Solutions to Selected Problems in Fundamentals of Parallel Processing

Harry Jordan
and Gita Alaghband
University of Colorado at Boulder
and University of Colorado at Denver
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1: Solutions</td>
<td>5</td>
</tr>
<tr>
<td>Chapter 2: Solutions</td>
<td>9</td>
</tr>
<tr>
<td>Chapter 3: Solutions</td>
<td>25</td>
</tr>
<tr>
<td>Chapter 4: Solutions</td>
<td>45</td>
</tr>
<tr>
<td>Chapter 5 - Solutions</td>
<td>69</td>
</tr>
<tr>
<td>Chapter 6: Solutions</td>
<td>83</td>
</tr>
<tr>
<td>Chapter 7: Solutions</td>
<td>93</td>
</tr>
<tr>
<td>Chapter 8: Solutions</td>
<td>101</td>
</tr>
<tr>
<td>Chapter 9: Solutions</td>
<td>107</td>
</tr>
<tr>
<td>Chapter 10: Solutions</td>
<td>111</td>
</tr>
<tr>
<td>Chapter 11: Solutions</td>
<td>117</td>
</tr>
</tbody>
</table>
Chapter 1: Solutions

Problem: 1.1 Write SIMD and MIMD pseudo code for the tree summation of \( n \) elements of a one dimensional array for \( n \) a power of two. Do not use recursive code but organize the computations as iterations. The key issue is to determine which elements are to be added by a single arithmetic unit or a single processor at any level of the tree. You may overwrite elements of the array to be summed.

Solution: 1.1 We write pseudo code to sum \( V[0], V[1], \ldots, V[n-1] \) for \( n \) not necessarily a power of two.

**SIMD pseudo code**

```plaintext
n := \lceil \log_2 n \rceil; /* Number of levels in tree. */
m := \lfloor N/2 \rfloor; /* Number of processors at top level. */
r := N \mod 2; /* Extra element? */
for k := 1 step 1 until n begin
    V[j x 2^k] := V[j x 2^k] + V[j x 2^k + 2^{k-1}], (0 ≤ j < m);
    q := (m + r) \mod 2; /* Figure out the number of processors */
    m := \lfloor (m + r)/2 \rfloor; /* needed at next level. */
    r := q;
end;
```

```plaintext
```

MIMD pseudo code

```
    private k;
    n := [log_2 n];
    m := [N/2];
    r := N mod 2;
    for k := 0 step 1 until n
        begin
            for j := 0 step 1 until m - 2 fork ADD;
                j := m - 1;
                ADD:
                    V[j*2^k] := V[j*2^k] + V[j*2^k + 2^{k-1}];
            join m;
            r := (m + r) mod 2;
            m := (m + r)/2;
        end;
```

Problem: 1.2 What mathematical property do sum, product, maximum, and minimum have in common that allows them to be done in parallel using a tree structured algorithm?

Solution: 1.2 The operators have the property of associativity. These operators can be applied to pairs of operands in any order, allowing for a tree-like sequence.

Problem: 1.3 The SIMD matrix multiply pseudo code of Program 1-1 is written to avoid doing an explicit reduction operation that is needed for the dot product of two vectors. Write another version of SIMD matrix multiply pseudo code that avoids the reduction operation. Describe the order of operations and compare it with both the sequential version and the SIMD version of Program 1-1.

Solution: 1.3 The SIMD code of Program 1-1 does operations on rows. A column wise version would be:

```
    for j := 0 step 1 until N-1
        begin/* Compute one column of C. */
            /* Initialize sums for elements of a column of C. */
            C[i, j] := 0, (0 ≤ i ≤ N-1);
            /* Loop over terms of the inner product. */
            for k := 0 step 1 until N-1
                /* Add the k-th inner product term across rows in parallel. */
                C[i, j] := C[i, j] + A[i, k]*B[k, j], (0 ≤ i ≤ N-1);
        end;
```

The sequential version could be called the ijk form, Program 1-1 the ikj form and this version the jki form, referring to the outermost to innermost loop variable ordering.

Problem: 1.4 Apply Bernstein’s conditions to the compiler codes generated for evaluation of expression in Figure 1-6 and Figure 1-7. In each case determine which statements are independent of each other and can be executed in parallel. Detect and identify the type of dependences for statements that are not independent. Explain what might happen if two dependent statements are executed concurrently.
Solution: 1.4 From Figure 1-6.

<table>
<thead>
<tr>
<th></th>
<th>Flow dependence</th>
<th>Anti dependence</th>
<th>Output dependence</th>
<th>Independence</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1:</td>
<td>T₁ = A + B</td>
<td>S2 on S1</td>
<td>S5 on S2</td>
<td>S2 on S1</td>
</tr>
<tr>
<td>S2:</td>
<td>T₁ = T₁ + C</td>
<td>S4 on S3</td>
<td>S6 on S5</td>
<td>S4 on S3</td>
</tr>
<tr>
<td>S3:</td>
<td>T₂ = D * E</td>
<td>S5 on S4</td>
<td>S7 on S6</td>
<td>S5 on S2</td>
</tr>
<tr>
<td>S4:</td>
<td>T₂ = T₂ * F</td>
<td>S5 on S2</td>
<td></td>
<td>S6 on S5</td>
</tr>
<tr>
<td>S5:</td>
<td>T₁ = T₁ + T₂</td>
<td>S6 on S5</td>
<td></td>
<td>S7 on S6</td>
</tr>
<tr>
<td>S6:</td>
<td>T₁ = T₁ + G</td>
<td>S7 on S6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7:</td>
<td>T₁ = T₁ + H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that although S2 is independent of S4 and S1 is independent of S4, S1 is not independent of S2. This demonstrates that independence is not transitive.

From Figure 1-7.

<table>
<thead>
<tr>
<th></th>
<th>Flow dependence</th>
<th>Anti dependence</th>
<th>Output dependence</th>
<th>Independence</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1:</td>
<td>T₁ = A + B</td>
<td>S4 on S1</td>
<td>S6 on S4</td>
<td>S4 on S1</td>
</tr>
<tr>
<td>S2:</td>
<td>T₂ = C + G</td>
<td>S4 on S2</td>
<td>S7 on S6</td>
<td>S6 on S4</td>
</tr>
<tr>
<td>S3:</td>
<td>T₃ = D * E</td>
<td>S5 on S3</td>
<td></td>
<td>S7 on S6</td>
</tr>
<tr>
<td>S4:</td>
<td>T₁ = T₁ * T₂</td>
<td>S6 on S4</td>
<td></td>
<td>S5 on S3</td>
</tr>
<tr>
<td>S5:</td>
<td>T₃ = T₃ + F</td>
<td>S6 on S5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6:</td>
<td>T₁ = T₁ + T₃</td>
<td>S7 on S6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7:</td>
<td>T₁ = T₁ + H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that anti dependence of a statement on itself, as in S4, is not usually useful because the rules for evaluating assignment statements automatically satisfy it.

Problem: 1.5 To apply Bernstein’s conditions to the statements of Figure 1-6 to determine the independence of operations between the statements, how many pairs of statements must be examined? How many conditions must be tested in general for a code consisting of \( N \) statements?

Solution: 1.5

Problem: 1.6 Assume each stage of the floating addition pipeline of Figure 1-9 takes one time unit. Compare the performance of this pipelined floating point add with a true SIMD machine with six arithmetic units in which a floating point addition takes six time units. Show how long it takes to add two vectors of size 20 for both true and pipelined SIMD.

Solution: 1.6

Problem: 1.7 Consider the execution of the sequential code segment

\[
\begin{align*}
S1: & \quad X = (B - A)(A + C) \\
S2: & \quad Y = 2D(D + C) \\
S3: & \quad Z = Z(X + Y) \\
S4: & \quad C = E(F - E) \\
S5: & \quad Y = Z + 2F - B \\
S6: & \quad A = C + B/(X + 1)
\end{align*}
\]

(a) Write the shortest assembly language code using \textit{add}, \textit{sub}, \textit{mul}, and \textit{div} for addition,
subtraction, multiplication and divide respectively. Assume an instruction format with register address field so that and \( R1 = R2 + R3 \) is equivalent to \( \text{add} \, R1, \, R2, \, R3 \). Assume there are as many registers as needed, and further assume that all operands have already been loaded into registers therefore ignoring memory reference operations such as load and store.

(b) Identify all the data dependences in part (a).

(c) Assume that add/sub takes one, multiply three, and divide 18 time units on this multiple arithmetic CPU respectively, and that there are two adders, one multiplier, and one divide unit. If all instructions have been prefetched into a look-ahead buffer and you can ignore the instruction issue time, what is the minimum execution time of this assembly code on this SISD computer?

Solution: 1.7
(a)
(b)
(c)
Problem: 2.1 Consider the program fragment consisting of the two lines:

\[
x := (((((a*b) + c) + d) + e) + f) + g;
\]

\[
y := (a * b) + c;
\]

(a) Taking the parentheses as indicating the order of evaluation, what is the size of the computation for \(x\)? What is its depth?

(b) Ignoring the specified evaluation order, find the minimum depth of a computation for \(x\). What is the size of a minimum depth computation?

(c) Taking advantage of the common sub expression show a computation graph for both \(x\) and \(y\) using the evaluation order specified by parentheses. Reorder the evaluation to obtain a minimum depth computation for \(x\) and \(y\). What is its size?

(d) If all the intermediate results obtained by evaluating \(x\) in the order specified were needed in subsequent statements, what could be said about the minimum depth computation and its size? That is, we need to store every sub expression enclosed in a pair of parentheses, as well as the final result \(x\).

Solution: 2.1 (a) The computation as specified has the dependence tree shown. The depth of the tree (exclusive of an assignment to \(x\)) is 6 and its size is also 6.
(b) A minimum depth computation would use a binary tree approach shown below. The depth of this computation is 3 and its size is 6.

(c) For the common subexpression in $x$ and $y$ we have the tree below. The second tree below has minimum depth is 3, and the size of the minimum depth tree in this case is also 6.

(d) If the "$x$" computation had been an $8 = 2^3$ factor product we would have had to do the computation shown below in order to get a depth 3 computation. This would have increased the size by one. Thus, obtaining intermediate results in the parallel form is not always free.

The discussion of parallel prefix computation tells us that we can get a depth $\log_2 n$ computation of size less than $4n$ or a depth $2\log_2 n$ computation of size less than $2n$ for all prefixes (intermediate results) in an $n$ term product. Thus, a depth 3 computation could be built of size less than 28. In fact,
the upper bound is not tight, for a depth 3 computation can be built with only 9 operations, as shown.

Problem: 2.2 The sequential pseudo code below defines the bubble sort algorithm.

```
for j := N-1 step -1 until 1
  for i:=0 step 1 until j-1
  begin
    s := x[i];
    t := x[i+1];
    x[i] := min(s, t);
    x[i+1] := max(s, t);
  end;
```

In the analysis below consider only min and max as operations in the algorithm.

(a) Draw a data dependence diagram for the algorithm for \( N = 4 \).

(b) What are the size and depth of the algorithm for \( N = 4 \)?

(c) Compute the size of the algorithm for arbitrary \( N \).

(d) Compute the depth of the algorithm for arbitrary \( N \).

Solution: 2.2 (a) \( N=4 \)

(b) Size = 12, Depth = 5

(c) For an arbitrary \( N \), the size is: \( S = 2[(N-1) + (N-2) + \ldots + 1] = 2\left(\frac{(N-1) + 1}{2}\right)(N-1) = N(N-1). \)

(d) For \( N=2 \), Depth=1, and for each increment in \( N \), Depth is incremented by 2. So \( D = 2N-3 \).

Problem: 2.3 The sequential pseudo code below is an idealized form of an iterative solution to a one dimensional Poisson’s equation.
for k := 1 step 1 until M
  for i := 1 step 1 until N-1
    a[i] := (a[i-1] + a[i+1])/2 + b[i];
  end;

Solution: 2.3 The two different algorithms are both ways of iterating to make the value at each point the
average of the values at the points on either side plus a constant.
(a) The first algorithm with \( N = 4 \) and \( M = 3 \) has Size = 9 and Depth = 7 with the data dependence
diagram below.

(b) For arbitrary \( N \) and \( M \), the size and depth of the part (a) algorithm are: Size = \( M(N - 1) \),
Depth = \( 2M + N - 3 \).
(c) This algorithm has a different dependence diagram, as shown below for \( N = 4, M = 3 \). For general
\( M \) and \( N \), this leads to a size and depth of: Size = \( M(N - 1) \), Depth = \( 2M \). Thus this algorithm is more
parallel than that of part (a). Which algorithm is a better solution of Poisson’s equation is a numerical
analysis topic.
Problem: 2.4  We have two \( n \) bit numbers of the form \( a = a_{n-1} \ldots a_1 a_0 \) and \( b = b_{n-1} \ldots b_1 b_0 \). We want a small, fast circuit to compute the number \( c = c_{n-1} \ldots c_1 c_0 \) containing the left most string of bits for which \( a \) and \( b \) match, with the right hand portion padded with zeros. For example, if \( a = 1101101 \) and \( b = 1101001 \) then \( c = 1101000 \).

(a) Show how to do this with two input AND, OR and XOR gates and inverters. Hint: generate the prefix vector (1111000 in the example) that has a one in a given bit position only if \( a \) and \( b \) match in that position and in all positions to the left.

(b) What is the number of gates (not counting inverters) in the smallest circuit for generating \( c \)? How long, in gate delays, does it take to compute \( c \)? Count AND and OR as one delay, XOR as two delays and ignore inverters.

(c) How fast can \( c \) be computed if we are willing to increase the number of gates and about how many gates are needed? Answer the same question if we are willing to increase the number of gates by no more than 50%.

Solution: 2.4  (a) Define two \( n \)-bit numbers \( e = e_{n-1} \ldots e_1 e_0 \) and \( f = f_{n-1} \ldots f_1 f_0 \), where \( e_i = 1 \) if and only if \( a_i = b_i \) and \( f_j = 1 \) if and only if \( e_j = 1 \) for all \( i \leq j \leq n-1 \). Thus a one in a bit position of \( f \) means \( a = b \) in all bit positions left of and including position \( i \). Then \( c = a \land f \) is just the bit-wise AND of \( f \) and either \( a \) or \( b \). \( e_i = a_i \oplus b_i \) and \( f \) is the AND prefix on the bits of \( e \) from left to right, \( f_j = e_{n-j-1} \land e_{n-j-2} \land \ldots \land e_j \), \( 0 \leq i \leq n-1 \).

(b) Doing the prefix sequentially, as shown in the circuit of part part (a), gives the fewest gates. The number of gates is \( 3n - 1 \) and the delay is \( 2 + n - 1 + 1 = n + 2 \).

(c) The prefix can be done with \( \lceil \log_2 n \rceil \) gate delays, for a total delay of \( 3 + \lceil \log_2 n \rceil \). Using Ladner and Fisher’s \( P_0 \) algorithm, the number of AND gates for the prefix is less than \( 4n - 2 \), so \( c \) can be computed with fewer than \( 6n - 2 \) gates. If we are willing to increase the number of gates by no more than 50%, we can use the recursive odd/even algorithm for the prefix with fewer than \( 2n \) AND gates with a delay of \( 3 + 2 \lceil \log_2 n \rceil \).

Problem: 2.5  A priority circuit takes as input an \( n \) bit number \( a = a_{n-1} \ldots a_1 a_0 \) and outputs an \( n \) bit number \( b = b_{n-1} \ldots b_1 b_0 \) such that \( b = 0 \) if and only if \( a = 0 \). For \( a \neq 0 \) exactly one bit of \( b \), \( b_i \), is one and the rest are zero, where \( a_i \) is the left most bit of \( a \) that is one.
(a) If the basic operations are two input AND and OR gates, give the design of the circuit with the fewest gates to perform this function. Assume inversion can be attached to any input or output at no cost so inverters are not counted. How many gate delays does it take to compute $b$?

(b) What is the minimum possible delay for computing $b$? Design a circuit with minimal delay and determine the number of gates.

**Solution:**

**2.5** (a) Form the $n$ bit priority vector $p$ such that $p_{n-1} = 1$ and $p_i = p_{i+1} \land \overline{a_i}, \ n - 2 \geq i \geq 0$. The value of $b$ is then the bitwise AND $a \land p$. Computing the $p$ vector by a chain of $n - 2$ AND gates makes the total delay $n - 1$ AND gates and one NOT gate.

(b) The priority vector $p$ is essentially an AND prefix on the vector $a$. It can be computed by a depth $\log_2 n$ parallel prefix algorithm. The simplest parallel prefix having this minimal depth is obtained by the recursive upper/lower construction. For example, for $n = 8$: 
Problem: 2.6 Write SIMD and MIMD pseudo code for the sum prefix algorithm obtained by repeated upper/lower construction for $n$ a power of two.
HINT: Though the algorithm is defined recursively, that is not a good way to write the code. For the MIMD code, arrange to assign a processor number to each processor needed at one of the $\log_2 n$ levels of the dependence diagram and have the processor compute what adds to perform at that level. For the SIMD code, calculate index vectors giving the right things to add in terms of the level and the index of the arithmetic unit doing the add.

Solution: 2.6 MIMD pseudo code

```plaintext
private integer r, p, q, k;
shared integer n;
for k := 0 step 1 until log2 n - 1
begin
  for r := 0 step 1 until n/2 - 2
  fork ADDOP;
  r := n/2 - 1;
  ADDOP: p := \lfloor r/2^k \rfloor + 1;
  q := r mod 2^k;
  join;
end;
```

A somewhat better program does the forks once at the very beginning and uses barrier synchronization between levels.

SIMD pseudo code

```plaintext
integer k, r, p[0:n/2];
for k := 0 step 1 until log2 n - 1
begin
  p[r] := 2^k ( \lfloor r/2^k \rfloor + 1), (0 \leq r < n/2);
  V[p[r]+r mod 2^k] := V[p[r]+r mod 2^k] + V[p[r]-1], (0 \leq r < n/2);
end;
```

There is no recursion. p[r] points to the start of a group and $r \mod 2^k$ is the group member.

Problem: 2.7 Consider Ladner and Fischer's sum prefix circuit $P_0(2^l)$ for $n = 2^l$. Show data dependence diagrams for $P_0(8)$, $P_0(16)$ and $P_0(32)$. Consider writing pseudo code for $P_0(2^l)$. Is the algorithm suitable for an SIMD architecture? an MIMD architecture?

Solution: 2.7 The computation pattern for $P_0(32)$ is:
All of the operations are adds, so the constraint that an SIMD machine must do all the same operation in parallel is not a problem. The index computation, however, is very complex. The computations needed to control the algorithm are probably more suitable for an MIMD machine, where different processors can do different control computations in parallel. If the size of the prefix computation were fixed, a set of index vectors could be precomputed for an SIMD machine and stored for use by the computation.

A problem for both SIMD and MIMD machines is that the amount of parallelism at different levels varies a lot. Thus if there are enough processors or arithmetic units to do the widest level in parallel, many of them will be idle on other levels, and the efficiency will be low.

Problem: 2.8 Prove by induction the upper bound on Ladner and Fischer’s parallel prefix calculation, i.e.

$$S_k < 2\left(1 + \frac{1}{2^k}\right)n - 2$$

for all \(n \geq 1\). HINT: First show that \(S_0 < 4n - 2\) for all \(n > 1\).

Solution: 2.8 Lemma: The size of Ladner and Fisher’s \(P_0\) algorithm \(S_0(n) < 4n - 2\) for all \(n \geq 1\).

Proof: Assume \(S_0(m) < 4m - 2\), for all \(m \leq 2^k\). Now assume \(n \leq 2^{k+1}\). We know from Ladner and Fisher’s constructions that

\[
S_0(n) = S_1([n/2]) + S_0([n/2]) + n/2,
\]

\[
S_j(n) = S_{j-1}([n/2]) + n - 1, \text{ for } j \geq 1 \text{ and } n \text{ even } \geq 2,
\]

\[
= S_{j-1}([n/2]) + n - 2, \text{ for } j \geq 1 \text{ and } n \text{ odd } \geq 3.
\]

Combining these formulae, we have

\[
S_j(n) = S_j([n/2]) + [n/2] + [n/2] + S_0([n/2]) - 1 \text{ if } [n/2] \text{ even },
\]

\[
= 2 \text{ if } [n/2] \text{ odd }.
\]

Since both \([n/2]\) and \([n/2]/2\] are \(\leq 2^k\), the induction hypothesis holds for them.

The values of \([n/2]\), \([n/2]\), and \([n/2]/2\] differ depending on whether \(n\) is twice an even integer, twice an odd integer, or one of these plus one. Therefore, let \(n = 4q + r, 0 \leq r \leq 3\).

If \(r = 0\): \(S_0(n) = S_0(q) + 2q + S_0(2q) + 2 = 16q - 5 = 4n - 5\);

If \(r = 1\): \(S_0(n) = S_0(q + 1) + 2q + 1 + S_0(2q) + 2q - 2 = 16q - 1 = 4n - 5\);

If \(r = 2\): \(S_0(n) = S_0(q + 1) + 2q + 1 + S_0(2q + 1) + 2q + 1 - 2 = 16q + 4 = 4n - 4\);

If \(r = 3\): \(S_0(n) = S_0(q + 1) + 2q + 2 + S_0(2q + 1) + 2q + 1 - 1 = 16q + 6 = 4n - 6\).

To get the induction started correctly, we want the induction steps to satisfy even \([n/2] \geq 2\) and odd \([n/2] \geq 3\), so we show the result explicitly for \(n = 1, 2, 3\).

\(S_0(1) = 0 < 4 \cdot 1 - 2 = 2\),

\(S_0(2) = 1 < 4 \cdot 2 - 2 = 6\),

\(S_0(3) = 2 < 4 \cdot 3 - 2 = 10\).

Finite induction completes the proof of the lemma. Q.E.D.

Theorem: The size of Ladner and Fisher’s algorithm \(P_k(n)\) satisfies

\[
S_k < 2\left(1 + \frac{1}{2^k}\right)n - 2\text{ for all } n \geq 0 \text{ and } n \geq 1.
\]
Proof: Taking \( k = 0 \) in the conclusion, we see that the lemma proves the result for all \( n \geq 1 \). Assuming as an inductive hypothesis that the result holds for some integer \( k \) and all \( n \geq 1 \), we can compute

\[
S_{k+1}(n) = S_k\left(\left\lceil \frac{n}{2} \right\rceil \right) + n - 1, \quad n \text{ even,}
\]
\[
\frac{n}{2}, \quad n \text{ odd.}
\]

Using the inductive hypothesis for \( S_k\left(\left\lceil \frac{n}{2} \right\rceil \right) \), we have

\[
S_{k+1}(n) < 2\left\lceil \frac{n}{2} \right\rceil + \frac{2^k}{2^k \cdot n - 2} - 2 + n - 1, \quad n \text{ even,}
\]
\[
2n + \frac{2^k}{2^k + 1} - 2, \quad n \text{ odd.}
\]

In both cases

\[
S_{k+1}(n) < 2\left(1 + \frac{1}{2^k} \right)n - 2.
\]

Finite induction completes the proof.

**Problem: 2.9** Assume that we have \( N = 2^j \) data structures, and that the amount of memory required by each is given by \( \text{len}_i, i = 0, 1, \ldots, N-1 \). It is desired to determine for all \( k \) whether data structures \( 0, 1, \ldots, k-1, k \) will all fit in a memory area of size \( M \). The result of the computation is a logical vector \( \text{fit}_i, i = 0, 1, \ldots, N-1 \) such that \( \text{fit}[k] = \text{true} \) if and only if data structures \( 0, 1, \ldots, k-1, k \) will all fit together in a memory area of size \( M \).

(a) What is the size of a sequential algorithm to perform this computation? Note: Comparison of two numbers is counted the same as any other arithmetic operation.

(b) What is the minimum depth for a parallel algorithm to perform this computation?

(c) For \( N = 8 \) show the dependence graph of an algorithm of size no more than 19 operations and depth no more than 5 that does the computation.

**Solution: 2.9** (a) Let \( S[i] \) be the sum prefix vector resulting from \( \text{len}_i \), that is, the sum of all the memory requirements up to and including the \( i \)th data structure. Then \( \text{fit}[i] = (S[i] \leq M) \). The sequential sum prefix has size \( N - 1 \), and there are \( N \) comparisons at the end. Thus the sequential algorithm has size \( 2N - 1 \).

(b) We know that sum prefix can be done with a depth \( \lceil \log_2 N \rceil \) algorithm. Parallel comparisons at the end add one to the depth. The minimum parallel depth is thus \( \lceil \log_2 N \rceil + 1 \).

(c) Minimum depth would be 4 but with \( 12 + 8 = 20 \) operations. Odd/even prefix has fewer operations, and for \( N = 8 \) it has depth \( 2 \times 3 - 2 = 4 \).
**Problem: 2.10** Assuming an arbitrary number of processors, show a fast way to evaluate the expression:

\[ v_1 (1 - p) p^{i-1} + \sum_{i=1}^{8} v_i p - \sum_{i=1}^{8} v_i p \cdot i = 1 \]

(Hint: Consider doing a dot product of the vector \( v \) with a vector of values you produce in parallel from \( p \) and the constant 1.)

(a) Show the data dependence diagram for your parallel algorithm.

(b) If the upper limit on the summation is \( N \) instead of 8, what would the depth and size of your algorithm be?

**Solution: 2.10** The expression is:

\[ \sum_{i=1}^{8} v_i \times p_i = v \cdot p \]

where \( p_i = (1 - p) p^{i-1} \) and each \( p_i \) can be calculated in parallel as shown:

(a) The algorithm could then be:

(b) With arbitrary limit \( N \), the depth would be \( \lceil \log_2(2N) \rceil \) for the second part and \( \lceil \log_2 N \rceil + 1 \) for the first part. The total depth would be \( 2 \cdot \lceil \log_2 N \rceil + 2 \)

**Problem: 2.11** \( N \) blocks of data of different sizes \( S_i \), \( 0 \leq i \leq N-1 \), are to be packed into a single area of memory, with block \( i \) starting immediately after block \( i-1 \). \( N \) processors can move the blocks in parallel once their starting addresses are known. Describe a parallel algorithm to compute the starting addresses in minimum time. Determine the number of processors required to execute the algorithm in minimum time, the time required, and the number of operations performed.
Solution: 2.11 If \( A \) is the base address of the memory area, then block start addresses are \( A, A + S_0, A + S_0 + S_1, \ldots \). This is just the result of a sum prefix computation on the vector \( A, S_0, S_1, \ldots \), which can be done by any of the parallel prefix algorithms. The recursive upper/lower prefix algorithm is one that will execute in the minimum time of \( \lceil \log_2 N \rceil \). The number of processors needed is \( \lfloor N/2 \rfloor \), and the number of operations performed is \( \lceil N/2 \rceil \lceil \log_2 N \rceil \).

Problem: 2.12 An arithmetic expression with \( N \) atoms can be transformed using commutativity and associativity so that its execution time using an optimal number of processors is bounded above by \( \lceil \log_2 N \rceil + 2d + 1 \). It can be transformed using commutativity, associativity, and distributivity so that its execution time with an (perhaps different) optimal number of processors is bounded above by \( \lceil 4\log_2 N \rceil \).

What is \( d \)? Why does it appear in the expression for one upper bound, but not the other?

Solution: 2.12 \( d \) is the depth of parentheses nesting. Each level of essential parenthesis nesting increases the depth of the evaluation tree. By using distributivity, we can eliminate the parentheses and the corresponding dependences, thus minimizing the depth of the algorithm, but at the cost of increasing the size of the algorithm.

Problem: 2.13 Consider the arithmetic expression: \( x = \frac{(a + bcd)e + f}{g} + pq \).

(a) With the normal mathematical evaluation order, what is the size and depth of the expression tree?

(b) Using only associativity and commutativity, transform the expression so that the tree has reduced depth. What are the new depth and size?

(c) Using distributivity as well as associativity and commutativity, reduce the depth further. What are the size and depth of this expression tree?

Solution: 2.13 (a) With \( N=9 \) atoms and sequential evaluation: Depth = Size = \( N-1 = 8 \).

(b) Depth = 7, Size = 8.

(c) The expression becomes: \( \frac{ae}{g} + pq + \frac{f}{g} + \frac{bcde}{g} = \left( \frac{ae}{g} + \left( \frac{pq + \frac{f}{g}}{g} \right) \right) + bc \left( \frac{d}{g} \right) \) and we get an eval-
Problem: 2.14 Use associativity, commutativity and distributivity to transform the expression \( a + b/c - d + e((f + g)/h)/x + y \) so that it can be evaluated in minimum time. Show the evaluation tree and find the minimum number \( \pi \) of processors needed for the minimum time evaluation, along with the time \( T_\pi \), speedup \( S_\pi \), and efficiency \( E_\pi \).

Solution: 2.14 The expression \( a + b/c - d + e((f + g)/h)/x + y \) can be distributed to get 
\[
    a - d + b/c + e(f/x + e + g/h + e) + y,
\]
and rearranged to get \((a - d) + b/c + e + y)(f + g/h)\). This can be evaluated by the expression tree of depth 4 shown below. There are 9 operations in the original expression, corresponding to the time with one processor, and at most 4 operations need to be done in parallel to evaluate the tree. Thus \( \pi = 4 \), \( T_\pi = 4 \), \( S_\pi = 9/4 = 2.25 \), and \( E_\pi = 9/16 = 56\% \).

Problem: 2.15 Which of the following statements about Gaussian elimination can be concluded just from knowing that the size of Gaussian elimination on an \( n \times n \) matrix is \( \Theta(n^3) \)?

1. The size of Gaussian elimination for \( n = 3 \) is about 27.
2. For large \( n \), doubling \( n \) increases the size by 8 times.
3. The size of Gaussian elimination increases monotonically as \( n \) increases.
4. Tripling \( n \) increases the size by 27 times.
5. The size increases monotonically as \( n \) increases for large \( n \).

Solution: 2.15 (a) No. We only know something for \( n \) greater than some large \( N \).
(b) Yes.
(c) No. This is not true for small \( n \).
(d) No. This is not true for small \( n \).
(e) No. The size can vary up and down between narrowing limits.

Problem: 2.16 Suppose that a program obeys Amdahl's law with a sequential fraction \( f = 5\% \) and that the execution time with one processor is 10 seconds. Graph \( T(P) \), \( S(P) \), and \( E(P) \) for \( 1 \leq P \leq 100 \).
Solution: 2.16

\[ T(P) = \left(0.5 + \frac{9.5}{P}\right) \text{ seconds} \]

\[ S(P) = \frac{10}{0.5 + 9.5/P} \]

The time decreases to an asymptote of 0.5 seconds, and the speedup can get no higher than 20.

\[ E(P) = \frac{10}{0.5P + 9.5} \]

The efficiency becomes 50% for 21 processes and goes to zero for an unlimited number of processes.

**Problem: 2.17** Assume that the size of a problem obeying Amdahl’s law is increased as the number, \( P \), of processors increases. The absolute amount of sequential work is a constant, \( S \) (in seconds), but the amount of Parallel work, \( Q \), increases with problem size, \( N \), as \( Q = qN \). The problem size scales with the number of processes as \( N = cP \).

For \( P = 10 \) processors, an appropriate size problem has \( N = 1,000 \) and a sequential fraction of 5%. Graph the speedup and efficiency from \( P = 10 \) to \( P = 100 \) in steps of 10 processes.

**Solution: 2.17** The dependence of the amount of parallel work on the problem size, \( N \), and the dependence of \( N \) on the number of processes, \( P \), makes the Amdahl’s law execution time independent of \( P \):

\[ T(P) = S + \frac{qN}{P} = S + \frac{qcP}{P} = S + qc \]

but the total work, or execution time with one process, depends on \( P \): \( T(1) = S + Q = S + qcP \).
The constants $q$ and $c$ can be evaluated from the $P = 10$ information. If $N = 1,000$ for $P = 10$, then $c = 100$. A sequential fraction of 5% for $P = 10$ means that $f = 0.05 = \frac{S}{S + 1000q}$, or $q = 0.19S$.

$$S(P) = \frac{1 + 19P}{20}$$

$$E(P) = \frac{19 + 1/P}{20}$$

**Problem 2.18** In a certain computation, 90% of the work is vectorizable. Of the remaining 10%, half is parallelizable for an MIMD machine. What are the speedups over sequential execution for a 10 PE SIMD machine and a 10 processor MIMD machine on this computation?

**Solution 2.18** For the SIMD machine, Amdahl’s law takes the fraction of parallel work as 90% and the fraction of serial work as 10% for a time $T_{10} = 0.1 \cdot T_1 + 0.9 \cdot T_1/10$ and a speedup of

$$\frac{T_1}{(0.1 + 0.9/10)T_1} = \frac{1}{0.19} = 5.26.$$

For the MIMD machine, the parallel fraction is 95%, since anything that can be vectorized can also be done in parallel by multiple processors. This gives a time with 10 processors of

$$T_{10} = 0.05 \cdot T_1 + 0.95 \cdot T_1/10$$

and a speedup of

$$\frac{T_1}{(0.05 + 0.95/10)T_1} = \frac{1}{0.145} = 6.90.$$

The SIMD machine only gets the effect of about 5.3 processing elements out of 10, while the MIMD machine gets the effect of 6.9 processors out of 10 because 5% more work can be done in parallel.

**Problem 2.19** Consider a parallel computer that can execute at three different degrees of parallelism at rates $R_1 < R_2 < R_3$. These correspond to using 1, 2, and 3 processors in an MIMD computer or 1, 2, and 3 arithmetic units in an SIMD computer. Suppose the fraction of total work $W$ that can be executed at rate $R_i$ in a particular program is $f_i$, where $f_1 + f_2 + f_3 = 1$. Work can be measured in terms of execution time on a single processor.

(a) Give an expression for the harmonic mean execution rate $R$ of the parallel computer in terms of $f_i$ and $R_i$. How is the execution time of the parallel computer related to $R$?

(b) What is the execution time $T$ of the parallel computer on the given program if $f_1 = 0.5$, $f_2 = 0.3$, $f_3 = 0.2$, and $R_1 = 6$ MIPS, $R_2 = 11$ MIPS, $R_3 = 16$ MIPS. Why might we not have $R_2 = 2R_1$ and $R_3 = 3R_1$?

(c) Suppose the program is rewritten to be more parallel, so that $f_1 = 0.1$, $f_2 = 0.3$, and $f_3 = 0.6$, with the same $R_i$ as in part b). What would the harmonic mean rate and corresponding execution time be in this case?
Solution: 2.19 (a)

\[ R = \left( \frac{f_1}{R_1} + \frac{f_2}{R_2} + \frac{f_3}{R_3} \right)^{-1}. \]

If the total number of operations to be done is \( W \), the execution time is \( T \) and rates are in operations per second.

\[ T = \frac{W}{R} = \frac{f_1 \cdot W}{R_1} + \frac{f_2 \cdot W}{R_2} + \frac{f_3 \cdot W}{R_3}. \]

(b)

\[ T = W \left( \frac{0.5}{6} + \frac{0.3}{11} + \frac{0.2}{16} \right) = \frac{W}{8.12} \]

where \( T \) is in seconds if \( W \) is in millions of operations. The effective rate is 8.12 MIPS.

\( R_i < i \cdot R_1 \) because there is usually some overhead in using multiple operating units that is not present using one.

(c) \( \frac{1}{R} = \left( \frac{0.1}{6} + \frac{0.3}{11} + \frac{0.6}{16} \right) \) or \( R = 12.27 \) MIPS.
Chapter 3: Solutions

Problem: 3.1 Write pseudo code for the \( ikj \) form of matrix multiplication and describe the algorithm in words.

Solution: 3.1 The algorithm is described as follows:

1. Initialize row \( i \) of \( C \) to zero;
2. Form the \( N \) element product vector of row \( i \) of \( A \) with matrix \( B \);
3. Add the \( N \) element vector of product terms to row \( i \) of \( C \);
4. Repeat all steps for all values of \( i \).

\[
\text{for } i := 1 \text{ step } 1 \text{ until } N \text{ begin}
\]
\[
\text{C}[i,j] := 0, \quad (1 \leq j \leq N);
\]
\[
\text{for } k := 1 \text{ step } 1 \text{ until } N \text{ begin}
\]
\[
\text{C}[i,j] := \text{C}[i,j] + A[i,k]*B[k,j], \quad (1 \leq j \leq N);
\]
\[
\text{end ;}
\]

Problem: 3.2 Counting floating point operations is a way of estimating execution time. Use it to analyze the following code fragment written in SIMD pseudo code for a true SIMD computer with \( N \) processing elements. The scalars \( x, y, \) and \( s \) and the vectors \( a, b, \) and \( v \) are all floating point. Scalar operations are done using only one of the PEs.

\[
s := (x - y)/(x*x + y*y);
\]
\[
v[i] := s*a[i] + b[i], \quad (0 \leq i < N);
\]

(a) Give the speedup obtained by executing the above code on the SIMD (vector) machine versus doing the same computation on a SISD (scalar) computer.

(b) What is the largest integer value of \( N \) for which the processing element efficiency is still at least 50%?

Solution: 3.2 For a SISD machine, we have \( 5+2N \) operations;

For a SIMD machine, we have \( 5+2 = 7 \) operations.

(a) \[ \text{Speedup} = \frac{T_1}{T_N} = \frac{5 + 2N}{7} \]
(b) **Efficiency** = \( \frac{\text{Speedup}}{N} = \frac{2N + 5}{7N} \).

For efficiency \( \geq 50\% \), we get \( N \leq \frac{10}{3} \), which is equivalent to about 3 PEs.

**Problem:** 3.3  An SIMD computer’s vector unit executes at an average rate of 100 MFLOPS and its scalar unit averages 20 MFLOPS. What is the average execution rate of this machine on an algorithm in which 90% of the work is vectorized and 10% is scalar work?

**Solution:** 3.3  Since we are dealing with execution rates, we need to use the weighted harmonic mean:

\[
R_a = \left( \frac{1}{R_v} + \frac{1}{R_s} \right)^{-1} = \left( \frac{0.1}{20} + \frac{0.9}{100} \right)^{-1} = 71.4 \text{ MFLOPS}.
\]

**Problem:** 3.4  An SIMD computer has a scalar execution rate of \( R_s = 10 \) MFLOPS and a vector execution rate of \( R_v = 100 \) MFLOPS.

(a)  What is the average execution rate for a program in which 50% of the work is scalar and 50% vector?

(b)  What is the average execution rate for a program that spends half its time executing in scalar mode and the other half in vector mode?

**Solution:** 3.4  (a)  Similarly to (2.3), we have:

\[
R_a = \left( \frac{0.5}{10} + \frac{0.5}{100} \right)^{-1} = 18.2 \text{ MFLOPS}.
\]

(b)  If half the time is spent executing in vector mode and half the time in scalar mode the average rate is the arithmetic mean of the rates, \( R_a = 0.5 \times 10 + 0.5 \times 100 = 55 \text{ MFLOPS} \).

**Problem:** 3.5  Write efficient vector pseudo code for a SIMD machine with \( N \) processing elements to do the following array computation.

\[
\text{for } i := 2 \text{ step } 1 \text{ until } N \\
\quad \text{for } j := 2 \text{ step } 1 \text{ until } N \\
\quad \quad X[i, j] := (X[i, j-1] + X[i, j+1]) / 2;
\]

**Solution:** 3.5  The sequential computation uses old values for \( X[i, j+1] \) and new values for \( X[i, j-1] \), which would not be the case if the \( j \) loop were made into a vector assignment. There is an easy way to exploit the parallelism between the rows of \( X \) by exchanging the inner and outer loops:

\[
\text{for } j := 2 \text{ step } 1 \text{ until } N \\
\quad X[i, j] := (X[i, j-1] + X[i, j+1]) / 2, \ (2 \leq i \leq N);
\]

**Problem:** 3.6  The following sequential pseudo code specifies a matrix-vector multiply:

\[
\text{for } i := 1 \text{ step } 1 \text{ until } n \\
\quad \text{begin} \\
\quad \quad y[i] := 0; \\
\quad \quad \text{end} \\
\quad \text{for } j := 1 \text{ step } 1 \text{ until } n \\
\quad \quad y[i] := y[i] + a[i, j]*x[j];
\]

(a) Using SIMD pseudo code, write the matrix-vector multiply for a vector processor in which memory access is unrestricted.

(b) If the pseudo code of part (a) is executed by a true SIMD machine in which each of \( n \)
PEs has direct access only to its own memory module, how should the arrays $x$, $y$, and $a$ be stored?

**Solution: 3.6 (a):**

\[
y[i] := 0 \quad (1 \leq i \leq n);
\]

\[
\text{for } j := 1 \text{ step } 1 \text{ until } n
\]

\[
y[i] := y[i] + a[i,j] \times x[j] \quad (1 \leq i \leq n);
\]

(b) Each PE memory $M[i]$ should contain element $y[i]$, row $a[i, ·]$, and element $x[i]$ which would be broadcast to all PEs by PE $j$. Alternatively, the entire vector $x$ could be placed in the control unit memory, which would take care of broadcasting its elements to all PEs.

**Problem: 3.7** In one form of scaling a set of linear equations, each row of a matrix $A$ is multiplied by the reciprocal of the maximum absolute value of any element in that row. A sequential algorithm for this would appear as follows.

\[
\text{for } i := 1 \text{ step } 1 \text{ until } n
\]

begin

\[
\text{max} := 0;
\]

\[
\text{for } j := 1 \text{ step } 1 \text{ until } n
\]

\[
\text{if } \text{abs}(A(i,j)) > \text{max} \text{ then } \text{max} := \text{abs}(A(i,j));
\]

\[
r := 1/\text{max};
\]

\[
\text{for } j := 1 \text{ step } 1 \text{ until } n
\]

\[
A(i,j) := r \times A(i,j);
\]

end;

(a) Using high level pseudo code for an SIMD computer, write a vector version of this algorithm.

(b) If the SIMD machine of part (a) has separate memories for each processing element, show the storage layout required by your solution in part (a).

**Solution: 3.7 (a)** We need to use the vector mask $m$ to selectively enable the PEs during the computation of $\text{max}$, which now needs to be a vector:

\[
\text{max}[i] := 0 \quad (1 \leq i \leq n);
\]

\[
\text{for } j := 1 \text{ step } 1 \text{ until } n \text{ begin}
\]

\[
m_i := \text{ABS}(A[i,j]) > \text{max}[i] \quad (1 \leq i \leq n);
\]

\[
\text{max}[i] := \text{ABS}(A[i,j]) \quad (m_i \mid 1 \leq i \leq n);
\]

end;

\[
\text{for } j := 1 \text{ step } 1 \text{ until } n \text{ begin}
\]

\[
m_i := \text{max}[i] \neq 0 \quad (1 \leq i \leq n);
\]

\[
A[i,j] := A[i,j]/\text{max}[i] \quad (m_i \mid 1 \leq i \leq n);
\]

end;

(b) Storage layout:

\[
\begin{array}{cccc}
M_1 & M_2 & \ldots & M_n \\
\text{max}[1] & \text{max}[2] & \ldots & \text{max}[n]
\end{array}
\]

**Problem: 3.8** Back substitution in Gaussian elimination is the solution of $y = Ux$, where $y = (y_1, y_2, \ldots, y_n)$ is given, $x = (x_1, x_2, \ldots, x_n)$ is to be found and $U$ is an upper triangular matrix of the form:
Working backwards from $x_n$, this is an $R(n, n-1)$ recurrence system:

$$x_j = \frac{1}{u_{ii}} \left( y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right), \quad i = n, n-1, \ldots, 2, 1,$$

which can be solved in parallel by the column sweep method.

Write vector pseudo code for the column sweep and indicate the memory layout required to achieve parallelism on an SIMD machine with separate memory-arithmetic unit pairs. You may assume $n$ arithmetic units.

**Solution: 3.8**

```plaintext
x[i] := y[i], (1 ≤ i ≤ n);

x[n] := x[n] / U[n,n];

tmp[i] := broadcast x[n], (1 ≤ i < n);

for j := n-1 step -1 until 1 begin
    x[i] := x[i] - U[i,j+1]*tmp[i], (1 ≤ i ≤ j);
    x[j] := x[j] / U[j,j];
    tmp[i] := broadcast x[j], (1 ≤ i < j);
end;
```

Memory layout:

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
<th>...</th>
<th>$M_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x[1]$</td>
<td>$x[2]$</td>
<td>...</td>
<td>$x[n]$</td>
</tr>
<tr>
<td>$y[1]$</td>
<td>$y[2]$</td>
<td>...</td>
<td>$y[n]$</td>
</tr>
<tr>
<td>$U[1,1:n]$</td>
<td>$U[2,1:n]$</td>
<td>...</td>
<td>$U[n,1:n]$</td>
</tr>
</tbody>
</table>

**Problem: 3.9** The sequential pseudo code below is to be parallelized for an SIMD computer with 32 PEs.

```plaintext
for i := 1 step 1 until n
    x[i] := (x[i] - y[i]) / y[i];
```

(a) Ignoring questions of data layout, strip mine the above loop for execution on 32 PEs. $n - k$ may be much larger than 32, but does not have to be. Express your result using the SIMD pseudo code extensions.

(b) Suppose there are 32 memory modules with module $i$ connected to PE number $i$. Suppose also that $x[j]$ and $y[j]$ are stored in memory module $(j \mod 32)$. Modify the strip mined code of part (a) for this case.
Solution: 3.9

(a):

\[ Q := \lfloor (n-k+1)/32 \rfloor \]  
/* number of full blocks */
\[ R := (n-k+1) \mod 32; \]  
/* number of remaining elements */
\[ s := k; \]  
/* starting index for each block */

/* compute full blocks (if any) */
\begin{verbatim}
for j := 1 step 1 until Q begin
    x[i+s] := (x[i+s] - y[i+s]) / y[i+s], (0 \leq i \leq 31);
    s := s + 32;
end;
\end{verbatim}

/* compute partial block (if any remaining elements) */
\[ x[i+s] := (x[i+s] - y[i+s]) / y[i+s], (0 \leq i < R); \]

(b):

\[ Qk := \lfloor k/32 \rfloor \]  
/* beginning block number */
\[ Rk := k \mod 32; \]  
/* beginning offset in block */
\[ Qn := \lfloor n/32 \rfloor \]  
/* finishing block number */
\[ Rn := n \mod 32; \]  
/* finishing offset in block */
\[ s := 32*Qk; \]  
/* starting index aligned at offset 0 */

/* compute (Qn–Qk+1) iterations */
\begin{verbatim}
for j := Qk step 1 until Qn begin
    if (j = Qk) then
        s1 = Rk;  /* first iteration */
    else
        s1 = 0;  /* any other than first */
    if (j = Qn) then
        s2 = Rn;  /* last iteration */
    else
        s2 = 31;  /* any other than last */
    x[i+s] := (x[i+s] - y[i+s]) / y[i+s], (s1 \leq i \leq s2);
    s := s + 32;
end;
\end{verbatim}

Problem: 3.10

In a “true” SIMD computer with N processing elements and N memories, it is sometimes necessary to access rows of a matrix as parallel vectors, while at other times it is necessary to access all elements of a column in parallel. Show a storage pattern for the elements of an N by N matrix which will support both these needs.

What addition to the PE hardware will allow such different accesses to be done on matrices using a “straight” storage scheme, either by rows or by columns?

Solution: 3.10

The skewed storage scheme supports both these needs, and is shown below:

\[
\begin{array}{ccc}
M_1 & M_2 & M_3 \\
A_{1,1} & A_{1,2} & A_{1,3} \\
A_{2,1} & A_{2,2} & A_{2,3} \\
A_{3,1} & A_{3,2} & A_{3,3} \\
\end{array}
\]

A PE index register allows for row and column accesses even if the matrix is stored in a “straight” scheme.
**Problem: 3.11** Consider an SIMD or vector processor with multiple arithmetic units and a single control unit.

(a) Use vector processor pseudo-code to write a program to compute the square of an \( n \times n \) matrix \( X \). Call the result \( Y \) and assume that any arithmetic unit can access any memory cell.

(b) Show how to store data (and modify the program if necessary) to do the matrix squaring in Illiac IV type machine where \( n \) arithmetic units are attached to \( n \) private memories.

**Solution: 3.11** The mathematical statement of the problem is:

\[
Y_{ij} = \sum_{k=1}^{n} X_{ik} \cdot X_{kj}, 1 \leq i, j \leq n
\]

(a) A sequential program might look like:

```plaintext
for i := 1 step 1 until n
    for j := 1 step 1 until n
    begin
        sum := 0 ;
        for k := 1 step 1 until n
            sum := sum + X[i,k] * X[k,j] ;
        Y[i,j] := sum ;
    end
end;
```

To make an efficient vector program, interchange the two inner loops and do vector operations over \( j \). A sum vector must be introduced to do this.

```plaintext
for i := 1 step 1 until n
begin
    sum[j] := 0, (1 \leq j \leq n) ;
    for k := 1 step 1 until n
    begin
        sum[j] := sum[j] + X[i,k] * X[k,j], (1 \leq j \leq n);
        Y[i,j] := sum[j], (1 \leq j \leq n);
    end
end;
```

(b) To use \( n \) independent arithmetic unit-memory pairs, \( X \) should be stored with a column in each memory, that is, \( X[\cdot,j] \) is stored in memory \( j \). The sums should be distributed across the memories, and can, in fact, be placed directly in \( Y \) which is stored the same way as \( X \). The only inter PE communication needed is a broadcast of \( X[i,k] \) from PE \( k \) to all others.

```plaintext
for i := 1 step 1 until n
begin
    Y[i,j] := 0, (1 \leq j \leq n) ;
    for k := 1 step 1 until n
    begin
        t[j] := broadcast(X[i,k]), (1 \leq j \leq n);
        Y[i,j] := Y[i,j] + t[j] * X[k,j], (1 \leq j \leq n);
    end
end;
```

**Problem: 3.12** Consider an SIMD computer with \( N \) PEs, each having a private memory and capable of routing by a shift of one, either to the left or right. Find a storage layout for an \( N \) by \( N \) matrix such
that column sums can be done optimally, in parallel, without routing, while row sums can be done in parallel using routing, but not requiring PE indexing. Show pseudo code to accomplish row and column sums with this layout.

**Solution: 3.12** The storage layout can be by forward diagonals stored across the memories with location \( \text{loc}[i, k] \) containing \( A[(i+k-2) \mod N + 1, k] \).

\[
\begin{array}{cccccc}
M_1 & M_2 & M_3 & \cdots & M_N \\
A_{1,1} & A_{2,2} & A_{3,3} & \cdots & A_{N,N} \\
A_{2,1} & A_{3,2} & A_{4,3} & \cdots & A_{1,N} \\
A_{3,1} & A_{4,2} & A_{5,3} & \cdots & A_{2,N} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
A_{N,1} & A_{1,2} & A_{2,3} & \cdots & A_{N-1,N} \\
\end{array}
\]

The column sums can be computed with no communication among PEs.

\[
\text{sum}[k] := 0, \ (1 \leq k \leq N); \\
\text{for } i := 1 \ \text{step 1 until } N \\
\quad \text{sum}[k] := \text{sum}[k] + A[(i+k-2) \mod N + 1, k], \ (1 \leq k \leq N);
\]

To compute row sums, the sum vector is shifted left one after each diagonal is added.

\[
\text{sum}[k] := 0, \ (1 \leq k \leq N); \\
\text{for } i := 1 \ \text{step 1 until } N \begin{align*}
\quad \text{sum}[k] & := \text{sum}[k] + A[(i+k-2) \mod N + 1, k], \ (1 \leq k \leq N); \\
\quad \text{sum}[k] & := \text{SHIFT} \ (\text{sum}[k+1], -1), \ (1 \leq k \leq N);
\end{align*}
\]

**Problem: 3.13** On a machine having no maximum operation, the computation of an \( N \) element vector \( C \) whose elements are the maxima of corresponding elements of vectors \( A \) and \( B \) can be done by comparisons.

\[
\text{for } i := 0 \ \text{step 1 until } N-1 \\
\quad \text{if } A[i] > B[i] \ \text{then } C[i] := A[i] \\
\quad \quad \text{else } C[i] := B[i];
\]

Using vector pseudo code, show how to do this operation on an SIMD machine with \( N \) PEs. What is the efficiency?

**Solution: 3.13** We need to use the vector mask to record the results of the test \( A[i] > B[i] \) and then assign \( A[i] \) to \( C[i] \) only for those PEs with true results and \( B[i] \) to \( C[i] \) only for those with false results.

\[
m_k := A[k] > B[k], \ (0 \leq k \leq N-1); \\
C[k] := A[k] \quad (m_k \mid 0 \leq k \leq N-1) ; \\
m := \text{not} \ m; \\
C[k] := B[k] \quad (m_k \mid 0 \leq k \leq N-1) ;
\]

The test operation is done at full efficiency of one, since all PEs participate and yield a useful result. The assignments are performed at 50% efficiency since each PE is idle for one of the two assignments.

**Problem: 3.14** The following calculation, expressed in sequential pseudo code is to be vectorized for a true
SIMD machine of the type considered in class having \( N \) PEs.

\[
\text{for } i := 1 \text{ step 1 until } N \\
\quad \text{if } (a[i] \geq 0) \text{ then} \\
\quad \quad v[i] := (w + b[i])/(b[i] + a[i]) \\
\quad \text{else} \\
\quad \quad v[i] := (w + b[i])/(b[i] - a[i] + z);
\]

Assume that we can estimate the execution time accurately by counting only floating point operations. On the SIMD machine, floating point operations are done by the PEs, and we assume that a vector floating point operation takes the same unit amount of time as a scalar floating point operation on a uniprocessor. All floating point operations take the same time.

(a) Suppose that the elements of the vector \( a[i] \) have a 50% probability of being negative. Give an execution time estimate for the calculation executed sequentially on a uniprocessor.

(b) Parallelize the computation for efficient execution on an \( N \) PE SIMD computer of the type discussed in Section 2.3. Express the parallelized algorithm in a restricted vector pseudo code which has a direct correspondence to the machine code that would be executed so that the execution time can be estimated.

(c) Under the same assumptions on the signs of \( a[i] \) as in part (a), give an execution time estimate for your SIMD computation of part (b).

**Solution: 3.14**

(a) \( T_S = N \times (50\% \times 3 + 50\% \times 4) = 3.5N \) time units

(b) Assuming that a sign test is not a floating point operation, we have:

Instructions: 

\[
\begin{align*}
\text{FP operations:} \\
\text{MASK}[i] &:= a[i] \geq 0, \ (1 \leq i \leq N); & 0 \\
tmp[i] &:= b[i] + a[i], \ (\text{MASK}[1] \leq N); & 1 \\
\text{MASK}[i] &:= \text{NOT} \ \text{MASK}[i], \ (1 \leq i \leq N); & 0 \\
tmp[i] &:= b[i] - a[i] + z, \ (\text{MASK}[1] \leq N); & 2 \\
v[i] &:= (w + b[i])/tmp[i], \ (1 \leq i \leq N); & 2
\end{align*}
\]

(c) \( T_P = 5 \) time units

**Problem: 3.15** Vectorize the following computation for a vector computer with length 64 vector registers. Express the result as SIMD pseudo code using vector statements with stride one index ranges starting at 0 and having restricted length. Hint: vectorize the index calculations.

\[
k := 0; \\
m := 0; \\
\text{for } i := 0 \text{ step 1 until } N-1 \\
\quad \text{begin} \\
\quad \quad a[i] := b[i] + b[m]; \\
\quad \quad k := k + 1; \\
\quad \quad \text{if } (k = 4) \text{ then } \{ k := 0; \ m := m + 1 \}; \\
\quad \text{end};
\]
Solution: 3.15

for x := 0 step 64 until (N-1) begin
    if (x+63) > (N-1) then
        len := N - x;
    else
        len := 64;
    M[i] := \lfloor (x+i)/4 \rfloor, (0 \leq i \leq len);
    a[i+x] := b[i+x] + b[M[i]], (0 \leq i \leq len);
end;

Problem: 3.16

In a program which solves for the roots of N quadratic equations, part of the computation is described by the following sequential pseudo code:

/* a[i], b[i], and c[i] are input vectors and x[i] and y[i] are output vectors, 0 \leq i < N. */
for i := 0 step 1 until N-1 begin
    if a[i]=0 and b[i]=0 then
        begin x[i] := 0;
        y[i] := 0;
        end;
    else if a[i]*b[i] \geq 0 then
        begin x[i] := (a[i] + b[i])/2;
        y[i] := c[i]/x[i];
        end;
    else if a[i]*b[i] < 0 then
        begin y[i] := (a[i] - b[i])/2;
        x[i] := c[i]/y[i];
        end;
end;

(a) Show how this calculation would be done on a “true” SIMD machine with N processing elements. You may use pseudo code, but be sure it is written at a low enough level to describe what happens at the assembly language level. In particular, it should be low-level enough to help answer part b).

(b) Compute the speedup and efficiency of the SIMD parallel version compared to the sequential version. Assume that a comparison takes the same amount of time as any other floating point operation. Make, state, and justify reasonable assumptions about the distribution of outcomes of the three tests.

Solution: 3.16

(a) Here all comparisons are floating point operations:

Instructions: FP operations:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK1[i] := (a[i]=0), (0 \leq i &lt; N);</td>
<td>1</td>
</tr>
<tr>
<td>MASK1[i] := (b[i]=0), (MASK1[i]0 \leq i &lt; N);</td>
<td>1</td>
</tr>
<tr>
<td>x[i] := 0, (MASK1[i]0 \leq i &lt; N);</td>
<td>0</td>
</tr>
<tr>
<td>y[i] := 0, (MASK1[i]0 \leq i &lt; N);</td>
<td>0</td>
</tr>
<tr>
<td>m[i] := a[i]*b[i], (0 \leq i &lt; N);</td>
<td>1</td>
</tr>
<tr>
<td>MASK2[i] := not(MASK1[i]), (0 \leq i &lt; N);</td>
<td>0</td>
</tr>
<tr>
<td>MASK2[i] := (m[i]0 \leq i &lt; N);</td>
<td>1</td>
</tr>
<tr>
<td>y[i] := (a[i]+b[i])/2, (MASK2[i]0 \leq i &lt; N);</td>
<td>2</td>
</tr>
<tr>
<td>x[i] := c[i]/x[i], (MASK2[i]0 \leq i &lt; N);</td>
<td>1</td>
</tr>
</tbody>
</table>
MASK2[i] := not(MASK1[i]), (0 ≤ i < N); 0
MASK2[i] := (m[i]<0), (MASK20 ≤ i < N); 1
x[i] := (a[i]-b[i])/2, (MASK2|0 £ i < N); 2
y[i] := c[i]/y[i], (MASK2|0 £ i < N); 1
end;

(b) The probability of a[i] and b[i] having equal signs should be the same as having different signs. The probability of either one of them being zero should be very small, to the extent that it will not affect performance noticeably. Under these assumptions, we have:

\[ T_S = N \times (2 + 0\% \times 0 + 100\% \times (2 + 50\% \times 3 + 2 + 50\% \times 3)) = 7N \] units of time;

\[ T_P = 11 \] units of time;

Speedup = \( \frac{T_S}{T_P} \) = 0.636N;

Efficiency = Speedup/N = 63.6%.

Problem: 3.17
The sequential code below is to be vectorized for an SIMD computer with N PEs.

for i:=1 step 1 until N
    if a[i] < L
        then b[i] := f(i)
        else b[i] := g(i);
end

The computations f(i) and g(i) are both vectorizable but require different numbers of floating point operations. f(i) uses \( W_f \) operations and g(i) uses \( W_g \) operations.

(a) Write vector pseudo code using a mask to vectorize the above code.

(b) If the probability of a[i] < L is \( p \), give an expression for the efficiency of the vectorized computation, counting only work involved in the floating point operations f and g.

(c) If \( p = 0.5 \), what is the efficiency?

(d) If \( W_g = 2W_f \), over what range does the efficiency vary as \( p \) goes from zero to one?

Solution: 3.17
(a) Strip mining is not necessary with \( N \) PEs, so the pseudo code could be:

\[
\text{mask}(i) := (a[i] < L), \quad (1 \leq i \leq N); \\
b[i] := f(i), \quad (\text{mask}(i) \mid 1 \leq i \leq N); \\
b(i) := g(i), \quad (\text{not mask}(i) \mid 1 \leq i \leq N);
\]

(b) In the sequential code loop body, \( W_f \) work is done with probability \( p \) and \( W_g \) with \((1-p)\).

\[ T_1 = N(pW_f + (1-p)W_g). \]

The SIMD computer goes through both \( f \) and \( g \) with part of the processors active.

\[ T_N = W_f + W_g. \]

The efficiency is thus,

\[ E = \frac{1}{N} \frac{T_1}{T_N} = \frac{pW_f + (1-p)W_g}{W_f + W_g}. \]

(c) For \( p = 0.5 \), \( E = (0.5W_f + 0.5W_g)/(W_f + W_g) = 0.5 \).

(d) With \( W_g = 2W_f \), \( E = \frac{pW_f + 2(1-p)W_f}{W_f + 2W_f} = \frac{2-p}{3} \), so for \( 0 \leq p \leq 1 \), \( E \) is in the range \( 2/3 \geq E \geq 1/3 \).

Problem: 3.18
In a “true” SIMD computer with one control unit and multiple PEs, some if statements will compile into conditional branch instructions and some will not.

(a) How does the compiler determine which if statements will be implemented with condi-
(b) List conditions that machine language branch instructions might test in a “true” SIMD computer.

**Solution: 3.18**

(a) When the if condition varies across vector components a control based conditional (conditional branch) cannot be used. Conditionals based on vector data values can be implemented using the vector mask. Others can be implemented using conditional branches.

(b) If we have a boolean vector which stores the result of the last vector comparison, some useful branch conditions would be

- branch if all;
- branch if any;
- branch if none.

Standard conditionals based only on control unit state: tests on index registers or single data values, would also be included.

**Problem: 3.19**

Below is a fragment of vector pseudo code for a “true” SIMD machine with $P$ processing elements.

```plaintext
clt zero ; Set mask to result of testing PE accumulators <0
brnon L1 ; Branch if no mask bits set
mask ; Set PE enables according to mask
add b ; Floating add vector b to accumulators
L1: not mask,mask ; Complement test vector
brnon L2 ; Branch if no mask bits set
mask ; Set PE enables
sub c ; Floating subtract vector c
L2:
```

Assume the floating point operations are the only instructions that take any time, and each executes in one unit of time. (They really represent larger code bodies.) If the signs of the PE accumulators are uniformly distributed on entry to the code fragment, write an expression for the execution time with $P$ processors. Compared to sequential execution for a length $P$ vector, how much do the branch instructions contribute to the efficiency with $P$ processing elements?

**Solution: 3.19**

Sequential code would execute only one floating point operation per value, depending on the sign. Vector code without branches would execute two operations in any case, resulting in an efficiency of 50%. Including the branches only helps in the cases where all values have the same sign (probability $= 2/2^P$), resulting in a time for length $P$ vectors of

$$T_P = \left(\frac{2}{2^P}\right) \times 1 + \left(1 - \frac{2}{2^P}\right) \times 2,$$

and an efficiency of $E_P = \frac{1}{2 - 2^{1-P}}$.

For large $P$, this tends to 50% efficiency, minimizing the contribution of the branches.

**Problem: 3.20**

An SIMD computer with $N = 2^n$ PEs has an architecture and assembly language like that of the machine described in the text. It uses a PE to PE routing network with a routing register in each PE that can be used in arithmetic or moved to or from the PE accumulator. Instead of shift routing, this machine has $\eta$ exchange routing operations that exchange the contents of routing registers in PE$_i$ and PE$_j$, $j = i \oplus 2^k$, $0 \leq k \leq n - 1$, $0 \leq j \leq N - 1$.

(a) Describe how these routing operations can be used to sum the values in the accumulators of all $N$ PEs.

(b) Prove that the contents of a routing register in any one source PE can be moved to an arbitrary destination PE in $n$ or fewer routing steps.
Solution: 3.20 (a)

\[
\text{for } k := 0 \text{ step } 1 \text{ until } n - 1 \\
\text{begin} \\
\text{copy accumulators to routing registers;} \\
\text{exchange}(k); \\
\text{add routing registers to accumulators;} \\
\text{end;}
\]

After \( n = \log_2 N \) steps, each accumulator will contain the sum of all the original accumulator values.

(b) Let the binary values of the source and destination PE indices be \( s = s_{n-1}s_{n-2}…s_1s_0 \) and \( d = d_{n-1}d_{n-2}…d_1d_0 \), respectively. Compute the tag \( t = s \oplus d \) by bitwise exclusive or, i.e. \( t_k = s_k \oplus d_k, 0 \leq k \leq n - 1 \). Then the following set of exchanges will move the routing register of \( s \) to the routing register of \( d \) (and move other routing registers other places).

\[
\text{for } k := 0 \text{ step } 1 \text{ until } n - 1 \\
\text{if } t_k \neq 0 \text{ then exchange}(k);
\]

Bits of the source index are successively converted to those of the destination index wherever they do not already match.

Problem: 3.21 Consider the “column sweep” algorithm for solving a linear recurrence system \( R(n,m) \).

(a) Using pseudo code extended for SIMD machines write a vector version of the algorithm. Consider PE disabling and data layout in memory.

(b) Use the MIMD extensions to the pseudo code to write a multiprocessor version of column sweep.

(c) Discuss the effect of the order, \( m \), of the system on efficiency in relation to the maximum vector length in the SIMD case and to the number of processors in the MIMD case.

Solution: 3.21 (a) With \( n \) processors, a vector column sweep for a general \( R(n, n-1) \) recurrence only
requires disabling processors which have already calculated their $x_j$.

\[
x[i] := c[i], \ (1 \leq i \leq n);
\]

/* $x[j]$ is complete at this point. */

/* Step through $x[j]$ as they are completed. */

for $j := 1$ step $1$ until $n-1$

  begin
    /* Disable processors for completed $x$’s. */
    \[
    M[i] := (i > j), \ (1 \leq i \leq n);
    \]
    /* $x[j]$ is broadcast. $a[i,j]$ is in memory. */
    \[
    x[i] := x[i] + a[i,j] \times x[j], \ (M[i] \land 1 \leq i \leq n);
    \]
  end;

But, for an R$(n, m)$ recurrence with $m < n-1$, only $m$ PEs are needed, and using $n$ would be wasteful.

Allocation of operations to PEs and storage layout in this case is more complex.

\[
T[i] := c[i], \ (1 \leq i \leq m);
\]

\[
x[1] := T[1];
\]

for $j := 1$ step $1$ until $n-1$

  begin
    Broadcast($x[j]$);
    \[
    M[i] := (j+i \leq n), \ (1 \leq i \leq m);
    \]
    \[
    T[i] := \text{SHIFT}(T[i+1], -1), \ (1 \leq i \leq m);
    \]
    \[
    T[m] := c[j+m];
    \]
    \[
    T[i] := T[i] + a[j+i,j] \times x[j], \ (M[i] \land 1 \leq i \leq m);
    \]
    \[
    x[j+1] := T[1];
    \]
  end;

Storage layout for $n = 8$, $m = 4$:

<table>
<thead>
<tr>
<th></th>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$M_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{21}$</td>
<td>$a_{31}$</td>
<td>$a_{41}$</td>
<td>$a_{51}$</td>
<td></td>
</tr>
<tr>
<td>$a_{32}$</td>
<td>$a_{42}$</td>
<td>$a_{52}$</td>
<td>$a_{62}$</td>
<td></td>
</tr>
<tr>
<td>$a_{43}$</td>
<td>$a_{53}$</td>
<td>$a_{63}$</td>
<td>$a_{73}$</td>
<td></td>
</tr>
<tr>
<td>$a_{54}$</td>
<td>$a_{64}$</td>
<td>$a_{74}$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>$a_{65}$</td>
<td>$a_{75}$</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>$a_{76}$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

(b)

for $i := 1$ step $1$ until $n-1$ fork INIT;

$i := n$;

INIT: $x[i] := c[i]$;

join $n$;

for $j := 1$ step $1$ until $n-1$

  begin
    for $i := j+1$ step $1$ until min($n$, $j+m$)$-1$ fork ADDTERM;
    $i := \text{min}(n, j+m)$;
    ADDTERM: $x[j+i] := x[j+i] + a[j+i, j] \times x[j]$;
    join $\text{min}(n-j, m)$;
  end

This code makes the MIMD machine look like SIMD. With appropriate synchronization primitives (to be discussed) it is possible to avoid the large number of (expensive) forks and joins.
(c) Since no more than \( m \) operations can be done at once, a maximum vector length or number of processors greater than \( m \) is not useful. If the number of processors \( NP \) is less than \( m \), vector operations must be done in \( \lceil m/NP \rceil \) groups of \( NP \) vector components, followed by a group of \( (m \mod NP) \) processors. Processors will be used most efficiently if \( (m \mod NP) = 0 \).

Problem: 3.22

Given an \( n \times n \) matrix \( A = (a_{ij}) \), we want to find the \( n \) column sums:

\[
S_j = \sum_{i=0}^{n-1} a_{ij}, \quad \text{for } j = 0, 1, \ldots, n-1,
\]

using an SIMD machine with \( n \) PEs. The matrix is stored in a skewed format, as shown below. The \( j \)-th column sum \( S_j \) is stored in location \( \beta \) in PEM\(_j\) at the end of the computation. Using the type of SIMD machine described in class, write an SIMD assembly language program for the algorithm and indicate the successive memory contents in the execution of the algorithm.

Solution: 3.22
Location \( \alpha+i \) in PE memory \( k \) contains \( a[i, (k-i) \mod n] \). We can add location \( \alpha+i \) in every PE to a column sum vector that is shifted so that what will become the final \( S[(k-i) \mod n] \) is in PE \( k \) when the add is performed. High level pseudo code:

\[
S[k] := 0, \quad (0 \leq k < n); \quad /*\text{Clear column sums.} */
\]

\[
\text{for } i := 0 \text{ step 1 until } n-1
\]

\[
\begin{align*}
&\text{begin} \\
&\quad /*\text{Add location } a+i \text{ to } b \text{ in all PEs.} */ \\
&\quad S[k] := S[k] + a[i, (k-i) \mod n], \quad (0 \leq k < n); \\
&\quad /*\text{Shift } S \text{ right circular one.} */ \\
&\quad S[k] := \text{shift}(S[(k-1) \mod n], 1), \quad (0 \leq k < n); \\
&\quad \text{end} \quad /*\text{After } n \text{ shifts, } S \text{ will return to the position with } S[k] \text{ in PE } k. */
\end{align*}
\]
Assembly language code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lod zero</code></td>
<td>Get location containing zero in each PE memory.</td>
</tr>
<tr>
<td><code>sto β</code></td>
<td>Clear the column sums.</td>
</tr>
<tr>
<td><code>ldxi L, n-1</code></td>
<td>Load n-1 into a limit register.</td>
</tr>
<tr>
<td><code>ldxi I, 0</code></td>
<td>Initialize the row index.</td>
</tr>
<tr>
<td><code>loop: lod α, I</code></td>
<td>Get ( a[i, (k-i) \mod n] ) and add to ( S[k] ), which will become ( S[(k-i) \mod n] ).</td>
</tr>
<tr>
<td><code>route +1</code></td>
<td>Shift the ( S ) vector</td>
</tr>
<tr>
<td><code>movA toR</code></td>
<td>One more place</td>
</tr>
<tr>
<td><code>sto β</code></td>
<td>Right circularly.</td>
</tr>
<tr>
<td><code>incx I, 1</code></td>
<td>Increment ( i ) and repeat</td>
</tr>
<tr>
<td><code>cmpx I, L, loop</code></td>
<td>Loop until ( i = n ).</td>
</tr>
</tbody>
</table>

All PEs are enabled; location \textit{zero} in each PE memory contains zero; \( L \) and \( I \) are CU index registers.

At step \( i \), location \( β \) in memory \( k \) contains \( \sum_{j=0}^{i} a[j, (k-i) \mod n] \).

**Problem: 3.23** For an SIMD machine with \( N \) arithmetic units, write a program in vector pseudo code to multiply the absolute value of an \( N \) by \( N \) matrix, \( A \), times an \( N \) element vector, \( x \). The pseudo code should correspond directly to the vector assembly language level so that you can answer the performance question that follows. Assume that there are no memory access conflicts. Absolute value is not an elementary operation but must be done with data dependent conditionals.

With a random, uniform distribution of signs, what are the expected values of the speedup and efficiency with respect to an efficient single processor program?

**Solution: 3.23**

Sequential code:

```plaintext
for i := 1 step 1 until N
    y[i] := 0;
for j := 1 step 1 until N
    for i := 1 step 1 until N
        if a[i,j] ≥ 0
            then y[i] := y[i] + a[i,j]×x[j]
        else y[i] := y[i] - a[i,j]×x[j];

Vector code using mask
y[I] := 0, (0 ≤ i ≤ N-1);
for j := 1 step 1 until N
    begin
        m[i] := a[i,j] ≥ 0, (0 ≤ i ≤ N-1);
        y[i] := y[i] + a[i,j]×x[j], (m[i] | 0 ≤ i ≤ N-1);
        m[i] := not m[i], (0 ≤ i ≤ N-1);
        y[i] := y[i] - a[i,j]×x[j], (m[i] | 0 ≤ i ≤ N-1);
    end
```

Counting only floating point operations, the sequential program does \( 2N^2 \) operations, regardless of the pattern of signs of the elements of \( A \). The vector code does \( 4N \) masked floating point operations.
The speedup and efficiency are \( S_N = \frac{2N^2}{4N} = \frac{N}{2} \), \( E_N = \frac{1}{2} \).

Considering other operations, the test of matrix element signs is done in either case, but complementing the mask is an extra step in the vector code.

**Problem: 3.24** The following sequential pseudo code is to be vectorized for an SIMD computer with \( N \) PEs.

```plaintext
for i := 1 step 1 until N
  if (v[i] < a) then x[i] := \exp(v[i] - a)
  else if (v[i] > b) then x[i] := \exp(b - v[i])
  else x[i] := \sin(v[i] - a);
```

(a) Write vector pseudo code with masking to do the SIMD computation. The level of the pseudo code should be low enough to use in answering part (b).

(b) Assume \( \sin() \) and \( \exp() \) each take 20 floating point operations while compare and subtract are each one floating point operation. Counting only floating point operations, what are the speedup and efficiency of the SIMD code if 50\% of the \( v[i] \) are between \( a \) and \( b \) and the rest are equally divided between \( v[i] < a \) and \( v[i] > b \)?

**Solution: 3.24**

(a) Instructions:

<table>
<thead>
<tr>
<th>Mask Condition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v[i] &lt; a )</td>
<td>( x[i] := \exp(v[i] - a) )</td>
</tr>
<tr>
<td>( v[i] &gt; b )</td>
<td>( x[i] := \exp(b - v[i]) )</td>
</tr>
<tr>
<td>( \text{not} (v[i] &lt; a \text{ or } v[i] &gt; b) )</td>
<td>( x[i] := \sin(v[i] - a) )</td>
</tr>
</tbody>
</table>

FP operations:

- \( \text{Mask1}[i] := v[i] < a, (1 \leq i \leq N) \), 1 operation
- \( x[i] := \exp(v[i] - a), (\text{Mask1}[1 \leq i \leq N]) \), 20+1 operations
- \( \text{Mask2}[i] := v[i] > b, (1 \leq i \leq N) \), 1 operation
- \( x[i] := \exp(b - v[i]), (\text{Mask2}[1 \leq i \leq N]) \), 20+1 operations
- \( \text{Mask3}[i] := \text{not} (\text{Mask1}[i] \text{ or } \text{Mask2}[i]), (1 \leq i \leq N) \), 0 operations
- \( x[i] := \sin(v[i] - a), (\text{Mask3}[1 \leq i \leq N]) \), 20+1 operations

(b) \( T_S = T_S = 22.75 \times N \) time units,

\( T_P = 65 \) time units,

Speedup = \( T_S / T_P = 0.35N \),

Efficiency = \( \text{Speedup}/N = 35\% \).

**Problem: 3.25** Refer to the diagram in Fig. 2-21 of the notes for the 4 in 5 skewed storage of an 8\( \times \)8 matrix.

(a) List all useful patterns that can be accessed in one memory cycle. Part of the problem is characterizing a “useful pattern” for matrix access.

(b) Given a linear \( P \)-vector \( V(1, 1, 1, 1) \). Calculate word addresses in memory modules.

**Solution: 3.25**

(a) The 4 matrix elements \( a_{ij} \) of a linear 4 vector \( V(a, b, c, e) \) are in different modules if \( 8a + c \) is relatively prime to 5. Thus the following patterns can be accessed in one cycle.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>( a )</th>
<th>( c )</th>
<th>( 8a + c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 row elements</td>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>4 column elements</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4 forward diagonal elements</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>4 reverse diagonal elements</td>
<td>-1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>odd/even elements of a row</td>
<td>0</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>odd/even elements of a column</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>odd/even elements of a forward diagonal</td>
<td>2</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>odd/even elements of a reverse diagonal</td>
<td>-2</td>
<td>2</td>
<td>14</td>
</tr>
</tbody>
</table>
Note that the forward diagonal with slope 2, i.e. $a_{11}$, $a_{33}$, $a_{74}$, cannot be accessed in one cycle because $8a + c = 10$ and $\gcd(10, 5) = 5$. The 4 elements of a 2x2 square submatrix of $A$ with upper left corner at $a_{ij}$ are not a linear 4 vector, but they can be accessed in one cycle because the 4 numbers, $8j + i + B$, $8(j + 1) + i + B$, $8(j + 1) + i + 1 + B$, where $B$ is the array base address, are all distinct modulo 5.

(b) The P vector $V(1, 1, 1, 1)$ consists of the matrix elements $\{a_{11}, a_{22}, a_{33}, a_{44}\}$. If the array base address is zero, the word addresses in the memory modules are $g(1, 1) = \lfloor 9/4 \rfloor = 2$, $g(2, 2) = \lfloor 8/4 \rfloor = 4$, $g(3, 3) = \lfloor 27/4 \rfloor = 6$, $g(4, 4) = \lfloor 36/4 \rfloor = 9$.

**Problem: 3.26** Turn the SIMD pseudo code in the solution to Problem 2.6 into a Fortran 90 program for upper/lower parallel prefix. Notes: If two variables declared INTEGER, say I and N, are divided in Fortran, the integer result I/N is mathematically $I \div N$ if they are both positive. Fortran has a modulo function, $\text{MODULO}(J, M) = J \mod M$, for integer $J$ and $M$.

**Solution: 3.26**

```fortran
PARAMETER N = 16
INTEGER L2N, TK, K, I, P(0:N/2-1), Q(0:N/2-1)
REAL V(0:N-1)
!
! Ensure that the log base 2 calculation doesn't round wrong.
L2N = INT(LOG(N)/LOG(2)+0.5)
DO K = 0, L2N-1
   TK = 2**K
   IDX = (/ (I, I = 0, N/2-1) /)
   ! (IDX/TK) is integer division of a vector by a constant.
   P = TK*(2*(IDX/TK) + 1)
   Q = P + MODULO(IDX, TK)
   V(Q) = V(Q) + V(P - 1)
END DO
```

**Problem: 3.27**

(a) Assume $V$ is a 31 element vector declared in Fortran 90 as $V(0:30)$. Strip mine the Fortran 90 assignment

$$V(1:30) = 0.5 \cdot (V(1:30) + V(0:29))$$

for a vector computer of 10 PEs. That is, write three Fortran 90 assignments with length 10 vectors that accomplish the same operation.

(b) If $V$ is a 32 element vector declared as $V(0:31)$, strip mine the Fortran 90 assignment

$$V(1:30) = 0.5 \cdot (V(0:29) + V(2:31))$$

for a vector computer of 10 PEs. The bulk of the work should be done by vector assignments of length no more than 10.

**Solution: 3.27(a)** Here we can order the assignments to achieve the right semantics.

$$V(21:30) = 0.5 \cdot (V(21:30) + V(20:29))$$
$$V(11:20) = 0.5 \cdot (V(11:20) + V(10:19))$$
$$V(1:10) = 0.5 \cdot (V(1:10) + V(0:9))$$
(b) Here we need to use temporary variables to get the semantics right.

\[
\begin{align*}
TV10 &= V(10) \\
V(1:10) &= 0.5 \times (V(0:9) + V(2:11)) \\
TV20 &= V(20) \\
V(11:20) &= 0.5 \times ((TV10, V(11:19)) + V(12:21)) \\
V(21:30) &= 0.5 \times ((TV20, V(21:29)) + V(22:31))
\end{align*}
\]

**Problem: 3.28**
The Fortran 90 code below is to be executed on a true SIMD architecture. It is known that the values of \(b(i)\) are equally distributed between zero and one. Comparison of real numbers takes one floating point (fp) operation, and \(\sin()\) and \(\exp()\) each take 20 fp operations.

```fortran
real a(1:N), b(1:N), x(1:N)
where ( b(1:N) > 0.75 )
  x(1:N) = exp( b(1:N) )
elsewhere
  x(1:N) = sin( b(1:N) ) + exp( a(1:N) )
end where
```

(a) If the SIMD machine has \(N\) PEs, compute the speedup and efficiency of vector execution over scalar execution of the same computation, counting only fp operations.

(b) If the machine has only 64 PEs, add Fortran 90 code to represent strip mining the computation when \(N\) is not necessarily a multiple of 64. Compute speedup and efficiency of the strip mined code.

**Solution: 3.28**

(a) There are \(N\) comparisons, \((1/4) \times 20N\) operations for the `WHERE`, and \((3/4) \times (20 + 20 + 1)N\) for the `ELSEWHERE`. \(T_1 = (1 + 5 + 123/4)N = 36.75N\), and \(T_N = 1 + 20 + (20 + 20 + 1) = 62\), so the speedup is \(S_N = 36.75N/62 = 0.59N\) and the efficiency is \(E = S/N = 0.59\).

(b)

```fortran
integer I, L
real a(1:N), b(1:N), x(1:N)
do I = 1, N, 64
  L = min(N, I+63)
  where ( b(I:L) > 0.75 )
    x(I:L) = exp( b(I:L) )
  elsewhere
    x(I:L) = sin( b(I:L) ) + exp( a(I:L) )
endwhere
end do
```

If we ignore integer min and loop overhead, \(T_{64} = \left\lceil \frac{N}{64} \right\rceil (1 + 20 + 20 + 20 + 1)\), for a speedup of

\[ S_{64} = \frac{T_1}{T_{64}} = \frac{36.75N}{62\left\lceil \frac{N}{64} \right\rceil} = \frac{0.59N}{\left\lceil \frac{N}{64} \right\rceil} \] and efficiency \(E_{64} = \frac{0.59}{64 \left\lceil \frac{N}{64} \right\rceil} \frac{N}{\left\lceil \frac{N}{64} \right\rceil}\).

If \(N \gg 64\), the speedup is about \(0.59\times64\) and the efficiency is about 0.59 because we counted no strip mining overhead. A fair accounting for loop overhead would require assigning some time to index increment, test, and branch, both for the loop over strips and for the sequential loop.

**Problem: 3.29**
Assume \(a, b, c, r,\) and \(s\) are vector registers in a pipelined vector computer. The computer has separate floating point add and multiply pipelines, each of length \(k\). Vector
registers have length \( n \). The independent pipelines can be used simultaneously by chaining, as shown below.

**Chained operations**

\[
\begin{align*}
r &:= a + b \\
s &:= c \cdot r
\end{align*}
\]

It takes \( l \) clock cycles to issue a vector instruction, where \( l < k \). Assume both pipelines are empty when the first instruction is issued. In these chained operations, what is the average number of floating point operations per clock cycle from the start of issuing the first instruction until the completion of the second? What is the efficiency of use of the hardware in the two pipelines?

**Solution: 3.29** For the chained operations, the time taken is the time to issue the first instruction plus the time for all the vector register components to pass through both pipelines. The issue of the second instruction overlaps with the start up of the first pipeline. The completion time is \( T = i + 2k - 1 + n \) cycles. In this time, \( 2n \) floating point operations are completed for an average of \( 2n(i + 2k - 1 + n) \) operations per cycle. The efficiency of hardware use is \( n(i + 2k - 1 + n) \).

**Problem: 3.30** Consider the following loop for adding two vectors:

\[
\begin{align*}
\text{DO 10 & I = 1, N} \\
C(I) &= A(I) + B(I) \\
10 &\text{ CONTINUE}
\end{align*}
\]

What would be the differences in the machine code generated by this loop for a register-to-register pipelined vector computer and for a memory-to-memory pipelined machine? Write example machine code for both cases; point out the differences; state any assumptions about the architectures needed to interpret your answer.

**Solution: 3.30**

**Load-Store:**

- load v1, a
- load v2, b
- add v3, v1, v2
- store c, v3

**Memory-Memory:**

- add c, a, b, n

For Load-Store, we assumed the simple case for register length = \( N \). For Memory-Memory, we assumed a simplified instruction form with no mask specification.

The Memory-Memory machine requires fewer instructions, but they would have a very big encoding since we need to specify 3 memory addresses and a value for vector length that could be the same size as a memory address. The Load-Store machine might need strip mining code due to limited vector register length.

**Problem: 3.31** Consider a pipelined vector processor with a statically scheduled multifunctional pipeline (all vector operations use the same pipeline). The pipeline has \( k \) stages, each having a delay of \( 1/k \) time units. The pipeline must be drained between different functions, such as addition and multiplication or additions on different pairs of vectors. Memory-access time, control-unit time, etc., can be ignored. There are sufficient numbers of temporary registers to use.

(a) Determine the number of unit time steps \( T_1 \) required to compute the product of two \( n \times n \) matrices on a non-pipeline scalar processor. Assume one unit time per each addition or each multiplication operation on the scalar processor.

(b) Determine the number of time steps \( T_k \) required to compute the matrix product, using the multifunction pipeline processor with a total pipeline delay of one time unit (step time is \( 1/k \) time units).

(c) Determine the speedup ratios \( T_1/T_k \), when \( n = 1 \), \( n = k \), and \( n = m k \) for some large \( m \).
Solution: 3.31 (a) If only floating point operations are counted, there are \( n^3 \) multiplies and \( n^3 \) adds in the scalar code, so \( T_1 = 2n^3 \).
(b) The vector pseudo code of Fig. 1-4 of the notes shows that \( A[i,k] \times B[k, \cdot] \) will be done as a vector multiply, taking \( n + k - 1 \) pipeline steps of \( 1/k \) time units each. After the pipeline empties from the multiply, the vector add \( C[i, \cdot] + \text{product}[\cdot] \) will be done in \( n + k - 1 \) steps. The time for each \([i, k]\) pair is thus \( t = 2(n + k - 1)/k \), and the total time is
\[
T_k = \frac{2n^2(n + k - 1)}{k}.
\]
(c) The speedup is
\[
\frac{T_1}{T_k} = \frac{nk}{n + k - 1}.
\]
This is 1 for \( n = 1 \) since no pipelining occurs in that case. It is \( k/(2 - 1/k) \) for \( n = k \), giving the expected efficiency of near 50%. Finally, it is \( k/(1 + \varepsilon) \), where \( \varepsilon = O(1/m) \) for \( n = mk \). Hence the speedup is always less than \( k \) but approaches \( k \) as \( n \) becomes large.

Problem: 3.32
(a) A pipelined vector computer has a floating point add pipeline of 6 stages and a stage cycle time of 20 ns. Characterize this vector pipeline in terms of its asymptotic rate \( r_\infty \) (in MFLOPS) and the vector length \( n_{1/2} \) required for 50% efficiency. Characterize only the pipeline behavior, i.e. ignore instruction issue time.
(b) Suppose the vector computer also has a floating point multiply pipeline of 9 stages, also with a 20 ns cycle time. The computer allows chaining the output of the multiply pipeline to the input of the add pipeline to do a SAXPY operation. Characterize the chained pipelines in terms of \( n_{1/2} \) and \( r_\infty \) (MFLOPS).

Solution: 3.32 (a) \( r_\infty = \frac{1}{20ns} = 50 \) MFLOPS and \( n_{1/2} = (K - 1) + \frac{T_i}{T_p} = K - 1 \), so for \( T_i = 0 \)
\[
n_{1/2} = 5.
\]
(b) \( r_\infty = \frac{2}{20ns} = 100 \) MFLOPS and \( n_{1/2} = (K_1 + K_2 - 1) + \frac{T_i}{T_p} = K_1 + K_2 - 1 \), so for \( T_i = 0, \quad n_{1/2} = 14 \).
Chapter 4: Solutions

Problem: 4.1  Rewrite the recurrence solver of Program 4-4 replacing the use of producer/consumer synchronization by critical section synchronization.

Solution: 4.1

procedure dorow(value i, done, n, a, x, c, ok)
    shared n, a[n, n], x[n], c[n], ok[n], done;
    private i, j, sum, priv, tok;
    sum := c[i];
    for j := 1 step 1 until i-1 begin
        tok := false;
        while (not tok) begin critical
            tok := ok[j];
        end critical
        sum := sum + a[i, j]x[j];
    end critical
    x[i] := sum;
    ok[i] := true;
    done := done - 1;
end critical
    return;
end procedure
Main program

\begin{verbatim}
shared n, a[n, n], x[n], c[n], ok[n], done;
private i;

done := n;
for i := 1 step 1 until n-1
{ ok[i] := false;
  create dorow(i, done, n, a, x, c, ok); }
i := n;
ok[n] := false;
call dorow(i, done, n, a, x, c, ok);
while (done $\neq 0$) {};
\end{verbatim}

Problem: 4.2 Why should the parameter $i$ in Program 4-4 not be passed by reference since it is only read and not modified by the created processes? When are call-by-value and call-by-reference interchangeable in created processes of a parallel program?

Solution: 4.2 Although the parameter $i$ is not modified in the created subroutine, it is modified in the main program that creates the subroutine. This main program runs concurrently with the created subroutine and may modify $i$ so that its value becomes different than it was at the time of the create. Passing $i$ by value prevents changes by the main program from influencing the value of $i$ in the created subroutine. If it is guaranteed that $i$ will not be changed by the created subroutine, the main program, or any other process that may share $i$ until the created subroutine quits, then call-by-reference and call-by-value will be equivalent for $i$.

Problem: 4.3 In a shared memory multiprocessor in which all memory is shared and has the same access time, why might a parallel language supply a private declaration for variables?

Solution: 4.3 Many variables have a usage pattern that ties them to a single process. Temporary variables, in particular, should be different for all processes. With a universally shared memory, processes would need to specify unique variable names for temporaries used by different processes, perhaps by appending the process ID to the variable name. A private declaration serves the same purpose by making the compiler manage the distinction between variables. This concept also extends to things like the subroutine call stack, which has a unique state for every process in an MIMD machine. Private versions of shared results even have meaning. Processes may sum partial results into a private sum variable first and then combine these private sums to get a shared version as a second step to get better performance.

Problem: 4.4 Consider a shared memory multiprocessor in which a new process is started with a create statement and terminates with a quit. The create statement takes a subroutine name and a list of call–by–reference parameters. For synchronization, this machine has only the operations lock and unlock. Show code fragments needed in creating and created routines to make the statement create $\text{sub}(x)$ act as if $x$ were a call–by–value parameter. Your code should work even if the create statement is contained in a loop.

Solution: 4.4 Assume the initial state of a lock variable, $v$, is clear. The creating program can do:

\begin{verbatim}
lock(v);
create $\text{sub}(x)$;
lock(v);
...;
\end{verbatim}
The created process obtains the value of the variable and releases the lock

```plaintext
subroutine sub(x);
  private privatex;
  privatex := x;
  unlock(v)
  ...
end
```

**Problem: 4.5** Calculate the efficiency \( E(n) \) of Program 4-4 as a function of \( n \) assuming one processor per process.

**Solution: 4.5** First let us try counting only floating point operations on \( a \) and \( x \). Process \( P_i \) executes \( 2(i-1) \) operations, one multiply and one add on each iteration of the \( j \) loop. Process \( P_n \) executes \( 2(n-1) \) operations and determines the completion time of the program. A total of \( 2n(n-1) \) operations could have been completed during this time by all processes, but only

\[
2 \sum_{i=0}^{n-1} i = n(n-1)
\]

are actually done, for an efficiency of \( \frac{n(n-1)}{2n(n-1)} = 50\% \).

An alternative analysis could count statements. Since the process executing the main program also does the most work in the procedure, it will determine the time. Counting each pass through a for loop as one statement overhead, \( P_n \) executes \( 1 + 3(n-1) + 3 \) statements before calling \texttt{dorow}. It then executes \( 1 + 3(n-1) + 5 \) statements in \texttt{dorow} for a total of \( 6n + 4 \) statements. Process \( P_i \) executes \( 1 + 3(i-1) + 5 \) statements for a total over all processes of

\[
\sum_{i=1}^{n-1} (3(i-1) + 6) + 6n + 4 = \frac{3n^2 + 15n - 1}{2}.
\]

All processes could have executed \( n(6n + 4) \) statements during the program running time, for an efficiency of

\[
E(n) = \frac{3n^2 + 15n - 1}{12n^2 + 8n}.
\]

This efficiency is about 25% for large \( n \).

The first calculation ignores process start up in the main program, so it is too optimistic. The second treats all statements the same, whereas the one involving multiply/add of an indexed array should take longer than others, so it is too pessimistic. Efficiency is probably between 25% and 50% on a real machine.
Problem: 4.6  Sequential pseudo code for Gaussian elimination without pivoting is shown below.

```latex
\text{for } k := 1 \text{ step 1 until } N
\begin{align*}
&\text{begin } p := 1/\{a[k, k]\}; \\
&\quad a[k, k] := p; \\
&\quad \text{for } i := k+1 \text{ step 1 until } N \\
&\quad \quad \text{begin } q := -a[i, k] \times p; \\
&\quad \quad \quad a[i, k] := q; \\
&\quad \quad \quad \text{for } j := k+1 \text{ step 1 until } N \\
&\quad \quad \quad \quad \text{begin } a[i, j] := a[i, j] + q \times a[k, j]; \\
&\quad \quad \text{end;}
&\text{end;}
\end{align*}
```

Write shared memory multiprocessor pseudo code for Gaussian elimination on a system with \( NP \) processors. Assume \( NP \ll N \). Use a minimum number of \texttt{fork} and \texttt{join} operations. Assume that \texttt{fork} assigns to the forked process a unique private variable, \( me \), that contains the number, starting at one, of that process. The initial process will have \( me = 0 \). For synchronization use a \texttt{barrier} operation with a sequential body having the following format:

```
\texttt{barrier}
\quad \{\text{Code body executed by one (any one) process}\}
\texttt{end barrier}
```

The behavior of the barrier is that all processes arrive at the \texttt{barrier} statement, one process executes the body of the barrier (process number is not specified), and then all processes exit the \texttt{end barrier}.

Solution: 4.6  \texttt{fork} assigns a unique identifier, \( me \), to each process. The original process has \( me = 0 \).

```
\text{shared integer } N, NP; \\
\text{shared array } a[N,N]; \\
\text{shared real } p; \\
\text{private real } q; \\
\text{private integer } me, ii, k, i, j; \\
\text{\hphantom{\text{private real } q;}} /* Assume } N\gg NP \text{ so dividing work only over rows or columns is sufficient */
\text{for } ii := 1 \text{ step 1 until } NP-1 \\
\quad \text{fork Gauss;}
\text{Gauss:

\text{for } k := 1 \text{ step 1 until } N
\begin{align*}
&\text{begin } \quad /* \text{Process } k\text{th diagonal element.} */ \\
&\quad \text{barrier} \\
&\quad \quad p := 1/\{a[k,k]\}; \quad /* \text{Compute pivot once} */ \\
&\quad \quad a[k,k] := p; \quad /* \text{for all processes} */
&\quad \text{end barrier} \\
&\quad \text{for } i := me+k+1 \text{ step } NP \text{ until } N \\
&\quad \quad \text{begin} \\
&\quad \quad \quad q := -a[i,k] \times p; \quad /* \text{Row multiplier} */ \\
&\quad \quad \quad a[i,k] := q; \\
&\quad \quad \quad \text{for } j := k+1 \text{ step 1 until } N \quad /* \text{Process row} */ \\
&\quad \quad \quad \quad \text{begin} \\
&\quad \quad \quad \quad \quad a[i,j] := a[i,j] + q \times a[k,j]; \\
&\quad \quad \quad \quad \text{end;}
&\quad \text{end;}
&\quad \text{join } NP;
```
```
Problem: 4.7 Consider both block and cyclic prescheduling of the `forall` over `i` in Program 4-6. Count the number of multiply-adds done by process `k`, `0 ≤ k ≤ P-1`, in each case. How does the load imbalance between the process with the most work and that with the least work vary with `P` and `n`? Assume that `P` divides `n` evenly.

Solution: 4.7 Suppose `P` divides `n` evenly and `m = n/P`, then for block scheduling process `P_k` does the parallel loop body for `km + 1 ≤ i ≤ (k + 1)m` for a total multiply and add count of

\[
2 \sum_{i = km + 1}^{(k+1)m} (i - 1) = 2 \sum_{i = km + 1}^{(k+1)m} (mk + j) = m^2(2k + 1) - m .
\]

The maximum number of operations, done by `P_{P-1}`, is `m^2(2P - 1) - m`, and the minimum number, done by `P_0` is `m^2 - m`, so the difference between maximum and minimum is

\[
m^2(2P - 2) = (2P - 2)\frac{n^2}{P^2} .
\]

For cyclic scheduling, process `P_k` does the loop body for values of `i = km + mj + 1`, `0 ≤ j ≤ m`, for a total number of operations \(2 \sum_{j = 0}^{m-1} (k + mj) = 2km + 2m \sum_{j = 0}^{m-1} j = 2km + m^3 - m^2\). Process `P_{P-1}` does the most operations, a total of \(2(P - 1)m + m^3 - m^2\), and process `P_0` does the fewest, a total of \(m^3 - m^2\). The difference is \(2(P - 1)m = 2(P - 1)\frac{n}{P}\).

The difference for block scheduling is about \(2n^2/P\) for large `n`, while for cyclic scheduling it is only about \(2n\) for large `n`.

Problem: 4.8 A job for a multiprocessor with `P` processors consists of `N` independent tasks of two types, long and short. Long tasks take `T_L` units of processor time, and short tasks take `T_S`. There are `N_L` long tasks and the rest are short. `P` divides `N` evenly and `m = N/P > N_L`. The `N_L` long tasks are randomly placed among the `N` tasks.

(a) If the tasks are executed by a prescheduled parallel loop, give an expression for the execution time in the worst case. Evaluate the worst case execution time for `N = 100`, `P = 10`, `N_L = 5`, `T_S = 1`ms, and `T_L = 10`ms.

(b) If the tasks are executed by a self scheduled parallel loop, find the worst case execution time and calculate it for the same parameter values.

Hint: Consider the worst case for processors being idle because there is no work left to do.

Solution: 4.8 a) In a prescheduled loop, the worst case is having one processor assigned all the long tasks. The execution time for this case is

\[
T_p = N_LT_L + \left(\frac{N}{P} - N_L\right)T_S .
\]

For the specific problem, `N=100`, `P=10`, `N_L=5`, `T_L=10`, `T_S=1`, and `T_p = 5*10 + 5*1 = 55`ms.

b) In a self-scheduled loop, each processor will be busy until there is no more work to be done. The worst case here is assigning a long task on the last iteration, keeping one processor busy while the others might have already finished. The execution time for this case is
\[ T_P = T_L + \left( \frac{(N - N_L)T_S + (N_L - 1)T_L}{P} \right) \]. Notice that any fraction in the second term is covered by the last length \( T_L \) task, hence the floor operator. Using the given parameters, we have \( T_P = 10 + \lfloor (95*1 + 4*10) / 10 \rfloor = 23 \text{ms} \).

**Problem: 4.9** Write prescheduling code to enable a process with identifier \( 0 \leq id \leq P-1 \) to generate the set of index pairs \((k, i)\) it will handle in executing a parallel two dimensional triangular loop corresponding to the sequential nested loops

```plaintext
for k := 1 step 1 until n
  for i := 1 step 1 until k
    \langle \text{parallelizable loop body} \rangle
```

Each process must compute its next index pair independently of other processes.

Hint: Linearize the index by numbering the pairs consecutively. For a given linearized index value handled by process \( id \), find \( k \) by solving a quadratic and chopping to an integer.

**Solution: 4.9** If we linearize the set of \((k, i)\) pairs for which the loop body needs to be executed, we obtain \((1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3), (4, 1), (4, 2), (4, 3), (4, 4), \ldots \). At the beginning of a group of \((k, i)\) pairs for a fixed \( k \), we know that the number of preceding pairs is \( \sum_{j=1}^{k-1} j = \frac{k(k-1)}{2} \). Thus if \( j \) is a linearized index that numbers \((k, i)\) pairs starting at \( j = 0 \), we can write \( (k^2 - k)/2 = j \). For an arbitrary value of \( j \), we can solve this quadratic equation for \( k \) and truncate the value to an integer to get the value of \( k \) at the preceding group boundary. The index \( i \) is the difference between the value of \( j \) at the beginning of the group and the actual value of \( j \). The nested sequential loops can thus be replaced by the one dimensional parallel loop

```plaintext
forall j := 0 step 1 until n*(n + 1)/2
begin
  k := \lfloor (1 + \sqrt{1 + 8j})/2 \rfloor;
  i := j + 1 - k*(k - 1)/2;
  \langle \text{loop body(k, i)} \rangle;
end;
```

Prescheduling code for process \( id \) of \( P \) processes would then be

```plaintext
forall j := me step 1 until n*(n + 1)/2
begin
  k := \lfloor (1 + \sqrt{1 + 8j})/2 \rfloor;
  i := j + 1 - k*(k - 1)/2;
  \langle \text{loop body(k, i)} \rangle;
end;
```

**Problem: 4.10** Complete the implementation of producer/consumer variables using critical sections by writing critical section based pseudo code procedures for \texttt{consume}, \texttt{copy}, and \texttt{void} to go with the \texttt{produce} implementation of Program 4-8.
Solution: 4.10

```pseudocode
procedure consume(x, px)
    shared struct (real var; boolean f) x;
    private ok;
    ok := false;
    repeat
        critical
            if (x.f) then {
                ok := true;
                x.f := false;
                px := x.var;
            }
        until ok;
    end procedure;

procedure copy(x, px)
    shared struct (real var; boolean f) x;
    private ok;
    ok := false;
    repeat
        critical
            if (x.f) then {
                ok := true;
                px := x.var;
            }
        until ok;
    end procedure;

procedure void(x)
    shared struct (real var; boolean f) x;
    critical
        x.var := false;
    end critical;
end procedure;
```

Problem: 4.11

A multiprocessor supports synchronization with lock/unlock hardware. The primitives are represented at the compiler language level by two subroutine calls, `lock(q)` and `unlock(q)`. The `lock(q)` operation waits for the lock to be clear, sets it and returns, while `unlock(q)` clears the lock unconditionally. It is desired to implement produce and consume on full/empty variables where `produce(x,v)` waits for x empty, writes it with value v and sets full while `consume(x,v)` waits for x full, copies its value to v and sets x to empty.

Using sequential pseudo code extended by the two operators `lock(q) and unlock(q)`, write code sections which implement `produce(x,v)` and `consume(x,v)` on an asynchronous variable x and normal variable v. Carefully describe your representation for an asynchronous variable. No synchronization operations other than `lock` and `unlock` may be used.

Solution: 4.11

We will represent the asynchronous variable X by a record of three items: the value of X, a boolean full flag and a unique lock for the variable X.
record X{       value : real;       full : boolean;       l : lock }  

Produce and consume can then be implemented as follows:

produce (X,V) {  
  lock (X.l)  
  if X.full then  
    unlock (X.l) ;  
    goto R  
  else  
    X.full := true ;  
    X.value := V ;  
    unlock (X.l)  
}  

consume (X,V) {  
  lock (X.l)  
  if not X.full then  
    unlock (X.l) ;  
    goto R  
  else  
    V := X.value ;  
    X.full := false ;  
    unlock (X.l)  
}  

(Alternate Solution) A simpler solution is possible by using only the lock operation to do a wait, but only if it is recognized that one lock is not enough. Two different conditions are being waited for in produce and consume, so two locks are needed.

record X{       value : real;      f : lock ;      e : lock }  

produce (X,V) {  
  lock (X.f) ;  
  lock (X.e) ;  
  X.value := V ;  
  unlock (X.e)  
}  

consume (X,V) {  
  V := X.value ;  
  unlock (X.f)  
}  

Problem: 4.12 What can happen if a processor is taken away from a process while it is in a critical section? Parallel programs may use many short critical sections to update shared variables. Must the operating system be notified on every entry to and exit from a critical section? What are the reasons for reallocating a processor in a shared memory multiprocessor? How do they differ from the reasons for reallocating a uniprocessor?

Solution: 4.12 No other processor would be able to execute another critical section, or one with the same name for a named critical section, even though the original process is making no progress. This could lead to a deadlock situation unless the operating or run-time system kept a record of all critical sections in which processes were suspended. Notifying the operating system of every entry/exit from a critical section would generate a lot of overhead. Preventing the operating system from reallocating the processor in short critical sections, say by disabling interrupts, is probably better. In a multiprocessor, many processes are running on parallel processors in contrast to the uniprocessor case in
which processor reallocation for time multiplexing is the only way to get multiple processes. Processor reallocation in a multiprocessor is usually a matter of load balancing and is not very time critical.

**Problem: 4.13** Write code to compute the lower and upper bounds, \( \text{lb} \) and \( \text{ub} \), for process \( \text{me} \) executing its portion of

\[
\forall i := \text{lwr} \text{ step } \text{stp} \text{ until } \text{upr}
\]

under block prescheduling. Balance the load so that each process handles at most one more value of the index \( i \) than any other.

**Solution: 4.13**

```
shared integer lwr, stp, upr, np;
private integer i, lb, ub, me;
private integer ni, Qip, Rip;
ni := (upr-lwr)/stp+ 1; /* number of iterations in loop. */
Qip := [ni/np]; /* quotient of iterations/processor. */
Rip := ni mod np; /* remainder of iterations/processor. */
/* The code below normalizes the lower/upper bounds for this process in terms of initial iteration number (\text{start}) and number of iterations (\text{count}). If the division \text{ni/np} is not even, the first (\text{ni mod np}) processes will execute one more iteration than the others. After computing these numbers, we calculate the actual lower/upper bounds for this process. */
private integer start, count;
if (me < Rip) then {
    count := Qip+1;
    start := me*count;
} else {
    count := Qip;
    start := me*count + Rip;
} lb := lwr + stp*start;
ub := lb + stp*(count-1);
for i := lb step stp until ub {
    loop body(i);
}
```

**Problem: 4.14** A shared memory multiprocessor has the basic synchronization instructions \textbf{lock} and \textbf{unlock}. Some users have implemented the \textbf{barrier} synchronization using the pseudo-code form:

```
procedure barrier(n);
begin
    lock(c);
    count := count + 1;
    last := (count = n);
    unlock(c);
    if last then unlock(b)
    else begin lock(b); unlock(b); end
    lock(c);
    count := count - 1;
    if count = 0 then lock(b);
    unlock(c);
end barrier;
```

The parameter \( n \) is the number of processes to cooperate in the barrier, and initial conditions are: c-unlocked, b-locked and count = 0.
The barrier behaves incorrectly when five processes execute the code:

```plaintext
phase1();
barrier(5);
phase2();
barrier(5);
phase3();
```

Tell what is wrong and show how to correct it.

**Solution: 4.14** The problem here is that the last process arriving at the barrier (and therefore the first one to unlock `b` and leave) can reenter the procedure via the next barrier statement while some processes are still passing through their first barrier. This will destroy the meaning of `count`.

Lock `c` is only used to provide atomicity to the updating and testing of variable `count`. Lock `b` is used to prevent processes from exiting the barrier until the last one has arrived. Symmetrically, we can create another lock, `a`, to prevent processes from entering the barrier until the last one has left. Lock `a` should be initially unlocked.

```plaintext
procedure barrier(n)
begin
  lock(a); unlock(a);
  lock(c);
  count := count + 1;
  lastin := (count = n);
  unlock(c);
  if lastin then begin
    lock(a);
    unlock(b);
  end else begin
    lock(b);
    unlock(b);
  end;
  lock(c);
  count := count - 1;
  lastout := (count = 0);
  unlock(c);
  if lastout then begin
    lock (b);
    unlock (a);
  end;
end barrier;
```

**Problem: 4.15** The procedure below is called by all $2^k$ processors of a shared memory multiprocessor. The unique identifier of each processor is $0 \leq me \leq 2^k - 1$, $V$ is a real valued variable, and $k$ is an integer
constant parameter. The function xor() returns the bit wise exclusive OR of its two integer arguments.

```plaintext
procedure lep(me, V)
begin
  private me, V;
  shared A[0:k-1, 0:2^k-1];
  private i, j, T;
  for i:=0 step 1 until k-1
  begin
    j := xor(me, 2^i);
    produce A[i, j] := V;
    consume A[i, me] into T;
    V := max(V, T);
  end
end
```

(a) What is the value of V returned in processor me?
(b) Name the synchronization performed when all processors call the procedure.

Solution: 4.15 The pattern of communication between processes using the full/empty variables is shown below. Dashed lines show the synchronizations and value communications while solid lines are the flow of process execution.

(a) The returned values of V are the maximum over all incoming values of V.
(b) No process can enter the procedure until all have entered it. This is the behavior of a barrier synchronization. This synchronization pattern is called a butterfly barrier because of the butterfly wing shape of pairwise synchronizations.

Problem: 4.16 Write efficient multiprocessor pseudo code for the computation:

```plaintext
for i := 1 step 1 until N
  for j := 1 step 1 until N
    a[j] := a[j] \times (b[i, j]/c[i]);
```

Assume a forall construct (like the OpenMP PARALLEL DO), and assume that all processors are running and executing the same code. There are order N, and not order N^2, processors.

Solution: 4.16 Here we have a perfect nested loop with all the parallelism contained in the inner loop. To exploit this parallelism with a MIMD machine, we exchange the inner and outer loops, and apply a
forall construct to what is now the outer loop:

```plaintext
shared a[N], b[N,N], c[N];
private i, j;
forall j := 1 step 1 until N
  for i := 1 step 1 until N
    a[j] := a[j] * (b[i,j] / c[i]);
```

Using a forall on the inner loop would invoke the work distribution overhead of the forall $N$ times and could have a severe performance penalty.

**Problem: 4.17** The following sequential pseudo code computes the histogram of a data array. $x_0$ and $x_1$ are the minimum and maximum values, respectively, in the data array $x[1:N]$, and the histogram vector is $h[0:M-1]$.

```plaintext
scale := M/(x_1 - x_0);
for i := 1 step 1 until N
  begin
    j := int(scale*(x[i] - x_0));
    h[j] := h[j] + 1;
  end;
```

(a) Write SIMD (vector) pseudo code for the histogram calculation. On the basis of the code, discuss what part of the calculation is vectorizable and not vectorizable, and why.

(b) Write MIMD pseudo code for the histogram calculation, extending sequential pseudo code with fork/join, shared and private variables, and using only un-named critical sections for synchronization. Discuss the amount of parallelism achieved in your program.

(c) Would other hardware level synchronizations improve the parallelism in the MIMD calculation? Which ones and how?

**Solution: 4.17 (a)**

```plaintext
scale := M/(x_1 - x_0);
for i := 1 step 1 until N
  begin
    j[i] := int(scale*(x[i] - x_0)), (1 ≤ i ≤ N);
    h[j[i]] := h[j[i]] + 1;
  end;
```

The computation of $j[i]$ is vectorizable because it is independent for different values of $i$. The increment of the corresponding element $h[j]$ is not vectorizable, because $j[i]$ can be the same for different $i$.

(b) `shared` scale, $M$, $x_1$, $x_0$, $x[N]$, $h[M]$;

```plaintext
private i, j;
scale := M/(x_1 - x_0);
for i := 1 step 1 until N-1 fork DOELEMENT;
i := N;
DOELEMENT:
  j := int(scale*(x[i] - x_0));
critical
  h[j] := h[j] + 1;
end critical
join(N);
```

The parallelism here is limited to the computation of $j$. The increment of $h[j]$ has to be done in a sequential fashion because we need to synchronize the updates of a shared variable.

(c) If we had named critical sections for each $h[j]$, or if we could lock the element $h[j]$, or if we
could use produce/consume on the elements $h[j]$, then we could avoid stalling the updates of $h[j]$ for different $j$, therefore achieving better parallelism.

**Problem: 4.18** Write two OpenMP subroutines that multiply $N \times N$ matrices. The subroutine parameters should be two operand matrices, the result matrix, and the dimension $N$. Do not use nested parallelism, but specify a fixed number of processes for the whole computation.

(a) Write the most straightforward OpenMP implementation in which the amount of work in each parallel task is $O(N^2)$ with $O(N)$ processes.

(b) Write a finer granularity program where each parallel task has only $O(N)$ operations with $O(N^2)$ processes. Suggest a new OpenMP construct that would better support this implementation.

**Solution: 4.18 (a)**

```fortran
subroutine matmul(N, a, b, c)
    real a(N, N), b(N, N), c(N, N)
    integer N, i, j, k
    !$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i, j, k)
    do i = 1, N
        do j = 1, N
            c(i, j) = 0.0
            do k = 1, N
                c(i, j) = c(i, j) + a(i, k)*b(k, j)
            end do
        end do
    end do
    !$OMP END PARALLEL DO
    return
end
```

(b)

```fortran
subroutine matmul(N, a, b, c)
    real a(N, N), b(N, N), c(N, N)
    integer n, i, j, k, m
    !$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i, j, k, m)
    do m = 1, N*N
        i = (m - 1)/N + 1
        j = modulo(m-1, N) + 1
        c(i, j) = 0.0
        do k = 1, N
            c(i, j) = c(i, j) + a(i, k)*b(k, j)
        end do
    end do
    !$OMP END PARALLEL DO
    return
end
```

The OpenMP compiler could linearize and parallelize perfectly nested loops with some restrictions, say fixed upper limits for rectangular loop nests. Under the OpenMP principle that an OpenMP direc-
tive applies to the following Fortran statement, a perfect nest of two parallel loops might be done by

```fortran
!$OMP DO
  do i = 1, n
  !$OMP DO NEST
    do j = 1, n
  end do
end do
```

The OpenMP compiler could generate code like that above to parallelize the nested loops.

**Problem: 4.19** A program with too many barriers can be safe because it is guaranteed to execute correctly in parallel, but unnecessary barriers have some performance penalty, if only because processes wait for the slowest at each one. For each OpenMP directive in Program 4-13 that allows a NOWAIT clause, tell whether such a clause is correct if present or may be inserted if absent to improve performance without damaging the program logic.

**Solution: 4.19**

```fortran
!$OMP END DO
  Should use NOWAIT because there will be another barrier before any \( a(i, j) \) is used.
  !$OMP END SINGLE
  Should use NOWAIT because there is a barrier at the END SINGLE at the beginning of the do 2000 loop.
  !$OMP END SINGLE
  Must leave off NOWAIT to ensure \( a(i, j) \) is completely generated, or updated before using it again.
  !$OMP END NOWAIT
  NOWAIT is correct here since a thread should proceed to update the shared maximum as soon as its private maximum calculation is complete.
  The explicit !$OMP BARRIER before the SINGLE section ensures correctness of gmax.
  !$OMP END SINGLE
  This must not have a NOWAIT because no thread may test \( msing \) until it is set inside the SINGLE construct.
  !$OMP END DO
  There should not be a NOWAIT because the rows must be completely swapped before they are used.
  !$OMP END SINGLE
  Must leave off NOWAIT so no thread can use \( pivot \) until it has been computed.
  !$OMP END DO NOWAIT
  NOWAIT is correct here because a barrier will occur at the end of the SINGLE section beginning the do 2000 loop.
```

**Problem: 4.20** If the particle dynamics simulation is run on a system with more than \( n \) processors, it would be beneficial to combine the \( i \) and \( j \) loops in Program 4-11 to obtain \( O(n^2) \) parallel work. Show how to do this by restructuring the code and introducing the appropriate OpenMP directives. It will be necessary to get perfectly nested \( i \) and \( j \) loops in order to combine them.
Solution: 4.20

```plaintext
pot = 0.0
kin = 0.0

!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i, k, m)
  do m = 1, 3*n
    i = (m - 1)/3 + 1
    j = modulo(m - 1, 3) + 1
    f(k, i) = 0.0
  end do

!$OMP END PARALLEL DO

!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i, j, k, m, rij, d) &
!$OMP REDUCTION(+: pot)
  do m = 1, n*n
    i = (m - 1)/n + 1
    j = modulo(m - 1, n) + 1
    if (i .ne. j) then
      call dist(p(1, i), p(1, j), rij, d)
      pot = pot + 0.5*v(d)
      do k = 1, 3
        f(k, i) = f(k, i) - rij(k)*dv(d)/d
      end do
    end if
  end do

!$OMP END PARALLEL DO

!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i) REDUCTION(+: kin)
  do i = 1, n
    kin = kin + dot(v(1, i), v(1, i))
  end do

!$OMP END PARALLEL DO

kin = kin*0.5*mass
```

Problem: 4.21 This problem considers two parallel organizations of code having the same serial time.

(a) The following OpenMP code is executed by $P$ threads on $P$ processors, where $N >> P$.
Write an expression for the execution time of the program. What is the speedup over a single processor execution if $N = 1000$, $P = 50$, $t_0 = 1$ sec., $t_1 = 4$ sec., and $t_2 = 495$ sec.? Assume the single processor must execute the critical section body code, and assume that the time for all the OpenMP statements is negligible compared to the (code) sec-
60 Fundamentals of Parallel Processing · Chapter 4: Solutions

!$OMP BARRIER
!$OMP SINGLE
  (code: time $t_0$)
!$OMP END SINGLE
!$OMP CRITICAL
  (code: time $t_1$)
!$OMP END CRITICAL
!$OMP DO SCHEDULE(STATIC, 1)
  DO 10    I = 1, N
  (code: time $t_2/N$)
  10 CONTINUE
!$OMP END DO

(b) Under the same assumptions as in (a), give an execution time expression for the code below, and estimate speedup over one processor using the numbers of part (a).

!$OMP BARRIER
!$OMP SINGLE
  (code: time $t_0$)
!$OMP END SINGLE
!$OMP DO SCHEDULE(STATIC, 1)
  DO 10    I = 1, N
  (code: time $t_2/N$)
!$OMP CRITICAL
  (code: time $t_1/N$)
  10 CONTINUE
!$OMP END CRITICAL

Solution: 4.21

(a) The time with $P$ processors is $T(P) = t_0 + P t_1 + \left\lceil \frac{N}{P} \cdot \frac{t_2}{N} \right\rceil$.

If $N >> P$ this is nearly $T(P) = t_0 + P t_1 + \frac{t_2}{P}$.

If the critical section must be executed, the time with one processor is $T(1) = t_0 + t_1 + t_2$.

If $N = 1,000$, $P = 50$, $t_0 = 1$ sec., $t_1 = 4$ sec., and $t_2 = 495$ sec., the speedup is $S = \frac{1 + 4 + 495}{1 + 4 \times 50 + \frac{495}{50}} = \frac{500}{210.9} = 2.37$.

(b) The time with one processor is $T(1) = t_0 + N \left( \frac{t_2}{N} + \frac{t_1}{N} \right) = t_0 + t_1 + t_2$, as it was in part (a). The time with $P$ processors depends on whether critical section wait or computation dominates the loop body. In general, $T(P) = t_0 + \left\lceil \frac{N}{P} \right\rceil \left( \max\left( \frac{t_2}{N}, (P-1) \frac{t_1}{N} \right) + \frac{t_1}{N} \right)$. With the numbers of part (a),

$$\frac{t_2}{N} = \frac{495\text{sec}}{1000} = 0.495\text{sec} \quad \text{and} \quad (P-1) \frac{t_1}{N} = 49 \times \frac{4\text{sec}}{1000} = 0.196\text{sec}$$

so computation time dominates the critical section. The speedup is $S = \frac{500}{1 + 20(0.004 + 0.495)} = 45.5$. 
In this problem the number of sequential loop iterations is only \( N/P = 20 \), so the first time through the loop should be taken into account for better accuracy. Processes wait for each other at the critical section the first time through the loop, and the longest wait is for \( P-1 \) others. Thus a more accurate formula for the time with \( P \) processors is \( T(P) = t_0 + \left( \frac{N}{P} \left( \frac{t_2}{N} + \frac{t_1}{N} \right) + (P - 1) \frac{t_1}{N} \right) \). This gives a revised speedup of \( S = \frac{500}{1 + 20(0.495 + 0.004) + \frac{49 \times 4}{1000}} = 44.7 \).

Discussion: The single processor execution time is the same in both parts (a) and (b), but part (a) is structured so that the critical section overhead increases with the number of processes and there is no opportunity to overlap critical section wait with useful computation. In part (b) the aggregate time each process spends in the critical section is only \( t_1/P \) instead of \( t_1 \), and the wait for other processes in the critical section is masked by computation for all but the first pass through the loop.

Problem: 4.22 Write an OpenMP program which eliminates the repeated barrier synchronizations in Program 4-13 for Gaussian elimination in the notes by implementing the algorithm described below.

Consider LU decomposition by Gaussian elimination organized as operations on columns. Column \( j \) is reduced \( j \) times, with the last reduction making it into a pivot column. A column \( j \) to the right of the pivot column \( k < j \) is reduced the \( k \)-th time by exchanging two elements according to the number of the \( k \)-th pivot row and adding a multiple of the pivot column.

\[
\begin{array}{cccccccc}
p_1 & u_{12} & u_{13} & u_{14} & u_{15} & u_{16} & u_{17} & u_{18} \\
p_2 & u_{23} & u_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\
p_3 & u_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\
p_4 & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\
p_5 & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\
p_6 & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\
p_7 & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\
p_8 & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \\
\end{array}
\]

Column 1 2 3 4 5 6 7 8
Pivot row 3 7 5 - - - -
Reduction 1 2 3 3 3 3 3

To make a column into a pivot column, a search for the maximum absolute value in the column locates the pivot row. The element in the pivot row is then swapped with the element in the pivot position, and the column is scaled to obtain the corresponding values of the L matrix.

The L and U values are computed once, as are the pivots, but there are several versions of most \( a_{ij} \).

Let the original matrix \( A \) consist of elements \( a_{ij}^{(l)} \), \( i = 1, 2, \ldots, n; j = 1, 2, \ldots, n \), where the superscript keeps track of the number of updates. Then the updates can be described by the formulas,

\[
p_{k} = a_{kk}^{(k)}, \quad k = 1, 2, \ldots, n;
\]

\[
u_{kj} = a_{kj}^{(k)}, \quad k = 1, 2, \ldots, n; \quad j = k+1, k+2, \ldots, n;
\]
Consider the problem of synchronizing updates on columns in this formulation of Gaussian elimination. The example Gaussian elimination program of Program 4-13 used a barrier to ensure that the \( k-1 \)st update was complete before the \( k \)-th update started. This is not necessary. Column \( j \) can be made a pivot column as soon as it has been updated \( j-1 \) times. The \( j \)-th nonpivot column can be updated the \( k \)-th time as soon as it has been updated \( (k-1) \) times and the \( k \)-th pivot column has been produced. If column update tasks are represented by a pair (update level, column) and scheduled in the order,

\((1,1), (1,2), \ldots, (1,n), (2,2), (2,3), \ldots, (2,n), \ldots, (n-1,n-1), (n-1,n), (n,n),\)

then a task will be assured that the tasks on which it depends have been previously scheduled.

This program is meant to give you experience in the SPMD style of programming. You may find it useful to think in terms of some of the pseudo code synchronizations we have discussed that are not available in OpenMP. In that case, design the program with those constructs, and then implement them with constructs that are available in OpenMP.

**Solution: 4.22**

Processes do tasks to update a column at one update level. Column and level are identified by the private variables (lev, col). A task handles a pivot column by searching for the maximum absolute value, swapping that element with the diagonal, and scaling the column to values of the \( L \) matrix. The other type of task reduces a column by doing a swap corresponding to the pivot for level lev and subtracting the correct multiple of the pivot column. Data structures used are:

<table>
<thead>
<tr>
<th>Column</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pivot row</td>
<td>pivot()</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Shared</td>
</tr>
<tr>
<td>Reduction</td>
<td>update()</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

Because the triangular set of tasks must be assigned to processes in a particular order, the self-scheduled DO construct is not useful. Processes will explicitly loop and obtain the next task by calling a subroutine which generates the next (lev, col) pair on the basis of a pair of variables (glev, gcol) that can be thought of as asynchronous, though OpenMP doesn’t directly support produce/consume access to variables.

The array update() is used to synchronize access to columns. The array ulock() of locks allows update() to be treated as if it were asynchronous. A process will consume the update level for its column when it is equal to lev-1, wait for the pivot column for lev to have been updated lev times (unless this update is on a pivot column), and produce update(col) equal to lev after the reduction. The variable update(col) will be empty while column col is being processed. The variable update(lev) indicating the status of the pivot column is read using copy since many tasks need to use it to determine that the same pivot column is ready. The array pivot() need not be asynchronous since it is only written when the corresponding pivot column is processed and only read after
update(lev) reports that that processing is complete.

c Subroutine to be used by process to get next task to update column

c col at level lev. Done is set non-zero if all tasks are exhausted.

subroutine gettsk(lev, col, done)
  integer lev, col, done

common /sched/ integer glev, gcol

!$OMP CRITICAL(sch)
  col = gcol
  lev = glev
  col = col + 1
  if (col .gt. n) then
    if (lev .eq. n) then
      done = 1
      col = n
    else
      lev = lev + 1
      col = lev
    endif
  endif
  glev = lev
  gcol = col
!$OMP END CRITICAL(sch)
return
end

program gauselim
  integer n,prflg,itim1,itim2,msing,pivot(500)
  real a(500,500)
  integer update(500), ulock(500)
  common /sched/ integer  glev, gcol
  integer  i, lev, col, itemp, lstup, done, mtmp
  real temp

  c
  c Read in matrix dimension and output flag
  c
  5 write(*,*) ' Input the Follwing: '
  write(*,*) ' n = matrix dimension <= 500'
  write(*,*) ' prflg=1->print results;0->don"t'
  read(*,10) n,prflg
10 format(2i4)
  if (n .gt. 500) goto 5
  call OMP_SET_DYNAMIC(.false.)
  nprocs = OMP_GET_NUM_PROCS()
  call OMP_SET_NUM_THREADS(nprocs)
!$OMP PARALLEL DEFAULT(SHARED)&
!$OMP& PRIVATE(i,lev,col,itemp,lstup,done,temp,mtmp)
Generate the matrix

```fortran
!$OMP DO SCHEDULE(STATIC)
DO 100 i = 1, n
   DO 110 j = 1, n
      if (i .eq. j) then
         a(i,j) = i*i/2.
      else
         a(i,j) = (i+j)/2.
      endif
110 continue
100 continue
!$OMP END DO NOWAIT
```

Initialize the shared array of column updates.

```fortran
!$OMP DO SCHEDULE(STATIC)
DO 105 i = 1, n
   call OMP_INIT_LOCK(ULOCK(I))
   update(i) = 0
105 continue
!$OMP END DO
```

Initialize shared scalars and wait for initialization complete.

```fortran
!$OMP SINGLE
itim1 = timer()
msing = 0
!$OMP END SINGLE
```

Initialize private done flag.

```fortran
done = 0
```

Outer loop where processes self-schedule over column update tasks.

```fortran
Get a task to update column col at level lev.
210 call gettsk(lev, col, done)
```

Exit if no more tasks.

```fortran
   if (done .eq. 1) goto 1000
```

Check to see if this is a pivot task.

```fortran
   if (lev .eq. col) then
```

Wait for penultimate update to col before processing pivot.

```fortran
   call OMP_SET_LOCK(ulock(col))
   itemp = update(col)
   if (itemp .ne. lev-1) then
      call OMP_UNSET_LOCK(ulock(col))
      goto 220
   endif
   ``
c Find maximum absolute value below the diagonal.
    temp = 0.0
    itemp = 0
    do 240 i = lev, n
           if (temp .lt. abs( a(i, lev) ) ) then
               temp = abs( a(i, lev) )
               itemp = i
           endif
    240 continue

    c Record singularity or record position and swap pivot element.
    if (temp .eq. 0.0) then
       !$OMP CRITICAL(slock)
       msing = 1
       !$OMP END CRITICAL(slock)
       goto 1000
    else
       pivot(lev) = itemp
       temp = a(lev, lev)
       a(lev, lev) = a(itemp, lev)
       a(itemp, lev) = temp
    endif

    c Make rest of column into elements of L.
    do 250 i = lev+1, n
           a(i, lev) = a(i, lev)/a(lev, lev)
    250 continue

c Report completion of pivot column processing.
    update(col) = lev
    call OMP_UNSET_LOCK(ulock(col))

else

c Process a non-pivot update to column col at level lev.
260 call OMP_SET_LOCK(ulock(col))
    lstup = update(col)

c Wait until col has been updated lev-1 times.
    if (lstup .ne. lev-1) then
       call OMP_UNSET_LOCK(ulock(col))
       goto 260
    endif

c Wait for pivot column to be available at level lev.
270 call OMP_SET_LOCK(ulock(lev))
    itemp = update(lev)
    call OMP_UNSET_LOCK(ulock(lev))
    if (itemp .ne. lev) goto 270
c Do pivot swap for this update.
        temp = a(lev, col)
        a(lev, col) = a(pivot(lev), col)
        a(pivot(lev), col) = temp

c Reduce elements below the diagonal.
        do 280 i = lev+1, n
            a(i, col) = a(i, col) - a(i, lev)*a(lev, col)
        280 continue

c Report completion of update of column col at level lev.
        update(col) = lev
        call OMP_unset_lock(ulock(col))
        endif

c Either a pivot or non-pivot column update is now complete.
c Go get next task unless a singularity was reported.
!$OMP CRITICAL(slock)
        mtmp = msing
!$OMP END CRITICAL(slock)
        if (mtmp .eq. 0) goto 210
1000 continue
!$OMP END PARALLEL

c Output section
        itim2 = timer()
        if (msing .eq. 1) then
            write(*,*) ' The matrix is singular'
        else
            if (prflg .eq. 1) then
                write(*,*) ' The eliminated matrix is as below:
                do 500 i = 1,n
                    write(*,20) i
                    do 510 j = 1,n,10
                        write(*,30) (a(i,k),k=j,j+9)
                        510 continue
                    500 continue
                20 format(//,2x,'Row :',i4)
                30 format(2x,10f6.2)
            endif
        endif
        getim = float(itim2 -itim1)/10**6
        write(*,*) ' Time for performing the elimination with
        write(*,*) nprocs,' processes is ', getim, 'secs'
        endif
end

Problem: 4.23 A pipelined MIMD or multithreaded computer inserts an instruction from a queued instruction stream into a 10 stage execution pipeline every cycle, as long as the queue is not empty. Memory reference instructions do not occupy the queue or execution pipeline while waiting for pipelined memory response. If the average memory request time is 50 cycles, and there is a 20% chance that a given instruction is a memory reference, how many instruction streams, N, must be active to ensure an instruction is issued every cycle?
Solution: 4.23 The problem can be rephrased as there being a 20% chance an instruction is issued into a 50 cycle memory pipeline and a 80% chance that it is issued into a 10 cycle execution pipeline. The average number of cycles to complete an instruction is then given by 
\[ 0.2 \cdot 50\text{cycles} + 0.8 \cdot 10\text{cycles} = 18\text{cycles} \]. Thus, an average of \( N = 18 \) processes must be active to ensure that a process will be in the queue for every issue. Statistical variation will require \( N > 18 \) processes to prevent the queue ever being instantaneously empty and delaying instruction issue.
Chapter 5 - Solutions

Problem: 5.1 In the program below, written in Hoare’s Communicating Sequential Processes language, CSP, there are three processes, \( P1 \), \( P2 \) and \( P3 \). Upper case letters represent statements, and lower case letters are variable names.

\[
\begin{align*}
P1 &:: A ; P2 ! u ; B ; P3 ! v ; C ||
P2 &:: D ; P3 ? w ; E ; P1 ? x ; F ||
P3 &:: G ; P2 ! y ; H ; P1 ? z ; K
\end{align*}
\]

(a) The sequential processes of CSP, along with the send (\(!\)) and receive (\(?\)) commands, impose a partial order on the statements. Diagram the covering relation for the partial order on \( A, B, C, D, E, F, G, H, \) and \( K \). The covering relation is the order with all relations implied by transitivity removed. No arrows which are implied by transitivity are to be included in the diagram.

(b) What happens if \( P2!y \) is exchanged with \( P1?z \) in process \( P3 \)?

Solution: 5.1 a) The orderings \( A \to B \to C, D \to E \to F \) and \( G \to H \to K \) are imposed by the sequentiality of the three processes. Each communication specifies two precedence relationships since both send and receive occur together. Thus the transmissions specify:

- \( P1 \to P2 \quad A \to F, E \to B \)
- \( P1 \to P3 \quad B \to K, H \to C \)
- \( P3 \to P2 \quad G \to E, D \to H \)

The covering relation is thus
b) If the send and receive in process P3 are exchanged, the chain of partial ordering relations: E → B, B → H, H → E yields a cycle which implies deadlock. If sends were buffered rather than having to wait on the corresponding receive, this would not happen.

**Problem: 5.2** Eight parallel CSP processes X(i:0..7) all compute partial results. It is required that X(0) obtain the sum of all the partial results. In the questions below, which require you to write CSP, do not worry too much about correct syntax, but be sure to use the correct semantics for parallelism, indeterminacy and inter process communication. Explain your notation in English if you are in doubt about how to write a CSP construct.

(a) Show CSP program fragments for X(0) and X(i:1..7) to do the summation in a “linear time” of eight steps.

(b) Show CSP code to do the summation in a “logarithmic time” of three steps.

**Solution: 5.2** (a) In the "linear time" case, each of the processes, X(i:1..7) can send its partial result to X(0), which receives them all and adds them up.

```csp
X(0):: ... sum := part; i := 1;
  *[ (i < 8) → (X(i)?t; sum := sum + t; i := i + 1)]; ... 

X(i:1..7):: ... X(0)!part ...
```

Since there are no shared variables, part is different in each process.

(b) In the "logarithmic time" version, each odd numbered process sends its partial result to the next lower even numbered process, processes 2 and 6 then send sums of two results to processes 0 and 4, and finally, process 4 sends a sum of 4 results to X(0).

```csp
X(i:1,3,5,7):: ... X(i-1)!part; ...
X(i:2,6):: ... X(i+1)?t; t := t + part;
  X(i-2)!t; ...
X(4):: ... X(4+1)?t; t := t + part;
  X(4+2)?t1; t := t + t1;
  X(4-4)!t; ...
X(0):: ... X(0+1)?t; t := t + part;
  X(0+2)?t1; t := t + t1;
  X(0+4)!t1; t := t + t1; ...
```

**Problem: 5.3** In a multiprocessor, a number 'n' of worker processes W(i) execute the same code. The programmer wants to specify 'k' different code sections which are each to be executed by one and only one process, but by whichever is available “first.” He conceptualizes this by adding a new statement to Hoare's CSP and writes pseudo code for the workers as:

```csp
W(i:1..'n')::: ... doeach case <sect.1> case <sect.2> case ... endeach ...
```

where 'n' and 'k' are fixed integers for any specific execution. He then proceeds to implement this construct using the primitives available in CSP.

(a) Show how to do this if 'k' ≤ 'n'. Be sure to use the correct semantics for parallelism, indeterminacy and inter process communication in CSP. Use sequential pseudo code style for sequential aspects of CSP programming of which you may be uncertain.

(b) Extend the solution to the case 'n' < 'k'. As in (a) work should be started as soon as there is a process available to do it, and the entire doeach should be exited by each process as soon as there is no more work it can do.

**Solution: 5.3**

**Problem: 5.4** Three processors in a distributed memory multiprocessor communicate by send and receive
running the code sketched below, where upper case letters represent local activities.

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>G</td>
</tr>
<tr>
<td>receive(P3)</td>
<td>send(P1)</td>
<td>receive(P2)</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>H</td>
</tr>
<tr>
<td>receive(P2)</td>
<td>send(P3)</td>
<td>send(P1)</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>I</td>
</tr>
</tbody>
</table>

(a) If `send` is non-blocking and `receive` is blocking, draw a diagram of the precedence relation on the local activities.

(b) What is the longest chain of activities which must be executed in sequence?

(c) If both `send` and `receive` were blocking, what would happen to the program and how would it relate to the precedence diagram?

**Solution: 5.4**

(a) The precedence relation diagram is:

(b) The longest path is: D, E, H, B, C

(c) There is now a circular precedence relation between B, E, H, which would cause a deadlock in the program. The new precedence relation diagram is:
Problem: 5.5 The three processes below run on three processors of a distributed memory multiprocessor in which all data movement between processors is by means of non-blocking send and blocking receive. Show the partial order imposed on the statements X1-X5, Y1-Y5, and Z1-Z5 by the three process program below. Assume that message transmission takes two time units and that statements take one time unit each except for send and receive, which take no time except for transmission. How many time units are required for program completion?

<table>
<thead>
<tr>
<th>Process X</th>
<th>Process Y</th>
<th>Process Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>Y1</td>
<td>Z1</td>
</tr>
<tr>
<td>receive i from Y</td>
<td>send p to Z</td>
<td>send a to X</td>
</tr>
<tr>
<td>X2</td>
<td>Y2</td>
<td>Z2</td>
</tr>
<tr>
<td>receive j from Z</td>
<td>receive q from Z</td>
<td>receive b from Y</td>
</tr>
<tr>
<td>X3</td>
<td>Y3</td>
<td>Z3</td>
</tr>
<tr>
<td>send k to Y</td>
<td>send r to X</td>
<td>send c to Y</td>
</tr>
<tr>
<td>X4</td>
<td>Y4</td>
<td>Z4</td>
</tr>
<tr>
<td>send n to Z</td>
<td>receive s from X</td>
<td>receive d from X</td>
</tr>
<tr>
<td>X5</td>
<td>Y5</td>
<td>Z5</td>
</tr>
</tbody>
</table>

Solution: 5.5 Covering relation for the partial order. Note that Z1 → X3 is implied by transitivity:

Longest path:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y1</td>
<td>1</td>
</tr>
<tr>
<td>Send p from Y to Z</td>
<td>2</td>
</tr>
<tr>
<td>Z3</td>
<td>1</td>
</tr>
<tr>
<td>Send c from Z to Y</td>
<td>2</td>
</tr>
<tr>
<td>Y3</td>
<td>1</td>
</tr>
<tr>
<td>Send r from Y to X</td>
<td>2</td>
</tr>
<tr>
<td>X2</td>
<td>1</td>
</tr>
<tr>
<td>X3</td>
<td>1</td>
</tr>
<tr>
<td>X4</td>
<td>1</td>
</tr>
<tr>
<td>Send n from X to Z</td>
<td>2</td>
</tr>
<tr>
<td>Z5</td>
<td>1</td>
</tr>
</tbody>
</table>

Problem: 5.6 \( N = 2^d \) processors are connected in a hypercube topology. Processor \( p \) can only transmit to, or receive from, processors \( p \oplus 2^k, \ k = 0,1,.., \ d-1 \).

Give an algorithm in SPMD pseudo code for all processes of a message processing multiprocessor that sends a message from source processor \( S \) to destination processor \( D \).

Solution: 5.6 If we consider the processor numbers as binary numbers, in the hypercube topology there can be only one bit changed between a sender and a receiver numbers. This algorithm calculates the changed bits between \( S \) and \( D \), then passes the message through a series of hops, where the first one is \( S \) and each other has one more bit changed in the direction of \( D \).

Assume we have a \( \text{xor} \) function which returns the bitwise exclusive or between two operands. The following code is for processor \( me \):
The discussion of Program 5-4 claims that single real numbers can be replaced by $b \times b$ matrix blocks, multiply replaced by matrix multiply, and add replaced by matrix add to obtain a correct matrix multiply program in which each process computes a $b \times b$ block of matrix $C$ instead of a single element.

(a) Show mathematically that this procedure gives the correct result for the matrix $C = A \times B$.

(b) Elaborate Program 5-4 for $b \times b$ blocks of $A$, $B$, and $C$ in each process. Assume blocking operations `send(buf, d, P)` and `recv(buf, d, S)` that send a buffer, $buf$, of length $d$ to process $P$ and receive the buffer from process $S$, respectively. Handle the details of packing and unpacking the buffer and multiplying the blocks of the matrices.

Solution: 5.7 For simplicity, adopt a Fortran 90 like notation where : separates lower and upper bounds of an index range. Then the private $b \times b$ block of $A$ in processor $P(i, j)$ is

$$\text{myA}[0:b-1, 0:b-1] = A[i*b:i*b+b-1, j*b:j*b+b-1]$$

and similarly for $\text{myB}$ and $\text{myC}$. The comments on the send and receive statements become

/* $P(i, k)$ sends $A[i*b:i*b+b-1, k*b:k*b+b-1]$ */
/* to $P(i, m)$ for all $i$. */
/* $P(k, j)$ sends $A[k*b:k*b+b-1, j*b:j*b+b-1]$ */
/* to $P(m, j)$ for all $j$. */

We replace the product of the real numbers $\text{tmpA}$ and $\text{tmpB}$ in Program 5-4 with the matrix product of the $b \times b$ matrices $\text{tmpA}$ and $\text{tmpB}$ and the add by a componentwise matrix add. Using a Fortran 90 like notation:

$$\text{myC}[0:b-1, 0:b-1] := \text{myC}[0:b-1, 0:b-1] + \text{matmul}(\text{tmpA}, \text{tmpB});$$

To convince ourselves that this gives the correct result, we write the operations as if they were done on the full $N_b \times N_b$ arrays, $A$, $B$, and $C$. For a given $k$ in the for loop, an element $C[ii, jj]$ in the block of $C$ owned by process $P(i, j)$ is updated by

$$C[ii, jj] = C[ii, jj] + \sum_{kk=0}^{b-1} A[ii, k \cdot b + kk] \times B[k \cdot b + kk, jj].$$

Thus for $k = 0$

$$C[ii, jj] = \sum_{kk=0}^{b-1} A[ii, kk] \times B[kk, jj],$$

and for $k = 1$
It is easy to see that when $k = N - 1$, the full matrix product will be correct for elements in that block. Using array slice assignments as in Fortran 90, we can formulate the code below for processor $P(i, j)$.

```fortran
real array myC[0:b-1, 0:b-1], myA[0:b-1, 0:b-1], myC[0:b-1, 0:b-1]
real array tmpA[0:b-1, 0:b-1], tmpB[0:b-1, 0:b-1];
integer i, j, k, m, n;

myC[0:b-1, 0:b-1] := 0;
for k := 0 step 1 until N-1
begin
  for m := 0 step 1 until N-1
  begin
    if (k ≠ m) then
      begin
        if (j = k) then P(i, m)!myA;
        if (j = m) then P(i, k)?tmpA;
        if (i = k) then P(m, j)!myB;
        if (i = m) then P(k, j)?tmpB;
      end;

    if (j = k) then tmpA[0:b-1, 0:b-1] := myA[0:b-1, 0:b-1];
    if (i = k) then tmpB[0:b-1, 0:b-1] := myB[0:b-1, 0:b-1];
  
  myC[0:b-1, 0:b-1] := myC[0:b-1, 0:b-1] + matmul(tmpA, tmpB);
end;

procedure matmul(P, Q)
real array P[0:b-1, 0:b-1], Q[0:b-1, 0:b-1], R[0:b-1, 0:b-1];
integer ii, jj, kk;
for ii := 0 step 1 until b-1
for jj := 0 step 1 until b-1
begin
  R[ii, jj] := 0;
  for kk := 0 step 1 until b-1
  begin
  end;
end procedure;
```

**Problem: 5.8** Consider the Gauss elimination solution of $Ax = b$ where rows of $A$ are distributed over $p$ processors of a machine that does not favor operations on contiguous vectors. The maximum absolute value search for a pivot can then be done in parallel over processors. The pivot row is scaled sequentially by the process owning it and then broadcast to all processes, which update the rows they own.
by row wise Gauss elimination. The forward and back substitution can now be done in parallel by the
column sweep method, with individual solution vector elements being broadcast one at a time to all
processes as they are produced.

Write the distributed program at the same level as the column wise Gauss elimination in the text.
Assume that communication is fairly efficient compared to computation so that building long mes-
sages is less important than maximizing parallel computation.

**Solution: 5.8** The distribution of the problem over processors by row of the matrix is shown below.

The drivers for the Host and Worker processors have the form:

**Host:** Input number \( p \) of processors and order \( n \) of linear system of equations

- Broadcast p and n to all workers;
- Receive time to generate test case from process 0, and print it;
- Receive time for LU factorization from process 0, and print time and rate;
- Receive solution time for test \( b \) vector from process 0, and print time and rate;
- Receive residual from process 0, check and print it.

**Worker (process \( id \)):**

- Receive \( p \) and n from Host, and set starting time;
- Compute number, \( m \), of rows for process \( id \);
- Generate \( m \) rows of the test matrix, \( A[i,j] = 1/(i - j + 0.5) \), for process \( id \);
- Process 0 sends time for generation to Host;
- Call PGEFA to factor \( A \) into the matrix product \( L \times U \);
- Process 0 sends time for factorization to Host;
- Compute \( m \) elements of test RHS vector \( b \) for process \( id \);
- Call PGESL to solve equations, leaving solution vector \( x \) distributed over processes;
- Process 0 sends time for solving to Host;
- Compute \( m \) elements of test \( A \times [i] \) for process \( id \);
- Call PGEMUL to compute \( A \times [i] \) with result vector distributed over processes;
- Cooperatively compute \( \sum (A[x[i]] - b[i]) \) with result appearing in process 0;
- Process 0 sends residual to Host.
The matrix multiply routine, PGEMUL, in process id has the form shown below:

**PGEMUL:**

Collectively gather \( x[p..m-1] \) and send result to all processes;
Reorder gathered \( xp \) elements into copy of vector \( x[0..n-1] \) in all processes;

```plaintext
for q := 0 .. m-1 begin ;compute m elements of b in each processor;
   bp[q] := 0;
   for k := 0 .. n-1
      bp[q] := bp[q] + Ap[q, k]x[k];
end of prescheduled for over elements of bp;
Return.
```

The LU factorization routine for process id appears as follows:

**PGEFA:**

Initialize pointer to current row for this process, \( q := 0; \)

```plaintext
for k := 0 .. n-1 begin ;loop over pivot positions;
   r := k mod p, identify root process as owner of kth row;
   lmax := 0.0; lidx := 0, initialize local maximum and its index;
   for ii := q .. m-1 begin ;search rows below diagonal for local maximum absolute value;
      if (|Ap[ii, k]| > lmax) then
         begin lmax := |Ap[ii, k]|; lidx := ii;p + id; end
      end for local maximum search;
      Collectively do index of maximum reduction, leaving result, \( piv[k] \), in all processes;
   if (piv[k] \ne k) then begin
      if (id = r) then begin, if this is the root process;
         Copy Ap[q, k..n-1] to buffer; Send buffer to process piv[k] mod p;
         Receive Ap[q, k..n-1] from process piv[k] mod p;
      end if root process;
      if (id = piv[k] mod p) then begin;
         Copy Ap[q, k..n-1] to buffer; Send buffer to process r;
         Receive Ap[q, k..n-1] from process r;
      end if pivot row owner;
   end if pivot row not on diagonal;
   if (id = r) then scale pivot row by reciprocal of pivot, \( Ap[q, k+1..n-1]/Ap[q, k] \);
   Broadcast scaled pivot row, \( pr[k..n-1] \) from root r to all processes;
   if (id = r) then q := q + 1; advance row pointer past pivot row in process r;
   for ii := q .. m-1 begin ;loop over rows below diagonal in each process;
      Ap[ii, k] := -Ap[ii, k]pr[k], scale by pivot to get multiplier for this row;
      Ap[ii, k+1..n-1] := Ap[ii, k+1..n-1] + Ap[ii, k]pr[k+1..n-1], reduce row;
   end for rows below the diagonal;
end for loop over pivot positions;
Return.
```
The solution of the linear system is completed by forward and back substitution in PGESL.

PGESL: (process id)

Forward substitution: Initialize pointer to first row for this process, \( q := 0 \);

\[ \text{for } k := 1 \text{.. } n-1 \text{ begin, solve forward recurrence using column sweep; } \]

\[ \text{if } ( piv[k] \neq k ) \text{ then} \]

\[ \text{Processes } k \text{ mod } p \text{ and } piv[k] \text{ mod } p \text{ swap their elements } bp[q] \text{ of the RHS vector; } \]

\[ \text{if } ( id = k \text{ mod } p ) \text{ then begin} \]

\[ bk := bp[q]; q := q + 1; \]

\[ \text{end if diagonal row owner; } \]

Broadcast \( bk \) to all processes;

\[ \text{for } ii := q \text{.. } m-1, \text{ column sweep on } k \text{th column; } \]

\[ bp[q] := bp[q] + Ap[q, k] \times bk; \]

\[ \text{end for } k \text{ forward sweep; } \]

Back substitution: Initialize pointer to last row for this process, \( q := m-1 \);

\[ \text{for } k := n-1 \text{.. } 0 \text{ begin} \]

\[ \text{if } ( id = k \text{ mod } p ) \text{ then begin (diagonal owner) } \]

\[ xp[q] := bp[q]/Ap[q, k]; xk := xp[q]; q := q - 1; \]

\[ \text{end if diagonal owner; } \]

Broadcast \( xk \) to all processes;

\[ \text{for } ii := q \text{.. } 0, \text{ column sweep on } k \text{th column; } \]

\[ bp[q] := bp[q] - xk \times Ap[q, k]; \]

\[ \text{end for loop for back substitution; } \]

Return

**Problem: 5.9** A computation on two length \( N \) vectors \( A \) and \( B \), defined by the sequential pseudo code below, is to be done on a distributed memory multiprocessor.

\[ \text{B[1]} := \text{A[1]}; \]

\[ \text{for } i := 2 \text{ step 1 until } N \]

\[ \text{B[i]} := ( \text{A[i]} + \text{A[i-1] } ) / 2. ; \]

The machine has \( P \) processors, where \( N = n \times P \), and communication is by means of non blocking send(proc, expr) and blocking recv(proc, var). The parameter proc is a processor number, expr is an expression giving the value to be sent, and var is the name of a variable into which the received value is stored.

Assume that in processor \( i \), \( 0 \leq i \leq P-1 \), the values of \( A[i \times m+1:i \times m+m] \) are stored as a vector \( a[1:m] \) and that \( B[] \) is similarly mapped to \( b[] \) in each processor. Write distributed memory multiprocessor pseudo code that will work for any processor \( i \) in the system to do the above computation. Structure the code to minimize completion time in view of message latency.

**Solution: 5.9** Since \( a[m] \) is needed by the next higher numbered processor, it is sent before any computation is done so the message latency can be overlapped by most of the computation.
Problem: 5.10 The pseudo code below represents a sequential version of odd/even transposition sort. It sorts an N element array, a[0:N-1], for N even.

while notsorted do
  begin
    for i := 0 step 2 until N-2 do
      compare&swap(a[i], a[i+1]);
    for i := 1 step 2 until N-3 do
      compare&swap(a[i], a[i+1]);
  end;

It is assumed that compare&swap is a subroutine which compares its two arguments and swaps them if they are not in order. It is also assumed that notsorted will be changed from true to false when sorting is complete.

(a) How should the array a[0:N-1] be spread over the memories of a P processor fragmented memory multiprocessor in order to implement a parallel version of this program? Assume that P divides N evenly, so N = Pm, where m is an integer.

(b) Write pseudo code for processor i of the P processors assuming i is neither the first nor the last processor. Assume a non blocking send(process),(value) and a blocking receive(process),(variable) for communication.

Solution: 5.10 (a) The array a[0, N-1] should be spread over the P processors so that processor i memory stores array elements a[m*i] to a[(m+1)*i – 1]. We give the private name ap[0:m-1] to this array section in each processor.

(b) We assume m is even and m ≥ 6. The code for processor i, where i is neither 0 or P-1 is

while (notsorted)
  compare&swap(ap[0], ap[1]);
  send(P_{i-1}, ap[0]);
  compare&swap(ap[m-2], ap[m-1]);
  send(P_{i+1}, ap[m-1]);
  for j := 2 step 2 until m-4
    compare&swap(ap[j], ap[j+1]);
  for j := 1 step 2 until m-3
    compare&swap(ap[j], ap[j+1]);
  receive(P_{i-1}, t);
  ap[0] := max(t, ap[0]);
  receive(P_{i+1}, t);
  ap[m-1] := min(t, ap[m-1]);
end while;
Problem: 5.11 $x = Ax + y$ is a general recurrence system, where $x$ and $y$ are $N$ element vectors and $A$ is an $N \times N$ lower triangular matrix, with $a_{ij} = 0$ for all $1 \leq i \leq j \leq N$. It is to be solved on an $N$ processor distributed memory multiprocessor where processors communicate only by `send(dest, value)` and `receive(source, value)`.

(a) How should elements of $x$, $y$, and $A$ be stored in the separate memories $M_i$ of processors $P_i$, $1 \leq i \leq N$, for solution using the column sweep method?

(b) Write a single MIMD pseudo code program for any processor $P_i$ using `send` and `receive` to accomplish the column sweep solution.

Solution: 5.11 (a) The sequential pseudo code for the recurrence solver can be written as below.

```plaintext
x[1] := y[1];
for j := 1 step 1 until N-1
  for i := j+1 step 1 until N
    x[i] := x[i] + a[i, j]x[j];
```

To process a column at a time, each process should do the operations in one row. This means that process $i$ will produce $x[i]$ using row $i$ of $A$, $a[i, 1:i-1]$, and $y[i]$. Process $i$ will need to send $x[i]$ to processes $i+1$ through $N$ as soon as its computation is complete so that they can use it in their computations. Private variables $ap[1:i-1]$, $xp$, and $yp$ represent the portions of $A$, $x$, and $y$ in processor $i$.

(b) After the correct portions of $y$ and $A$ have been distributed to each processor, the code for processor $P_i$ can be written as shown below.

```plaintext
real xp, yp, ap[1:i-1], xx;
integer i, j, k;
xp := yp; /* Initialize x[i] to y[i].*/
if (i ≠ 1) then /* x[1] is complete and can be sent.*/
  for j := 1 step 1 until i-1 /* Step through previous x[j].*/
    begin
      receive(j, xx); /* Receive an x[j].*/
      xp := xp + ap[j]x*xx; /* and do the multiply/add.*/
    end;
  for k := i+1 step 1 until N /* Send x[i] to all.*/
    send(k, xp); /* following processes.*/
```

Problem: 5.12 The function below is called in parallel by $N = 2^K$ processors of a distributed memory multiprocessor. Processes run on separate processors and communicate using:

- `Send(processor_number, value)` Send the value to the specified processor and do not wait for a response.
- `Receive(processor_number, value)` Wait for a message to arrive from the specified processor and return its value.
REAL FUNCTION Q(K, VALUE, ME)
INTEGER K, ME
C = VALUE
DO 10 L = 1, K
   NB = IXOR(ME-1, 2**((L-1)))+1
   Send (NB, C)
   Receive (NB, C1)
   C = C + C1
10 CONTINUE
Q = C
RETURN
END

K is the same constant in all processors. ME is the unique number of each processor, 1 ≤ ME ≤ N. The function IXOR performs the bit wise exclusive or on its two integer arguments, and ** is the Fortran exponentiation operator.

(a) What function is performed by Q when called by all N processors?
(b) Describe the synchronization properties of the statement S = Q(K, V, ME) performed in parallel by N=2**K processors.

Solution: 5.12 (a) Each processor obtains the sum of the N values passed in the VALUE parameter to Q by the N calls of Q on N different processors.

(b) Since each processor exits the function Q with the sum of values passed by other processes on entering Q, it is clear that no process may exit Q before all have entered it. Therefore, S = Q(K, V, ME) performs a barrier synchronization.

Problem: 5.13 Suppose that processors of a machine executing Cannon’s algorithm can transmit left or upward by any power of two steps. sendl(v, k) sends v to the processor 2^k steps to the left and recvr(v, k) receives v from the processor 2^k steps to the right. sendu(v, k) and recvb(v, k) operate similarly for upwards transmission.

Rewrite Program 5-9 using these operations to reduce the number of sequential transmission steps needed to get elements of A and B to their starting positions for Cannon’s matrix multiply. No more than log2(N) transmissions should be used to get an element to its destination.

Solution: 5.13 Processes in column j have to rotate their B elements j steps upwards. If we have send/receive functions that are able to transmit over any power of two steps, we only need to break the transmission over j steps into a series of up to log2(j) such functions, corresponding to the breakdown of j in binary digits. Similarly for the rows.

Assume we have an and function which returns the bitwise AND between two operands. Note that recvb(*, k) only receives a message from sendu(*, k) for the same k, and so do recvr and sendl. The following code is for processor P(i, j):
for k := 0 step 1 until log₂(N)-1 begin
  if and(j, 2^k) ≠ 0 then begin
    sendu(myB, k);
    recvB(myB, k);
  end;
  if and(i, 2^k) ≠ 0 then begin
    sendl(myA, k);
    recvR(myA, k);
  end;
end;

Problem: 5.14
Modify Program 5-11 to use MPI collective communication to gather the blocks of a row for output. Start by considering a simple case where all processes can output to the same printer. Then consider the case in which only the process with rank zero in MPI_COMM_WORLD can do output. Reordering processes in building the comsq communicator could be reconsidered, causing a modification in the procgrid subroutine.

Solution: 5.14
If all processes can access the printer, then the first process in each process row can gather results and print b rows of the bN×bN matrix. We will pack b×b blocks into the message buffer to optimize transmission time.

REAL blockbuf(2500), printbuf(25000)
DO j = 1, b
  DO i = 1, b
    blockbuf(i+(j-1)*b) = Cp(i, j)
  END DO
END DO
CALL MPI_GATHER(blockbuf, b*b, MPI_REAL, printbuf, b*b, MPI_REAL, 0, comrow, ierr)
DO ii = 0, N-1
  IF (mycol.EQ.0 .AND. myrow.EQ.ii) THEN
    CALL printrow(printbuf, b, N)
  END IF
CALL MPI_BARRIER(comsq, ierr)
END DO
CALL MPI_FINALIZE
END

Since reordering has been allowed in applying the topology, we do not know to which row or column of processes the process with rank 0 in MPI_COMM_WORLD belongs. We will thus let the process in (row 0, column 0) gather the results and send them to the processor connected to the printer. There is a chance the processor will send a message to itself.

REAL blockbuf(2500), printbuf(25000), allprint(250000)
DO j = 1, b
  DO i = 1, b
    blockbuf(i+(j-1)*b) = Cp(i, j)
  END DO
END DO
CALL MPI_GATHER(blockbuf, b*b, MPI_REAL, printbuf, b*b, MPI_REAL, 0, comrow, ierr)
IF (mycol .EQ. 0) THEN
    CALL MPI_GATHER(printbuf, N*N*b*b, MPI_REAL, allprint, b*b,
&           MPI_REAL, 0, comrow, ierr)
    IF (myrow .EQ. 0) THEN
       CALL MPI_SEND(allprint, N*N*b*b, MPI_REAL, 0, 1,
&                   MPI_COMM_WORLD, ierr)
    END IF
END IF
CALL MPI_COMM_RANK(MPI_COMM_WORLD, iorank, ierr)
IF (iorank .EQ. 0) THEN
    CALL MPI_RECV(allprint, N*N*b*b, MPI_REAL, MPI_ANY_SOURCE, 1,
&                   MPI_COMM_WORLD, ierr)
    DO 100 i = 1, N
       DO 100 ii = 1, b
          DO 100 j = 1, N
             DO 100 jj = 1, b
                PRINT *, allprint(ii+(jj-1)*b+(j-1)*b*b+(i-1)*b*b*N)
            100 CONTINUE
        END IF
    END IF
CALL MPI_FINALIZE
END
Problem: 6.1  A network implements one property from each of the following three categories:
(a) Circuit or packet switched
(b) Central or distributed control
(c) Synchronous or asynchronous mode of operation
Describe the implications of all possible combinations.

Solution: 6.1

Problem: 6.2  Design an efficient prefix sum algorithm for a \( m \)-level binary tree network. What is the complexity of your algorithm?

Solution: 6.2  Assume a full binary tree network with \( N = 2^m \) processors. Let \( v_i \), \( 0 \leq i \leq N-1 \), be the input values to the prefix from each processor, and let \( q_i \), \( 0 \leq i \leq N-1 \), be the prefix results in each processor. Then three types of nodes, root, intermediate, and leaf, act differently to compute the prefix sum.

Root node:
- Receive \( \text{lsum} \) from left child;
- Receive \( \text{rsum} \) from right child;
- Send zero to left child;
- Send \( \text{lsum} \) to right child;

Intermediate node:
- Receive \( \text{lsum} \) from left child;
- Receive \( \text{rsum} \) from right child;
- Send \( \text{lsum} + \text{rsum} \) to parent;
- Receive \( \text{psum} \) from parent;
- Send \( \text{psum} \) to left child;
- Send \( \text{psum} + \text{lsum} \) to right child;
Leaf node:
Send \( v[i] \) to parent;
Receive sum from parent;
Set \( q[i] := \text{sum} + v[i] \);

Counting lines of the code description, the complexity is \( 4 + 6(m - 2) + 3 = O(\log_2 N) \).

Problem: 6.3 Design and implement an all-to-all broadcast algorithm for a \( N = 2^m \) node hypercube network. Use MIMD pseudo code with one \( \text{send} \) or one \( \text{receive} \) at a time. What is the time complexity of your algorithm?

Solution: 6.3 We assume a \( N = 2^m \) node hypercube with nodes numbered \( 0 \leq me \leq N - 1 \). \( me \) is available to the node through a system call or fixed parameter. Assume a non-blocking send operation of the form \( \text{send}(\text{buf}, \text{size}, \text{dest}) \) that sends a buffer of \( \text{size} \) items to processor \( \text{dest} \) and a blocking receive of the form \( \text{receive}(\text{buf}, \text{size}, \text{source}) \). The algorithm has \( m \) transmission steps, \( 0 \leq k \leq m - 1 \), where pairs of processors differing in their \( k \)th index bit exchange buffers that double in size for each new \( k \). At the end, the order of the items in the buffer is unscrambled to produce a result array, where \( \text{result}[i] \) is the item broadcast by the \( i \)th processor.

```plaintext
integer m, N, me, k, i, metmp, size;
real myitem, sbuf[0:N-1], rbuf[0:N-1], result[0:N-1];

sbuf[0] := myitem;
size := 1;
for k := 0 step 1 until m-1
    begin
        send(sbuf, size, me ⊕ 2^k);
        receive(rbuf, size, me ⊕ 2^k);
        for i := 0 step 1 until size-1
            sbuf[size+i] := rbuf[i];
        size := 2Xsize;
    end;

for i := 0 step 1 until N-1
    result[i] := sbuf[me ⊕ i];
```

In parallel, each process does \( \log_2 N \) sends and receives. There is a bidirectional \( \text{send} \) on each link at each step if possible. If only unidirectional transmissions are supported, one of the sends will have to be buffered, giving \( 2\log_2 N \) transmissions. There are \( 2N \) word copy operations to merge the buffers and put the results in order at the end. If \( T_m \) is the time for a bidirectional transmission start up, \( T_w \) is the extra time for each word transmitted, and \( T_c \) is the time for copying a word, then the time complexity is \( T_m \log_2 N + (N - 1)T_w + 2NT_c \).

Problem: 6.4 Design and implement a multicast algorithm, i.e. a single input can be broadcast to any subset of outputs, for a \( N=2^m \) node hypercube network.

Solution: 6.4

Problem: 6.5 Describe a routing algorithm for the butterfly network similar to the destination tag method described for the omega network in Section 6.4.3.3.

Solution: 6.5 For an \( m \) column butterfly network, there are \( m-1 \) permutations, \( P_i \), \( 1 \leq i \leq m-1 \). \( P_i \) carries an input \( q_{m-1} q_{m-2} \ldots q_i \ldots q_1 q_0 \) to the output \( q_{m-1} q_{m-2} \ldots q_0 \ldots q_i \). If all exchange boxes were
set to a bar state, a source \( s = s_{m-1} s_{m-2} \ldots s_1 s_0 \) would pass through
\( s_{m-1} s_{m-2} \ldots s_1 s_0 s_1, \ s_{m-1} s_{m-2} \ldots s_1 s_0 s_2, \ \ldots, \ s_{m-1} s_{m-2} \ldots s_{i+1} s_i \ldots s_1 s_0 s_1, \ s_{m-2} \ldots s_1 s_0 s_{m-1} \). In this sequence, each bit of the source index appears at the low order bit position and can be changed to a bit of the destination index by proper setting of the associated exchange box. In particular, we want to change bit \( s_i \) to destination bit \( d_{i+1} \), \( 0 \leq i \leq m-2 \), and \( s_{m-1} \) to \( d_0 \). If this is done, then the switch encountered on the path from \( s \) to \( d \) in column \( i \) will be in row \( r_i \), where \( r_i = s_{m-1} s_{m-2} \ldots s_i d_{i-1} \ldots d_1 d_0 \). The settings should be \( E_{r_i} = s_i \oplus d_{i+1} \), for \( 0 \leq i \leq m-2 \), and \( E_{r_{m-1}, m-1} = s_{m-1} \oplus d_0 \) in column \( m-1 \).

**Problem: 6.6** We know that any single source in an omega network, \( s = s_{n-1} s_{n-2} \ldots s_1 s_0 \), can be routed to any destination. If \( s \) is routed to the destination, \( d = s_{n-1} s_{n-2} \ldots s_1 s_0 \) with destination tag equal to the complement of the source tag, then exhibit some other source-destination pair which cannot be connected simultaneously and prove that it cannot be.

**Solution: 6.6** The destination tag method for routing in an omega network is constructive and specifies that, in order to route from \( s = s_{n-1} s_{n-2} \ldots s_1 s_0 \) to \( d = s_{n-1} s_{n-2} \ldots s_1 s_0 \) with destination tag equal to the complement of the source tag, then exhibit some other source-destination pair which cannot be connected simultaneously and prove that it cannot be.

Routing any source of the form \( q_{n-1} \ldots q_1 s_{n-2} \ldots s_1 s_0 \) to a destination of the form \( d_{n-1} \ldots d_1 d_0 \) for any values of the \( q \) and \( d \) bits requires this same switch to be set in the bar state. More specifically, for any \( i \neq n-1 \), source \( \hat{s} = s_{n-1} s_{n-2} \ldots s_i s_0 \) cannot be routed to destination \( \hat{d} = s_{n-1} s_{n-2} \ldots s_1 s_0 \) simultaneously with the routing of \( s \) to \( d \).

**Problem: 6.7** Consider using “exchange” boxes with three inputs and three outputs each to build a multi-stage switching network. Assume initially that a box can connect inputs to outputs in any permuted order. (There is no broadcast capability.)

(a) Using these elements and a three way “perfect shuffle” interconnection, show how to build a two stage network connecting nine inputs to nine outputs that is analogous to the omega network.

(b) Define the three way perfect shuffle permutation of \( n = 3^k \) elements mathematically.

(c) Use your definition to prove that a \( k \) stage “ternary omega network” can connect any specific one of \( n = 3^k \) inputs to any one of \( n \) outputs.

(d) Is it necessary that the three input, three output “exchange” boxes be able to perform all permutations of three elements for the proof of part c to work? If not, give a minimal set of permutations which is sufficient.
Solution: 6.7 (a)

(b) Let the index $S$, $0 \leq S \leq 3^k$, of the input to the three way perfect shuffle permutation have the base 3 digits $s_{k-1}s_{k-2}...s_0$. Then this input is connected to the output with the index $D = (s_{k-2}s_{k-3}...s_0s_{k-1})_3$.

(c) Using the "destination tag" method, a constructive proof that the input with index $S = (s_{k-1}s_{k-2}...s_1s_0)_3$ can be connected to the output with index $D = (d_{k-1}d_{k-2}...d_0)_3$ is as follows.

Number the exchange boxes at stage $m$ from top to bottom as $E_{i,m}$, $i = 0, 1, ..., 3^{k-1} - 1$. After the first shuffle, input $S$ will appear at the $s_{k-1}$st input to exchange box $E_{j,1}$, where $j = (s_{k-2}s_{k-3}...s_1s_0)_3$. Set the permutation for $E_{j,1}$ to route input $s_{k-1}$ to output $d_{k-1}$. In general, set the exchange box $E_{j,m}$, $j = (s_{k-m}s_{k-m-1}...s_1s_0d_{k-1}...d_{k-m+1})_3$, to route input $s_{k-m}$ to output $d_{k-m}$. The proof of the correctness of this construction is that if the input signal has been routed to position $(s_{k-m}s_{k-m-1}...s_1s_0d_{k-1}...d_{k-m+1})_3$ by the first $m-1$ stages of the network, then it will be routed to position $(s_{k-m}s_{k-m-1}...s_1s_0d_{k-1}...d_{k-m})_3$ after stage $m$. Recursive application of this fact proves that input $S$ will be routed to output $D$ after $m = k$ stages.

(d) The proof only requires a permutation that will route any one specified input of a 3-way exchange element to any one specified output. This can be accomplished by only three of the six possible permutations, e.g. the identity, a cyclic left shift of one, and a cyclic left shift of two.

Problem: 6.8 An $N$ input multistage cube interconnection network consists of $m = \log_2 N$ exchange networks (each network is a column of $N/2$ function exchange elements) each followed by a permutation interconnection $P_k$, $k = 1, 2, ..., m$ reading from left to right across the network, with inputs at the left. The permutation interconnections transform the binary representations of their input indices as follows:

for $1 \leq k < m$, $P_k(i_{m-1}, ..., i_k, i_{k+1}, i_{k-1}, ..., i_1, i_0) = (i_{m-1}, ..., i_{k+1}, i_{k-1}, ..., i_1, i_k)$,

for $k = m$, $P_m$ is an inverse shuffle $P_m(i_{m-1}, i_{m-2}, ..., i_2, i_1, i_0) = (i_0, i_{m-1}, i_{m-2}, ..., i_2, i_1)$.

(a) Draw an 8 input multistage cube network.

(b) Prove that any input $S = s_{m-1}, s_{m-2}, ..., s_1, s_0$ can be connected to any output.
\[ D = d_{m-1} d_{m-2} \ldots d_1 d_0 \] by a proper setting of exchange boxes.

(c) Assume input \( S = s_{m-1} \ldots s_k + 1 \bar{s}_k s_{k-1} \ldots s_1 s_0 \) is connected to output \( S = s_{m-1} \ldots s_k + 1 \bar{s}_k s_{k-1} \ldots s_1 s_0 \), where \( \bar{s}_k = 1 - s_k \) is the complement of the bit. Exhibit an input/output pair which cannot also be connected at the same time.

**Solution: 6.8** (a)

\[
\begin{array}{ccccccc}
000 & 000 & 000 & 000 & 000 & 000 & 000 \\
\overline{001} & 001 & 001 & 001 & 001 & 001 & 001 \\
010 & 010 & 010 & 010 & 010 & 010 & 010 \\
011 & 011 & \overline{011} & 011 & 011 & 011 & 011 \\
000 & 000 & 000 & 000 & 000 & 000 & 000 \\
100 & 100 & 100 & 100 & 100 & 100 & 100 \\
101 & 101 & \overline{101} & 101 & 101 & 101 & 101 \\
110 & 110 & 110 & 110 & 110 & 110 & 110 \\
111 & 111 & 111 & 111 & 111 & 111 & 111 \\
\end{array}
\]

Permutation \( P_k \) exchanges the low order bit, bit 0, of the input index with bit \( k \) of the input index to get the output index. \( P_m \) puts the bits back in order with a circular right shift.

(b) As an example for \( N = 8 \). \( P_1(a_2a_1a_0) = a_2a_0a_1 \), \( P_2(a_2a_0a_1) = a_1a_0a_2 \), and the perfect shuffle, \( P_2(a_1a_0a_2) = a_2a_1a_0 \).

Let the source index be \( A = a_{m-1} \ldots a_1 a_0 \) and the destination index be \( B = b_{m-1} \ldots b_1 b_0 \). In general, \( P_k \), \( 1 \leq k \leq m - 1 \), moves source address bit \( a_k \) into the low order bit position and moves \( b_{k-1} \) one place to the left of where it belongs in the destination index. The exchange at stage \( k \) is set to map bit \( a_k \) now in the low order position, to \( b_k \). The last inverse shuffle right circularly shifts the index bits back into their correct positions.

(c) If input \( S = s_{m-1} \ldots s_k + 1 s_k s_{k-1} \ldots s_1 s_0 \) is connected to output \( D = s_{m-1} \ldots s_k + 1 s_k s_{k-1} \ldots s_1 s_0 \), then \( \hat{S} = s_{m-1} \ldots s_k + 1 s_k s_{k-1} \ldots s_1 s_0 \) cannot be connected to \( S = s_{m-1} \ldots s_k + 1 s_k s_{k-1} \ldots s_1 s_0 \). The first pair requires that the exchange box in the \( k \)th column with index \( s_{m-1} \ldots s_k + 1 s_k s_{k-1} \ldots s_1 s_0 \) be set in the cross state, while connecting the second pair requires this same switch to be in the bar state. For example, in the \( N = 8 \) network of part (a), the two (input, output) pairs \((000, 010)\) and \((010, 110)\) cannot be connected at the same time.

**Problem: 6.9** Consider an omega network with \( N \) inputs and \( N \) outputs, where \( N = 2^m \). Describe how to
set the exchange switches so that every input $i$ is connected to output $j$, where $j$ differs from $i$ by $2^k$, for some fixed $k < m$, $0 \leq i \leq N - 1$.

**Solution: 6.9** The simplest interpretation of this problem notes that $s_{m-1}s_{m-2}\ldots s_1s_0$ differs from $s_{m-1}s_{m-2}\ldots s_1s_0$ by $2^i$, though by plus or minus depending on the value of $s_i$. Thus setting all switches in column $m-i$ to the cross state and all others to the bar state will perform a permutation that fits the problem specifications.

**Problem: 6.10** Can the switching network below, built of $2 \times 2$ exchange elements perform all permutations from inputs to outputs? Prove or disprove as simply as possible.

**Solution: 6.10** There are $6! = 720$ permutations of 6 inputs but only $2^9 = 512$ different ways of setting the 9 exchange switches, so the network cannot perform all permutations.

**Problem: 6.11** A switching network connecting $N$ processors to $N$ memories has queueing at the nodes so that a message which cannot be switched to the correct node output is delayed for a cycle time $\Delta$. The omega network has 2-input, 2-output switching elements and depth $\log_2 N$. This network has $k$-input, $k$-output nodes and depth $\log_2 N$. We assume a simplified operation model in which, at some initial time, all processors make memory requests simultaneously, and the requests are then satisfied before anything else occurs. A request thus takes a minimum of $2\Delta \log_2 N$ time units to be satisfied. Conflicts on the way to memory may cause the time to be longer. (There are no conflicts on the return path.) Assume that the expected waiting time at a node, $E(\text{wait}) = pk\Delta$, is proportional to the number of inputs $k$ and the probability $p$ of collision with one other input. Derive an equation for the number of inputs (and outputs) $k$ which will minimize the expected time for a request to be satisfied. Discuss qualitatively the behavior of the optimal value of $k$ for very small and very large collision probability $p$. 
Solution: 6.11 The structure of the network is

\[ E(\text{wait}) = pk\Delta \]

Expected round trip time to memory

\[ E(t_M) = 2\Delta \log_k N + pk\Delta \log_k N \]

where \( \log_k N \) is the travel time and \( pk\Delta \) is the expected wait per stage. To minimize \( E(t_M) \) we want to solve

\[
\frac{d}{dk}(\log_k N(2 + pk)\Delta) = \Delta \frac{d}{dk}\left(\frac{\ln N}{\ln k}\right) = \Delta \frac{d}{dk}\left(\frac{2 + pk}{\ln k}\right) = 0 , \text{ or}
\]

\[
\frac{d}{dk}\left(\frac{2 + pk}{\ln k}\right) = \frac{1}{\ln k} - \frac{1}{(\ln k)^2}(2 + pk) = 0 , \text{ or}
\]

\[
p - \frac{2 + pk}{k \ln k} = 0 , \text{ or } pk\ln k = 2 + pk , \text{ or } pk(\ln k - 1) = 2 .
\]

As \( p \to 0 \) we find that \( k \to \infty \) which just says that, if there are no conflicts, then a single switching level minimizes the expected memory access time. If \( p=1 \) we have a high probability of collision and must solve \( k(\ln k - 1) = 2 \) for \( k \).

<table>
<thead>
<tr>
<th>( k )</th>
<th>( k(\ln k - 1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7 ( \approx e )</td>
<td>0</td>
</tr>
<tr>
<td>7.29 ( \approx e^2 )</td>
<td>7.29</td>
</tr>
<tr>
<td>4.46 ( \approx e^{1.5} )</td>
<td>2.23</td>
</tr>
</tbody>
</table>

Hence \( k = 4 \) is about right for \( p = 1 \). Under these assumptions we should use an "omega network" using base 4 or more with the base increasing as the probability of conflict decreases.

Problem: 6.12 Give a mathematical description of the perfect shuffle permutation on \( N = 2^k \) inputs.

Solution: 6.12

Problem: 6.13 Can an \( N \)-input \( m \)-stage omega network, \( N = 2^m \), pass a perfect shuffle? Show your work.
Solution: 6.13

**Problem: 6.14** Consider an \( N \)-input multistage omega network where each switch cell is individually controlled and \( N = 2^n \).

(a) How many different permutation functions (one-to-one and onto mappings) can be defined over \( N \) inputs?

(b) How many different permutation functions can be performed by the omega network in one pass? If \( N = 8 \), what is the percentage of permutation functions that can be performed in one pass?

(c) Given any source-destination \((S - D)\) pair, the routing path can be uniquely controlled by the destination address. Instead of using the destination address \( D \) as the routing tag, we define \( T = \overline{S} \oplus D \) as the routing tag. Show that \( T \) alone can be used to determine the routing path. What is the advantage of using \( T \) as the routing tag?

(d) If the 2\( \times 2 \) switches have all 4 functions, the omega network is capable of performing broadcasting (one source and multiple destinations). If the number of destination PEs is a power of two, can you give a simple routing algorithm to achieve this capability?

*Proof or disprove that the multistage omega network can perform any shift permutation in one pass. The shift permutation is defined as follows: given \( N = 2^n \) inputs, a shift permutation is either a circular left shift or a circular right shift of \( k \) positions, where \( 0 \leq k \leq N \).

**Solution: 6.14**

(a) The number of permutations of \( N \) inputs is \( N! \).

(b) The number of permutations for an \( N \) input Omega network is the number of ways of setting \( N/2 \log_2 N \) switches to cross or bar, or

\[
\frac{N \log_2 N}{2} = \frac{N}{2}.
\]

For \( N = 8 \) this is \( 8 \times 4 = 40,966 \), but \( 8! = 40,320 \) so only \(~10\%\) of the permutations can be done.

(c) To route from \( S = s_{n-1}s_{n-2}...s_0 \) to \( D = d_{n-1}d_{n-2}...d_0 \), switch \( E_{ij} \) in column \( j \) and row \( r = s_{j-1}s_{j-2}...s_0d_{n-1}d_{n-2}...d_{j+1} \) is set to \( s_j \oplus d_j \). If we attach a tag \( T = S \oplus D \) to the message to be switched, and do a left circular shift of the tag at every switch input, the low order tag bit determines the switch setting.

The switch is not explicitly located by \( T \), but previous switch settings connect the message and associated tag to the proper switch.

(d) If a routing to destination \( D = d_{n-1}d_{n-2}...d_0 \) is modified by broadcasting to both switch outputs in column \( j \), then the message will reach the two destinations, \( D = d_{n-1}d_{n-2}...d_{j+1}x_{j+1}x_{j-1}...d_0 \). If switches in \( q \) columns on the path are set to broadcast, the message reaches \( 2^q \) destinations of the form \( d_{n-1}d_{n-2}...x...d_{j+1}x_{j+1}...d_0 \) where the \( q \) positions marked \( x \) take both values 0 and 1 independently, and the other \( d_i \) are set by the usual routing algorithm.

Algorithm: To broadcast from \( S = s_{n-1}s_{n-2}...s_0 \) to \( 2^q \) destinations whose indices match, except in \( q \) "don't care" positions, set switches in columns corresponding to fixed destination bits to \( s_j \oplus d_j \) and set switches in "don't care" columns \( k \) to broadcast from input \( s_k \).

The algorithm limits the destinations but can be used with a destination tag of \( n \) base 3 digits representing 0, 1, or "don't care". Broadcast to an arbitrary set requires sending all destinations with the
message and partitioning them at each two way broadcast. This is unlikely to be implemented in hardware.
Chapter 7: Solutions

Problem: 7.1 Suppose the upper loop limit $n$ in Figure 7-13 were not constant but represented by an incoming token. Expand the data flow diagram to make the number of copies of this token needed by the $n$ termination tests.

Solution: 7.1

Problem: 7.2 Revise the data flow graph of Figure 7-17 to protect against erroneous computation if new inputs arrive before a previous computation is complete. New inputs should only be accepted after iterations for a previous computation are finished.

Solution: 7.2
Problem: 7.3 Draw a data flow diagram that accepts an integer input \( n \) and constructs a one dimensional array of the first \( n + 1 \) Fibonacci numbers, \( F(0), F(1), \ldots, F(n) \), where \( F(0) = 0, F(1) = 1 \), and 
\[ F(i) = F(i-1) + F(i-2) \] for \( i \geq 2 \).

Solution: 7.3

Problem: 7.4 Consider the following Fortran program fragment.

```
DO 20  K = 1, N
    DO 10  J = K, M
        C(K) = C(K) + A(K, J)*B(K, J)
    10  CONTINUE
20  CONTINUE
```

(a) Draw dependence graphs, both for the assignment to show loop carried dependence and in iteration space to show how statements for different \((K, J)\) pairs are dependent.

(b) Take the program from part (a) and interchange the \( K \) and \( J \) loops. Draw the two dependency graphs of the resulting program.

(c) If the goal is vectorization, which of the two preceding programs would you prefer? What if the goal is parallelization for a shared-memory MIMD computer? Explain your answer.

Solution: 7.4 (a) The nested loops make sense for \( N \leq M \).

Two types of dependence graphs can be drawn. The diagram on the left is of more use, especially in light of the loop interchange asked for in the next part.

(b) 
```
do 20  J = 1, M
    do 10  K = 1, \text{min}(J, N)
        C(K) = C(K) + A(K, J)*B(K, J)
    10  continue
20  continue
```
(c) The version of part (b) is better for vectorization because the inner loop executions are independent, allowing it to be turned into a vector statement.

\[
\text{do 20 } J = 1, M \\
\quad C(K) = C(K) + A(K, J) \times B(K, J), \ (1 \leq K \leq \min(J, N) ) \\
\text{20 continue}
\]

Problem 7.5 Identify the basic blocks in the following C program:

```c
void f(int x) {
    int a, b, c;
    a = x + 3;
    b = a + x;
    if (b > a) {
        c = x + b;
        b += c;
    } else {
        c = x - b;
        b *= c;
    }
    while (x > b) {
        x -= a*a + 1;
    }
    printf ("x is %d\n", x);
}
```

Solution 7.5

Problem 7.6 Show the dependence graphs among statements of each of the following Fortran program fragments, with justification.

(a) 

\begin{align*}
\text{S1: } & A = B + D \\
\text{S2: } & C = A*3 \\
\text{S3: } & A = A + C \\
\text{S4: } & E = A/2
\end{align*}
(b) S1: \( X = \sin(Y) \)
S2: \( Z = X + W \)
S3: \( Y = -2.5 \times W \)
S4: \( X = \cos(Z) \)

(c) Determine the dependences in the following Do-loop. Distinguish dependences within the same iteration from those between iterations.

\[
\text{do } I = 1, N \\
\text{S1: } A(I+1) = B(I-1) + C(I) \\
\text{S2: } B(I) = A(I) \times K \\
\text{S3: } C(I) = B(I) - 1 \\
\text{10# continue}
\]

**Solution: 7.6**
(a) \( S1 \delta S2 \) flow dep. on A
\( S2 \delta S3 \) anti-dep. on A
\( S2 \delta S4 \) flow dep. on C
\( S1 \delta S3 \) output dep. on A
\( S3 \delta S4 \) flow dep. on A
\( S1 \delta S2 \) flow dep. on X
\( S2 \delta S3 \) anti-dep. on Y
\( S2 \delta S4 \) flow dep. on Z
\( S1 \delta S4 \) output dep. on X
(can be omitted)

(b) \( S1 \delta S2 \) flow dep. on X
\( S1 \delta S3 \) anti-dep. on Y
\( S2 \delta S4 \) anti-dep. on X
\( S1 \delta S4 \) output dep. on X
\( S2 \delta S3 \) flow dep. on \( B(I) \) not loop carried
\( S1 \delta S3 \) anti-dep. on \( C(I) \) not loop carried
\( S2 \delta S1 \) flow dep. on \( B(I), B(I-1) \) loop carried
\( S1 \delta S2 \) flow dep. on \( A(I+1), A(I) \) loop carried

**Problem: 7.7**
Determine the dependence relations among the three statements in the following loop nest.

Give the direction vector and distance vector for all dependence relations.

\[
\text{do } I = 1, N \\
\text{do } J = 2, N \\
\text{S1: } A(I, J) = A(I, J-1) + B(I, J) \\
\text{S2: } C(I, J) = A(I, J) + D(I+1, J) \\
\text{S3: } D(I, J) = 0.1 \\
\text{end do} \\
\text{end do}
\]

**Solution: 7.7**
\( S1 \delta_{0,0} S2 \) flow dep. on \( A(I, J) \) not loop carried Direction \((=, =)\)
\( S1 \delta_{0,1} S1 \) flow dep. on \( A(I, J), A(I, J-1) \) loop carried Direction \((=, <)\)
\( S2 \delta_{1,0} S3 \) anti-dep. on \( D(I+1, J), D(I, J) \) loop carried Direction \((<, =)\)

**Problem: 7.8**
Draw data flow graphs to represent the following computations:

(a) \( \textbf{if } (a=b) \textbf{ and } (c<d) \textbf{ then } c := c-a \)
\( \textbf{else } c := c+a \;
\)

(b) \( Z = N! = N \times (N-1) \times (N-2) \ldots \times 2 \times 1 \)

Use the \textbf{merge} operator, the \textbf{true} gate, the \textbf{false} gate, the \textbf{multiply}, \textbf{add} and \textbf{subtract} opera-
tors, the logical and, and the compare operator in your graph construction.

**Solution: 7.8 (a)**

![Graph construction diagram]

**Solution: 7.8 (b)**

![Graph construction diagram]
Problem: 7.9 The function below is written in SISAL.

\[
\text{function } f(x: \text{real}, a: \text{array}[..] \text{of real}, i, j: \text{integer} \text{ returns real, real)}
\]

\[
\text{if } i = j \text{ then } x, a[i]
\]

\[
\text{else}
\]

\[
\text{let}
\]

\[
m: \text{integer} := (i + j)/2;
\]

\[
x1, r1: \text{real} := f(x, a, i, m);
\]

\[
x2, r2: \text{real} := f(x, a, m+1, j);
\]

\[
in
\]

\[
x1\times x2, x1\times r2 + r1
\]

\[
\text{end let}
\]

\[
\text{end if}
\]

\[
\text{end function}
\]

Note that division of integers yields the integer part of the quotient.

(a) If \([a[0..n-1]]\) is a vector of reals, what is the pair of real values returned by \(f(x, a, 0, n-1)\)?

(b) If only add and multiply operations on real numbers are counted, what are the depth and size of the computation done by \(f(x, a, 0, n-1)\)?

Solution: 7.9 (a) The function returns \(x^n\) and \(\sum_{i=0}^{n-1} a[i]x^i\).

(b)

There are \(n - 1\) nodes in the tree with 2 multiplies and 1 add per node for a size of \(3(n - 1)\). The maximum depth occurs when \(n = 2^k\) and is \(2k\). For arbitrary \(n\), the depth is bounded by

\[
\text{Depth} \leq 2\lceil \log_2 n \rceil.
\]

Problem: 7.10 A sequential algorithm for finding a root \(x\) of a function \(f(x) = 0\) is the binary chopping method shown below assuming that we start with two real numbers \(b\) and \(u\) such that \(b < u, f(b) < 0\)
and \( f(u) > 0 \).

\[
\text{while } u-b > \varepsilon \text{ do} \\
\text{begin } m := (b+u)/2; \\
\quad \text{if } f(m) > 0 \text{ then } u := m \text{ else } b := m; \\
\text{end; } \\
x := m;
\]

Show how this algorithm would appear in SISAL. Write it as a procedure with inputs \( b, u, \) and \( \varepsilon \) and output \( x \).

**Solution: 7.10**

```sisa
function chop(b, u, eps: real returns real)
  if u-b <= eps then (b+u)/2 else 
    let m: real := (b+u)/2 in 
      if f(m) > 0 then 
        chop(b, m, eps) 
      else 
        chop(m, u, eps) 
    end if
  end if 
end function
```

**Problem: 7.11**

Given a sequence of weights \( \{\omega_0, \omega_1, \ldots, \omega_{k-1}\} \) and a sequence of input signals \( \{x_1, x_2, \ldots, x_n\} \), design two linear systolic arrays with \( k \) processing cells to solve the convolution problem

\[ y_j = \omega_0 x_j + \omega_1 x_{j-1} + \cdots + \omega_{k-1} x_{j-k+1}, \]

The length \( n \) of the input stream is much larger than \( k \), and it is the steady state behavior of the computation that is of interest.

(a) In the first design, you are to use the unidirectional cells which compute \( y_{\text{out}} \leftarrow y_{\text{in}} + \omega x_{\text{in}} \) as shown below.

```
\[ \begin{array}{c}
\text{in} \\
y_{\text{in}} \\
\omega \\
\text{out}
\end{array} \quad \begin{array}{c}
y_{\text{in}} \\
\omega x_{\text{in}} \\
y_{\text{out}} = y_{\text{in}} + \omega x_{\text{in}} \\
\text{out}
\end{array} \]
```

Explain your design, the distribution of the inputs, and the systolic flow of the partial results \( y_j \)'s from left to right.

(b) In the second design, you are given the bidirectional cells which compute \( y_{\text{out}} \leftarrow y_{\text{in}} + \omega x_{\text{in}} \) and \( x_{\text{out}} \leftarrow x_{\text{in}} \) as shown below.

```
\[ \begin{array}{c}
\text{in} \\
x_{\text{in}} \\
\omega \\
x_{\text{out}} \\
\end{array} \quad \begin{array}{c}
\text{in} \\
y_{\text{in}} \\
\omega x_{\text{in}} \\
y_{\text{out}} \\
\text{out}
\end{array} \quad \begin{array}{c}
x_{\text{in}} \\
x_{\text{out}} \\
\end{array} \]
```

Explain the design and operation of this systolic convolution array. Why might it be a better design than that of part (a) in terms of data movement?

**Solution: 7.11**
Chapter 8: Solutions

Problem: 8.1 This problem assumes a computer which does all synchronization using full/empty variables. The operations on them are defined as:

produce(a,x) — waits for a to be empty, stores the value of x into it, sets a full and proceeds, all in an atomic manner.

consume(a,priv) — atomically waits for a to be full, copies its value into the private variable priv, sets a empty and proceeds.

It is required to have synchronized access to two word variables considered as a unit. For the purposes of this problem, assume a representation by two element vectors a2[1:2], x2[1:2], and priv2[1:2].

(a) Using only produce and consume for synchronization, write procedures produce2(a2,x2) and consume2(a2,priv2) which treat a2 as a single synchronized unit.

(b) Does your solution extend to multiple word variables of more than two words? Why or why not?

Solution: 8.1 (a)

procedure produce2(a2,x2)
produce a[1] := x[1];
produce a[2] := x[2];
end procedure;

procedure consume2(a2,priv2)
consume a[2] into priv2[2];
consume a[1] into priv2[1];
end procedure;

Note: No other produce/consume should be performed on a[1] or a[2] outside these procedures.

(b) This solution can be extended to variables with any number of words, by consuming words in the reverse order from that in which they are produced.

Since produce2 always starts from the first word, no other instance of produce2 will be able to start after one is started. Similarly for consume2.
Since consume2 always starts from the last word, it will not be able to start until produce2 has finished producing all the words. Similarly, produce2 will not be able to start until consume2 has finished consuming all the words.

**Problem: 8.2** Assume a multiprocessor with \( P \) processes running on different processors. The set of processes is divided into \( K \) disjoint subsets, 0, 1, \ldots, \( K-1 \), by assigning a color to each process. For example, the process with index \( ME \) could be assigned color \( \text{color} \left( ME \right) = ME \mod K \).

Define a control oriented synchronization called a rainbow section which has an entry, `begin rsect`, and an exit, `end rsect`. It imposes the synchronization condition that no two processes of the same color can occupy a rainbow section simultaneously. Processes of different colors can overlap their executions of the section.

Implement rainbow sections for a shared memory multiprocessor having only unconditional critical sections for synchronization. That is, using only `begin critical` and `end critical` as synchronization operations, write pseudo code to implement `begin rsect` and `end rsect`. Comment well, and describe the purpose of any auxiliary variables introduced.

**Solution: 8.2**

```plaintext
shared boolean ColorCrit[0..K-1];
    /* Array of boolean flags that tell if a process of color x is inside a rainbow section.*/
    /* All flags should be initialized to false. */
private boolean CanProceed;

procedure begin_rsect
loop:
    critical
        if not ColorCrit[color(ME)] then {
            ColorCrit[color(ME)] := true;
            CanProceed := true;
        }
        else CanProceed:= false;
    end critical;
    if not CanProceed then goto loop;
end procedure;

procedure end_rsect
    ColorCrit[color(ME)] := false;
end procedure;
```

**Problem: 8.3** A \( k \)-restricted section is like a critical section except that, instead of only one process, no more than \( k \) processes can be inside the section simultaneously.

(a) Extend Amdahl’s law to give the execution time with \( N \) processors for a program containing \( S \) seconds of strictly sequential work, \( L \) seconds of perfectly parallelizable work, and a \( k \)-restricted section of length \( R \) seconds that must be executed by all processes.

(b) Show how to implement \( k \)-restricted sections using only critical section synchronization by writing MIMD pseudo code for the entry, `ksect`, and the exit, `end ksect`, statements of a \( k \)-restricted section. Specify shared and private variables and initial conditions.

**Solution: 8.3**

(a) Execution time = \( S + \frac{L}{N} + R \left\lceil \frac{N}{k} \right\rceil \).
shared integer nc;
    /* Counter for the current number of processors inside the k-restricted section. */
    /* Should be initially 0. */
private boolean CanProceed;

procedure begin_ksect
loop:   critical
    if (nc < k) then {
        nc := nc + 1;
        CanProceed := true;
    }
    else CanProceed := false;
end critical;
    if not CanProceed then goto loop;
end procedure;

procedure end_ksect
critical
    nc := nc -1;
end critical
end procedure;

Problem: 8.4 The IBM370 multiprocessors had a compare and swap synchronization instruction, which
did the following atomically with respect to all other accesses to Var:
CAS(old, new, Var, match)—If old=Var, set match:=true and Var:=new. Otherwise, replace old:=Var
and set match:=false.
Other synchronizations were implemented in terms of this assembly language primitive.
Show how to implement critical sections with pseudo code using CAS as the only primitive synchro-
nization operation.

Solution: 8.4 Assume that Var is initially 0. Writing Var in the assignment Var := 0 must be
atomic.
shared integer Var;
private integer old := 0, new := 1;
private boolean match;
procedure critical
loop:   CAS(old, new, Var, match);
    if not match then goto loop;
end procedure;

procedure end_critical
    Var := 0;
end procedure;

Problem: 8.5 Consider implementing a barrier synchronization on a machine that has only produce/con-
sume operations as its basic synchronization mechanism. For simplicity, assume that the barrier is
executed only once so that only one blocking variable, called outlock below, is needed. Ignore the
lock needed to prevent processes from executing the barrier a second time before all have completed
the first. Pseudocode for the barrier could be:
consume count into c
The problem is that a potentially large number of processes may be accessing count or reading outlock to test its state in competition with the one process which wants to write it. The solution to this problem is to divide up the competition and have each process synchronize with a limited number of others.

Let the processes be broken up into groups of size $G$. When all processes of a group have arrived at the barrier, a single process will report for that group to the next higher level group of $G$. $P$ processors will require $\log_G P$ levels of reporting.

Write MIMD pseudo code similar to the above to show how this solution for the barrier competition can be implemented. You may assume a language with recursive procedures, but use only the full/empty data mechanism for synchronization.

**Solution: 8.5** Let the number of processes be $P$ and let $k = \lceil \log_G P \rceil$. Then there will be $k$ levels to the algorithm, and at each level, $1 \leq i \leq k$, there will be $NG(i)$ groups, where $NG(1) = \lceil P/G \rceil$ and $NG(i) = \lceil (NG(i-1))/G \rceil$, $2 \leq i \leq k$. At any level, all groups but the last will have $G$ elements, and the last group will have $NG(i) \mod G$ if this number is nonzero, otherwise it will have $G$. The size of group $j$ at level $i$ is thus

$$
size(i, j) = G, \text{ if } j \neq NG(i) \text{ or } NG(i-1) \mod G = 0, \\
= NG(i-1) \mod G, \text{ if } j = NG(i) \text{ and } NG(i-1) \mod G = 0.
$$

Note that $k$, $NG(i)$, and $size(i, j)$ can be precomputed given $G$ and $P$.

The barrier is then a procedure called by all processes $1 \leq p \leq P$ with the call $\text{barrier}(1, \lfloor (p-1)/G \rfloor + 1)$. The two arguments of barrier are the level and group number. The shared array of arrays $\text{count}(\text{level, group})$ is initialized to zeros. The similarly shaped shared logical array $\text{outlock}(\text{level, group})$ has all elements initialized to empty.

```plaintext
shared count(1 \leq i \leq k, NG(i)), outlock(1 \leq i \leq k, NG(i));
procedure barrier(level, group)
begin
    private level, group, C;
    consume count(level, group) into C;
    C := C + 1;
    if (C \neq size(level, group))
        then produce count(level, group) := C;
    else if (level \neq k)
        then begin
            barrier(level+1, \lfloor (group-1)/G \rfloor + 1);
            produce outlock(level, group) := true;
        end;
    else produce outlock(level, group) := true;
waitfull outlock(level, group);
end
```

**Problem: 8.6** Consider the atomic fetch&add operation. $\text{fetch\&add}(x, e)$ will return the old value of $x$ and will replace its value with $x + e$ in one memory cycle. As an example, if the initial value of $x$
is one, then fetch&add(x, 2) will return a one and replace x with three in memory.

The following is an attempt to implement the semaphore operations P and V using the fetch&add operation. What is wrong with the implementation? show a scenario that demonstrates your answer clearly. Provide the necessary correction.

```plaintext
shared integer S;
private boolean OK;
S := 1;
OK := false;
procedure P(S)
    repeat
        begin
            if (fetch&add(S, -1) > 0) then OK := true;
            else
                begin
                    OK := false;
                    fetch&add(S, 1);
                end;
        end;
    until OK;
end procedure;
procedure V(S)
    fetch&add(S, 1);
end procedure;
```

Solution: 8.6

**Problem: 8.7** Suppose you are given a machine in which a memory word has three distinct atomic operations, read, write and an increment instruction, INC, that will in one memory cycle increment the integer in a word by one, and, if the word contains the maximum representable integer, the word is not altered but a testable overflow bit is set.

(a) Show how the lock and unlock synchronizations can be implemented using the INC instruction.

(b) How important is the fact that incrementing the largest integer does not change its value?

Solution: 8.7

**Additional Problem**

**Problem: 8.8** Consider a group of processes running on the Ultracomputer which will use fetch&add to sum a set of partial results. Each process will compute a part of the sum and add it in, but the final sum is not complete until all processes have added their contributions. Another process will wish to use the final sum, but may attempt to do so before it is complete.

(a) Using fetch&add as the only synchronizing mechanism, show how to sum the partial results and prevent the consumer from using the sum until it is complete. Write pseudo code for the P partial sum producers and the one consumer.

(b) Extend your solution to the case in which several sums will be computed, one after another. In this case, the consumer must clear the sum and notify the producers that they may begin adding partial sums for the next result.
Solution: 8.8  Producer $i$ has partial sum $S_i$ in private variable $sp$. Shared variable $S$ will get the sum. The number of producers is in the shared variable $P$ and the number which have contributed to the sum so far is in shared variable $D$.

(a) Initialize $D$ to zero.

producer:

procedure add(sp)
begin
  f&a(S, sp);
  f&a(D, 1);
end;

c consumer:

procedure get(sum)
begin
  while (f&a(D, 0) $\neq$ P) ;
  sum := f&a(S, 0);
end;

(b) producer: initialize lastsum := 0, thissum := 1

procedure add(sp)
private sp, lastsum;
shared P, S, D, thissum;
begin
  while (lastsum = thissum) ;
  lastsum := f&a(thissum, 0);
  f&a(S, sp);
  f&a(D, 1)
end;

c consumer:

procedure get(sum)
begin
  while (f&a(D, 0) $\neq$ P) ;
  sum := f&a(S, 0);
  f&a(S, -sum);
  f&a(D, -P);
  f&a(thissum, 1);
end;

This solution increments thissum indefinitely, but it only needs to be kept modulo 2. Note that because fetch&add does an integer sum, it is seldom used for numerical computation as in this problem. It would normally be used for synchronization of a floating point numeric computation.
Chapter 9: Solutions

Problem: 9.1 Write equation (9.2) in the form of equation (9.1) plus a granularity correction. Describe the behavior of the correction term as \( P \) varies for fixed \( N \).

Solution: 9.1

Problem: 9.2 Study the performance predicted by the static and dynamic models for two loop iteration execution times as a function of the ratio of the two times. Assume equal probabilities of execution of the two if branches. Give the asymptotic behavior for both extremes of the ratio of \( T_1 \) to \( T_2 \), and draw graphs similar to Figure 9-21 and Figure 9-22, but with the horizontal axis being the \( T_1/T_2 \) ratio.

Let the length, \( T_S \), of the scheduling critical section be 10% of the length of the short iteration, and let \( N = 1000 \) and \( P = 10 \). Normalize the execution time so that the graphs illustrate the behavior of the performance as a function of the \( T_1/T_2 \) ratio and not a variation in the total work of the loop.

Solution: 9.2 To prevent the total work of the loop from obscuring the curves, we take the expected iteration time to be one, \( \frac{T_1 + T_2}{2} = 1 \). Then if the ratio is \( R = \frac{T_1}{T_2} \), the loop execution and critical section times are, \( T_1 = \frac{2R}{1+R} \), \( T_2 = \frac{2}{1+R} \), and \( T_S = \frac{0.2}{1+R} \).

For the prescheduled loop with \( N = 1000 \), \( P = 10 \), and equal probability of long and short loops, the expected value of the maximum number of long loops assigned to one of the 10 processes is \( E(max_1) = 57.678 \). This was obtained from Matlab's incomplete beta function, but could also be computed by a short C program for Equation 9.51. The expected execution time for the prescheduled loop as a function of \( R \) is

\[
E(T_e) = E(max_1)^2 \frac{(R+1)}{(R-1)} + \left( \frac{N}{P} \right) \frac{2}{1+R}.
\]

The selfscheduled loop is not saturated because \( E(T_e)/T_2 = 10T_S > (P-1)T_S = 9T_S \). Thus for
the self scheduled loop $E(T_e) = \frac{N}{P} \left( 1 + \frac{0.2}{1 + R} + T_W \right)$, where the wait time is given by

$$T_W = \frac{1}{2} \left( \frac{\sqrt{\left( 1 - 9 \left( \frac{0.2}{1 + R} \right) \right)^2}}{1 + R} + 20 \left( \frac{0.2}{1 + R} \right)^2 - \left( 1 - 9 \left( \frac{0.2}{1 + R} \right) \right) \right).$$

Since $T_1$ is the longer loop body, the minimum of $R$ is one. The curves reach asymptotes as $R$ increases. The behavior of the prescheduled curve results from there being no difference between the long loop and average at $R = 1$ and the extra 7.68 $T_1$ loops in the maximum having a larger and larger effect as the differential between the two loop sizes increases. An asymptote is reached because the effect of the $T_2$ loop is negligible when $R$ is large. The behavior of the self scheduled loop is determined by the size of the scheduling critical section, $T_S$, which becomes smaller and smaller as $R$ increases. It might be better to make the time, $T_S$, 10% of the average loop execution instead of $T_2$. If this is done, the selfscheduled curve is a straight line from its $R = 1$ value.

**Problem: 9.3** Adapt the performance time model for a loop with a critical section developed for a pipelined MIMD computer in Section 9.2.2.2 to a true MIMD machine with $P$ processors. Plot a comparison graph of time versus $P$ for your model with the model of Section 9.2.2.2. Choose model parameters and graph range to clearly show the difference between the two models.

**Solution: 9.3**

**Problem: 9.4** Develop an execution time model for a loop with a critical section for an architecture having $K$ pipelined MIMD processors sharing memory. That is, adapt the execution time model of Section 9.2.2.2 to an architecture like that of Section 9.2.3 but having $K$ PEMs instead of four. Take
the hardware parallelism limit in each pipelined MIMD processor to be \( U \), and distribute \( P \) processes over \( K \) processors as evenly as possible.

**Solution: 9.4**

**Additional Problems**

**Problem: 9.5** Certain sequential algorithms involve doing \( N \) units of work of size \( \Delta t \) in sequence. When these algorithms are parallelized with \( P < N \) processes, each process can do \( N/P \) units of work sequentially, but then must combine the group results obtained with a tree of depth \( \log_2 P \). Assume one step of combining group results also takes \( \Delta t \).

(a) Ignoring the problem of keeping numbers integral, i.e. suppressing ceiling and floor functions, develop an execution time model as a formula for \( T(P) \) in terms of \( N, P, \) and \( \Delta t \), and compute the speedup and efficiency of this algorithm model compared to the simple \( N \) step sequential algorithm.

(b) Analyze the efficiency by determining how much bigger \( N \) must be than \( P \) to get at least 50% efficiency. Give values for \( N \) at several convenient values of \( P \) between 2 and 100.

**Solution: 9.5** (a) The problem did not specify a sequential section of code, so the execution time model is

\[
T(P) = \frac{N}{P} \Delta t + \Delta t \log_2 P.
\]

A single process does not need the base 2 tree, so \( T(1) = N \Delta t \). The speedup is then

\[
S(P) = \frac{N \Delta t}{(N \Delta t)/(P + \Delta t \log_2 P)} = \frac{P}{P \log_2 P + \Delta t \log_2 P}.
\]

The efficiency is

\[
E(P) = \frac{1}{P + \frac{\Delta t \log_2 P}{N}}.
\]

(b) To get greater than 50% efficiency, we must have \( N > P \log_2 P \).

<table>
<thead>
<tr>
<th>( P )</th>
<th>( N )</th>
<th>( P )</th>
<th>( N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>32</td>
<td>160</td>
</tr>
<tr>
<td>8</td>
<td>24</td>
<td>64</td>
<td>384</td>
</tr>
</tbody>
</table>

**Problem: 9.6** Consider a program having a strictly sequential amount of work, \( T_S \), an amount, \( T_P \), of perfectly parallelizable work, and a critical section time of \( t_c \) in each process. \( T_S, T_P \) and \( t_c \) are all expressed in units of time. In Section 8.2.2.2 of Chapter 8, we developed a model for the execution time \( T(P) \) under the assumption of best case timing on arrival at the critical section. Derive a formula for \( T(P) \) that models worst case arrival timing.

**Solution: 9.6** In the worst case, some process will have to wait for all \( P - 1 \) others to execute the critical section and still have all of its computation to do when it exits. The time would then be

\[
T(P) = T_S + P t_c + \frac{T_P}{P}.
\]

**Problem: 9.7** The OpenMP code below uses \( P \) threads to execute a block prescheduled parallel loop. The
management of the chunk size is shown explicitly rather than being handled by the system.

```
CALL OMP_SET_DYNAMIC(FALSE)
CALL OMP_SET_NESTED(FALSE)
CALL OMP_SET_NUM_THREADS(P)
CHUNK = N/P
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(ME, I)
  DO ME = 0, P-1
    DO I = ME*CHUNK+1, (ME+1)*CHUNK
      CALL COMPUTE(I)
    END DO
  END DO
!$OMP END PARALLEL DO
```

Assume that all execution times are negligible compared to that for forking a new thread, which takes \(a\) seconds, and the call to `COMPUTE(I)` takes \(b\) seconds for any \(I\). Assume processes are forked one at a time, so that setting up the parallel DO takes \(\alpha P\) seconds. Find the number \(P\) of processes that minimizes the total execution time for a fixed value of \(N\). Consider how the answer varies for different relations of thread creation time to compute time, say \(\alpha = \beta\), \(\alpha = 4\beta\), and \(\beta = 4\alpha\).

**Solution:**

The computation time in one process is \(\beta N/P\). Adding the time for forking threads, the total time becomes \(T(P) = \alpha P + \beta N/P\).

To minimize the time, we solve

\[
\frac{d}{dP} T(P) = \alpha - \frac{\beta N}{P^2} = 0 \quad \text{or} \quad P = \sqrt[\beta/\alpha]{N}.
\]

If thread creation takes the same time as a loop iteration, \(\beta = \alpha\), and the optimum value of \(P\) is \(\sqrt{N}\).

For \(\alpha = 4\beta\), thread creation is more expensive, and only half as many processes should be used. If \(\beta = 4\alpha\), threads are cheap, and \(P = 2\sqrt{N}\) is optimal.
Chapter 10: Solutions

Problem: 10.1 This program and its cache oriented version compute a pairing between numbers in one array and the nearest number in a second array. Use the techniques of Section 10.1 to analyze the advantage of the cache oriented version in terms of the locality metric. Assuming one word lines, what size cache is needed to get the advantage of the cache oriented version. The parameter $K$ in the cache oriented version divides $M$ evenly.

Naive version

```pascal
real x[1:M], y[1:M], c;
integer n[1:M], i, j, nn;
for i := 1 step 1 until M
begin
  c := maxreal;
nn := 0;
  for j := 1 step 1 until M
    if (|y[j] - x[i]| < c) then
      begin
        c := |y[j] - x[i]|;
nn := j;
      end
    n[i] := nn;
end;
```
Cache oriented version

```plaintext
real x[1:M], y[1:M], c[1:M];
integer n[1:M], i, j, jj;
for i := 1 step 1 until M
begin   c[i] := maxreal;
      n[i] := 0;
end;
for j := 1 step 1 until M/K
  for i := 1 step 1 until M
    for jj := 1 step 1 until K
      if (|y[j*K+jj] - x[i]| < c[i]) then
        begin
          c[i] := |y[j] - x[i]|;
          n[i] := j*K+jj;
        end;
```

Solution: 10.1
Problem: 10.2 Two processes on a distributed memory multiprocessor communicate by non-blocking send and blocking receive while executing the code below. Execution time for this program uses the following information. Arithmetic assignments take time $t_a$; send takes $t_s$ in the sender plus $t_L$ for message delivery to the receiver; and receive takes $t_s$ to prepare for the receive plus any waiting time for the message to arrive from the sender.

(a) Calculate the waiting time at each receive.

(b) Is there a program restructuring which will reduce the waiting time, and what are the minimal assumptions on $t_a$, $t_s$, and $t_L$ which guarantee a reduction?

Solution: 10.2
Problem: 10.3 Assume we have a global address space multiprocessor similar to the KSR-1 architecture presented in Section 10.2.2. Consider the following two program segments.

Segment 1

\[
\begin{align*}
\text{low} & := (\text{me} - 1) \times \frac{N}{P} + 1; \\
\text{high} & := \text{low} + \frac{N}{P} - 1; \\
\text{for } i & := \text{low} + 1 \text{ step 1 until high} \\
& \quad \text{v[i]} := \text{v[i]} + \text{v[i-1]}; \\
\text{sum[me]} & := \text{v[high]}; \\
& \quad \text{barrier}; \\
& \quad \text{start} := 0; \\
\text{for } j & := (\text{me} - 1) \text{ step -1 until 1} \\
& \quad \text{start} := \text{start} + \text{sum[j]}; \\
\text{for } i & := \text{low} \text{ step 1 until high} \\
& \quad \text{v[i]} := \text{start} + \text{v[i]}; \\
\end{align*}
\]

Segment 2

\[
\begin{align*}
\text{low} & := (\text{me} - 1) \times \frac{N}{P} + 1; \\
\text{high} & := \text{low} + \frac{N}{P} - 1; \\
\text{for } i & := \text{low} + 1 \text{ step 1 until high} \\
& \quad \text{v[i]} := \text{v[i]} + \text{v[i-1]}; \\
\text{sum[me]} & := \text{v[high]}; \\
& \quad \text{barrier}; \\
& \quad \text{start} := 0; \\
\text{for } j & := 1 \text{ step 1 until (me - 1)} \\
& \quad \text{start} := \text{start} + \text{sum[j]}; \\
\text{for } i & := \text{low} \text{ step 1 until high} \\
& \quad \text{v[i]} := \text{start} + \text{v[i]}; \\
\end{align*}
\]

The vector size, \(N\), and the number of processors, \(P\), are powers of two and \(N \gg P\). Assume the entire vector is initially in the memory of a processor other than the \(P\) processors doing the computation. Use the same memory hierarchy and access times as in Section 10.2.2, i.e. a local cache access takes 20 processor cycles, a remote access in the same ring takes 150 cycles, and an access to a different ring takes 570 cycles. Also assume processors execute the code at the same rate in lock-step.

(a) What operation is being performed by the two program segments?

(b) Draw a graph that shows the communication between the processors at different time steps for each program segment.

(c) What are the total number of communications needed in each implementation?

(d) Compare the efficiency and performance of the two implementation.

Solution: 10.3
Problem: 10.4 Given are a vector of \( N \) distinct integers, \( V[N] \), and \( P \) distinct integer values, \( \text{search}[P] \), where \( N \) and \( P \) are powers of two and \( N \gg P \). We want to search the vector, \( V \), for each of the values in \( \text{search}[P] \) and return the index of the value in \( V \). Assume a shared address space multiprocessor similar to the KSR-1 architecture presented in Section 10.2.2 and that processes execute the code at the same rate in lock-step. Assume the vector, \( V \), is initially in the memory of a processor other than the \( P \) processors doing the search and that every processor’s cache is large enough to hold the entire vector. For simplicity, we will ignore the search stop condition and assume each search value can definitely be found in \( V \). Consider two implementations.

**Implementation 1**

\[
\text{myvalue} := \text{search}[\text{me}]; \\
\text{index}[\text{me}] := 0; \\
\text{for } i := 1 \text{ step 1 until } N \\
\quad \text{if } (\text{myvalue} = V[i]) \text{ then } \text{index}[\text{me}] := i;
\]

**Implementation 2**

\[
\text{low} := (\text{me} - 1)*N/P + 1; \\
\text{high} := \text{low} + N/P - 1; \\
\text{for } j := 1 \text{ step 1 until } P \\
\quad \text{begin} \\
\quad \quad \text{myvalue} := \text{search}[j]; \\
\quad \quad \text{for } i := \text{low} \text{ step 1 until } \text{high} \\
\quad \quad \quad \text{if } (\text{myvalue} = V[i]) \text{ then } \text{index} := i; \\
\quad \text{end;}
\]

(a) Compare the performance and efficiency of the two implementations for the KSR-1 style machine.

(b) If the vector is stored in increasing order, can you implement a search with better overall performance than either of the above two implementations?

Solution: 10.4
Problem: 10.5 Consider the solution of a discretized partial differential equation where at each step every point on a two-dimensional grid indexed by \([i, j]\) is approximated as a function of itself and its four neighbors, \(f(A[i-1, j], A[i, j-1], A[i, j], A[i, j+1], A[i+1, j])\). The sequential pseudo code for the heart of the solver for an \(N \times N\) grid is given below.

\[
\text{alpha} := 0.1; \\
alldpoints := N \times N; \\
done := \text{false}; \\
\textbf{while} (\text{not } \text{done}) \textbf{do} \\
\hspace{1em} \textbf{begin} \\
\hspace{2em} \text{change} := 0.0; \\
\hspace{2em} i := 1 \textbf{ step } 1 \textbf{ until } N \\
\hspace{3em} j := 1 \textbf{ step } 1 \textbf{ until } N \\
\hspace{4em} \textbf{begin} \\
\hspace{5em} \text{oldval} := A[i, j]; \\
\hspace{5em} A[i, j] := \text{alpha} \times f(A[i-1, j], A[i, j-1], A[i, j], A[i, j+1], A[i+1, j]); \\
\hspace{5em} \text{change} := \text{change} + \text{abs}(A[i, j] - \text{oldval}); \\
\hspace{4em} \textbf{end}; \\
\hspace{2em} \textbf{if} (\text{change/allpoints} < \text{tolerance}) \textbf{ then } \text{done} := \text{true}; \\
\hspace{1em} \textbf{end}; \\
\]

(a) Assume a uniprocessor with a separate data cache memory, and describe how well the above algorithm exploits temporal locality.

(b) Describe a method to increase the temporal locality of the solver in the uniprocessor of part (a).

(c) Now consider a shared address space multiprocessor like the KSR-1 described in Section 10.2.2. Assume 64 processors, \(N = 1024\), and row major allocation for the two dimensional arrays. Discuss and compare locality and the amount of communication for the following two partitioning strategies:

i. Partition the grid into square blocks.

ii. Partition the grid into \(N/64\) contiguous rows per processor.

Solution: 10.5
Chapter 11: Solutions

**Problem: 11.1** In the discussion of RAID 4 on page 454, step 3 of the update process specifies updating the parity sector by “subtracting the old data and adding the new.” Give a precise mathematical description of this process.

**Solution: 11.1**
Problem: 11.2 Estimate the number of processors, \( P \), for which the parallel input format conversion of Section 11.4.1 becomes advantageous in spite of its extra overhead. Assume \( NX = NY = NZ = 100 \). You will need to make, and justify, several assumptions about execution times. In order to concentrate on the advantage of parallel formatting, it can be assumed that the actual file read operations take the same time for both the parallel and serial versions. Thus, you can assume that the number of elementary operations for equivalent \( f\text{scanf}() \), for direct serial file read, and \( s\text{scanf}() \), for in-memory format conversion, are the same and ignore the time for \( \text{stat}() \) and \( \text{fread}() \).

An elementary operation time should probably be some basic idea of a C assignment statement execution. In such units, a format scan time is probably proportional to the number of characters scanned. Judging from the \%.10f formats specified in the example output program, Program 11-2, floating point numbers probably require about 12 characters each. Make and justify the same style assumptions for the other operations used by the parallel input formatting to complete the estimate of \( P \).

Solution: 11.2
Problem: 11.3 Using similar techniques and assumptions to those of Problem 11.2, estimate the number of processors, $P$, for which the parallel output formatting of Section 11.4.2 becomes advantageous for $NX = NY = NZ = 100$.

Solution: 11.3
Problem: 11.4 Rewrite the distmult subroutine of Program 5-13 using the subarray data type constructed in Program 11-5 so that only $b \times b$ elements are transmitted in each broadcast instead of the 2500 elements comprising the full buffer.

Solution: 11.4