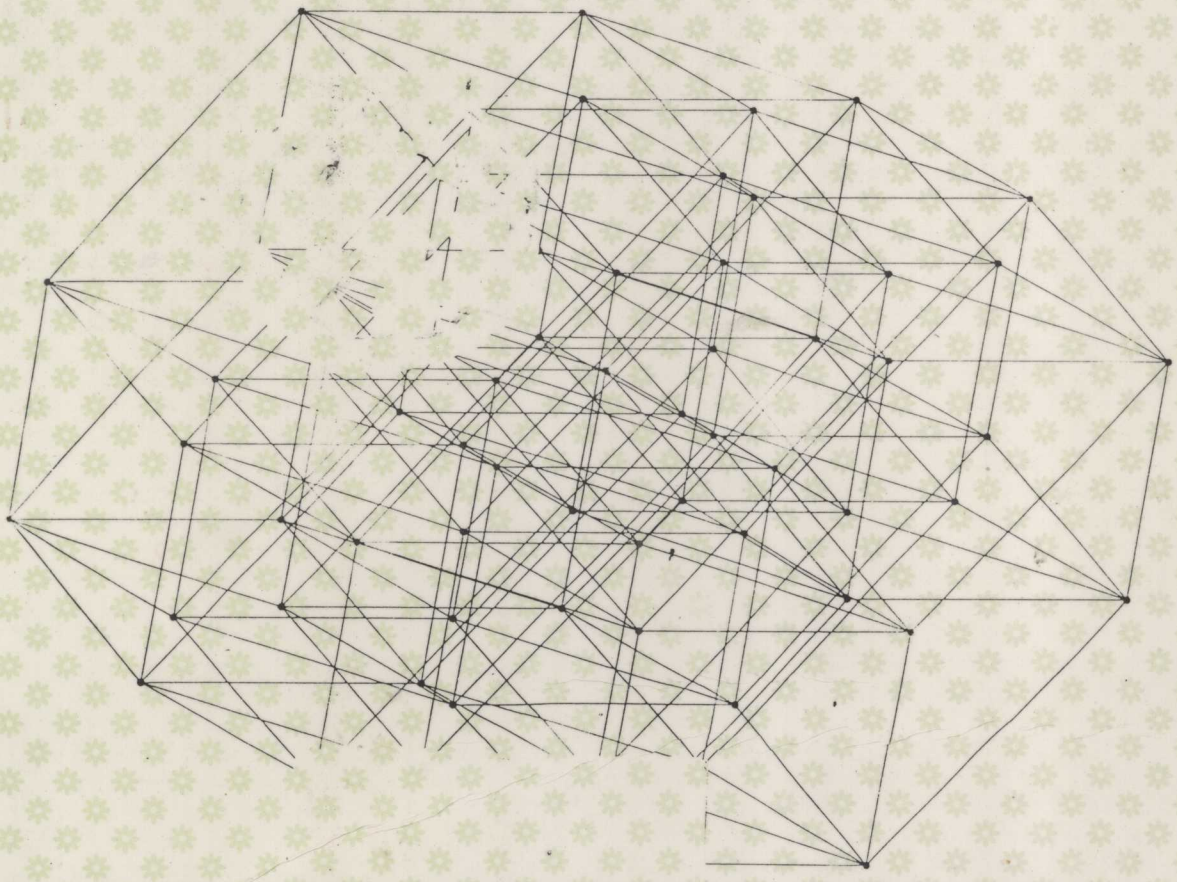
F. THOMSON LEIGHTON

INTRODUCTION TO PARALLEL ALGORITHMS AND ARCHITECTURES:

ARRAYS • TREES • HYPERCUBES

(2)



MORGAN KAUFMANN PUBLISHERS
SAN MATEO, CALIFORNIA

the techniques that were used to solve Problems 1.237, 1.238, and 1.239.)

1.241 Can any off-line permutation routing problem on a $\sqrt{N} \times \sqrt{N}$ array be solved in $2\sqrt{N} + o(\sqrt{N})$ steps using queues of size at most 4?

1.242 Show how to solve any off-line permutation routing problem on an $N_1 \times N_2 \times \cdots \times N_r$ array in

$$2(N_1 + N_2 + \cdots + N_r) - \max_{1 \le i \le r}\{N_i\} - (2r - 1)$$

steps using queues of size 1.

1.243 Show that if each node starts with one packet, and if up to m packets can have the same destination, then the basic greedy algorithm can take a maximum of $\Theta(\min(N, m\sqrt{N}))$ steps.

*1.244 Show that the result of Problem 1.243 also holds for the randomized routing algorithm of Subsection 1.7.3 in terms of expected running time.

1.245 Show that the result of Problem 1.243 also holds for the sorting-based routing algorithms described in Subsection 1.7.4.

1.246 Show that any packet-routing algorithm can be forced to use $\Omega(\sqrt{mN})$ steps for an N-packet m-to-one routing problem on a $\sqrt{N} \times \sqrt{N}$ array.

1.247 Design a constant queue size packet-routing algorithm that solves any N-packet m-to-one problem in $O(\sqrt{mN})$ steps on a $\sqrt{N} \times \sqrt{N}$ array.

.248 Show that if we first route every packet to its correct column, and then to its correct destination, then we can avoid deadlock in the wormhole model of routing.

1.249 Show that if every packet consists of b flits, then the basic greedy algorithm solves any one-to-one routing problem in $O(b\sqrt{N})$ steps on a $\sqrt{N} \times \sqrt{N}$ array.

.250 Extend Theorem 1.14 to hold for wormhole routing of packets with b flits each if the arrival rate is less than $\frac{1}{b\sqrt{N}}$. The bounds on delay and queue size can be larger by a factor of b.

251 Improve the result of Problem 1.250 to handle arrival rates up to $4/(b\sqrt{N})$.

.252 Can the result of Problem 1.250 be improved to allow constant queue size?

*1.253 Extend the result of Problem 1.250 to work for a torus.

1.254 Design a protocol for avoiding deadlock on a torus with bounded queue size.

(R*)1.255 Determine tight constant factors for the bound in Problem 1.249.

(R)1.256 Determine tight constant factors for the problem of routing b-flit packets on a ring. How well does the greedy algorithm do in the worst case?

***1.257** Assume that an N-edge directed weighted linear chain is stored in a $\sqrt{N} \times \sqrt{N}$ mesh so that each processor initially contains one weighted edge specified by $(i, S(i), W(i))$ where $(i, S(i))$ is the edge and $W(i)$ is its weight. Let $T(i)$ be the sum of the weights of the edges leading up to and including $(i, S(i))$. Show how to compute $T(i)$ for all $i < N$ in $O(\sqrt{N} \log N)$ steps. (Hint: First compute the sum of the weights on all subchains of length 2, then on all subchains of length 4, then on all subchains of length 8, and so forth. Overall, you may need to use $O(\log N)$ packet routing problems.)

****1.258** Improve the algorithm for Problem 1.257 so that it runs in $O(\sqrt{N})$ steps.

Problems based on Section 1.8.

1.259 Show that the simple 2-phase local update algorithm for labelling components of an image described in Subsection 1.8.1 runs in $O(\sqrt{N})$ steps if every component is vertically and horizontally convex. (A component is said to be *vertically convex* if the intersection of the component with each column forms a single interval. The component is *horizontally convex* if the intersection of the component with each row forms a single interval. For example, components C, D, and E of Figure 1-102 are vertically and horizontally convex, but component A is not vertically convex, and component B is not horizontally convex.)

1.260 Does the result of Problem 1.259 still hold if the components are all horizontally convex or vertically convex, but not both?

1.261 Show how to modify Levialdi's algorithm to label components in the model where diagonally adjacent pixels are not considered to be contiguous. (Hint: Represent each pixel with a 3×3 array of pixels in such a way that the original algorithm can be applied.)

1.262 Show how to modify Levialdi's algorithm to count the number of connected components in $O(\sqrt{N})$ word steps using only $O(\log N)$ bits of memory per processor.

*1.263 Improve the result of Problem 1.262 to run in $O(\sqrt{N})$ bit steps. (How much memory do you need per processor?)

*1.264 Show how to modify Levialdi's algorithm so that only $O(\log N)$ bits of memory are needed per processor, and so that the algorithm still runs in $O(\sqrt{N}\log N)$ bit steps. (Hint: Remember the history of the shrinking phase at steps $\sqrt{N}$, $\frac{3}{2}\sqrt{N}$, $\frac{7}{4}\sqrt{N}$, $\frac{15}{8}\sqrt{N}$, ..., $2\sqrt{N}$. In order to perform the expansion phase within one of the intervals $\left[(2-2^{-i})\sqrt{N},(2-2^{-i-1})\sqrt{N}\right]$, rerun the algorithm recursively within the interval.)

*1.265 Determine a good constant c for which the recursive component-labelling algorithm described in Subsection 1.8.1 runs in $c\sqrt{N}$ word steps. (Hint: You don't really need to use the general routing and sorting algorithms described in Sections 1.6 and 1.7 for this algorithm.)

*1.266 Show how to modify the recursive component-labelling algorithm so that it runs in $O(\sqrt{N})$ bit steps. (Hint: See the hint for Problem 1.265.)

1.267 Show that at most $\sqrt{N}\sin\theta+\sqrt{N}\cos\theta+1$ unit-width bands intersect a $\sqrt{N}\times\sqrt{N}$ pixel array when computing a Hough transform at angle θ.

1.268 Show how to compute a Hough transform for angles larger than $\frac{\pi}{4}$.

(R)1.269 Is the $O(\sqrt{N})$-step algorithm for computing a Hough transform for $\sqrt{N}$ angles of a $\sqrt{N}\times\sqrt{N}$ image work efficient up to constant factors?

1.270 Given a $\sqrt{N}\times\sqrt{N}$ image, show how to compute the farthest object from every pixel in $O(\sqrt{N})$ steps on a $\sqrt{N}\times\sqrt{N}$ array.

(R)1.271 Given a collection of N points specified by coordinate pairs, show how to compute all nearest neighbors in $O(\sqrt{N})$ steps on a $\sqrt{N}\times\sqrt{N}$ array. (Hint: Use sorting and divide-and-conquer.)

1.272 Given a collection of N points $p_1=(x_1,y_1)$, ..., $p_N=(x_N,y_N)$ for which $x_1\le x_2\le\cdots\le x_N$, show that if p_i is the kth point in the upper hull, then p_j is the $(k+1)$st point in the upper hull if $j>1$ is the smallest value for which θ_{ij} is maximized. (As in Subsection 1.8.4, θ_{ij} denotes the angle of the line from p_i to p_j with respect to the negative vertical axis.)

***1.273** Show how to compute the convex hull of N arbitrary points in $O(\sqrt{N})$ steps on a $\sqrt{N}\times\sqrt{N}$ array. (Hint: Use sorting and

divide-and-conquer.)

*1.274 Show how to compute the convex hull of every connected component of a $\sqrt{N} \times \sqrt{N}$ image in $O(\sqrt{N})$ steps on a $\sqrt{N} \times \sqrt{N}$ array of processors. (Hint: Use the result of Problem 1.273.)

Problems based on Section 1.9.

1.275 Show that an $N_1 \times N_2 \times \cdots \times N_r$ array has a bisection of size $N_1 N_2 \cdots N_r / \max_{1 \le i \le r}\{N_i\}$ if $\max_{1 \le i \le r}\{N_i\}$ is even.

*1.276 Show that any bisection of an $N_1 \times N_2 \times \cdots \times N_r$ array contains at least $N_1 N_2 \cdots N_r / \max_{1 \le i \le r}\{N_i\}$ edges.

(R)1.277 What is the bisection width of an $N_1 \times N_2 \times \cdots \times N_r$ array when $\max_{1 \le i \le r}\{N_i\}$ is odd?

1.278 Show that the bisection width of an r-dimensional N-sided array is close to but larger than N^{r-1} if N is odd.

(R)1.279 What is the bisection width of an r-dimensional N-sided array when N is odd?

***1.280** Show that the bisection width of an r-dimensional N-sided torus is $2N^{r-1}$ if N is even.

(R)1.281 What is the bisection width of an r-dimensional N-sided torus if N is odd?

1.282 Show that an r-dimensional N-sided torus can be simulated by an r-dimensional N-sided array with a slowdown factor of 2.

*1.283 How long does it take to multiply $N \times N$ matrices on an r-dimensional array? How many processors are needed?

1.284 Show how to multiply two $N \times N$ matrices in $2 \log N$ steps on an N^3-node hypercube.

1.285 Pipeline the algorithm from Problem 1.284 to multiply $\log N$ pairs of matrices in $O(\log N)$ steps.

1.286 Show how to multiply two $N \times N$ matrices in $O(\log N)$ steps on an $\frac{N^3}{\log N}$-node hypercube.

(R)1.287 Can a nonsingular $N \times N$ matrix be inverted in $O(N^{3/4})$ steps on a three-dimensional $N^{3/4}$-sided array?

*1.288 Show that the three-dimensional array sorting algorithm described in Subsection 1.9.3 is similar to the Columnsort algorithm of Problem 1.187 in which $r = N^{2/3}$ and $s = N^{1/3}$.

*1.289 Show how to sort N items in $O(N^{1/r})$ steps on an r-dimensional $N^{1/r}$-sided array for any constant $r > 3$.

1.290 Show that any algorithm for sorting N items into zyx-order on a three-dimensional $N^{1/3}$-sided array must take at least $5N^{1/3} - o(N^{1/3})$ steps.

1.291 Generalize the lower bound in Problem 1.290 for r-dimensional arrays.

*1.292 Describe an algorithm for sorting N items on a three-dimensional $N^{1/3}$-sided array that takes at most $5N^{1/3} + o(N^{1/3})$ steps.

**1.293 Generalize the upper bound in Problem 1.292 for r-dimensional arrays.

1.294 Consider a three-dimensional array where the items in each xz-plane are sorted into zx-order, and then the items in each yz-plane are sorted into zy-order. Show that every item is within one of the correct xy-plane for an overall zyx-order.

1.295 Show how to solve any many-to-one routing problem in $O(N^{1/3})$ steps on an $N^{1/3} \times N^{1/3} \times N^{1/3}$ array if combining is allowed.

1.296 Show that the greedy algorithm (i.e., Phases 2–4 of the algorithm in Subsection 1.9.4) can take $\Omega(N^{2/3})$ steps to solve a worst-case one-to-one routing problem on an $N^{1/3} \times N^{1/3} \times N^{1/3}$ array if the packets are not rearranged ahead of time.

1.297 Show that the greedy algorithm never uses more than $O(N^{2/3})$ steps for a one-to-one routing problem on an $N^{1/3} \times N^{1/3} \times N^{1/3}$ array.

1.298 How do the results of Problems 1.296 and 1.297 generalize for r-dimensional $N^{1/r}$-sided arrays? (Be sure to consider the parity of r.)

1.299 Generalize the routing algorithm described in Subsection 1.9.4 to run in $T + r(N^{1/r} - 1)$ steps on an r-dimensional $N^{1/r}$-sided array, where T is the time needed to sort on an r-dimensional $N^{1/r}$-sided array.

***1.300** Show that the time needed to route on any network is never more than a constant factor times the time needed to sort on the network. (Hint: Find a way of generating dummy packets for destinations that do not receive an ordinary packet, and then route the packets by sorting.)

**1.301 Generalize the average case analysis of Subsection 1.7.2 to work for three-dimensional arrays.

*1.302 Design a randomized algorithm for routing on an $N^{1/3} \times N^{1/3} \times N^{1/3}$ array that does not require sorting.

(R)1.303 Is there a deterministic $(3N^{1/3} + o(N^{1/3}))$-step algorithm for routing on an $N^{1/3} \times N^{1/3} \times N^{1/3}$ array that uses $O(1)$-size queues?

1.304 Give two good reasons why an N-node two-dimensional array cannot simulate an N-node three-dimensional array with constant slowdown.

1.305 Given a problem of size M that can be solved in T steps on an M-node, s-dimensional $M^{1/s}$-sided array, how large must M be (in terms of N) for us to be able to solve the problem on an N-node, r-dimensional $N^{1/r}$-sided array in $O(TM/N)$ steps? (You may assume that the simulation of high-dimensional arrays on low-dimensional arrays described in Subsection 1.9.5 is the best one can hope for.)

1.306 Show how to simulate an $N2^{N-1}$-node butterfly on an N-node linear array with slowdown 2^N. (Hint: Look ahead to Chapter 3 for the definition of a butterfly.)

1.307 Given a problem of size M that can be solved in T steps on an M-node butterfly, how large must M be (in terms of N) in order for us to be able to solve the problem in $O(TM/N)$ steps on an N-cell linear array? (You may assume that the simulation result described in Problem 1.306 is the best possible.)

***1.308** Show how to simulate a $\sqrt{N}2^{\sqrt{N}-1}$-node butterfly on an $\sqrt{N} \times \sqrt{N}$ array with $O(2^{\sqrt{N}}/\sqrt{N})$ slowdown. (Hint: You may need to use the packet routing results from Section 1.7.)

1.309 How large does the size of the problem described in Problem 1.307 need to be in order to solve the problem in $O(TM/N)$ steps on a $\sqrt{N} \times \sqrt{N}$ array? (You may assume that the simulation described in Problem 1.308 is optimal.)

*1.310 Extend the result of Problem 1.308 to higher-dimensional arrays.

1.311 Extend the result of Problem 1.309 to higher-dimensional arrays.

1.11 Bibliographic Notes

There is a very large body of literature on the subject of array and tree algorithms for parallel computation. We will not attempt to cite all (or even a large portion) of this literature in this text. Rather, we will be content to provide pointers to some of the most relevant and useful references on this subject matter and to sources that were particularly helpful in the preparation of the text.

1.1

The simple sorting algorithm for linear arrays described in Section 1.1 is sometimes referred to as a "zero-time sorter," although it uses substantially more than zero time to sort N numbers. Miranker, Tang, and Wong describe a variation of this algorithm in [176], as do Armstrong and Rem in [14]. The use of bisection width to prove lower bounds on the running time of an algorithm is due to Thomborson (a.k.a. Thompson) [242, 243]. Lower bound arguments such as those briefly mentioned in Section 1.1 will be discussed in much greater detail in Volume II. Problem 1.9 was contributed by Atallah. Related results involving a hybrid model of parallel computation in which a parallel network of processors (each with a bounded memory) is connected to a sequential front-end processor (with unlimited memory) can be found in the work of Atallah and Tsay [20]. Problem 1.10 was suggested by Thomborson. Additional results on selection and median finding using a tree network (see Problem 1.19) can be found in the work of Frederickson [76].

1.2

The carry-lookahead addition algorithm of Subsection 1.2.1 is very similar to algorithms described by Winograd [262], Brent [38], and Brent and Kung [39], among others. The parallel prefix algorithm described in Subsection 1.2.2 is contained in the work of Ladner and Fischer [138] and Brent and Kung [39]. The carry-save addition algorithm described in Subsection 1.2.3 is implicit in the work of Wallace [259] and Dadda [61]. The integer multiplication algorithm of Subsection 1.2.4 is similar to many algorithms described in the literature. (See the article by Wu [263] for a summary and survey of this literature.) In addition, the result of Problem 1.130 is due to Atrubin [21] and Muller [178]. Related algorithms for convolution and integer multiplication are described in Section 1.4. Linear

array algorithms for division that are not based on Newton iteration are described by Brickell in [41]. The greatest common divisor algorithm described in Problem 1.50 is due to Brent and Kung [40]. Problem 1.32 was contributed by Leiserson.

1.3

The odd-even reduction algorithm for solving tridiagonal systems of equations in Subsection 1.3.3 is similar to algorithms described by Golub and Hockney [100, 101] and Ericksen [72]. The prefix-based algorithm for computing an LU-factorization of a tridiagonal matrix described in Subsection 1.3.3 is due to Stone [236]. For more information and references on numerical methods and parallel linear algebra, we refer the reader to the survey papers by Ortega and Voigt [189] and Gallivan, Plemmons, and Sameh [80], and the texts by Bertsekas and Tsitsiklis [28], Golub and Van Loan [83], Strang [240], and Kung [137]. Problems 1.54 and 1.59 are solved by Culik and Yu in [55].

1.4

Most of the material on retiming in Section 1.4 (including several of the exercises) was derived from the work Seiferas [223] and Leiserson and Saxe [153]. Further material on this subject can be found in the work of Culik and Fris [54], Leiserson and Saxe [154], and Even and Litman [73]. Much of the early work on systolic computation appears in the automata literature (e.g., see the text edited by Shannon and McCarthy [226] and the work of Kosaraju [122]). Problem 1.131 was solved by Smith in [231]. The multiplication algorithm described in Problem 1.130 is due to Atrubin [21] and Muller [178]. In addition, the material covered in Problems 1.129–1.132 is explored in much greater detail and generality by Even and Litman in [73]. A history and efficient solution of the Firing Squad Problem (see Problems 1.134–1.137) is given by Culik and Dube in [53].

1.5

Much of the material and many of the exercises from Section 1.5 are due to Christopher [50], Guibas, Kung and Thomborson [88], and Atallah and Kosaraju [19]. The algorithm for minimum-weight spanning trees in Subsection 1.5.5 is due to Maggs and Plotkin [164]. Some interesting algorithms for tree problems and data structures are described by Atallah and

Hambrusch in [18]. References to other work on graph algorithms can be found in these sources.

1.6

The 0–1 Sorting Lemma is due to Knuth [117]. The odd-even transposition sort algorithm described in Subsection 1.6.1 was analyzed by Haberman in [89]. The Shearsort algorithm described in Subsection 1.6 was discovered by Sado and Igarishi [215] and Scherson, Sen and Shamir [216]. The faster sorting algorithm described in Subsection 1.6.3 is due to Schnorr and Shamir [217]. Many $O(\sqrt{N})$-step algorithms for sorting on an array have been discovered, beginning with the algorithm of Thomborson and Kung [244]. Some of these algorithms are based on the early work of Batcher [23]. The lower bound of Subsection 1.6.4 was discovered by many, including Schnorr and Shamir [217] and Kunde [130].

The Revsort algorithm described in Problem 1.164 is due to Schnorr and Shamir [217]. A solution to Problem 1.174 is described by Chlebus in [48]. The solutions to Problems 1.183 and 1.184 are due to Mansour and Schulman [166]. The Columnsort algorithm described in Problem 1.187 is due to Leighton [144]. Lower bounds for sorting into arbitrary orders (as in Problem 1.179) are described by Han and Igarashi in [91] and Kunde in [132]. Toroidal sorting algorithms are described by Kunde in [133]. Very recent work on array sorting algorithms is reported by Kunde in [134] and Kaklamanis, Krizanc, Narayanan, and Tsantilas in [109]. References to other work on sorting algorithms for arrays and tori can be found in these sources as well as the survey paper by Chlebus and Kukawka [49].

1.7

The material in Subsection 1.7.2 is due to Leighton [145]. Related material on randomized algorithms for packet routing is contained in the work of Valiant and Brebner [253] and Krizanc, Rajasekaran, and Tsantilas [124]. For a review of the probabilistic and analytical methods used in Subsections 1.7.2–1.7.3 (including Stirling's formula), we refer the reader to the text by Graham, Knuth, and Patashnik [84].

The use of sorting as a preconditioner to routing is described by Kunde in [133, 135]. A $(2\sqrt{N}-2)$-step algorithm for routing with constant queue size is described by Leighton, Makedon, and Tollis in [150]. The constant in the queue size of this algorithm was recently improved by Rajasekaran and Overholt in [208]. Algorithms for routing more than N packets on

an N-node array are described by Kunde and Tensi in [136] and Simvonis in [230]. Recently discovered algorithms for sorting and selection on arrays are reported by Kaklamanis, Krizanc, Narayanan, and Tsantilis in [109].

The off-line algorithm described in Subsection 1.7.5 is generalized to a wide variety of networks by Annexstein and Baumslag in [12]. Theorem 1.17 is due to Hall [90]. Problems 1.239 and 1.240 were contributed by Krizanc. Algorithms for bit-serial, cut-through, and wormhole routing on arrays are contained in work by Kermani and Kleinrock [115], Dally and Seitz [64], Dally [63], Flaig [74], Kunde [134], Ngai [187], Ngai and Seitz [188], Makedon and Simvonis [165], and Borkar, Cohn, Cox, Gross, Kung, Lam, Levine, Wire, Peterson, Susman, Sutton, Urbanski, and Webb [36]. Problems 1.257 and 1.258 are solved by Atallah and Hambrusch in [18].

1.8

The region-labelling algorithms of Subsection 1.8.1 can be found in the work of Levialdi [156], Nassimi and Sahni [182], and Cypher, Sanz, and Snyder [58]. Optimal algorithms for computing Hough transforms can be found in the work of Cypher, Sanz, and Snyder [59] and Guerra and Hambrusch [87]. Algorithms for finding convex hulls and for solving many related problems in image processing and computational geometry on arrays are contained in the work of Miller and Stout [171, 175], Jeong and Lee [107], and Lu and Varman [158]. References to additional material can also be found in these papers. The text by Kung [137] also describes a variety of algorithms for image processing on arrays.

1.9

The proof technique of Theorem 1.21 is due to Leighton [141, 142], although other proofs of this result are well known in the literature. The sorting algorithm in Subsection 1.9.3 is due to Kunde [129]. Improved bounds and algorithms for sorting in multidimensional arrays (e.g., see Problems 1.290–1.293) are described by Kunde in [130, 131, 133]. Additional material on routing in multidimensional arrays can be found in the work of Kunde [134] and Kunde and Tensi [136]. Extensions of the material described in Subsection 1.9.5 can be found in the work of Atallah [17], Kosaraju and Atallah [123], and Koch, Leighton, Maggs, Rao, and Rosenberg [121]. Interesting variations of multidimensional arrays are described by Dally in [62] and Draper in [69].

Miscellaneous

Many array algorithms appear in the literature under the heading of cellular arrays or cellular automata. Readers interested specifically in the subject of cellular automata are referred to the text by Toffoli and Margolus [245]. There has also been a fair amount of work on techniques for automatically mapping algorithms with a certain structure onto arrays. Readers interested in this subject are referred to the papers by Chen [47] and Kumar and Tsai [128]. References to additional material on this subject can be found in these sources.

Algorithms for the pyramid network are described by Miller and Stout in [172]. References to related work can also be found in this source. Algorithms for arrays with busses can be found in the work of Stout [239]. For more information on signal processing algorithms on arrays, we refer the reader to the text by Kung [137]. For more information on algorithms where inputs are provided more than once, we refer the reader to the paper by Duris and Galil [70] and the references contained therein.

CHAPTER 2

MESHES OF TREES

Although arrays and trees are relatively simple to build and are quite efficient for some algorithms, they suffer from two major drawbacks: large diameter and/or small bisection width. As a result, the speed with which they can be used to solve many problems is highly limited.

In this chapter, we consider a hybrid network architecture based on arrays and trees called the *mesh of trees*. Meshes of trees have both small diameter and large bisection width, and are the fastest networks known when considered solely in terms of speed. In fact, every problem discussed in Chapter 1 can be solved in $\Theta(\log N)$ or $\Theta(\log^2 N)$ steps on a suitably large mesh of trees. This is dramatically faster than the typical running times of $\Theta(\sqrt{N})$ or $\Theta(N)$ for algorithms on arrays and trees.

Running times that are bounded by a constant power of $\log N$ such as $\Theta(\log N)$ or $\Theta(\log^2 N)$ are said to be *polylogarithmic*. Algorithms that run in polylogarithmic time on a network with a polynomial (e.g., N^2 or N^3) number of processors form the class NC. For example, the parallel prefix algorithm described in Chapter 1 is in NC, but most of the other algorithms are not. Henceforth, we will be primarily concerned with networks that are capable of supporting NC algorithms.

The tremendous advantage of meshes of trees over arrays is that the dramatic speedups in time can often be accomplished without increasing the number of processors. For example, we will need only $\Theta(N^2)$ processors to compute the connected components of an N-node graph in $\Theta(\log^2 N)$ steps using a mesh of trees, whereas the same problem requires $\Theta(N)$ steps

on an $N \times N$ array. Unfortunately, such economies are not possible for all problems. For example, sorting N numbers in $\Theta(\log N)$ steps requires $\Theta(N^2)$ processors on a mesh of trees, instead of the N processors required to sort in $\Theta(\sqrt{N})$ steps on an array. For such problems, we will have to wait until Chapter 3 before finding a network that is both fast *and* small.

Even for problems such as sorting, however, where the mesh of trees is not processor efficient, we will later find that the mesh of trees is *area efficient.* In particular, we will show in Volume II that the mesh of trees is *area universal* (i.e., that it can simulate any other network with the same VLSI wire area with only a polylogarithmic factor slowdown). We will also show in Chapter 3 that mesh of trees algorithms can be run on hypercubic networks without slowing the algorithm down or increasing the number of processors. Hence, the mesh of trees is a very important network, and worthy of considerable attention.

We start our discussion of meshes of trees by defining the two-dimensional mesh of trees in Section 2.1. The computational power of the mesh of trees will become immediately apparent in Section 2.2 when we describe elementary $O(\log N)$-step algorithms for a wide variety of problems on the $N \times N$ mesh of trees. Included are algorithms for packet routing, sorting, matrix-vector multiplication, Jacobi relaxation, pivoting, convolution, and convex hull. We continue with some more sophisticated $O(\log N)$-step algorithms for integer multiplication, powering, division, and root finding in Section 2.3.

The three-dimensional mesh of trees is defined in Section 2.4. Like the two-dimensional mesh of trees, the three-dimensional mesh of trees can be used to solve many problems quickly, but is most naturally suited to problems involving matrix multiplication (which takes just $2 \log N$ steps on an $N \times N \times N$ mesh of trees). Several such problems are discussed in Section 2.4, including matrix inversion, decomposition, and powering.

Meshes of trees are also particularly well suited for graph problems. For example, in Section 2.5, we describe algorithms for finding the minimum-weight spanning tree, connected components, transitive closure, all pairs shortest paths, and maximum matching of an N-node graph in $O(\log^2 N)$ steps on a mesh of trees. The algorithms for finding a minimum-weight spanning tree and the connected components of a graph run on an $N \times N$ mesh of trees, and make use of a powerful doubling up technique known as *pointer jumping.* The algorithms have many applications, and can be applied to solve several of the image-processing problems discussed in Sec-

tion 1.8 using only $O(\log^2 N)$ steps on a $\sqrt{N} \times \sqrt{N}$ mesh of trees.

The algorithms for transitive closure, shortest paths, and maximum matching run on an $N \times N \times N$ mesh of trees, and are closely related to the matrix multiplication and inversion algorithms of Section 2.4. The maximum matching algorithm, in particular, inverts a randomly weighted adjacency matrix of a graph to find good matchings in the graph. Although the algorithm is not deterministic, it will find the maximum matching in any N-node graph in $O(\log^2 N)$ steps with probability very close to one. This result provides our first example of a problem that is in RNC (i.e., it can be solved by a randomized NC algorithm), but for which there is no known NC solution.

In Section 2.6, we describe a general procedure for evaluating straight-line arithmetic code in parallel. The material in this section is particularly interesting because it is as close as we will ever get to "automatic" parallelization of sequential algorithms. In particular, we will show how to automatically parallelize any N-step straight-line (i.e., nonbranching) arithmetic code so that it runs in $O(\log dN \log N)$ steps, where d is a parameter that reflects the complexity of the code. In the worst case, d can be exponential in N, and the parallel running time won't be very good. For many important problems, however, d is polynomial in N, and the resulting parallel code will run in $O(\log^2 N)$ steps. Using this process, we can automatically derive $O(\log^2 N)$-step parallel algorithms for such difficult problems as matrix inversion and computing determinants of matrices with multivariable polynomial entries. Unfortunately, the parallel algorithms derived by this procedure are not processor efficient.

We describe higher-dimensional meshes of trees and some related networks in Section 2.7. The material in this section provides a good background for Chapter 3, where we define the shuffle-exchange graph and discuss its relationship with the hypercube.

We conclude the chapter with several exercises and bibliographic notes in Sections 2.8 and 2.9.

2.1 The Two-Dimensional Mesh of Trees

In this section, we define the two-dimensional mesh of trees and discuss its properties. The section is divided into five subsections. Each of the first four subsections presents a different way to define and/or think about the network. The various definitions of the mesh of trees will prove to be useful later in the chapter when we will describe how to implement algorithms on the network.

We conclude in Subsection 2.1.5 by comparing the structure and computational power of the mesh of trees to the pyramid and multigrid. Although the mesh of trees appears to be similar to the pyramid and multigrid in many respects, we will find that the mesh of trees is a substantially more powerful interconnection network.

2.1.1 Definition and Properties

The $N \times N$ mesh of trees is constructed from an $N \times N$ grid of processors by adding processors and wires to form a complete binary tree in each row and each column. The leaves of the trees are precisely the original N^2 nodes of the grid, and the added nodes are precisely the internal nodes of the trees. Overall, the network has $3N^2 - 2N$ processors. The leaf and root processors have degree 2, and all other processors have degree 3. For example, see Figure 2-1.

It is not difficult to check that the $N \times N$ mesh of trees has diameter $4 \log N$. For example, to construct a path of length at most $4 \log N$ from any node u in the ith row tree to any node v in the jth column tree, we first construct the path of length at most $2 \log N$ from u to z in the ith row tree where z is the unique leaf shared by the ith row tree and the jth column tree, and then finish with the path of length at most $2 \log N$ from z to v in the jth column tree. In order to construct a path of length at most $4 \log N$ from any node u in the rth level of the ith row tree to any node v in the sth level of the jth row tree where $r \geq s$ (without loss of generality), we start with the path of length $\log N - r$ from u to one of its descendant leaves in the ith row tree, continue with the path of length at most $2 \log N$ from this leaf to a leaf in the jth row tree, and finish with the path of length at most $\log N + s$ from that leaf to v in the jth row tree. Since $s \leq r$, the total path length is at most $4 \log N$. A symmetric argument reveals that the distance between any two column tree nodes is also at most $4 \log N$. For example, see Figure 2-2.

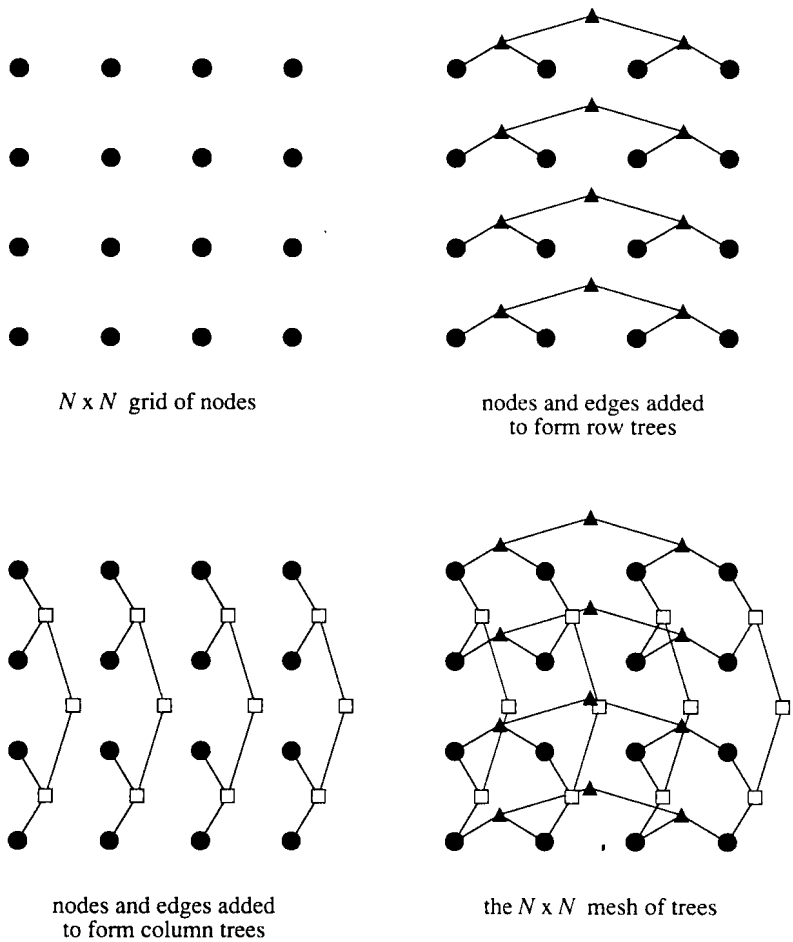


Figure 2-1 *The two-dimensional mesh of trees. Leaf nodes from the original grid are denoted with circles. Nodes added to form column trees are denoted with squares, and nodes added to form row trees are denoted with triangles.*