors, as in the tree machine.

In some cases, computational partitioning can be performed on an algorithm. If several subprograms must be performed, and each of these subprograms is independent of the other subprograms, then each can be assigned its own processor. These processors have their own code and execute independently, at their own speed. Moitra (1986) discusses a general technique for data and computation partitioning for obtaining distributed programs.

If several processors must make changes on a certain variable, memory contention can be minimized through localization. To decrease the access to these shared variables, a copy of the variable is made in the memory local to a processor. The processor uses this local variable to do its computations, then, if necessary, the global variable is updated at the end of the subprogram. It is possible, in some cases, to relax the control mechanism so that the parallel processor system is working asynchronously instead of synchronously. This method is called *relaxation*, and involves replacing explicit synchronization with implicit synchronization to provide more flexibility.

Software pipeline is a software implementation of the pipeline machine discussed earlier. Each processor behaves like a pipeline segment, with items being passed down the line from one process to another. An item is said to be consumed by process i when process i is finished with it; when an item is consumed by process i, an item is produced for process $i + 1$. The output rate of the pipeline is dependent on the time spent in the slowest segment of the pipe, since this forms a sort of bottleneck behind which other items must wait before they can be processed. It is therefore wise to divide the algorithm into a number of processes that require approximately the same execution time. Synchronization is done by software also, most often using shared variables. When a process has consumed an item, it may set a flag in the shared memory which the next process checks when it is free. If this flag is not set, meaning that the previous processes are not through with the item, the process continues checking the flag; essentially waiting until the item is available for processing. Synchronization cost can be said to be $O(1)$, since only two processes need to read or write into a given cell.

To sum up, it can be said that the amount of parallelism which can be achieved is dependent on all the above factors, with emperical evidence indicating that the granularity of the processes and communication requirements are usually the most critical and time consuming for parallel algorithms.

For excellent surveys see Kung (1980) for a discussion on the structure and characterization of parallel algorithms and Kuck (1976) where parallel processing of ordinary sequential programs is considered.

1.3 Computation Model Used

Observation 1. The parallel architectures mentioned above have some inherent drawbacks, namely, the focus on a specific architecture does not provide insight into the logical structure of the algorithms. Further, it also makes the transfer of an algorithm to a different architecture difficult.

Observation 2. Computational models for asynchronous parallelism are very abstract in nature (for example, Petri nets). This level of abstraction, while very powerful for structuring process interactions, often hides the actual computation process of the algorithm and makes the analysis of execution difficult.

Thus, there is a need for restricting ourselves to some specific but general model. Consequently, we will mainly use the SIMD-SM model (and its various submodels) for the purpose of this paper. This choice is a natural one since most parallel algorithms are indeed based on this model.

1.4 Paradigms For Organizing Computation

The design of parallel algorithms involves choosing appropriate data structures, allocating processors and finally the algorithm to do the computation. All these three aspects can be combined under the heading of ***computation organization.*** Certain simple computation organizations are used frequently in the problems solved in the literature and hence we will first identify these basic techniques in the form of paradigms. There are three paradigms that we will introduce for computation organization.

1.4.1 Binary Tree Paradigm

Consider the problem of calculating a function F of n data items $(d_1,...,d_n)$. If the function F is such that $(d_1,...,d_n)$ can be calculated by combining the results $F(d_1,...,d_{n/2})$ and $F(d_{n/2+1},...,d_n)$ then we can organize the computation of F in the form of a binary tree. The data form the leaves of the binary tree and the computation proceeds in steps. The processors are assigned so that all the computation at a level can be done as one step (and the same processors are reassigned for the next step). For a problem of size n where n is a power of 2, the number of processors required for the entire computation is $n/2$ and if the processing at each step takes c units of time then the entire computation takes $c*\log n$ units of time. A simple illustration of this technique is the problem of computing the sum of n numbers which uses $O(n)$ processors and requires $O(\log n)$ time.

In the above description notice that the computation proceeds from the leaves to the root and that the entire problem is solved when the processor at the root does its computation. Most of the problems that are solved using the binary tree method are of this form. However, the binary tree method is more general, Dekel, Sahni (1983a), and lends itself to problems in which a number of passes over the computation tree have to be made, each odd pass from leaves to the root and each even pass from the root to the leaves.

A simple illustration of this idea is the partial sums problem where $S_j = \sum_{i=1}^{j} d_i$, $1 \leq j \leq n$ is to be calculated. Notice that in the first pass of the computation tree not all the S_j's are computed, but the remaining S_j's can be computed by making the second pass from the root to the leaves. In the second pass, each vertex transmits to its children the sum of all the values to the left of this child.

1.4.2 Growing by Doubling

In the binary tree method, the first pass can be visualized in another way. At each step a (active) processor doubles the number of elements for which it has calculated the function F. Hence 'growth' is achieved by doubling at each step. There are other ways in which this type of doubling technique can be used even when there is no binary tree structure associated with the problem. A simple illustration is the problem of counting the number of elements in a linked list. This is done by associating a processor with each element of a list which maintains a pointer to an element further down the list; and at each step will double the distance its pointer spans by setting its pointer to the value of the pointer belonging to the element pointed to on the previous iteration.

1.4.3 Spanning Tree for Graphs

A very important class of parallel algorithms present in the literature are graph theoretic. A large majority of graph-theoretic problems are solved sequentially by first finding a depth-first spanning tree. However, the construction of depth-first spanning tree seems to be inherently sequential and hence cannot be done efficiently in a parallel model. Consequently, most of the sequential algorithms based on depth-first spanning trees are rather different from their parallel counterparts.

One way that graph theoretic problems relying on depth-first spanning tree are converted into efficient parallel algorithms is by making use of any spanning tree (Tarjan, Vishkin, 1984). In general, an algorithm based on any arbitrary spanning tree is more complicated than the one based on depth-first spanning tree, but this is a price that must be paid to obtain efficient parallel algorithms.

We now describe the spanning tree construction of Tarjan, Vishkin (1984) which is based on the connectivity algorithm of Shiloach, Vishkin (1982). Each vertex v has a pointer field $D[v]$ through which it points to another vertex or itself. The algorithm proceeds in a number of iterations. Initially, $D[i] = i$ for $i=1, .., n$; i.e. all vertices are in component trees by themselves. At any iteration, the graph is always a forest of rooted trees plus self-loops that occur only in the roots, each one of these is called a *D-tree*. The algorithm proceeds by ***hooking*** one tree onto another, this has the effect of decreasing the number of trees as well as increasing the size of individual trees. For efficiency, each rooted tree is kept so that each vertex is connected directly to the root and since the process of hooking violates this condition the algorithm also performs ***path compression*** or ***shortcutting*** in which a vertex is brought closer to the root. The D-pointers are therefore changed by two kinds of steps :

Step 1. Shortcutting: $D[i] = D[D[i]]$.
Step 2. Hooking: $D[D[i]] = D[j]$, where $D[i]$ is the root of a D-tree and j is a vertex in another D-tree and (i,j) is an edge in the graph. That is trees are hooked together. At this point simultaneous writing to the same location is used and it does not matter which one succeeds.

The edges used in the hooking step correspond to the edges in the spanning tree. This algorithm runs in $O(\log n)$ time using $O(n+m)$ processors.

1.5 Paradigms for Improving Efficiency of Parallel Algorithms

The efficiency of any parallel algorithm A is measured by defining what is known as the *effective processor utilization* or *EPU* defined as

$$\text{EPU} = \frac{\text{complexity of the fastest sequential algorithm for problem}}{\text{number of processors used in } A * \text{ time complexity of } A}$$

Note that $0 \leq \text{EPU} \leq 1$ and that an EPU equal to 1 is the best possible and corresponds to maximum speed up possible. As an illustration, the EPU for the sum of n numbers algorithm in the previous section is $\frac{n}{n * \log n} = \frac{1}{\log n}$.

In any parallel algorithm there is always a trade off between the number of processors and the computation time (within some limits). For instance, for any algorithm a new version may be obtained by allocating the processing of a number of processors to a single processor, thereby reducing the number of processors required at the expense of computation time. But notice that for any such simple scheme the EPU remains constant.

However, there are times where reallocation of the processing to the processors can be done in such a way as to increase the EPU of the new version of the existing algorithm. We describe two such general techniques below.

1.5.1 Processor Reduction Without Increasing Time Complexity

Consider the problem of finding the sum of n numbers discussed before. Suppose $p \leq n/2$ processors are available. Then to compute the lowest level sum would require at most $\lceil n/2p \rceil$ time, the next higher level would require at most $\lceil n/4p \rceil$, and so on. The total time T required satisfies

$$\begin{aligned}
T &\leq \lceil\frac{n}{2p}\rceil + \lceil\frac{n}{4p}\rceil + \cdots + \lceil\frac{n}{2^{\lceil \log n\rceil}p}\rceil \\
&\leq 1 + \frac{n}{2p} + 1 + \frac{n}{4p} + \cdots + 1 + \frac{n}{2^{\lceil \log n\rceil}p} \\
&= \lceil \log n\rceil + \frac{n}{p}[\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{\lceil \log n\rceil}}] \\
&\leq c_1 * \log n + c_2 * \frac{n}{p} \\
&= O(\log n + \frac{n}{p})
\end{aligned}$$

Notice that we can obtain a running time of $O(\log n)$ if $p = \frac{n}{\log n}$. That is we have reduced the number of processors required while keeping the same time complexity. This type of improvement can be made for a number of problems and we call it the ***processor reduction*** or *PR* paradigm.

1.5.2 Time Complexity Reduction Without Increasing The Number Of Processors

Consider again the problem of finding the sum of n numbers. This problem, as we had indicated above can be solved recursively by adding the sum of two smaller list of numbers. This recursive structure was reflected in the algorithm of the previous section where the recursion was unraveled all the way down. We now show how the time complexity can be reduced by unraveling the recursion only part of the way down; and this will be done in such a way as to not increase the number of processors required.

Let the recursion be unraveled until we are left with blocks of size x whose sum has to be computed. Now for each block we can allocate one processor to obtain the sum in time $O(x)$. The number of such blocks is n/x and hence the computation time required is

$$T = O(x + \log \frac{n}{x})$$

To minimize the time we require that $x = \log (\frac{n}{x})$ to give us $n = x*2^x$. So when $n = x*2^x$ we can compute the sum of n numbers in time $O(\log 2^x)$ and the number of processors required is 2^x. We call this technique the ***time reduction*** or *TR* paradigm and another example of its usage is a sorting algorithm in Hirschberg (1978).

Notice that this technique allows us to decrease the time complexity as well as the number of processors required from that required in the algorithm of the previous section. Also, this technique yields a faster algorithm than the one by the PR paradigm, however this speed up is at the expense of the number of processors. However, for both of the algorithms in this section the product of the number of processors and the time complexity is the same; that is

$$\begin{aligned}
\text{using paradigm PR, number of processors } * \text{ time complexity} &= \frac{n}{\log n} * \log n = n \\
\text{using paradigm TR, number of processors } * \text{ time complexity} &= \log 2^x * 2^x \\
&= x*2^x = n
\end{aligned}$$

Further, for both the algorithms obtained in this section the EPU = 1; the best possible that can be achieved theoretically.

There are two different assumptions that can be made about the number of processors available. Under ***unbounded parallelism***, arbitrarily many processors are assumed to be available. Under ***bounded parallelism***, a limited number of processors, independent of the problem size, are assumed to be available.

1.6 Organization And Scope Of The Paper

Section 2 present a survey of parallel algorithms for finding the connected and biconnected components of a graph. Section 3 discusses the various parallel algorithms for the minimum spanning tree problem. Section 4 discusses various other parallel graph algorithms for shortest path, maximum matching, planarity testing, maximal independent set etc. Section 5 considers parallel algorithms for various non-graph theoretic problems like arithmetic expression and polynomial evaluation, string matching, tree balancing and alpha-beta search. Finally, section 6 presents some conclusions.

In our treatment of parallel algorithms, we have excluded discussion on sorting and numerical algorithms as well as any algorithms for VLSI and systolic arrays. An extensive amount of literature exists for parallel sorting and numerical algorithms. For instance, Bitton et al. (1984), Chen (1975), Dekel et al. (1981), Gentleman (1978), Lakshmivarahan et al. (1984), Sameh (1977). Kuck (1977) deals with parallel machine organization and programming; while Kuhn, Padua (1981) contain a collection of papers dealing with parallel processor taxonomies, pipeline, dataflow and array processors, compilers for translating sequential programs into parallel ones, operating systems for parallel computers etc. Lee et al.

(1985) deal with implementation details required for restructuring various nonnumerical programs for parallel processing.

The parallel algorithms we discuss are all deterministic. even though numerous randomized algorithms exist in the literature. Randomization can often achieve better efficiency then deterministic algorithms. The study of random algorithms is of current research focus and a lot more work needs to be done in this area.

In this paper we have attempted to give a survey of parallel algorithms for a class of computational problems. Undoubtedly, some algorithm may have been overlooked, but any such omissions were unintentional.

2. Parallel Connectivity Algorithms

2.1 Parallel Connected Components Algorithms

The various parallel algorithms for finding the connected components of undirected graphs can be broadly classified according to two major criteria - the basic technique employed and the format of the input. The basic techniques used in these algorithms have been breadth-first search, transitive closure and vertex collapse. The most common form of the input is adjacency matrix. The popularity of the adjacency matrix form of input in general stems from the fact that it allows graph theoretic problems to be stated and solved in terms of matrix manipulation problems. This works very well for dense graphs, but for sparse graphs this may lead to inefficient algorithms and so quite often input in the form of adjacency lists is used. While most of the algorithms for finding connected components are very general, some of them are more suitable for sparse or dense graphs. As we shall see, this extra information about the sparsity of graph allows for more efficient allocation of processors.

2.1.1 Breadth-first Search

Arjomandi, Corneil (1975) have proposed a technique for finding the connected components of a dense graph based on a breadth-first search of the given graph. The search starts from a vertex and processors are assigned to edges emanating from that vertex. A vertex at a distance of i from the start vertex is searched before any vertex at a greater distance from the start vertex. After all the vertices at a distance i have been searched, the partial lists of all the new vertices at a distance $i + 1$ are joined together so that a new vertex at a distance $i + 1$ can be picked efficiently for searching. When the average degree of each vertex in the graph is approximately greater than or equal to k, where k is the number of processors available, then the complexity of the resulting algorithm is near optimal and is $T/k + L\lceil \log k \rceil + 2n$ where T is the complexity of the optimal sequential breadth-first search algorithm and L is the distance of the vertex furthest away from the start vertex.

2.1.2 Transitive Closure

The basic idea of connected component (sequential) algorithms based on transitive closure is as follows. If the input is in the form of adjacency matrix A then the transitive closure of A, call it B, can be obtained so that the element $b_{i,j}$ of B is 1 if and only if there is a path of length 0 or more from vertex i to vertex j in the given graph. The component number of any vertex i is then just the smallest j such that $b_{ij} = 1$.

The first parallel algorithm for finding the connected components based on transitive closure is due to Reghbati, Corneil (1978). Their algorithm is essentially what is described above. On a SIMD-SM-R model the transitive closure of a n x n matrix can be obtained when n^3 processors are available by squaring the matrix $\log n$ times which requires $O(\log^2 n)$ time; finding the first nonzero entry for each row can be done in $O(\log n)$ time. Hence the complexity of the algorithm is $O(\log^2 n)$ using n^3 processors on an SIMD-SM-R model.

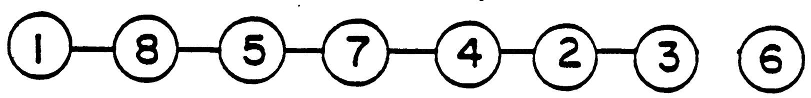
The other algorithms based on transitive closure improve upon the efficiency of the transitive closure step. In Chandra (1976), the parallel matrix multiplication algorithm is used thereby reducing the number of processors required from n^3 to $\lceil n^{\log 7}/\log n \rceil$.

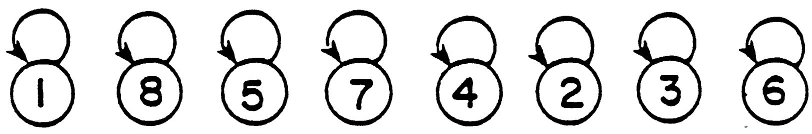
In Kucera (1982) a stronger model, SIMD-SM-RW model, is used. This allows n^2 processors to find the minimum of a set of n elements in constant time and hence allows the transitive closure of a binary matrix and the connected components to be computed in $O(\log n)$ time using n^4 processors.

2.1.3 Vertex Collapse

The input format is again in the form of adjacency matrix. The basic idea is to combine adjacent vertices into 'supervertices' which are then combined themselves. This process is repeated until no further collapses are possible; at which point each supervertex corresponds to a connected component. Some commonly used terminology is as follows. A *tree-loop* is a tree having directed edges with an additional edge from the root to one of its descendants. A *club* is a tree-loop in which all the vertices other than the root are the sons of the root.

The first algorithm using the vertex collapse technique is due to Hirschberg (1976) and is presented below. The basic idea is to grow each connected component as a club with the root of the club being the vertex with minimal label in that club. Fig. 1 illustrates how this algorithm works. The algorithm proceeds by having each vertex identifying the smallest numbered vertex that it has seen so far which is in the same connected component (in Fig. 1 this relationship is indicated by solid arrows). Each iteration of the algorithm consists of three steps. In the first step, each vertex finds lowest-numbered (new) neighboring root. In the second step, each root is linked to the root of the lowest-numbered neighboring root. In the third step, path compression is done so as to construct clubs.

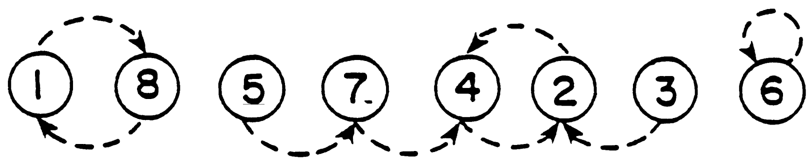




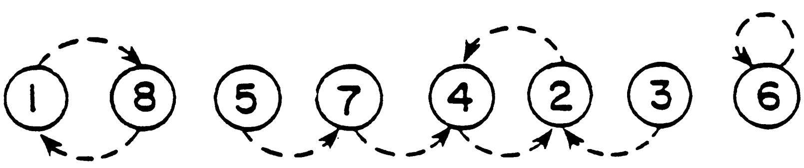
Smallest numbered vertex seen so far which is in the same connected component.

Iteration 1

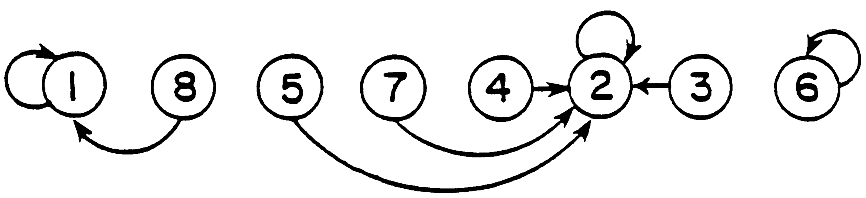
Find lowest-numbered (new) neighboring root.



Connect roots.



Build new club.



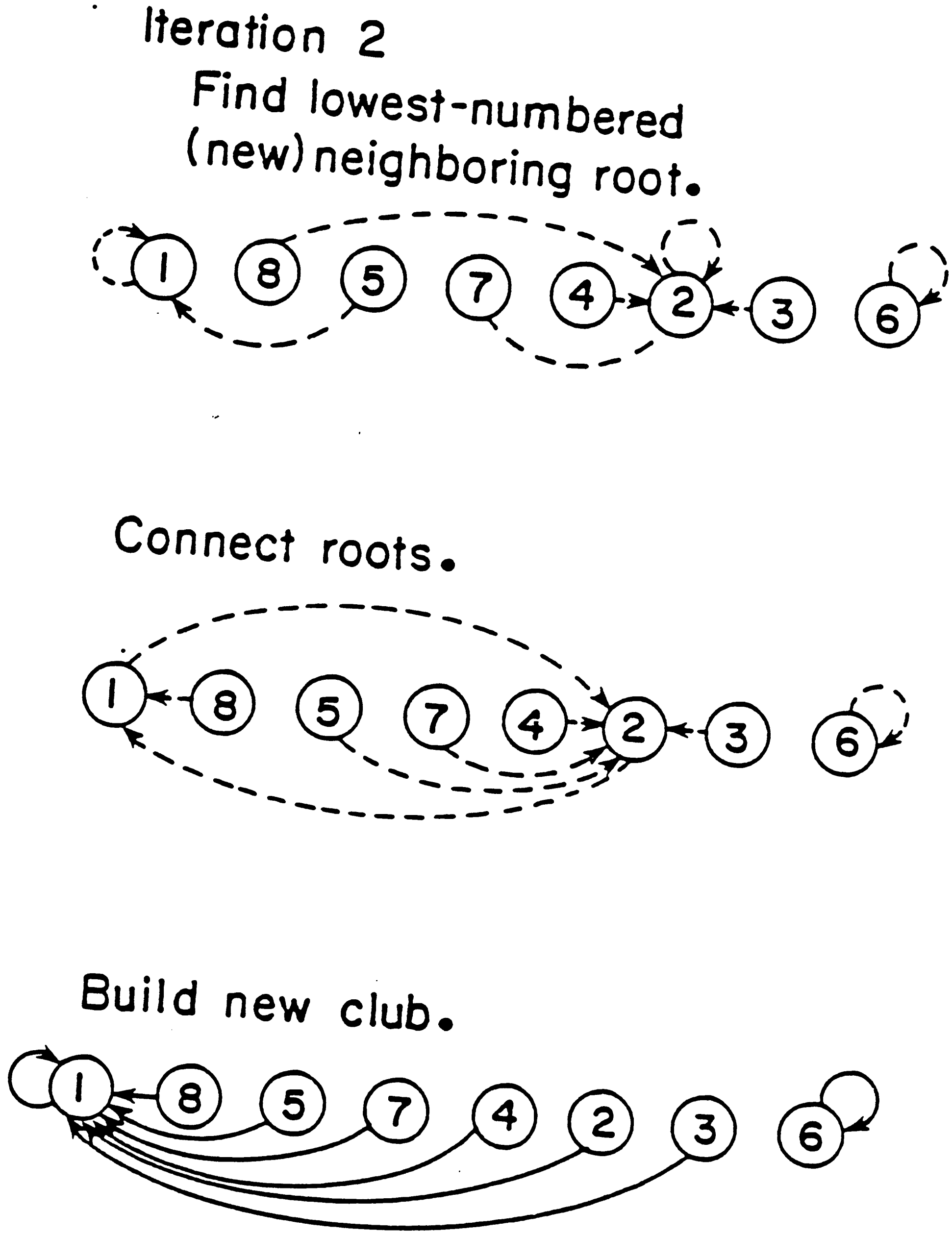


FIG. 1. Vertex collapse.

---

algorithm Hirschberg;
Input: Adjacency matrix $A[n \text{ by } n]$
Output: Vector D of length n such that $D[x]$ is the smallest vertex y reachable from vertex x.

```
begin
1. for all x do D[x]=x;
2. for ⌈log n⌉ iterations do
    begin
3.    for all x
         do C[x]=min {D[y] | A[x,y]=1 and D[y]≠D[x]}
                 if none then D[x];
4.    for all x
          do C[x]=D[x]=min {C[y] | D[y]=x and C[y]≠x}
                 if none then D[x];
5.    for all x do D[x]=C[x];
6.    for ⌈log n⌉ iterations do
         for all x do C[x]=C[C[x]];
7.    for all x do D[x]=min { C[x],D[C[x]] };
   end;
end;
```

Algorithm Hirschberg has time complexity $O(\log^2 n)$ on n^2 processors for a SIMD-SM-R model. Basically, both the path compression and finding the minimum requires $O(\log n)$ time.

Also, since each iteration reduces the number of clubs corresponding to a connected component by a factor of 2 until a single club corresponds to a connected component, $O(\log n)$ iterations are sufficient for the algorithm.

Hirschberg et al. (1979) improve upon the performance of the original Hirschberg algorithm by recognizing that the number of processors required can be reduced without increasing the time complexity. The idea is very similar to what we outlined as the PR paradigm in 1.5.1 and amounts to having each processor assign values to $\log n$ elements instead of to one element and that each processor can find the minimum of $\log n$ values instead of finding the minimum of two values. Both of these changes can be accomplished without increasing the time complexity so the new algorithm has a complexity of $O(\log^2 n)$ using $n \lceil n / \lceil \log n \rceil \rceil$ processors on a SIMD-SM-R model.

Chin et al. (1981, 1982) improve upon the algorithm of Hirschberg et al. This improvement is obtained by considering in any iteration only those vertices that are non-isolated supervertices. This implies that the number of active vertices is reduced by a factor of at least two after each iteration and therefore the same time bound $O(\log^2 n)$ can still be achieved by using $n \lceil n / \log^2 n \rceil$ processors on the SIMD-SM-R model. This reduction in the number of processors is essentially the PR paradigm.

Savage, Ja'Ja' (1981) develop two parallel connected components algorithms that provide improved performances for dense and sparse graphs respectively. For an undirected graph $G = (V,E)$ with $|V| \geq 2$, the ***diameter d*** of G is defined to be

$$d = \max \{d(v, w), 2\} \quad v, w \in V$$

where $d(v, w)$ is the length of the shortest path from v to w if it exists, otherwise $-\infty$.

The observation they made is that it is not necessary to run the Hirschberg algorithm for $\lceil \log n \rceil$ iterations; rather it should be run until two consecutive iterations produce the same result. In that case the running time of the algorithm can be made $O(\log n * \min \{ \log n, d/2\}$ where d is the diameter of the graph. When $d < 2 \log n$ (i.e. for dense graphs), this algorithm is faster than the original. When the graph is dense this algorithm requires $O(n^3/\log n)$ processors to obtain time complexity $O(\log n \log d)$.

When the graph is sparse, the number of processors required can be reduced by organizing the input in the form of adjacency list rather than adjacency matrix. Certain reorganization of the original Hirschberg's algorithm then leads to an algorithm with time complexity $O(\log^2 n)$ using $O(m + n \log n)$ processors where m is the number of edges in the graph.

Nath, Maheshwari (1982) consider the problem of finding the connected components on a weaker model, a model that does not permit simultaneous reading from the same memory location by different processors. Their algorithm is based on the one due to Hirschberg et al. (1979) but avoids read conflicts by organizing the intermediate data structure as a chain (which avoids read conflicts when children access nonroot parents) and by keeping multiple copies of data. This algorithm has complexity $O(\log^2 n)$ with n^2 processors on an SIMD-SM model.

Shiloach, Vishkin (1982) consider a stronger model, SIMD-SM-RW, where simultaneous reads as well as writes are allowed. In the latter case one processor succeeds but which one is not known. This algorithm, SV, has complexity $O(\log n)$ using only $n + 2m$ processors where $|E| = m$, and it is presented below.

algorithm SV;

Input: Vertices are represented by the numbers $1,\ldots,n$. The edges are represented by a vector E of length $2m$ in which each edge (i,j) appears twice : once as the ordered pair $<i,j>$ and once as the ordered pair $<j,i>$.
Output: Vector D_{s_0} of length n, s_0 is the index of the last iteration. $D_{s_0}[n]$ is the representative vertex of the connected component containing vertex n.
Processor allocation: If $i \leq n$, processor P_i is allocated to vertex i. If $i > n$, processor P_i is allocated to the ordered pair $E[i-n] = <i_1, i_2>$ (this processor is denoted as P_{i_1, i_2}).

```
 begin
1.   if i≤n then
     begin
2.     D_0[i]=i; Q[i]=0; s=1; s'=1;
     end;
3.   while s'=s do
      begin
4.       if i≤n then
         begin
5.           D_s[i]=D_{s-1}[D_{s-1}[i]]
6.           if D_s[i]≠D_{s-1}[i] then Q[D_s[i]]=s;
         end;
7.       if i>n
8.       then if D_s[i_1]=D_{s-1}[i_1]
9.           then if D_s[i_2]<D_s[i_1]
10.               then begin D_s[D_s[i_1]]=D_s[i_2]; Q[D_s[i_2]]=s; end;
11.      if i>n
12.      then if D_s[i_1]=D_s[D_s[i_1]] and Q[D_s[i_1]]<s
13.          then if D_s[i_1]≠D_s[i_2]
14.               then D_s[D_s[i_1]]=D_s[i_2];
15.      if i≤n then D_s[i]=D_s[D_s[i]];
16.      if i≤n and Q[i]=s then s'=s'+1;
17.      s=s+1;
      end;
 end;
```

In the SV algorithm, the processors are allocated by considering each undirected edge to be two directed edges, and each vertex and each directed edge is allocated one processor. In contrast to the Hirschberg algorithm, it is no longer required that each connected component be identified by its lowest-numbered member vertex; this allows the algorithm to utilize the power of this model by letting the vertices try simultaneous 'hooking' instead of connecting to minimum numbered neighboring root. This allows one of the two $\log^2 n$ factors to be replaced by a $\log n$ factor. The other $\log^2 n$ factor is replaced by a $\log n$ factor by reorganization that allows the path compression step to be done twice in each iteration rather than $O(\log n)$ times.

Awerbuch, Shiloach (1983) improve upon the results of Shiloach, Vishkin (1982) to obtain a $O(\log n)$ time complexity connected components algorithm using $2m$ processors for the SIMD-SM-RW model (an arbitrary process succeeds in case of simultaneous writes).

Wyllie (1979) uses vertex collapse to obtain a $O(\log^2 n)$ algorithm using $O(n+2m)$ processors on a synchronized MIMD-TC-R model. For sparse graphs this is a very efficient algorithm. A vertex x can be collapsed into vertex y by making any edge incident to x become incident to y instead. Each edge is viewed as a pair of two directed edges and the input is in the form of adjacency list matrix. This input is converted in constant time so that each vertex has a circular doubly linked list with a list header. The list consists of two types of elements : directed edge elements and dummy elements and at all times edge elements are separated by at least one dummy element. Each directed edge element contains (in addition to the forward and backward pointer) the number of vertex to which this directed edge is incident and a pointer to its 'brother' directed edge (the oppositely oriented directed edge representing the same undirected edge). The structure of the directed edge element allows for efficient collapse of two vertices into one vertex and the use of dummy elements permit simultaneous collapse of different vertices.

The various algorithms discussed for the connected components problem are summarized below in Table I.

Table I. Parallel Connected Components Algorithms

Reference	Model	Complexity	Processors
Arjomandi, Corneil (1975)	SIMD-SM-R	$T/k + L\lceil \log k \rceil + 2n$	k
Reghbati, Corneil (1978)	SIMD-SM-R	$O(\log^2 n)$	n^3
Chandra (1976)	SIMD-SM-R	$O(\log^2 n)$	$n^{\log 7}/\log n$
Kucera (1982)	SIMD-SM-RW	$O(\log n)$	n^4
Hirschberg (1976)	SIMD-SM-R	$O(\log^2 n)$	n^2
Hirschberg et al. (1979)	SIMD-SM-R	$O(\log^2 n)$	$n\lceil n/\lceil \log n \rceil \rceil$
Chin et al. (1981, 1982)	SIMD-SM-R	$O(\log^2 n)$	$n\lceil n/\log^2 n \rceil$
Savage, Ja'Ja' (1981)	SIMD-SM-R	$O(\log n \log d)$	$O(n^3/\log n)$
	SIMD-SM-R	$O(\log^2 n)$	$O(m + n \log n)$
Nath, Maheshwari (1982)	SIMD-SM	$O(\log^2 n)$	$n^2/\log n$
Shiloach, Vishkin (1982)	SIMD-SM-RW	$O(\log n)$	$n + 2m$
Awerbuch, Shiloach (1983)	SIMD-SM-RW	$O(\log n)$	$2m$
Wyllie (1979)	MIMD-TC-R	$O(\log^2 n)$	$n + 2m$

where d is the diameter of the graph; L is the distance of vertex furthest away from start vertex and T is the complexity of optimal sequential breadth-first search algorithm.

2.2 Parallel Biconnected Components Algorithms

For a connected undirected graph $G=(V,E)$, a vertex $a \in V$ is called an ***articulation*** point of G it there are vertices x and y in V distinct from a, such that every path in G joining x and y contains a. The graph is biconnected if it contains no articulation points. A biconnected component of G is a biconnected subgraph, H, of G which is not properly contained in any biconnected subgraph of G.

There are a number of parallel algorithms for finding the biconnected components of an undirected graph and they are summarized in Table II.

Table II. Parallel Biconnected Component Algorithms

Reference	Model	Complexity	Processors
Eckstein (1979b)	SIMD-SM-R	$O(d \log^2 n)$	$O((n+m)/d)$
Savage, Ja'Ja' (1981)	SIMD-SM-R	$O(\log^2 n)$	$O(n^3/\log n)$
	SIMD-SM-R	$O(\log^2 n \log k)$	$O(mn + n^2 \log n)$
Tsin, Chin (1984)	SIMD-SM-R	$O(\log^2 n)$	$O(n \lceil n/\log^2 n \rceil)$
Tarjan, Vishkin (1984)	SIMD-SM-RW	$O(\log n)$	$O(n+m)$
	SIMD-SM-R	$O(n^2/p)$	$p \leq n^2/\log^2 n$

where d is the diameter of the graph and k is the number of biconnected components in the graph.

Eckstein (1979b) algorithm uses $O(d \log^2 n)$ time and $O((n+m)/d)$ processors on the SIMD-SM-R model where d is the diameter of the graph. Savage, Ja'Ja' (1981) give two parallel algorithms, both based on the SIMD-SM-R model. The first algorithm uses $O(\log^2 n)$ time and $O(n^3/\log n)$ processors. The second algorithm uses $O(\log^2 n \log k)$ time and $O(mn + n^2 \log n)$ processors where k is the number of biconnected components and hence is more suitable for handling sparse graphs.

The parallel algorithms of Tsin, Chin (1984) and Tarjan, Vishkin (1984) are both based on the following ideas. The sequential algorithm of Tarjan (1972) for finding the biconnected components of an undirected graph is based on manipulating the depth-first spanning tree of the original graph. However, there is strong reason to believe that depth-first search (Reif 1985) is inherently sequential, i.e. it does not seem possible to implement it in polylog parallel time. Therefore, a new way of finding the biconnected components is devised so that the problem of computing the biconnected components of the input graph can be reduced to the problem of computing the connected components of an auxiliary graph. The actual details of the algorithms by Tsin, Chin (1984) and Tarjan, Vishkin (1984) are however different.

The algorithm of Tsin, Chin (1984) uses $O(\log^2 n)$ time and $O(n \lceil n/\log^2 n \rceil)$ processors on the SIMD-SM-R model. Briefly, the Tarjan, Vishkin algorithms can be described as follows. First a sequential algorithm for finding the biconnected components is developed so that it makes use of any spanning tree (rather than a depth-first spanning tree). This sequential algorithm is then parallelized to yield an algorithm for the SIMD-SM-RW model that runs in $O(\log n)$ time and uses $O(n+m)$ processors. A second algorithm is also presented for the SIMD-SM-R model that uses $O(n^2/p)$ time and $p \leq n^2/\log^2 n$ processors.

In comparing the algorithms due to Tsin, Chin and Tarjan, Vishkin, notice that the algorithm of Tsin, Chin and the second algorithm of Tarjan, Vishkin are both optimal since the time-processor product is $O(n^2)$, and the best sequential algorithm with adjacency matrix input is also $O(n^2)$. Also, by using a general simulation it is possible to convert the first algorithm of Tarjan, Vishkin so that it becomes one using $O(\log^2 n)$ time and $O(n+m)$ processors on the SIMD-SM-R model. For sparse graphs this has a much better performance than the algorithm of Tsin, Chin and the second algorithm of Tarjan, Vishkin.

3. Parallel Minimum Spanning Tree Algorithms

In this section of our paper, we present several parallel minimum spanning tree algorithms for different types of parallel computational models. The problem of finding a minimum spanning tree of a weighted graph arises in many diverse applications such as communication system, (minimum-cost spanning tree represents a communication network that connects all cities at minimal cost), transportation networks, data analysis and operations research.

A minimum spanning tree of a weighted, connected and undirected graph G is defined as a set of edges of the graph that connects all vertices and whose total edge weight is minimum. There are many different methods for constructing a minimum-cost spanning tree. The following property is used to define the minimum-cost method. Let $G = (V,E)$ be a connected weighted undirected graph. Let U be some proper subset of the set of vertices V. If (u,v) is an edge of minimum cost such that $u \in U$ and $v \in V-U$, then there is a minimum cost spanning tree that includes (u,v) as an edge. The proof that every minimum-cost spanning tree satisfies the above property is given in Aho et al. (1974).

Traditionally, sequential methods to compute the minimum spanning tree of a weighted graph in parallel have focussed on three classes of algorithms all of which use greedy strategies. They are: (1) Sollin (1977) algorithm based on lightest-edge-from-each-vertex method, (2) Prim (1957) and Dijkstra (1959) algorithm based on nearest-neighbor method, and (3) Kruskal (1956) algorithm based on lightest-edge-first method. For details on results related to minimum spanning tree problem see Cook (1981), Kucera (1982), Chin et al. (1982), Awerbuch, Shiloach (1983), Hirschberg (1983), and Kwan, Ruzzo (1984). Among these minimum spanning tree algorithms, Sollin algorithm is the most suitable candidate for parallel processing.

3.1 Parallel Sollin's Algorithm

The Sollin's algorithm for finding the minimum spanning tree is presented in the algorithm SOLLIN. Sollin's algorithms starts with the forest F_0 obtained by forming a tree with each vertex of the graph. In each iteration, the lightest edge incident on each tree is selected. All these edges are added to the existing forest F_i to form the new forest F_{i+1}. This process is continued till the forest F_i consists of single tree. In the worst-case this algorithm requires $\log n$ iterations. The direct implementation of the sequential algorithm of Sollin requires $O(n^2 \log n)$ time. Researchers have also looked at efficient implementations of Sollin algorithm on a sequential machine. Yao (1975) presents an algorithm with $O(n^2 \log \log n)$ time complexity for this problem. The Cheriton, Tarjan (1976) algorithm has $O(m \log \log n)$ time complexity.

Parallel implementation of algorithm SOLLIN calls for suitable parallel implementation of the lines 4 through 6 of algorithm SOLLIN. We now present the parallel versions of the algorithm SOLLIN for different types of computer configurations.

```
algorithm SOLLIN(G);
 begin
1.     F0=(V,φ);      // begin with a forest F of n vertices of V //
2.     i=0;
3.     while there is more than one tree in Fi do
       begin
4.        for each tree Tj in forest Fi do
5.            choose the minimum weighted edge (u,v) joining some vertex u
              in Tj to a vertex v in some other tree Tk in forest Fi;
6.        form the forest Fi+1 by joining all Tj and Tk of Fi with the
          corresponding selected edges;
7.        i=i+1;
       end;
 end;
```

3.1.1 Sollin's Algorithm For SIMD Machine

A number of researchers have implemented Sollin's algorithm on SIMD machine. Savage, Ja'Ja' (1981) and Nath, Maheshwari (1982) base their algorithms on Hirschberg's (1976) connected component algorithm to identify the newly combined trees. The algorithm of Savage, Ja'Ja' (1981) achieves the time complexity of $O(\log^2 n)$ using $O(n^2)$ processors on the SIMD-SM-R model while Nath, Maheshwari (1982) achieve time complexity $O(\log^2 n)$ using $n^2/\log n$ processors on the SIMD-SM model. Chin et al. (1982) modify Hirschberg's connected component algorithm to obtain $O(\log^2 n)$ using only $O(n \lceil n/\log^2 n \rceil)$ processors on the SIMD-SM-R model.

The technique used by Savage, Ja'Ja' (1981) is outlined in algorithm PSOLLIN-SIMD. In the statements 1-2 of algorithm PSOLLIN-SIMD each vertex is represented as a single tree whose root is itself. The complexity of this task is constant with $O(n)$ processors. The statements 5-6 requires finding n times the minimum of at most n numbers. The minimum of n numbers can be found in $\log n$ time using n processors. Thus, the computation of statements 5-6 can be performed in $O(\log n)$ time using $O(n^2)$ processors. For each component K of the existing forest, the vertex i with minimum value for $W[i, NEAR[i]]$ is found in statements 7-8. Thus, the statements 7-8 can be executed in time $O(\log n)$ using $O(n^2)$ processors. Statements 10-11 find for each vertex i a representative component where vertex i belongs. This can be done in time $O(\log n)$ with $O(n^2)$ processors using Hirschberg's technique (1976). The statement 12 checks whether each vertex belongs to the same component. This step requires $O(\log n)$ time because SIMD-SM-R machine model does not allow simultaneous writing into the same memory location. The time complexity of each iteration is $O(\log n)$, and there are $O(\log n)$ iterations. Thus the total time complexity of this algorithm is $O(\log^2 n)$ with $O(n^2)$ processors.

```
algorithm PSOLLIN-SIMD;
begin
    for each vertex i do
        ROOT[i]=i;
    over= false;
    while not(over) do
    begin
        for each vertex i do
            NEAR[i] =j such that (ROOT[i] ≠ ROOT[j]) and
                                (W[i,j] < W[i,k], for all k ≠ j)
        for each component K of the forest F_t do
            choose vertex i such that W[i,NEAR[i]] is minimum over all
            vertices of K;
        combine and create a new forest F_{t+1};
        for each vertex i do
            search ROOT[i];
        if ROOT[i]=ROOT[j] for all i,j then over=true;
    end;
end;
```

3.1.2 Sollin's Algorithm For MIMD Machine

Deo, Yoo (1981) and Yoo (1983) present algorithms for the MIMD-TC model with the assumption that the number of available processors p is less than or equal to n. Each process handles approximately n/p vertices. The time required to compute the lightest edge incident on each vertex is $O(n^2/p)$ if a weight matrix is used. This step is repeated until a spanning tree is obtained. A detailed description of the algorithm presented in Yoo (1983) is given in algorithm PSOLLIN-MIMD. The functions FIND and procedure UNION used in this algorithm are for set operations. The function FIND(i) returns the set in which i is contained. The invocation of the procedure UNION($r1$,$r2$) computes the set union of the sets $r1$ and $r2$.

In each iteration of algorithm PSOLLIN-MIMD, every process examines n/p of the total vertices, and finds a vertex nearest to a non-tree neighbour. Therefore a single iteration has complexity $O(n^2/p)$ and the entire algorithm has the time complexity of $O((n^2/p)\log n)$. When $p=n$ the time complexity becomes $O(n \log n)$, and the processor time product becomes $O(n^2 \log n)$. In the worst-case the number of process creation becomes $(p \log n)$. The parallel version of Cheriton and Tarjan's (1976) minimum spanning tree algorithm is similar to the parallel version of Sollin's algorithm. For details see Deo, Yoo (1981).

```
algorithm PSOLLIN-MIMD;
// parallel version of Sollin's algorithm for MIMD-TC machine //
process MAIN;
// initialization //
begin
    T=φ; T_w=0;
    over=0;
    for i= 2 to p do
        create TASK1(i);
    call TASK1(1);
    while over < p do wait;
    for i = 1 to n do
    begin
        j= V_j[i];
        r1= FIND[i]; r2=FIND[j];
        if (r1 ≠ r2) then
        begin
            T=T ∪ {(i,j)};
            T_w=T_w+ V_min[i];
            call UNION(r1,r2);
        end;
    end;
    while there is more than one tree do
    begin
        over=0;
        for i=2 to p do
            create TASK2(i);
        call TASK2(1);
        while over < p do wait;
        for i= 1 to n do
        begin
            if V_min[i] ≠ ∞ then
            begin
                r1 = FIND(V_i[1]); r2 = FIND(V_j[i]);
                if r1 ≠ r2 then
                begin
                    T=T ∪ {(V_i[i],V_j[i])};
                    T_w=T_w+ V_min[i];
                    call UNION (r1,r2);
                end;
            end;
        end;
    end;
end;
```

```
    process TASK1(k);
    // computation process for finding the nearest vertex in PSOLLIN-MIMD //
    begin
1.      for i = k to n by p do
        begin
2.          TREE[i]= -1; m =1;
3.          for j=1 to n do
4.              if W[i,j]< W[i,k] then m=j;
5.          V_j[i]=m; V_min[i]= W[i,m];
        end;
6.      lock(over); over=over+ 1; unlock(over);
    end;
```

```
    process TASK2(k)
    begin
1.      for i=k to n by p do
2.      V_min[i]=∞;
3.      lock(over);
4.      over=over+ 1;
5.      if over=p then over=0;
6.       unlock(over);
7.       while over > 0 do wait;
8.       for i=k to n by p do
         begin
9.           r1= FIND[i];
10.          for j=1 to n do
             begin
11.              r2= FIND[j];
12.              if r1 ≠ r2 then
                 begin
13.                  a= min {i,j}; z= max {i,j};
14.                  lock(V_min[r1]);
15.                    if (W[i,j]< V_min[r1]) ∨ (W[i,j]= V_min[r1] and
                             W(a,z) < min {V_i[r1],V_j[r1]}) then
                       begin
16.                        V_i[r1]=a; V_j[r1]=z; V_min[r1]= W[i,j]
                       end;
17.                  unlock(V_min[r1]);
                 end;
             end;
        end;
18.     lock(over); over=over+ 1; unlock(over);
    end;
```

3.2 Parallel Prim-Dijkstra Algorithm

The minimum spanning tree algorithm was first developed by Prim (1957) and its computer implementation was given by Dijkstra (1959). The algorithm (we name it Prim-Dijkstra) works by starting with a forest consisting of n isolated vertices. Then a vertex is chosen arbitrarily from the forest to be the first vertex of the partial tree. Then, the tree is grown successively by bringing in the nearest neighbour from one of the isolated vertices. In other words, at an intermediate step an edge (u,v) is added to the current (partially formed) tree T if $T \cup \{(u,v)\}$ is also a tree and (u,v) is the lightest weight edge with this property. This method is described in algorithm PRIM-DIJKSTRA.

```
   algorithm PRIM-DIJKSTRA;
   // this algorithm finds the minimal spanning tree for a given graph //
   begin
   // initialization //
1.     T=φ; T_w=0;
2.     NEAR[1]=0;
3.     for i=2 to n do
       begin
4.         NEAR[i]= 1;
5.         DIST[i]= W[1,i]
       end;
   // iteration step //
6.     while |T|≤n-1 do
       begin
7.         choose j such that DIST[j]=min {DIST[i] | NEAR[i]≠0};
8.         T=T∪ {(j,NEAR[j])};
9.         T_w=T_w+ DIST[j];
10.        NEAR[j]=0;
11.        for i=1 to n do
12.            if (NEAR[i]≠0) and (W[i,j]< W[i,NEAR[i]]) then
               begin
13.                NEAR[i]=j;
14.                DIST[i]= W[i,j];
               end;
       end;
   end;
```

The PRIM-DIJKSTRA algorithm uses the array W that contains the weights of the edges. The arrays *NEAR* and *DIST* keep the current nearest neighbor and the corresponding weights of each vertex. T_w is the weight of the tree and T the set of edges in the tree at any stage during the execution of the algorithm. This algorithm has $n-1$ iterations, and at each iteration a new edge is added to the partially built tree T_w. At each iteration the computation of the required edge can be performed in $O(n)$ time using the weight matrix as the data structure. Thus, the minimum spanning tree can be computed in $O(n^2)$ time on a sequential machine.

We discuss two parallel implementation of the Prim-Dijkstra minimum spanning tree algorithm. Bentley (1980) uses a tree machine to obtain a parallel version of the Prim-Dijkstra algorithm. Deo, Yoo (1981) present a MIMD version of the Prim-Dijkstra algorithm.

3.2.1 Prim-Dijkstra Algorithm For Tree Machine

In this section we present the parallel version of Prim-Dijkstra algorithm of Bentley (1980) for the tree machine. Initially, we assume that there are n square processors. The initialization is implemented on the driver computer of the tree machine. The values are inserted into the tree machine by pipeline process. Thus, the time required for implementation of this step is $O(n)$.

In each iteration, the triangle processors compute the next edge to be added to the partially built minimum spanning tree. Thus, the iteration step of the parallel algorithm consists of finding the minimum of n numbers in $O(\log n)$ time by letting the triangle processors compute the minimum of two incoming values. During each iteration the vertex nearest to the present partial tree is transferred down to the square processors in $\log n$ steps. If $W[i,j] < W[i,NEAR[i]]$, then the square processor i alters $NEAR[i]$ to j and $DIST[i]$ to $W[i,j]$. This step takes $\log n$ time. Since there are $n-1$ iterations, the total time taken for the execution of the algorithm is $O(n \log n)$.

The same complexity of $O(n \log n)$ can be obtained by storing $\log n$ vertices with each square processor. Each square processor handles the corresponding vertices by finding the minimum and updating if necessary. The triangle processors find the minimum of $O(n/\log n)$ values. Therefore, each iteration still takes $O(\log n)$ time. Hence, the total running time of algorithm PRIM-TREE remains $O(n \log n)$, with $O(n/\log n)$ processors. This reduction in the number of processors without increasing the time complexity is an instance of the PR paradigm described in section 1.5.1.

```
    algorithm PRIM-TREE;
    process DRIVER;
    begin
1.      for each vertex i do
        begin
2.          if i=1 then
            begin
3.              NEAR[i]=0;
4.              DIST[i]=∞;
            end
5.          else
            begin
6.              NEAR[i] =1;
7.              DIST[i]= W[1,i];
            end;
8.          transmit (NEAR[i], DIST[i]) to the first broadcast processor in the
            tree machine;
        end;
    end;
```

```
    process SQUARE(i);
    begin
1.      repeat
2.          receive(x,dx);
3.          if NEAR[i] ≠ 0 then
            begin
4.              if x=i then
                begin
5.                  NEAR[i]=0;
6.                  DIST[i]= ∞;
                end
7.              else
8.              if W[i,x]< DIST[i] then
                begin
9.                  NEAR[i]=x;
10.                 DIST[i]= W[i,x];
                end;
            end;
11.         send (NEAR[i],DIST[i]) to triangle processor;
12.     forever;
    end;
```

```
  process TRIANGLE(i);
  begin
1.    repeat
2.        (receive (x,dx) from left, receive (y,dy) from right);
3.        if dx < dy
4.        then send (x,dx)
5.        else send (y,dy);
6.    forever;
  end;
```

3.2.2 Prim-Dijkstra Algorithm For MIMD Machine

Deo, Yoo (1981) and Yoo (1983) present the parallel version of Prim-Dijkstra algorithm for an MIMD machine. They assume p processors available with the machine where $2 \leq p \leq n$. For each of the $n-1$ iterations of the Prim-Dijkstra algorithm p processes are created. Each process updates the distance of n/pth of the non-tree vertices and all these processes synchronize after their computations. After synchronization, another set of p processes is created for updating thereby completing one iteration. The number of iterations is same as for the sequential algorithm and hence if $p = n^{.5}$ the time complexity is $O(n^{1.5})$ on an MIMD-TC model with $O(n^{.5})$ processors. (The frequent number of synchronizations required and the contention for the global memory limits the number of processors that can be effectively utilized.) The details are presented in algorithm PRIM-MIMD.

```
  algorithm PRIM-MIMD;
  // parallel version of Prim-Dijkstra algorithm for minimal spanning tree //
  // problem with p, 2≤p≤n processors available //
  process MAIN;
  begin
  // initialization //
1.    T=φ; T_w=0;
2.    j=0; over =0;
3.    for i=2 to p do
4.        create TASK(i);
5.    call TASK(1);
6.    while j<p do wait;
  end;
```

Statements 10 and 24 are two synchronization points in the iteration section of the algorithm. The global variable *over* is set to 0 before any process enters the synchronization block. For further details on the proof of correctness and implementation details see Yoo (1983).

```
process TASK(j);
begin
    for i=j to n by p do
    begin
        NEAR[i]=1; DIST[i]= W[1,i];
    end;
    NEAR[1]=0; DIST[1]= ∞;
    while |T|<n-1 do
    begin
        lock(over);
        over=over+ 1;
        if over=p then
        begin
            Vmin=∞; over=0;
        end;
        unlock(over);
        while over >0 do wait;
        k=1;
        for i=j to n by p do
            if (NEAR[i] ≠ 0) and (DIST[i] < DIST[k]) then k=i;
        lock(Vmin);
        if DIST[k] < Vmin then
        begin
            m=k; Vmin= DIST[k];
        end;
        unlock(Vmin);
        lock(over);
        over=over+ 1;
        if over=p then
        begin
            T=T∪{(m, NEAR[m])}; Tw=Tw+ Vmin;
            NEAR[k]=0; over=0;
        end;
        unlock(over);
        while over >0 do wait;
        for i=j to n by p do
        begin
            if (NEAR[i] ≠ 0) and (W[i,m]<DIST[i]) then
            begin
                NEAR[i]= m; DIST[i]= W[i,m];
            end;
        end;
    end;
end;
```

3.3 Parallel Kruskal's Algorithm

The development of data structures for graph problems has resulted primarily because of the parallel versions of Kruskal's algorithm. Initially, the graph structure consists of a forest of isolated vertices. The edges are sorted in non-decreasing order of their weights, and every edge that connects two disjoint trees is added to the minimum spanning tree. In other words, edges that do not cause cycles with existing edges are selected. The termination of the algorithm takes place when the graph transforms to a single tree. Yoo (1983) has shown that an MIMD computer with $\lceil \log m \rceil$ processors can delete an element from an input of m element heap in constant time. This software pipeline technique is described in Yoo (1983).

Some other references to parallel minimum spanning tree algorithms in the literature are included in Quinn, Deo (1984). A number of different parallel minimum spanning tree algorithms are summarized in Table III.

3.4 Adaptive Parallel Algorithms

Recently, Kwan, Ruzzo (1984) have developed parallel algorithms for finding minimum spanning trees based on Prim's, Kruskal's and Sollin's schemes using the SIMD-SM-R model of computation. The runtime complexites of parallel algorithms based on all the above schemes have been shown to be $O((m \log n)/p)$, where n is the number of vertices, m is the number of edges and p is the number of processors. The salient features of these algorithms are that their running time is a function of the number of processors and edges (and hence are adaptive), and make use of some interesting parallel algorithms for sorting and the union-find problems as intermediate steps. Furthermore, their parallel minimum spanning tree algorithm based on Prim's scheme improves the results obtained by Bentley (1980) and Deo, Yoo (1981) for sparse graphs. Since many applications of the minimum spanning tree problem in computer science involves the use of sparse graphs, improvements obtained by Kwan, Ruzzo (1984) are quite significant.

Table III. Parallel Minimum Spanning Tree Algorithms

Reference	Model	Complexity	Processors
Savage (1977)	SIMD-SM-SR	$O(\log^2 n)$	$O(n^2/\log n)$
Bentley (1980)	Tree	$O(n \log n)$	$n/\log n$
Deo, Yoo (1981)	MIMD-TC	$O(n^{1.5})$	$n^{0.5}$
	MIMD-TC	$O((n^2 \log n)/p)$	$p \le n$
Savage, Ja'Ja' (1981)	SIMD-SM-R	$O(\log^2 n)$	$O(n^2)$
Nath, Maheshwari (1982)	SIMD-SM	$O(\log^2 n)$	$n^2/\log n$
Kucera (1982)	SIMD-SM-RW	$O(\log n)$	n^5
Chin et al. (1982)	SIMD-SM-R	$O(\log^2 n)$	$n \lceil n/\log^2 n \rceil$
Hirschberg (1982)	SIMD-SM-RW	$O(\log n)$	$O(n^3)$
Hirschberg, Volper (1983)	SIMD-SM-RW	$O(\log n)$	$O(n^3)$
Yoo (1983)	MIMD-TC	$O(m)$	$O(m)$
Awerbuch, Shiloach (1983)	SIMD-SM-RW	$O(\log n)$	$2m$
Kwan, Ruzzo (1984)	SIMD-SM-R	$O((m \log n)/p)$	$p \log p \le (m \log n)/n$
	SIMD-SM-R	$O((m \log n)/p)$	$p \le \log n$
	SIMD-SM-R	$O((m \log n)/p)$	$p \le m/\log n$

4. Other Parallel Graph Algorithms

There are numerous parallel graph algorithms in the literature. In this section we present some of the more interesting such algorithms. Some other parallel graph algorithms are also described in Quinn, Deo (1984).

4.1 Parallel Algorithms for Shortest Path Problems

The shortest path problems are perhaps the most important class of problems in the study of communication and transportation networks and many efficient sequential algorithms have been developed for these problems. For an excellent taxonomy on the sequential shortest path algorithms, Deo, Pang (1984) may be referred to. Of relevance here are the so called

(i) the single-source shortest path problem - where the aim is to find the shortest path/distance from a given vertex to all other vertices in a graph and

(ii) the all-pairs shortest path - where the aim is to determine the shortest path/distance between every pair of vertices in a graph.

The shortest path algorithms are classified into two major categories : (a) the label-setting type and (b) the label-correcting type. For details on these types and for other classifications see Deo, Pang (1984). We first briefly outline the well-known algorithm for the single-source shortest path problem due to Dijkstra (1959). This algorithm is of the label-setting category and is presented below as algorithm SP.

---

algorithm SP;

Input: A graph $G = (V, E)$, a source $s \in V$, and a function f from edges to non-negative real numbers.

Output: A shortest path from s to every other vertex of V.

```
begin
    S = {s};
    D[s] = 0;
    for each v in V - S do D[v] = f(s,v);
    while S ≠ V do
    begin
        pick a vertex w in V - S such that D[w] is a minimum;
        S = S ∪ {w};
        for each v in V - S do D[v] = min {D[v],D[w]+ f(w,v)};
    end;
end;
```

The algorithm SP has a time-complexity $O(n^2)$, where n is the number of vertices. The all-pairs shortest path problem can be solved by using the above algorithm n times, for every $v \in V$. This and another algorithm due to Floyd (1962) which makes use of matrix multiplication, have a time-complexity $O(n^3)$.

The assumptions required for the parallel shortest path algorithms described in the literature are as follows :

(i) edges in the graph may have positive, zero or negative weights, but there are no negative weight cycles; and

(ii) edges in graphs must have non-negative weights.

We now consider Kucera's method for solving the all-pairs shortest path problem when edges have non-negative weights. The model of computation used is SIMD-SM-RW where simultaneous writing is allowed only to specific memory locations which can contain the number 0 or 1 only; further all processes writing simultaneously in the same memory location must store the value 1.

---

algorithm ALSP-Kucera;
Input: Adjacency matrix $M[i, j]$ for graph G.
Output: Matrix $S[i,j]$ containing all-pairs shortest distances.
begin
1. $m[i, j] = M[i, j]$ for all i and j;
2. **repeat** $\log_2 n$ times, the following steps :
3. $p[i, j, k] = m[i, j] + m[j, k]$ for all i, j, k;
4. $m[i, j] = \min \{m[i, j], p[i,1, j], \ldots, p[i, n, j]\}$ for all i, j;
end;
5. $S[i, j] = m[i, j], i \neq j$;
6. $S[i, i] = 0$;
end;

The algorithm ALSP-Kucera makes use of a constant time algorithm for finding the minimum of n numbers using $O(n^2)$ processors. The time complexity of the ALSP-Kucera algorithm is $O(\log n)$ using n^4 processors. Further, it is easy to see that the Kucera's minimum finding algorithm can be used to parallelize Dijkstra's algorithm for the single-source shortest path problem in linear-time using $O(n^2)$ processors. Other than these algorithms, Dekel et al. (1981) have obtained parallel algorithms for the all-pairs shortest path problem on the SIMD-Perfect-Shuffle and SIMD-Cube-Connected models. These algorithms have a time-complexity $O(\log^2 n)$ using $O(n^3)$ processors. Further, Levitt, Kautz (1972) have developed a parallel all-pairs shortest path algorithm having a linear time-complexity using $O(n^2)$ processors on the systolic array model. We give a summary of results on the parallel shortest path algorithms in Tables IV and V.

It should be noted that Kucera's algorithm can be modified to yield an $O(\log n)$ algorithm using $O(n^3)$ processors for the single-source shortest path problem.

Table IV. Parallel All-Pairs Shortest Path Algorithms

Reference	Model	Complexity	Processors	Weights
Savage (1977)	SIMD-SM-R	$O(\log^2 n)$	$n^3/\log n$	-
Kucera (1982)	SIMD-SM-RW	$O(\log n)$	n^4	non-negative
Paige, Kruskal (1985)	SIMD-SM-RW	$O(\log n)$	$O(n^{3+\epsilon})$	arbitrary

Table V. Parallel Single-Source Shortest Path Algorithms

Reference	Model	Complexity	Processors	Weights
Mateti, Deo (1982)	MIMD	$O(n^2/p + np)$	p	arbitrary
Yoo (1983)	SIMD-SM-RW	$O(n)$	$O(n^2)$	arbitrary
	SIMD-R	$O(n \log n)$	$O(n)$	arbitrary
	SIMD	$O(mn)$	p	arbitrary
Paige, Kruskal (1985)	SIMD-SM	$O(nm/p + n\log p)$	$p \leq m$	arbitrary
	SIMD-SM-RW	$O(nm/p + n\log\log p)$	$p \leq m$	arbitrary
	SIMD-SM	$O(n^2/p + n\log p)$	$p \leq n$	non-negati\
	SIMD-SM-R	$O((m \log n)/p + n \log n)$	$p \leq d_{out}$	non-negati\
	SIMD-SM-RW	$O(n^2/p + n \log\log p)$	$p \leq n$	non-negati\

4.2 Parallel Matching Algorithms

For a given graph $G=(V,E)$, a set of edges of G is called a *matching* if no two of them have a vertex in common. *Maximum matchings* of a graph G is a matching with maximum cardinality. The matching is said to be *perfect* if it covers all the vertices. Various matching related questions can be formulated for a given graph. For example:

1. Whether a given graph has a perfect matching or not.
2. For a graph with a unique perfect matching, find the perfect matching.
3. Find a maximum matching for a given graph.

Various researchers have investigated these and related problems, however most of the work has been on randomized parallel algorithms, which we are not considering in this survey. We will therefore ignore some very important results of Karp et al. (1985), Mulmuley et al. (1985) etc., and concentrate on deterministic parallel algorithms for matching related problems.

4.2.1 Existence of Perfect Matching

Tutte (1947) gave necessary and sufficient conditions for the existence of a perfect matching in a graph $G=(V,E)$. This test is applied as follows. Let D be the adjacency matrix for G, i.e.

$$d_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise}. \end{cases}$$

Obtain *Tutte Matrix* T of G as follows from D. If $d_{ij}=d_{ji}=1$ then replace these two entries by a unique indeterminate, say x_{ij}, and its negative $-x_{ij}$ so that the entries above the diagonal get the positive sign. Then $|T| \neq 0$ if and only if there exists a perfect matching in G.

4.2.2 Unique Perfect Matching

If the graph G has a unique perfect matching then Rabin, Vazirani (1984) have shown that the perfect matching can be obtained by an NC algorithm. (An algorithm is said to be in NC if the complexity is polylog with polynomial number of processors.) The perfect matching is obtained by inverting the adjacency matrix D of the graph G. The problem of determining whether a graph G has a unique perfect matching has been solved by Kozen et al. (1985). They obtain an NC algorithm by using the matrix obtained by substituting $x_{ij} = 1$ for each indeterminate in Tutte matrix T of G.

Dekel, Sahni (1984) present a parallel algorithm for maximum matching in convex bipartite graphs. Their algorithm has time complexity $O(\log^2 n)$ with $O(n)$ processors on the SIMD-SMM model. They use that algorithm to derive parallel algorithms for scheduling problems.

4.3 Parallel Planarity Testing Algorithms

Ja'Ja', Simon (1982) present two parallel planarity testing algorithms for the SIMD-SM-R model. The first algorithm has a complexity of $O(\log^2 n)$ on $O(n^4)$ processors where the processors are bit processors that cannot add or multiply. The second algorithm has a complexity of $O(\log^2 n)$ on $O(n^{3.29}/\log^2 n)$ processors. Both the algorithms are based on the idea of embedding triconnected graphs in a plane mesh and on the fact that a graph is planar if and only if all its triconnected components are planar (see Tutte 1947). The triconnected components of a graph are found by using a technique that is a generalization of that used by Savage, Ja'Ja' (1981) for finding the biconnected components of a graph.

4.4 Parallel Maximal Independent Set Algorithms

An *independent set* in a graph is a set of vertices, no two of which are adjacent. A *maximal independent set, MIS,* is an independent set that is not properly contained in any independent set. A sequential algorithm for the MIS problem is extremely simple and is given in MIS-SEQ.

---

algorithm MIS-SEQ;
Input. Graph $G=(V,E)$ with vertices $x_1, \cdots, x_n$.
Output. Set I contains a maximally independent set of vertices for graph G.
begin
1. $I = \phi$;
2. **for** $i = 1$ **to** n **do**
3. **if** $x_i \notin N(I)$ **then** $I = I \cup \{x_i\}$;

end;
where $N(I)$ is the neighborhood of any set $I \subseteq V$, defined as follows :
$N(I)=\{x_j \in V \mid \exists\, x_k \in I : (x_j, x_k) \in E\}$

It was conjectured by Valiant (1982) that the MIS problem may not have a fast parallel algorithm despite the fact that it has a very easy sequential algorithm. However, fast parallel algorithms (i.e. in NC) have been devised for this problem. Karp, Wigderson (1985) have a parallel algorithm for the SIMD-SM model that runs in $O(\log^4 n)$ time on $O(n^3/\log^3 n)$ processors. The algorithm of Luby (1984) is for the SIMD-SM model and has complexity $O(\log^2 n)$ with $O(n^2\, m)$ processors.

For the Karp, Wigderson (1985) algorithm, we first introduce some terminology. If $K \subseteq V$, then K is also used to denote the subgraph induced by K — this subgraph has vertex set K and its edge set $E(K)$ consists of those edges of E that have both their end points in K. For $S \subseteq K$, $N_K(S) = N(S) \cap K$. An outline of a parallel algorithm (which is however not efficient) for the MIS problem is given in MIS-KW.

The important point is that they show S can be chosen in $O(\log^2 n)$ time so that $|S \cup N_H(S)| = \Omega(\frac{|H|}{\log |H|})$. I and H can then be updated in $O(\log n)$ time. Therefore, the number of iterations is $O(\log^2 n)$ and hence the time complexity is $O(\log^4 n)$.

Luby (1984) develops three algorithms for the MIS problem. The first two algorithms are randomized algorithms and what is interesting is that these are used to develop a (deterministic) algorithm of complexity $O(\log^2 n)$ with $O(n^2\, m)$ processors. Briefly, the derivation of the deterministic algorithm is as follows. In the first random algorithm, mutually independent choices (for adding a vertex to the independent set) are made thus leading to a probability space with a large sample space. In the second random algorithm, a new probability space with a smaller sample space is obtained by realizing that the choices need

algorithm MIS-KW;
begin
1. $I = \phi$; $H = V$;
2. **while** $H \neq \phi$ **do**
   // I, $N(I)$, H are disjoint and together exhaust vertex set V //
   begin
3. $S =$ an independent set in induced subgraph H;
4. $I = I \cup S$;
5. $H = H - (S \cup N_H(S))$;
   end;
end;

only be pairwise independent. This sample space contains only $O(n^2)$ points and so the deterministic algorithm is obtained by sampling in parallel all the points in the sample space.

4.5 Parallel Algorithms For Euler Circuits

A circuit (path) for a graph $G = (V,E)$ is an Euler circuit (path) if every edge in E appears exactly once. (For simplicity we will consider only the problem of finding Euler circuits.) A serial algorithm for this problem is as follows. For a directed graph the edges are partitioned into circuits and these edge disjoint circuits are combined to give a single Euler circuit by performing a number of 'switches'. These switches can be performed one by one and in any order. However, for an efficient direct parallelization we must be able to perform these switches in parallel.

Awerbuch et al. (1984) and Atallah, Vishkin (1984) both present parallel algorithms for finding Euler circuits for directed and undirected graphs. The time complexity of Awerbuch et al. algorithm is $O(\log |E|)$ with $|E|$ processors on the SIMD-SM-RW model (in case of simultaneous writes any arbitrary processor can succeed). The time complexity of the Atallah, Vishkin algorithm is $O(\log |V|)$ with $|E| + |V|$ processors on the SIMD-SM-RW model (in case of simultaneous writes the lowest numbered processor succeeds). In both cases, an auxiliary graph is constructed from G and that is used to obtain Euler circuits. However, the auxiliary graphs constructed by Awerbuch et al. and Atallah, Vishkin are different.

The algorithm of Awerbuch et al. for a directed graph proceeds as follows.

1. Generate an Euler partition $P = \{C_1, .., C_m\}$.
2. Obtain circuit graph $CG = (P,S)$ where $S = \{(e_i,e_j) \mid (e_i,e_j)$ is a switch with $e_i \in C_h$, $e_j \in C_k\}$.

3. Obtain CG' a sparse connected subgraph of CG.
4 Find a spanning tree T of CG'.
5. Execute the switches of T.

For undirected graphs, they first orient the edges appropriately and then use the algorithm described above.

The algorithm of Atallah, Vishkin for directed graphs proceeds as follows.

1. The edges of $G = (V,E)$ are partitioned into pairwise disjoint circuits, $C_1, .., C_m$.
2. Auxiliary graph $G' = (V \cup \{C_1, .., C_m\}, E')$ is constructed where a vertex v_j in V and C_k are connected if v_j lies on circuit C_k in G.
3. Find a spanning tree T for G'.
4. From T construct a directed graph G'' by replacing each edge in T by two antiparallel edges.
5. Find an Euler circuit for G''.
6. Construct an Euler circuit for G from that of G''.

This algorithm for directed graphs can be used to obtain Euler circuit for undirected graphs by suitably orienting the edges of the input graph.

4.6 Miscellaneous Parallel Graph Algorithms

An undirected graph $G = (V,E)$ is called a ***comparibility*** graph if its edges can be directed to obtain a transitively closed acyclic digraph H. Kozen et al. (1985) give an NC algorithm that checks whether an undirected graph G is a comparability graph, and if so, it also obtains the transitive orientation of the edges.

If σ is a linearly ordered finite set then an ***interval*** α of σ is any set of contiguous elements of σ. Two intervals intersect if and only if they have an element in common. (α,I) where I is a set of intervals on α is an ***interval representation*** of graph $G = (V,E)$ if there exists a bijection $f: V \rightarrow I$ such that $(u,v) \in E$ if and only if the intervals $f(u)$ and $f(v)$ intersect. Graph $G=(V,E)$ is an ***interval graph*** if and only if there exists α and I such that (α,I) is an interval representation of G. Kozen et al. (1985) give an NC algorithm that checks whether a graph G is an interval graph, and if so, it obtains an interval representation for it.

Ghosh (1986) describes parallel algorithms for connectivity problems in graph theory. In particular, algorithms for obtaining a fundamental set of cycles of a graph, for computing the bridges of a connected graph and for strongly orienting a bridgeless graph are given. These algorithms have polylog complexity and require $O(n^3)$ processors. Atallah (1984) also discusses parallel algorithms for strongly orienting undirected graphs.

5. Other Parallel Algorithms

So far we have discussed various graph theoretic algorithms but in this section we consider some other interesting algorithms.

5.1 Parallel Arithmetic Expression Evaluation Algorithms

Dekel, Sahni (1983b) consider the generation of the postfix form of an arithmetic expression starting with its infix form. They also translate the postfix form into a binary tree representation. Their model is the SIMD-SM and uses p processors. The input arithmetic expression can contain operands (constants and simple variables), operators (the binary operators $+$, $-$, $*$, $/$, $\uparrow$ and the unary operators $+$, $-$) and parentheses ('(' and ')').

Their parallel algorithm originates from the standard priority-based sequential infix-to-postfix algorithm POSTFIX (Horowitz, Sahni 1976). If the infix expression is in $E[1..n]$ then for every $E[i]$, define a value $AFTER[i]$ such that $E[i]$ immediately follows $E[AFTER[i]]$ in the postfix form of $E[1..n]$. Determination of $AFTER[i]$ is the first phase of the parallel algorithm; in the second phase the actual postfix form is derived from the values of $AFTER[i]$.

The calculation of $AFTER[i]$ can be done by calculating $U[i]$ the index of the token that pops $E[i]$ from the operator stack and $LU[i]$ the index of the last operator to be unstacked by $E[i]$. To calculate these values, each left parenthesis determines the position of its matching right parenthesis by doing a stable sort. This stable sorting can be done in $O(\log^2 n)$ time using n processors (Preparata 1973). Now all the operators needed for the calculation of $U[i]$, $LU[i]$ can be linked together. $U[i]$ and $LU[i]$ (and hence the $AFTER$ value) can therefore be calculated in $O(\log^2 n)$ time using $O(n)$ processors. The calculation of the actual postfix form is done using the linked list defined by the $AFTER$ values. This can be done using $O(\log n)$ time and n processors by recursively splitting the linked list. Thus the postfix form can be calculated in time $O(\log^2 n)$ using n processors on a SIMD-SM model.

They also give an alternative strategy for computing the values of U and LU. In that strategy a number of copies are made to avoid read conflicts and it leads to an algorithm of complexity $O(\log n)$ using $n^2/\log n$ processors on a SIMD-SM model.

Bar-On, Vishkin (1985) also present a parallel algorithm for obtaining the binary tree representation of an arithmetic expression. The complexity of their algorithm is $O(\log n)$ using $n/\log n$ processors on a SIMD-SM-R model. Even though their model is more powerful than that used by Dekel, Sahni (1983b), the algorithm of Bar-On, Vishkin (1985) can be translated to obtain an algorithm for the SIMD-SM model with time complexity $O(\log^2 n)$ with $n/\log n$ processors. This translation can be obtained by using the simulations of Eckstein (1979a) and Vishkin (1983).

The heart of their algorithm consists of developing an efficient parentheses matching algorithm. This is accomplished by using $n/\log n$ processors and allocating a segment of length $\log n$ to each processor. Each segment consists of

```
algorithm POSTFIX(E,P,n,m);
   // Translate the infix expression E[1..n] into postfix form. The postfix //
   // form is output in P[1..m]. "-∞" is used as bottom of stack character //
   // and has ISP=0. //
begin
   STACK[1] = -∞; top=1; m=0;
   for i = 1 to n do
      if E[i] is an operand then
      begin
         m=m+1; P[m]=E[i];
      end
      else if E[i] = ')' then
      begin
         while STACK[top] ≠ '(' do
         begin // unstack until '(' //
            m=m+1; P[m]=STACK[top]; top=top - 1;
         end;
         top=top - 1;
      end
      else
      begin
         while ISP[STACK[top]] ≥ ICP[E[i]] do
         begin
            m=m+1; P[m]=STACK[top]; top=top - 1;
         end;
         top=top+1; STACK[top]=E[i];
      end;
   while top>1 do
   begin
      m=m+1; P[m]=STACK[top]; top=top - 1;
   end;
end;
```

some pairs of matching parentheses and some parentheses whose matches are in other segments. The matching pairs within a segment can be identified by making one pass through the segment. For the remaining parentheses, their matches are found by performing a variant of binary search on a balanced binary tree that they construct.

This parentheses matching algorithm is made use of as follows. In the original expression, parentheses are introduced so that all subexpressions become simple expressions, where a simple expression is one in which all operators not enclosed by parentheses are of the same precedence. The parentheses matching algorithm is then applied to this expression. At this point the original problem of obtaining a tree form for the input expression has been reduced to finding the tree form of a number of simple expressions. This can be done in a straightforward manner and the result can then be combined to obtain the tree form of the original expression.

Apart from the parentheses matching part, the remaining steps can all be done in $O(1)$ time using n processors and therefore in $O(\log n)$ time using $n/\log n$ processors.

5.2 Parallel Algorithms for Polynomial Evaluation

Polynomial evaluation is a fundamental and widely used operation and the problem of designing efficient parallel algorithms for it has received a great deal of attention in recent years. A polynomial can be expressed in a variety of ways e.g. the Power form, the Root Product form, the Newton form and the Orthogonal form. For details on different representation, see Kronsjo (1979). Different representations lend themselves to different techniques for exploiting parallelism from the sequential polynomial evaluation algorithms.

Munro, Patterson (1973) developed optimal algorithms for polynomial evaluation which use large amount of parallelism. Their algorithm proceeds by defining a recursive evaluation procedure for the polynomial. They ignore all memory access problems and assume the existence of k identical arithmetic processors, each of which can perform any one of the binary operations of $+$, $-$, $*$ and $/$ in unit time. A lower bound of $2n$ arithmetic operations has been shown to be required for the evaluation of a general n degree polynomial. When the degree of the polynomial exceeds $k\lceil \log k \rceil$ then the Munro, Patterson algorithm is optimal within one time unit.

Hyafil (1978) presents a parallel evaluation algorithm of multivariate polynomials. Hyafil's algorithm shows that a polynomial P computed sequentially in $n\{+,-,*\}$ steps with a degree d can be computed in parallel time $O((\log d)(\log n + \log d))$. The number of processors required for this algorithm is $n^{\log d}$ even when n and d are bounded polynomially, in terms of the number of indeterminates. Valiant, Skyum (1981) present an improved version of the above algorithm for any multivariate polynomial. The time complexity is the same as that for Hyafil's algorithm but it uses only $(nd)^{O(1)}$ processors. Another result that uses only a polynomial number of processors is the result of Csanky (1976) for evaluating the determinant.

There are other types of parallel polynomial evaluation algorithms in the literature. Recently Bini, Pan (1985) developed parallel polynomial division algorithms by preserving the full efficiency of the best sequential algorithms. This algorithm divides two polynomials of integer coefficients of degree at most m and can be accelerated by a factor $\log m$. They further show that all the known algorithms for polynomial division can be represented as algorithms for triangular Toeplitz matrix inversion. For improving the efficiency of the determinant and of the inverse of matrix see Pan, Galil (1985). Recent results on algebraic complexity of computing polynomial zeros is described in Pan (1985).

5.3 Parallel String Matching Algorithms

The string matching problem consists of finding all occurrences of a given pattern in a given text. Galil (1984) presents families of parallel algorithms for this problem. For the SIMD-SM-RW model (the only simultaneous writes allowed are when processors attempt to write 0 simultaneously), his algorithms can be summarized as follows.

1. $O(n/p)$ time complexity for any number of $p \leq n/\log n$ processors.
2. Constant time complexity for $p \leq n^{1+\epsilon}$ processors where $\epsilon > 0$.
3. Time complexity $O(\log n/\log \log n)$ for n processors.

For the SIMD-SM-R model, he has an algorithm with $O(n/p)$ time complexity using any number $p \leq n \log^2 n$ processors.

All his algorithms solve the problem under the assumption that both the pattern and the text is over a given alphabet of fixed size. He develops these families of algorithms by first presenting a non-optimal ($O(\log n)$ time with $O(n)$ processors) solution for the special case when the length of the text is twice the length of the pattern. Then he shows that the number of processors required can be reduced to $O(n/\log n)$ by using the four Russian trick (Aho et al. 1974). Finally, he shows how the restriction on the lengths of the pattern and text can be removed and still maintain optimality by appropriate assignment of processors. The resulting algorithms can then be modified to obtain optimal algorithm for the SIMD-SM-R model with the performance described above.

Vishkin (1985) also presents optimal parallel algorithm for this problem. His algorithm has time complexity $O(n/p)$ for any number $p \leq n/\log n$ processors on a SIMD-SM-RW model. (In this paper, the smallest numbered processor attempting to write succeeds; but in a future paper they show that these results are valid when simultaneous writes are allowed only if the processors attempt to write the same value.) In contrast to Galil's algorithms, the symbols are not constrained to be taken from an alphabet of fixed size. Briefly, his algorithm works as follows.

1. Analyze the pattern so as to compute the following table. The ith entry in the table will indicate either that the suffix starting at position i of the pattern is a prefix of the pattern or not (in the later case it provides a counterexample).
2. This table allows us to infer for appropriate indices, j_1 and j_2 in the text that an occurrence of the pattern can not start at both j_1 and j_2. Therefore a sparse set of indices where an occurrence of the pattern may start can be obtained.
3. A character by character test is applied to the indices obtained in the previous step to determine whether an occurrence of the pattern in fact occur at those locations.

5.4 Parallel Tree Balancing Algorithms

The task of balancing a binary tree is to adjust the left and right pointers of the vertices in the tree so that the height of the tree is minimized. In Moitra, Iyengar (1986), an optimal parallel algorithm for obtaining route balanced binary trees is derived. (In a route balanced binary tree all the leaves at the highest level are located at the leftmost side of the tree.) In this section unless otherwise stated, a balanced tree will mean a route balanced binary tree. The balancing algorithm is systematically derived by starting from a recursive algorithm and transforming it first into an iterative algorithm and then into an optimal parallel algorithm. This parallel algorithm has time complexity 0(1) with n processors on a SIMD-SM model and can be described as follows.

The algorithm has two parts; in the first part a parallel inorder traversal of the binary tree is performed and the result is stored in an array *LINK*. In the second part, a balanced tree is built from the array *LINK*. The parallel traversal can be done in constant time if additional information is stored with each vertex giving the number of vertices with values less than or equal to its own value.

To describe the building up process, we introduce some terminology. Let $<i,j>$'th vertex refer to the jth vertex from the left on the ith level if it exists in a binary tree. Also, let $VAL_N(<i,j>)$ denote the number of vertices with data values less than or equal to data value of the vertex $<i,j>$ in a balanced tree of size N. It can be shown that

$$VAL_N(<i,j>) = \min \{VAL_{2^n-1}(<i,j>),[VAL_{2^n-1}(<i,j>)+N-S+1]/2\}$$
where $n=\lceil \log(N+1) \rceil$, $VAL_{2^n-1}(<i,j>) = 2^{n-i}+(j-1)*2^{n-i+1}$
and $S=2^n-1-N$

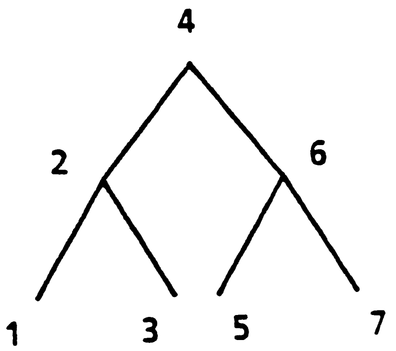
This gives a method of determining which cell in the array *LINK* corresponds to the $<i,j>$th vertex in the final balanced tree. To construct the balanced tree, a processor P_K is associated with $LINK[K]$ and it must do the following

C1. determine i, j such that $VAL_N(<i,j>)=K$; that is its final position in the balanced tree.

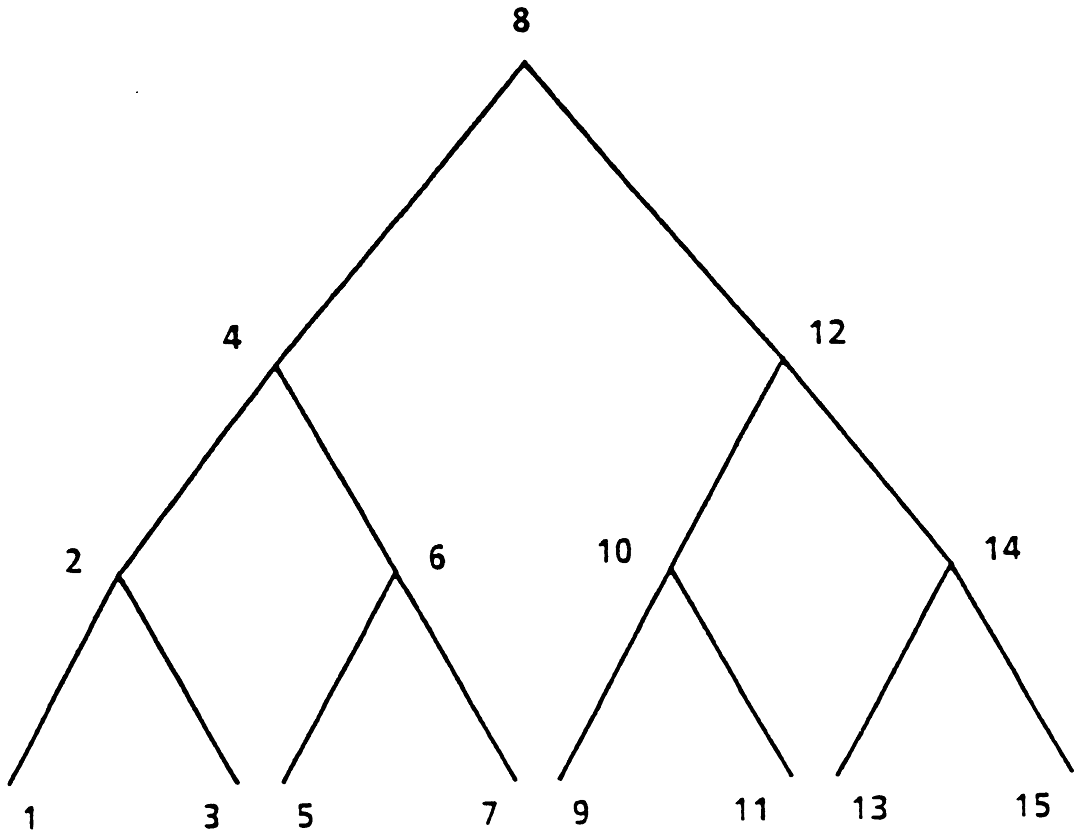
C2. if it has a left son then determine $K1$ such that $VAL_N(<i+1,2*j-1>)=K1$

C3. if it has a right son then determine $K2$ such that $VAL_N(<i+1,2*j>)=K2$

Computations C2 and C3 can be easily done in constant time once computation C1 has been done. For simplicity consider the computation C1 when $N=2^n-1$. Note that if processor P_K is allocated to process the vertex with *VAL* equal to K then P_K may correspond to different indices $<i+M,j>$, $M=1, 2, ..$ depending on the size of the balanced tree. However, the values $n-i$ and j are invariant over the size of complete balanced binary tree as shown in Fig. 2.



(a) vertex with value 6 has $j=2$, $n-i=2$



(b) vertex with value 6 has $j=2$, $n-i=2$

FIG. 2. Vertex identification that is invariant over size of complete

balanced binary tree.

So, if the values $n-i$ and j are permanently associated with each processor, then computation C1 can also be done in constant time. Building up the balanced tree is given in algorithm COM-BAL.

```
algorithm COM-BAL;
// parallel growing algorithm for the construction of a balanced tree by //
// simultaneously executing processes P_1, .., P_N, one for each cell in the //
// array LINK, where N=2^n-1 //
begin
    for each P_K do in parallel              // h, j associated with each P_K //
    begin
        n=⌈log(N+1)⌉;
        i=n-h;
        if odd (K) then                      // leaves //
        begin
            LSON[LINK[K]]=null;
            RSON[LINK[K]]=null;
        end;
        if even (K) then                     // interior vertices //
        begin
            LSON[LINK[K]]=LINK[K-2^(n-i-1)];
            RSON[LINK[K]]=LINK[K+2^(n-i-1)];
        end;
    end;
end;
```

For arbitrary binary trees, cell $LINK[K]$ may correspond to different indices $<i,j>$ and $<i',j'>$ with $i \neq i'$ and $j \neq j'$ as the size of the tree changes thereby making computation C1 expensive. The computation can however be reorganized so that a processor $P_{h'}$ sets up links for $LINK[h]$ where

$$h = \min \{h', [h' + N - S + 1]/2\}$$

This allows for a simpler calculation of indices $<i,j>$. However, a direct implementation of this idea would require the use of 2^n-1 processors where $n = \lceil \log(N+1) \rceil$ for a tree of size N. To reduce the number of processors so that only N processors are required to grow a tree of size N further reorganization of the computation has to be done. All the details are presented in algorithm BAL.

The result presented above can be extended to balance m-way trees as discussed in Dekel et al. (1985).

```
     algorithm BAL;
     // parallel growing algorithm for the construction of a balanced tree by //
     // executing simultaneously processes P_1, .., P_N //
     begin
1.     for each P_K do in parallel
       begin
2.       n=⌈log(N+1)⌉; i=n-h; S=2^n-1-N;
3.       K=2^(n-i)+(j-1)*2^(n-i+1); K'=(K+N-S+1)/2;
4.       if K'>K then K'=K;                    // K'=VAL_N(<i,j>) //
5.       if odd (K) and (K=K') then            // leaves on the highest level //
         begin
6.         LSON[LINK[K']]=null; RSON[LINK[K']]=null;
         end;
7.       if odd (K) and (K≠K') then            // reorganization of computation //
         begin
8.         K=K+S; K'=(K+N-S+1)/2;
9.         if K'>K then K'=K;
         end;
10.      if even (K) then
         begin
11.        K1=2^(n-i+1)+(2*j-2)*2^(n-i)        // set up left link //
12         K1'=(K1+N-S+1)/2
13.        if K1'>K1 then K1'=K1               // K1'=VAL_N(<i+1,2*j-1>) //
14.        if odd (K1) and (K1≠K1')            // odd (K1) ⇒ K' on level n-1 //
15.          then LSON[LINK[K']]=null
16.          else LSON[LINK[K']]=LINK[K1'];
17.        K2=2^(n-i-1)+(2*j-1)*2^(n-i)        // set up right link //
18.        K2'=(K2+N-S+1)/2
19.        if K2'>K2 then K2'=K2               // K2'=VAL_N(<i+1,2*j>) //
20.        if odd (K2) and (K2≠K2')            // odd (K2) ⇒ K' on level n-1 //
21.          then RSON[LINK[K']]=null
22.          else RSON[LINK[K']]=LINK[K2'];
           end;
       end;
     end;
```

5.5 Parallelism In Alpha-Beta Algorithms

Efficient search of trees is fundamental to many areas of study such as Artificial Intelligence, Operations Research and Computer Science at large. Most AI problems, especially game playing programs use heuristic information to direct the search that is relevant to the goal.

It has been well documented in Berliner (1978) that an important component of search trees in game playing is the speed at which the search is conducted. The focus of this section is to explore parallelism in alpha-beta search algorithm used extensively in game trees. We first present definition necessary for the formulation of alpha-beta search algorithm (Knuth, Moore 1975, Nilsson 1980, Slagle, Dixon 1969, Pearl 1982). Then we describe a technique developed by Finkel, Fishburn (1982) to exploit parallelism in alpha-beta search algorithm.

5.5.1 Definitions And Formulation Of Alpha-Beta Search Algorithm

Decision and game trees have been extensively used in AI problems such as chess playing programs. However since the size of decision or game tree encountered in such problems is very large, it becomes physically impossible to completely search a tree. Thus it becomes important to develop an efficient method for searching and/or pruning the tree; for more details see (Knuth, Moore 1975, Nilsson 1980, and Slagle, Dixon 1969).

We restrict the word game to mean a two person, zero sum, perfect information The play strictly alternates between two players. Each player, on his move is allowed to choose a move from only a finite number of possible moves. The game is characterized by a set of positions and by a set rules for moving from one position to another.

If from a given state or position P, a player is allowed to move to any of the possible P_1, .., P_k positions, then in the corresponding game tree, one would find a connecting branch from vertex P to each of the P_1, .., P_k vertices. There are two procedures for assigning a static value to the vertices of a decision or game tree. They can be described as follows :

1) Minimax procedure
2) Negamax procedure

In minimax procedure even level vertices (0, 2, 4, ..) (the root is at level 0) are called MAX vertices and odd level vertices (1, 3, ..) are called MIN vertices. Player1 moves from MAX vertices and player2 moves from MIN vertices. Note that this notation of naming the vertices and assigning each player to even or odd vertices is arbitrary and is chosen by convention. Having named the vertices, a static evaluation function is used to preassign a value to the terminal vertices. After assigning these values, the values for all other non-terminal vertices up to the root are calculated. The procedure used is as follows :

For each MAX vertex, the value assigned is the maximum value of it's sons.
For each MIN vertex, the value assigned is the minimum value of it's sons.

This procedure is repeatedly used until the value of the root vertex is calculated. The thinking behind this procedure is that each player wants to maximize its own gain and minimize it's loss to the other player. In minimax procedure we always evaluate the game position from any one of the player's viewpoint.

In the negamax procedure there is no differenciation between MAX and MIN vertices. It assigns a static value to a terminal vertex say $f(p)$ from the point of view of a player whose turn it is to make a move. The value from the other player's point of view is $-f(p)$. If there are more than one possible moves, one needs to select the best move, which will result in the greatest possible gain when the game ends. Thus the value assigned to a non-terminal vertex is the maximum of negative of it's sons. Remember that the value assigned to a terminal vertex is from the point of view of player whose turn it is to make a move and that the players strictly alternate. Negamax approach is useful when it is undesirable to deal with two or more separate cases.

5.5.2 Pruning Strategy

Having described the basic background material, let us now describe one of the possible pruning strategy. As remarked earlier the size of a decision or game tree is extremely large in most AI applications, making pruning desirable. The basic thinking behind pruning is as follows : if we can somehow determine that a given vertex can never improve the value of it's parent, it is not necessary to continue the search beyond that vertex. The subtree below that vertex can be cutoff, thus pruning the entire tree.

It is obvious that the negamax procedure as described above is not efficient as it makes no attempt to use the information content of vertices processed earlier. Thus our line of thinking should necessarily include the use of that information to improve the basic negamax procedure. We illustrate this point by the example in Fig. 3 in which we use the information available from earlier vertices to prune the tree. In Fig. 3(a) the negamax procedure tells us that

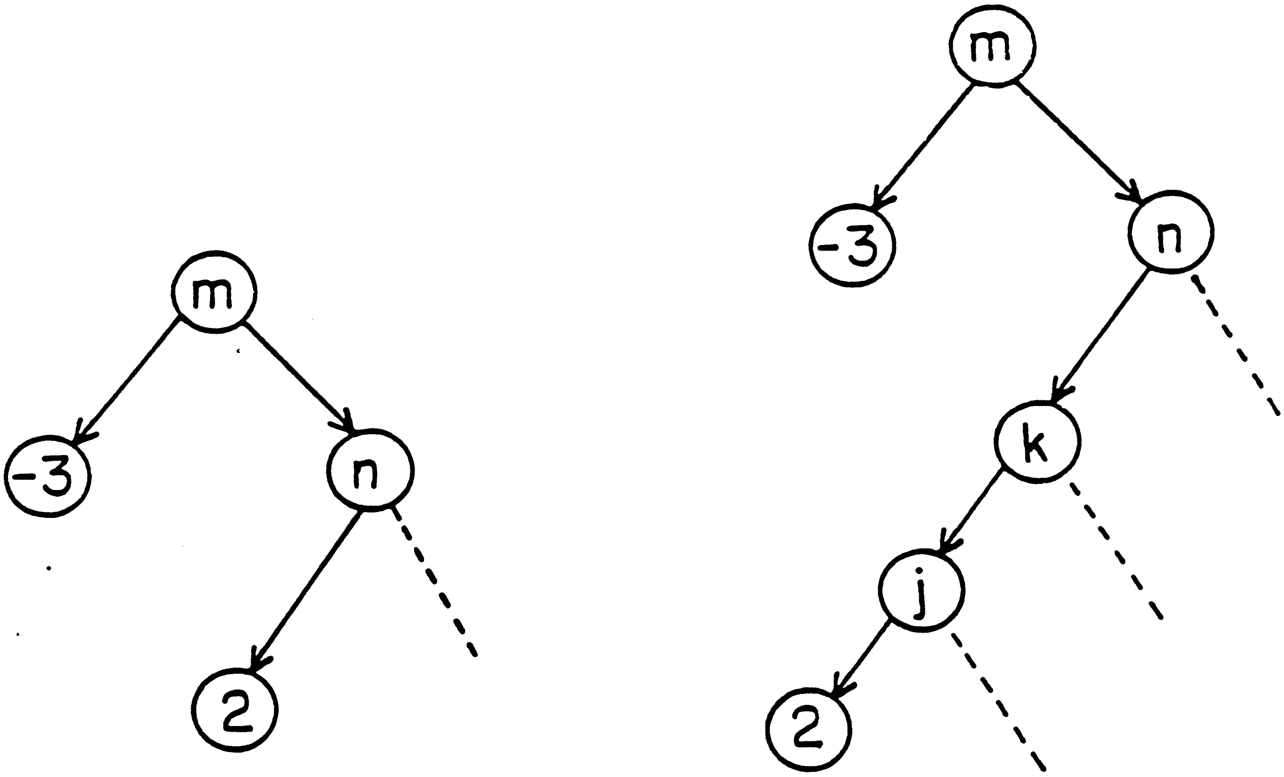
$$m = \max \{ 3 , -n \}$$
$$\text{where } n = \max \{ -2, .. \}$$

Hence $n \geq -2$ and so whatever may be the value of n, we will always get $m = 3$. Thus we need not search the vertex n and everything below vertex n. This gives shallow cutoffs.

Following this argument for more than one level gives us deeper cutoffs. Consider the subtree below and inclusive of vertex k in Fig. 3(b) where $k = \max \{ -j, .. \}$ and $j = \max \{ -2, .. \}$. This gives us two possible cases, viz $k > -j$ or $k = -j$.

Consider $k > -j$. This means that the value of k is not determined by vertex j but is determined by siblings to the right of vertex j. Thus subtree rooted at vertex j can be cutoff.

Consider $k = -j$. Since $j \geq -2$ we get $k \leq 2$. Now $n \geq -k$ and therefore $-n \leq 2$. Since $m = \max \{3, -n\}$ and $-n \leq 2$ we get $m = 3$ which is independent of value of n. Thus the subtree rooted at vertex n can be cutoff.



(a). Shallow cutoff. (b). Deeper cutoff.

FIG. 3. Pruning of search tree.

5.5.3 Definition Of Alpha-Beta Bounds

Previous examples illustrate that by using the information obtained from earlier vertices, we can prune the decision or game tree to a considerable extent.

Using two bounds α and β, we get the Alpha-Beta search algorithm as described by Knuth, Moore (1975).

```
1.  function alphabeta (p : position; α , β : integer) : integer;
2.       var i, d : 1..MAXCHILD;
3.            succ : array [1..MAXCHILD] of position;
    begin
4.  determine the successor position succ[1], .., succ[d];
5.  if d = 0 then return (static_value (p));              // terminal vertex //
6.  for i = 1 to d do
     begin
7.     α = max {α, - alphabeta (succ[i],-β,-α)};
8.     if α ≥ β then return (α);                         // cutoff //
     end;
9.  return (α);
    end;
```

We can thus summerize the characteristics of the function alphabeta as follows :

a) For a given position p and for values of α, β with $\alpha < \beta$
   1) if negamax(p) $\leq \alpha$ then alphabeta $(p,\alpha,\beta) \leq \alpha$
   2) if negamax(p) $\geq \beta$ then alphabeta $(p,\alpha,\beta) \geq \beta$
   3) if $\alpha <$ negamax(p) $< \beta$ then alphabeta $(p,\alpha,\beta) =$ negamax(p)

b) The pair (α,β) is called the window for the search.

c) The alpha-beta search algorithm is strongly serial in the sense that the earlier work is used in deciding cutoffs at the later stages.

5.5.4 Parallelization Of Alpha-Beta Search Algorithm

Attempts aimed at a parallel version of alpha-beta include parallel "aspiration search" by Baudet (1978), "mandatory-work-first" by Akl, Barnard, Doran (1982), tree splitting algorithm by Finkel, Fishburn (1982) and Kumar, Kanal (1984). We now look more closely at the algorithm of Finkel, Fishburn.

Finkel, Fishburn (1982) present a parallel version of alpha-beta search algorithm. This tree splitting algorithm speeds up the search of a large tree of potential continuations by dynamically assigning subtree searches for parallel execution. The root processor evaluates the root position. Each of the interior processors evaluate the position assigned to it by generating successors, queuing them for parallel execution to it's slave processor. A processor tree is used in the execution of this parallel algorithm. Thus a processor which is at the kth level in the processor tree corresponds to the position at level k in the lookahead tree used by the game playing expert system. The terminal processor evaluates it's position using a serial alpha-beta algorithm. Thus on completion of it's slaves evaluation (or at cutoff) the master can evaluate it's position. The masters inform their working slaves about the updated narrow window as it receives information from it's slaves.

As is clear from the above discussion, there are two algorithms viz. the leaf algorithm and the interior algorithm. We now describe these algorithms in more detail.

5.5.5 The Leaf Algorithm

We can describe the interaction of algorithm Leaf with its master either by remote procedure calls or by message passing/shared memory form. In the algorithm described below the master calls the function leaf$\alpha\beta$ remotely (line 11) and interrupts the active slaves, to inform them of updated windows, by invoking the asynchronous procedure update (line 2).

5.5.6 The Interior Algorithm

As we described earlier, the slaves report to the master. The creation and co-ordination of the slaves is done by the master. We now give this algorithm and discuss how it works.

```
algorithm Leaf;
α,β : array [1..MAXDEPTH] of integer;  // global arrays //
asynchronous procedure update(newα,newβ : integer);
  // the asynchronous procedure update is called by my master to //
  // inform me of new window (newα, newβ) //
var tmp : integer;
    k : 1..MAXDEPTH;
begin
   for k = 1 to MAXDEPTH do
   begin                                    // update arrays α and β //
      α[k] = max {α[k],newα};
      β[k]  = min {β[k],newβ};
      tmp  = newα;
      newα  = - newβ;
      newβ  = - tmp;
   end; // update //
end;

function leafαβ(p : position; α,β : integer) : integer;
begin
   α[1] = α;
   β[1] = β;
   return(alphabeta(p,1));
end;

function alphabeta(p : position; depth : integer) : integer;
var succ : array [1..MAXCHILD] of position;     // successors //
    succno : 1..MAXCHILD;                        // which successors //
    succlim : 1..MAXCHILD;                       // how many successors //
begin
   determine the successors succ[1], .., succ[succlim];
   if succlim = 0 then return (static_value(p));
   for succno = 1 to succlim do
   begin                                         // evaluate each successor //
      α[depth + 1] = -β[depth];
      β[depth + 1] = -α[depth];
      α[depth] = max {α[depth], -alphabeta(succ[succno],depth + 1)};
      if α[depth] ≤ β[depth] then
            return(α[depth]);                    // a cutoff has occurred //
   end; // for succno //
   return (α[depth]);
end; // alphabeta //
```

```
algorithm Interior;
1.  var g1α, g1β : integer;              // global variables //
2.      q : integer;                      // variable used for the depth of tree //
3.  asynchronous procedure update(newα,newβ : integer);
      // update is called asynchronously by my master to inform me of the //
      // new window (newα, newβ) //
    begin
4.     atomically do
       begin
5.         g1α = max {g1α,newα};
6.         g1β = min {g1β,newβ};
       end; // of atomically do //
7.     for all slaveid do slaveid.update(-g1β,-g1α);
    end; // update//

8.  function interiorαβ(p : position; α,β : integer) : integer;
9.  var succ : array [1..MAXCHILD] of position;   // for denoting successors //
10.     succno : 1..MAXCHILD;                      // the number of successors //
11.     succlim : 1..MAXCHILD;                     // limit on number of successors //
12.     tmp : array [1..MAXCHILD] of integer;
13.     g : integer;
    begin
14.    g1α = α; g1β = β;
15.    determine the successors succ[1], .., succ[succlim];
16.    if succlim = 0 then return(static_value(p));
17.    if depth(succ[1]) < q
18.        then g = interiorαβ
19.        else g = leafαβ;
20.    parfor succno = 1 to succlim do              // parallel operation //
       begin
21.       slaveid = idle_slave();
22.       if g1α < g1β then
          begin
23.          tmp[succno]-slaveid.g(succ[succno],-g1β,-g1α);
24.          if tmp[succno] > g1α then
             begin
25.             atomically do g1α = max {tmp[succno],g1α};
26.             for all slaveid do slaveid.update(-g1β,-g1α);
             end;
          end;
       end; // of parfor //
27.    return(g1α);
  end; // interiorαβ //
```

The interior algorithm generates all successors of the position to be evaluated (as indicated in line 15). If there are sufficient number of slaves available, then each slave is assigned to evaluate one of the position. Otherwise a queue is formed. Although there are several ways of forming the queue, the queue is implemented as a *parallel for* loop (starting at line 20). One of the following action is taken when one of the slave returns :

1) If the returned value causes the value of current α to increase, then all active slaves are informed of the updated value (line 26).
2) If α increases beyond β then a cutoff has occurred. Since a non-positive updated window is now sent to slaves (line 26) the slaves get terminated. The queue of waiting successors positions (line 22) is emptied.
3) On a update interrupt from the master, the new window is transmitted to the slaves.

When all successors are evaluated, the final value is returned to its master (line 27).

5.5.7 Optimizations

There are several ways in which this algorithm can be optimized. These can be listed as follows :

1) Majority of the time, a master is waiting for messages from the slaves. Hence this waiting time can be used for subtree searches. However since slaves have further slaves below them to help, masters are slower than slaves. Hence only deep masters can use this optimizations.
2) One can group several higher level masters onto a single processor.
3) The root processor can send a special window $(-\alpha-1,\alpha)$ instead of $(-\beta,-\alpha)$. The narrower window speeds up the search.

Finkel, Fishburn (1983) discuss some improved speed up bounds for parallel alpha-beta search.

6. CONCLUSIONS

The design of parallel algorithms for various parallel computer architectures is motivated by factors such as speed and the need to solve complex problems of practical interest. With the continuing decrease in hardware cost, the objective is to trade the number of processors for a gain in computational speed. The advent of commercial multiprocessors has sparked off considerable interest in the study and development of parallel algorithms. An extensive study of parallel algorithms for useful computational problems calls for a systematic and thorough investigation of research efforts spanning over a decade. The attempt we have made here is to illustrate and emphasize the following aspects :

1. The need to systematize the study of parallel algorithms and the design process.
2. To survey the various techniques employed in exploiting the inherent parallelism in problems.
3. To unify parallel algorithms by means of well-defined paradigms.
4. To present a bird's eye-view of some of the current research efforts in the area of concurrent algorithms.

These are expected to result in a better understanding of the working as well as the development of parallel algorithms. In view of the ongoing Fifth Generation Computer Project and the increased use of high speed super computers, a detailed study of the techniques for the design of parallel algorithms becomes quite important.

Due to the growing number of parallel computer architectures and the algorithms developed on these for a large class of problems, it has become increasingly difficult for a user to select a particular parallel algorithm for any given application. In fact, we feel that such a choice is usually decided by factors such as ease of implementation of the algorithms and cost-effectiveness of the computers. What best could be done under such circumstances is to identify some of the inherent features of the parallel algorithms which are invariant and base one's judgement on a critical analysis of these.

The notions of time and space complexity carry forward from the sequential algorithmic domain and are indeed helpful in making an appropriate choice of a parallel algorithm. But it is often difficult to analyze the complexity of parallel algorithms on MIMD machines because of the close interleaving and the indeterminacy of communication and synchronization costs. Such asynchronicity has benefits in the form of the ease of design of parallel algorithms, usually from an efficient sequential version, and more generality. The synchronous models of parallel computation such as the SIMD machines have come to be accepted as useful models because of their simple structures and amenability to formal analysis. In fact, the complexity classes with regard to parallel computation have been mostly for the synchronous models of computation. Still, it should be observed here that unlike in the case of sequential algorithms there is no 'satisfactory' model of parallel computation.

A large number of results on developing efficient parallel algorithms have enriched the complexity classes and thrown open newer issues. The notion of P-completeness is a useful analogue of NP-completeness in the sense that just as deterministic polynomial time sequential algorithms are unlikely for NP-complete problems, polylog parallel time algorithms are unlikely for P-complete problems. The class NC (Nick's class) characterizes the problems having efficient parallel algorithms. Though it is known that NC is a subset of P, it is widely believed that the inclusion is proper. Problems such as depth-first search, unification etc., are examples of problems in P but not known to be in NC. Further they are not likely to be in NC, because they have been shown to be P-complete. The question is whether P-complete problems can be efficiently parallelized in some sense. Recently, Vitter, Simons (1986) have proposed some new complexity classes for parallel algorithms called PC and PC*. These classes of problems do not have a speedup comparable to those in NC. The speedup is usually by a constant factor or proportional to the number of processors used. Another important point to be observed is that, the parallel algorithms in the literature have been mostly on problems in P. This is understandable since for NP-hard algorithms we cannot devise a polynomial-time parallel algorithm with a polynomial number of processors, unless P = NP. Browning (see Mead, Conway 1980) has proposed a tree machine with an exponential number of processors and using this he has developed polynomial-time algorithms for some NP-complete graph problems. It is to be borne in mind that quite a large number of useful problems are NP-complete. Though we cannot expect to achieve a significant speedup by parallel algorithms for these problems, such a study has not been undertaken so far in a systematic way. It is possible to achieve some useful results in the speedup for these problems using polynomial number of processors. Research efforts are needed in this direction.

We can view the paradigms discussed here at two different levels: (i) abstract level and (ii) concrete level (relatively). At the abstract level the paradigms for the design of parallel algorithms can be split up into three categories.

1. Parallelizing existing sequential algorithms,
2. Deriving by formal means a parallel algorithm from a sequential algorithm (this is a special case of (1) above and the emphasis here is on the derivation technique), and
3. By using elegant characterizations and identifying computationally efficient properties.

Under this setting, almost all the MIMD algorithms fall in category (1). This includes the algorithms of Prim-Dijkstra and Kruskal for the minimum spanning tree problem. Algorithms such as the parallel connected components, parallel biconnected components, maximum matching, maximal independent set fall into the category (3). Parallel algorithms for balancing trees of section 5 typify category (2). At the concrete level we discuss paradigms like the binary-tree method, growing by doubling and spanning trees for graphs.

We also have substantiated these techniques with suitable examples. We would like to emphasize here that there is no universal method for designing parallel algorithms. Usually the trick lies in identifying the right formulation of the problem amenable to parallelization and the use of one or more of the techniques we have elucidated.

References

Aho, A., Hopcroft, J., and Ullman, J. (1974). "The Design and Analysis of Computer Algorithms." Addison-Wesley, Reading, Massachusetts.

Akl, S.G., Barnard, D.T., and Doran, R.J. (1982). Design, analysis, and implementation of a parallel tree search algorithm. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-4*, 192-203.

Arjomandi, E., and Corneil, D.G. (1975). Parallel computations in graph theory. *Proc. of the 16th Annu. Symp. on Foundations of Comput. Sci.*, 13-18.

Atallah, M.J. (1984). Parallel strong orientation of an undirected graph. *Inf. Process. Lett. 18*, 37-39.

Atallah, M.J., and Kosaraju, S.R. (1985). A generalized dictionary machine for VLSI. *IEEE Trans. Comput. C-35* (2), 151-155.

Atallah, M., and Vishkin, U. (1984). Finding Euler tours in parallel. *J. of Comp. and Sys. Sci. 29* (3), 330-337.

Awerbuch, B., Israeli, A., and Shiloach, Y., (1984). Finding Euler circuits in logarithmic parallel time. *Proc. of the 16th Annu. ACM Symp. Theory Comput.*, 249-257.

Awerbuch, B., and Shiloach, Y., (1983). New connectivity and MSF algorithms for Ultracomputer and PRAM. *Proc. 1983 Int. Conf. on Parallel Process.*, Aug 1983, 175-179.

Bar-On, I., and Vishkin, U. (1985). Optimal parallel generation of a computation tree form. *ACM Trans. Program. Lang. Syst. 7* (2), 348-357.

Baudet, G.M. (1978). The design and analysis of algorithms for asynchronous multiprocessors. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Bentley, J.L. (1980). A parallel algorithm for constructing minimum spanning trees. *J. Algorithms 1* (1), 51-59.

Bentley, J.L., and Kung, H.T. (1979). A tree machine for searching problems. *Proc. of the 1979 Int. Conf. on Parallel Process.*, 256-266.

Berliner, H.J. (1978). A chronology of computer chess and its literature. *Artif. Intell. 10* (2), 201-214.

Bini, D., and Pan, V. (1985). Parallel polynomial division can be accelerated preserving full efficiency of the best sequential algorithms. Computer Science Dept., Tech. Rep. 85-3, State University of New York at Albany.

Bitton, D., DeWitt, D.J., Hsaio, D.K., and Menon, J. (1984). A taxonomy of parallel sorting. *ACM Comput. Surv. 16* (3), 287-318.

Chandra, A.K. (1976). Maximal parallelism in matrix multiplication. IBM Tech. Rep. RC 6193, Thomas J. Watson Research Center, Yorktown Heights, N.Y.

Chen, S. (1975). Speedups of iterative programs in multiprocessing systems. Ph.D. dissertation, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign.

Cheriton, D., and Tarjan, R.E. (1976). Finding minimum spanning trees. *SIAM J. Comput. 5* (4), 724-742.

Chin, F.Y., Lam, J., and Chen, I.-N. (1981). Optimal parallel algorithms for the connected component problem. *Proc. of the 1981 Int. Conf. on Parallel Process.*, 170-175.

Chin, F.Y., Lam, J., and Chen, I.-N. (1982). Efficient parallel algorithms for some graph problems. *Commun. ACM 25* (9), 659-665.

Cook, S.A. (1981). Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathematique XXVII*, 99-124.

Csanky, L. (1976). Fast parallel inversion algorithms. *SIAM J. on Comput. 5* (4), 618-623.

Dekel, E., and Sahni, S. (1983a). Binary trees and parallel scheduling algorithms. *IEEE Trans. Comput. C-32* (3), 307-315.

Dekel, E., and Sahni, S. (1983b). Parallel generation of postfix and tree forms. *ACM Trans. Program. Lang. Syst. 5* (3), 300-317.

Dekel, E., and Sahni, S. (1984). A parallel matching algorithm for convex bipartite graphs and applications to scheduling. *J. of Parallel and Distributed Comput. 1*, 185-205.

Dekel, E., Nassimi, D., and Sahni, S. (1981). Parallel matrix and graph algorithms. *SIAM J. Comput. 10* (4), 657-675.

Dekel, E., Peng, S., and Iyengar, S.S. (1985). Optimal parallel algorithm for constructing and maintaining a balanced m-way search tree. Dept. Mathematical Sciences, Tech. Rep. CS-209, Univ. of Texas at Dallas, Dallas, Texas.

Deo, N., and Yoo, Y.B. (1981). Parallel algorithms for the minimum spanning tree problem. *Proc. of the 1981 Int. Conf. on Parallel Process.*, 188-189.

Deo, N., Pang, C.Y. (1984). Shortest path algorithms: taxonomy and annotation. *Networks 14* (2), 275-324.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numer. Math. 1*, 269-271.

Eckstein, D.M. (1979a). Simultaneous memory access. Tech. Rep. 79-6, Dept. of Computer Science, Iowa State Univ. of Science and Technology, Ames, Iowa.

Eckstein, D.M. (1979b). BFS and biconnectivity. Tech. Rep. 79-11, Dept. of Computer Science, Iowa State Univ. of Science and Technology, Ames, Iowa.

Finkel, R.A., and Fishburn, J.P. (1982). Parallelism in alpha-beta search. *Artif. Intell. 19*, 89-106.

Finkel, R.A., and Fishburn, J.P. (1983). Improved speedup bounds for parallel alpha-beta search. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-5* (1), 89-92.

Fisher, A.L. (1984). Dictionary machines with a small number of processors. *Proc. of the 11th Annu. ACM Conf. Comput. Architecture*, 154-156.

Floyd, R.W. (1962). Algorithm 97: Shortest path. *Commun. ACM 5* (6), 362.

Galil, Z. (1984). Optimal parallel algorithms for string matching. *Proc. of the*

16th Annu. ACM Symp. Theory Comput., 240-248.

Gentleman, W.M. (1978). Some complexity results for matrix computations on parallel processors. *J. Assoc. Comput. Mach. 25* (1), 112-115.

Ghosh, R.K. (1986). Parallel algorithms for connectivity problems in graph theory. *J. Comput. Math. 18*, 193-218.

Hirschberg, D.S. (1976). Parallel algorithms for the transitive closure and the connected component problems. *Proc. of the 8th Annu. ACM Symp. Theory Comput.*, 55-57.

Hirschberg, D.S. (1978). Fast parallel sorting algorithms. *Commun. ACM 21* (8), 657-661.

Hirschberg, D.S. (1982). Parallel graph algorithms without memory conflicts. *Proc. of the 20th Allerton Conf.*, 257-263.

Hirschberg, D.S., Chandra, A.K., and Sarwate, D.V. (1979). Computing connected components on parallel computers. *Commun. ACM 22* (8), 461-464.

Hirschberg, D.S., and Volper, D.J. (1983). A parallel solution for the minimum spanning tree problem. *Proc. of the 17th Annu. Conf. Inf. Sci. Syst.*, 680-684.

Horowitz, E., and Sahni, S. (1976). "Fundamentals of Data Structures." Computer Science Press, Woodland Hills, Calif.

Hyafil, L. (1978). On the parallel evaluation of the multivariate polynomials. *Proc. 10th ACM Symp. on Theo. of Comput.*, 193-195.

Ja'Ja', J., and Simon, J. (1982). Parallel algorithms in graph theory: planarity testing. *SIAM J. Comput. 11* (2), 314-328.

Karp, R.M., and Wigderson, A. (1985). A fast parallel algorithm for the maximal independent set problem. *J. Assoc. Comput. Mach. 32* (4), 762-773.

Karp, R.M., Upfal, E., and Wigderson, A. (1985). Constructing a perfect matching is in random NC. *Proc. of the 17th Annu. ACM Symp. Theory Comput.*, 22-32.

Keller, R.M. (1973). Parallel program schemata and maximal parallelism I: fundamental results. *J. Assoc. Comput. Mach. 20*, 514-537.

Knuth, D.E., and Moore, R.W. (1975). An analysis of alpha-beta pruning. *Artif. Intell. 6* (4), 293-326.

Kozen, D., Vazirani, U.V., and Vazirani, V.V. (1985). NC algorithms for comparibility graphs, interval graphs, and testing for unique perfect matching. *Fifth Conf. on Foundations of Software Tech. and Theoretical Comp. Sc.*

Kronsjo, L. (1979). "Algorithms: their complexity and efficiency." John Wiley and Sons Ltd.

Kruskal, J.B. (1956). On the shortest subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc. 7*, 48-50.

Kucera, L. (1982). Parallel computation and conflicts in memory access. *Inf. Process. Lett. 14* (2), 93-96.

Kuck, D.J. (1976). Parallel processing of ordinary programs. in *Advances in*

Computers, vol. 15, M. Rubinoff and M. Yovits, Eds. Academic Press, New York, 119-179.

Kuck, D.J. (1977). A survey of parallel machine organization and programming. *ACM Comput. Surv. 9*, 29-59.

Kuhn, R.H., and Padua, D.A. (1981). "Tutorial on Parallel Processing." IEEE Computer Society Press, Catalog No. EH0182-6.

Kumar, V., and Kanal L.N. (1984). Parallel branch and bound formulations for understanding and synthesising AND/OR TREE search. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-6* (6), 768-778.

Kung, H.T. (1980). The structure of parallel algorithms. in *Advances in Comput. 19*, M. Yovits, Ed. Academic Press, New York, 65-112.

Kwan, S.C., and Ruzzo, W.L. (1984) Adaptive parallel algorithms for finding minimum spanning trees. *Proc. 1984 Int. Conf. on Parallel Process.*, 439-443.

Lakshmivarahan, S., Dhall, S.K., and Miller, L.L. (1984). Parallel sorting algorithms. in *Advances in Comput. 23*, M. Yovits, Ed. Academic Press, New York, 295-354.

Lee, G., Kruskal, C.P., and Kuck D.J. (1985). An empirical study of automatic restructuring of nonnumerical programs for parallel processors. *IEEE Trans. Comput. C-34* (10), 927-933.

Leiserson, C.E. (1979). Systolic priority queues. Tech. Rep. CMU-CS-79-115, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Levitt, K.N., and Kautz, W.T. (1972). Cellular arrays for the solution of graph problems. *Commun. ACM 15* (9), 789-801.

Luby, M. (1985). A simple parallel algorithm for the maximal independent set problem. *Proc. of the 17th Annu. ACM Symp. Theory Comput.*, 1-10.

Mateti, P., and Deo, N. (1981). Parallel algorithms for the single source shortest path problems. Tech. Rep. CS-81-078, Computer Science Dept., Washington State Univ., Pullman, Wash.

Mead, C.A., and Conway, L.A. (1980). "Introduction to VLSI Systems." Addison-Wesley, Reading, Massachusetts.

Moitra, A. (1986). Automatic construction of CSP programs from sequential non-deterministic programs. *Sci. Comput. Program. 5*, 277-307.

Moitra, A., and Iyengar, S.S. (1986). Derivation of a parallel algorithm for balancing binary trees. *IEEE Trans. Software Eng. SE-12* (3), 442-449.

Mulmuley, K., Vazirani, U.V., and Vazirani, V.V. (1986). Matching is as easy as matrix inversion. To appear in *Combinatorics*.

Munro, I., and Patterson, M. (1973). Optimal algorithms for polynomial evaluations. *J. of Comp. and Sys. Sci. 7*, 189-198.

Nath, D., and Maheshwari, S.N. (1982). Parallel algorithms for the connected components and minimal spanning tree problems. *Inf. Process. Lett. 14* (1), 7-11.

Nicolau, A. (1985). Uniform parallelism exploitation in ordinary programs. *Proc. of the 1985 IEEE Int. Conf. on Parallel Process.*, 614-618.

Nilsson, N.J. (1980). "Principles of Artificial Intelligence." Tioga Publishing Co., Palo Alto, CA.

Ottaman, T.A., Rosenberg, A.I., and Stockmeyer L.J. (1982). A dictionary machine for VLSI. *IEEE Trans. Comput. C-31* (9), 892-897.

Paige, R.C., and Kruskal, C.P. (1985). Parallel algorithms for shortest path problems. *Proc. of the 1985 IEEE Int. Conf. on Parallel Process.*, 14-20.

Pan, V. (1985). Complexity of computing polynomial zeros. Computer Science Dept., Tech. Rep. 85-27, State University of New York at Albany.

Pan, V., and Galil, Z. (1985). Improving the efficiency of parallel algorithms for the evaluation of the determinant and of the inverse of a matrix. Computer Science Dept., Tech. Rep. 85-5, State University of New York at Albany.

Pearl, J. (1982) The solution for the branching factor of alpha-beta pruning algorithm and its optimality. *Commun. ACM 25* (8), 559-563.

Preparata, F.P. (1973). New parallel-sorting schemes. *IEEE Trans. Comput C-27* (7), 669-673.

Prim, R.C. (1957). Shortest connection networks and some generalizations. *Bell Syst. Tech. J. 36*, 1389-1401.

Quinn, M.J., and Deo, N. (1984). Parallel graph algorithms. *ACM Comput. Surv. 16* (3), 319-348.

Rabin, M.O., and Vazirani, V.V. (1984). Maximum matchings in general graphs through randomization. Tech. Rep. 15-84, Center for Research in Computing Technology, Harvard University, Cambridge, Mass.

Reghbati (Arjomandi), E., and Corneil, D.G. (1978). Parallel computations in graph theory. *SIAM J. Comput. 2* (2), 230-237.

Reif, J.H. (1985). Depth-first search is inherently sequential. *Inf. Process. Lett. 20* (5), 229-234.

Sameh, A.H. (1977). Numerical parallel algorithms - a survey. in *Tutorial on Parallel Processing*, R.H. Kuhn and D.A. Padua, Eds. IEEE Computer Society Press, pp. 412-427.

Savage, C. (1977). Parallel algorithms for graph theoretic problems. Ph.D. dissertation, Mathematics Dept., University of Illinois, Urbana, Ill.

Savage, C., and Ja'Ja', J. (1981). Fast, efficient parallel algorithms for some graph problems. *SIAM J. Comput. 10* (4), 682-691.

Shiloach, Y., and Vishkin, U. (1982). An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms 3* (1), 57-67.

Shrira, L., Francez, N., and Rodeh, M. (1983). Distributed k-selection : from a sequential to a distributed algorithm. *Proc. of the Second Annu. ACM Symp. on Principles of Distributed Computing.* ACM, New York, 143-153.

Slagle, J.R., and Dixon, J.D. (1969). Experiments with Soe program that search game trees. *J. Assoc. Comput. Mach. 16* (2), 189-207.

Sollin, M. (1977). An algorithm attributed to Sollin. in *Introduction to the Design and Analysis of Algorithms*, S.E. Goodman, and S.T. Hedetniemi, Eds. McGraw-Hill, New York, sect. 5.5.

Somani, A.K., and Agarwal, V.K. (1984). An efficient VLSI dictionary machine. *Proc. of the 11th Annu. ACM Int. Symp. Comput. Architecture*, 142-150.

Stone, H.S. (1980). Parallel computers. in *Introduction to Computer Architecture*, H.S. Stone, Ed. Science Research Associates Chicago, Il., chap. 8.

Strom, R., and Yemini, S. (1985). Synthesizing distributed and parallel programs through optimistic transformations. *Proc. of the 1985 IEEE Int. Conf. on Parallel Process.,* 632-642.

Tanaka, Y., Nozoka, Y., and Masuyama, A. (1980). Pipeline searching and sorting modules as components of data flow database computer. *Proc. of IFIP*, 427-432.

Tarjan, R.E. (1972). Depth first search and linear graph algorithms. *SIAM J. Comput. 1* (2), 146-160.

Tarjan, R.E., and Vishkin, U. (1984). Finding biconnected components and computing tree functions in logarithmic parallel time. *Proc. of the 25th Annu. Symp. Foundations of Comput. Sci.*, 12-20.

Tsin, Y.H., and Chin, F.Y. (1984). Efficient parallel algorithms for a class of graph theoretic problems. *SIAM J. Comput. 13* (3), 580-599.

Tutte, W.T. (1947). The factorization of linear graphs. *J. London Math. Soc. 22*, 107-111.

Valiant, L.G. (1982). Parallel computation. *Proc. of 7th IBM Symp. on Math. Foundations of Comput. Sci.*

Valiant, L.G., and Skyum, S. (1981). Fast parallel computation of polynomials using few processors. Lecture Notes in Computer Science, Edited by J. Gruska and M. Chytil, 118, Springer-Verlag, 132-139.

Vishkin, U. (1983). Implementation of simultaneous memory address access in models that forbid it. *J. Algorithms 4* (1), 45-50.

Vishkin, U. (1985). Optimal parallel pattern matching in strings. Lecture Notes in Computer Science, Edited by W. Brauer, 194, Springer-Verlag, 497-508.

Vitter, J.S., and Simons, R.S. (1986). New classes for parallel complexity : a study of unification and other complete problems for P. *IEEE Trans. Comput. C-35* (5), 403-418.

Wyllie, J.C. (1979). The complexity of parallel computations. Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, NY.

Yao, C.-C. (1975). An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees. *Inf. Process. Lett. 4* (1), 21-23.

Yoo, Y.B. (1983). Parallel processing for some network optimization problems. Ph.D. dissertation, Computer Science Dept., Washington State Univ., Pullman, Wash.

Acknowledgements

We thank our graduate students S.V.N. Rao, Chandrasekhran, and Girish Kumthekar for their contributions in this paper. Comments by Professors Kraft and Bastani helped in revising the paper.