Instructor's Guide to Parallel Programming in C with MPI and OpenMP

Michael J. Quinn

July 30, 2003

Contents

Preface

This booklet contains solutions to the "pencil and paper" exercises found in *Parallel Programming in C with MPI and OpenMP*. It does not contain solutions to the programming assignments, except for a few of the shortest ones. Likewise, it does not report results for the benchmarking exercises, since the times that students observe will vary greatly from system to system.

I apologize for any mistakes that have survived my proofreading. If you identify any errors in this solutions manual, or if you have any ideas for additional exercises, I would appreciate hearing from you.

Michael J. Quinn
Oregon State University
Department of Computer Science
Corvallis, OR 97331

quinn@cs.orst.edu

Chapter 1

Motivation and History

1.1 I like to do this exercise in class during the first lecture. It helps motivate many of the concepts discussed in the course, including speedup, contention, and scalability.

(a) The best way to sort cards by hand can spark an entertaining debate. I believe it is faster to sort the cards initially by suit, then to use an insertion sort to order each suit in turn while holding it in your hands. (Knuth disagrees: see *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison-Wesley, 1973, pp. 170–171, for another opinion.)

(b) Assuming no great variance in card-handling skills, the best way for p people to sort p decks of cards is to let each person sort one deck of cards. Assuming person i requires t_i time units to sort a deck of cards, the time needed for p persons to sort p decks of cards is $\max\{t_1, t_2, \ldots, t_p\}$.

(c) The time needed for p people to sort 1 deck of cards depends heavily upon the parallel algorithm used. Students often use the following algorithm when $2 \leq p \leq 4$. Each person is assigned responsibility for one or two suits. The unsorted deck is divided among the people. Each person keeps the cards associated with his or her suit and tosses the other cards to the appropriate partner. After working through the unsorted deck, each person sorts the suits he or she has been assigned.

One of my classes discovered an interesting parallel algorithm for six people that exhibits both pipelining and data parallelism. Persons 1 and 2 distribute the cards, while persons 3, 4, 5, and 6 are each responsible for sorting cards in a particular suit.

Person 1 starts with the deck of cards. He or she hands roughly one-half the cards to person 2. Persons 1 and 2 work through their cards, tossing each card to one of the other persons, depending upon the suit. Persons 3, 4, 5, and 6 take the cards tossed to them by persons 1 and 2 and use insertion sort to order the cards in their suit. When these people have finished sorting their cards, they hand them to person 1, who concatenates them, completing the sort.

1.2 It requires $\lceil \log_2 n \rceil$ folds to put n holes in a piece of paper with a single stroke of the hole punch.

Proof of correctness (by induction on number of holes). Let $f(i)$ represent the number of folds needed to put i holes in a piece of paper. Base case: With no folds of the paper, the punch can put one hole in it. Hence $f(1) = 0 = \lceil \log_2 1 \rceil$. Induction: Assume $f(k) = \lceil \log_2 k \rceil$ for $1 \leq n \leq k$. Prove true for $2k$. We want to make $2k$ holes. If we fold the paper in half $\lceil \log_2 k \rceil$ times, we can make k holes (induction hypothesis). Hence if we fold the paper in half once more we can make $2k$ holes. Suppose $k = 2^j + r$, where $r < 2^j$. Then $2k = 2^{j+1} + 2r$, where $2r < 2^{j+1}$. Hence

$$f(2k) = \lceil \log_2 k \rceil + 1 = (j+1) + 1 = j + 2 = \lceil \log_2 2k \rceil$$

Optimality proof (by induction on number of folds): Let $h(i)$ denote the maximum number of holes that can be made with i folds of the paper. Base case: With no folds of the paper, the punch can put at most one hole in it. $h(0) = 2^0 = 1$. Induction: Assume $h(n) = 2^n$ for $0 \leq n \leq k$. Prove true for $k+1$. Folding the paper once more allows us to double the number of holes. Hence $h(k+1) = 2h(k) = 2 \times 2^k = 2^{k+1}$.

Since at most 2^i holes can be made with i folds of the paper, an algorithm that makes j holes with $\lceil \log_2 j \rceil$ folds of the paper is optimal.

1.3 (a) Hand the cards to the accountant at the leftmost desk in the front row. The accountant keeps 25 cards and passes 975 cards to the accountant on her right. This accountant keeps 25 cards and passes 950 to the accountant on her right. This process continues until every accountant in the front row has 25 cards. Meanwhile, the front row accountants pass cards to the accountants behind them. The number of accountants receiving cards depends upon how many cards each accountant keeps. For example, if each accountant keeps 5 cards, then a total of 200 accountants (5 in each column) will receive cards.

(b) After each accountant has computed the total of the cards she controls, the subtotals are accumulated by reversing the flow used to distribute the cards. The last accountant in each column that received cards passes her subtotal to the accountant in front of her. This accountant adds the subtotal she received with her own subtotal and passes the sum to the accountant in front of her. Eventually the subtotals reach the front row. The accountants accumulate subtotals right to left until the leftmost accountant in the front row has the grand total.

(c) Increasing the number of accountants decreases the total computation time, but increases the total communication time. The overall time needed is the sum of these two times. For awhile, adding accountants reduces the overall time needed, but eventually the decrease in computation time is outweighed by the increase in communication time. At this point adding accountants increases, rather than decreases, overall execution time. The point at which this occurs depends upon your estimates of how long an addition takes and how long a communication takes. However, if 1,000 accountants are actually used, the computation must take longer than if 500 accountants are used, because there is nothing for the extra accountants to do other than waste time collecting single cards and then handing them back. (An accountant with only one card cannot do an addition.) For this reason the curve must slope up between 500 and 1,000 accountants.

(d) One accountant should require 10 times longer to add 10,000 numbers than 1,000 numbers. When adding 10,000 numbers, we should be able to accommodate many more accountants before overall execution time increases.

(e) One thousand accountants cannot perform the task one thousand time faster than one accountant because they spend time communicating with each other. This represents extra work that the single accountant does not have to do.

1.4 We will arrange the desks of the accountants into a binary tree. One accountant is at the root. The desks of two other accountants are near the desk of the "root" accountant. Each of these accountants have two accountants near them, and so on. The root accountant is given 1000 cards. She divides them in half, giving 500 to each of the two accountants near her. These accountants divide their portions in half, giving 250 cards to both of the accountants near them, and so on.

This communication pattern reduces the communication overhead, meaning that the overall execution time of the algorithm for a given number of accountants should be lower than the execution time of the algorithm described in the previous exercise.

(In Chapter 3 we'll see how a binomial tree is even better than a binary tree, because it avoids the problem of inactive interior nodes while computing gets done at the leaves.)

1.5 It takes m time units for the first task to be processed. After that, a new task is completed every time unit. Hence the time needed to complete n tasks is $m + n - 1$.

1.6 The time needed to copy k pages is $15 + 4 + (k - 1) = 18 + k$ seconds, or $(18 + k)/60$ minutes. We want to choose k such that

$$\frac{k}{(18+k)/60} = \frac{60k}{18+k} = 40$$

Solving for k, we get $k = 36$.

1.7 All of today's supercomputers are parallel computers. Today's supercomputers are capable of trillions of operations per second, and no single processor is that fast. Not every parallel computer is a supercomputer. A parallel computer with a few commodity CPUs is not capable of performing trillions of operations per second.

1.8 Performance increases exponentially. Since performance improves by a factor of 10 every 5 years, we know that $x^5 = 10$. Hence

$$5 \ln x = \ln 10 \Rightarrow \ln x = \ln 10/5 \Rightarrow \ln x = 0.46 \Rightarrow x = e^{0.46} \Rightarrow x = 1.58$$

We want to find y such that $x^y = 2$.

$$1.58^y = 2 \Rightarrow y \ln 1.58 = \ln 2 \Rightarrow y = 1.51$$

So the performance of CPUs doubles every 1.51 years (about every 18 months).

1.9 (a) The 64-CPU Cosmic Cube had a best case performance of 5–10 megaflops. This translates into a single-CPU speed of 0.078–0.156 megaflops. Caltech's Supercomputing '97 cluster had 140 CPUs executing at more than 10 gigaflops. This translates into a speed of at least 71.4 megaflops per CPU. The overall speed increase is in the range 458–915.

(b) We want to find x_1 such that ${x_1}^{15} = 458$.

$$ {x_1}^{15} \Rightarrow 15 \ln x_1 = \ln 458 \Rightarrow x_1 = 1.50 $$

We want to find x_2 such that ${x_2}^{15} = 915$.

$$ {x_1}^{15} \Rightarrow 15 \ln x_1 = \ln 915 \Rightarrow x_1 = 1.57 $$

The annual microprocessor performance improvement needed to account for the speed increase calculated in part (a) was between 50% and 57%.

1.10 The faster computer will be capable of performing 100 times as many operations in 15 hours as the existing computer. Let n denote the size of the problem that can be solved in 15 hours by the faster computer.

(a) $n/100,000 = 100 \Rightarrow n = 10,000,000$.

(b) $n \log_2 n/(100,000 \log_2 100,000) = 100 \Rightarrow n \approx 7,280,000$

(c) $n^2/(100,000)^2 = 100 \Rightarrow n^2 = 1 \times 10^{12} \Rightarrow n = 1,000,000$

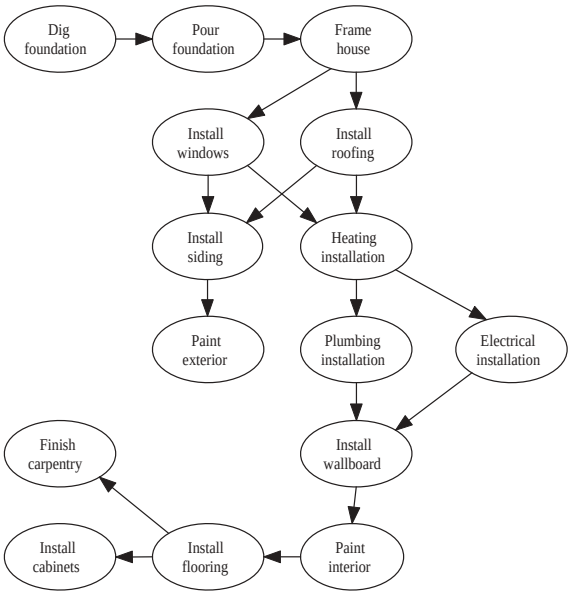
(d) $n^3/(100,000)^3 = 100 \Rightarrow n^3 = 1 \times 10^{17} \Rightarrow n = 464,159$

1.11 Advantages of commodity clusters: The latest CPUs appear in commodity clusters before commercial parallel computers. Commodity clusters take advantage of small profit margins on components, meaning the cost per megaflop is lower. Commodity clusters can be built entirely with freely available software. Commodity clusters have a low entry cost.

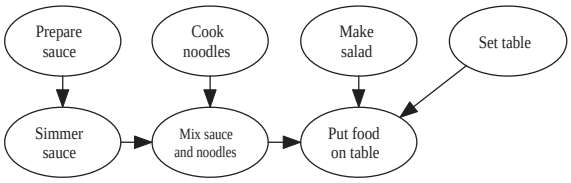
Advantages of commercial parallel computers: Commercial systems can have a higher performance interprocessor communication system that is better balanced with the speed of the processors.

1.12 Students will come up with many acceptable task graphs, of course. Here are examples.

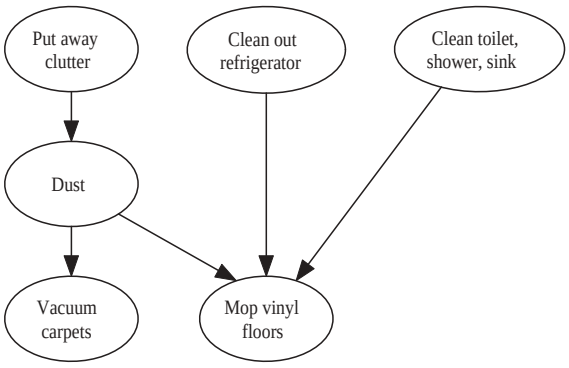
(a) Building a house



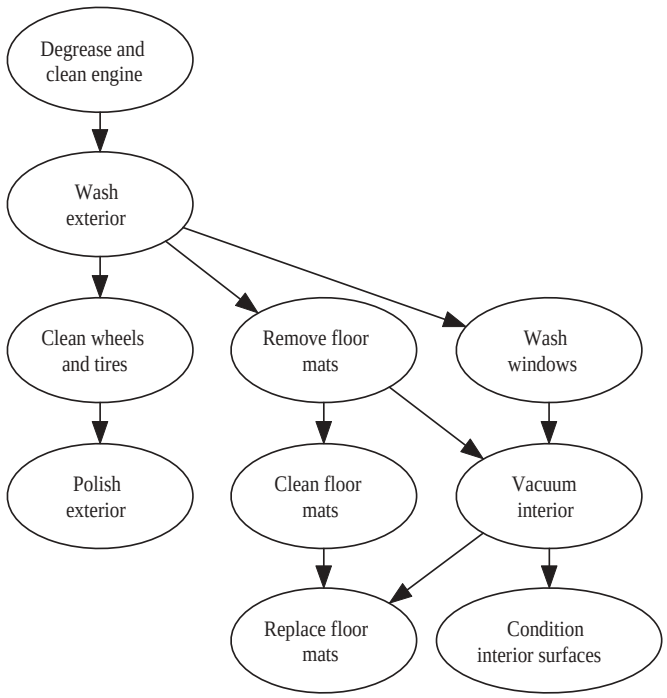
(b) Cooking and serving a spaghetti dinner



(c) Cleaning an apartment



(d) Detailing an automobile



1.13 The upper set of three nodes labeled A is a source of data parallelism. The lower let of three nodes labeled A is another source of data parallelism. B and C are functionally parallel. C and D are functionally parallel. D and the lower group of three nodes labeled A are functionally parallel.

1.14 The advantage of preallocation is that it reduces the overhead associated with assigning documents to idle processors at run-time. The advantage of putting documents on a list and letting processors remove documents as fast as they could process them is that it balances the work among the processors. We do not have to worry about one processor being done with its share of documents while another processor (perhaps with longer documents) still has many to process.

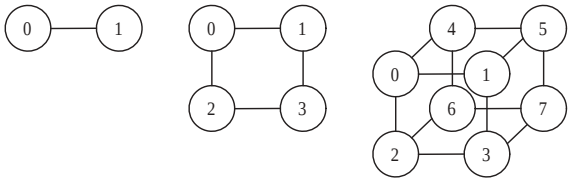
1.15 Before allocating any documents to processors we could add the lengths of all the documents and then use a heuristic to allocate documents to processors so that each processor is responsible for roughly the same total document length. (We need to use a heuristic, since

optimally balancing the document lengths is an NP-hard problem.) This approach avoids massive imbalances in workloads. It also avoids run-time overhead associated with access to the shared list of unassigned documents.

Chapter 2

Parallel Architectures

2.1 Hypercube networks with 2, 4, and 8 nodes. A hypercube network with n nodes is a subgraph of a hypercube network with $2n$ nodes.



2.2 A 0-dimensional hypercube can be labeled 1 way. A d-dimensional hypercube can be labeled $2^d(\log_2 d)!$ ways, for $d \geq 1$. Explanation: We can give any of the 2^d nodes the label 0. Once this is done, we label the neighbors of node 0. We have $\log_2 d$ choices for the label 1, $\log_2 d - 1$ choices for the label 2, $\log_2 d - 2$ choices for the label 4, $\log_2 d - 3$ choices for the label 8, etc. In other words, there are $(\log_2 d)!$ ways to label the nodes adjacent to node 0. Once this is done, all the other labels are forced. Hence the number of different labelings is $2^d(\log_2 d)!$.

2.3 The number of nodes distance i from s in a d-dimensional hypercube is $\binom{d}{i}$ (d choose i).

2.4 If node u is distance i from node v in a hypercube, exactly i bits in their addresses (node numbers) are different. Starting at u and flipping one of these bits at a time yields the addresses of the nodes on a path from u to v. Since these bits can be flipped in any order, there are $i!$ different paths of length i from u to v.

2.5 If node u is distance i from node v in a hypercube, exactly i bits in their addresses (node numbers) are different. Starting at u and flipping one of these bits at a time yields the addresses of the nodes on a path from u to v. To generate paths that share no intermediate vertices, we order the bits that are different:

$$b_0, b_1, \ldots, b_{i-1}$$

We generate vertices on the first path by flipping the bits in this order:

$$b_0, b_1, \ldots, b_{i-1}$$

We generate vertices on the second path by flipping the bits in this order:

$$b_1, b_2, \ldots, b_{i-1}, b_0$$

We generate vertices on the third path by flipping the bits in this order:

$$b_2, b_3, \ldots, b_{i-1}, b_0, b_1$$

This generates i distinct paths from u to v.

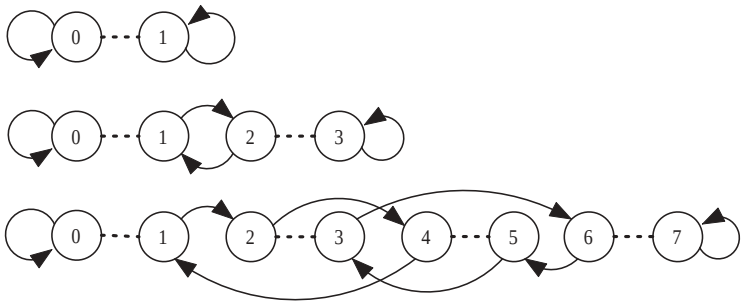
2.6 A hypercube is a bipartite graph. Nodes whose addresses contain an even number of 1s are connected to nodes whose addresses contain an odd number of 1s. A bipartite graph cannot have a cycle of odd length.

2.7 Compute the bitwise "exclusive or" operation on v and v. The number of 1 bits in the result represents the length of the shortest path from u to v. Since u and v have $\log n$ bits, there are no more than $\log n$ bits set, and the shortest path has length at most $\log n$. Number the indices from right to left, giving them values $0, 1, \ldots, \log n - 1$. Suppose the exclusive or results in k bits being set, and these bits are at indices $b_1, b_2, \ldots, b_k$. This algorithm prints the shortest path from u to v.

```
w ← u
print w
for i ← 1 to k
   w ← w ⊗ 2^(b_k)          {exclusive or}
   print w
endfor
```

Example: Find a shortest path from node 2 to node 11 in a four-dimensional hypercube. $0010 \otimes 1011 = 1001$. Hence $i_1 = 0$ and $i_1 = 3$. First jump: $0010 \otimes 0001 = 0011$. Second jump: $0011 \otimes 1000 = 1011$. Path is $0010 - 0011 - 1011$.

2.8 Shuffle-exchange networks with 2, 4, and 8 nodes. Dashed lines denote exchange links; solid arrows denote shuffle links. A shuffle-exchange network with n nodes is not a subgraph of a shuffle-exchange network with $2n$ nodes.



2.9 Nodes u and v are exactly $2k - 1$ link traversals apart if one is node 0 and the other is node $2^k - 1$. When this is the case, all k bits must be changed, which means there must be at least k traversals along exchange links. There must also be at least $k - 1$ traversals along shuffle links to set up the bit reversals. The total number of link traversals is $2k - 1$.

In all other cases, u and v have at least one bit in common, and following no more than $k - 1$ shuffle links and $k - 1$ exchange links is sufficient to reach v from u.

2.10 Algorithm to route message from u to v in an n-node shuffle-exchange network:

```
for i ← 1 to log_2 n do
    b ← u ⊗ v        {exclusive or}
    if b is odd then
        print "exchange"
    endif
    u ← left_cyclic_rotate(u)
    v ← left_cyclic_rotate(v)
    print "shuffle"
```

Example: Routing message from 6 to 5 in a 16-node shuffle-exchange network. $u = 0110$ and $v = 0101$. Algorithm produces the following route: exchange – shuffle – shuffle – shuffle – exchange – shuffle.

2.11 (a) $(n/2)\log_2 n$

(b) $\log_2 n$

(c) $(n/2)\log_2 n$

(d) 4

(e) No

2.12 Suppose the address of the destination is $b_{k-1}b_{k-2}\ldots b_1b_0$. Every switch has two edges leaving it, an "up" edge (toward node 0) and a "down" edge (away from node 0). The following algorithm indicates how to follow the edges to reach the destination node with the given address.

```
for i ← k − 1 downto 0 do
  if b_i = 0 then
    print "up"
  else
    print "down"
  endif
endfor
```

2.13 Data parallelism means applying the same operation across a data set. A processor array is only capable of performing one operation at a time, but every processing element may perform that operation on its own local operands. Hence processor arrays are a good fit for data-parallel programs.

2.14 Let i be the vector length, where $1 \leq i \leq 50$. The total number of operations to be performed is i. The time required to perform these operations is $\lceil i/8 \rceil \times 0.1$ μsecond. Hence performance is

$$\frac{i}{\lceil i/8 \rceil} \times 10^7 \text{ operations/second}$$

For example, when vector length is 8, performance is 80 million operations per second. When vector length is 50 performance is $(50/7)\times 10^7$ operations per second, or about 71.4 million operations per second.

2.15 (a) The efficiency is $1/k$

(b) The efficiency is

$$\frac{\sum_{i=1}^{k} I_i P_i}{\sum_{i=1}^{k} I_i}$$

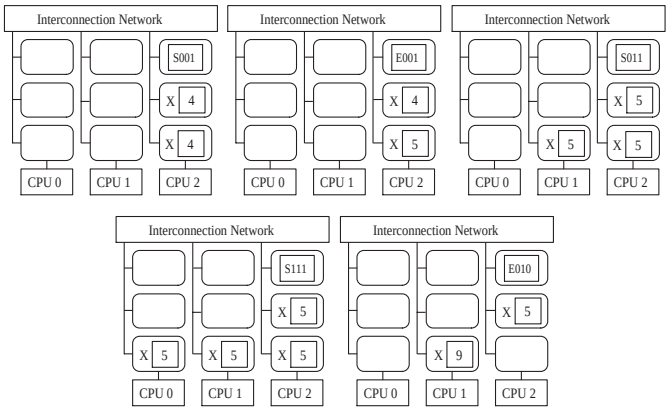
2.16 Large caches reduce the load on the memory bus, enabling the system to utilize more processors efficiently.

2.17 Even with large instruction and data caches, every processor still needs to access the memory bus occasionally. By the time the system contains a few dozen processors, the memory bus is typically saturated.

2.18 (a) The directory should be distributed among the multiprocessors' local memories so that accessing the directory does not become a performance bottleneck.

(b) If the contents of the directories were replicated, we would need to make sure the values were coherent, which is the problem we are trying to solve by implementing the directory.

2.19 Continuation of directory-based cache coherence protocol example.



2.20 Here is one set of examples.

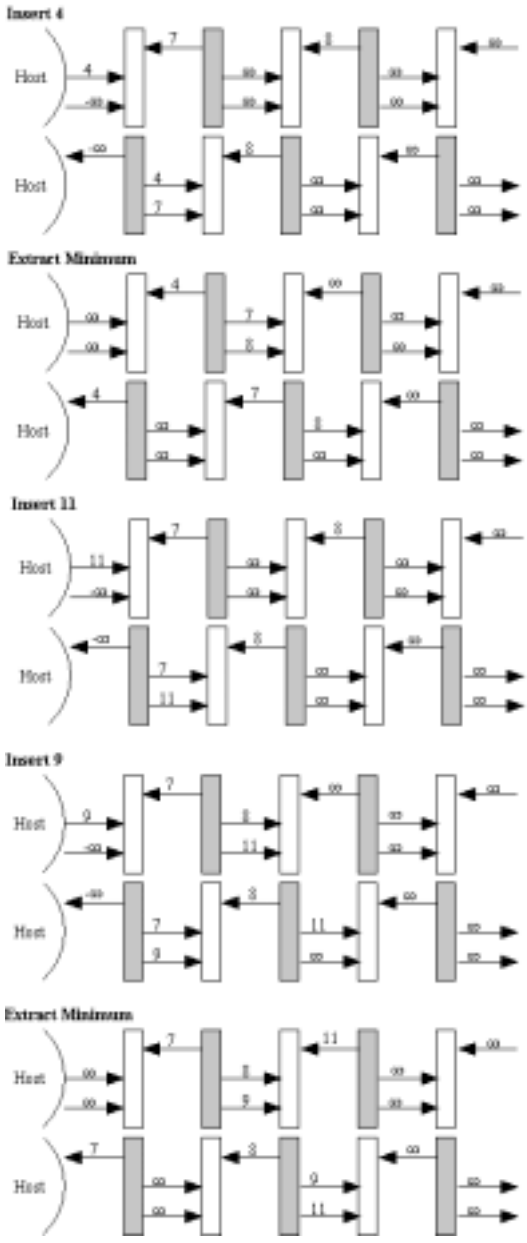
SISD: Digital Equipment Corporation PDP-11

SIMD: MasPar MP-1

MISD: Intel iWarp

MIMD: Sequent Symmetry

2.21 Continuation of systolic priority queue example.



2.22 Contemporary supercomputers must contain thousands of processors. Even distributed multiprocessors scale only to hundreds of processors. Hence supercomputers are invariably multicomputers. However, it is true that each node of a supercomputer is usually a multiprocessor. The typical contemporary supercomputer, then, is a collection of multiprocessors forming a multicomputer.

Chapter 3

Parallel Algorithm Design

3.1 Suppose two processors are available to find the sum of n values, which are initially stored on a single processor. If one processor does all the summing, the processor utilization is 50%, but there is no interprocessor communication. If two processors share the computation, the processor utilization is closer to 100%, but there is some interprocessor communication. One processor must send the other processor $n/2$ values and receive in return the sum of those values. The time spent in interprocessor communication prevents utilization from being 100%.

3.2 (a) $\log 3 \approx 1.58$, $\lfloor \log 3 \rfloor = 1$, $\lceil \log 3 \rceil = 2$

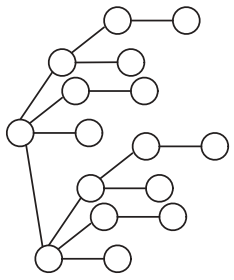
(b) $\log 13 \approx 3.70$, $\lfloor \log 13 \rfloor = 3$, $\lceil \log 13 \rceil = 4$

(c) $\log 32 = 5$, $\lfloor \log 32 \rfloor = 5$, $\lceil \log 32 \rceil = 5$

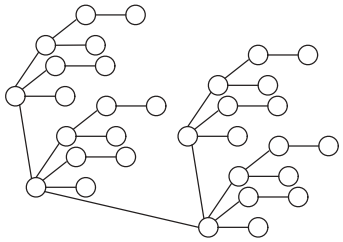
(d) $\log 123 \approx 6.94$, $\lfloor \log 123 \rfloor = 6$, $\lceil \log 123 \rceil = 7$

(e) $\log 321 \approx 8.33$, $\lfloor \log 321 \rfloor = 8$, $\lceil \log 321 \rceil = 9$

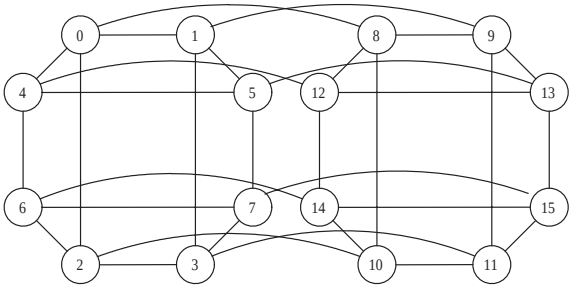
3.3 (a) Binomial tree with 16 nodes



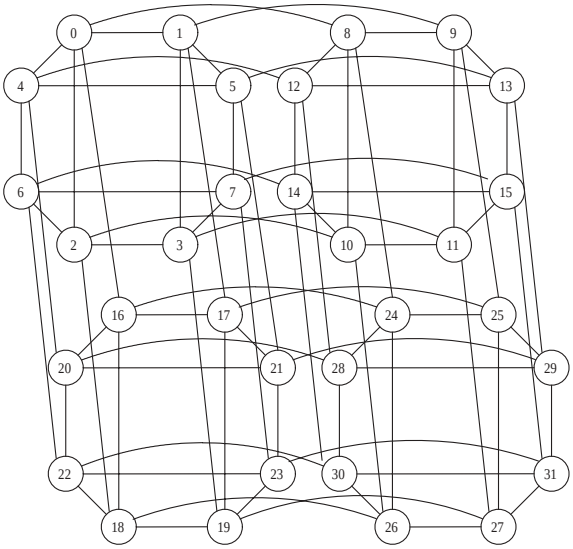
(b) Binomial tree with 32 nodes



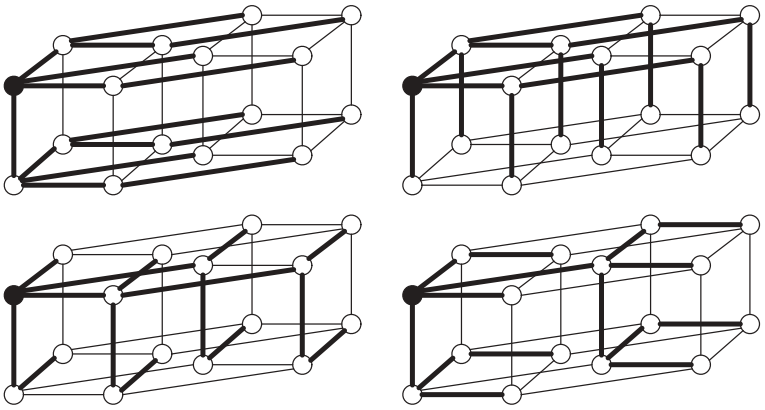
3.4 (a) Hypercube with 16 nodes



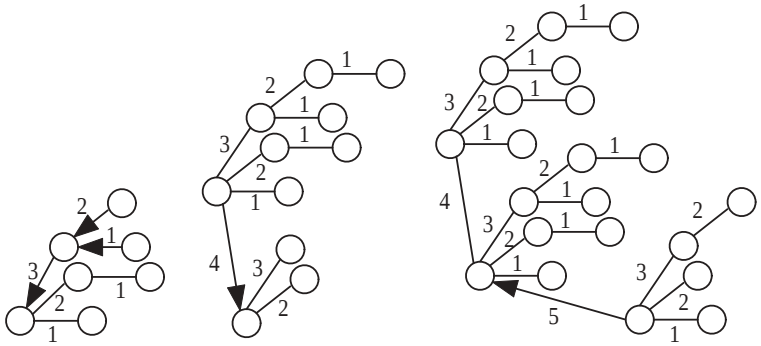
(b) Hypercube with 32 nodes



3.5 Here are four binomial trees embedded in a four-dimensional hypercube. All four binomial trees have the same hypercube node as their root.



3.6 Three reductions, each performed with $\lceil \log n \rceil$ communication steps.



3.7 A C program that describes the communications performed by a task participating in a reduction.

```
int main (int argc, char *argv[])
{
   int i;
   int id;  /* Task ID */
   int p;   /* Number of tasks */
```

```
   int pow; /* ID diff between sending/receiving procs */

   if (argc != 3) {
      printf ("Command line: %s <p> <id>\n", argv[0]);
      exit (-1);
   }
   p = atoi(argv[1]);
   id = atoi(argv[2]);
   if ((id < 0) || (id >= p)) {
      printf ("Task id must be in range 0..p-1\n");
      exit (-1);
   }
   if (p == 1) {
      printf ("No messages sent or received\n");
      exit (-1);
   }
   pow = 1;
   while(2*pow < p) pow *= 2;
   while (pow > 0) {
      if ((id < pow) && (id + pow <= p))
         printf ("Message received from task %d\n",
            id + pow);
      else if (id >= pow) {
         printf ("Message sent to task %d\n", id - pow);
         break;
      }
      pow /= 2;
   }
}
```

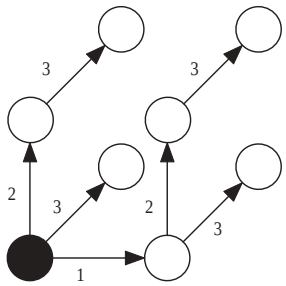
3.8 From the discussion in the chapter we know that we can reduce n values in time $\Theta(\log n)$ using n tasks. The only question, then, is whether we can improve the execution time by using fewer tasks.

Suppose we use only k tasks. Each task reduces about n/k values, and then the tasks communicate to finish the reduction. The time complexity of this parallel algorithm would be $\Theta(n/k + \log k)$. Let's find the value of k that minimizes this function. If $f(k) = n/k + \log k$, then $f'(k) = -n/k^2 + 1/k \ln 2$. $f'(k) = 0$ when $k = n \ln 2$; i.e., when $k = \Theta(n)$. Hence the complexity of the parallel reduction is $\Omega(n/n + \log n) = \Omega(\log n)$.

3.9 Our broadcast algorithm will be based on a binomial tree communi-

cation pattern. Data values flow in the opposite direction they do for a reduction.

(a) Here is a task/channel graph showing how the root task (shown in black) broadcasts to 7 other tasks in three steps. In general, use of a binomial tree allows a root task to broadcast to $p-1$ other tasks in $\lceil \log p \rceil$ communication steps.



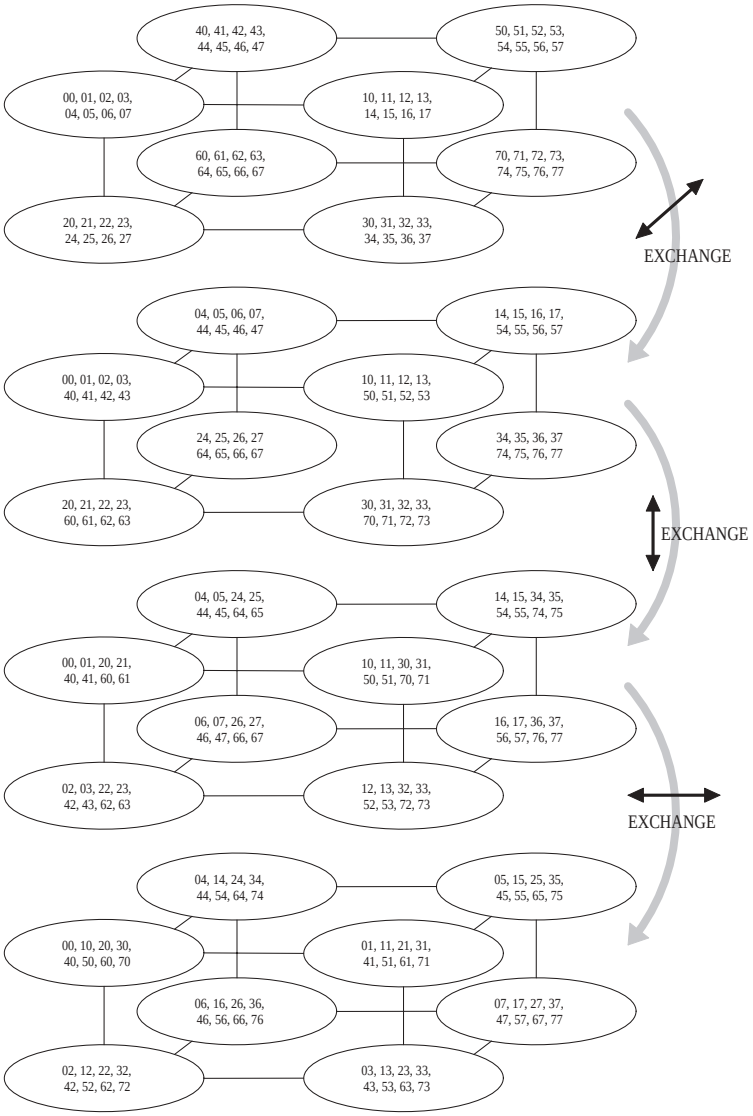
(b) We prove by induction that broadcast among p tasks (1 root task broadcasting to $p-1$ other tasks) requires at least $\lceil \log p \rceil$ communication steps. Basis. Suppose $p = 1$. No communications are needed. $\lceil \log 1 \rceil = 0$. Induction. Suppose we can broadcast among p tasks in $\lceil \log p \rceil$ communication steps, for $1 \leq p \leq k$. Then $\lceil \log k \rceil + 1$ communication steps are needed to broadcast among $2k$ tasks, because $\lceil \log k \rceil$ communication steps allow k tasks to get the value, and 1 additional step allows each of the k tasks to communicate the value to one of the k tasks that needs it. $\lceil \log k \rceil + 1 = \lceil \log 2k \rceil$. Hence the algorithm devised in part (a) is optimal.

3.10 The all-gather algorithm is able to route more values in the same because its utilization of the available communication channels is higher. During each step of the all-gather algorithm every task sends and receives a message. In the scatter algorithm the tasks are engaged gradually. About half of the tasks never send a message, and these tasks only receive a message in the last step of the scatter.

3.11 We organize the tasks into a logical hypercube. The algorithm has $\log p$ iterations. In the first iteration each task in the lower half of the hypercube swaps $p/2$ elements with a partner task in the upper half of the hypercube. After this step all elements destined for tasks in the upper half of the hypercube are held by tasks in the upper half of the hypercube, and all elements destined for tasks in the lower half of the

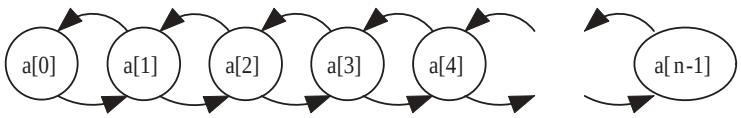
hypercube are held by tasks in the lower half of the hypercube. At this point the algorithm is applied recursively to the lower and upper halves of the hypercube (dividing the hypercube into quarters). After $\log p$ swap steps, each process controls the correct elements.

The following picture illustrates the algorithm for $p = 8$:

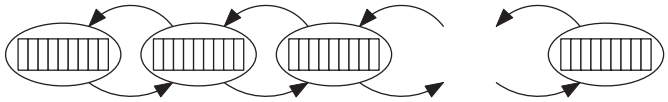


During each step each process sends and receives $p/2$ elements. There are $\log p$ steps. Hence the complexity of the hypercube-based algorithm is $\Theta(p \log p)$.

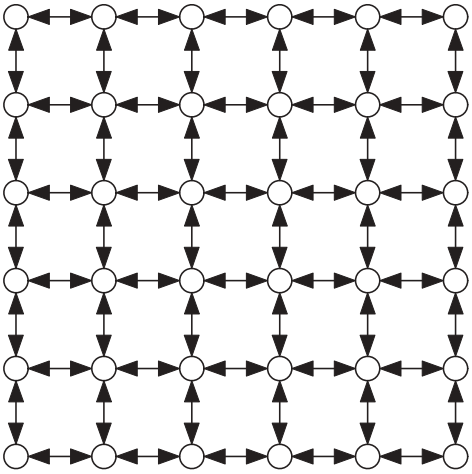
3.12 Initially we create a primitive task for each key. The algorithm iterates through two steps. In the first step the tasks associated with even-numbered keys send their keys to the tasks associated with the successor keys. The receiving tasks keep the larger value and return the smaller value. In the second step the tasks associated with the odd-numbered keys send their keys to the tasks associated with the successor keys. The receiving tasks keep the larger value and return the smaller value. After $n/2$ iterations the list must be sorted. The time complexity of this algorithm is $\Theta(n)$ with n concurrent tasks.



After agglomeration we have p tasks. The algorithm iterates through two steps. In the first step each task makes a pass through its subarray, from low index to high index, comparing adjacent keys. It exchanges keys it finds to be out of order. In the second step each task (except the last) passes its largest key to its successor task. The successor task compares the key it receives with it smallest key, returning the smaller value to the sending task. The computational complexity of each step is $\Theta(n/p)$. The communication complexity of each step is $\Theta(1)$. After at most n iterations the list is sorted. The overall complexity of the parallel algorithm is $\Theta(n^2/p + n)$ with p processors.

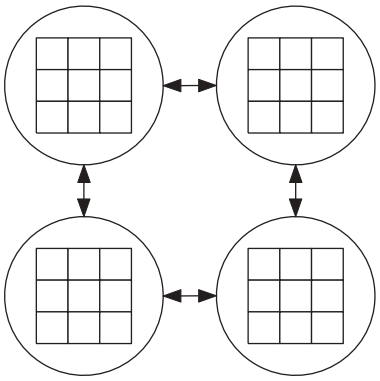


3.13 Initially we associate a primitive task with each pixel. Every primitive task communicates with tasks above it, below it, to its left, and to its right, if these tasks exist. Interior tasks have four neighbors, corner tasks has two neighbors, and edge tasks have three neighbors.

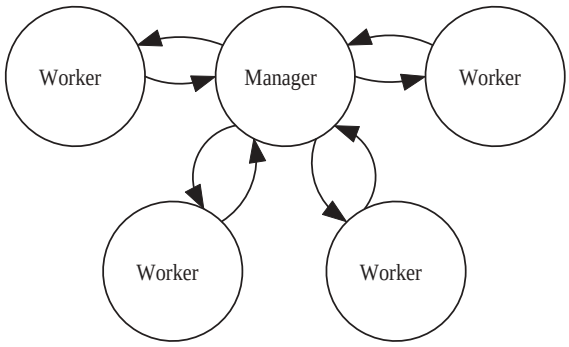


Each task associated with a 1 pixel at position (i, j) sets its component number to the unique value $n \times i + j$. At each step of the algorithm each task sends its current component number to its neighbors and receives from its neighbors their component numbers. The minimum of the component numbers received from a neighboring task is less than the task's current component number, then the task replaces its component number with this minimum value. In fewer than n^2 iterations the component numbers have stabilized.

A good agglomeration maximizes the computation/communication ratio by grouping primitive tasks into rectangles that are as square as possible.



3.14 We set up one "manager" task and a bunch of "worker" tasks. A pair of channels connects the manager with each worker. All tasks have a copy of the dictionary.



The manager partially fills out a crossword puzzle and then assigns it to a worker to complete. By assigning partially-completed puzzles to the workers, the manager ensures no two workers are doing the same thing. A worker sends a message back to the manager either when it has successfully filled in a puzzle or when it has determined there is no way to solve the partially filled-in puzzle it received. When the manager receives a solved puzzle from a worker it tells all the workers to stop.

3.15 Each primitive task is responsible for the coordinates of one of the houses. All primitive tasks have the coordinates of the train station. Each task computes the euclidean distance from its house to the train station. Each task i contributes a pair (d_i, i) to a reduction in which the operation is minimum. The result of the reduction is the pair (d_k, k), where d_k is the minimum of all the submitted values of d_i, and k is the index of the task submitting d_k. The communication pattern is a binomial tree, allowing the reduction to take place in logarithmic time. A sensible agglomeration assigns about n/p coordinate pairs to each of p tasks. Each task find the closest house among the n/p houses it has been assigned, then participates in the reduction step with the other tasks.

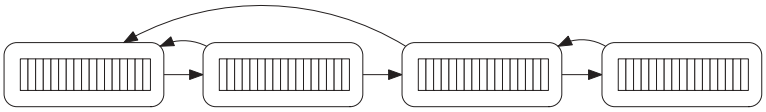
The "before" and "after" task/channel graphs should resemble Figure 3.16.

3.16 The text is divided into p segments, one per task. Each task is responsible for searching its portion of the text for the pattern. We

need to be able to handle the case where the pattern is split between two tasks' sections. Suppose the pattern has k letters. Each of the first $p-1$ tasks must pass the last $k-1$ letters of its portion of the text to the subsequent task. Now all occurrences of the pattern can be detected.

After the tasks have completed their portions of the search, a gather communication brings together information about all occurrences of the pattern.

In the following task/channel graph, the straight arrows represent channels used to copy the last final $k-1$ letters of each task's section to its successor task. The curved arrows represent channels used for the gather step.



3.17 If the pattern is likely to appear multiple times in the text, and if we are only interested in finding the first occurrence of the pattern in the text, then the allocation of contiguous blocks of text to the tasks is probably unacceptable. It makes it too likely that the first task will find the pattern, making the work of the other tasks superfluous. A better strategy is to divide the text into jp portions and allocate these portions in an interleaved fashion to the tasks. Each task ends up with j segments from different parts of the text.

As before, the final $k-1$ letters in each segment must be copied into the successor segment, in case the pattern is split across two segments.

Communication tasks enable one task to inform the other tasks as soon as it has found a match.

3.18 Associate a primitive task with each key. In the first step the tasks perform a max-reduction on the keys. In the second step the maximum key is broadcast to the tasks. In the third step each task compares its key with the maximum. In the fourth step the tasks perform another max-reduction. Every task submits a candidate key to the second max-reduction except the task whose key is equal to the maximum key.

We can reduce the communication complexity of the parallel algorithm without increasing the computation time on p processors by agglomerating groups of n/p primitive tasks into larger tasks.

3.19 This solution is similar to the previous solution, except that in the max-reduction step each task submits the pair (k_i, i), where k_i is the value of the key and i is its index in the list. The result of the max-reduction is the pair (m_i, i), where m_i is the smallest key submitted and i is its list position. This pair is broadcast to all the primitive tasks. In the fourth step the tasks perform another max-reduction. Every task submits a candidate key except the task whose index is equal to the index received in the broadcast step.

We can reduce the communication complexity of the parallel algorithm without increasing the computation time on p processors by agglomerating groups of n/p primitive tasks into larger tasks.

Chapter 4

Message-passing Programming

4.1 (a) Give the pieces of work the numbers $0, 1, 2, \ldots, n-1$. Process k gets the pieces of work $k, k+p, k+2p$, etc.,

(b) Piece of work j is handled by process $j \bmod p$.

(c) $\lceil n/p \rceil$

(d) If n is an integer multiple of p, then all processes have the same amount of work. Otherwise, $n \bmod p$ processes have more pieces of work—$\lceil n/p \rceil$. These are processes $0, 1, \ldots, (n \bmod p) - 1$.

(e) $\lfloor n/p \rfloor$

(f) If n is an integer multiple of p, then all processes have the same amount of work. Otherwise, $p - (n \bmod p)$ processes have less pieces of work—$\lfloor n/p \rfloor$. These are processes $(n \bmod p), (n \bmod p) + 1, \ldots, p - 1$.

4.2 (a) 241

(b) Overflow. In C, get 128 as result.

(c) 99

(d) 13

(e) 127

(f) 0

(g) 1

(h) 1

4.3 Modified version of function `check_circuit`.

```
int check_circuit (int id, int z) {
   int v[16];        /* Each element is a bit of z */
   int i;

   for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);
   if ((v[0]||v[1]) && (!v[1]||!v[3]) && (v[2]||v[3])
      && (!v[3] || !v[4]) && (v[4] || !v[5])
      && (v[5] || !v[6]) && (v[5] || v[6])
      && (v[6] || !v[15]) && (v[7] || !v[8])
      && (!v[7] || !v[13]) && (v[8] || v[9])
      && (v[8] || !v[9]) && (!v[9] || !v[10])
      && (v[9] || v[11]) && (v[10] || v[11])
      && (v[12] || v[13]) && (v[13] || !v[14])
      && (v[14] || v[15])) {
      printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n",
         id,v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],
         v[8],v[9],v[10],v[11],v[12],v[13],v[14],v[15]);
      fflush (stdout);
      return 1;
   } else return 0;
}
```

4.4 See Figure 4.6.

4.5 (a) The circuit can be represented by a logical expression using the same syntax as the C programming language. Parentheses are used as necessary to indicate the order in which operators should be applied.

(b) Creating a syntax-free grammar for logical expressions is easy. We could write a top-down or a bottom-up parser to construct a syntax tree for the circuit.

(c) The circuit would be represented by an expression tree. The leaves of the tree would be the inputs to the circuit. The interior nodes of the tree would be the logical gates. A *not* gate would have one child; *and* and *or* gates would have two children.

4.6 Parallel variant of Kernighan and Ritchie's "hello, world" program.

```
#include "mpi.h"
#include <stdio.h>

int main (int argc, char *argv[]) {
   int id;               /* Process rank */

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);

   printf ("hello, world, from process %d\n", id);
   fflush (stdout);
   MPI_Finalize();
   return 0;
}
```

Chapter 5

The Sieve of Eratosthenes

5.1 (a) The first $p-1$ processes get $\lceil n/p \rceil$ elements and the last process does not get any elements when $n = k(p-1)$, where k is an integer greater than 1 and less than p. Here are the cases where this happens for $n \leq 10$: $n = 4$ and $p = 3$; $n = 6$ and $p = 4$; $n = 8$ and $p = 5$; $n = 9$ and $p = 4$; and $n = 10$ and $p = 6$.

(b) In order for $\lfloor p/2 \rfloor$ processes to get no values, p must be odd and n must be equal to $p + 1$. Here are the cases where this happens for $n \leq 10$: $n = 4$ and $p = 3$; $n = 6$ and $p = 5$; $n = 8$ and $p = 7$; and $n = 10$ and $p = 9$.

5.2 We list the number of elements held by each process for each scheme. For example, (4, 3, 3) means that the first process has four elements and the second and third processes have three elements each.

(a) Scheme 1: (4, 4, 4, 3); Scheme 2: (3, 4, 4, 4)

(b) Scheme 1: (3, 3, 3, 2, 2, 2); Scheme 2: (2, 3, 2, 3, 2, 3)

(c) Scheme 1: (4, 3, 3, 3, 3); Scheme 2: (3, 3, 3, 3, 4)

(d) Scheme 1: (5, 5, 4, 4); Scheme 2: (4, 5, 4, 5)

(e) Scheme 1: (4, 4, 3, 3, 3, 3); Scheme 2: (3, 3, 4, 3, 3, 4)

(f) Scheme 1: (4, 4, 3, 3, 3, 3, 3); Scheme 2: (3, 3, 3, 4, 3, 3, 4)

5.3 This table shows the predicted execution time of the first parallel implementation of the Sieve of Eratosthenes on 1–16 processors.

Processors	*Time*
1	24.910
2	12.727
3	8.846
4	6.770
5	5.796
6	4.966
7	4.373
8	3.928
9	3.854
10	3.577
11	3.350
12	3.162
13	3.002
14	2.865
15	2.746
16	2.643

5.4 This table compares the predicted versus actual execution time of the second parallel implementation of the Sieve of Eratosthenes on a commodity cluster.

Processors	*Predicted Time*	*Actual Time*	*Error (%)*
1	12.455	12.237	1.78
2	6.499	6.609	1.66
3	4.695	5.019	6.46
4	3.657	4.072	10.19
5	3.305	3.652	9.50
6	2.890	3.270	11.62
7	2.594	3.059	15.20
8	2.371	2.856	16.98

The average error in the predictions for 2–7 processors is 10.23%.

5.5 This table compares the predicted versus actual execution time of the third parallel implementation of the Sieve of Eratosthenes on a commodity cluster.

Processors	*Predicted Time*	*Actual Time*	*Error (%)*
1	12.457	12.466	0.07
2	6.230	6.378	2.32
3	4.154	4.272	2.76
4	3.116	3.201	2.66
5	2.494	2.559	2.54
6	2.078	2.127	2.30
7	1.782	1.820	2.09
8	1.560	1.585	1.58

The average error in the predictions for 2–7 processors is 2.32%.

5.10 The first disadvantage is a lack of scalability. Since every process must set aside storage representing the integers from 2 through n, the maximum value of n cannot increase when p increases. The second disadvantage is that the OR-reduction step reduces extremely large arrays. Communicating these arrays among the processors will require a significant amount of time. The third disadvantage is that the workloads among the processors are unbalanced. For example, in the case of three processors, the processor sieiving with 2, 7, 17, etc. will take significantly to mark multiples of its primes than the processor sieving with 5, 13, 23, etc.

Chapter 6

Floyd's Algorithm

6.1 Suppose $n = kp + r$, where k and r are nonnegative integers and $0 \leq r < p$. Note that this implies $k = \lceil n/p \rceil$.

If r (the remainder) is 0, then all processes are assigned $n/p = \lceil n/p \rceil$ elements. Hence the last process is responsible for $\lceil n/p \rceil$ elements.

If $r > 0$, then the last process is responsible for elements $\lfloor (p-1)n/p \rfloor$ through $n-1$.

$$\begin{aligned}
\lfloor (p-1)n/p \rfloor &= \lfloor (p-1)(kp+r)/p \rfloor \\
&= \lfloor p(kp+r)/p - kp/p - r/p \rfloor \\
&= (kp+r) - k - \lfloor r/p \rfloor \\
&= kp + r - k - 1 \\
&= n - (k+1)
\end{aligned}$$

The total number of elements the last process is responsible for is

$$(n-1) - (n-(k+1)) + 1 = k + 1 = \lceil n/p \rceil$$

6.2 Process 0 only needs to allocate $\lfloor n/p \rfloor$ elements to store its portion of the file. Since it may need to input and pass along $\lceil n/p \rceil$ elements to some of the processes, process 0 cannot use its file storage buffer as a temporary buffer holding the elements being passed along to the other processes. (Room for one extra element must be allocated.) In contrast, process 3 will eventually be responsible for $\lceil n/p \rceil$ elements of the file. Since no process stores more elements, the space allocated

for process 3's portion of the file is sufficient to be used as a temporary message buffer.

We see this played out in Figure 6.6, in which process 0 is responsible for $\lfloor 14/4 \rfloor = 3$ elements, while process 3 is responsible for $\lceil 14/4 \rceil = 4$ elements.

6.3 We need to consider both I/O and communications within Floyd's algorithm itself. File input will be more complicated, assuming the distance matrix is stored in the file in row-major order. One process is responsible for accessing the file. It reads in one row of the matrix at a time and then scatters the row among all the processes. After n read-and-scatter steps, each process has its portion of the distance matrix.

During iteration k of Floyd's algorithm, the process controlling column k must broadcast this column to the other processes. No other communications are needed to perform the computation.

In order to print the matrix, one process should be responsible for all of the printing. Each row is gathered onto the one process that prints the row before moving on to print the next row.

6.4 Several changes are needed when switching from a rowwise block striped decomposition to a rowwise interleaved striped decomposition.

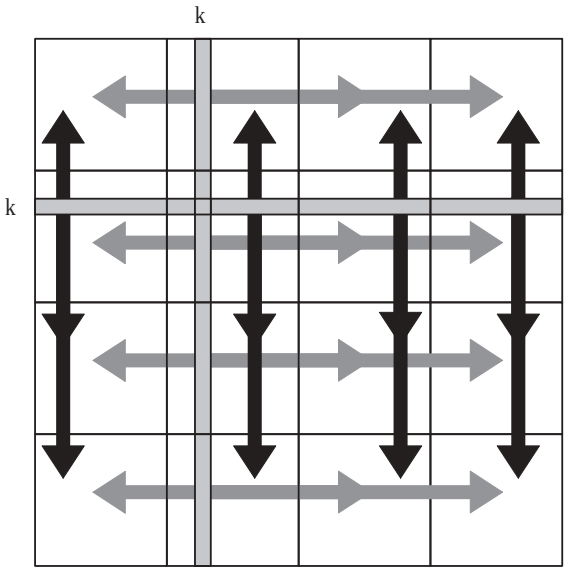
First, the process that inputs the matrix from the file can only read one row at a time. In each of n iterations, the process reads row i and sends it to process $i \bmod p$ (except when it keeps rows it is responsible for). Hence this decomposition results in more messages being sent by the process responsible for reading the matrix from the file.

Second, in the performance of Floyd's algorithm, the computation of the process broadcasting row k is different. In the block decomposition the process controlling row k is $\lfloor p(k+1) - 1)/n \rfloor$. In the interleaved decomposition the process controlling row k is $k \bmod p$.

Third, when the matrix is printed, processes send their rows to the printing process one at a time, instead of all at once. The printing process must print the first row held by process 0, the first row held by process 1, ..., the first row held by process $p-1$, the second row held by process 0, etc. Hence this decomposition results in more messages being received by the process responsible for printing the matrix.

6.5 (a) The processes are logically organized into a square mesh. During each iteration of the outer loop of the algorithm, a process needs

access to `a[i][k]` and `a[k][j]` in order to update `a[i][j]`. Hence every process controlling a portion of row `k` must broadcast it to other processes in the same column of the process mesh. Similarly, every process controlling a portion of column `k` must broadcast it to other processes in the same row of the process mesh.



Note that the $\sqrt{p}$ broadcasts across rows of the process mesh may take place simultaneously. The $\sqrt{p}$ broadcasts across columns of the process mesh may also take place simultaneously.

(b) During each iteration there is are two broadcast phases. The communication time is the same for both phases. A total of $n/\sqrt{p}$ data items, each of length 4 bytes, are broadcast among a group of $\sqrt{p}$ processes. The time needed for the broadcast is

$$\lceil \log \sqrt{p} \rceil (\lambda + 4n/(\sqrt{p}\beta)$$

Since there are two broadcast phases per iteration and n iterations, the total communication time is

$$2n\lceil \log \sqrt{p} \rceil (\lambda + 4n/(\sqrt{p}\beta))$$

(c) Since $2n\lceil \log \sqrt{p} \rceil = n\lceil \log p \rceil$, the number of messages sent by both algorithms is the same. However, the parallel version of

Floyd's algorithm that assigns each process a square block of the elements of $\mathbf{A}$ has its message transmission time reduced by a factor of $\sqrt{p}$. Hence it is superior.

6.6 We need to compute

$$n^2 \lceil n/p \rceil \chi + n \lceil \log p \rceil (lambda + 4n/\beta)$$

where $n = 1000$, $p = 16$, $\chi = 25.5$ nanosec, $\lambda = 250$ μsec, and $\beta = 10^7$ bytes/sec. The result is 4.19 seconds.

6.7 This table predicts the execution times (in seconds) when executing Floyd's algorithm on problems of size 500 and 2,000 with $1, 2, \ldots, 8$ processors. The computer has the same values of χ, λ, and β as the system used for benchmarking in this chapter.

Processors	$n = 500$	$n = 2000$
1	3.19	204.0
2	1.72	102.5
3	1.32	69.0
4	1.05	52.0
5	1.01	42.3
6	0.91	35.6
7	0.83	30.7
8	0.78	27.0