Introduction to Parallel Computing

Solution Manual

Ananth Grama
Anshul Gupta
George Karypis
Vipin Kumar

Contents

Preface

This instructors guide to accompany the text ”Introduction to Parallel Computing” contains solutions to selected problems.

For some problems the solution has been sketched, and the details have been left out. When solutions to problems are available directly in publications, references have been provided. Where necessary, the solutions are supplemented by figures. Figure and equation numbers are represented in roman numerals to differentiate them from the figures and equations in the text.

CHAPTER 1

Introduction

1 At the time of compilation (11/02), the five most powerful computers on the Top 500 list along with their peak GFLOP ratings are:

1. NEC Earth-Simulator/ 5120, 40960.00.
2. IBM ASCI White, SP Power3 375 MHz/8192 12288.00.
3. Linux NetworX MCR Linux Cluster Xeon 2.4 GHz -Quadrics/ 2304, 11060.00.
4. Hewlett-Packard ASCI Q - AlphaServer SC ES45/1.25 GHz/ 4096, 10240.00.
5. Hewlett-Packard ASCI Q - AlphaServer SC ES45/1.25 GHz/ 4096 10240.00.

2 Among many interesting applications, here are a representative few:

1. Structural mechanics: crash testing of automobiles, simulation of structural response of buildings and bridges to earthquakes and explosions, response of nanoscale cantilevers to very small electromagnetic fields.
2. Computational biology: structure of biomolecules (protein folding, molecular docking), sequence matching for similarity searching in biological databases, simulation of biological phenomena (vascular flows, impulse propagation in nerve tissue, etc).
3. Commercial applications: transaction processing, data mining, scalable web and database servers.

3 Data too fluid to plot.

4 Data too fluid to plot.

CHAPTER 2

Models of Parallel Computers

1 A good approximation to the bandwidth can be obtained from a loop that adds a large array of integers:

```
for (i = 0; i < 1000000; i++)
  sum += a[i];
```

with `sum` and array `a` suitably initialized. The time for this loop along with the size of an integer can be used to compute bandwidth (note that this computation is largely memory bound and the time for addition can be largely ignored).
To estimate L1 cache size, write a 3-loop matrix multiplication program. Plot the computation rate of this program as a function of matrix size n. From this plot, determine sudden drops in performance. The size at which these drops occur, combined with the data size ($2n^2$) and word size can be used to estimate L1 cache size.

2 The computation performs 8 FLOPS on 2 cache lines, i.e., 8 FLOPS in 200 ns. This corresponds to a computation rate of 40 MFLOPS.

3 In the best case, the vector gets cached. In this case, 8 FLOPS can be performed on 1 cache line (for the matrix). This corresponds to a peak computation rate of 80 MFLOPS (note that the matrix does not fit in the cache).

4 In this case, 8 FLOPS can be performed on 5 cache lines (one for matrix `a` and four for column-major access to matrix `b`). This corresponds to a speed of 16 MFLOPS.

5 For sample codes, see any SGEMM/DGEMM BLAS library source code.

6 Mean access time = $0.8 \times 1 + 0.1 \times 100 + 0.8 \times 400 \approx 50$ns. This corresponds to a computation rate of 20 MFLOPS (assuming 1 FLOP/word).
Mean access time for serial computation = $0.7 \times 1 + 0.3 \times 100 \approx 30$ns. This corresponds to a computation rate of 33 MFLOPS.
Fractional CPU rate = $20/33 \approx 0.60$.

7 Solution in text.

8 Scaling the switch while maintaining throughput is major challenge. The complexity of the switch is $O(p^2)$.

9 CRCW PRAM is the most powerful because it can emulate other models without any performance overhead. The reverse is not true.

10 We illustrate the equivalence of a butterfly and an omega network for an 8-input network by rearranging the switches of an omega network so that it looks like a butterfly network This is shown in Figure 2.1 [Lei92a].

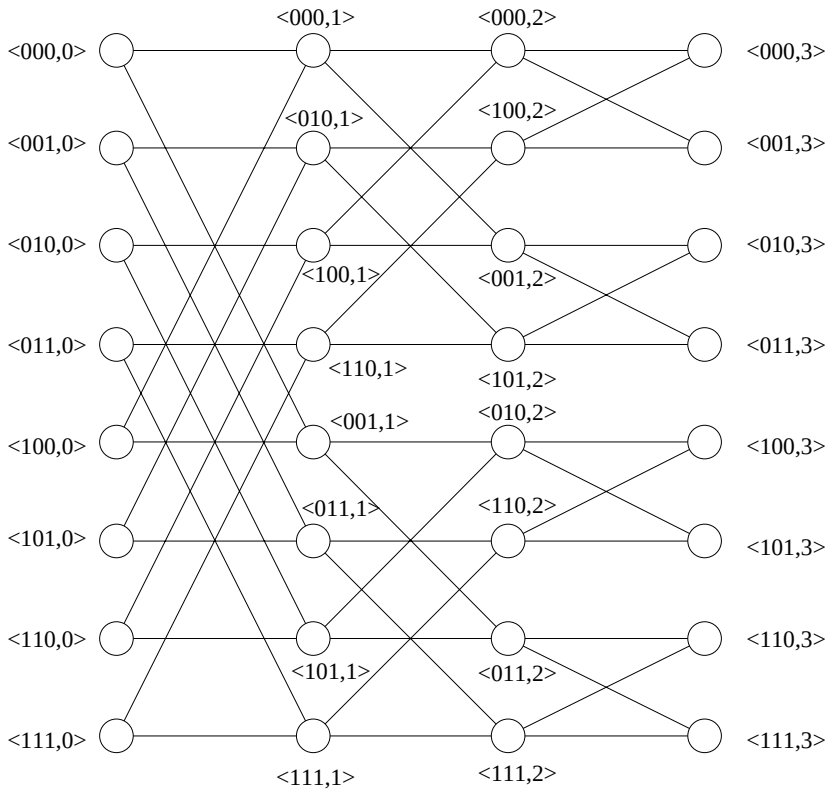

Figure 2.1 An 8-input omega network redrawn to look like a butterfly network. Node $\langle i, l \rangle$ (node i at level l) is identical to node $\langle j, l \rangle$ in the butterfly network, where j is obtained by right circular shifting the binary representation of i l times.

12 Consider a cycle $A_1, A_2, \ldots, A_k$ in a hypercube. As we travel from node A_i to A_{i+1}, the number of ones in the processor label (that is, the parity) must change. Since $A_1 = A_k$, the number of parity changes must be even. Therefore, there can be no cycles of odd length in a hypercube.
(Proof adapted from Saad and Shultz [SS88]).

13 Consider a 2^d processor hypercube. By fixing k of the d bits in the processor label, we can change the remaining $d - k$ bits. There are 2^{d-k} distinct processors that have identical values at the remaining k bit positions. A p-processor hypercube has the property that every processor has $\log p$ communication links, one each to a processor whose label differs in one bit position. To prove that the 2^{d-k} processors are connected in a hypercube topology, we need to prove that each processor in a group has $d - k$ communication links going to other processors in the same group.
Since the selected d bits are fixed for each processor in the group, no communication link corresponding to these bit positions exists between processors within a group. Furthermore, since all possible combinations of the $d - k$ bits are allowed for any processor, all $d - k$ processors that differ along any of these bit positions are also in the same group. Since the processor will be connected to each of these processors, each processor within a group is connected to $d - k$ other processors. Therefore, the processors in the group are connected in a hypercube topology.

14 Refer to Saad and Shultz [SS88].

15 **NOTE**

The number of links across the two subcubes of a d-dimensional hypercube is 2^{d-1} and not $2^d - 1$.
The proposition can be proved by starting with a partition in which both halves form subcubes. By construction, there are $p/2 (= 2^{d-1})$ communication links across the partition. Now, by moving a single processor

from one partition to the other, we eliminate one communication link across the boundary. However, this processor is connected to $d-1$ processors in the original subcube. Therefore, an additional $d-1$ links are added. In the next step, one of these $d-1$ processors is moved to the second partition. However, this processor is connected to $d-2$ processors other than the original processors. In this way, moving processors across the boundary, we can see that the minima resulting from any perturbation is one in which the two partitions are subcubes. Therefore, the minimum number of communication links across any two halves of a d-dimensional hypercube is 2^{d-1}.

16 Partitioning the mesh into two equal parts of $p/2$ processors each would leave at least $\sqrt{p}$ communication links between the partitions. Therefore, the bisection width is $\sqrt{p}$. By configuring the mesh appropriately, the distance between any two processors can be made to be independent of the number of processors. Therefore, the diameter of the network is $O(1)$. (This can however be debated because reconfiguring the network in a particular manner might leave other processors that may be more than one communication link away from each other. However, for several communication operations, the network can be configured so that the communication time is independent of the number of processors.) Each processor has a reconfigurable set of switches associated with it. From Figure 2.35 (page 80), we see that each processor has six switches. Therefore, the total number of switching elements is $6p$. The number of communication links is identical to that of a regular two-dimensional mesh, and is given by $2(p-\sqrt{p})$.

The basic advantage of the reconfigurable mesh results from the fact that any pair of processors can communicate with each other in constant time (independent of the number of processors). Because of this, many communication operations can be performed much faster on a reconfigurable mesh (as compared to its regular counterpart). However, the number of switches in a reconfigurable mesh is larger.

17 Partitioning the mesh into two equal parts of $p/2$ processors each would leave at least $\sqrt{p}$ communication links between the partitions. Therefore, the bisection width of a mesh of trees is $\sqrt{p}$. The processors at the two extremities of the mesh of trees require the largest number of communication links to communicate. This is given by $2\log(\sqrt{p}) + 2\log(\sqrt{p})$, or $2\log p$. A complete binary tree is imposed on each row and each column of the mesh of trees. There are $2\sqrt{p}$ such rows and columns. Each such tree has $\sqrt{p}-1$ switches. Therefore, the total number of switches is given by $2\sqrt{p}(\sqrt{p}-1)$, or $2(p-\sqrt{p})$.

Leighton [Lei92a] discusses this architecture and its properties in detail.

18 In the d-dimensional mesh of trees, each dimension has $p^{1/d}$ processors. The processor labels can be expressed in the form of a d-tuple. The minimum number of communication links across a partition are obtained when the coordinate along one of the dimensions is fixed. This would result in $p^{(d-1)/d}$ communication links. Therefore, the bisection width is $p^{(d-1)/d}$.

Connecting $p^{1/d}$ processors into a complete binary tree requires $p^{1/d}-1$ switching elements. There are $p^{(d-1)/d}$ distinct ways of fixing any one dimension and there are d dimensions. Therefore, the total number of switching elements is given by $dp^{(d-1)/d}(p^{1/d}-1)$, or $d(p-p^{(d-1)/d})$.

Similarly, the number of communication links required to connect processors along any one dimension is given by $2(p^{1/d}-1)$. Using a procedure similar to the one above, we can show that the total number of communication links is given by $dp^{(d-1)/d}2(p^{1/d}-1)$, or $2d(p-p^{(d-1)/d})$.

The diameter of the network can be derived by traversing along each dimension. There are d dimensions and traversing each dimension requires $2\log(p^{1/d})$ links. Therefore, the diameter is $d2\log(p^{1/d})$, or $2\log p$.

The advantages of a mesh of trees is that it has a smaller diameter compared to a mesh. However, this comes at the cost of increased hardware in the form of switches. Furthermore, it is difficult to derive a clean planar structure, as is the case with 2-dimensional meshes.

19 Leighton [Lei92a] discusses this solution in detail.

20 Figure 2.2 illustrates a 4×4 wraparound mesh with equal wire lengths.

21 Consider a $p \times q \times r$ mesh being embedded into a 2^d processor hypercube. Assume that $p = 2^x$, $q = 2^y$, and $r = 2^z$. Furthermore, since $p \times q \times r = 2^d$, $x + y + z = d$.

The embedding of the mesh can be performed as follows: Map processor (i, j, k) in the mesh to processor

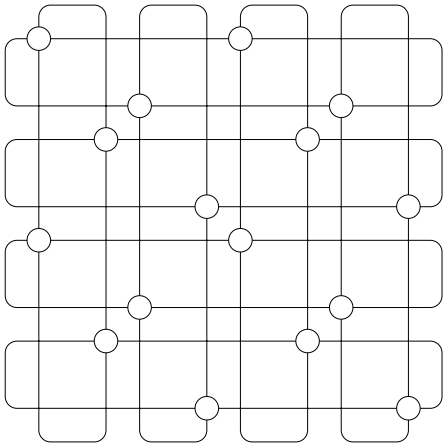

Figure 2.2 A 4×4 wraparound mesh with equal wire lengths.

$G(i, x)G(j, y)G(k, z)$ (concatenation of the Gray codes) in the hypercube using the Gray code function G described in Section 2.7.1 (page 67).

To understand how this mapping works, consider the partitioning of the d bits in the processor labels into three groups consisting of x, y, and z bits. Fixing bits corresponding to any two groups yields a subcube corresponding to the other group. For instance, fixing $y+z$ bits yields a subcube of 2^x processors. A processor (i, j, k) in the mesh has direct communication links to processors $(i + 1, j, k)$, $(i - 1, j, k)$, $(i, j + 1, k)$, $(i, j-1, k)$, $(i, j, k+1)$, and $(i, j, k-1)$. Let us verify that processors $(i+1, j, k)$ and $(i-1, j, k)$ are indeed neighbors of processor (i, j, k). Since j and k are identical, $G(j, y)$ and $G(k, z)$ are fixed. This means that the two processors lie in a subcube of 2^x processors corresponding to the first x bits. Using the embedding of a linear array into a hypercube, we can verify that processors $(i+1, j, k)$ and $(i-1, j, k)$ are directly connected in the hypercube. It can be verified similarly that the other processors in the mesh which are directly connected to processor (i, j, k) also have a direct communication link in the hypercube.

23 Ranka and Sahni [RS90] present a discussion of the embedding of a complete binary tree into a hypercube.

24 The mapping of a mesh into a hypercube follows directly from an inverse mapping of the mesh into a hypercube. Consider the congestion of the inverse mapping. A single subcube of $\sqrt{p}$ processors is mapped onto each row of the mesh (assuming a $\sqrt{p} \times \sqrt{p}$ mesh). To compute the congestion of this mapping, consider the number of links on the mesh link connecting one half of this row to the other. The hypercube has $\sqrt{p}/2$ links going across and a single row of the mesh (with wraparound) has two links going across. Therefore, the congestion of this mapping is $\sqrt{p}/4$.

It can be shown that this mapping yields the best congestion for the mapping of a hypercube into a mesh. Therefore, if the mesh links are faster by a factor of $\sqrt{p}/4$ or more, the mesh computer is superior. For the example cited, $p = 1024$. Hence, for the mesh to be better, its links must be faster by a factor of $\sqrt{1024}/4 = 8$. Since the mesh links operate at 25 million bytes per second and those of the hypercube operate at 2 million bytes per second, the mesh architecture is indeed strictly superior to the hypercube.

25 The diameter of a k-ary d-cube can be derived by traversing the farthest distance along each dimension. The farthest distance along each dimension is $k/2$ and since there are d such dimensions, the diameter is $dk/2$.

Each processor in a k-ary d-cube has $2d$ communication links. Therefore, the total number of communication links is pd.

The bisection width of a k-ary d-cube can be derived by fixing one of the dimensions and counting the number of links crossing this hyperplane. Any such hyperplane is intersected by $2k^{(d-1)}$ (for $k > 2$) communication links. The factor of 2 results because of the wraparound connections. (Note that the bisection width can also be written as $2k^{d-1}$).

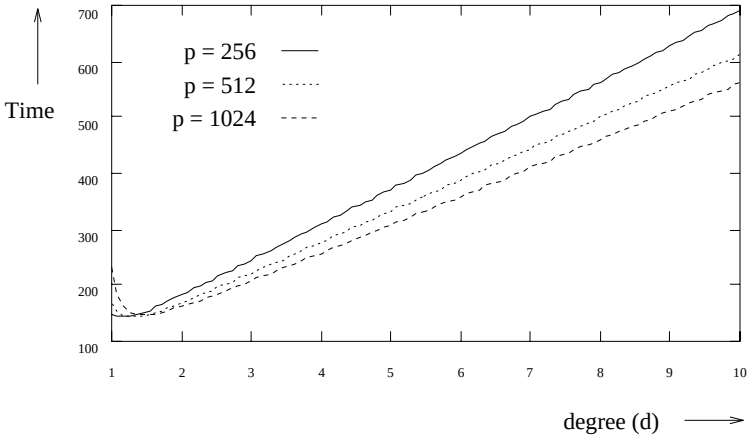

Figure 2.3 Communication time plotted against the degree of a cut-through network routing using number of communication links as a cost metric.

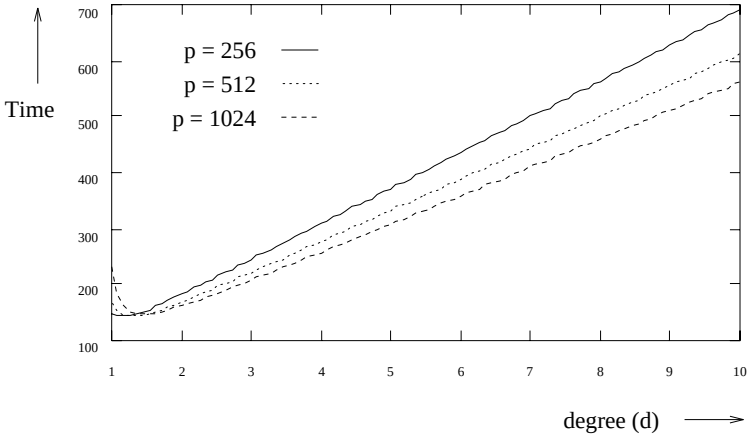

Figure 2.4 Communication time plotted against the degree of a cut-through routing network using bisection width as a cost metric.

The average communication distance along each dimension is $k/4$. Therefore, in d dimensions, the average distance l_{av} is $kd/4$.

27 (1) The cost of a k-ary d-cube of p processors in terms of number of communication links is given by dp (for $k > 2$). The corresponding cost for a binary hypercube is given by $p \log p/2$. Therefore, if the width of each channel in the k-ary d-cube is r, then the total cost is given by dpr. If this cost is identical to the cost of a binary hypercube, then $dpr = p \log p/2$, or $r = \log p/(2d)$.
(2) The bisection width of a k-ary d-cube of p processors is given by $2k^{d-1}$ and that of a binary hypercube is given by $p/2$. If the channel width of the k-ary d-cube is r, then equating the costs yields $r \times 2k^{d-1} = p/2$. Therefore, the channel width is $r = p/(4 \times k^{d-1})$. Since $k^d = p$, $r = k/4$.
(The cost and bisection width of these networks is given in Table 2.1 (page 44))

28 The average distance between two processors in a hypercube is given by $\log p/2$. The cost of communicating a message of size m between two processors in this network with cut-through routing is

$$T_{comm} = t_s + t_h \frac{\log p}{2} + t_w m.$$

The average distance between two processors in a k-ary d-cube is given by $kd/4$. The cost of communicating

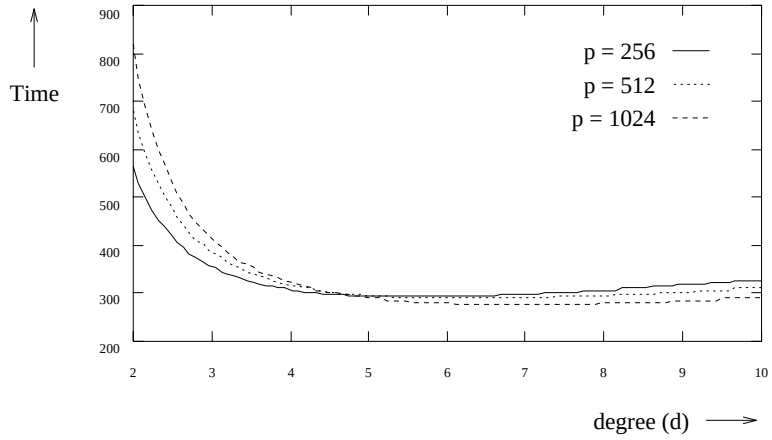

Figure 2.5 Communication time plotted against the degree of a store-and-forward network routing using number of communication links as a cost metric.

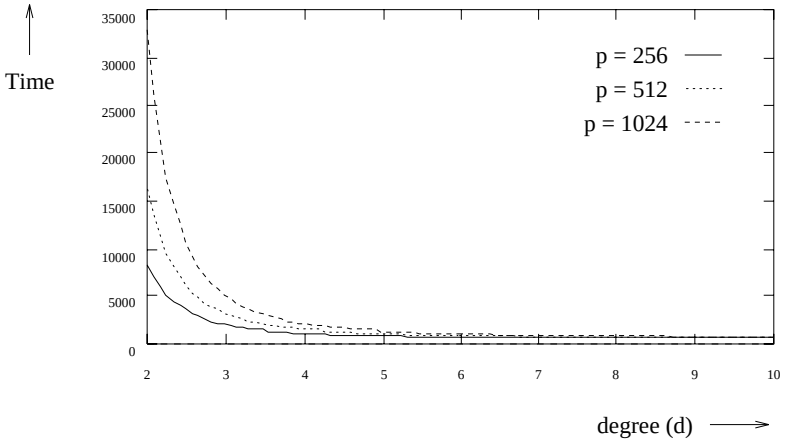

Figure 2.6 Communication time plotted against the degree of a store-and-forward network routing using bisection width as a cost metric.

a message of size m between two processors in this network with cut-through routing is

$$T_{comm} = t_s + t_h \frac{kd}{4} + \frac{t_w}{r} m,$$

where r is the scaling factor for the channel bandwidth.

From Solution 2.20, if number of channels is used as a cost metric, then we have $r = s = \log p/(2d)$. Therefore,

$$T_{comm} = t_s + t_h \frac{kd}{4} + \frac{2t_w d}{\log p} m.$$

Similarly, using the bisection width as a cost metric, we have $r = s = k/4$. Therefore,

$$T_{comm} = t_s + t_h \frac{kd}{2} + \frac{kt_w}{4} m.$$

The communication times are plotted against the dimension of the k-ary d-cube for both of these cost metrics in Figures 2.3 and 2.4.

29 The cost of communicating a message of size m between two processors in a hypercube with store-and-forward

routing is

$$T_{comm} = t_s + t_w m \frac{\log p}{2}.$$

Using the number of links as a cost metric, for a k-ary d-cube the corresponding communication time is given by

$$T_{comm} = t_s + t_w \frac{2d}{\log p} \frac{kd}{2} m.$$

This communication time is plotted against the degree of the network in Figure2.5.

Using the bisection width as a cost metric, for a k-ary d-cube the corresponding communication time is given by

$$T_{comm} = t_s + \frac{kt_w}{4} \frac{kd}{2} m.$$

This communication time is plotted against the degree of the network in Figure2.6.

CHAPTER 3

Principles of Parallel Algorithm Design

2 We assume eacg node to be of unit weight.
1. (a) 8, (b) 8, (c) 8, (d) 8.
2. (a) 4, (b) 4, (c) 7, (d) 8.
3. (a) 15/4, (b) 15/4, (c) 2, (d) 15/8.
4. (a) 8, (b) 8, (c) 3, (d) 2.
5. Number of parallel processes limited to 2: (a) 15/8, (b) 15/8, (c) 7/4, (d) 15/8. Number of parallel processes limited to 4: (a) 3, (b) 3, (c) 2, (d) 15/8. Number of parallel processes limited to 8: (a) 15/4, (b) 15/4, (c) 2, (d) 15/8.

4 Since any path from a start to a finish cannot be longer than l, there must be at least $\lceil t/l \rceil$ independent paths from start to finish nodes to accomodate all t nodes. Hence d must be $\geq \lceil t/l \rceil$. If $d > t - l + 1$, then it is impossible to have a critical path of length l or higher because because $l - 1$ more nodes are needed to construct this path. Hence $\lceil t/l \rceil d \leq d \leq t - l + 1$.

5 See Figure 3.1.

6 1,2,6,10,11,13,14, 1,2,6,10,12,13,14, 1,4,6,10,11,13,14, 1,4,6,10,12,13,14.

7 See Figure 3.1. Using three processes achieves the maximum possible speedup of 2.

8 See Figure 3.1.

9 They both take the same amount of time.

12 $\max(2m - 2, (m-1)^2)$.

13 $2m - 1$.

15 See Chapter 13.

16 See Chapter 13.

17 See Chapter 13.

19 See Chapter 9.

21 Best-case speedup = 2, worst-case sppedup = 1.5.

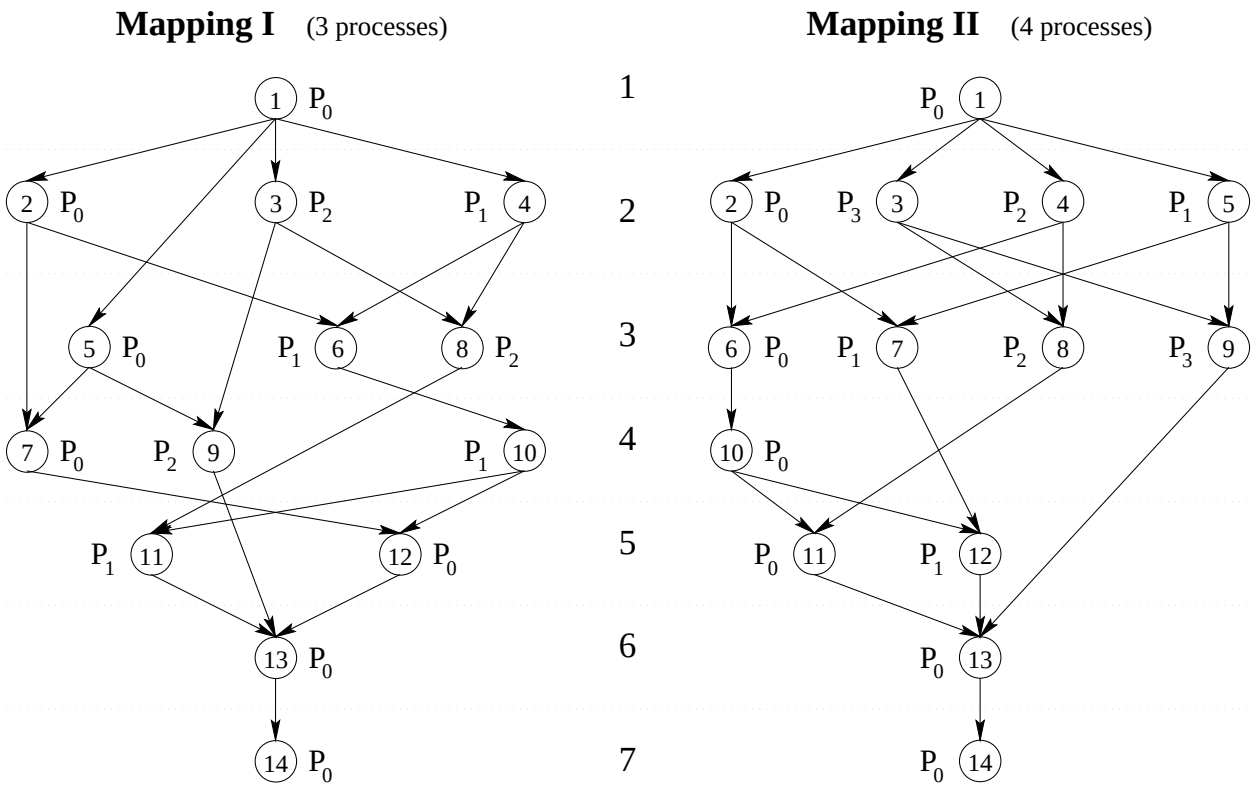

Figure 3.1 Task-dependency graphs, critical-path lengths, and mappings onto 3 and 4 processes for 3×3 block LU factorization.

CHAPTER 4

Basic Communication Operations

1 Add an extra iteration to the outer loop. Some processes in the first iteration in 4.1 and 4.2 and the last iteration in 4.3 (iteration d is all cases) will not participate in the communication.

2 **Note:** There is a typo in the problem statememt. Algorithm 3.1 should read Algorithm 4.1.

5 Refer to Figure 4.7 (page 155). In the first iteration, the following pairs of processors exchange their data of size m: (0,4), (1,5), (2,6), and (3,7). This step takes $t_s + 4t_w m$ time because four messages of size m pass through the root of the tree. At the end of this step, each processor has data of size $2m$. Now the following pairs of processors exchange their data: (0,2), (1,3), (4,6), and (5,7). In this step, the size of each message is $2m$, and two messages traverse in each channel in the same direction. Thus, this step takes $t_s + 4t_w m$. In the final step, messages of size $4m$ are exchanged without channel contention between the following pairs of processors: (0,1), (2,3), (4,5), and (6,7). This step also takes $t_s + 4t_w m$ time. In general, all-to-all broadcast can be performed on a p-processor tree of the type shown in Figure 4.7 (page 155) in $(t_s + t_w mp/2)\log p$ time.

Note that if the order of the steps is changed, then the communication cost will increase. For example, if the pairs of processors that exchange data in the first step are (0,1), (2,3), (4,5), and (6,7), and the pairs that exchange data in the last step are (0,4), (1,5), (2,6), and (3,7), then the total communication time will be $t_s \log p + t_w m(p^2 - 1)/3$.

On the other hand, another scheme can perform all-to-all broadcast on a p-processor tree with CT routing in $(t_s + t_w m)(p - 1)$ time. This can be done by embedding a logical ring onto the tree, as shown in Figure 4.1. Note that no two messages travelling in the same direction share a communication link. It is easy to verify that executing the algorithms of Figure 4.9 (page 159) of the text results in total communication time of $(t_s + t_w m)(p - 1)$. This scheme leads to a higher t_s term, but a lower t_w term.

7 In the hypercube algorithm for all-to-all personalized communication described in Section 4.4 (page 167), the entire message received by a processor is not required by that processor. On a completely connected network, this operation can be performed in $(t_s + t_w m)(p - 1)$ time. However, the best communication time for one-to-all broadcast is $(t_s + t_w m)\log p$ (the same as in the case of a hypercube) if an entire message is routed along the same path.

8 The communication pattern for multinode accumulation on a hypercube is exactly the opposite of the pattern shown in Figure 4.11 (page 164) for all-to-all broadcast. Each processor has p messages to start with (as in Figure 4.11 (page 164)(d)), one for each processor (Figure 4.8 (page 157)). In the i^{th} iteration $(0 < i \leq \log p)$ of the multinode accumulation algorithm, processors communicate along the $(\log p - i + 1)^{th}$ dimension

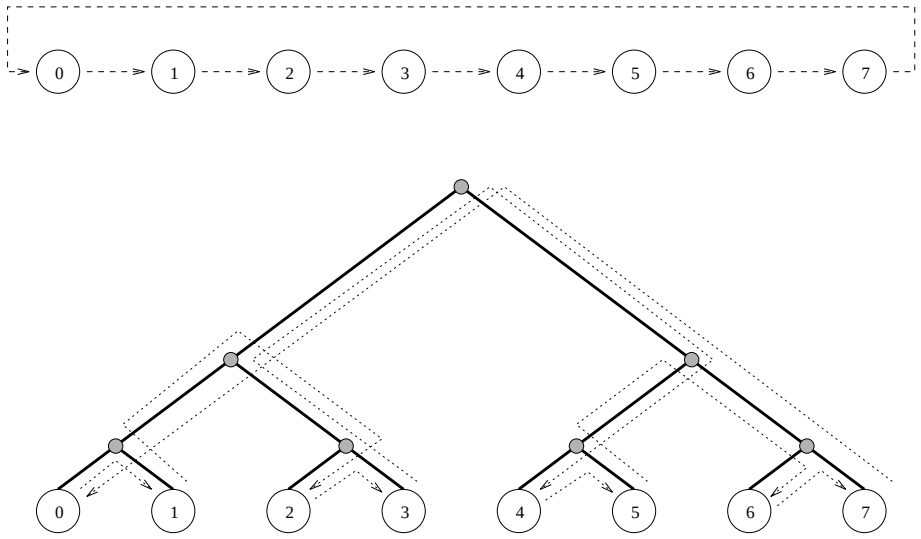

Figure 4.1 All-to-all broadcast on an eight-processor tree by mapping a logical eight-processor ring onto it.

of the hypercube so that processor j communicates with processor $j \pm 2^{\log p - i}$. For example, in the first step, processor 0 sends the messages destined for processors 4, 5, 6, and 7 to processor 7, and processor 7 sends the messages destined for processors 0, 1, 2, and 3 to processor 0. After the first communication step, each processor adds the $mp/2$ numbers received to the $mp/2$ numbers already residing on them. The size of the data communicated and added in an iteration is half of that in the previous iteration. The total time is $\Sigma_{i=1}^{\log p}(t_s + (t_w + t_{add})mp/2^i)$, which is equal to $t_s \log p + m(t_w + t_{add})(p-1)$.

9 The shortest distance between processors i and j is their Hamming distance $H_{i,j}$ (that is, the numbers of 1's in bitwise exclusive-or of i and j). Let $\log p = d$. The solution to this problem is the same as the number of d-bit numbers with exactly $H_{i,j}$ 1's in their binary representation. The solution is $d!/(H_{i,j}!(d - H_{i,j})!)$.

10 First, each processor performs a local prefix sum of its n/p numbers in $(n/p - 1)t_{add}$ time. In the second step, the p processors compute prefix sums of p numbers by using the last prefix sum resulting from the local computation on each processor. This step takes $(t_{add} + t_s + t_w)\log p$ time. Finally, the result of the parallel prefix sums operation of the second step is added to all the n/p prefix sums of the first step at each processor in $t_{add}n/p$ time. Therefore,

$$T_P = (2\frac{n}{p} - 1)t_{add} + (t_{add} + t_s + t_w)\log p.$$

11 Consider a square mesh without wraparound connections.

$$\begin{aligned}
\text{Number of words transmitted by a processor} &= m(p-1) \\
\text{Total number of processors} &= p \\
l_{av} &= \sqrt{p} \\
\text{Total traffic} &= m(p-1)p\sqrt{p} \\
\text{Total number of links in the network} &= 4p \\
T^{lower_bound}_{all_to_all_pers} &= \frac{m(p-1)p\sqrt{p}}{4p} \\
&= \frac{m(p-1)\sqrt{p}}{4}
\end{aligned}$$

13 The total number of links in a regular 3-D mesh of p processors is $3p$. Therefore, one-to-all broadcast and one-to-all personalized communication are more cost-effective on a sparse 3-D mesh (taking approximately

$3t_w mp^{1/3}$ and $t_w mp$ time, respectively) than on a regular 3-D mesh (on which they take the same amount of time as on the sparse 3-D mesh). On the other hand, a sparse 3-D mesh is less cost-effective for all-to-all broadcast (approximately $2t_w mp$ time) than a regular 3-D mesh (approximately $t_w mp$ time).

14 In this communication model, one-to-all broadcast, all-to-all broadcast, and one-to-all personalized communication take approximately $3t_s p^{1/3}$ time on both the architectures. Thus, a sparse 3-D mesh is more cost-effective.

15 In all-to-all broadcast on a hypercube, the message size doubles in each iteration. In k-to-all broadcast, in the worst case, the message size doubles in each of the first $\log k$ iterations, and then remains mk in the remaining $\log(p/k)$ iterations. The total communication time of the first k iterations is $t_s \log k + t_w m(k-1)$. The total communication time of the last $\log(p/k)$ iterations is $(t_s + t_w mk)\log(p/k)$. Thus, the entire operation is performed in $t_s \log p + t_w m(k \log(p/k) + k - 1)$ time.

21 As the base case of induction, it is easy to see that the statement is true for a 2-processor hypercube. Let all the p data paths be congestion-free in a p-processor hypercube for all $q < p$. In a $2p$-processor hypercube, if q-shifts for all $q < p$ are congestion-free, then q-shifts for all $q < 2p$ are also congestion-free (***Hint*** (1)). So the proof is complete if we show that q-shifts for all $q < p$ are congestion-free in a $2p$-processor hypercube. Consider a circular q-shift for any $q < p$ on a $2p$-processor hypercube. All the $p - q$ data paths leading from a processor i to a processor j such that $i < j < p$ are the same as in a p-processor hypercube, and hence, by the induction hypothesis, do not conflict with each other. The remaining q data paths leading from a processor i to a processor j on the p-processor hypercube, such that $j < i < p$, lead to processor $j + p$ on a $2p$-processor hypercube. Processor $j + p$ is connected to processor j by a single link in the highest (that is, $(\log p + 1)^{\text{th}}$) dimension of the $2p$-processor hypercube. Thus, following the E-cube routing, the data path from processor i to processor $j + p$ in a circular q-shift on the $2p$-processor hypercube is the data path from processor i to processor j in a circular q-shift on a p-processor hypercube appended by a single link. The original path from processor i to j is congestion free (induction hypothesis) and the last link is not shared by any other message because it is unique for each message. Thus, q-shifts for all $q < p$ are congestion-free in a $2p$-processor hypercube.

22 In a 1-shift on a 2-processor hypercube, the highest integer j such that 1 is divisible by 2^j is 0. Since $\log 2 = 1$, the statement is true for $p = 2$ (because $q = 1$ is the only possible value of q for $p = 2$). Let the statement hold for a p-processor hypercube. As shown in the solution to Problem 4.22 (page 193), the length of a path between processors i and j, such that $j < i < p$, increases by a single link if the same shift operation is performed on a $2p$-processor hypercube. Thus, the length of the longest path increases from $\log p - \gamma(q)$ to $\log p - \gamma(q) + 1$ (which is equal to $\log(2p) + \gamma(q)$) as the number of processors is increased from p to $2p$. Note that a circular q shift is equivalent to a $(2p - q)$-shift on a $2p$-processor hypercube. So the result holds for all q such that $0 < q < 2p$.

24 Cost of Network $\propto$ Total Number of Links:

$$\begin{aligned} \text{No. of links in 2-D mesh with wraparound} &= 2p \\ \text{No. of links in a hypercube} &= \frac{p \log p}{2} \\ s &= \frac{\log p}{4} \end{aligned}$$

The communication times for various operations on a hypercube with cut-through routing can be found in Table 4.1 (page 187). For a 2-D wraparound mesh with CT routing and each link $(\log p/4)$-channel wide, the communication times for various operations are as follows:

$$\begin{aligned} T_{one\_to\_all\_broadcast} &= t_s \log p + 4t_w m \\ T_{all\_to\_all\_broadcast} &= 2t_s(\sqrt{p} - 1) + \frac{4t_w m(p-1)}{\log p} \end{aligned}$$

$$T_{one\_to\_all\_personalized} = 2t_s(\sqrt{p}-1) + \frac{4t_w m(p-1)}{\log p}$$
$$T_{all\_to\_all\_personalized} = 2t_s(\sqrt{p}-1) + \frac{4t_w mp(\sqrt{p}-1)}{\log p}$$

If the cost of a network is proportional to the total number of links in it, then a mesh is asymptotically more cost-effective than a hypercube for all operations except all-to-all personalized communication.

Cost of Network $\propto$ Bisection Width:

$$\text{Bisection width of 2-D mesh with wraparound} = 2\sqrt{p}$$
$$\text{Bisection width of hypercube} = \frac{p}{2}$$
$$s' = \frac{\sqrt{p}}{4}$$

The following are the communication times on a 2-D wraparound mesh with CT routing and each link $(\sqrt{p}/4)$-channel wide:

$$T_{one\_to\_all\_broadcast} = t_s \log p + \frac{4t_w m \log p}{\sqrt{p}}$$
$$T_{all\_to\_all\_broadcast} = 2t_s(\sqrt{p}-1) + 4t_w m\sqrt{p}$$
$$T_{one\_to\_all\_personalized} = 2t_s(\sqrt{p}-1) + 4t_w m\sqrt{p}$$
$$T_{all\_to\_all\_personalized} = 2t_s(\sqrt{p}-1) + 4t_w mp$$

If the cost of a network is proportional to its bisection width, then a mesh is asymptotically more cost-effective than a hypercube for all operations except all-to-all personalized communication. For all-to-all personalized communication, both interconnection networks have the same asymptotic cost-performance characteristics.

25 Assuming the one-port communication model, one-to-all broadcast, all-to-all broadcast, and one-to-all personalized communication take the same amount of time on a completely connected network as on a hypercube. All-to-all personalized communication can be performed in $(t_s + t_w m)(p-1)$ time on a completely connected network.

CHAPTER 5

Analytical Modeling of Parallel Programs

1

$$S = \frac{W}{T_P} = \frac{W}{W_S + \frac{W - W_S}{p}}$$

As p increases, the fraction $(W - W_S)/p$ approaches zero. However, no matter how large p is, S cannot exceed W/W_S.

2 On a single processor, 11 arcs are traversed by DFS before the solution is found. On two processors, only four arcs are traversed by P_1 before it finds the solution. Therefore, speedup on two processors is $11/4 = 2.75$. The anomaly is due to the fact that in Figure 5.10 (page 228)(b), the algorithm being executed is not the same as in Figure 5.10 (page 228)(a). Tha parallel algorithm performs less overall work than the serial algorithm. The algorithm in Figure 5.10 (page 228)(b) is not true depth-first search; it is part depth-first and part breadth-first. If a single processor alternates between the nodes of the left and right subtress (thereby emulating the two-processor algorithm of Figure 5.10 (page 228)(b)), then the serial algorithm traverses only eight arcs.

3 The degree of concurrency, the maximum speedup, and speedup when $p = 1/2 \times$ degree of concurrency for the graphs are given in the following table:

	(a)	(b)	(c)	(d)
Degree of concurreny	2^{n-1}	2^{n-1}	n	n
Maximum possible speedup	$\frac{2^n-1}{\log n}$	$\frac{2^n-1}{\log n}$	$\frac{n^2}{2n-1}$	$\frac{n+1}{2}$
Speedup when $p = 1/2 \times$ (degree of concurrency)	$\frac{2^n-1}{\log n+1}$	$\frac{2^n-1}{\log n+1}$	$\frac{n^2}{3n-2}$	$\frac{n+1}{3}$

The efficiency corresponding to various speedups can be computed by using $E = S/p$, and the overhead function can be computed by using the following relations:

$$\begin{aligned} E &= \frac{W}{pT_P} \\ pT_P &= \frac{W}{E} \\ T_o &= pT_P - W \\ &= W(\frac{1}{E} - 1) \end{aligned}$$

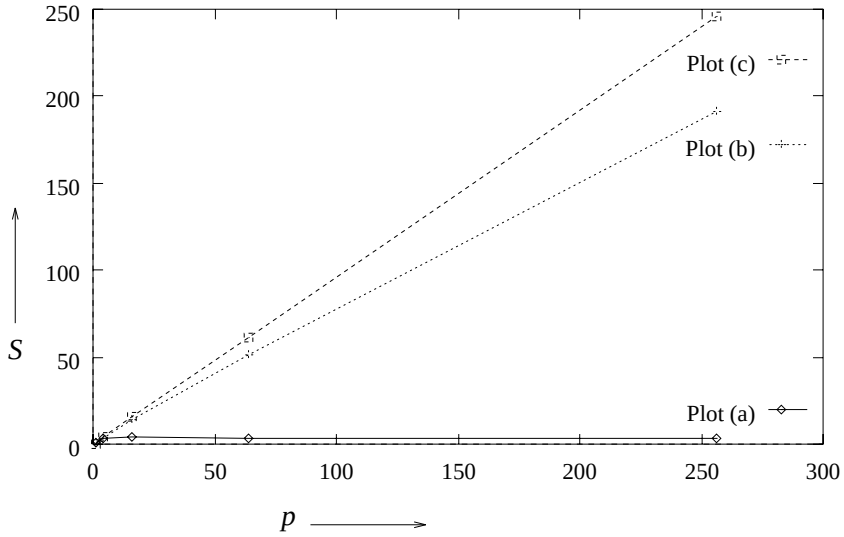

Figure 5.1 Standard (Plot (a)), scaled (Plot (b)), and isoefficient (Plot (C)) speedup curves for Problems 5.5 (page 230) and 5.6 (page 230).

4 Let the isoefficiency function $f(p)$ be greater than $\Theta(p)$. If $p = \Theta(W)$, or $W = \Theta(p)$, then $W < f(p)$ because $f(p) > \Theta(p)$. Now if $W < f(p)$, then $E < \Theta(1)$, or $pT_P > \Theta(W)$.
The converse implies that $pT_P = \Theta(W)$ only if $p < \Theta(W)$, or $E = \Theta(1)$ only if $p < \Theta(W)$; that is, $W > \Theta(p)$. Thus, the isoefficiency function is greater than $\Theta(p)$.

5 Plots (a) and (b) in Figure 5.1 represent the standard and scaled speedup curves, respectively.

6 See Plot (c) in Figure 5.1.

7 Figure 5.2 plots the efficiency curves corresponding to the speedup curves in Figure 5.1.

8 Scaled speedup, in general, does not solve the problem of declining efficiency as the number of processors is increased. The scaled speedup curve is linear only if the isoefficiency function of a parallel system is linear (that is, $\Theta(p)$).

9

$$\begin{aligned} 512 &\geq T_P = \frac{n}{p} - 1 + 11 \log p \\ (513 - 11 \log p) \times p &\geq n \end{aligned}$$

The following table gives the largest n corresponding to a p.

p	1	4	16	64	256	1024	4096
n	513	1964	7504	28,608	108,800	412,672	1560,576

It is not possible to solve an arbitrarily large problem in a fixed amount of time, even if an unlimited number of processors are available. For any parallel system with an isoefficiency function greater than $\Theta(p)$ (such as, the one in Problem 5.5 (page 230)), a plot between p and the size of the largest problem that can be solved in a given time using p processors will reach a maximum.
It can be shown that for cost-optimal algorithms, the problem size can be increased linearly with the number of processors while maintaining a fixed execution time if and only if the isoefficiency function is $\Theta(p)$. The proof is as follows. Let $C(W)$ be the degree of concurrency of the algorithm. As p is increased, W has to be increased at least as $\Theta(p)$, or else p will eventually exceed $C(W)$. Note that $C(W)$ is upper-bounded by $\Theta(W)$ and p is upper-bounded by $C(W)$. T_P is given by $W/p + T_o(W, p)/p$. Now consider the following

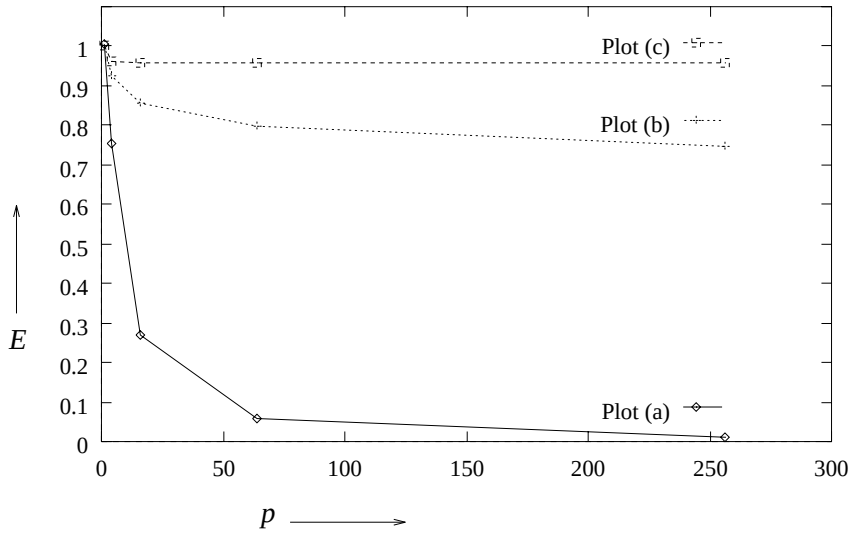

Figure 5.2 Efficiencies corresponding to the speedups in Figure 5.1.

two cases. In the first case $C(W)$ is smaller than $\Theta(W)$. In this case, even if as many as $C(W)$ processors are used, the term $W/C(W)$ of the expression for T_P will diverge with increasing W, and hence, it is not possible to continue to increase the problem size and maintain a fixed parallel execution time. At the same time, the overall isoefficiency function grows faster than $\Theta(p)$ because the isoefficiency due to concurrency exceeds $\Theta(p)$. In the second case in which $C(W) = \Theta(W)$, as many as $\Theta(W)$ processors can be used. If $\Theta(W)$ processors are used, then the first term in T_P can be maintained at a constant value irrespective of W. The second term in T_P will remain constant if and only if $\frac{T_o(W,p)}{p}$ remains constant when $p = \Theta(W)$ (in other words, T_o/W remains constant while p and W are of the same order). This condition is necessary and sufficient for linear isoefficiency.
For further details, see paper by Kumar and Gupta [KG91].

11 See solution to Problem 4.10 (page 191).

$$\begin{aligned} T_P &= 2\frac{n}{p} - 1 + 10\log p \\ S &= \frac{n/p - 1}{2\frac{n}{p} - 1 + 10\log p} \\ \text{Isoefficiency function} &= \Theta(p \log p) \end{aligned}$$

12 The algorithm is a single-node accumulation in which the associative operation is a comparison that chooses the greater of the two numbers. The algorithm is not cost-optimal for finding the maximum of n numbers on an n-processor mesh.

13 Refer to the paper by Gupta and Kumar [GK93].

14

$$\begin{aligned} Equation\ 5.21, in this case, leads to the following: T_P &= \frac{n\log n}{p} + \frac{10n\log p}{p} \\ \frac{\mathrm{d}T_P}{\mathrm{d}p} &= \frac{-n\log n}{p^2} - \frac{10n\log p}{p^2} + \frac{10n}{p^2} = 0 \\ 10\log p &= 10 - \log n \\ \log p &= 1 - \log n^{1/10} \\ p &= \frac{2}{n^{1/10}} \end{aligned}$$

This is actually a condition for a maximum, rather than a minimum (this can be verified by a taking a second derivative). Since $T_o < \Theta(p)$ in this case, the minimum parallel run time is obtained when as many processors as possible are used to solve the problem [GK93]. Hence, the actual value of T_P^{min} should be derived by putting $p = n$ in the expression for T_P. This leads to $T_P^{min} = 11 \log n$.

15 Refer to the paper by Gupta and Kumar [GK93].

CHAPTER 6

Programming Using the Message-Passing Paradigm

The solutions for this chapter will be provided at a later time.

CHAPTER 7

Programming Shared Address Space Platforms

Since all of the questions in this chapter are implementation based and implementations (and their performance) differ markedly, we only give outlines of solutions.

1
- To time thread creation, create a large number of threads in a for loop and time this loop.
- To time thread join, time the corresponding join functions executed in a for loop.
- To time successful locks, create a large array of locks and lock all of these in a for loop.
- To time successful unlocks, time corresponding unlocks in a for loop.
- To time successful trylock, repeat the process above for successful locks.
- To time unsuccessful trylocks, first lock a mutex and repeatedly try to lock it using trylock in a for loop.
- To time a condition wait, set up two threads, one that waits and another that signals. Execute a large number of wait-signal pairs. The smallest of these times, with high probability, gives the condition-wait time.
- To time a condition signal, time the function with various number of waiting threads.
- The condition broadcast case is similar to the condition signal case, above.

2 The queue is implemented as a heap. The heap is protected by a single lock. Each insertion/extraction is protected by a lock associated with the heap. At 64 threads, in most thread systems, students should observe considerable contention.

3 The condition wait case is similar to the one above, except, instead of locking, a thread executes a condition wait and instead of unlocking, a thread does a condition signal. The performance of this formulation should be marginally better but will saturate as well.

4 This is a simple extension of a producer-consumer paradigm in which the network thread is the producer for the decompressor thread, which in turn is the producer for the render thread. The code can be implemented starting from the code in Example 7.6.

7 The read-write locks are illustrated in 7.8.1. These functions can be easily adapted to the problem of binary tree search. Only insertions require write locks. All other threads only need read locks.

8 The performance saturates as the number of threads increases because of contention on the global lock.

9 The saturation point is pushed back because of the use of read-write locks since reads can be concurrently executed.

10 There are several strategies for implementing a Sieve. As the numbers pass through, the first number installs itself as a thread. The thread is responsible for filtering all multiples of the associated number. The key programming task is to ensure coordination between threads eliminating different multiples (i.e., the thread eliminating multiples of 3 must be able to filter a block before the thread responsible for eliminating multiples of 5).

An alternate strategy is to determine all the primes whose multiples need to be eliminated first and them to instantiate them as threads. This algorithm has a significant serial component and does extra work.

11 The implementation described saturates because of contention on the open list.

12 The saturation point is pushed back (depending on the choice of k). Also, as the number of heaps is increased, excess work increases, causing a reduction in the speedup.

14 When the function becomes computationally expensive, all of the formulations can be expected to perform similarly.

16 The implementation is very similar to the pthreads version, except the task of creation and joining is accomplished using sections, and explicit locks can be used in OpenMP, much the same way as in pthreads.

CHAPTER 8

Dense Matrix Algorithms

1 The communication time of the algorithm for all-to-all personalized communication on a hypercube with cut-through routing (Section 4.4 (page 167)) is

$$T_{all\_to\_all\_pers} = t_s(p-1) + t_w m(p-1).$$

With the given parameters, the communication time is 22428. Thus the algorithm for cut-through routing is faster for the given parameters. For a different set of parameters, the algorithm for store-and-forward routing may be faster.

4 Let us start from the central relation that must be satisfied in order to maintain a fixed efficiency.

$$\begin{aligned}
W &= KT_o \\
n^2 &= K(t_s p \log p + t_w n\sqrt{p}\log p \\
0 &= n^2 - n(Kt_w\sqrt{p}\log p) - Kt_s p \log p \\
n &= \frac{Kt_w\sqrt{p}\log p \pm \sqrt{K^2t_w^2 p(\log p)^2 + 4K(t_s p \log p)}}{2} \\
n^2 &= \frac{K^2t_w^2 p(\log p)^2}{2} + Kt_s p \log p \\
&\quad \pm \frac{Kt_w p \log p\sqrt{K^2t_w^2(\log p)^2 + 4K(t_s \log p)}}{2}
\end{aligned}$$

¿From the preceding equation, it follows that if $Kt_w^2(\log p)^2$ is greater than (which is most likely to be the case for practical values of K, t_w, and p), then the overall isoefficiency function is $\Theta(p(\log p)^2)$ due to the t_w term of T_o.

5 The communication time remains $2(t_s \log p + t_w n^2/\sqrt{p})$. The algorithm performs $\sqrt{p}$ matrix multiplications with submatrices of size $n/\sqrt{p} \times n/\sqrt{p}$. Each of these multiplications take $\Theta(n^{2.81}/p^{1.41})$ time. The total parallel run time is $\Theta(n^{2.81}/p^{0.91}) + 2(t_s \log p + t_w n^2/\sqrt{p})$. The processor-time product is $\Theta(n^{2.81}p^{0.09}) + 2(pt_s \log p + t_w n^2\sqrt{p})$. The $\Theta(n^{2.81}p^{0.09})$ term of the processor-time product is greater than the $\Theta(n^{2.81})$ sequential complexity of the algorithm. Hence, the parallel algorithm is not cost-optimal with respect to a serial implementation of Strassen's algorithm

6 This variant of the DNS algorithm works with n^2q processors, where $1 < q < n$. The algorithm is similar to that for one element per processor except that a logical processor array of q^3 superprocessors (instead of n^3 processors) is used, each superprocessor containing $(n/q)^2$ hypercube processors. In the second step,

multiplication of blocks of $(n/q) \times (n/q)$ elements instead of individual elements is performed. Assume that this multiplication of $(n/q) \times (n/q)$ blocks is performed by using Cannon's algorithm for one element per processor. This step requires a communication time of $2(t_s + t_w)n/q$.

In the first stage of the algorithm, each data element is broadcast over q processors. In order to place the elements of matrix A in their respective positions, first $A[j,k]$ stored at processor $P_{(0,j,k)}$ is sent to processor $P_{(k,j,k)}$ in $\log q$ steps. This data is then broadcast from processor $P_{(k,j,k)}$ to processors $P_{(k,j,l)}, 0 \leq l < q$, again in $\log q$ steps. By following a similar procedure, the elements of matrix B are transmitted to their respective processors. In the second stage, groups of $(n/q)^2$ processors multiply blocks of $(n/q) \times (n/q)$ elements each processor performing n/q computations and $2n/q$ communications. In the final step, the elements of matrix C are restored to their designated processors in $\log q$ steps. The total communication time is $(t_s + t_w)(5 \log q + 2n/q)$ resulting in the parallel run time given by the following equation:

$$T_P = \frac{n^3}{p} + (t_s + t_w)(5 \log(\frac{p}{n^2}) + 2\frac{n^3}{p})$$

The communication time of this variant of the DNS algorithm depends on the choice of the parallel matrix multiplication algorithm used to multiply the $(n/q) \times (n/q)$ submatrices. The communication time can be reduced if the simple algorithm is used instead of Cannon's algorithm.

7 Let the division operation on the i^{th} row $(0 \leq i < n)$ be performed during time step t_i. Then in time step $t_i + 1$, the active part of the i^{th} row after division is sent to the processor storing the $(i+1)^{\text{st}}$ row. In time step $t_i + 2$, the processor storing the $(i+1)^{\text{st}}$ row sends this data to the processor storing the $(i+2)^{\text{nd}}$ row. In time step $t_i + 3$, the elimination operation is performed on the $(i+1)^{\text{st}}$ row. and in time step $t_i + 4$, the division operation on the $(i+1)^{\text{st}}$ row. Thus, $t_{i+1} = t_i + 4$. In other words, four time steps separate the division operations on any two consecutive rows. Hence, in $4n$ steps, all the n division operations can be performed and Gaussian elimination can be completed on an $n \times n$ matrix. However, the communication operations of time steps $t_{n-2} + 3$, $t_{n-1} + 1$, $t_{n-1} + 2$, and $t_{n-1} + 3$ are not performed. Thus, the actual number of steps is $4(n-1)$.

8 See papers by Geist and Romine [GR88] and Demmel et al. [DHvdV93] for solutions to Problems 8.8 (page 373), 8.9 (page 374), and 8.10 (page 374).

11 The communication time of each one-to-all broadcast is $\Theta(\sqrt{n})$. Therefore, the total time spent in performing these broadcast in all the iterations is $\Theta(n\sqrt{n})$. Since the number of processors is n^2, the communication time contributes $\Theta(n^3\sqrt{n})$ to the processor-time product, which is greater than the $\Theta(n^3)$ complexity of serial Gaussian elimination.

12 As shown in Section 8.3.1 (page 353), disregarding the communication time, the parallel run time of Gaussian elimination with block-striped and block-checkerboard partitioning is n^3/p and $2n^3/p$, respectively. Since the sequential run time is $2n^3/3$, the total overhead due to processor idling in the two cases is $n^3/3$ and $4n^3/3$, respectively.

With cyclic-striped mapping, the difference between the maximally loaded processor and the minimally loaded processor in any iteration is, at most, that of one row. Thus, the overhead due to idle time per iteration is $O(np)$, and the overhead due to idle time over all the iterations s $O(n^2 p)$.

With cyclic-checkerboard mapping, the difference between the maximally loaded processor and the minimally loaded processor in any iteration is, at most, that of one row and one column. The maximum overhead due to idle time per iteration is $O(p \times (n^2/p - (n/sqrtp - 1)^2))$, which is $O(n\sqrt{p})$. The overhead due to idle time over all the iterations s $O(n^2\sqrt{p})$.

13 As shown in Section 8.3.1 (page 353), the parallel run time of the asynchronous (pipelined) version of Gaussian elimination with checkerboard partitioning is $\Theta(n^3/p)$. This implies that for all values of n and p, the order of magnitude of the communication time is either the same as or less than that of the time spent in useful computation. The reason is that the sequential complexity of Gaussian elimination is $\Theta(n^3)$, and the parallel run time on p processors has to be at least $\Theta(n^3/p)$. Hence, the overall isoefficiency function is the same as

that due to concurrency. Since $p \leq n^2$, $W = n^3 = \Theta(p^{3/2})$ is the isoefficiency function due to concurrency.

16 The communication pattern (and hence, the parallel run time) of pipelined Cholesky factorization with checkerboard partitioning is very similar to that of pipelined Gaussian elimination without pivoting. Figure 8.1 illustrates the first 16 steps of Cholesky factorization on a 5×5 matrix on 25 processors. A comparison of Figures 8.11 (page 364) and 8.1 shows that the communication and and computation steps in the upper triangular halves of coefficient matrices are the same in both cases. For simplicity we remove the step of line 5 of Program 8.6 (page 375) and replace line 7 by $A[k, j] := A[k, j]/\sqrt{A[k, k]}$. The original $A[k, k]$ can be replaced by $\sqrt{A[k, k]}$ later, whenever $P_{k,k}$ is free, and this step is not shown in Figure 8.1.
The actions in first eight steps of Figure 8.1 are as follows:

(a) $A[0, 0]$ sent to $P_{0,1}$ from $P_{0,0}$ (iteration $k = 0$ starts).

(b) $A[0, 1] := A[0, 1]/\sqrt{A[0, 0]}$.

(c) $A[0, 0]$ sent to $P_{0,2}$ from $P_{0,1}$;
$A[0, 1]$ (after division) sent to $P_{1,1}$ from $P_{0,1}$.

(d) $A[0, 2] := A[0, 2]/\sqrt{A[0, 0]}$;
$A[1, 1] := A[1, 1] - A[0, 1] \times A[0, 1]$.

(e) $A[0, 0]$ sent to $P_{0,3}$ from $P_{0,2}$;
$A[0, 2]$ (after division) sent to $P_{1,2}$ from $P_{0,2}$;
$A[0, 1]$ sent to $P_{1,2}$ from $P_{1,1}$.

(f) $A[0, 3] := A[0, 3]/\sqrt{A[0, 0]}$;
$A[1, 2] := A[1, 2] - A[0, 1] \times A[0, 2]$.

(g) $A[1, 1]$ sent to $P_{1,2}$ from $P_{1,1}$ (iteration $k = 1$ starts);
$A[0, 0]$ sent to $P_{0,4}$ from $P_{0,3}$;
$A[0, 3]$ (after division) sent to $P_{1,3}$ from $P_{0,3}$;
$A[0, 1]$ sent to $P_{1,3}$ from $P_{1,2}$;
$A[0, 2]$ sent to $P_{2,2}$ from $P_{1,2}$.

(h) $A[1, 2] := A[1, 2]/\sqrt{A[1, 1]}$;
$A[0, 4] := A[0, 4]/\sqrt{A[0, 0]}$;
$A[1, 3] := A[1, 3] - A[0, 1] \times A[0, 3]$;
$A[2, 2] := A[2, 2] - A[0, 2] \times A[0, 2]$.

17 Plots (a) and (b) in Figure 8.2 represent the standard and scaled speedup curves, respectively.

18 See Plot (c) in Figure 8.2.

19 Figure 8.3 plots the efficiency curves corresponding to the speedup curves in Figure 8.2.

20 Scaled speedup, in general, does not solve the problem of declining efficiency as the number of processors is increased. Only if the isoefficiency function of a parallel system is linear (that is, $\Theta(p)$), then the efficiency corresponding to the scaled speedup curve will be constant.

21 The following table gives the largest n corresponding to a p such that two $n \times n$ matrices can be multiplied in time less than or equal to 512 units. However, it should be noted that these values are based on the expression for T_P given by Equation 8.14 (page 347). Equation 8.14 (page 347) gives an expression for the parallel run time when the matrices are partitioned uniformly among the processors. For the values of n given in the following table, it may not be possible to partition the matrices uniformly.

p	1	4	16	64	256	1024	4096
n	8	13	22	35	58	97	167

It is not possible to solve an arbitrarily large problem in a fixed amount of time, even if an unlimited number of processors are available. For any parallel system with an isoefficiency function greater than $\Theta(p)$ (such as, the one in Problem 8.17 (page 375)), a plot between p and the size of the largest problem that can be solved in a given time using p processors will reach a maximum.
See the solution to Problem 5.9 (page 231) for more details.

23 Steps 3 and 6 of Program 8.7 (page 376) involve the initialization of the resultant matrix and can be performed in constant time. Step 8 requires processor (i, j) to read the values of elements $A[i, k]$ and $B[k, j]$ for $k = 0$ to $n - 1$. A CREW PRAM allows these read operations to be performed concurrently. There are 3 memory read operations and one memory write operation and one multiply and add operation. Therefore the total time for step 8 is $4t_{local} + t_c$. For n iterations of this loop, the total time taken is $n(4t_{local} + t_c)$. The formulation is cost optimal because the processor time product of $\Theta(n^3)$ is equal to the serial runtime of the algorithm.

24 In absence of the concurrent read operation, memory reads in step 8 will be serialized. During iteration 1, $(k = 0)$, a processor (i, j) accesses values $C[i, j]$, $A[i, 0]$, and $B[0, j]$. There is no conflict in accessing $C[i, j]$. However, there are n processors trying to access a single element of A and n processors trying to access a single element of B. Similarly, in any iteration, there are n accesses to any memory location. Serializing these accesses, we have a single memory read and write operation (for $C[i, j]$), n read operations for $A[i, k]$, and n read operations for $B[k, j]$. Therefore, the time taken for this step is $2t_{local} + 2nt_{local} + t_c$. For n iterations of the loop, the total time taken is $n(2t_{local} + 2nt_{local} + t_c)$.
The algorithm is not cost optimal because the processor-time product is $\Theta(n^4)$, which is higher than the serial runtime of $\Theta(n^3)$.

25 Let us consider the time taken in step 8 of Program 8.7 (page 376). Elements $C[i, j]$, $A[i, j]$, and $B[i, j]$ are available locally. The remaining values have to be fetched from other processors. We assume that at any point of time, only one processor can access a processors' memory. In any iteration, there are $n - 1$ non-local accesses to each element of matrices A and B, and one local reference. This takes time $2(t_{local} + (n - 1)t_{nonlocal})$. Furthermore, $2t_{local}$ time is taken for reading and writing the values of $C[i, j]$. Therefore, step 8 takes time $t_c + 2t_{local} + 2(t_{local} + (n - 1)t_{nonlocal})$. For n iterations, the total time is $n(t_c + 2t_{local} + 2(t_{local} + (n - 1)t_{nonlocal}))$, which is equal to $nt_c + 4nt_{local} + 2n(n - 1)t_{nonlocal}$.

26 (a) Access times described in Problem 5.31: In Program 8.8 (page 377), step 8 staggers access to elements of matrices A and B such that there are no concurrent read operations. Therefore, step 8 can be performed in time $4t_{local} + t_c$. For n iterations, the total time taken is $n(4t_{local} + t_c)$.
The formulation is cost optimal because the processor time product of $\Theta(n^3)$ is equal to the serial runtime of the algorithm.

(b) Access times described in Problem 5.32: During any iteration of step 8, one processor is accessing an element of A locally and the remaining processors are accessing $n-1$ different elements of A non-locally. This is also the case with matrix B. Therefore, the time taken for executing step 8 is $t_c+2t_{local}+2t_{nonlocal}$.
The formulation is still cost optimal because the processor time product of $\Theta(n^3)$ is equal to the serial runtime of the algorithm.

27 Consider the case in which p processors are organized in a logical mesh of $\sqrt{p} \times \sqrt{p}$. Each processor is assigned a block of $n/\sqrt{p} \times n/\sqrt{p}$ elements. In the k^{th} step, the processor responsible for computing $C[i, j]$ reads the values of $A[i, k]$ and $B[k, j]$ and keeps a running sum $C[i, j] = C[i, j] + A[i, (i + j + k) \text{mod} n] \times B[(i + j + k) \text{mod} n, j]$. The processor then computes the values of the other product matrix elements assigned to it. To compute each value of C, the processors read $2(n - n/\sqrt{p})$ values from non-local memories and $2n/\sqrt{p}$ values locally. The time taken for computing each value of the resulting matrix is $nt_c + 2t_{local} + 2t_{local}n/\sqrt{p} + 2t_{nonlocal}(n - n/\sqrt{p})$. The parallel runtime of this formulation is

$$T_P = \frac{n^3}{p}t_c + 2\left(n - \frac{n}{\sqrt{p}}\right) \times \frac{n^2}{p}t_{nonlocal} + \frac{n}{\sqrt{p}} \times \frac{n^2}{p}t_{local}.$$

28 In this algorithm, each processor first receives all the data it needs to compute its n^2/p elements and then

performs the computation. The total amount of non-local data required by a processor is $2n^2/\sqrt{p} - 2n^2/p$. The time taken to compute all n^2/p elements assigned to a processor is

$$T_P = \frac{n^3}{p}t_c + 2t_{local} + 2t_{local}\frac{n^2}{p} + 2t_{nonlocal}(2\frac{n^2}{\sqrt{p}} - 2\frac{n^2}{p})$$

For large values of n and p, this expression becomes

$$T_P = \frac{n^3}{p}t_c + 2\frac{n^2}{\sqrt{p}}t_{nonlocal}.$$

The relative performance of the two formulations of Problems 8.27 (page 377) and 8.28 (page 377) becomes obvious when we compute the speedups obtained from the two. The speedup from the previous formulation (Problem 5.34) is

$$S = \frac{p}{1 + 2t_{nonlocal}/t_c}.$$

¿From the above expression, the algorithm's maximum speedup is determined by the ratio $2t_{nonlocal}/t_c$. Typically the time for a non-local memory access $t_{nonlocal}$ is much higher than t_c; consequently, the speedup may be poor. The seedup of the latter formulation is

$$S = \frac{p}{1 + \sqrt{p}/n \times 2t_{nonlocal}/t_c}.$$

As the size of the matrices increase for a fixed value of p, the contribution of essential computation grows faster than that of the overhead. It is thus possible to obtain parallel run times close to optimal by solving larger problem instances. Furthermore, the speedup approaches p when $n/p \gg 2t_{nonlocal}/t_c$. This algorithm is therefore better than the first.

29 Problems 8.23 (page 376)–8.28 (page 377) illustrate the inadequacy of PRAM models for algorithm design. They fail to model the communication costs of parallel algorithms on practical computers. Consequently, an algorithm designed the PRAM model may exhibit very poor performance, even on well connected practical architectures like shared address space computers. In general, for an architecture in which the cost of non-local data access is more expensive than local data access, maximizing locality is critical. All practical architectures fall into this category.

Furthermore, these problems also illustrate that shared address space computers should in fact be programmed with a view to maximizing data locality, much the same way message passing computers are programmed. Ignoring data locality in a shared address space computer leads to poor performance.

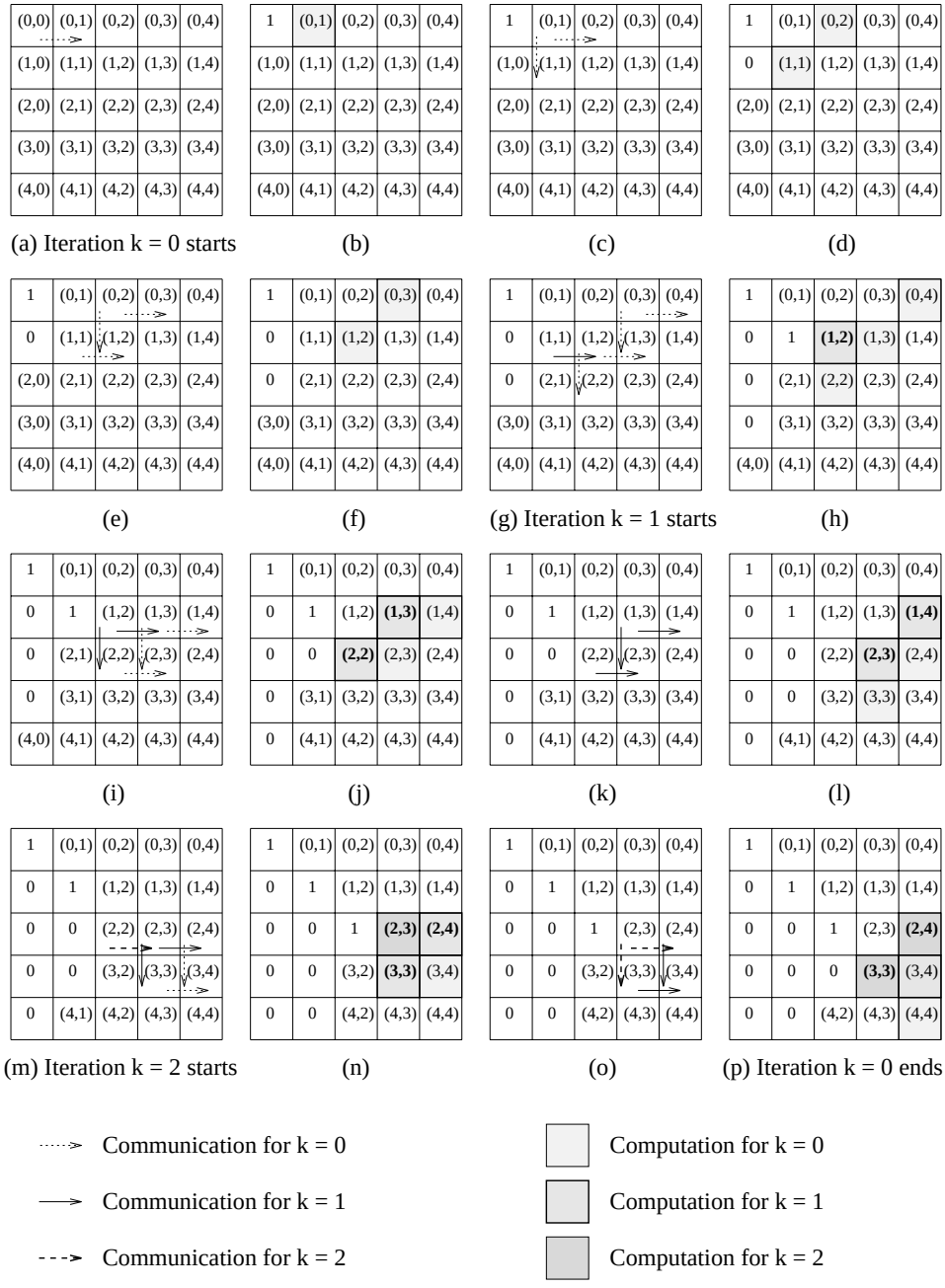

Figure 8.1 Pipelined Cholesky factorization on a 5 $\times$ 5 matrix on 25 processors.

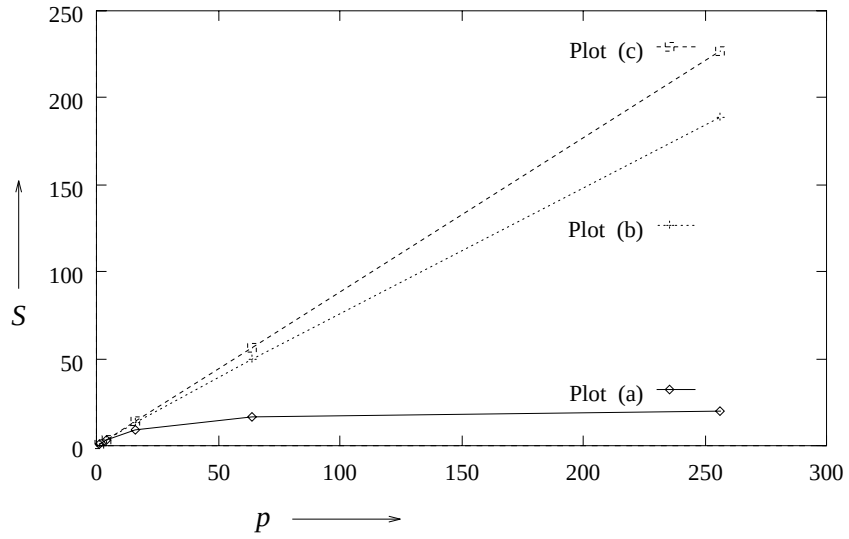

Figure 8.2 Standard, scaled, and isoefficient speedup curves for Problems 8.17 (page 375) and 8.18 (page 375).

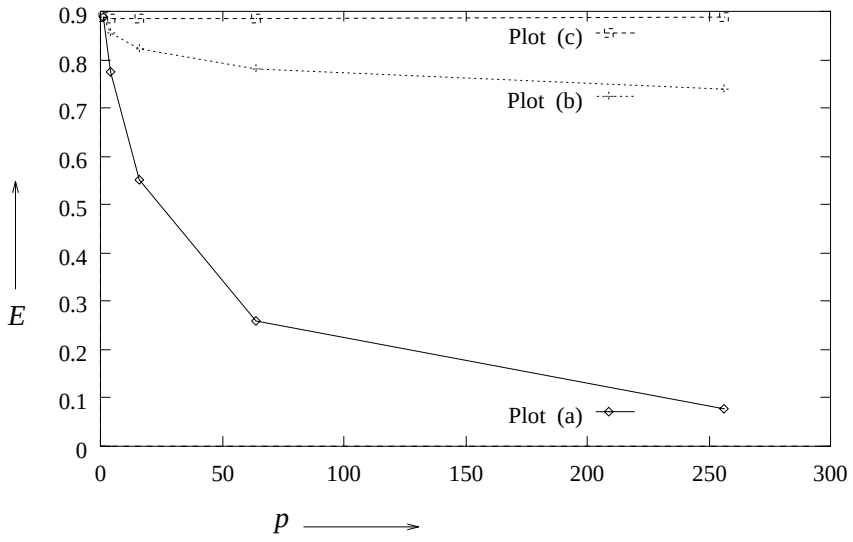

Figure 8.3 Efficiencies corresponding to the speedups in Figure 8.2.

CHAPTER 9

Sorting

1 The correctness of this compare-split operation can be proved using induction on the number of compare-exchange operations. Processor P_i has a sequence of k elements $x = \langle x_1, x_2, \ldots, x_k \rangle$ sorted in increasing order and processor P_j has k elements $y = \langle y_1, y_2, \ldots, y_k \rangle$ sorted in decreasing order. During the nth step of the compare-split operation processors P_i and P_j exchange their nth elements. Processor P_i keeps the $\max\{x_n, y_n\}$ and P_j keeps $\min\{x_n, y_n\}$. The inductive proof is as follows:

1. After the first step of the compare-split operation, P_i holds $z = \max\{x_1, y_1\}$ and P_j holds $w = \min\{x_1, y_1\}$. For the operation to be correct z must be larger than at least k elements of the combined sequences x and y, and w must be smaller than at least k elements of the combined sequence x and y. Since sequence x is sorted in increasing order, we know that x_1 is smaller than the remaining $k-1$ elements of sequence x. Since $w = \min\{x_1, y_1\}$, w is either x_1 or y_1. If $w = x_1$, then $x_1 < y_1$; thus, y_1 and $x_2, x_3, \ldots, x_k$ are the k elements that are larger than w. If $w = y_1$, then the elements of sequence x are the k elements that are larger than w. Therefore w is correctly assigned to processor P_j. Similarly we can show that z is also correctly assigned to processor P_i.
2. Assume that up to step $n-1$ of the bitonic-split operation, elements were correctly assigned to processors P_i and P_j.
3. We need to prove that $w = \min\{x_n, y_n\}$, and $z = \max\{x_n, y_n\}$ are assigned to processors P_i and P_j correctly. Again, consider the case where $w = x_n$. w is smaller than the remaining $k-n$ elements of sequence x. Also, w is smaller than y_n. However, since sequence y is sorted in decreasing order, y is smaller than the first $n-1$ elements of sequence y. Thus, there are k elements that are larger than w. If $w = y_n$, then w is smaller than the $n-k+1$ elements of x (that is, $x_n, x_{n+1}, \ldots, x_k$) and also is smaller than the previous $n-1$ elements of y (that is, $y_1, y_2, \ldots, y_{n-1}$). Therefore, in either case, w is correctly assigned to P_j. Similarly, it can be shown that z is assigned to P_i correctly.

As mentioned in the problem, the compare-split operation can terminate as soon as $x_n \geq y_n$. The reason is that $x_i \geq y_i, \forall i > n$, as the x sequence is sorted in increasing order and the y sequence is sorted in decreasing order. Consequently, if the compare-exchange operations are performed, all elements $x_1 \; \forall i > n$ go to processor P_i and all the y_i elements $\forall i > n$ go to processor P_j.
Because, the compare-exchanges can stop as soon as $x_n \geq y_n$, the run time of this operation varies. If we have to sort the sequences after each compare-split operation, the run time of this formulation will be $\Theta(k \log k)$. However, at the end of the compare split operation, processor P_i has two subsequences, one from x and one from y. Furthermore, each of these subsequences is sorted. Therefore, they can be merged in linear time to obtained the final sorted sequence of size k. An advantage of this operation is that requires no extra memory.
Fox *et al.* [fox] provide further details of this compare-split operation.

Table 9.1 Communication requirements of row-major mapping.

1st iteration	1																			
2nd iteration	1	+	2																	
3rd iteration	1	+	2	+	4															
⋮																				
$(d/2)$th iteration	1	+	2	+	4	+	$\cdots$	+	$2^{d/2-1}$											
$(d/2+1)$st iteration	1	+	2	+	4	+	$\cdots$	+	$2^{d/2-1}$	+	1									
$(d/2+2)$nd iteration	1	+	2	+	4	+	$\cdots$	+	$2^{d/2-1}$	+	1	+	2							
$(d/2+3)$rd iteration	1	+	2	+	4	+	$\cdots$	+	$2^{d/2-1}$	+	1	+	2	+	4					
⋮																				
dth iteration	1	+	2	+	4	+	$\cdots$	+	$2^{d/2-1}$	+	1	+	2	+	4	+	$\cdots$	+	$2^{d/2-1}$	

2 A relation R on A is

- ***reflexive*** if for every $a \in A$ $(a, a) \in R$.
- ***antisymmetric*** if for every $a, b \in A$ with $(a, b) \in R$ and $(b, a) \in R$, then $a = b$.
- ***transitive*** if for every $a, b, c \in A$ whenever $(a, b) \in R$, and $(b, c) \in R$, then $(a, c) \in R$.

From the above definitions the fact that $\leq$ is a partial ordering can be easily shown.

3 To prove (a) note that there is an element a_i $(0 < i < n/2)$ of sequence s such that for all $j < i$, $\min\{a_j, a_{n/2+j}\}$ belongs to $\{a_0, a_1, \ldots, a_{n/2-1}\}$ and for all $i < j < n/2$ $\min\{a_j, a_{n/2+j}\}$ belongs to $\{a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}\}$. Similarly, for all $j < i$, $\max\{a_j, a_{n/2+j}\}$ belongs to $\{a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}\}$ and for all $n/2 > j > i$, $\max\{a_j, a_{n/2+j}\}$ belongs to $\{a_0, a_1, \ldots, a_{n/2-1}\}$. Therefore,

$$s_1 = \{a_0, a_1, \ldots, a_i, a_{n/2+i+1}, \ldots, a_{n-1}\}$$

and

$$s_2 = \{a_{n/2}, a_{n/2+1}, \ldots, a_{n/2+i}, a_{i+1}, \ldots, a_{n/2-1}\}.$$

Note that both s_1 and s_2 are bitonic sequences.
Parts (b) and (c) can be proved in a similar way. For more detailed proof of the correctness of the bitonic split operation refer to Knuth [Knu73], Jaja [Jaj92], Quinn [Qui87], or Batcher [Bat68].

5 From Figure 9.10 (page 389) we see that bitonic merge operations of sequences of size 2^k take place during the kth stage of the bitonic sort. The elements of each bitonic sequence merged during the kth stage correspond to wires whose binary representation have the same $(\log n - k)$ most significant bits (where n is the number of element to be sorted). Because each wire is mapped to a hypercube processor whose label is the same as that of the wire's, each sequence is mapped onto a k-dimensional subcube. Since the most significant $(\log n - k)$ bits of each bitonic sequence of length 2^k are different, these sequences are mapped onto distinct subcubes.

7 Let n be the number of processors such that $d = \log n$.

(a) From Figure 9.11 (page 391) we can see that the communication requirements are shown in Table 9.1. Let A be the communication required during the first $d/2$ iterations. Then

$$A = \sum_{i=0}^{d/2-1} \sum_{j=0}^{i} 2^j = \sum_{i=0}^{d/2-1} \left(22^i - 1\right) = 2\sqrt{n} - 2 - \frac{1}{2}\log n.$$

Let B be the communication required during the first $d/2$ steps of the last $d/2$ iterations. Then

$$B = \frac{d}{2} \sum_{i=0}^{d/2-1} 2^i = \frac{1}{2}\log n(\sqrt{n} - 1) = \frac{1}{2}\sqrt{n}\log n - \frac{1}{2}\log n.$$

Table 9.2 Communication requirements of snakelike mapping.

1st iteration	1																			
2nd iteration	1	+	2																	
⋮																				
$(d/2)$th iteration	1	+	2	+	$\cdots$	+	$2^{d/2-1}$													
$(d/2+1)$st iteration	1	+	2	+	$\cdots$	+	$2^{d/2-1}$	+	$1+\sqrt{n}$											
$(d/2+2)$nd iteration	1	+	2	+	$\cdots$	+	$2^{d/2-1}$	+	$1+\sqrt{n}$	+	2									
$(d/2+3)$rd iteration	1	+	2	+	$\cdots$	+	$2^{d/2-1}$	+	$1+\sqrt{n}$	+	2	+	$4+\sqrt{n}$							
⋮																				
dth iteration	1	+	2	+	$\cdots$	+	$2^{d/2-1}$	+	$1+\sqrt{n}$	+	2	+	$4+\sqrt{n}$	+	$\cdots$	+	$2^{d/2-1}$			

Table 9.3 Communication requirements of the row-major shuffled mapping.

	1		2		3		4		5		6		$\ldots$		$d-1$		d
1st iteration	1																
2nd iteration	1	+	1														
3rd iteration	1	+	1	+	2												
4th iteration	1	+	1	+	2	+	2										
5th iteration	1	+	1	+	2	+	2	+	4								
6th iteration	1	+	1	+	2	+	2	+	4	+	4						
⋮																	
$(d/2-1)$th iteration	1	+	1	+	2	+	2	+	4	+	4	+	$\cdots$	+	$2^{d/2-1}$		
dth iteration	1	+	1	+	2	+	2	+	4	+	4	+	$\cdots$	+	$2^{d/2-1}$	+	$2^{d/2-1}$

The total communication performed by the row-major formulation of the bitonic sort is

$$B + 2A = \frac{1}{2}\sqrt{n}\log n + 4\sqrt{n} - \frac{3}{2}\log n - 4.$$

(b) From Figure 9.11 (page 391) we can see that the communication requirements are shown in Table 9.2. Note that the communication requirements for the first $d/2$ iterations are similar to the row-major mapping. However, during the last $d/2$ iterations, the communication requirements of snakelike mapping is higher than that of the row-major mapping. The additional communication is approximately

$$\sqrt{n}2\sum_{i=1}^{d/4} i \approx \frac{\sqrt{n}\log^2 n}{16}.$$

Therefore, the total communication requirements is

$$\frac{\sqrt{n}\log^2 n}{16} + \frac{1}{2}\sqrt{n}\log n + 4\sqrt{n} - \frac{3}{2}\log n - 4.$$

(c) The communication requirements of the row-major shuffled mapping is shown in Table 9.3. To derive the total communication required we first concentrate on the odd numbered columns. Note that the sum of the odd numbered columns is

$$2\sum_{i=0}^{d/2-1}\sum_{j=0}^{i} 2^i = 4\sqrt{n} - \log n - 4.$$

Now note that in Table 9.3, column two differs from column one by 1, column four differs from column two by 2 and so on. Therefore, the sum of the even numbered columns is equal to that of the odd

numbered columns minus

$$\sum_{i=0}^{d/2-1} 2^i = \sqrt{n} - 1.$$

Therefore, the total communication requirements of row-major shuffled mapping is

$$2(4\sqrt{n} - \log n - 4) - \sqrt{n} + 1 = 7\sqrt{n} - 2\log n - 7.$$

Therefore, row-major shuffled mapping requires less communication than the other two mappings. However, Nassimi and Sahni [NS79] have developed variants of bitonic sorting for the row-major and the snakelike mapping that have the same communication requirement as the row-major shuffled mapping.

8 The solution is presented by Knuth [Knu73] (merging network theorem).

9 In a ring-connected parallel computer, input wires are mapped onto processors in a way similar to the hypercube mapping. The wire with label a is mapped onto processor with label a.
The total amount of communication performed by the processors is:

$$\sum_{j=0}^{\log n-1} \sum_{i=0}^{j} 2^i = 2n - \log n - 2 \approx 2n$$

The parallel run time, speedup, and efficiency when p processors are used to sort n elements are:

$$\begin{aligned} T_P &= \Theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \Theta\left(\frac{n}{p}\log^2 p\right) + \Theta(n) \\ S &= \frac{\Theta(n\log n)}{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \Theta\left(\frac{n}{p}\log^2 p\right) + \Theta(n)} \\ E &= \frac{1}{1 - \Theta\left(\frac{\log p}{\log n}\right) + \Theta\left(\frac{\log^2 p}{\log n}\right) + \Theta\left(\frac{p}{\log n}\right)} \end{aligned}$$

For this formulation to be cost optimal $p/\log n = O(1)$. Thus, this formulation can efficiently utilize up to $p = \Theta(\log n)$ processors. The isoefficiency function is $\Theta(p2^p)$.

10 This proof is similar to that presented in Solution 8.

11 Recall that in the hypercube formulation of shellsort, each processor performs compare-split operations with each of its neighbors. In a d-dimensional hypercube, each processors has d neighbors. One way of applying the spirit of shellsort to a mesh-connected parallel computer is for each processor to perform compare-split operations with each of its four neighboring processors. The power of shellsort lies in the fact that during the first few iterations, elements move significant closer to their final destination. However, this is not the case with the mesh formulation of shellsort. In this formulation elements move at most $\sqrt{p}$ processors.
An alternate way of performing shellsort on a mesh-connected computer is shown in Figure 9.1. The mesh is divided in to equal halves along the y-dimension, and corresponding processors compare-exchange their elements. Then, the mesh is halved and the same is repeated for the smaller meshes. This is applied recursively until communication takes place between adjacent mesh rows. A similar sequence of compare-exchanges is performed along the x-dimension. Note that the processors paired-up for compare-split are similar to those paired-up in the hypercube formulation of shellsort.

12 The sequential shellsort algorithm consists of a number of iterations. For each iteration i there is an associated distance d_i. The n elements to be sorted are divided into groups, such that each group consists of elements whose indices are multiples of d_i. These groups are then sorted. In each iteration d_i decreases. During the last step, $d_i = 1$, and the algorithms sorts the entire list. Shellsort can be parallelized by performing each iteration in parallel. In order to take advantage of the fact that as the sort progresses sequences tend to become almost sorted, a sorting algorithm such as the odd-even transposition must be used.

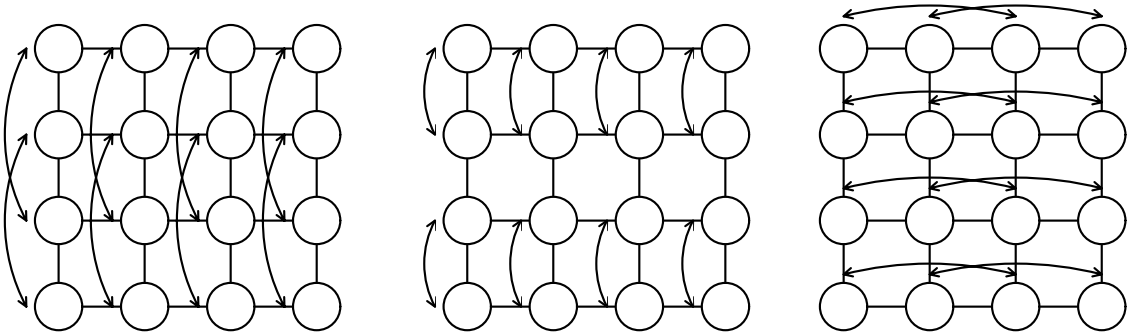

Figure 9.1 An example of performing shellsort on a mesh with 16 processors.

13 [fox] The worst case can be summarized as follows. Let n be the number of elements to be sorted on a d-dimensional hypercube. Each processors is assigned $n/2^d = m$ elements. Now if more than m of the largest $2m$ elements occupy a particular pair of processors in the first compare-split step, some of the items will be put in the wrong subcube in this first step. The subsequent $(d-1)$ stages will just rearrange the items in each $(d-1)$-dimensional subcube, but these elements will stay in the wrong subcube. Consequently, in the final odd-even transposition sort, these items must travel a long distance to reach the top of the list, requiring $2^{d-1}-1$ steps. The probability that such a collection of items will occur, for random initial lists, is given by:

$$\left(\frac{1}{2^{d-1}}\right)^{m-1} \frac{(2m)!}{m!m!}$$

This probability becomes small rapidly as m increases.

14 The CREW formulation of quicksort that assigns each subproblem to a different processor can be extended for a p processor system by stopping the recursive subdivision as soon as p subproblems have been created. At this point, each of the p processors internally sorts one of the p subsequences. At the end of this step, the n elements are sorted.

If we assume that the pivot selection leads to perfectly balanced splits, then, if $n = 2^d$ and $p = 2^k$, the parallel run time is:

$$\frac{n}{p}\log\frac{n}{p} + \sum_{i=0}^{k-1}\frac{n}{2^i} = \frac{n}{p}\log\frac{n}{p} + 2n(1-\frac{1}{p}).$$

The efficiency of this formulation is:

$$\begin{aligned} E &= \frac{n\log n}{n\log n - n\log p + 2np - 2n} \\ &= \frac{1}{1 - \log p/\log n + 2p/\log n - 2/\log p} \end{aligned}$$

From this equation, we see that for a cost efficient formulation $p = O(\log n)$. Therefore, this formulation cannot use more that $O(\log n)$ processors. The isoefficiency function is $\Theta(p2^p)$.

15 The algorithm is described by Chlebus and Vrto [CV91].

16 The expected height of the tree is less than or equal to $3\log n + 6$. The analysis is discussed by Chlebus and Vrto [CV91].

17 The solution is provided in the first edition of this book.

18 The proof that the hypercube formulation of quicksort correctly sorts n elements on p processors, is as follows. After the first split, $p/2$ processors will end up having elements greater than the pivot while the remaining $p/2$ processors have elements smaller than the pivot. In particular, the processors whose processor label has

a zero at the most significant bit get the smaller, while the other processors get the larger elements. Thus, the first split, puts the smaller elements in the right half-cube.

Assume that after k splits, the elements are sorted according to the k most significant bits of the processors' labels. That is, if we treat each subcube of size 2^{d-k} as a single processor, and the elements assigned to this subcube, as a block of elements, then these blocks are sorted. Now, during the $(k+1)$st split each block is partitioned into two smaller blocks. In each subcube of size 2^{d-k}, each of these two smaller blocks is assigned to one of the two 2^{d-k-1} half-cubes contained into a 2^{d-k} subcube. This assignment is such that the block with the smaller elements is assigned to the half-cube that has a zero at the $(d-k-1)$st bit, while the block with larger elements is assigned to the half-cube that has a one. Therefore, the label of the half-cube determines the ordering. Furthermore, since this is a local rearrangement of the elements in each 2^{d-k} subcube, the global ordering has not changed.

20 This pivot selection scheme was proposed by Fox *et al.*, [fox]. The performance of the algorithm depends on the sample size l; the larger the sample the better the pivot selection. However, if the sample is large, then the algorithm spends a significant amount of time in selecting the pivots. Since l cannot be too large, a good choice seems to be $l = \Theta(\log n)$. Furthermore, if the distribution of elements on each processor is fairly uniform, this pivot selection scheme leads to an efficient algorithm.

21 This problem was motivated by the $\Theta(n)$ worst case selection algorithm is described in [CLR89]. Following a proof similar to that in [CLR89] it can be shown that this scheme selects a pivot element such that there are $\Theta(n)$ elements smaller than the pivot, and $\Theta(n)$ elements larger than the pivot.

Initially, the sequences stored in each processor are sorted. For each processor pair, computing the median of the combined sequences takes $\Theta(\log n)$ time. Sorting 2^{i-1} medians can be done using bitonic sort, which takes $\Theta(i^2)$ time. Finally, the median of the median is broadcasted in $\Theta(i)$ time.

22 The barrier synchronization is not necessary. Removing it does not affect neither the correctness nor the asymptotic performance.

23 The parallel run time of the hypercube formulation of quicksort if we include the t_c, t_s, and t_w costs is approximately

$$T_P = \frac{n}{p}\log\frac{n}{p}t_c + \left(\frac{n}{p}t_w + t_s\right)\log p + \log^2 p(t_s + t_w)$$

The efficiency is:

$$\begin{aligned} E &= \frac{n\log nt_c}{n\log nt_c - n\log pt_c + (nt_w + pt_s)\log p + p\log^2 p(t_s+t_w)} \\ &= \left(1 - \frac{\log p}{\log n} + \frac{\log p}{\log n}\frac{t_w}{t_c} + \frac{p\log p}{n\log n}\frac{t_s}{t_c} + \frac{p\log^2 p}{n\log n}\left(\frac{t_s+t_w}{t_c}\right)\right)^{-1} \end{aligned} \tag{9.1}$$

If $K = E/(1-E)$, then from Equation 9.1 we have that the isoefficiency function due to the $(\log p/\log n)(t_w/t_c)$ term is

$$\frac{\log p}{\log n}\frac{t_w}{t_c} = \frac{1}{K} \Rightarrow \log n = \frac{t_w}{t_c}K\log p \Rightarrow n = 2^{(Kt_w/t_c)\log p} = p^{Kt_w/t_c} \Rightarrow n\log n = \frac{t_w}{t_c}Kp^{Kt_w/t_c}\log p.$$

The isoefficiency function due to the $(p\log p)/(n\log n)(t_s/t_c)$ term is $K(t_s/t_c)p\log p$, due to the $(p\log^2 p)/(n\log n)(t_s/t_c)$ term is $K(t_s/t_c)p\log^2 p$, and due to the $(p\log^2 p)/(n\log n)(t_w/t_c)$ term is $K(t_w/t_c)p\log^2 p$. Therefore, the most significant isoefficiency functions are:

$$\frac{t_w}{t_c}Kp^{Kt_w/t_c}\log p, \quad \frac{t_w}{t_c}Kp\log^2 p, \quad \text{and} \quad \frac{t_s}{t_c}Kp\log^2 p.$$

Note that the asymptotic complexity of the first term depends on the value of K and the t_w/t_c ratio.

(a) $t_c = 1, t_w = 1, t_s = 1$

$E = 0.5 \quad K = E/(1-E) = 0.5/(1-0.5) = 1.$
The isoefficiency functions are: $p \log p$, $p \log^2 p$, and $p \log^2 p$. Therefore, the overall isoefficiency function is $p \log^2 p$.

$E = 0.75 \quad K = 0.75/(1-0.75) = 3.$
The isoefficiency functions are: $3p^3 \log p$, $3p \log^2 p$, and $3p \log^2 p$. Therefore, the overall isoefficiency function is $3p^3 \log p$.

$E = 0.95 \quad K = 0.95/(1-0.95) = 19.$
The isoefficiency functions are: $19p^{19} \log p$, $19p \log^2 p$, and $19p \log^2 p$. Therefore, the overall isoefficiency function is $19p^{19} \log p$.

(b) $t_c = 1, t_w = 1, t_s = 10$

$E = 0.5 \quad K = E/(1-E) = 0.5/(1-0.5) = 1.$
The isoefficiency functions are: $p \log p$, $p \log^2 p$, and $10p \log^2 p$. Therefore, the overall isoefficiency function is $10p \log^2 p$.

$E = 0.75 \quad K = 0.75/(1-0.75) = 3.$
The isoefficiency functions are: $3p^3 \log p$, $3p \log^2 p$, and $30p \log^2 p$. Therefore, the overall isoefficiency function is $3p^3 \log p$.

$E = 0.95 \quad K = 0.95/(1-0.95) = 19.$
The isoefficiency functions are: $19p^{19} \log p$, $19p \log^2 p$, and $190p \log^2 p$. Therefore, the overall isoefficiency function is $19p^{19} \log p$.

(c) $t_c = 1, t_w = 10, t_s = 100$

$E = 0.5 \quad K = E/(1-E) = 0.5/(1-0.5) = 1.$
The isoefficiency functions are: $10p^{10} \log p$, $10p \log^2 p$, and $100p \log^2 p$. Therefore, the overall isoefficiency function is $10p^{10} \log p$.

$E = 0.75 \quad K = 0.75/(1-0.75) = 3.$
The isoefficiency functions are: $30p^{30} \log p$, $30p \log^2 p$, and $300p \log^2 p$. Therefore, the overall isoefficiency function is $30p^{30} \log p$.

$E = 0.95 \quad K = 0.95/(1-0.95) = 19.$
The isoefficiency functions are: $190p^{190} \log p$, $190p \log^2 p$, and $1900p \log^2 p$. Therefore, the overall isoefficiency function is $190p^{190} \log p$.

Therefore, as the efficiency increases, the hypercube formulation becomes less scalable. The scalability also decreases, as the t_w/t_c ratio increases.

24 The solution is provided in the first edition of this book.

25 [SKAT91] Let n be the number of elements to be sorted on a p-processor mesh-connected computer. Each processor is assigned n/p elements. In order to extend this algorithm to apply to the case of multiple elements per processor, the following changes have to be made:

(a) One of the elements in the first processor is chosen as the pivot. Pivot distribution remains the same. This step takes $\Theta(\sqrt{p})$ time, since the worst case distance traveled in the vertical and horizontal directions is $\sqrt{p}$ hops.

(b) On receiving the pivot, each processor divides its elements into sets of elements smaller and larger than the pivot. Also, it maintains the number elements in each of these two sets. Information propagated from the leaves of the binary tree to the root takes into account the number of smaller and larger elements at each node. Since there are n/p elements per processor, it takes $\Theta(n/p)$ time to divide the set. Propagating the information back takes $\Theta(\sqrt{p})$ time.

(c) The information propagated from the root to the leaves about the next free processor is modified to account for multiple elements per processors. In particular, the two partitions are separated at a processor boundary. Therefore, the number of elements per processor may differ somewhat in the two partitions.

The next free processor is specified along with the number of elements that can be added to the processors. This takes $\Theta(\sqrt{p})$ time.

(d) Elements are moved in large messages. Notice that a message from one processor may have to be split across two destination processors. Then the portion for the other destination processor can be sent in one hop. This takes $\Theta(n/\sqrt{p})$ time.

Since we assume perfect pivot selection, the algorithm requires $\Theta(\log p)$ iterations, the parallel run time is:

$$T_P = \frac{n}{p}\log\frac{n}{p} + \left(\Theta(\sqrt{p}) + \Theta\left(\frac{n}{p}\right) + \Theta\left(\frac{n}{\sqrt{p}}\right)\right)\Theta(\log p)$$

The isoefficiency function of the algorithm is $\Theta(2^{\sqrt{p}\log p}\sqrt{p}\log p)$.

26 The CRCW PRAM algorithm can be easily adapted to other architectures by emulation.

CREW PRAM The main difference between a CREW and a CRCW PRAM architecture is the ability to perform a concurrent write. However, a concurrent write operation performed by n processors can be emulated in a CREW PRAM in $\Theta(\log n)$ time. Therefore, the parallel run time of the enumeration sort is $\Theta(\log n)$.

EREW PRAM In this model, in addition to requiring $\Theta(\log n)$ time to emulate a concurrent write operation, it needs a $\Theta(\log n)$ time to perform a concurrent read operation. Therefore, the parallel run time of the enumeration sort is $\Theta(\log n)$.

Hypercube The hypercube formulation is similar to the EREW PRAM formulation. Each read or write operation takes $\Theta(\log n)$ time. Also, the elements can be permuted to their final destination in $\Theta(\log n)$ time, since n^2 processors are used. Therefore, the parallel run time of the enumeration sort is $\Theta(\log n)$.

Mesh In a mesh-connected computer we can emulate a concurrent write or read operation in $\Theta(\sqrt{n})$ time. Therefore, the parallel run time of the enumeration sort is $\Theta(\sqrt{n})$.

When p processors are used, we assign n/p elements to each processor. Each processor now performs computations for each of its elements, as if a single element is assigned to it. Thus, each physical processor emulates n/p virtual processors. This results in a slowdown by a factor of n/p.

27 Since the sequential run time is $\Theta(n)$, the speedup and efficiency are:

$$\begin{aligned} S &= \frac{\Theta(n)}{\Theta(n/p) + \Theta(p\log p)} \\ E &= \frac{\Theta(n)}{\Theta(n) + \Theta(p^2\log p)} \\ &= \frac{1}{1 + \Theta((p^2\log p)/n)} \end{aligned}$$

Therefore, the isoefficiency function of bucket sort is $\Theta(p^2\log p)$.
Comparing the isoefficiency function of bucket sort with the other sorting algorithms presented in the chapter, we see that bucket sort has better isoefficiency function than all but the quicksort algorithm for the hypercube. Recall that the bucket sort algorithm assumes that data is uniformly distributed. Under the same assumption the hypercube formulation of quicksort has good performance, and its isoefficiency functions is $\Theta(p\log^2 p)$. Note also, that the isoefficiency function of bucket sort is similar to that of sample sort, under the assumption of uniform distribution. However, if the distribution is not uniform, then the parallel run time of bucket sort becomes

$$T_P = \Theta\left(\frac{n}{p}\right) + \Theta(n) + \Theta(p\log p).$$

Since the processor-time product is $\Theta(np)$, bucket sort is cost optimal only when $p = \Theta(1)$.

28 The proof is presented by Shi and Schaeffer [SS90].

29 When the sizes of the subblocks are roughly the same, the speedup and efficiency of sample sort are:

$$
\begin{aligned}
S &= \frac{\Theta(n \log n)}{\Theta((n/p)\log(n/p)) + \Theta(p^2 \log p) + \Theta(p \log(n/p)) + \Theta(n/p) + \Theta(p \log p)} \\
E &= \frac{\Theta(n \log n)}{\Theta(n \log(n/p)) + \Theta(p^3 \log p) + \Theta(p^2 \log(n/p)) + \Theta(n) + \Theta(p^2 \log p)} \\
&= \left(1 + \Theta\left(\frac{\log p}{\log n}\right) + \Theta\left(\frac{p^3 \log p}{n \log n}\right) + \Theta\left(\frac{p^2}{n}\right) + \Theta\left(\frac{p^2 \log p}{n \log n}\right) + \Theta\left(\frac{1}{\log n}\right)\right)^{-1}
\end{aligned}
$$

The isoefficiency function is $\Theta(p^3 \log p)$.
When the size of the subblocks can vary by a factor of $\log p$, then there is a processor that has a block of size $n/p + \log p$. The time spent in communication the is $\Theta(n/p + \log p) + O(p \log p)$. However, this new term will only add a $(p \log p)/(n \log n)$ term in the denominator of the efficiency function. Therefore, it does not affect the asymptotic scalability.

30 The algorithm can be easily modified to sort the sample of $p(p-1)$ elements in parallel using bitonic sort. This can be done by using the hypercube formulation of bitonic sort presented in Section 9.2.2 (page 392). Each processor has $(p-1)$ elements; thus, sorting the $p(p-1)$ elements takes $\Theta(p \log^2 p)$ time.
At the end of the bitonic sort, each processor stores $(p-1)$ elements. To select the $(p-1)$ splitter elements, all the processors but the last one (that is, processor P_{p-1}), select their $(p-1)$st element. These selected elements are the $(p-1)$ splitters. The splitters are send to all the processors by performing an all-to-all broadcast operation (Section 4.2 (page 157)). This operation takes $\Theta(p)$ time.
Assuming a uniform distribution of data, the parallel run time is:

$$
T_P = \Theta\left(\frac{n}{p} \log \frac{n}{p}\right) + \Theta(p \log^2 p) + \Theta\left(p \log \frac{n}{p}\right) + \Theta(n/p) + O(p \log p)
$$

The isoefficiency function of this formulation is $\Theta(p^2 \log p)$.

31 Recall that the parallel runtime is

$$
T_P = \frac{b}{r} 2^r (\Theta(\log n) + \Theta(n)) \tag{9.2}
$$

The optimal value of r is such that it minimizes Equation 9.2. This value can be found if we compute the exact parallel run time of the algorithm, using t_c, t_w, and t_s, and then differentiate the obtained parallel run time. By setting the derivative to zero and solving for r we obtain the value that minimizes the run time.

32 Let n be the number of elements to be sorted using p processors. We can use the radix sort algorithm described in Algorithm 9.8 (page 416) to perform the task, by using virtual processors. This is done by emulating n/p processors on each physical processor. The run time of this formulation is slower, at most, by a factor of n/p. However, the procedures *parallel_sum* and *prefix_sum* can be optimized to use the fact that each processor stores n/p elements. Similarly, the communication step can be optimized taking into account that n/p processors are assigned to each physical processor.
These optimizations are described by Blelloch *et al.* [BLM+91] that is based on ideas by Cole and Vishkin [CV86].

CHAPTER 10

Graph Algorithms

1 The parallel run time of Prim's algorithm on a hypercube is

$$T_P = \Theta\left(\frac{n^2}{p}\right) + \Theta(n \log p). \tag{10.1}$$

If $p = \Theta(n)$, the parallel run time is $\Theta(n \log n)$ and is determined by the second term of the above equation. The minimum parallel run time can be obtained by differentiating Equation 10.1 with respect to p, setting the derivative equal to zero, and solving for p as follows:

$$\Theta\left(-\frac{n^2}{p^2}\right) + \Theta\left(\frac{n}{p}\right) = 0 \Rightarrow p = \Theta(n)$$

Therefore, the minimum run time is obtained when $p = \Theta(n)$ and is

$$T_P = \Theta\left(\frac{n^2}{n}\right) + \Theta(n \log n) = \Theta(n) + \Theta(n \log n) = \Theta(n \log n). \tag{10.2}$$

However, if we use $\Theta(n/\log n)$ processors, then the parallel run time is

$$T_P = \Theta(n \log n) + \Theta(n \log n - n \log \log n) = \Theta(n \log n). \tag{10.3}$$

From Equations 10.2 and Equation 10.3 we see that the parallel run times obtained using $\Theta(n)$ and $\Theta(n/\log n)$ processors are the same in asymptotic terms. However, the exact run time when $\Theta(n/\log n)$ processors are used is larger than the one obtained when $p = n$, by a constant factor.

2 Dijkstra's single-source shortest paths algorithm can record the shortest path by using an additional array *path*, such that *path*[v] stores the predecessor of v in the shortest path to v. This array can be updated by modifying Line 12 of Algorithm 10.2 (page 437). Whenever $l[u] + w(u, v)$ is smaller than $l[v]$ in Line 12, *path*[v] is set to u. This modification does not alter the performance of the algorithm.

3 Let $G = (V, E)$ be a graph. For each edge $(u, v) \in E$ let $w(u, v) = 1$. The breadth-first ranking can be obtained by applying Dijkstra's single-source shortest path algorithm starting at node u on graph $G = (V, E, W)$. The parallel formulation of this algorithm is similar to that of Dijkstra's algorithm.

4 The solution is discussed by Bertsekas and Tsitsiklis [BT89].

5 The sequential all-pairs shortest paths algorithm requires $\Theta(n^2)$ memory. This memory is used to store the weighted adjacency matrix of $G = (V, E)$. The source-partitioning formulation of Dijkstra's algorithm requires $\Theta(n^2)$ memory on each processor, and the source-parallel formulation requires $\Theta(n^3/p)$ memory on

		Dijkstra's Algorithm		Floyd's Algorithm	
	Matrix-Multiplication Based Algorithm	Source-Partitioning-Formulation	Source-Parallel-Formulation	Block-Checkerboard Formulation	Pipelined Formulation
Memory per Processor	$\Theta(n^2/p)$	$\Theta(n^2)$	$\Theta(n^3/p)$	$\Theta(n^2/p)$	$\Theta(n^2/p)$
Memory Overhead	$\Theta(1)$	$\Theta(p)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Table 10.1 Memory overhead of the all-pairs shortest paths algorithm presented in Section 10.4 (page 437). Memory overhead is defined as the ratio of the memory required by all the p processors with the memory required by the sequential algorithm.

each processor. The reason is that in the source-parallel formulation, there are n groups of p/n processors, each running a copy of Dijkstra's single-source shortest path. For Floyd's algorithm (both the 2D block-mapping and its pipelined variant) requires $\Theta(n^2/p)$ memory per processor. Table 10.1 summarizes the memory requirements, and the memory overhead factors of each formulation. For further details refer to Kumar and Singh [KS91].

6 Replacing Line 7 of Algorithm 10.3 (page 440) by

$$d_{i,j} := \min\{d_{i,j}, (d_{i,k} + d_{k,j})\},$$

we perform the updates of the D matrix in place, without requiring any additional memory for the various $D^{(k)}$ matrices. The replacement is correct because during the kth iteration, the values of $d_{i,k}^{(k-1)}$ and $d_{k,j}^{(k-1)}$ are the same as $d_{i,k}^{(k)}$ and $d_{k,j}^{(k)}$. This is because the operation performed at Line 7, does not change these values.

7 Recall from Section 10.4.2 (page 441) that in each iteration of Floyd's algorithm, the algorithm performs two broadcasts, one along the rows and one along the columns. Each broadcast requires a message of size $\Theta(n/\sqrt{p})$ to be sent to all the processors in a row or column.
On a mesh with store-and-forward routing, each broadcast operation takes $\Theta(n)$ time, and on a mesh with cut-through routing, it takes $\Theta((n \log p)/\sqrt{p} + \sqrt{p})$ time. Since the algorithm requires n iterations, the parallel run time on a mesh with store-and-forward routing is

$$T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta(n^2),$$

and on a mesh with cut-through routing is

$$T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta\left(\frac{n^2}{\sqrt{p}} \log p + n\sqrt{p}\right).$$

The efficiency of the formulation on a mesh with store-and-forward routing is

$$E = \frac{1}{1 + \Theta(p/n)}.$$

Therefore, the isoefficiency function is $\Theta(p^3)$.
The efficiency of the formulation on a mesh with cut-through routing is

$$E = \frac{1}{1 + \Theta((\sqrt{p} \log p)/n) + \Theta(p^{1.5}/n^2)}.$$

The isoefficiency function due to the first term is $\Theta(p^{1.5} \log^3 p)$ and the isoefficiency function due to the second term is $\Theta(p^{2.25})$. Thus, the overall isoefficiency function is $\Theta(p^{2.25})$.

8 (a) In each iteration of Floyd's algorithm, each processor needs to know the $d_{i,k}^{(k)}$ and $d_{k,j}^{(k)}$ values to compute $d_{i,j}^{(k+1)}$. By partitioning the $D^{(k)}$ matrix in a block-striped fashion, the $d_{k,j}^{(k)}$ values are stored locally in

each processor. However, the $d_{i,k}^{(k)}$ values needs to be obtained by the processor that stores the kth column of the $D^{(k)}$ matrix. Therefore, in each iteration of the block-striped formulation, the processor that has the kth column needs to broadcast it to all the other processors. Since each column contains n elements, this operation takes $\Theta(n \log p)$ time on a hypercube-connected parallel computer. Therefore, the parallel run time is

$$T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta(n^2 \log p).$$

The isoefficiency function due to communication is $\Theta((p \log p)^3)$, which is the overall isoefficiency function. The isoefficiency function of the block-striped formulation is worse than that of the 2D block formulation, which is $\Theta(p^{1.5} \log^3 p)$. Therefore, this formulation is less scalable, and has no advantages over the 2D block partitioning.

(b) On a mesh with store-and-forward routing, it takes $\Theta(n\sqrt{p})$ time to broadcast a column. Therefore, the parallel run time is

$$T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta(n^2\sqrt{p}).$$

Consequently, the isoefficiency function is $\Theta(p^{4.5})$.

On a mesh with cut-through routing, the column broadcast takes $\Theta(n \log p + \sqrt{p})$ time. Therefore, the parallel run time is

$$T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta(n^2 \log p) + \Theta(n\sqrt{p}).$$

The isoefficiency function due to the first communication term is $\Theta((p \log p)^3)$ and due to the second communication term is $\Theta(p^{2.25})$. Therefore, the overall isoefficiency function is $\Theta((p \log p)^3)$.

On a ring with store-and-forward routing, the column broadcast takes $\Theta(np)$ time. The parallel run time is

$$T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta(n^2 p).$$

The isoefficiency function is $\Theta(p^6)$. The same column broadcast takes $\Theta(n \log p + p)$ time on a ring with cut-through routing. The parallel run time is

$$T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta(n^2 \log p) + \Theta(np).$$

The isoefficiency function is $\Theta((p \log p)^3)$. Thus block-striped partitioning has similar performance on a ring with CT routing and on a mesh with CT routing.

9 The pipelined formulation of the block-striped partitioning proceeds along the same lines as that with the 2D block partitioning. As soon as the processor containing the kth column has finished the $(k-1)$st iteration, it sends the $(k-1)$st column to its two neighbor processors. When a processor receives elements of the $(k-1)$st column it stores them locally and then forwards them to its other neighbor. A processor starts working on the kth iteration as soon as it has computed the $(k-1)$st iteration and has received the $(k-1)$st columns.
It takes $\Theta(np)$ time for the first column to reach processor P_p. After that each subsequent column follows after $\Theta(n^2/p)$ time in a pipelined mode. Hence, processor P_p finishes its share of the shortest path computation in $\Theta(n^3/p) + \Theta(np)$ time. When processor P_p has finished the $(n-1)$st iteration, it sends the relevant values of the nth column to the other processors. These values reach processor P_1 after $\Theta(np)$ time. The overall parallel run time is

$$T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta(np).$$

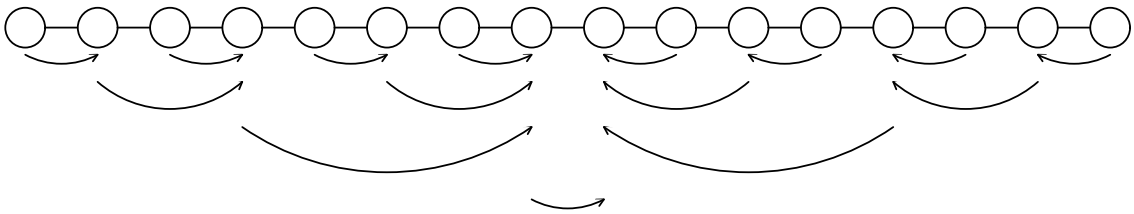

Figure 10.1 Merging the spanning forests along a row of a mesh. Each arrow indicates the direction of data movement. In this example, each row has 16 processors.

The efficiency is

$$E = \frac{1}{1 + \Theta(p^2/n^2)}.$$

Therefore, the isoefficiency function is $\Theta(p^3)$.

10 The analysis is presented by Kumar and Singh [KS91]. The exact parallel run time is

$$T_P = \frac{n^3}{p} t_c + 4(\sqrt{p} - 1)\left(t_s + \frac{n}{\sqrt{p}} t_w\right), \tag{10.4}$$

where t_c is the time to perform the operation in Line 7 of Algorithm 10.3 (page 440), t_s is the message startup time, and t_w is the per-word transfer time. Note that Equation 10.4 is valid under the following two assumptions (a) $n^2 t_c/p \approx t_s + n t_w/\sqrt{p}$ and (b) when a processor transmits data it is blocked for t_s time; after that it can proceed with its computations while the message is being transmitted.

11 The main step in deriving a parallel formulation of the connected components algorithm for a mesh-connected parallel computer is finding an efficient way to merge the spanning forests. One possible way is to first merge the spanning forests along each row of the mesh. At the end of this step, a column of mesh processors stores the merged spanning forests of each row. The global spanning forest is found by merging these spanning forests. This is accomplished by performing a merge along a single column of the mesh.

One possible way of merging the spanning forests in each row is shown in Figure 10.1. Notice that in each but the last step, the distance among processors that need to communicate doubles.

Recall that in order for a pair of processors to merge their spanning forests, one processor must send $\Theta(n)$ edges to the other processor. In a mesh with store-and-forward routing in order to merge the spanning forests along a row, the total time spent in sending edges is

$$\Theta(n)\left(\sum_{i=0}^{\log(\sqrt{p}/4)} 2^i + 1\right) = \Theta(n\sqrt{p}).$$

Similarly, the time to perform the merge along the column is also $\Theta(n\sqrt{p})$. Therefore, the parallel run time of the algorithm is

$$T_P = \Theta\left(\frac{n^2}{p}\right) + \Theta(n\sqrt{p}).$$

The speedup and efficiency are:

$$\begin{aligned} S &= \frac{\Theta(n^2)}{\Theta(n^2/p) + \Theta(n\sqrt{p})}, \\ E &= \frac{1}{1 + \Theta(p^{1.5}/n)}. \end{aligned}$$

Therefore, the isoefficiency function is $\Theta(p^3)$.

In a mesh with cut-through routing, the time spent merging the spanning forests along each row is:

$$\Theta(n \log p) + \sum_{i=0}^{\log(\sqrt{p}/4)} 2^i = \Theta(n \log p) + \Theta(\sqrt{p}).$$

The parallel run time is

$$T_P = \Theta\left(\frac{n^2}{p}\right) + \Theta(n \log p) + \Theta(\sqrt{p}).$$

The speedup and efficiency are:

$$\begin{aligned} S &= \frac{\Theta(n^2)}{\Theta(n^2/p) + \Theta(n \log p) + \Theta(\sqrt{p})}, \\ E &= \frac{1}{1 + \Theta((p \log p)/n) + \Theta((p\sqrt{p})/n^2)}. \end{aligned}$$

The isoefficiency function is $\Theta(p^2 \log^2 p)$.

Note that the isoefficiency function for a mesh with cut-through routing is the same as that for a hypercube.

12 This formulation is analyzed by Woo and Sahni [WS89]. The performance of this formulation is similar to the block-striped partition.

13 Consider a sparse graph $G = (V, E)$ with average out degree of m. That is, there is on the average m edges (u, v) for each vertex u. Recall that in each step of Johnson's algorithm, a vertex with minimum l value is extracted from the priority queue and its adjacency list is traversed. The traversal of the adjacency list may require updating the priority queue. These updates correspond to Line 12 of Algorithm 10.5 (page 455). On an average, m priority queue updates are performed in each iteration. Therefore, during each iteration of Johnson's algorithm, at most $p_2 = m$ processors can be used to compute new l values.

The priority queue can be maintained using a scheme proposed by Kwan and Ruzzo [KR84]. According to this scheme, m updates are done in $\Theta((m \log n)/p)$ time on a CREW PRAM where $p \leq m \leq n$. Note that n is the number of elements in the priority queue and p is the number of processors used. Therefore, at most $p_1 = m$ processors can be used to maintain the priority queue.

Note that computing updated values of l and updating the priority queue is done one after the other. Therefore, instead of using p_1 and p_2 processors, we can only use $p = \max\{p_1, p_2\}$ processors. Since both p_1 and p_2 can be at most m, the maximum number of processors that can be used is m.

The parallel run time of the formulation on a CREW PRAM is

$$T_P = \Theta\left(\frac{nm \log n}{p}\right) + \Theta\left(\frac{nm}{p}\right).$$

Since the processor-time product is $\Theta(nm \log n) = \Theta(|E| \log n)$, this formulation is cost-optimal.

14 Recall that in each iteration of Dijkstra's single-source shortest path, a vertex u with the smallest l-value is found, and its adjacency list is processed. Dijkstra's algorithm is shown in Algorithm 10.2 (page 437). Furthermore, let m be the average out-degree of the vertices.

First consider the case where n/p vertices and their adjacency lists are assigned to each processor. Furthermore, assume that each processor also stores the l-values for its assigned vertices. In each iteration, the algorithm spends $\Theta(n/p) + \Theta(\log p)$ time finding a vertex u with minimum $l[u]$ value. Let P_i be the processor that has u. In order for Line 12 of Algorithm 10.2 (page 437) to be executed, processor P_i sends $w(u, v)$ and $l[u]$ to the processor that has v. Assuming that the destinations of the messages are distinct, this operation is similar to one-to-many personalized communication. The time required is $\Theta(\log p + m)$. Therefore, the parallel run time for n iterations is

$$T_P = \Theta\left(\frac{n^2}{p}\right) + \Theta(n \log p) + \Theta(nm + n \log p).$$

Since the sequential complexity of Dijkstra's algorithm is $\Theta(n^2)$, the speedup and efficiency are as follows:

$$S = \frac{\Theta(n^2)}{\Theta(n^2/p) + \Theta(n \log p) + \Theta(nm + n \log p)}$$
$$E = \frac{1}{1 + \Theta((p \log p)/n) + \Theta(mp/n + (p \log p)/n)}$$

For a cost optimal formulation, $\Theta(p \log p) = \Theta(n)$ and $\Theta(mp) = \Theta(n)$. Therefore, this formulation can use $\min\{\Theta(n/m), \Theta(n/\log n)\}$ processors.

Now consider the case where each processor is assigned m/p elements from each adjacency list. Assume that the l-values of the vertices are distributed among the p processors, so each processor has n/p l-values. Finding the vertex u with the minimum value of l takes $\Theta(n/p) + \Theta(\log p)$ time. In order for the l-values to be updated for each vertex v adjacent to u, the processor responsible for $l[v]$ must know $w(u, v)$. Each processor has m/p elements of the adjacency list of u. These elements need to be send to different processors. Therefore each processor spends at least $\Theta(m \log p/p)$ time. Thus, in the worse case the overall time spent in communication is $\Theta(m \log p)$. Therefore, the overall run time for n iterations is

$$T_P = \Theta\left((\frac{n^2}{p}\right) + \Theta(nm \log p).$$

Since the sequential complexity is $\Theta(n^2)$, this formulation is cost-optimal. However, this formulation can use only $p = \min\{n/(m \log n), m\}$ processors.

Note that Dijkstra's algorithm has the same complexity for both sparse and dense graphs. The operation that determines the complexity is finding the vertex with the minimum value of l.

15 The solution to this problem is similar to Solution 14.

17 If we ignore overhead due to extra work, we essentially make the assumption that computation progresses in a wave fashion. That is, at any time, only the vertices belonging to a diagonal of the grid graph are performing computations. The average length of each diagonal is $n/2$ vertices, and there are a total of $(2n - 1)$ diagonals. In the 2D cyclic mapping, each diagonal of vertices is mapped onto a diagonal of the $\sqrt{p} \times \sqrt{p}$ processor grid. This diagonal of processors is wrapped-around in such a way that at any time, at most $\sqrt{p}$ processors are performing computation. This can be verified by looking at Figure 10.2.

Since there are an average of $n/2$ vertices in each diagonal and $\sqrt{p}$ processors assigned to it, the shortest path computation performed by these processors takes $\Theta(n/(2\sqrt{p})) = \Theta(n/\sqrt{p})$ time. However, each shortest path computation requires sending the data to neighbor processors. Therefore, after computing each diagonal, the algorithm spends $\Theta(n/\sqrt{p})$ time sending path information. Therefore, the overall run time is

$$T_P = \Theta\left(\frac{n^2}{\sqrt{p}}\right) + \Theta\left(\frac{n^2}{\sqrt{p}}\right).$$

The processor-time product of this formulation is $\Theta(n^2\sqrt{p})$; therefore, this algorithm is not scalable.

18 Let b be the block size of the 2D block-cyclic-mapping. As in Solution 17, each diagonal has $n/2$ vertices. Each diagonal now can intersect up to $2\sqrt{p}$ processors. Therefore, the algorithm spends $\Theta(n/\sqrt{p})$ time for each diagonal. Since the block size is b, each of the $2\sqrt{p}$ processors has approximately $\Theta(n/(b\sqrt{p}))$ blocks assigned to it. Each processor needs to send path information for only the boundary vertices of each block, therefore the communication is $\Theta(n/(b\sqrt{p}))$. The parallel run time of this formulation is

$$T_P = \Theta\left(\frac{n^2}{\sqrt{p}}\right) + \Theta\left(\frac{n^2}{b\sqrt{p}}\right).$$

The processor-time product is $\Theta(n^2\sqrt{p})$; thus, this formulation is not cost-optimal.

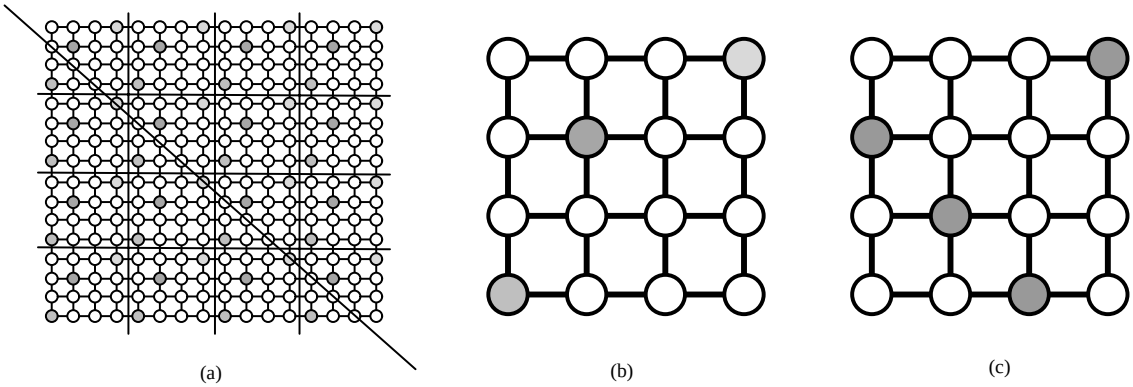

Figure 10.2 (a) A grid graph, (b) Mapping the grid graph onto a processor grid using 2D cyclic mapping. (c) The processors intersected by the diagonal in (a). Note that each diagonal intersects at most $\sqrt{p}$ processors.

19 The solution to this problem is along the lines of Solution 17.

20 The amount of extra computation performed depends on the grid graph. In particular, if the shortest paths from the source to any other vertex follow more or less a straight path then the amount of extra computation is minimal. However, if the shortest paths are snakelike, then the amount of extra computation is significant. Experimental results obtained by Wada and Ichiyoshi [WI89] suggest that the 2D block and the 1D block mappings perform less extra computation than the 2D cyclic and the 2D block-cyclic mappings.

21 The parallel formulation of Sollin's algorithm is described by Leighton [Lei92b].

CHAPTER 11

Search Algorithms for Discrete Optimization Problems

1 The value of $V(p)$ for the GRR-M scheme is identical to that of the GRR scheme. However, the number of accesses to target are reduced drastically in this case. Each request now incurs a delay of time δ at each of the $\log p$ processors in the path. Therefore, there is a delay time of $O(\delta \log p)$ associated with each request. This does not change the overall time, since the time to transfer work across the network is also $\Theta(\log p)$. The isoefficiency function due to communication overhead is given by $O(V(p) \log p \log W)$, or $O(p \log p \log W)$. Equating this with the essential work W, we have the isoefficiency function of this scheme to be $O(p \log^2 p)$. Since there is no contention in this load balancing scheme, the overall isoefficiency is given by $O(p \log^2 p)$. Kumar, Ananth, and Rao [KGR91] discuss this scheme in detail and provide experimental evaluation.

2 This load balancing technique uses a distributed scheme for implementing the counter. The step in the counter is used to locate the processor to be requested for work. Each time a request is made, the step moves to the right. In this way, after p requests, the step returns to the processor it started at. The step in the value of counters can be detected in $\Theta(\log p)$ using the algorithm presented by Lin [Lin92].
This algorithm can be used in conjunction with the load balancing strategy described in the problem. $V(p)$ for this scheme is p. Therefore, the communication overhead is given by $O(p \log p \log W)$. Equating this with the essential work W, we can determine the isoefficiency of this scheme to be $O(p \log^2 p)$.
The isoefficiency term resulting from the number of messages required to balance load is the same as that of GRR (and GRR-M), which is $O(p^2 \log p)$. Therefore, the overall isoefficiency of this scheme on a network of workstations is $O(p^2 \log p)$.
Lin [Lin92] discusses this result in detail.

3 **Size of message grows as $\sqrt{w}$:** The contribution to the overall isoefficiency due to contention remains the same and can be obtained by equating

$$W/p = O(V(p) \log W)$$

This yields an isoefficiency of $O(p^2 \log p)$.
There are a total of $V(p) \log W$ messages. Of these, in the first $V(p)$ messages, the maximum work transferred is W. In the second $V(p)$ messages, the maximum work transferred is $(1-\alpha)W$. Each of these messages may potentially be communicated over a distance of $\log p$. Therefore the total communication overhead is given

by

$$T_{comm} = O(\sum_{i=0}^{\log W-1} V(p)\sqrt{(1-\alpha)^i W}\log p)$$

For GRR, $V(p) = p$. If $\sqrt{(1-\alpha)} = r$, this expression reduces to

$$T_{comm} = O(p\sqrt{W}\log p \sum_{i=0}^{\log W-1} r^i)$$

$$T_{comm} = O(p\sqrt{W}\log p \frac{(1-r^{\log W-1})}{1-r})$$

Since $r < 1$, this expression reduces to

$$T_{comm} = O(p\sqrt{W}\log p)$$

Equating this overhead with essential computation W, we get

$$W = O(p^2 \log^2 p)$$

Therefore the overall isoefficiency is now $O(p^2 \log^2 p)$.

4 The run time of a termination detection algorithm is considered as the time taken by an algorithm to signal termination after the last processor has finished its computation. Since the token may need to traverse the ring once after all processors have gone idle, in the worst case, the run time of the termination algorithm is $\Theta(p)$. Since all processors have to spend this time, the total overhead due to termination detection is $\Theta(p^2)$.
The constant associated with this isoefficiency term is very small. Therefore, unless the value of p is very large, the contribution of termination detection to overall isoefficiency can be neglected.

6 The solution is discussed in detail by Mahapatra and Dutt [DM93].

7 The solution is discussed in detail by Kimura and Nobuyuki [KI90]. Ignoring communication latency, they show that the isoefficiency function of the single level load balancing scheme is given by $O(p^2)$.

8 The solution is discussed in detail by Kimura and Nobuyuki [KI90]. Ignoring communication latency, they show that the isoefficiency function of the multi-level load balancing scheme is given by $O(p^{(l+1)/l}(\log p)^{(l+1)/l})$, where l is the number of levels.

9 Ferguson and Korf [FK88] discuss this scheme in detail.

10 Let the total number of nodes to be expanded at a certain moment by the search algorithm be r and number of processors with at least one node in its local open list be X_r. As shown by Manzini [MS90], $E[X_r] = \Theta(p)$ as r becomes $\Theta(p)$ (which is a reasonable assumption).
In each iteration, $\Theta(p)$ nodes are expanded and the corresponding communication overhead is $\Theta(p)$. This corresponds to an isoefficiency function of $\Theta(p)$. Since there are no other overheads, the overall isoefficiency of this scheme is given by $\Theta(p)$.
The analysis considers only the number of nodes and not the quality of nodes expanded. The quality of the nodes can be analyzed in a similar manner. Manzini [MS90] discusses these results in detail. Dutt and Mahapatra [MD93] experimentally evaluate this scheme and propose improved variants.

11 Each node in the parallel best-first search formulation is hashed to a random processor. The time taken to perform this communication is given by $O(m + \log p)$ for a message of size m. Since there are W/p such hash operations, the total communication time is given by $O(Wm/p + n/p \log p)$. The total communication overhead is given by $O(Wm + W \log p)$. Equating this with the total essential work, which is $\Theta(W)$, we can see that the formulation is unscalable. This is because no matter how fast the problem size W is increased, the efficiency cannot be held constant.
The formulation is scalable only for architectures for which the cost of communicating a node across is $O(1)$; e.g., on PRAMs.

CHAPTER 12

Dynamic Programming

1 The parallel run time of the algorithm is given by

$$T_P = n(t_c \frac{c}{p} + 2t_s + t_w \frac{c}{p})$$

The corresponding speedup is given by

$$\begin{aligned} S &= \frac{T_1}{T_P} = \frac{nct_c}{n(t_c \frac{c}{p} + 2t_s + t_w \frac{c}{p})} \\ &= \frac{1}{\frac{1}{p} + \frac{2t_s}{ct_c} + \frac{t_w}{pt_c}} \end{aligned}$$

The efficiency of the parallel formulation is given by

$$E = \frac{1}{1 + \frac{2pt_s}{ct_c} + \frac{t_w}{t_c}}$$

Now, as we increase the problem size (by increasing c), all terms in the denominator reduce except $1 + t_w/t_c$. Therefore, $1 + t_w/t_c$ is the lower bound on the value of the denominator. Therefore, the efficiency is bounded from above by

$$E = \frac{1}{1 + \frac{t_w}{t_c}}$$

2 The solution is discussed by Lee and Sahni in [LSS88].

3 **NOTE**

The hint in the problem should point to block-cyclic striped mapping and not cyclic striped mapping.

In the cyclic striped mapping shown in Figure 12.1, in the computation of the first diagonal, one processor is busy; in the computation of the second diagonal, two processors are busy; and so on. This continues until diagonal p. The computation of each of these diagonals takes time $t_c + t_s + t_w$. This is because a single word needs to be communicated from the processor on its left and a single entry needs to be computed. Therefore the total time for computing these p diagonals is $p(t_c + t_s + t_w)$. The computation of each of the next p diagonals takes time $2(t_c + t_s + t_w)$ since some of the processors may be computing two entries in the table. The total time taken for these p diagonals is therefore $2p(t_c + t_s + t_w)$. Similarly, the computation of the next p diagonals takes time $3(t_c + t_s + t_w)$ and the total time is $3p(t_c + t_s + t_w)$. The time for the p diagonals leading up to the longest diagonal is given by $n/p(t_c + t_s + t_w)$. The total parallel run time of the algorithm is

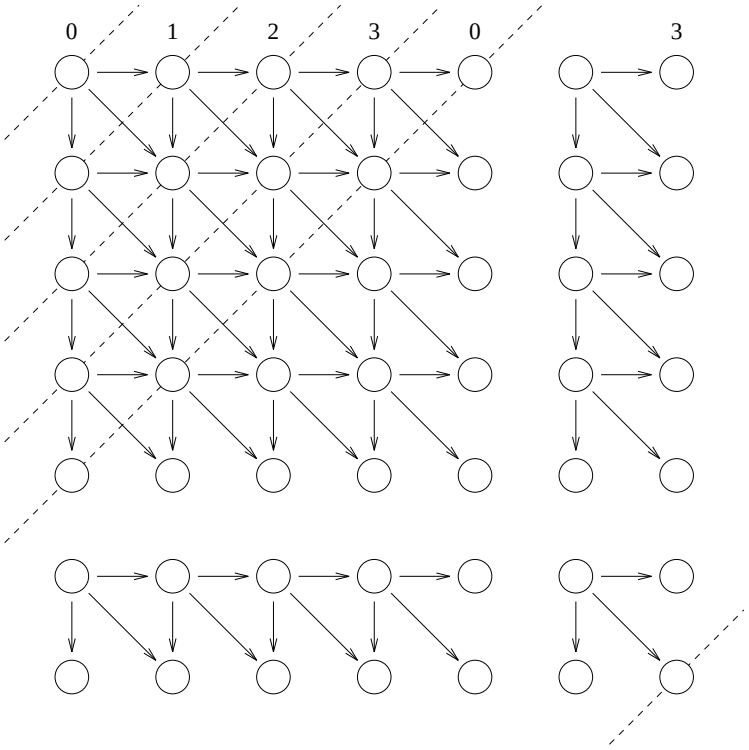

Figure 12.1 Cyclic partitioning of the LSC problem for $p = 4$.

twice the time taken to compute the first n diagonals. This is given by

$$\begin{aligned} T_P &= 2p(t_c + t_s + t_w)\sum_{i=1}^{n/p} i \\ &= p(t_c + t_s + t_w)\frac{n}{p}(\frac{n}{p} + 1) \\ &= (t_c + t_s + t_w)n(\frac{n}{p} + 1) \\ &= (t_c + t_s + t_w)\frac{n^2}{p} + (t_c + t_s + t_w)n \end{aligned}$$

The corresponding speedup and efficiency are given by

$$\begin{aligned} S &= \frac{n^2 t_c}{(t_c + t_s + t_w)\frac{n^2}{p} + (t_c + t_s + t_w)n} \\ E &= \frac{1}{1 + \frac{t_s + t_w}{t_c} + \frac{p}{n}(t_c + t_s + t_w)} \end{aligned}$$

We can see that on increasing the problem size (by increasing n), the efficiency increases. However, it is bounded from above by

$$E = \frac{1}{1 + \frac{t_s + t_w}{t_c}}$$

This upper bound can be removed by using block-cyclic striped mapping. In this case, processors are assigned $\sqrt{p}$ columns in a cyclic manner. Therefore, the first $\sqrt{p}$ columns are assigned to processor P_0, the next $\sqrt{p}$

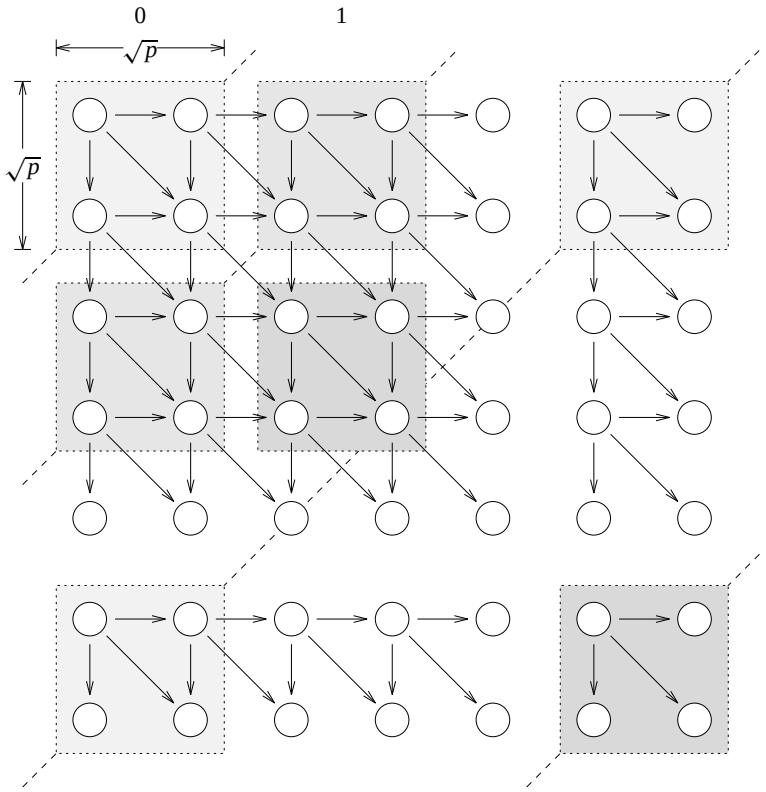

Figure 12.2 Block cyclic partitioning of the LSC problem. The nodes are organized into blocks of size $\sqrt{p} \times \sqrt{p}$ and assigned to the processors in a cyclic fashion.

columns to processor P_1 and so on. The assignment wraps back to processor P_0 after all processors have been assigned $\sqrt{p}$ columns. Computation is performed in blocks of size $\sqrt{p} \times \sqrt{p}$ as shown in Figure 12.2. The computation of the block at the top left corner is performed first. After this block has been processed, two blocks can be processed. One by processor P_0 and the other by processor P_1. Using this mapping scheme, the computation of the first block is done by processor 0 in time $(\sqrt{p} \times \sqrt{p})t_c$. The next set of two blocks is computed by processors P_0 and P_1 in time $(t_s + t_w\sqrt{p}) + pt_c$. This continues until all processors become busy (that is, until diagonal $p^{1.5}$). For the next $p^{1.5}$ diagonals, computing each block takes time $2((t_s+t_w\sqrt{p})+pt_c)$. This continues and the longest diagonal. The total parallel run-time of the algorithm is twice the time for the computation of the first n diagonals. This is given by

$$\begin{aligned} T_P &= 2((t_s + t_w\sqrt{p}) + pt_c)p \sum_{i=1}^{n/(p^{1.5})} i \\ &= ((t_s + t_w\sqrt{p}) + pt_c)p\frac{n}{p^{1.5}}(\frac{n}{p^{1.5}} + 1) \\ &= t_s(\frac{n}{\sqrt{p}} + \frac{n^2}{p^2}) + t_w(\frac{n^2}{p^{1.5}} + n) + t_c(\frac{n^2}{p} + n\sqrt{p}) \end{aligned}$$

The corresponding speedup and efficiency are given by

$$\begin{aligned} S &= \frac{n^2 t_c}{t_s(\frac{n}{\sqrt{p}} + \frac{n^2}{p^2}) + t_w(\frac{n^2}{p^{1.5}} + n) + t_c(\frac{n^2}{p} + n\sqrt{p})} \\ E &= \frac{1}{1 + \frac{p^{1.5}}{n} + \frac{t_w}{t_c}(\frac{1}{\sqrt{p}} + \frac{p}{n} + \frac{t_s}{t_c}(\frac{\sqrt{p}}{n} + \frac{1}{p})} \end{aligned}$$

From this expression, we can see that the efficiency does not have an upper bound.

4 The TSP can be solved by constructing a table which stores values of $f(S,k)$ for increasingly larger sets S. Starting at city v_1, we can construct $n-1$ sets of two cities with the second city being the terminating city. Each of these $n-1$ sets now leads to $n-2$ sets of 3 cities, with the added city being the terminating city. In general, at the k^{th} level, there are $(n-1)^{n-2}C_k$ nodes (since the initial and terminating cities are fixed). The total number of entires to be computed, N, is given by:

$$N = \sum_{k=0}^{n-2} (n-1)^{n-2} C_k = (n-1)2^{n-2}$$

Since computing an entry in the $(n-2)_{\text{nd}}$ row takes time $(n-1)t_c$, the serial complexity of this algorithm is $\Theta(n^2 2^n)$.

The formulation is serial monadic, but is considerably harder to parallelize than the standard multistage graph procedure. This is because the average number of nodes at a level is $O(2^n)$, but each node is connected to a small number of nodes ($O(n)$) at the next level. A simple parallel formulation of this algorithm is given below. The formulation uses only $O(n)$ processors, but it uses them efficiently.

The table can be constructed in parallel by assigning the responsibility of one terminating city to each processor. Therefore, using $n-1$ processors, processor i in step k computes all paths through nodes in set k terminating at node i. After each step, an all-to-all broadcast is used to communicate all the results from the current step to all the processors. Therefore, in step k, an all to all broadcast of ${}^{n-2}C_k$ elements is required. The corresponding computation for the step is given by $\Theta(k^{n-2}C_k)$. All-to-all broadcast takes time $t_s \log p + t_w m(p-1)$. The first term is dominated by the second and can be ignored without significant loss in accuracy. The all-to-all broadcast operation can be performed in approximately $t_w mp$ time. Since $m = {}^{n-2}C_k$ and $p = n-1$, the time taken is given by $t_w n^{n-2} C_k$.

The total parallel time is therefore given by

$$T_P = \sum_{k=0}^{n-2} k^{n-2} C_k t_c + \sum_{k=0}^{n-2} t_w n^{n-2} C_k$$

and the speedup is given by

$$S = \frac{\sum_{k=0}^{n-2} (n-1)k^{n-2} C_k}{\sum_{k=0}^{n-2} k^{n-2} C_k t_c + \sum_{k=0}^{n-2} t_w n^{n-2} C_k}$$

This corresponds to $T_P = \Theta(n2^n)$. Since $\Theta(n)$ processors are used, the formulation is cost optimal. The same formulation gives identical performance on the mesh connected computer because the dominant term in all-to-all broadcast is the same for the mesh.

5 Let $c(k)$ be the number of records in file k and $f(S)$ be the cost of merging the set of files S. Furthermore, let uv represent the file of length $c(u)+c(v)$ resulting from the merger of files u and v. The recursive equation for solving the optimal merge problem is given by

$$f(S) = \begin{cases} 0 & S = \{\} \text{ or } |S| = 1 \\ \min\limits_{u,v \in S} \{ f(S - \{u\} - \{v\} + \{uv\}) + c(u) + c(v) \} & \end{cases}$$

However, a better greedy formulation of the algorithm is represented by the following recursive equation

$$f(S) = \begin{cases} 0 & S = \{\} \text{ or } |S| = 1 \\ f(S - \{u\} - \{v\} + \{uv\}) + \min\limits_{u \in S} c(u) + \min\limits_{v \in S - \{u\}} c(v) & \end{cases}$$

The serial algorithm requires the two smallest files to be determined at each step. This is based on the heap data structure or on explicit determination of the smallest files. Parallel formulations of the algorithm can be derived based on the technique used to determine the smallest files.

7 Given a polygon $\langle v_0, v_1, \ldots, v_{n-1} \rangle$, the optimal triangulation problem can be solved using the following DP formulation: define $C[i, j]$ as the weight of an optimal triangulation of vertices $\langle v_{i-1}, \ldots, v_j \rangle$. The objective is to determine $C[1, n-1]$. The following recursive equation can be used to determine $C[i, j]$:

$$C[i, j] = \begin{cases} \min\limits_{i \le k \le j} \{C[i, k] + C[k+1, j] + f(v_{i-1}, v_k, v_j)\} & i < j \\ 0 & i = j \end{cases} \tag{12.1}$$

Here, $f(v_{i-1}, v_k, v_j)$ is a weight function corresponding to the triangle formed by vertices v_{i-1}, v_k, and v_j. Using the perimeter of the triangle as the weight function,

$$f(v_{i-1}, v_k, v_j) = |v_{i-1}v_k| + |v_k v_j| + |v_j v_{i-1}|$$

Comparing this with the recursive DP formulation for the matrix parenthesization problem, we can see that the two are identical. It is a nonserial polyadic DP formulation and parallel formulations identical to the optimal matrix parenthesization problem can be used here.
Cormen [CLR90] provides details of the DP formulation to solve this problem.

CHAPTER 13

Fast Fourier Transform

1 (a) This problem assumes a hypothetical parallel computer with a hypercube interconnection network, in which there is no message startup time and a message contains only one word. The parallel run time of the binary exchange algorithm is

$$\begin{aligned} T_P &= \frac{t_c n \log n}{p} + \frac{t_w n \log p}{p} \\ S &= \frac{p}{1 + \frac{t_w \log p}{t_c \log n}} \\ E &= \frac{1}{1 + 1.2\frac{\log p}{\log n}} \end{aligned}$$

(b) For maintaining a fixed efficiency E,

$$\begin{aligned} 1.2\frac{\log p}{\log n} &= \frac{1}{E} - 1 \\ \log n &= \frac{1.2E}{1-E} \log p \\ n &= p^{1.2E/(1-E)} \\ n \log n &= \frac{1.2E}{1-E} p^{1.2E/(1-E)} \log p \end{aligned}$$

For $E = 0.6$,

$$n \log n = 1.8 p^{1.8} \log p$$

(c) For $E = 0.4$,

$$n \log n = 0.8 p^{0.8} \log p$$

Since the lower bound on the isoefficiency function due to concurrency is $p \log p$, the actual isoefficiency function for $E = 0.4$ is $p \log p$.

(d)

$$E = \frac{1}{1 + 2\frac{\log p}{\log n}}$$

For maintaining a fixed efficiency E,

$$n \log n = \frac{2E}{1-E} p^{2E/(1-E)} \log p$$

Table 13.1 The binary representation of the various powers of ω calculated in different iterations of an 16-point FFT (also see Figure 13.1 (page 539)). The value of m refers to the iteration number of the outer loop, and i is the index of the inner loop of Program 13.2 (page 540).

	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$m=0$	0000	0000	0000	0000	0000	0000	0000	0000	1000	1000	1000	1000	1000	1000	1000	1000
$m=1$	0000	0000	0000	0000	1000	1000	1000	1000	0100	0100	0100	0100	1100	1100	1100	1100
$m=2$	0000	0000	1000	1000	0100	0100	1100	1100	0010	0010	1010	1010	0110	0110	1110	1110
$m=3$	0000	1000	0100	1100	0010	1010	0110	1110	0001	1001	0101	1101	0011	1011	0111	1111

For $E = 0.6$,

$$n \log n = 3p^3 \log p$$

For $E = 0.4$,

$$n \log n = \frac{4}{3} p^{4/3} \log p$$

2 Refer to the paper by Thompson [Tho83].

3 On a linear array of p processors, the distance between the communicating processors in the i^{th} iteration (of the $\log p$ iterations that require interprocessor communication) is 2^i $(0 \leq i < \log p)$.

$$\begin{aligned} T_P &= \frac{t_c n \log n}{p} + \sum_{i=0}^{\log p - 1} (t_s + t_w \frac{n}{p} 2^i) \\ &\approx \frac{t_c n \log n}{p} + t_s \log p + t_w n \end{aligned}$$

The isoefficiency function due to the t_w term is $\Theta(p 2^{t_w E p/(t_c(1-E))})$.

4 If $t_s = 0$, the expression for efficiency is as follows:

$$E = \frac{1}{1 + \frac{2t_w\sqrt{p}}{t_c \log n}}$$

In order to maintain a fixed efficiency E,

$$t_c n \log n = 2\frac{E}{1-E} t_w \sqrt{p}$$

Let $E/(1-E) = K$.

$$\begin{aligned} t_c n \log n &= 2K t_w \sqrt{p} \\ n &= 2^{2K t_w \sqrt{p}/t_c} \\ n \log n = W &= 2K \frac{t_w}{t_c} \sqrt{p} 2^{2K t_w \sqrt{p}/t_c} \end{aligned}$$

5 The first equation of the recurrence follows from the fact that there are only n unique twiddle factors that a single processor needs to compute. When $n = p$, all but the first $p/4$ processors compute a different twiddle factor in each of the $\log n = \log p$ iterations. This verifies the second equation of the recurrence.

Table 13.1 shows the binary representation of the powers of ω required for all values of i (inner loop index) and m (outer loop index) for a 16-point FFT. Table 13.2 shows the values of $h(16, p)$ for p = 1, 2, 4, 8, and 16. The table also shows the maximum number of new twiddle factors that any single processor computes in each iteration.

Disregarding the case in which $p = 1$, notice from Tables 13.2 and 13.2 (page 552) that when the number of processors is reduced by a factor of two, an entry $(m, 2p)$ in the tables occupies the position $(m-1, p)$. The

Table 13.2 The maximum number of new powers of ω used by any processor in each iteration of an 8-point FFT computation.

	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$
$m = 0$	2	1	1	1	1
$m = 1$	2	2	1	1	1
$m = 2$	4	4	2	1	1
$m = 3$	8	8	4	2	1
Total = $h(16,p)$	16	15	8	5	4

value of the entry $(\log n - 1,\ p)$ is n/p; that is, except for $p = 1$, all entries in the last row of the tables are n/p. Thus, if the number of processors is changed from $2p$ to p, all but the first entry of column $2p$ are present in column p, but these entries are offset by one row. An entry n/p is added to the last row of column p and the original entry $(0, 2p) = 1$ does not move from column $2p$ to column p. Hence, $h(n,\ p) = h(n, 2p)+n/p-1$, which is the last equation of the recurrence.

$$\begin{aligned}
h(n,\ p) &= h(n, 2p) + \frac{n}{p} - 1 \\
&= h(n, 4p) + \frac{n}{2p} + \frac{n}{p} - 2 \\
&\vdots \quad \vdots \\
&= h(n, n) + \frac{n}{n/2} + \frac{n}{n/4} + \cdots + \frac{n}{p} - \log(\frac{n}{p}) \\
&= \log n + 2 + 4 + \cdots + \frac{n}{p} - \log n + \log p \\
&= 2\frac{n}{p} - 2 + \log p
\end{aligned}$$

8 Since the computation time is the same in each case, we give the communication times in the following tables. A "—" indicates that the algorithm is not applicable for the given values of n and p. The least run time in each case is boldfaced, and hence, indicates the algorithm of choice.

- $n = 2^{15}$, $p = 2^{12}$:

Communication constants	Runtimes				
	2-D trans.	3-D trans.	4-D trans.	5-D trans.	Bin. ex.
$t_s = 250, t_w = 1$	—	3.15E4	—	8032	**3096**
$t_s = 50, t_w = 1$	—	6316	—	1632	**696**
$t_s = 10, t_w = 1$	—	1276	—	352	**216**
$t_s = 2, t_w = 1$	—	268	—	**96**	120
$t_s = 0, t_w = 1$	—	**16**	—	32	96

- $n = 2^{12}$, $p = 2^{6}$:

Communication constants	Runtimes				
	2-D trans.	3-D trans.	4-D trans.	5-D trans.	Bin. ex.
$t_s = 250, t_w = 1$	1.58E4	3628	2442	—	**1884**
$t_s = 50, t_w = 1$	3214	828	**642**	—	684
$t_s = 10, t_w = 1$	694	**268**	282	—	444
$t_s = 2, t_w = 1$	190	**156**	210	—	396
$t_s = 0, t_w = 1$	**64**	128	192	—	384

- $n = 2^{20}$, $p = 2^{12}$:

Communication constants	Runtimes				
	2-D trans.	3-D trans.	4-D trans.	5-D trans.	Bin. ex.
$t_s = 250, t_w = 1$	1.04E6	—	1.28E4	9024	**6072**
$t_s = 50, t_w = 1$	2.05E5	—	3168	**2624**	3672
$t_s = 10, t_w = 1$	4.12E4	—	**1248**	1344	3192
$t_s = 2, t_w = 1$	8446	—	**864**	1088	3096
$t_s = 0, t_w = 1$	**256**	—	768	1024	3072

9 With a constant t_w, the parallel run time of the binary exchange algorithm on a two-dimensional mesh is approximately $t_c n \log n/p + t_s \log p + 2t_w n/\sqrt{p}$. If the per-word time is t_w/p^x for a p-processor mesh, then the parallel run time is $t_c n \log n/p + t_s \log p + 2t_w n/p^{0.5+x}$. The isoefficiency function due to t_s is $\Theta(p \log p)$. To compute the isoefficiency function due to t_w,

$$\begin{aligned} \frac{t_c n \log n}{p} &\propto \frac{2t_w n}{p^{0.5+x}} \\ \log n &\propto 2\frac{t_w}{t_c} p^{0.5-x} \\ n &\propto 2^{2t_w p^{0.5-x}/t_c} \\ n \log n = W &\propto 2\frac{t_w}{t_c} p^{0.5-x} 2^{2t_w p^{0.5-x}/t_c} \end{aligned}$$

If the channel width is p^x, then in each communication step, at least p^x words must be transferred between any two communicating processors to utilize all links of a channel. Hence,

$$\begin{aligned} \frac{n}{p} &\geq p^x \\ n &\geq p^{1+x} \\ n \log n = W &\geq (1+x) p^{1+x} \log p \end{aligned}$$

Thus, the isoefficiency function due to communication overhead is $\Theta(p^{0.5-x} 2^{2(t_w/t_c)p^{0.5-x}})$ and that due to concurrency is $\Theta(p^{1+x} \log p)$. For $x > 0.5$, the isoefficiency function due to concurrency is greater than $\Theta(p^{1.5} \log p)$ and that due to communication is less than $\Theta(p^{1.5} \log p)$ if all channels are fully utilized. For $x < 0.5$, the isoefficiency function due to communication exceeds $\Theta(p^{1.5} \log p)$. The best isoefficiency function is therefore $\Theta(p^{1.5} \log p)$ for $x = 0.5$. A higher rate of increase of channel width with respect to the number of processors does not improve the overall scalability because the size of data stored at each processor cannot be increased beyond $\Theta(n/p)$ without causing the scalability due to concurrency to deteriorate.

Bibliography

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the 1968 Spring Joint Computer Conference*, 307–314, 1968.

[BLM+91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. C. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine cm-2. Technical report, Thinking Machines Corporation, 1991.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, NJ, 1989.

[CLR89] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, Cambridge, MA, 1989.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, New York, NY, 1990.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, 206–219, 1986.

[CV91] B. S. Chlebus and I. Vrto. Parallel quicksort. *Journal of Parallel and Distributed Processing*, 1991.

[DHvdV93] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel numerical linear algebra. *Acta Numerica*, 111–197, 1993.

[DM93] S. Dutt and N. R. Mahapatra. Parallel A* algorithms and their performance on hypercube multiprocessors. In *Proceedings of the Seventh International Parallel Processing Symposium*, 797–803, 1993.

[FK88] C. Ferguson and R. Korf. Distributed tree search and its application to alpha-beta pruning. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, 1988.

[fox]

[GK93] A. Gupta and V. Kumar. Performance properties of large scale parallel systems. *Journal of Parallel and Distributed Computing*, 19:234–244, 1993. Also available as Technical Report TR 92-32, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[GR88] G. A. Geist and C. H. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM Journal on Scientific and Statistical Computing*, 9(4):639–649, 1988. Also available as Technical Report ORNL/TM-10383, Oak Ridge National Laboratory, Oak Ridge, TN, 1987.

[Jaj92] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[KG91] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. Technical Report TR 91-18, Department of Computer Science Department, University of Minnesota, Minneapolis, MN, 1991. To appear in *Journal of Parallel and Distributed Computing*, 1994. A shorter version appears in *Proceedings of the 1991 International Conference on Supercomputing*, pages 396-405, 1991.

[KGR91] V. Kumar, A. Y. Grama, and V. N. Rao. Scalable load balancing techniques for parallel computers. Technical Report 91-55, Computer Science Department, University of Minnesota, 1991. To appear in *Journal of Distributed and Parallel Computing*, 1994.

[KI90] K. Kimura and N. Ichiyoshi. Probabilistic analysis of the optimal efficiency of the multi-level dynamic load balancing scheme. Technical report, ICOT, 1990.

[Knu73] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[KR84] S. C. Kwan and W. L. Ruzzo. Adaptive parallel algorithms for finding minimum spanning trees. In *Proceedings of the 1984 International Conference on Parallel Processing*, 439–443, 1984.

[KS91] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest path problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, October 1991. A short version appears in the *Proceedings of the International Conference on Parallel Processing*, 1990.

[Lei92a] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, San Mateo, CA, 1992.

[Lei92b] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, San Mateo, CA, 1992.

[Lin92] Z. Lin. A distributed fair polling scheme applied to or-parallel logic programming. *International Journal of Parallel Programming*, 20(4), August 1992.

[LSS88] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing*, (5):438–456, 1988.

[MD93] N. R. Mahapatra and S. Dutt. Scalable duplicate pruning strategies for parallel A* graph search. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, 1993.

[MS90] G. Manzini and M. Somalvico. Probabilistic performance analysis of heuristic search using parallel hash tables. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, 1990.

[NS79] D. Nassimi and S. Sahni. Bitonic sort on a mesh connected parallel computer. *IEEE Transactions on Computers*, C–28(1), January 1979.

[Qui87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, New York, NY, 1987.

[RS90] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*. Springer-Verlag, New York, NY, 1990.

[SKAT91] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Efficient algorithms for parallel sorting on mesh multicomputers. *International Journal of Parallel Programming*, 20(2):95–131, 1991.

[SS88] Y. Saad and M. H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37:867–872, 1988.

[SS90] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, (14):361–372, 1990.

[Tho83] C. D. Thompson. Fourier transforms in VLSI. *IBM Journal of Research and Development*, C-32(11):1047–1057, 1983.

[WI89] K. Wada and N. Ichiyoshi. A distributed shortest path algorithm and its mapping on the Multi-PSI. In *Proceedings of International Conference of Parallel Processing*, 1989.

[WS89] J. Woo and S. Sahni. Hypercube computing: Connected components. *Journal of Supercomputing*, 3:209–234, 1989.