Parallel Graph Algorithms

MICHAEL J. QUINN AND NARSINGH DEO

Computer Science Department, Washington State University, Pullman, Washington 99164-1210

Algorithms and data structures developed to solve graph problems on parallel computers are surveyed. The problems discussed relate to searching graphs and finding connected components, maximal cliques, maximum cardinality matchings, minimum spanning trees, shortest paths, and traveling salesman tours. The algorithms are based on a number of models of parallel computation, including systolic arrays, associative processors, array processors, and multiple CPU computers. The most popular model is a direct extension of the standard RAM model of sequential computation. It may not, however, be the best basis for the study of parallel algorithms. More emphasis has been focused recently on communications issues in the analysis of the complexity of parallel algorithms; thus parallel models are coming to be more complementary to implementable architectures. Most algorithms use relatively simple data structures, such as the adjacency matrix and adjacency lists, although a few algorithms using linked lists, heaps, and trees are also discussed.

Categories and Subject Descriptors: B.7.1 **[Integrated Circuits]**: Types and Design Styles—*algorithms implemented in hardware, VLSI*; C.1.2 **[Processor Architectures]**: Multiple Data Stream Architectures (Multiprocessors)—*parallel processors*; E.1 **[Data]**: Data Structures—*arrays, graphs, lists, trees*; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—*computations on discrete structures*; G.2.2 **[Discrete Mathematics]**: Graph Theory—*graph algorithms*

General Terms: Algorithms

INTRODUCTION

Graph theory is widely applied to problems in science and engineering. Practical graph problems often require large amounts of computer time. As very large scale integration (VLSI) technology has advanced, parallel computers—computers consisting of a number of processing elements dedicated to solving a single problem at a time—have become increasingly feasible, spurring a flurry of research to find efficient parallel graph algorithms. This paper is a survey of that research.

Our purpose is to introduce the reader to some of the techniques used to develop parallel graph algorithms. In some cases an existing sequential algorithm can be transformed into an efficient parallel algorithm; in other cases the parallel algorithm must be created "from scratch." The reader is expected to have a basic knowledge of algorithm analysis, a familiarity with various parallel computer architectures, and an understanding of the elementary terms of graph theory. Weide [1977] presents a tutorial in analysis of algorithms. Topics in complexity theory relevant to parallel com-

M. J. Quinn's present address is Department of Computer Science, University of New Hampshire, Durham, NH 03824.

CONTENTS

putations are discussed by Garey and Johnson [1979] and Johnson [1983]. Introductions to parallel computers include Haynes et al. [1982], Hockney and Jesshope [1981], Mead and Conway [1980], and Stone [1980]. Harary [1969] is the standard source for graph theoretic terms; other references include Deo [1974] and Reingold et al. [1977]. The graphs described here are finite and simple (without parallel edges and self-loops), and they may be directed or undirected—the distinction will be clear from the context. Since we concern ourselves only with localized systems, we do not describe distributed graph algorithms used to solve network problems (e.g., Chandy and Misra [1982], Gallager et al. [1983], and Misra and Chandy [1982]).

There is general agreement that the evaluation of the complexity of an algorithm executed on a VLSI chip must take the physical layout of the chip into consideration [Bilardi et al. 1981; Chazelle and Monier 1981; Dymond and Cook 1980; Kedem and Zorat 1981; Leighton 1981; Leiserson 1980, 1983; Leiserson and Saxe 1981; Lipton and Valdes 1981; J. Savage 1981; Thompson 1979, 1980; Vuillemin 1983]. Our survey does not address VLSI time–area complexity issues.

An indication of the diversity within the field of parallel algorithms is the lack of a widely accepted definition of "correctness." In fact, there is disagreement as to the need for formal techniques of proving the correctness of parallel algorithms. This survey steers clear of the debate; the interested reader is referred to Berg et al. [1982], Greif [1977], Jones and Schwarz [1980], Keller [1976], Lamport [1977, 1979], Owicki and Gries [1976], and Owicki and Lamport [1982].

We introduce the terminology used in the paper in Section 1. In Section 2 we examine some models of parallel computation. In Sections 3 and 4 we survey existing parallel graph theoretic algorithms for unweighted and weighted graphs. We pay particular attention to those data structures and those aspects of algorithms that may be useful in developing new parallel algorithms.

1. TERMINOLOGY

Throughout this paper n refers to the number of vertices (nodes), and m to the number of edges, in a graph. The *average vertex degree* of an undirected graph is $2m/n$.

The letter p will be used to denote the number of processors executing an algorithm. When referring to algorithms running on an MIMD (multiple instruction stream, multiple data stream) model of computation (see Section 2.4), we often refer to logical processes rather than physical processors. In these cases assume that each process is running on its own processor.

When we refer to the *worst-case time complexity* (or simply *time* or *complexity*) of a parallel graph algorithm, we are referring to a function $f(n)$ that is the maximum, over all inputs of size n, of the "time" elapsed from when the first processor begins execution of the algorithm until the last processor terminates algorithm execution. For example, a sequential algorithm

to determine the sum of k values has complexity $O(k)$, since it requires $k - 1$ additions. If, however, additions are allowed to be done in parallel, and $k/\log k$ processors are available, the sum can be determined in less than $2\lceil \log k \rceil$ steps by computing partial sums in a treelike fashion [C. Savage 1977]. Thus parallel addition has complexity $O(\log k)$ with $k/\log k$ processors. Some authors use the term *depth* to refer to the complexity of a parallel algorithm [Shiloach and Vishkin 1982a]. The *cost* of a parallel algorithm is defined as its complexity times p.

The parallel computation thesis states that the notion of time on an unbounded parallel model is (polynomially) equivalent to the notion of space on a sequential model of computation [Chandra and Stockmeyer 1976; Goldschlager 1978, 1982]. Thus an open question in the theory of computational complexity—whether all problems solvable in time $T(n)$ on a sequential model are also solvable in space $O(\log^k T(n))$, for some k independent of n—is especially relevant to designers of parallel algorithms. In particular, it is not known whether all members of P, the problems solvable in polynomial time, are solvable in $O(\log^k n)$ space. A problem L in P is *log space complete for* P if every other problem in P is reducible in log space to L. Cook [1974] has conjectured that problems that are log space complete for P require polynomial space. If this conjecture is true, then it follows by the parallel computation thesis that problems that are log space complete for P cannot be solved quickly (i.e., in $O(\log^k n)$ time) on a "practical" parallel computer—one whose interconnection pattern is polynomially computable. A graph theoretic problem that has been shown to be log space complete for P is the maximum flow problem [Goldschlager et al. 1982]. Other problems that are at least as difficult in terms of space complexity are given by Dobkin et al. [1979], Galil [1976], Goldschlager [1977a], Jones and Laaser [1976], Kozen [1977], and Ladner [1975].

Two important measures of the quality of parallel algorithms implemented on actual parallel computers are *speedup* and *efficiency*. The *speedup* of a parallel algorithm solving a problem Π is the ratio between the execution time of the fastest sequential algorithm solving Π and the execution time of the parallel algorithm. For example, if the fastest known sequential algorithm runs in 10 seconds, while a parallel algorithm to solve the same problem runs in 2 seconds when 8 processors are used, then we say that the parallel algorithm exhibits a "speedup of 5 with 8 processors." The *efficiency* of a parallel algorithm is the ratio between the speedup and the number of processors used. Thus a parallel algorithm that exhibits a speedup of 4 with 8 processors has an efficiency of 0.5 with 8 processors.

All logarithms in this survey are to base 2.

2. MODELS OF PARALLEL COMPUTATION

Unlike sequential algorithms, for which standard models of computation exist (such as RAM and RASP [Aho et al. 1974]), there are many contending models of computation on which to base parallel algorithms. Since commercial parallel computers have been in existence only a short time, it is unclear which models, if any, will become dominant. The models that have been used to develop parallel algorithms for graph theory problems may be divided into four major categories: (1) systolic arrays, (2) associative processors, (3) processor arrays, and (4) multiple central processing unit (CPU) computers.

2.1 Systolic Arrays

A systolic array is a collection of synchronized, special-purpose, rudimentary processors with a fixed interconnection network [Kung 1980, 1982; Kung and Leiserson 1980]. The function of the processors and the type of interconnection scheme depend upon the problem being solved. The number of processors is usually assumed to be potentially unbounded. This assumption is not as farfetched as it may seem, since the simplicity of the processors and the uniformity of the processor interconnections allow large systolic arrays to be implemented on a single chip by using VLSI technology [Foster and Kung 1980]. Sys-

tolic arrays are designed to be high-speed special-purpose functional units attached to a general-purpose computer. Bentley and Kung's tree-structured searching machine is a noteworthy example of a systolic array [Bentley and Kung 1979].

2.2 Associative Processors

An associative processor is a special-purpose processor built around an associative memory that allows the simultaneous searching of the whole memory for specified contents. Sanders Associates' OMEN series of computers [Higbie 1972] and the Goodyear Aerospace STARAN [Batcher 1979] are examples of associative processors. Yau and Fung [1977] have surveyed associative processor architectures.

2.3 Processor Arrays

Easily the most popular model of parallel computation is the processor array, a set of identical synchronized processing elements capable of simultaneously performing the same operation on different data. Although the processing elements execute in parallel, units may be programmed to ignore any particular instruction. This ability to mask out processing elements allows synchronization to be maintained through the various paths of control structures, such as clauses of an if–then–else statement.

Flynn [1966] has categorized parallel computer architectures by the parallelism of their instruction and data streams. Computers executing a single instruction sequence, with each instruction potentially affecting many data items simultaneously, fall into the SIMD (single instruction stream, multiple data stream) category. As in Harrison and Wilson [1983], we shall use the terms SIMD computer and processor array synonymously. SIMD computers should not be confused with *algorithmic* array processors, such as the Datawest 400 and Floating Point Systems Model AP120B, which achieve parallelism through pipelining scalar instructions.

The SIMD models used in computational graph theory can vary in two respects: The number of processors may be fixed or unbounded, and processors may communicate with each other via shared memory (SM), a mesh-connected network (MC), a perfect shuffle network (PS), a cube-connected network (CC), or a cube-connected cycles network (CCC).

The mechanism through which processors in an SIMD model access data has a significant impact on the time complexity of a parallel algorithm. Most reported algorithms assume a shared memory model that allows the p processors to access simultaneously any p locations in the entire memory space in constant time. However, some investigators have placed certain restrictions on memory accesses. In the most restrictive SIMD-SM (shared memory) model, no two processors may access the same location during the same instruction. The SIMD-SM-R model allows any number of processors to address the same location for reading (accessing), but not for writing. The SIMD-SM-RW model allows multiple processors to address the same location during both read and write instructions. In the case of simultaneous writing, different assumptions are made about which processor's value actually gets written into the memory location.

Eckstein [1979b] has examined the relationship between the SIMD-SM, SIMD-SM-R, and SIMD-SM-RW models. She shows how binary trees can be used to resolve read and write conflicts. To resolve read conflicts, only one processor retrieves the value from the memory location and broadcasts it to the other processors by using a tree. Write conflicts are resolved by using a tree as a tournament mechanism, guaranteeing that only one processor writes its value. An algorithm having time complexity t and space complexity s on an SIMD-SM-RW or an SIMD-SM-R model can be transformed into an algorithm for the SIMD-SM model with time complexity $O(t \log p)$ and space complexity $O(sp)$. Thus the use of binary trees to resolve read and write conflicts is costly in terms of space—they multiply the space complexity of any algorithm by $O(p)$.

Vishkin [1983] has proposed an alternate method that, while it uses much less space than Eckstein's method, requires an increase in time complexity. In his scheme,

algorithms translated from SIMD-SM-RW and SIMD-SM-R models to the SIMD-SM model have a time complexity of $O(t \log^2 p)$ and a space complexity of $O(s+p)$. Vishkin's method relies on an efficient sorting of the addresses to bring together requests for the same address [Batcher 1968]. Once demands for the same memory location have been brought together, resolution of the conflicts is straightforward.

Despite the popularity of the various SIMD-SM models, the fact remains that no actual processor arrays have been built that are based on the shared memory model, because it is not feasible to allow p processors to access any p memory addresses simultaneously. A more realistic assumption is that each processor has its own private memory, and processors can pass data only via a limited interconnection network [Dekel et al. 1981]. In an SIMD-MC (mesh-connected) model, the processors are arranged into a q-dimensional lattice. Communication is allowed only between neighboring processors. The SIMD-MC model requires $2q$ connections per processor. The ILLIAC IV [Falk 1976; Feierbach and Stevenson 1979], Burroughs PEPE [Crane et al. 1972], Goodyear Aerospace MPP [Batcher 1980; Fung 1977], and ICL DAP [Flanders et al. 1977; Reddaway 1979] all fall into the SIMD-MC category with $q = 2$. The SIMD-MC model suffers from the disadvantage that data-routing requirements often prevent the development of fast parallel algorithms. Two examples illustrate this point: At least $0.35n$ routing steps are needed to compute the product of two $n \times n$ matrices on the SIMD-MC model [Gentleman 1978], and sorting n elements on the SIMD-MC model requires time $\Omega(\sqrt{n})$ [Thompson and Kung 1976]. Communications issues on the SIMD-MC model are also addressed by Lint and Agarwala [1981] and Nassimi and Sahni [1979, 1980a].

Assume that there are $p = 2^k$ processors to be connected in an SIMD model and that each processor is assigned a unique identifying number in the range 0 through $p - 1$. The SIMD-CC (cube-connected) model connects those processors whose identifying numbers differ by only a single bit in their binary representations. This is called a *cube-connected network of degree k*. The SIMD-CC model requires $\log p$ connections per processor. For large p, this value is unsuitably high. The following two models are able to emulate the SIMD-CC model, while requiring only three connections per processor.

Processors in the SIMD-PS model are connected using Stone's perfect shuffle method [Stone 1971]. Siegel [1979] has shown that a composition of k shuffle–exchange networks (called an *omega network*) is equivalent to a cube-connected network with degree k. The same effect can be achieved by building only one stage of the network and cycling through it k times [Lawrie 1975].

Preparata and Vuillemin [1981] have proposed another interconnection scheme called the cube-connected cycles (SIMD-CCC). In this network no processor has more than three connections, yet it, too, can emulate the SIMD-CC and SIMD-PS models. The SIMD-CCC model has the additional advantage that its VLSI layout is more compact and regular.

A parallel model is *reasonable* if the number of processors each processor can communicate with is bounded by a constant [Goldschlager 1982]. A natural question is whether each of these reasonable processor interconnection networks (MC, PS, and CCC) is superior for some class of graph problems, or whether one of these is the best overall model. Galil and Paul [1981, 1983] show that a universal parallel model can simulate every reasonable parallel model "with only a small loss of time and with essentially the same number of processors." The heart of their universal computer is a sorting network that is used as a "post office" for sending and requesting information. Galil and Paul have shown that, since CCCs are used as the sorting networks, the CCC is an efficient general-purpose network.

2.4 Multiple CPU Computers

Multiple CPU computers consist of a number of fully programmable processors, each

capable of executing its own program. In Flynn's taxonomy, such machines are referred to as MIMD (multiple instruction stream, multiple data stream) computers [Flynn 1966]. Two important attributes in which MIMD models may differ from each other are that the processors may be synchronous or asynchronous and the number of processors may be fixed or unbounded. A third significant attribute is the processor intercommunication pattern.

The simplest processor intercommunication pattern assumes that all the processors work through a central switching mechanism to reach a global memory. Carnegie-Mellon University's C.mmp [Fuller and Oleinick 1976; Mashburn 1979; Oleinick 1978; Wulf and Harbison 1978] and Denelcor's HEP [Denelcor 1981; Smith 1978] fit into this category, hereafter denoted MIMD-TC (for tightly coupled). It is impractical to build large systems of this type for the cost of the switch would soon become the dominant factor [Stone 1980]. Thus, architectures based on the MIMD-TC model have a limited number of processors: for example, C.mmp has 16 processors, and the HEP is limited to eight (pipelined) processors.

Another MIMD model consists of a larger number of processor–memory pairs, interconnected so as to form a binary tree [Browning 1980a, 1980b]. This model, referred to as MIMD-BT, seems suitable for implementation in VLSI. A single chip is capable of containing a large subtree of processors. Work is being done at the California Institute of Technology to implement a parallel computer based on the MIMD-BT model with at least 1023 processors.

A third method connects processors in a hierarchical fashion. Cm* at Carnegie-Mellon University is the best-known example of this model [Stone 1980; Swan et al. 1977]. This model will be denoted MIMD-LC (for loosely coupled) [Stone 1980]. The Cm* system is based on a *computer module* consisting of a processor, local memory, local I/O devices, and a local switch connecting the module to the rest of the system. All processor references to I/O or memory go through the local switch. If an access is local, the reference is directed to the local memory or I/O device. Nonlocal references are directed to the Map bus, the connection to the rest of the processors. Several computer modules are combined to form a cluster sharing a single Map bus. Clustering allows processors to share data. Competition for the Map bus limits the number of processors that can be connected into a single cluster. Clusters are connected via intercluster buses. If the Kmap (Map controller) detects a memory reference to another cluster, it redirects the reference via an intercluster bus to another Kmap, which places the reference on its own Map bus. Since the MIMD-LC model does not have a centralized switching mechanism, a large number of processors may be connected together. Currently Cm* has 50 processors [Haynes et al. 1982].

3. PARALLEL ALGORITHMS FOR UNWEIGHTED GRAPHS

3.1 Searching a Graph

Searching is of fundamental importance, since it forms the basis of a great many graph algorithms. Various forms of search are used to determine connectivity, to find a set of fundamental cycles, and to solve shortest-path problems, to name a few examples.

Given an SIMD-SM-R model of computation in which there are a fixed number of processing elements p, Reghbati (Arjomandi) and Corneil [1978] have determined the number of operations required for three parallel search techniques. Depth-first search seems to be an inherently sequential process because searching always occurs along a single edge from a single vertex, restricting opportunities for parallelism. (Eckstein and Alton [1977a, 1977b] have worked on this problem with limited success.) Reghbati and Corneil thus consider a parallel variant of depth-first search, called *p-depth search* (defined later), as well as parallel breadth–depth search and parallel breadth-first search.

If a systolic array or an unbounded SIMD model is being used, then the adjacency matrix is a natural representation of a graph to be searched, because a one-to-one

correspondence can be established between the processors and the elements of the adjacency matrix. Otherwise, the process of searching through the elements of an adjacency matrix to find edges would consume too much processor time. For this reason Reghbati and Corneil use adjacency lists to represent the graph.

How quickly can a graph be searched? Initially a master list of vertices still to be searched contains a single vertex. In any search procedure one vertex is chosen from the list of vertices to be searched. Each processor examines one or more edges emanating from that vertex. If the edge leads to a previously undiscovered vertex, it is added to that processor's partial list containing vertices to be added to the master list. At certain intervals the partial lists formed by the processors are linked together and combined with the master list. Let us assume that the only operations that consume time are the vertex-selection process and the list linking and combining process. Assume that it takes one of these *active operations* to select a vertex. For the sequential algorithm, only one active operation is required for the lone processor to add a new vertex to the master list. Let d_i denote the degree of vertex i. It is clear, then, that an upper bound for a sequential algorithm to search a graph is

$$T_1 = \sum_{i=1}^{n} (d_i + 1) = 2m + n,$$

since vertex i can be added to a partial list only once, and d_i is the maximum number of times that vertex i can be chosen as the vertex from which searching is to be done.

In p-depth search, p edges incident upon a selected vertex are simultaneously searched. (In other words, processors are assigned to edges, one processor per edge.) One of the most recently searched vertices is then chosen as the point to continue the search. This procedure ends when the master list of vertices having unexplored edges is empty. If $p = 1$, the result is a depth-first search. The new vertices found at each state of the algorithm can be added to the master list of vertices in $\lceil \log p \rceil + 1$ active operations, by linking lists in a treelike manner. The number of times that searching begins at vertex i is $\lceil (d_i + 1)/p \rceil$. Therefore the number of active operations required for parallel p-depth search is

$$T_p^1 = \sum_{i=1}^{n} (\lceil (d_i + 1)/p \rceil)(\lceil \log p \rceil + 1);$$

thus

$$T_p^1 \leq T_1(\lceil \log p \rceil + 1)/p + n(\lceil \log p \rceil + 1).$$

The term $n(\lceil \log p \rceil + 1)$ is a bound on the time spent in combining lists. It can be shown that for $p \geq 2$, p-depth search requires fewer active operations than a sequential search for graphs in which the average vertex degree is at least $\lceil \log p \rceil + 17$ [Reghbati and Corneil 1978].

A breadth–depth search proceeds by examining all the edges adjacent to a vertex before selecting one of the most recently reached vertices and continuing the search from the vertex. In parallel breadth–depth search, each processor keeps track of the new vertices it has discovered. Once all the edges from a vertex have been examined, these partial lists are linked and added to the master list. Since this parallel algorithm requires partial lists to be linked and combined with the master list less often than p-depth search, one might suspect that this algorithm requires fewer active operations. This suspicion is correct: Parallel breadth–depth search has an upper bound of

$$T_p^2 = \sum_{i}^{n} (\lceil d_i/p \rceil + 1 + \lceil \log p \rceil + 1),$$

where $\lceil \log p \rceil + 1$ is the time spent linking the vertices found during exploration from vertex i, and $d_i/p + 1$ is the number of examination steps beginning from vertex i. This expression simplifies to

$$T_p^2 \leq T_1/p + n(\lceil \log p \rceil + 3).$$

Thus, if $p \geq 2$, parallel breadth–depth search requires fewer active operations than sequential breadth–depth search for graphs whose average vertex degree is at least $\lceil \log p \rceil + 7$.

Parallel breadth-first search requires even fewer link-and-combine steps, because processors examine all vertices at level i of the search tree before moving on to level

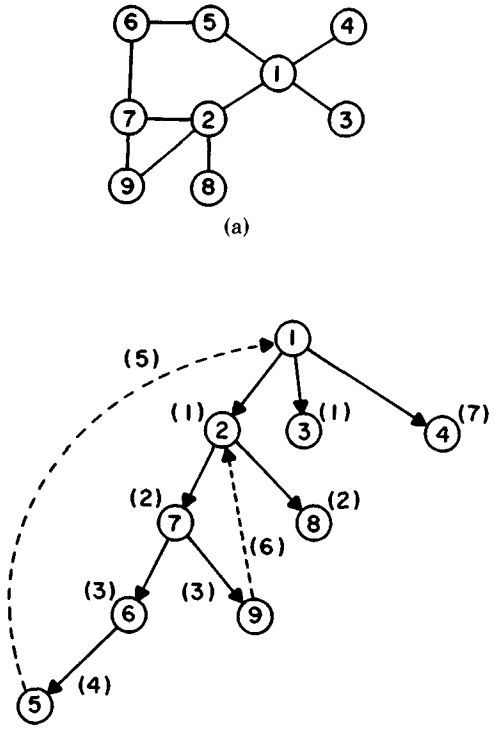


Figure 1. Illustration of p-depth search. (a) Graph. (b) Two-processor p-depth search (numbers in parentheses indicate order of traversal)

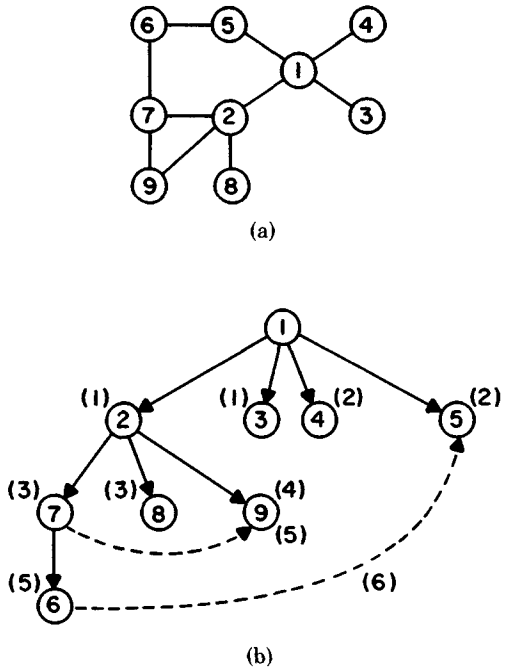


Figure 2. Illustration of parallel breadth–depth search. (a) Graph. (b) Two-processor breadth–depth search (numbers in parentheses indicate order of traversal).

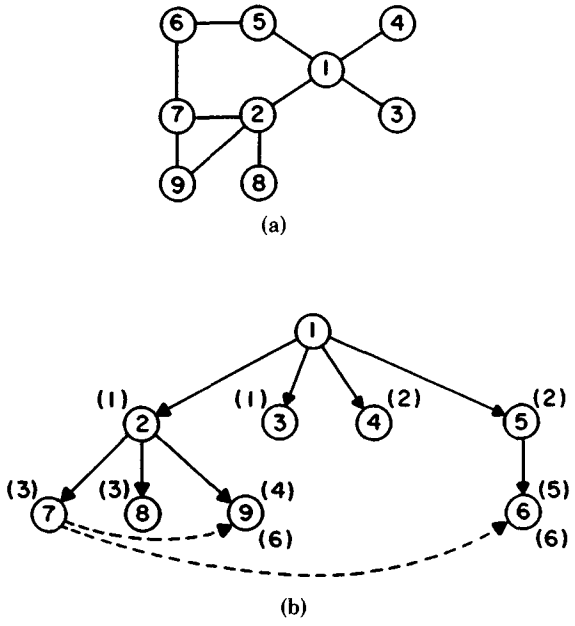


Figure 3. Illustration of parallel breadth-first search. (a) Graph. (b) Two-processor breadth-first search (numbers in parentheses indicate order of traversal).

Table 1. Parallel Connected Components Algorithms*

Reference	Method	Model	Complexity	Processors
Reghbati and Corneil [1975]	Breadth-first search	SIMD-SM-R	$T/p + L \log p + 2n$	p
Reghbati and Corneil [1975]	Transitive closure	SIMD-SM-R	$O(\log^2 n)$	n^3
Chandra [1976]	Transitive closure	SIMD-SM-R	$O(\log^2 n)$	$n^{\log 7}/\log n$
Hirschberg [1976]	Vertex collapse	SIMD-SM-R	$O(\log^2 n)$	n^2
Hirschberg et al., [1979]	Vertex collapse	SIMD-SM-R	$O(\log^2 n)$	$n\lceil n/\log n \rceil$
Wyllie [1979]	Vertex collapse	MIMD-TC-R	$O(\log^2 n)$	$n + 2m$
Nassimi and Sahni [1980b]	Vertex collapse	SIMD-MC	$O(n^{1/2} \log n)$	n
Chin et al. [1981]	Vertex collapse	SIMD-SM-R	$O(\log^2 n)$	$n\lceil n/\log^2 n \rceil$
Savage and Ja'Ja' [1981]	Vertex collapse	SIMD-SM-R	$O(\log n \log d)$	$n^3/\log n$
	Vertex collapse	SIMD-SM-R	$O(\log^2 n)$	$m + n \log n$
Savage [1981]	Vertex collapse	Systolic array	$O(n + m)$	$n + 1$
Nath and Maheshwari [1982]	Vertex collapse	SIMD-SM	$O(\log^2 n)$	$m + n \log n$
Shiloach and Vishkin [1982a]	Vertex collapse	SIMD-SM-RW	$O(\log n)$	$n + 2m$
Kučera [1982]	Transitive closure	SIMD-SM-RW	$O(\log n)$	n^4
Reif and Spirakis [1982]	Vertex collapse	SIMD-SM-RW (probabilistic)	$O(\log \log n)$ (expected)	$O(n^2)$ (expected)
Hambrusch [1982]	Vertex collapse	Systolic array	$O(n^{3/2})$	n

* Key: d, diameter of graph; L, distance of node farthest from start node; T, complexity of optimal sequential breadth-first search algorithm.

$i + 1$. There is thus only one link-and-combine step for each level of the search tree, and parallel breadth-first search has bound

$$T_p^3 = \sum_{i=1}^{n} (\lceil d_i/p \rceil + 1) + L\lceil \log p \rceil,$$

where L is the distance of the furthest vertex from the start vertex. Hence

$$T_p^3 \leq T_1/p + L\lceil \log p \rceil + 2n,$$

and for $p \geq 2$, parallel breadth-first search is superior to sequential breadth-first search for graphs with average vertex degree of at least $\lceil \log p \rceil + 5$. Figures 1–3 contrast parallel p-depth, breadth-depth, and breadth-first searches of the same graph.

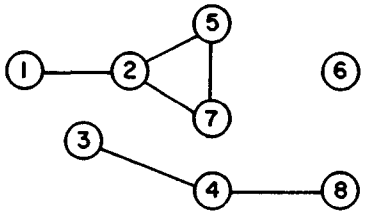
Alton and Eckstein [1979] have reported a breadth-first search algorithm for searching sparse graphs using the SIMD-SM-R model. They allow processors to examine any available edge emanating from any vertex at the current level of the search tree, whereas the algorithm of Reghbati and Corneil requires that all edges from one vertex be exhausted before the edges from another vertex are considered. Alton and Eckstein's technique does not drastically change the complexity of the algorithm.

3.2 Connected Components of Undirected Graphs

A good deal of work has been reported on parallel algorithms to find the connected components of undirected graphs. These algorithms have been based upon breadth-first search, transitive closure, or vertex collapse. The work done to date on this problem is summarized in Table 1.

Reghbati (Arjomandi) and Corneil [1978] have shown that the connected component problem can be solved for sufficiently dense graphs by using a parallel breadth-first search algorithm. They have also suggested the use of transitive closure as a method of solving the connected component problem. Letting A denote the adjacency matrix of the original undirected graph G and B denote the transitive closure of A, B is computed by repeated ($\lceil \log n \rceil$) plus–min multiplications of A. Element $b_{i,j}$ of B is 1 if and only if there is a path (of length 0 or more) from i to j in G. A third matrix C is constructed as follows: $c_{i,j} = j$, if $b_{i,j} = 1$; otherwise $c_{i,j} = \infty$. That is, row i of matrix C contains the names of the vertices in the same component as vertex i. By determining the smallest entry in each of the n rows of C, vertices in G that are in the same component can be assigned identical com-

Graph:



A	1	2	3	4	5	6	7	8
1	1	1	0	0	0	0	0	0
2	1	1	0	0	1	0	1	0
3	0	0	1	1	0	0	0	0
4	0	0	1	1	0	0	0	1
5	0	1	0	0	1	0	1	0
6	0	0	0	0	0	1	0	0
7	0	1	0	0	1	0	1	0
8	0	0	0	1	0	0	0	1

B	1	2	3	4	5	6	7	8
1	1	1	0	0	1	0	1	0
2	1	1	0	0	1	0	1	0
3	0	0	1	1	0	0	0	1
4	0	0	1	1	0	0	0	1
5	1	1	0	0	1	0	1	0
6	0	0	0	0	0	1	0	0
7	1	1	0	0	1	0	1	0
8	0	0	1	1	0	0	0	1

C	1	2	3	4	5	6	7	8
1	1	2	∞	∞	5	∞	7	∞
2	1	2	∞	∞	5	∞	7	∞
3	∞	∞	3	4	∞	∞	∞	8
4	∞	∞	3	4	∞	∞	∞	8
5	1	2	∞	∞	5	∞	7	∞
6	∞	∞	∞	∞	∞	6	∞	∞
7	1	2	∞	∞	5	∞	7	∞
8	∞	∞	3	4	∞	∞	∞	8

Vertex	Component
1	1
2	1
3	3
4	3
5	1
6	6
7	1
8	3

Figure 4. Finding connected components via transitive closure.

ponent numbers (see Figure 4). Thus Reghbati and Corneil's connectivity algorithm has complexity $O(\log^2 n)$ using n^3 processors on an SIMD-SM-R model: Squaring matrix A $\lceil \log n \rceil$ times requires $O(\log^2 n)$ time, constructing C requires constant time, and finding the minimum entry in each row requires $O(\log n)$ time. On the other hand, if Chandra's parallel matrix multiplication algorithm is used to compute the transitive closure of B [Chandra 1976], the number of processors needed is only $\lceil n^{\log 7}/\log n \rceil$, instead of n^3.

Kučera [1982], too, has taken the transitive closure approach, using an SIMD-SM-RW model. This model of parallel computation enables n^2 processors to find the minimum of a set of n elements in constant time. Assuming that one wants to find the minimum of values $c_1, c_2, \ldots, c_n$, n^2 processors are used, labeled $P_{i,j}$, $i, j = 1, 2, \ldots, n$, and n temporary memory locations, labeled $t_1, t_2, \ldots, t_n$. The algorithm has four stages:

(1) $P_{i,1}$ sets the value of t_i to 0, for all i.
(2) If $c_i < c_j$, then $P_{i,j}$ sets the value of t_j to 1. At the end of this stage $t_i = 0$ if and only if $c_i = \min\{c_1, c_2, \ldots, c_n\}$. There may be more than one element with the

minimum value, however. Stage 3 ensures that only one processor will return a minimum value from the procedure.

(3) If $t_i = 0$ and $i < j$, then $P_{i,j}$ sets the value of t_j to 1. At this point exactly one of the temporary memory locations has the value 0.

(4) If $t_i = 0$, then $P_{i,1}$ returns c_i as the minimum.

The transitive closure of a binary matrix can be computed using minimization operations. Kučera uses the constant-time minimization procedure to construct an $O(\log n)$ transitive closure algorithm. It requires n^4 processors.

A third approach for solving the connected component problem has been presented by Hirschberg [1976; Hirschberg et al. 1979]. The primary data structure is again the adjacency matrix. Instead of computing the transitive closure, however, adjacent vertices are combined into "supervertices," which are themselves combined until each remaining supervertex represents a connected component of the graph. Like the transitive closure algorithm on the MIMD-SM-R model, this algorithm has a complexity of $O(\log^2 n)$, but requires only n^2 processors.

Each vertex is always a member of exactly one supervertex, and each supervertex is identified by its lowest-numbered member vertex, which is called the *root*. At the beginning of the algorithm every vertex is the root of its own supervertex. The parallel algorithm iterates through three stages. In the first, the lowest-numbered neighboring supervertex of each vertex is found. The second stage consists of connecting each supervertex root to the root of the lowest-numbered neighboring supervertex. In the third stage all newly connected supervertices are collapsed into larger supervertices. Since the number of supervertices is reduced by a factor of at least two in each iteration, $\lceil \log n \rceil$ iterations are sufficient to collapse each connected component into a single supervertex. The operation of this algorithm is illustrated in Figure 5.

Hirschberg's original algorithm uses n^2 processors to assign values to the matrix containing the root numbers of neighboring supervertices. Preparata and Probert have observed that $\lceil n/\log n \rceil$ processors are sufficient to assign n values and find the minimum of n elements, both in $O(\log n)$ time [Hirschberg et al. 1979], the reason being that each processor can assign values to log n elements, instead of to one element, without increasing the time complexity of the algorithm. Similarly, in the first phase of minimization, each processor can find the minimum of log n values, rather than two values, without increasing the complexity of the algorithm. Hirschberg's algorithm thus can be implemented using $n\lceil n/\log n \rceil$ processors, instead of n^2.

Chin et al. [1981, 1982] have made the algorithm even more efficient, reducing the number of processors to $n\lceil n/\log^2 n \rceil$. This is accomplished by restricting some operations to nonisolated supervertices. Every vertex is involved during each iteration of Hirschberg's algorithm. Chin and his coworkers have noted that by restricting participation in each iteration to a representative vertex of each supervertex (the root), and by removing isolated supervertices from further consideration, the algorithm requires fewer processors.

Savage and Ja'Ja' [1981] also have based their algorithms on the supervertex approach. One of their algorithms is designed to solve the problem quickly for dense graphs: Given $n^3/\log n$ processors, it has complexity $O(\log n \log d)$, where d is the diameter of the graph. The algorithm for sparse graphs uses somewhat fewer processors by storing the graph as adjacency lists. It requires $m + n \log n$ processors and has complexity $O(\log^2 n)$.

Nath and Maheshwari [1982] have modified Hirschberg's algorithm for an SIMD-SM model, in which simultaneous reads from the same memory location are forbidden. In order to avoid read conflicts, each rooted tree corresponding to a supervertex is split into chains, so that every interior vertex in the tree has exactly one child. This prevents read conflicts from occurring as children access nonroot parents. The read conflicts that remain can be eliminated by creating multiple copies of data. Their algorithm has complexity $O(\log^2 n)$

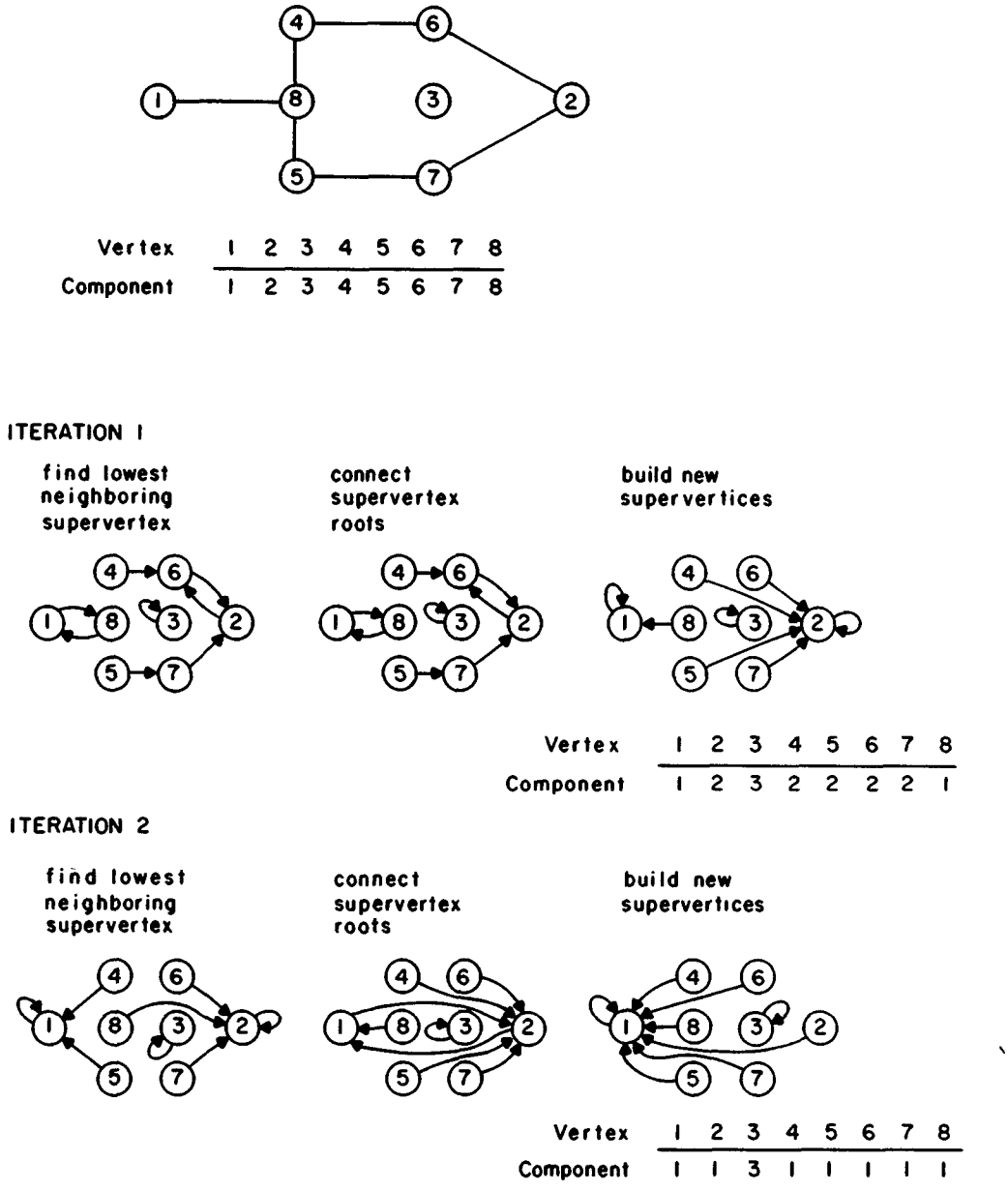


Figure 5. Hirschberg's [1976] connected components algorithm.

with n^2 processors. By using the technique described by Hirschberg et al. [1979], the number of processors needed can be reduced to $n\lceil n/\log n \rceil$.

By using a more powerful model of parallel computation (SIMD-SM-RW), Shiloach and Vishkin [1982a] have developed a connected component algorithm with complexity $O(\log n)$ using only $n + 2m$ processors. In the case of simultaneous writing into the same memory location, the processor that succeeds in writing its value is arrived at nondeterministically.

The algorithm also falls into the vertex-collapse category. One processor is assigned to each vertex and another processor to

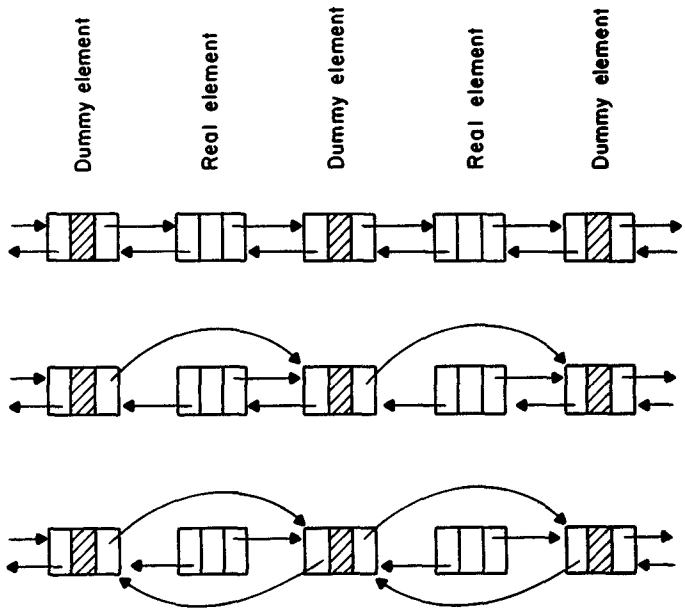


Figure 6. Parallel deletion of "adjacent" real elements.

each of the two directed arcs constituting an undirected edge, thus using $n + 2m$ processors. As in Hirschberg's algorithm, a pointer array indicates the component membership of each vertex; this array represents "a forest of rooted trees plus self-loops that occur only in the roots." The algorithm links trees at their roots and then decreases their height. It repeats this pair of operations until an iteration causes no changes in the pointer array. In its final state this pointer array indicates the component membership of each vertex.

Reif and Spirakis [1982] have presented a connected components algorithm that uses a *probabilistic* SIMD-SM-RW model; the processors are allowed to choose random bits independently, and write conflicts are resolved randomly. If i processors try to write into the same memory location at the same time, exactly one is successful, and the probability of each processor succeeding is $1/i$. Reif and Spirakis' algorithm is also of the vertex collapse type, consisting of a probabilistic phase followed by execution of Shiloach and Vishkin's algorithm.

The 2 log log n iterations of the probabilistic phase initiate the process of collapsing vertices into supervertices. Each of these iterations requires constant time. Shiloach and Vishkin's algorithm is then executed. Reif and Spirakis show that the expected parallel time of this algorithm is $O(\log \log n)$ on the probabilistic SIMD-SM-RW model; the expected number of processors used is $O(n^2)$, and the probability that the parallel time complexity exceeds $O(\log \log n)$ is $o(1/n)$.

Wyllie [1979] has used Hirschberg's approach to produce a connected component algorithm with complexity $O(\log^2 n)$ for a synchronized MIMD-TC-R model. The number of processors needed by his algorithm is $n + 2m$; it may be therefore superior for sparse graphs. Every vertex has its own circular doubly linked adjacency list, which consists of alternating dummy elements and elements representing arcs. The use of doubly linked adjacency lists allows efficient deletion of arcs, and the presence of the dummy elements allows simultaneous manipulation of arc elements, enabling the vertices to be combined into supervertices in parallel. Figure 6 illustrates how the use of dummy elements in a doubly linked list allows "adjacent" real elements to be deleted in parallel.

Savage has designed an $O(n + m)$ algorithm to solve the connected components problem on a systolic array with $n + 1$ cells [C. Savage 1981]. Her algorithm is based on the vertex collapse approach. The cells of the systolic array are arranged linearly. Cell i, $1 \leq i \leq n$, corresponds to vertex i. Each cell i has two fields: One field contains the vertex (cell) number, and the second field, Comp(i), contains the current component number of vertex i. Initially both these fields have the value i. Cell $n + 1$ serves as a turnaround mechanism. The edges of the graph are input one at a time into cell 1. Odd-numbered cells are active on odd-numbered clock pulses; even-numbered cells are active on even-numbered clock pulses. Thus one edge record enters the systolic array every two pulses. Edge records travel from cell 1 to cell $n + 1$, reverse, and travel back to cell 1, where they are discarded. When the last edge record has been discarded, Comp(i) contains the component number of vertex i, for all i, $1 \leq i \leq n$.

Right-moving edge record k contains the edge (u_k, v_k), the component number of u_k, denoted cu_k, and the component number of v_k, denoted cv_k. Left-moving edge record j contains the edge (u_j, v_j), $\min\{\text{cu}_j, \text{cv}_j\}$, denoted cmin_j, and $\max\{\text{cu}_j, \text{cv}_j\}$, denoted cmax_j. When an edge record k enters cell i, cu_k is assigned the current value of Comp(i), if $u_k = i$. Similarly, if $v_k = i$, then cv_k is assigned the current value of Comp(i). At cell $n + 1$, record k is reversed. Record element cmin_k is assigned the value $\min\{\text{cu}_k, \text{cv}_k\}$ and cmax_k is assigned the value $\max\{\text{cu}_k, \text{cv}_k\}$. When left-moving record j enters cell i, if Comp(i) = cmax_j, then Comp(i) is assigned the value cmin_j.

Component information in right-moving records may be updated by information carried by left-moving records. Given right-moving record k and left-moving record j in the same cell, if $\text{cu}_k = \text{cmax}_j$, then cu_k is assigned the value cmin_j, and if $\text{cv}_k = \text{cmax}_j$, then cv_k is assigned the value cmin_j. This step ensures that when right-moving edge k enters cell $n + 1$ and reverses, cmin_k and cmax_k will reflect the collapsing already done by edges preceding edge k.

Figure 7 illustrates Savage's [1981] connected components algorithm.

3.3 Other Connectivity-Related Algorithms

The problems of finding weakly connected components and strongly connected components in a directed graph may be reduced to transitive closure operations [Reghbati and Corneil 1978]. Eckstein [1979a], Savage and Ja'Ja' [1981], and Tsin and Chin [1982] have developed parallel algorithms to find the biconnected components of an undirected graph. In addition, Savage and Ja'Ja' [1981] and Tsin and Chin [1982] have reported algorithms with a complexity of $O(\log^2 n)$ for a class of related problems, including finding lowest common ancestors and articulation points. In all cases the algorithms of Tsin and Chin have a lower cost than the others. An algorithm to test for k-connectivity has been reported by Goldschlager [1977b]. All of these algorithms use the SIMD-SM-R model of parallel computation.

Ja'Ja' and Simon [1982] have devised fast algorithms for finding triconnected components and testing planarity using the SIMD-SM-R model. They define the triconnected components of the graph in such a way that the graph is planar if and only if its triconnected components are planar. Generalizing on Savage and Ja'Ja's [1981] biconnected components algorithm, Ja'Ja' and Simon find the triconnected components of a graph in $O(\log^2 n)$ time with $O(n^4)$ processors. The first planarity testing algorithm, given a triconnected graph, either builds a plane mesh or reports that the graph is not planar. This algorithm has a complexity of $O(\log^2 n)$, given $O(n^4)$ processors that do not need to add or multiply. The second algorithm, which yields a barycentric representation if the graph is planar, has a complexity of $O(\log^2 n)$ given $O(n^{3.29}/\log^2 n)$ processors.

Further references to connectivity-related algorithms in the literature include Attalah [1983], Attalah and Kosaraju [1982], Guibas et al. [1979], Hambrusch [1982, 1983], Kosaraju [1979], Levitt and Kautz [1972], Levialdi [1972], Lipton and Valdes [1981], Nassimi and Sahni [1980b], Reif [1982], C. Savage [1977], and Van Scoy [1976]. With the exception of Nassimi and Sahni [1980b], Reif [1982], and C. Sav-

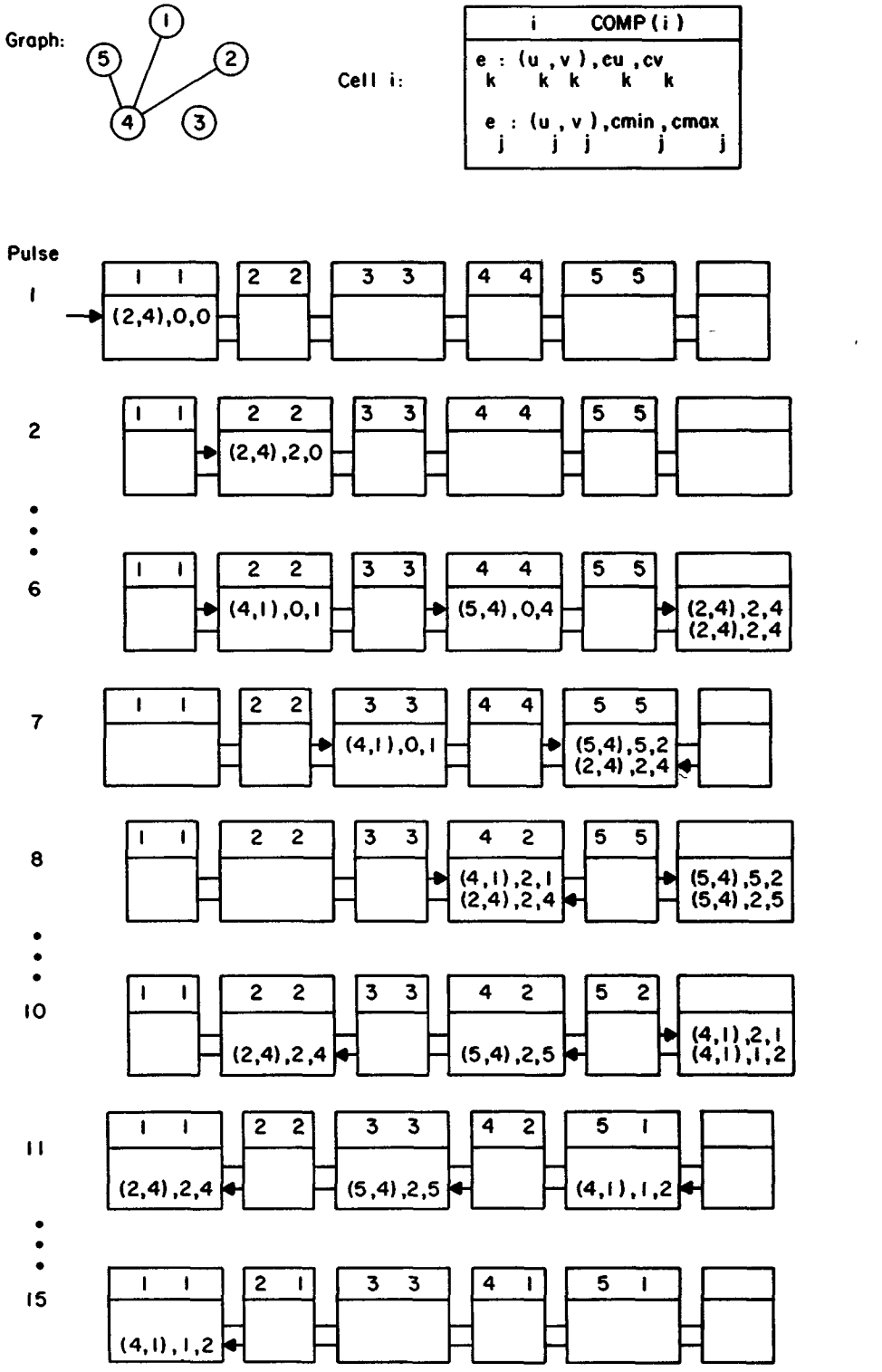


Figure 7. Savage's [1981] connected component algorithm.

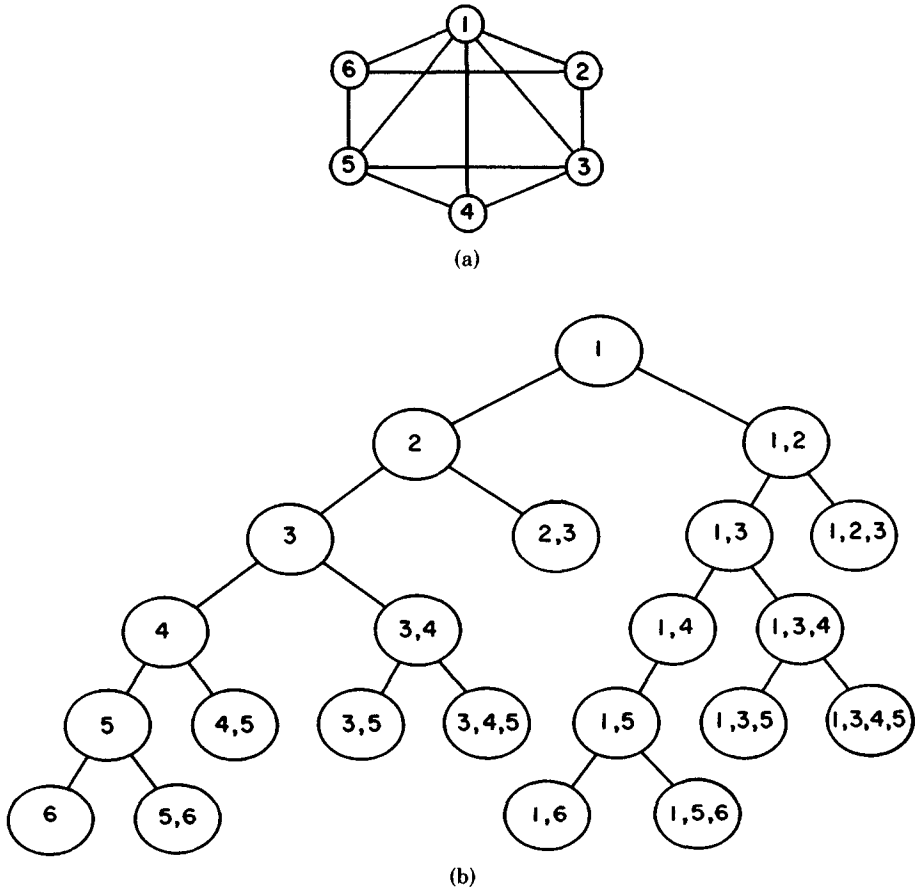


Figure 8. Example of Browning's [1980b] algorithm to find the maximum clique. (a) Graph. (b) Activated processors.

age [1977], all these references are to algorithms for systolic arrays.

3.4 Maximum Clique

Browning [1980b] has designed an algorithm to find the maximum clique in an undirected graph using the MIMD-BT model. (This problem is NP-hard [Garey and Johnson 1979].) Each level i in the tree of processors represents the addition of vertex i into the pool of considered vertices (assuming that the root is at level 1); thus the tree has depth $n - 1$, and the algorithm requires a tree of $2^n - 1$ processors, one for each potential clique.

Each processor stores an adjacency matrix of the graph, an integer containing the size of the clique the processor represents, and an array *clique* containing the vertices in the clique. Initially the root processor is the only active processor. When a processor at level i is activated, it considers whether the subgraph formed by adding vertex i to *clique* is also a clique; if it is, the processor activates its right child. If a clique is formed by replacing the last vertex in *clique* with vertex i, then the left child is activated. When the entire activation process has completed, all active processors represent cliques in the graph. Since the tree of processors has depth $n - 1$, and each processor needs no more than $O(n)$ time to determine which of its children, if any, are to be activated, the complexity of this algorithm is $O(n^2)$. Figure 8 shows the tree of activated processors for a small graph.

This algorithm cannot be used, of course, to solve any nontrivial problem: The exponential growth in the number of processors required quickly outstrips the capacity of any conceivable system. One solution, made possible by the fact that half the processors in the MIMD-BT model are leaf processors, would be to partition the problem until each leaf processor has been activated, and then to let the leaf processors finish the computation. Given $p = 2^k - 1$ processors, $1 \leq k \leq n$, the parallel algorithm would have complexity $O(kn2^{n-k})$. Browning [1980a] has proposed similar enumerative parallel algorithms to solve the color cost and traveling salesman problems on the MIMD-BT model.

3.5 Maximum Cardinality Matching

A *matching* in an undirected graph $G = (V, E)$ is a set $F \subseteq E$ such that no two edges in F are incident on the same vertex. A *maximum cardinality matching* (or simply *maximum matching*) in G is a set F such that F is a matching and there is no matching H in G such that $|H| > |F|$. A graph $G = (V, E)$ is *bipartite* if V consists of the union of two disjoint sets of vertices X and Y such that each edge in E connects a vertex in X with a vertex in Y. A bipartite graph is *convex* if there is an ordering of the vertices $X = \{x_1, x_2, \ldots, x_{|X|}\}$ and $Y = \{y_1, y_2, \ldots, y_{|Y|}\}$ such that for all triplets i, j, k with $i < j < k$, $(x_h, y_i) \in E$ and $(x_h, y_k) \in E \Rightarrow (x_h, y_j) \in E$.

Dekel and Sahni [1982] have developed an algorithm based on the SIMD-SM model to find the maximum matching of a convex bipartite graph. This problem has applications to scheduling [Dekel and Sahni 1983a, 1983b]. Their algorithm is a parallel version of Glover's [1967] algorithm.

Given a convex bipartite graph and an ordering for X and Y, s_i and h_i denote the smallest and highest vertices in Y to which x_i is adjacent. The first step of the parallel algorithm is to sort the vertices in X lexicographically by their s_i and h_i values. The vertices in X and Y are then partitioned, and subsets are assigned to leaves of a binary computation tree. For any node N, Avail(N) denotes the set of nodes of X that are available to be matched, and Choice(N) denotes the contiguous subset of the vertices of Y assigned to N. A maximum matching between Avail(N) and Choice(N) is obtained for all leaf nodes N. The matching at node N partitions Avail(N) into three disjoint subsets: Match(N), the nodes that are matched to nodes in Choice(N); Infeasible(N), the nodes that cannot be matched to any nodes in Y; and Transfer(N), the nodes that may be able to be matched to nodes in the Choice set of a node to the right of N in the binary tree.

When matching has been performed for both child nodes in the computation tree (Left(N) and Right(N)), matching is done for an interior node N as follows: Avail(N) is formed by merging Transfer(Left(N)) and Match(Right(N)). Then matching is done, using Glover's algorithm, with this new set of available vertices and Choice (Right(N)). The vertices matched in this step are merged with Match(Left(N)) to form Match(N). Infeasible(N) is the union of Infeasible(Left(N)) and the set of infeasible nodes generated from the matching done at node N.

The matching process works its way up the computation tree. All matchings at a given level can be done in parallel. The matching done at the root of the tree completes the matching of the entire graph. Once the matching has been done for the root node, the matched set of vertices in X must be propagated back down the tree in order to determine which vertices in Y are matched with the vertices in the root's Match set.

Sorting the elements of X has a complexity of $O(\log^2 n)$, given $n/2$ processors. Glover's algorithm has a complexity of $O(\log n)$. Matching requires $O(\log n)$ time; repeated for no more than $\log n$ levels, the total time taken to find a matching at the root is $O(\log^2 n)$. Propagating the matched set of vertices in X back down the tree requires $O(\log n)$ time for each of $\log n$ levels. The parallel maximum matching algorithm of Dekel and Sahni thus has a complexity of $O(\log^2 n)$ on an SIMD-SM model, given $O(n)$ processors.

Table 2. Parallel Minimum Spanning Tree Algorithms

Reference	Method	Model	Complexity	Processors
Levitt and Kautz [1972]	Kruskal	Systolic array	$O(n^2)$	$O(n^2)$
Savage [1977]	Sollin	SIMD-SM-SR	$O(\log^2 n)$	$n^2/\log n$
Bentley [1980]	Prim–Dijkstra	Tree	$O(n \log n)$	$n/\log n$
Deo and Yoo [1981]	Prim–Dijkstra	MIMD-TC	$O(n^{1.5})$	$n^{0.5}$
	Sollin	MIMD-TC	$O(n^2 \log n/p)$	$p \leq n$
Savage and Ja'Ja' [1981]	Sollin	SIMD-SM-R	$O(\log^2 n)$	n^2
Nath and Maheshwari [1982]	Sollin	SIMD-SM	$O(\log^2 n)$	$n^2/\log n$
Kučera [1982]	Kruskal	SIMD-SM-RW	$O(\log m)$	mn^4
Chin et al. [1982]	Sollin	SIMD-SM-R	$O(\log^2 n)$	$n\lceil n/\log^2 n \rceil$
Hambrusch [1982]	Sollin	Systolic array	$O(n)$	n^2
Hirschberg [1982]	Sollin	SIMD-SM-RW	$O(\log n)$	$O(n^3)$
Yoo [1983]	Kruskal	MIMD-TC	$O(m)$	$O(m)$

4. PARALLEL ALGORITHMS FOR WEIGHTED GRAPHS

Although most parallel graph algorithms reported in the literature are for unweighted graphs, some work has been done on developing parallel algorithms for weighted graphs. In this section we survey the results that have been reported thus far. As before, our primary goal is to pinpoint data structures and procedures that may be useful in developing new parallel algorithms.

4.1 Minimum Spanning Tree

Efforts to find the minimum spanning tree of a weighted, connected, undirected graph in parallel have focused on the three classical algorithms: Sollin's [1977] algorithm, the Prim–Dijkstra algorithm [Prim 1957; Dijkstra 1959], and Kruskal's [1956] algorithm. The work reported is summarized in Table 2.

Sollin's algorithm is the most obvious candidate for investigation. In Sollin's algorithm one starts with the forest of n isolated vertices, with every vertex regarded as a tree. In an iteration, the algorithm simultaneously determines for each tree in the forest the smallest edge joining any given vertex in that tree to a vertex in some other tree. All such edges are added to the forest, with the exception that two trees are never joined by more than one edge. (Ties between edges, which would cause a cycle, are resolved arbitrarily.) This process continues until there is only one tree in the forest—the minimum spanning tree. Since the number of trees is reduced by a factor of at least two in each iteration, Sollin's algorithm requires at most $\log n$ iterations to find the minimum spanning tree. An iteration requires at most $O(n^2)$ comparisons to find the smallest edge incident on each vertex. Thus the sequential algorithm has complexity $O(n^2 \log n)$. Yao's modification of Sollin's algorithm partitions the set of edges incident on each vertex before combining the trees, and has a complexity of $O(n^2 \log \log n)$ [Yao 1975].

The minimum spanning tree problem can also be solved by using Hirschberg's connected component algorithms and by recording the shortest edge between each pair of (super) vertices, which collapse to form a supervertex. Sollin's algorithm is similar to this approach. Indeed, Savage and Ja'Ja' [1981] have implemented Sollin's algorithm for an SIMD-SM-R model by utilizing Hirschberg's connected component algorithm to identify the new trees in the forest. They assume that n^2 processors are available and produce an algorithm of complexity $O(\log^2 n)$.

Chin et al. [1982] have also modified Hirschberg's algorithm to solve the minimum spanning tree problem. Their algorithm, which is based on the SIMD-SM-R model, has the same complexity: $O(\log^2 n)$. However, by restricting some operations to supervertices and by removing isolated components, their algorithm requires only $n\lceil n/\log^2 n \rceil$ processors. Thus it has the lowest cost of all parallel versions of Sollin's algorithm—$O(n^2)$.

Deo and Yoo [1981] have designed an algorithm for the MIMD-TC model on the

assumption that $p \leq n$ processors are available. Their parallel version of Sollin's algorithm is a straightforward conversion of the sequential algorithm. In each iteration every process examines $1/p$th of the vertices, finding each vertex's nearest nontree neighbor. Thus a single iteration has complexity $O(n^2/p)$, and the entire algorithm has complexity $O((n^2/p)\log n)$. Deo and Yoo have also reported that Cheriton and Tarjan's [1976] minimum-spanning tree algorithm, when parallelized, is identical to the parallel version of Sollin's algorithm.

Both Bentley [1980] and Deo and Yoo [1981] have parallelized the Prim–Dijkstra algorithm. The Prim–Dijkstra method chooses an arbitrary vertex, then repeatedly adds the nontree vertex closest to the partially formed minimum spanning tree. After $n - 1$ vertices have been added, all the vertices are in the tree, and the algorithm halts. The steps that find the nearest nontree vertex and update the distance from the tree to each nontree vertex are parallelized.

Bentley's algorithm is designed for the tree-structured systolic array. Every computation node is assigned log n vertices; thus $\lceil n/\log n \rceil$ computation nodes are required to accommodate the entire graph. Each time a vertex is added to the minimum spanning tree, the name of that vertex is broadcast through the tree to the computation nodes in $O(\log n)$ time. The computation nodes update the status of their assigned vertices. Of the nontree vertices that remain, each computation node issues the name of the vertex closest to the tree, the tree vertex to which it is closest, and the distance between them. This process takes $O(\log n)$ time. The combining processors then choose the candidate with the smallest distance, and the candidate nontree vertex closest to the tree is determined after $O(\log n)$ units of time. This process, illustrated in Figure 9, is repeated for $n - 1$ iterations, at which time the minimum spanning tree has been determined. Thus Bentley's algorithm has complexity $O(n \log n)$ on a tree machine with $n/\log n$ processors.

Deo and Yoo's algorithm is a direct parallelization of the sequential algorithm. Each processor updates the distance of $1/p$th of the nontree vertices, then contends with the other processors for access to a global variable, which contains the nearest nontree vertex. Owing to the frequent number of synchronizations required and the contention for the global variable, the overhead limits the number of processors to $O(n^{0.5})$. Thus Deo and Yoo's algorithm has complexity $O(n^{1.5})$ on an MIMD-TC model with $O(n^{0.5})$ processors.

Parallelization of Kruskal's algorithm has produced the most interesting results in terms of the development of data structures. The graph initially consists of a forest of isolated vertices. The edges are scanned in nondecreasing order of their weights, and every edge that connects two disjoint trees is added to the minimum spanning tree. (In other words, all edges that do not cause cycles with existing edges are selected.) The algorithm halts when the graph consists of a single tree, the minimum spanning tree.

Yoo [1983] has shown that an MIMD computer with $\lceil \log m \rceil$ processors can remove an element from an m-element heap in constant time. This *software pipeline* is illustrated in Figure 10. An array is used to implement the heap in the usual way [Aho et al. 1974]. The heap is a full binary tree with p levels, p being the number of processors. Some nodes in the bottom level of the tree are assigned the value ∞, if necessary, to fill the tree. During the course of an algorithm's execution, a node is *full* if it contains a value (including ∞). A node is *empty* if its value has been transmitted to its parent and no replacement value has been received from any of its children. The array *flag* indicates which levels contain an empty node; flag[i] = empty if level i has an empty node, otherwise flag[i] = full. If flag[i] = empty, then empty_node[i] indicates which node is the empty one. For all i, $2 \leq i \leq p$, processor i is assigned the task of keeping all the nodes at level $i - 1$ full. If flag[$i - 1$] = empty and flag[i] = full, then processor i fills the empty node at level $i - 1$ with the appropriate child at level i. Then flag[$i - 1$] becomes full, and flag[i] becomes empty. When a leaf node is emptied, it is filled with an ∞. Eventually

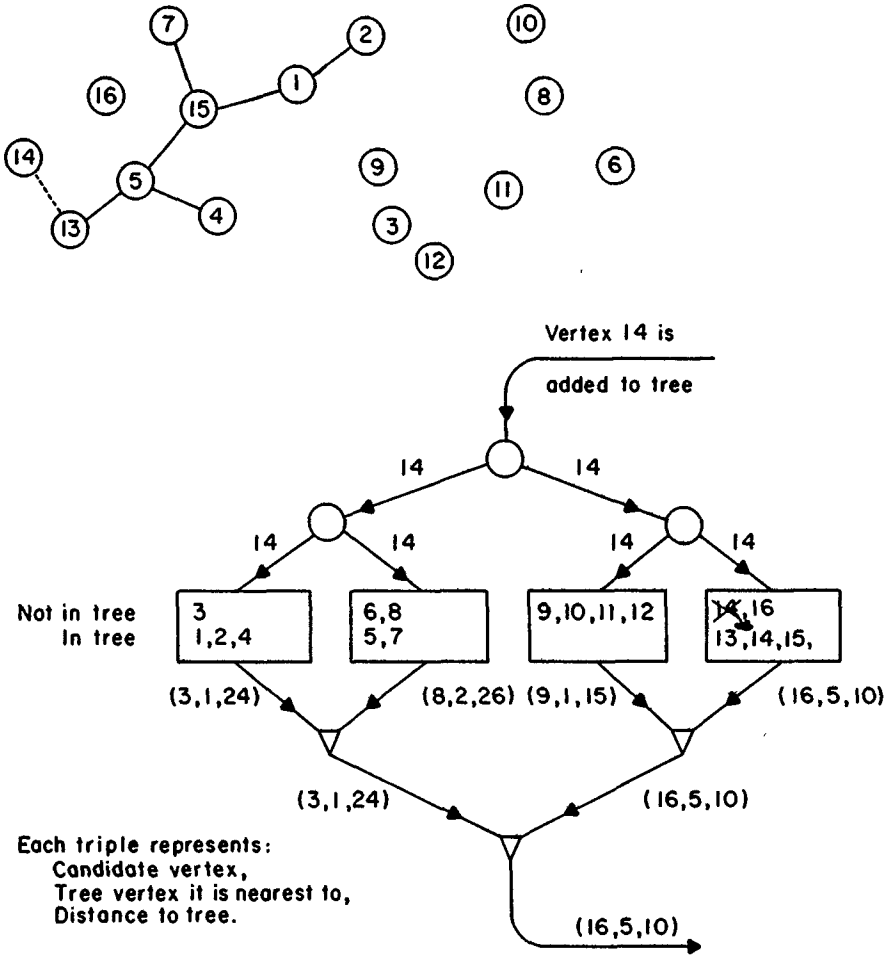


Figure 9. An iterative step of Bentley's [1980] minimum spanning tree algorithm. Each triple represents (1) the candidate vertex, (2) the tree vertex that it is nearest to, and (3) the distance to the tree.

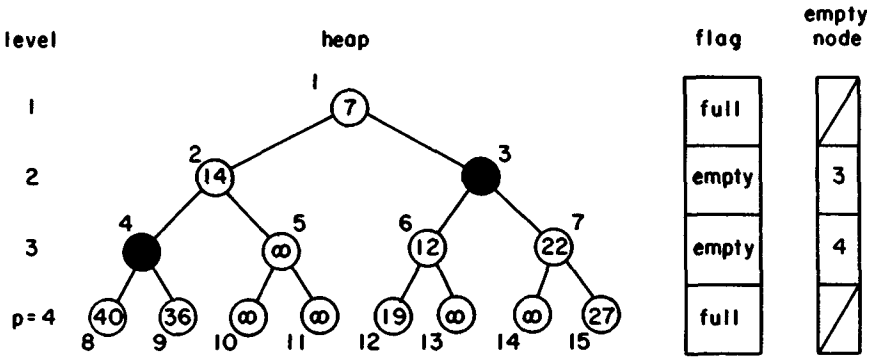


Figure 10. Yoo's [1983] software pipeline to an empty heap.

Table 3. Parallel All-Pairs Shortest Path Algorithms

Reference	Method	Model	Complexity	Processors
Levitt and Kautz [1972]	Warshall–Floyd	Systolic array	$O(n)$	n^2
Arjomandi [1975]	Warshall–Floyd	MIMD-TC	—	—
Savage [1977]	Repeated plus-min	SIMD-SM-R	$O(\log^2 n)$	$n^3/\log n$
Deo et al. [1980]	Warshall–Floyd	MIMD-TC	$O(n^3/p + pn)$	$p \ll n$
Dekel et al. [1981]	Repeated plus-min	SIMD-PS, SIMD-CC	$O(\log^2 n)$	n^3
Kučera [1982]	Repeated plus-min	SIMD-SM-RW	$O(\log n)$	n^4

the ∞'s fill the tree. Processor 1 empties node 1 whenever it is full, and terminates the procedure when node 1 has the value ∞ (i.e., when the heap is empty).

Yoo has also described a parallel method for initializing a heap. This method is a straightforward adaptation of the common sequential algorithm [Aho et al. 1974]; its difference is that all interior nodes at the same level of the heap are "heapified" in parallel. With his algorithm, $\lceil m/4 \rceil$ processors can heapify a binary tree in $\lceil \log m \rceil$ iterations, with each iteration requiring $O(\log m)$ time. Thus Kruskal's algorithm can be implemented on an MIMD-TC computer by using a heap whose initialization and emptying costs are $O(m \log^2 m)$ and $O(m \log m)$, respectively. The entire algorithm has complexity $O(m)$ with $O(m)$ processors.

Other references to parallel minimum spanning tree algorithms in the literature include Atallah [1983], Atallah and Kosaraju [1982], Hambrusch [1982], Hirschberg [1982], Kučera [1982], Levitt and Kautz [1972], Reif [1982], and C. Savage [1978].

4.2 Shortest Path

Shortest-path problems have enormous practical importance in the study of transportation and communication networks. Parallel algorithms for two kinds of shortest-path problems have been reported: (1) finding the shortest path from a specified vertex to all other vertices in a network (the single-source shortest-path problem), and (2) finding the shortest path between every pair of vertices in a network (the all-pairs shortest-path problem). A chronology of these research efforts is illustrated in Table 3. In a directed network edges may have positive, zero, or negative weights, as long as there are no negative weight cycles (which would make the shortest path to at least some of the vertices undefined). Likewise, since edges in an undirected network can be traversed in either direction, undirected networks must have nonnegative edge weights.

Crane [1968] has proposed a parallel single-source shortest-path algorithm for use on a machine with associative memory. Beginning at the source, all paths are explored at the same rate, so that the first path to reach a vertex is by definition the shortest path. Each iteration advances all the paths by "a distance equal to the shortest distance between a present path ending and a vertex not already contained in a path [p. 691]." The paths are advanced by subtracting this shortest distance from the lengths of all the edges containing path endings. Associative memory is used during each iteration to find the minimum of p values and to do p subtractions, where p—the number of processors—is also the current under of path endings. Crane does not analyze the complexity of this algorithm. (It should be noted that Crane's algorithm is suitable only for graphs with nonnegative edge weights, since once the distance to a vertex is determined, it is not modified. Thus it cannot handle the case in which the shortest path to a vertex begins by going to a farther vertex and then returning via a path of negative weight.)

Dekel et al. [1981] have devised matrix multiplication algorithms for the SIMD-CC and SIMD-PS models. Given $n^3 = 2^{3q}$ processors, both algorithms can multiply two $n \times n$ matrices in $O(\log n)$ time. The key to their algorithms is the data-routing strategy. For the SIMD-CC model, $5q = 5 \log n$ routing steps are sufficient to broadcast the initial values through the processor

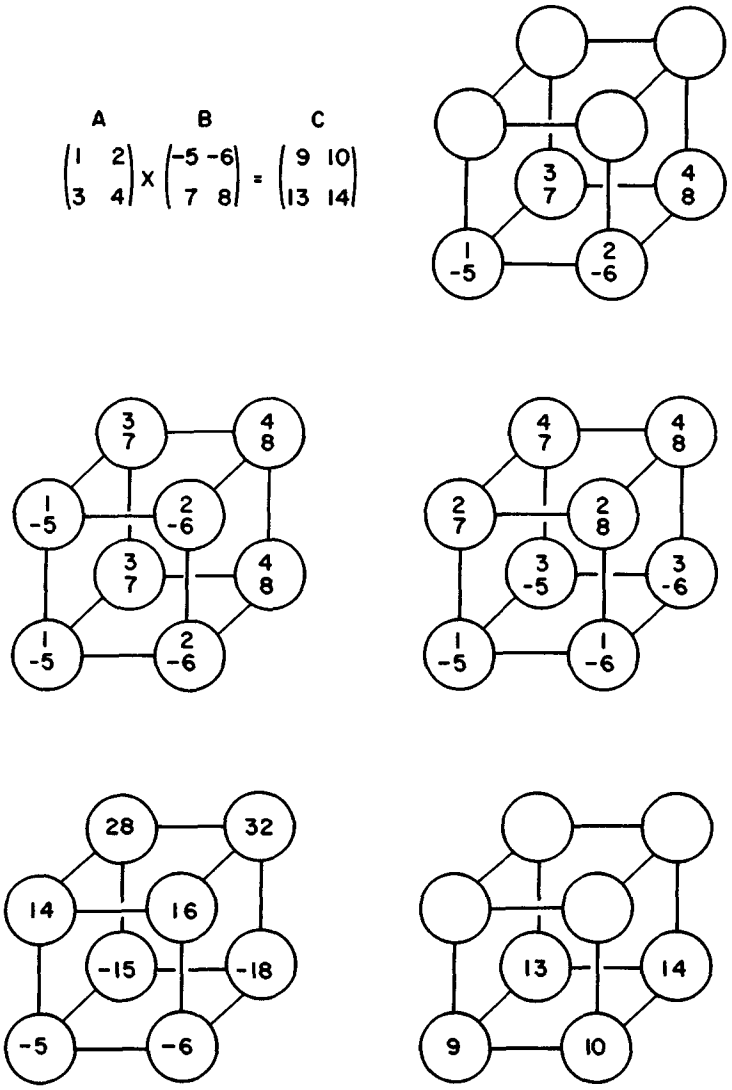


Figure 11. Parallel matrix multiplication on an SIMD-CC model [Dekel et al. 1981].

array and to combine the results. Figure 11 illustrates the multiplication of two 2×2 matrices on an eight-element processor array. Dekel and his co-workers also demonstrate that $10q = 10 \log n$ routing steps are sufficient to perform matrix multiplication on the SIMD-PS model. This result is significant because the SIMD-PS model is more realistic than the SIMD-CC model, as it requires only three connections (as opposed to $\log p$ connections) per processor.

The fast matrix multiplication algorithms of Dekel and his associates can be used to solve a number of graph problems efficiently, including the all-pairs shortest-path problem. For example, given an n-vertex weighted graph G, the goal would be to produce an $n \times n$ matrix A such that a_{ij} is the length of the shortest path from i to j in G. The a_{ij}^k denotes the length of the shortest path from i to j with at most k intermediate vertices. Since there are no

negative weight cycles in G, $a_{ij} = a_{ij}^n$. In this example, $a_{ii}^0 = 0$, for all i, $1 \leq i \leq n$, and for all distinct i and j, a_{ij}^0 is the weight of the edge from i to j; if no such edge exists, $a_{ij}^0 = \infty$. It follows from the principle of combinatorial optimality that $a_{ij}^k = \min_m\{a_{im}^{k/2} + a_{mj}^{k/2}\}$. Hence A^n may be computed from A^0 by repeated plus–min multiplication. Substituting + for * and min for +, $\lceil \log n \rceil + 1$ matrix multiplications are sufficient to generate the matrices A^1, A^2, A^4, ..., A^n. Thus the all-pairs shortest-path problem can be solved in $O(\log^2 n)$ time on the SIMD-CC and SIMD-PS models, given n^3 processors.

Other reported work on parallel shortest-path algorithms includes Arjomandi [1975], Deo et al. [1980], Kučera [1982], Levitt and Kautz [1972], Mateti and Deo [1981], Price [1982, 1983], Quinn [1983], and Yoo [1983]. Some of these algorithms are parallelizations of the Warshall–Floyd all-pairs shortest-path algorithm [Warshall 1962; Floyd 1962]; others are based on Moore algorithms to solve the single-source shortest-path problem [Moore 1957; Pape 1974].

4.3 The Traveling Salesman Problem

Given a complete weighted graph in which the weight of the edge (i, j) represents the distance from i to j, the traveling salesman problem (TSP) is to find a cycle (tour) of minimum length that goes through every vertex exactly once. The TSP is NP-hard, and all known algorithms to find an exact solution to the TSP require exponential time in the worst case. Lessening the difficulty of the TSP does not concomitantly reduce complexity. For example, the Euclidean traveling salesman problem (ETSP), in which vertices are mapped to points in Euclidean space and all distances are the Euclidean distances, is also NP-hard.

Approaches to the parallelization of TSP algorithms reflect the relationship between a parallel algorithm and the model of parallel computation for which it is designed. Mohan [1983] and Quinn and Deo [1983] have implemented their algorithms on Cm* and the HEP, respectively, and are primarily concerned with the issues of speedup and efficiency. Browning [1980a], on the other hand, designing for the VLSI-based MIMD-BT model, suggests taming the difficulty of the problem by using an exponential number of processors (see Section 3.4).

Mohan [1983] has parallelized the branch-and-bound algorithm of Little et al. [1963] to find an exact solution to the TSP. One way to view a branch-and-bound algorithm is to imagine a tree in which nodes represent subproblems and the children of a node are the subproblems generated when a given node is decomposed. The root of the tree constitutes, of course, the initial problem. The search strategy used determines the size and the shape of this tree. Best-bound search minimizes the number of subproblems examined (the number of interior nodes) before termination, although the amount of memory space it requires (the total number of nodes) is often exponential in the problem size (i.e., depth of the search tree) [Ibaraki 1976]. It is this exponential growth of the size of the search space that makes the parallelization of branch and bound so attractive: There is sufficient work for the processors to do, and the processors are able to work independently.

Mohan creates a number of processes that asynchronously explore the tree of subproblems until a solution has been found. Each process repeatedly removes the unexplored subproblem with the smallest lower bound from the ordered list of unexplored subproblems, decomposes the problem (unless it can be solved directly), and inserts the two newly created subproblems in their proper places in the ordered list of problems to be explored. A process must have exclusive control of the list in order to insert and delete elements, but the time taken for these tasks is relatively small compared to the time needed to decompose a problem. Thus contention for this list should not be a significant inhibitor of speedup. In fact, Mohan's experience with this algorithm on Cm* indicates that speedup is good: His algorithm has achieved a speedup of about 8 with 16 processors when solving a 30-vertex TSP. The major obstacle to higher speedup is intra-

cluster contention, or contention by computer modules within a cluster for shared resources. These resources include the Kmap, the Map bus, and the Object Manager. The Object Manager is used to create the nodes of the search tree.

The fact that all known exact algorithms to solve the TSP require exponential time in the worst case has prompted a variety of approximate algorithms that find a suboptimal solution in polynomial time [Reingold et al. 1977]. Quinn and Deo [1983] have parallelized the farthest insertion heuristic [Rosenkrantz et al. 1974], one of the better approximate algorithms for the ETSP, because of its low-order complexity ($O(n^2)$) and its good suboptimal solutions [Golden et al. 1980]. Quinn and Deo's algorithm, which is based on the MIMD-TC model, is a straightforward parallelization of the sequential farthest-insertion algorithm. Synchronization overhead limits the number of processors that can be used to $O(n^{0.5})$. Nevertheless, the use of $O(n^{0.5})$ processors is sufficient to reduce the complexity of the algorithm from $O(n^2)$ to $O(n^{1.5})$. Run on Denelcor's HEP, their algorithm achieves a speedup of about 4.5 with eight processes, given a 100-vertex network.

4.4 Other Parallel Algorithms for Weighted Graphs

Shiloach and Vishkin [1982b] have developed a parallel algorithm for finding the maximum flow in a directed, weighted graph. Their algorithm utilizes the usual layered network approach. Given $p \leq n$ processors. Shiloach and Vishkin's algorithm has a complexity of $O(n^3 \log n/p)$. This problem is log space complete for p [Goldschager et al. 1982]. Chen and Feng [1973] and Chen [1975] discuss parallel algorithms for the maximum capacity path problem.

5. SUMMARY

Parallel graph algorithms have been based on several models of parallel computation. The most popular of these has been the SIMD-SM (shared memory) model, a direct extension of the standard RAM model of sequential computation. Its popularity may derive from its ancestry, as well as the fact that it submerges communications issues, allowing designers to concentrate on minimizing the number of operations performed. However, the SIMD-SM model is unrealistic: Too many connections are required between the processors and memory. Thus algorithms designed for the SIMD-SM model may never be implemented.

Among other SIMD models used, the SIMD-MC (mesh connected) is the basis for the ILLIAC-IV and other parallel computers. However, interprocessor communication costs often dictate the complexity of a parallel algorithm, and therefore many problems, such as sorting and matrix multiplication, cannot be solved quickly on this model due to the time spent on routing data. The SIMD-CC (cube-connected) model allows for more efficient data routing, but the number of connections per processor grows logarithmically in the number of processing elements. The SIMD-PS (perfect shuffle) model uses three connections per processing element and has been the basis for several fast parallel algorithms. The SIMD-CCC (cube-connected cycles) model can emulate the SIMD-CC and SIMD-PS models, and possess the additional advantage that it can be implemented in VLSI with more regularity.

The MIMD-TC (tightly coupled) model has spawned actual parallel computers, such as C.mmp and Denelcor's HEP, but the complexity of the processor–memory switching mechanism puts a damper on the maximum number of processors that can be assembled into a single computer. The MIMD-LC (loosely coupled) and MIMD-BT (binary tree) models represent efforts to maintain generality inherent in MIMD architectures while avoiding the problems caused by the processor–memory switching mechanism of the MIMD-TC model. Advances in VLSI technology also have led to a growing interest in systolic arrays for the solution of graph problems.

The connected components problem has been the focus of much activity—a wide range of problems are related to it. Parallel solutions to the connected components

problem have been based on three fundamental approaches: breadth-first search, transitive closure, and vertex collapse.

Dekel and his associates have shown that matrix multiplication can be performed in $O(\log^2 n)$ time on an n^3 processing element SIMD-PS or SIMD-CC model. This fast parallel matrix multiplication algorithm is the basis for a number of parallel graph algorithms, including transitive closure, breadth-first spanning tree, and all-pairs shortest path.

Dekel and Sahni have developed an SIMD-SM algorithm to solve the maximum cardinality matching problem for convex bipartite graphs. This problem is amenable to a divide-and-conquer approach; its parallelization fits a binary computation tree.

Mohan's solution of the traveling salesman problem on Cm* illustrates the suitability of using an MIMD model to solve intractable problems. As there is plenty of work for the processors to do, they do not wait for other processors to provide them with work. Contention for the shared list of unexplored subproblems is limited by the large amount of processing needed to expand a single subproblem. The parallel algorithm thus achieves good speedup. Browning has examined the use of exhaustive search to solve intractable problems on the MIMD-BT model.

We began by noting that graph theory has wide applications in science and engineering. The development of parallel graph algorithms is still in its infancy. Few implementations of graph algorithms on parallel computers have been reported. As parallel computers become more readily available, we anticipate great advances in the quantity and applicability of parallel graph algorithms.

ACKNOWLEDGMENTS

We wish to thank J. Denbigh Starkey and Karl Winklmann for their careful reading of the manuscript and their constructive suggestions. We are grateful to the referees and editors for their helpful criticisms that led to significant improvements in the survey.

This work was partially supported by U.S. Army Research Office grant DAAG29-82-K-0107.

REFERENCES

AHO, A., HOPCROFT, J., AND ULLMAN, J. 1974. *The Design and Analysis of Computer Algorithms* Addison-Wesley, Reading, Mass.

ALTON, D. A., AND ECKSTEIN, D. M. 1979. Parallel breadth-first search of p-sparse graphs. In *Proceedings of the West Coast Conference on Combinatorics, Graph Theory and Computing* (Arcata, Calif., Sept. 5–7). Humbolt State Univ. *Congressus Numerantium 26.*

ARJOMANDI, E. 1975. A study of parallelism in graph theory. Ph.D dissertation, Dept. of Computer Science, University of Toronto, Toronto, Ontario.

ATALLAH, M. J. 1983. Algorithms for VLSI networks of processors. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, The Johns Hopkins Univ., Baltimore, Md.

ATALLAH, M. J., AND KOSARAJU, S. R. 1982. Graph problems on a mesh-connected processor array (preliminary version). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing* (San Francisco, Calif., May 5–7). ACM, New York, pp. 345–353.

BATCHER, K. E. 1968. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference* (Atlantic City, N.J., Apr. 30–May 2), vol. 32. AFIPS Press, Reston, Va., pp. 307–314.

BATCHER, K. E. 1979. The STARAN computer. In *Infotech State of the Art Report: Supercomputers*, vol. 2, C. R. Jesshope and R. C. Hockney, Eds. Infotech, Maidenhead, England, pp. 33–49.

BATCHER, K. E. 1980. Design of massively parallel processor. *IEEE Trans Comput. C-29*, 836–840.

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

BENTLEY, J. L., AND KUNG, H. T. 1979. A tree machine for searching problems. In *Proceedings of the 1979 International Conference on Parallel Processing*. IEEE, New York, pp. 257–266.

BERG, H. K., BOEBERT, W. E., FRANTA, W. R., AND MOHER, T. G. 1982. *Formal Methods of Program Verification and Specification.* Prentice-Hall, Englewood Cliffs, N.J., chap. 6.

BILARDI, G., PRACCHI, M., AND PREPARATA, F. P. 1981. A critique and appraisal of VLSI models of computation. In *VLSI Systems and Computations*, H. T. Kung, R. Sproull, and G. Steele, Eds. Computer Science Press, Rockville, Md., pp. 81–88.

BROWNING, S. A. 1980a. The tree machine: A highly concurrent computing environment. Ph.D. dissertation, Dept. of Computer Science, California Inst. of Technology, Pasadena, Calif.

BROWNING, S. A. 1980b. Algorithms for the tree machine. In *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Addison-Wesley, Reading, Mass.

CHANDRA, A. K. 1976. Maximal parallelism in matrix multiplication. IBM Tech. Rep. RC6193, Thomas J. Watson Research Center, Yorktown Heights, N.Y. (Sept.), 9 pp.

CHANDRA, A. K., AND STOCKMEYER, L. J. 1976. Alternation. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 98–108.

CHANDY, K. M , AND MISRA, J. 1982. Distributed computation on graphs: Shortest path algorithms. *Commun ACM 25*, 11 (Nov.), 833–837.

CHAZELLE, B., AND MONIER, L. 1981. A model of computation for VLSI with related complexity results. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing* (Milwaukee, Wis., May 11–13). ACM, New York, pp. 318–325.

CHEN, I.-N. 1975. A new parallel algorithm for network flow problems. In *Parallel Processing, Lecture Notes in Computer Sciences*, vol. 24. Springer-Verlag, Berlin and New York, pp. 306–307.

CHEN, Y. K., AND FENG, T. 1973. A parallel algorithm for maximum flow problem. In *Proceedings of the 1973 Computer Conference on Parallel Processing* (Sagamore, N.Y.), p. 60.

CHERITON, D., AND TARJAN, R. E. 1976. Finding minimum spanning trees. *SIAM J. Comput. 5*, 4 (Dec.), 724–742.

CHIN, F. Y., LAM, J., AND CHEN, I.-N. 1981. Optimal parallel algorithms for the connected component problem. In *Proceedings of the 1981 International Conference on Parallel Processing.* IEEE, New York, pp. 170–175.

CHIN, F. Y., LAM, J., AND CHEN, I.-N. 1982. Efficient parallel algorithms for some graph problem. *Commun ACM 25*, 9 (Sept.), 659–665.

COOK, S. A. 1974. An observation on time-storage trade-off. *J. Comput. Syst Sci. 9*, 3, 308–316.

CRANE, B. A. 1968. Path finding with associative memory. *IEEE Trans Comput. C-17*, 7 (July), 691–693.

CRANE, B. A., GILMARTIN, M. J., HUTTENHOFF, J. H., RUX, P. T., AND SHIVELY, R. R 1972. PEPE computer architecture. *COMPCON 72 Digest*, IEEE, New York, pp. 57–60.

DEKEL, E., AND SAHNI, S. 1982. A parallel matching algorithm for convex bipartite graphs. In *Proceedings of the 1982 International Conference on Parallel Processing.* IEEE, New York, pp. 178–184.

DEKEL, E , AND SAHNI, S. 1983a. Parallel scheduling algorithms. *Oper Res. 31*, 1 (Jan.–Feb.), 24–49.

DEKEL, E., AND SAHNI, S. 1983b. Binary trees and parallel scheduling algorithms. *IEEE Trans Comput. C-32*, 3 (Mar.), 307–315.

DEKEL, E., NASSIMI, D., AND SAHNI, S. 1981. Parallel matrix and graph algorithms. *SIAM J. Comput. 10*, 4 (Nov.), 657–675.

DENELCOR 1981. Heterogeneous element processor principles of operation. Publ. No. 9000001, HEP Technical Documentation Series, Denelcor, Inc., Denver, Colo.

DEO, N. 1974. *Graph Theory with Applications to Engineering and Computer Science.* Prentice-Hall, New York.

DEO, N., AND YOO, Y. B. 1981. Parallel algorithms for the minimum spanning tree problem. In *Proceedings of the 1981 International Conference on Parallel Processing* IEEE, New York, pp. 188–189.

DEO, N., PANG, C. Y., AND LORD, P. E. 1980. Two parallel algorithms for shortest path problems. In *Proceedings of the 1980 International Conference on Parallel Processing.* IEEE, New York, pp. 244–253.

DIJKSTRA, E. 1959. A note on two problems in connexion with graphs. *Numer. Math 1*, 269–271.

DOBKIN, D., LIPTON, R. J., AND REISS, S. 1979. Linear programming is log-space hard for *P. Inf. Process Lett. 9*, 2 (Aug.), 6–97.

DYMOND, P. W., AND COOK, S. A. 1980. Hardware complexity and parallel computation (preliminary version). In *Proceedings of the 21st Annual Symposium on Foundations of Computing.* IEEE, New York, pp. 360–372.

ECKSTEIN, D. M. 1979a. Simultaneous memory accesses. 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, 55 pp.

ECKSTEIN, D. M., AND ALTON, D. A. 1977a. Parallel searching of non-sparse graphs. Tech. Rep. 77-02, Dept. of Computer Science, The Univ. of Iowa, Iowa City, Iowa, 35 pp.

ECKSTEIN, D. M., AND ALTON, D. A. 1977b. Parallel graph processing using depth-first search. In *Proceedings of the Conference on Theoretical Computer Science* (Waterloo, Ontario, Aug.). Univ. of Waterloo, Waterloo, Ontario, pp. 21–29.

FALK, H. 1976. Reaching for the gigaflop. *IEEE Spectrum 13*, 10 (Oct.), 64–70.

FEIERBACH, G., AND STEVENSON, D. 1979 The ILLIAC IV. In *Infotech State of the Art Report. Supercomputers*, vol. 2, C. R. Jesshope and R. W. Hockney, Eds. Infotech, Maidenhead, England, pp. 77–92.

FLANDERS, P. M., HUNT, D. J., REDDAWAY, S. F., AND PARKINSON, D. 1977. Efficient high speed computing with the distributed array processor. In *High Speed Computer and Algorithm Organisation.* Academic Press, London, pp. 113–128.

FLOYD, R. W. 1962. Algorithm 97: Shortest path. *Commun. ACM 5*, 6 (June), 345.

FLYNN, M. J. 1966. Very high-speed computing systems. *Proc IEEE 54*, 12 (Dec.), 1901–1909.

FOSTER, M. J., AND KUNG, H. T. 1980. Design of special-purpose VLSI chips—Example and opinions. *Computer 13*, 1 (Jan.), 26–40.

FULLER, S. H., AND OLEINICK, P. N. 1976. Initial measurements of parallel programs in a multi-

miniprocessor. In *Proceedings of the 13th IEEE Computer Society International Conference.* IEEE, New York, pp. 358–363.

FUNG, L. 1977. A massively parallel processing computer. In *High Speed Computer and Algorithm Organisation*, D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Eds. Academic Press, London, England, pp. 203–204.

GALIL, Z. 1976. Hierarchies of complete problems. *Acta Inf. 6*, 77–88.

GALIL, Z., AND PAUL, W. J. 1981. An efficient general purpose parallel computer. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing* (Milwaukee, Wis., May 11–13). ACM, New York, pp. 247–262.

GALIL, Z., AND PAUL, W. J. 1983. An efficient general-purpose parallel computer. *J ACM 30*, 2 (Apr.), 360–387.

GALLAGHER, R. G., HUMBLET, P. A., AND SPIRA, P. M. 1983. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang Syst. 5*, 1 (Jan.), 66–77.

GAREY, M. R., AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of the NP-Completeness.* Freeman, San Francisco, Calif.

GENTLEMAN, W. M. 1978. Some complexity results for matrix computations on parallel processors. *J. ACM 25*, 1 (Jan.), 112–115.

GLOVER, F. 1967. Maximum watching in a convex bipartite graph. In *Naval Res Logist Q. 14*, 313–316.

GOLDEN, B., BODIN, L., DOYLE, T., AND STEWART, W., JR 1980 Approximate traveling salesman algorithms. *Oper. Res. 28*, 3 (May–June), Part 2, 694–711.

GOLDSCHLAGER, L. M. 1977a. The monotone and planar circuit value problems are log space complete for *P SIGACT News 9*, 2 (Summer), 25–29.

GOLDSCHLAGER, L. M. 1977b Synchronous parallel computation. Tech. Rep. 114, Computer Science Dept., University of Toronto, Toronto, Ontario.

GOLDSCHLAGER, L. M. 1978. A unified approach to models of synchronous parallel machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing* ACM, New York, pp. 89–94.

GOLDSCHLAGER, L. M. 1982. A universal interconnection pattern for parallel computers. *J. ACM 29*, 4 (Oct.), 1073–1086.

GOLDSCHLAGER, L. M., SHAW, R. A., AND STAPLES, J. 1982. The maximum flow problem is log space complete for *P. Theor Comput. Sci 21* (Oct.), 105–111.

GREIF, I. 1977. A language for formal problem specifications. *Commun ACM 20*, 12 (Dec.), 931–935.

GUIBAS, L. J., KUNG, H. T , AND THOMPSON, C D. 1979. Direct VLSI implementation of combinatorial algorithms. In *Proceedings of the Conference on Very Large Scale Integration· Architecture, Design, Fabrication* (Pasadena, Calif., Jan.). California Institute of Technology, Pasadena, Calif., pp. 509–525.

HAMBRUSCH, S. E. 1982. The complexity of graph problems on VLSI. Ph.D. dissertation, Computer Science Dept., The Pennsylvania State Univ., University Park, Pa.

HAMBRUSCH, S. E. 1983. VLSI algorithms for the connected component problem. *SIAM J. Comput. 12*, 2 (May), 364–365.

HARARY, F. 1969. *Graph Theory.* Addison-Wesley, Reading, Mass.

HARRISON, T. J., AND WILSON, M. W. 1983. Special-purpose computers. In *Encyclopedia of Computer Science and Engineering*, 2nd ed., A. Ralston, Ed. Van Nostrand-Reinhold, New York, pp. 1385–1393.

HAYNES, L. S., LAU, R. L., SIEWIOREK, D. P., AND MIZELL, D. 1982. A survey of highly parallel computing. *Computer 14*, 1 (Jan.), 9–24.

HIGBIE, L. C. 1972. The OMEN computers: Associative array processors. In *COMPCON 72 Digest*, IEEE, New York, pp. 287–290.

HIRSCHBERG, D. S. 1976. Parallel algorithms for the transitive closure and the connected component problem. In *Proceedings of the 8th Annual ACM Symposium on the Theory of Computing.* ACM, New York, pp. 55–57.

HIRSCHBERG, D. S. 1982. Parallel graph algorithms without memory conflicts. In *Proceedings of the 20th Allerton Conference.* University of Illinois, Urbana-Champaign, Ill., pp. 257–263.

HIRSCHBERG, D. S., CHANDRA, A. K., AND SARWATE, D. V. 1979. Computing connected components on parallel computers. *Commun. ACM 22*, 8 (Aug.), 461–464.

HOCKNEY, R. W., AND JESSHOPE, C. R. 1981. *Parallel Computers: Architecture, Programming, and Algorithms.* Adam Hilger, Bristol, England.

IBARAKI, T. 1976. Theoretical comparisons of search strategies in branch-and-bound algorithms. *Int. J Comput Inf. Sci. 5*, 4, 315–344.

JA'JA', J., AND SIMON, J. 1982. Parallel algorithms in graph theory: Planarity testing. *SIAM J. Comput 11*, 2 (May), 314–328.

JOHNSON, D. S. 1983. The NP-completeness column: An ongoing guide. *J. Algorithms 4*, 189–203.

JONES, A. K., AND SCHWARZ, P. 1980. Experience using multiprocessor systems—A status report. *ACM Comput. Surv. 12*, 2 (June), 121–165.

JONES, N. D., AND LAASER, W. T. 1976. Complete problems for deterministic polynomial time. *Theor. Comput. Sci. 3*, 1, 105–117.

KEDEM, Z. M., AND ZORAT, A. 1981. On relations between input and communication/computation in VLSI (preliminary report). In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 37–44.

KELLER, R M 1976. Formal verification of parallel programs. *Commun. ACM 19*, 7 (July), 371–384.

KOSARAJU, S. R. 1979. Fast parallel processing array algorithms for some graph problems. In *Proceed-*

ings of the 11th Annual ACM Symposium on Theory of Computing ACM, New York, pp. 231–236.

KOZEN, D. 1977. Complexity of finitely presented algebras. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing.* ACM, New York, pp. 164–177.

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

KUČERA, L. 1982. Parallel computation and conflicts in memory access. *Inf Process Lett. 14*, 2 (20 Apr.), 93–96.

KUNG, H. T. 1980. The structure of parallel algorithms. In *Advances in Computers*, vol. 19, M. Yovits, Ed Academic Press, New York, pp. 65–112

KUNG, H. T. 1982. Why systolic architectures? *Computer* 15, 1 (Jan.), 37–46.

KUNG, H. T., AND LEISERSON, C. E. 1980. Systolic arrays for VLSI. In *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Addison-Wesley, Reading, Mass., pp. 260–292.

LADNER, R. E. 1975. The circuit value problem is log space complete for *P. SIGACT News 7*, 1 (Jan.), 18–20.

LAMPORT, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans Soft. Eng. SE-3*, 7 (Mar.), 125–143.

LAMPORT, L 1979. Proving the correctness of multiprocess programs. *ACM Trans Progress Lang. Sys 1*, 1 (July), 86–97.

LAWRIE, D. H. 1975. Access and alignment of data in an array processor. *IEEE Trans Comput. C-24*, 12 (Dec.), 1145–1155.

LEIGHTON, F. T. 1981. New lower bound techniques for VLSI In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science* IEEE, New York, pp. 1–12.

LEISERSON, C. E. 1980. Area efficient graph layouts. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science* IEEE, New York, pp. 270–281.

LEISERSON, C. E. 1983. *Area-Efficient VLSI Computation* MIT Press, Cambridge, Mass.

LEISERSON, C. E., AND SAXE, J. B. 1981. Optimizing synchronous systems. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 23–36.

LEVIALDI, S. 1972. On shrinking binary picture patterns. *Commun ACM 15*, 1 (Jan.), 2–10.

LEVITT, K N., AND KAUTZ, W. T. 1972 Cellular arrays for the solution of graph problems. *Commun ACM 15*, 9 (Sept), 789–801

LINT, B., AND AGERWALA, T. 1981 Communication issues in the design and analysis of parallel algorithms *IEEE Trans Softw Eng. SE-7*, 2 (Mar.), 174–188.

LIPTON, R J., AND VALDES, J. 1981. Census functions: An approach to VLSI upper bounds (preliminary version). In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 13–22.

LITTLE, J. D. C., MURTY, K. G., SWEENEY, D. W., AND KAREL, C. 1963. An algorithm for the traveling salesman problem. *Oper. Res 11*, 6 (Nov.–Dec.), 972–989.

MASHBURN, H. H. 1979. The C.mmp/Hydra project: An architectural overview. Tech. Rep., Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

MATETI, P., AND DEO, N. 1981. Parallel algorithms for the single source shortest path problem. Tech. Rep. CS-81-078, Computer Science Dept., Washington State Univ., Pullman, Wash., 38 pp.

MEAD, C., AND CONWAY, L. 1980. *Introduction to VLSI Systems* Addison-Wesley, Reading, Mass.

MISRA, J., AND CHANDY, K. M. 1982. A distributed graph algorithm: Knot detection. *ACM Trans Progress. Lang. Syst 4*, 4 (Oct.), 678–686.

MOHAN, J. 1983. Experience with two parallel programs solving the traveling salesman problem. In *Proceedings of the 1983 International Conference on Parallel Processing* IEEE, New York, pp. 191–193

MOORE, E. F. 1957. The shortest path through a maze In *Proceedings of the International Symposium on the Theory of Switching*, vol. 2, pp. 285–292.

NASSIMI, D , AND SAHNI, S. 1979. Bitonic sort on a mesh connected parallel computer. *IEEE Trans Comput. C-28*, 1 (Jan.), 2–7.

NASSIMI, D., AND SAHNI, S 1980a An optimal routing algorithm for mesh-connected parallel computers. *J. ACM 27*, 1 (Jan.), 6–29

NASSIMI, D., AND SAHNI, S. 1980b. Finding connected components and connected ones on a mesh-connected parallel computer. *SIAM J Comput 9*, 4 (Nov.), 744–757.

NATH, D., AND MAHESHWARI, S. N. 1982. Parallel algorithms for the connected components and minimal spanning tree problems. *Inf. Process. Lett 14*, 1 (27 Mar.), 7–11.

OLEINICK, P. 1978. The implementation of parallel algorithms on an asynchronous multiprocessor. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

OWICKI, S., AND GRIES, D. 1976. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM 19*, 5 (May), 279–285.

OWICKI, S., AND LAMPORT, L. 1982. Proving liveness properties of concurrent programs. *ACM Trans Program. Lang Sys 4*, 3 (July), 455–495.

PAPE, U. 1974. Implementation and efficiency of Moore-algorithms for the shortest route problem *Math Program 7*, 2 (Oct.), 212–222.

PREPARATA, F. P., AND VUILLEMIN, J. 1981. The cube-connected cycles: A versatile network for parallel computation. *Commun ACM 24*, 5 (May), 300–309.

PRICE, C. C. 1982. A VLSI algorithm for shortest path through a directed acyclic graph., *Congressus Numerantium 34*, 363–371.

PRICE, C C. 1983. Task assignment using a VLSI shortest path algorithm Tech. Rep., Dept. of Computer Science, Stephen F. Austin State Univ., Nacogdoches, Tex.

PRIM, R. C. 1957. Shortest connection networks and some generalizations. *Bell Syst Tech J. 36*, 1389–1401.

QUINN, M. J. 1983. The design and analysis of algorithms and data structures for the efficient solution of graph theoretic problems on MIMD computers. Ph.D. dissertation, Computer Science Dept., Washington State Univ, Pullman, Wash.

QUINN, M J., AND DEO, N. 1983. An approximate algorithm for the Euclidean traveling salesman problem Tech. Rep. CS-83-105, Computer Science Dept., Washington State Univ., Pullman, Wash

REDDAWAY, S. F 1979 The DAP approach In *Infotech State of the Art Report· Supercomputers*, vol. 2, C. R. Jesshope and R. W. Hockney, Eds. Infotech, Maidenhead, England, pp. 311–329.

REGHBATI (ARJOMANDI), E., AND CORNEIL, D. G. 1978. Parallel computations in graph theory. *SIAM J. Comput 2*, 2 (May), 230–237.

REIF, J. H. 1982 Symmetric complementation. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing* (San Francisco, Calif., May 5–7). ACM, New York, pp. 201–214.

REIF, J. H., AND SPIRAKIS, P 1982. The expected time complexity of parallel graph and digraph algorithms. Tech. Rep. TR-11-82, Aiken Computation Laboratory, Harvard Univ , Cambridge, Mass

REINGOLD, E., NIEVERGELT, J., AND DEO, N. 1977. *Combinatorial Algorithms Theory and Practice* Prentice-Hall, New York.

ROSENKRANTZ, D., STEARNS, R , AND LEWIS, P. 1974. Approximate algorithms for the traveling saleperson problem. In *Proceedings of the 15th Annual IEEE Symposium on Switching and Automata Theory*. IEEE, New York, pp. 33–42.

SAVAGE, C. 1977. Parallel algorithms for graph theoretic problems Ph D. dissertation, Mathematics Dept., Univ. of Illinois, Urbana, Ill.

SAVAGE, C. 1981. A systolic data structure chip for connectivity problems. In *VLSI Systems and Computations*, H. T. Kung, R. F. Sproull, and G. L. Steele, Jr., Eds. Computer Science Press, Rockville, Md.

SAVAGE, C., AND JA'JA', J. 1981. Fast, efficient parallel algorithms for some graph problems. *SIAM J Comput. 10*, 4 (Nov.), 682–690.

SAVAGE, J. E 1981. Planar circuit complexity and the performance of VLSI algorithms. In *VLSI Systems and Computations*, H T. Kung, R F. Sproull, and G. Steele, Jr., Eds. Computer Science Press, Rockville, Md., pp. 61–66.

SHILOACH, Y., AND VISHKIN, U. 1982a. An $O(\log n)$ parallel connectivity algorithm. *J Algorithms 3*, 1 (Mar.), 57–67.

SHILOACH, Y., AND VISHKIN, U. 1982b. An $O(n^2 \log n)$ parallel MAX-FLOW algorithm. *J. Algorithms 3*, 2 (June), 128–146.

SIEGEL, H. J. 1979. A model of SIMD machines and a comparison of various interconnection networks *IEEE Trans. Comput. C-28*, 12 (Dec.), 907–917.

SMITH, B. J. 1978 A pipelined shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing*. IEEE, New York, pp. 6–8.

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.

STONE, H S. 1971. Parallel processing with the perfect shuffle. *IEEE Trans Comput. C-20*, 2 (Feb.), 153–161

STONE, H. S. 1980. Parallel computers. In *Introduction to Computer Architecture*, H. S. Stone, Ed. Science Research Associates Chicago, Ill., chap. 8.

SWAN, R. J., BECHTOLSHEIM, A., LAI, K.-W., AND OUSTERHOUT, J. K. 1977. The implementation of the Cm* multi-microprocessor. In *Proceedings of the National Computer Conference* (Dallas, Tex., June 13–16), vol. 46. AFIPS Press, Reston, Va., pp. 645–655.

THOMPSON, C. D. 1979. Area-time complexity for VLSI. In *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*. ACM, New York, pp. 81–88.

THOMPSON, C. D. 1980. A complexity theory for VLSI. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

THOMPSON, C. D., AND KUNG, H. T. 1976. Sorting on a mesh connected computer. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing*. ACM, New York, pp. 58–64.

TSIN, Y. H., AND CHIN, F. Y. 1982. Efficient parallel algorithms for a class of graph theoretic problems. Tech. Rep., Dept. of Computing Science, Univ. of Alberta, Edmonton, Alberta, 45 pp. (to appear in *SIAM J Comput*).

VAN SCOY, F. L. 1976. Parallel algorithms in cellular spaces. Ph.D. dissertation, School of Engineering and Applied Science, Univ. of Virginia, Charlottesville, Va.

VISHKIN, U. 1983. Implementation of simultaneous memory access in models that forbid it. *J. Algorithms 4*, 1 (Mar.), 45–50.

VUILLEMIN, J. E. 1983. A combinatorial limit to the computing power of VLSI circuits. *IEEE Trans. Comput C-32*, 3 (Mar.), 294–300.

WARSHALL, S. 1962. A theorem on Boolean matrices. *J ACM 9*, 1 (Jan.), 11, 12.

WEIDE, B. 1977. A survey of analysis techniques for discrete algorithms. *ACM Comput. Surv. 9*, 4 (Dec.), 291–313.

WULF, W., AND HARBISON, S. P. 1978. Reflections in a pool of processors. Tech. Rep., Dept. of

Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

WYLLIE, J. C. 1979. The complexity of parallel computations. Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y.

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

YAU, S. S., AND FUNG, H. S. 1977. Associative processor architecture—A survey. *ACM Comput. Surv. 9*, 1 (Mar.), 3–27.

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

Received October 1982; final revision accepted October 1984.