HOMEWORK VI

DUE: December 11, 2013

READ:
i) Related portions of Chapters 3, 5 and Appendix I of the Hennessy book
ii) Related portions of Chapters 2, 4, 5, 7, 8, 9 and 10 of the Jordan book

ASSIGNMENT: There are four problems.

Solve all homework and exam problems as shown in class and past exam solutions.

1) Develop a dot-product algorithm on two vectors, A and B. Store the result in “k” for a UMA MIMD system with “p” processors.

- Indicate the time complexity of your algorithm.
- Make observations relevant to the execution of your UMA MIMD algorithm, including the data decomposition, load balancing, synchronization, etc.

2) Develop a matrix multiply algorithm on two square matrices, A and B. Store the result in “C” for a UMA MIMD system with “p” processors.

- Indicate the time complexity of your algorithm.
- Make observations relevant to the execution of your UMA MIMD algorithm, including the data decomposition, load balancing, synchronization, etc.

3) Develop a dot-product algorithm on two vectors, A and B. Store the result in “k” for a 2-d square-mesh NORMA MIMD system with “p” processors.

- Indicate the time complexity of your algorithm.
- Make observations relevant to the execution of your NORMA MIMD algorithm, including the data decomposition, load balancing, the communication graph, etc.

4) Develop a dot-product SIMD algorithm on two vectors, “A” and “B” with “p” processing elements. Each one stores the result in “k” which is a local variable. Processing Element 0 computes the final value of “k.” Then, it writes to global “k” location which you will not show.

The SIMD has a 2-d square mesh interconnection network.
- Indicate the time complexity of your algorithm.
- Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc.
Q1) One of the most common scientific operations is the transposition of a matrix of size nxn. The operation makes all rows columns (or all columns rows). As an example, given the 3x3 matrix A as:

\[ A = \begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix} \]

Its transpose, \( A^T \), is obtained as:

\[ A^T = \begin{pmatrix}
1 & 4 & 7 \\
2 & 5 & 8 \\
3 & 6 & 9
\end{pmatrix} \]

Matrix transpose is performed often to implement the matrix multiply operation which requires that the second matrix is accessed in the column major order. By transposing the second matrix, both matrices are accessed in the row major order.

The parallel machine is a UMA MIMD machine with “p” processors with fetch-and-add type operations.

Write down the algorithm in pseudo-code for this UMA machine. Indicate the time complexity. Discuss applicable parallel processing issues, such as load balancing, synchronization, data mapping, etc.

Note that your algorithm obtains \( A^T \) and stores in the original matrix A (overwrites it). Assume that all divisions, logarithms, etc., generate integer numbers.

A1)

Transposed (UMA MIMD)

Constant : p, my_id

Global : A[0, 1, .., (n - 1)][0, 1, .., (n - 1)], m

Local : TARR[0, 1, .., ((n/p) - 1)][0, 1, .., (n - 1)], k, j, l , i

Begin

for k = 1 to (p -1) do

FORK Comp_transp(k)
endfor

m = n/p

Comp_transp : For all Pi where 0 <= i <= (p - 1) do

for j = 0 to (m - 1) do

k = (my_id * m) + j

for l = 0 to (n - 1) do

TARR[j][l] = A[k][l]
endfor

endfor

Barrier (p)

For j = 0 to (m - 1) do

k = (my_id * m) + j

for l = 0 to (n - 1) do

A[l][k] = TARR[j][l]
endfor
endfor

Join (p)

End

\[ O(n/p)*O(n) = O(n^2/p) \]

\[ O(n/p)*O(n) = O(n^2/p) \]
The time complexity is a function of the two major loops marked above, each moving elements \(O(n^2/p)\). Thus, the time complexity is \(O(n^2/p)\) or polynomial.

A **barrier** is used to synchronize the processors so they would overwrite the matrix after it is completely copied to local temporary arrays. There is no need to use locks, but the barrier is implemented by fetch-and-add operations.

Load balancing is sustained if processors execute the code almost simultaneously. That is, if a processor does **not** fall behind until completion, all processors are busy.

The data decomposition is NOT static. The initial A matrix decomposition is rowwise while the program output data (the range) is columnwise:

<table>
<thead>
<tr>
<th>The initial data decomposition of matrix A is rowwise:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m = n/p) rows for (P_0)</td>
</tr>
<tr>
<td>(m = n/p) rows for (P_1)</td>
</tr>
<tr>
<td>(m = n/p) rows for (P_2)</td>
</tr>
<tr>
<td>. . . . . . . . . . . . . . . . . . . . . . . . . . . . .</td>
</tr>
<tr>
<td>(m = n/p) rows for (P_{(p-1)})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The transposed A matrix (the range) is obtained columnwise:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m = n/p) columns for (P_0)</td>
</tr>
<tr>
<td>(m = n/p) columns for (P_1)</td>
</tr>
<tr>
<td>(m = n/p) columns for (P_2)</td>
</tr>
<tr>
<td>. . . . . . . . . . . . . . . . . . . . . . . . . . . . .</td>
</tr>
<tr>
<td>(m = n/p) columns for (P_{(p-1)})</td>
</tr>
</tbody>
</table>

**Q2)** Consider the following sequential algorithm, named Compare:

\[
k = 0 \\
\text{for} (i = 0 ; i < n ; i++) \\
\quad \text{if } A[i] < B[i] \text{ then } \{A[i] = B[i] ; k = k + 1 \}
\]

Develop the corresponding algorithm for a UMA MIMD computer with \(p\) processors.

If possible, try to overlap communication with computations. You can use the **Fetch & Add** during the computation, assuming an NYU Ultracomputer type system. Indicate the time complexity of your UMA algorithm.

Make observations relevant to the execution of the UMA algorithm, including the data decomposition, load balancing, the communication graph, etc.

**A2)** The algorithm is below.

The time complexity is \(O(n/p)\) due to the large loop time in the algorithm.

Load balancing is good during the large for loop (with the \(O(n/p)\) time complexity) and also during the Fetch & Add.
The data decomposition is static. Both vectors A and B are decomposed as follows:

Q3) Consider the following sequential algorithm:

```
For (j = 0 ; j < n ; j++)
  C[j] = 0 ;
For (r = 0 ; r < n ; r++)
Endfor
Endfor
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>[5 6]</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>[7 8]</td>
</tr>
<tr>
<td>(1 * 5) + (2 * 6)</td>
<td>5 + 12 = 17</td>
<td></td>
</tr>
<tr>
<td>(3 * 7) + (4 * 8)</td>
<td>21 + 32 = 53</td>
<td></td>
</tr>
</tbody>
</table>
It works on two \( n \times n \) matrices named \( A \) and \( B \) and generates an \( n \)-element vector named \( C \). An example with \( n = 2 \) and arbitrary values is also shown above.

Develop the corresponding \textit{cost-efficient} \( p \)-processor UMA MIMD algorithm, named Q3. \textbf{Explain} the time complexity of the UMA MIMD algorithm.

\textbf{Make} observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, synchronization, etc.

\textbf{A3)} The UMA algorithm is as follows:

\begin{verbatim}
Q3 (UMA MIMD)
Constant : p
Global : A[0,...,(n-1)][0,...,(n-1)]; B[0,...,(n-1)][0,...,(n-1)]; C[0,...,(n-1)]; n; s
Local : i; j; r
Begin
    ----
    s = n/p
    for k = 1 to (p - 1) do
        FORK Comp_Q3(k)
    endfor
    Comp_Q3 :
    For all \( P_i \) where \( 0 \leq i \leq (p-1) \) do
        for (\( j = 0 ; j < s ; j++ \))
            \( C[i*s + j] = 0 \)
        endfor
        for (\( r = 0 ; r \leq n - 1 ; r++ \))
            \( C[i*s+j] = C[i*s+j] + (A[i*s+j, i*s + r] + B[i*s+j, i*s + r]) \)
        endfor
    endfor
    Join (p)
End
\end{verbatim}

The parallel time complexity is \( O(n^2/p) \) since each processor executes the loop pointed by the two parallel lines \( n^2/p \) times. The cost : \( O(n^2/p) \times O(p) = O(n^2) \). The time complexity of the sequential algorithm is \( O(n^2) \) since the inner loop in the algorithm is executed “\( n \)” times for each iteration of the outer loop which is executed \( n \) times. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

There is no communication among the processors : an \textit{embarrassingly parallel} algorithm! There is \textit{no communication graph}! Load balancing is good since all processors are busy until the computation completes. Synchronization : None.

The data decompositions are below. The decompositions do \textbf{not} change during the execution:

\begin{center}
\begin{tabular}{c}
\hline
\textbf{C vector :} \\
\hline
\begin{tabular}{cccc}
\( P_0 \) & \( P_1 \) & \( P_2 \) & \ldots & \( P_{p-1} \) \\
\hline
\( n/p \) & \( n/p \) & \( n/p \) & \ldots & \( n/p \) \\
\end{tabular}
\end{tabular}
\end{center}

The initial data decomposition of matrices \( A \) and \( B \) is rowwise:

\begin{center}
\begin{tabular}{c}
\hline
\begin{tabular}{c}
\( m = n/p \) rows for \( P_0 \) \\
\hline
\( m = n/p \) rows for \( P_1 \) \\
\hline
\( m = n/p \) rows for \( P_2 \) \\
\hline
\ldots \\
\hline
\( m = n/p \) rows for \( P_{p-1} \) \\
\end{tabular}
\end{tabular}
\end{center}
**Q4)** Consider the following **sequential** algorithm where all vectors above have “n” elements each:

\[
\begin{align*}
&k = 0 \\
&\text{For } (j = 0 ; j < n ; j++) \\
&\quad \{ \text{ If } (A[j] > 0) \text{ then} \\
&\quad \quad \quad k = k + A[j] \} \\
&\quad \} \\
&\text{For } (j = 0 ; j < n ; j++) \\
\end{align*}
\]

**a)** What is the time complexity of the sequential algorithm? Explain.

**b)** Develop the corresponding **UMA MIMD** algorithm with “p” processors named Q4.

**Explain** the time complexity of the UMA MIMD algorithm. **Make** observations relevant to the execution of your UMA MIMD algorithm, including the data decomposition, load balancing, the communication graph, etc. Is your algorithm cost efficient? Explain.

**A4) a)** The **sequential** algorithm is given:

\[
\begin{align*}
&k = 0 \\
&\text{For } (j = 0 ; j < n ; j++) \\
&\quad \{ \text{ If } (A[j] > 0) \text{ then} \\
&\quad \quad \quad k = k + A[j] \} \\
&\quad \} \\
&\text{For } (j = 0 ; j < n ; j++) \\
\end{align*}
\]

**b)** The **UMA MIMD** algorithm is below.

The first loop and the second loop are executed in O(n/p) time. The last loop adds the partial sums, taking O(p) time. Therefore, the time complexity of the parallel algorithm is O(n/p). The Barrier is needed to make sure all A vector elements are computed before all A elements are used in the second loop.

The cost : O(n/p) * O(p) = O(n)

The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decompositions are as follows:

- **A in the first loop, B, C, D:**
  - A in the first loop:
    - P0 P1 P2 \ldots P_{p-1}
    - n/p n/p n/p \ldots n/p
  - B, C, D:

- **A in the second loop:**
  - A in the second loop:
    - P_{p-1} P_{p-2} P_{p-3} \ldots P_0
    - n/p n/p n/p \ldots n/p
Load balance is good in the first two loops. However, in the last loop, only one processor is busy in the critical section.

There is communication among the processors in the second loop and in the critical section:

Q4 (UMA MIMD)
Constant : p, my_id
Global : n ; s ; k ; lock_var ; A[0,...(n-1)] ; B[0,...(n-1)] ; C[0,...(n-1)] ; D[0,...(n-1)]
Local : Local_k ; j

Begin
k = 0
s = n/p
for j = 1 to (p -1) do
    FORK Comp_Q3(j)
endfor

Comp_Q3 : For all Pi where 0 <= i <= (p - 1) do
    Local_k = 0
    For (j = 0 ; j < s ; j++)
        If (A[my_id *s + j] > 0) then
        Endif
    Endfor
    Barrier (p)
    For (j = 0 ; j < s ; j++)
        C[my_id *s + j] = A[n - (my_id * s) - j - 1] * D[my_id *s + j]
    Endfor
    Lock (lock_var)
    k = k + Local_k
    Unlock (lock_var)
Endfor
Join (p)
End

The communication in the second loop is as follows: The communication graph in the last loop is a linear array:

0 ————  P_{p-1}

1 ————  P_{p-2}

......

Q5) A shared memory UMA computer uses an 8x8 Omega network. Discuss how the “chained directory protocol” studied in class can be implemented efficiently. That is caching shared writable elements does not slow down the system. Among the points you must discuss is how caches communicate with each other and with home directories fast. Would you rather prefer the full-map or the limited directory protocol in this UMA as opposed to the chained directory protocol?

A5) We will consider read misses and writes separately.
Read misses:

a) One read miss occurs for a block at a time: The cache with the read miss sends its own id number and the address of the missing block. This transmission takes \(O(\log n)\) time since the Omega network has \(\log n\) stages. The main memory creates a pointer in the home directory of the block needed and sends the missing block to the cache, taking again \(O(\log n)\) time steps. On receiving the block, the cache stores the block and writes “CT,” indicating it is the last one on the chain. Subsequent read miss requests from other caches on the same block are serviced as follows: for each new miss request received for the block, the main memory sends the block and the id number of the last cache node whose miss request on the block was serviced. The main memory updates its pointer of the block for the cache receiving the block now. The cache after receiving the block, creates a pointer pointing at the last cache node whose id number was received with the block.

b) Simultaneous read misses occur for a block at a time: The main memory performs the actions in part (a) in the order they are received. The process can be sped up if interconnection network switches are sophisticated such that they “combine” cache miss requests into one request: A message is created from several requests such that the new message contains the missing block address and the id numbers of the caches that created the misses. Eventually, the main memory receives just one request for up to eight cache misses on one block. The main memory responds to this by sending the block and the id numbers of the cache nodes. Individual switches then duplicate the block and send the blocks with id numbers toward the cache nodes which finally receive their missing block and the id number of the cache to which this block directory will point to. The cache which does not receive any cache id number, will write “CT” for the block. Meanwhile, the main memory will have the block pointer pointing at one of the caches.

Writes:

a) One write occurs for a block at a time: there are two possibilities. The first one is that the block is not dirty and the second one is that the block is dirty. In the first case, the main memory has to invalidate all copies of the block in caches. So, the main memory sends out an invalidate signal to the cache pointed by the home directory pointer. The signal is transferred from cache to cache along the chain. The cache with the “CT” pointer eventually sends back the invalidate signal to the memory, meaning all block copies have been invalidated. In the second case, there can be only one cache with the block and has written to it. It has to send the acknowledgement signal and the dirty block back to the main memory to update it. Then, the main memory sends the (missing) block to the cache (assuming it is the write-allocate technique used) which also signals that the cache can write to the block as it is the only owner of the block. The cache writes “CT” in the block directory. The main memory also appropriately adjusts the home directory pointer to this cache.

b) Simultaneous writes occur for a block at a time: this is something that needs to be avoided, since there will be a number of invalidation sequences, one for each write. The first invalidation for the first cache write is done as in part (a) above where the block may or may not be dirty. After, that write, the main memory sends an invalidation signal to that cache for that block, without any chain traversal since there is only one copy of the block among the caches. After receiving the acknowledgment and the dirty block from that cache, the main memory sends the block to the second cache that wants to write, after which the main memory sends an invalidate signal to the second cache and so on.

This process has to be sped up. One way is by combining write requests through the interconnection network so that the main memory receives only one write request for a block with the id numbers of the caches that want to write. This would reduce the number of messages in the network. Still, the main memory has to send out a separate invalidate signal for each write.

The chain traversal for invalidation and chain forming require that caches communicate with each other. This is not possible with Omega network however, unless, the switches are sophisticated enough to route a message from one cache to another, with care taken not to cause deadlocks.

As a special case, the Full-Map Directory or the Limited directory method can be used instead of the Chained Directory since there are only eight nodes and if the interconnection network allows broadcasting from right to left (from the memory side to the processor side).
Q6) Consider the following sequential algorithm:

\[
\text{For } (j = 0 \text{ ; } j < n \text{ ; } j++) \\
{ \text{ If } j < 4 \text{ then } A[j] = B[j]*C[j] \\
\quad \text{ else } A[j] = B[j]*A[j \mod 4] } \\
\text{For } (j = 0 \text{ ; } j < n \text{ ; } j++) \\
\quad D[j] = E[j] + F[j]
\]

All vectors above have “n” elements each.

a) What is the time complexity of the sequential algorithm? Explain.

b) Develop the corresponding UMA MIMD algorithm with “p” processors, named QUMA.

Explain the time complexity of the algorithm. Is your algorithm cost efficient? Explain.

Make observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, etc.

A6) a) The sequential algorithm is given:

\[
\text{For } (j = 0 \text{ ; } j < n \text{ ; } j++) \quad \quad \quad \quad \quad \quad \text{O(n)} \\
{ \text{ If } j < 4 \text{ then } A[j] = B[j]*C[j] \\
\quad \text{ else } A[j] = B[j]*A[j \mod 4] } \\
\text{For } (j = 0 \text{ ; } j < n \text{ ; } j++) \quad \quad \quad \quad \text{O(n)} \\
\quad D[j] = E[j] + F[j]
\]

For “n” elements, the two loops are executed O(n) times. Therefore, the sequential time complexity is O(n), linear time complexity.

b) The UMA MIMD algorithm is below.

Both loops are executed in O(n/p) time. Therefore, the time complexity of the parallel algorithm is O(n/p).

The cost: O(n/p) * O(p) = O(n)

Barrier is needed to make no processor starts working on vector before processor 0 finishes it work on vector A.

The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.
The data decompositions are **not** static for all the vectors:

**Q7)** Consider the following *sequential* algorithm below. All vectors above have “$n$” elements each.

```plaintext
For (j = 0 ; j < n ; j++)

For (j = 0 ; j < n ; j++)
    { 
    }

For (j = 0 ; j < n ; j++)
```
a) What is the time complexity of the sequential algorithm? Explain.
b) Develop the corresponding UMA MIMD algorithm with "p" processors, named Q7. Assume that there are even number of processors. Explain the time complexity of the algorithm. Is your algorithm cost efficient? Explain. Make observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, etc.

A7)

a) The sequential algorithm is given:

\[
\begin{align*}
&\text{For } (j = 0 ; j < n ; j++) \\
&\quad A[j] = B[j] + C[j] & \text{(O(n))} \\
&\quad \{ \\
&\quad \quad \text{If } B[j] \neq 0 \text{ then } D[j] = A[n - 1 - j] + (A[j] / E[j]) \\
&\quad \quad E[j] = B[j] - F[j] \\
&\quad \} \\
&\text{For } (j = 0 ; j < n ; j++) \\
\end{align*}
\]

\(O(n)\)

For "n" elements, the three loops are executed \(O(n)\) times. Therefore, the sequential time complexity is \(O(n)\), linear time complexity.

b) The UMA MIMD algorithm is as follows:

Q7 (UMA MIMD)

Constant : p ; my_id

Global : n ; s ; A[0,..., (n-1)] ; B[0,..., (n-1)] ; C[0,..., (n-1)] ; D[0,..., (n-1)] ; E[0,..., (n-1)] ; F[0,..., (n-1)] ; G[0,..., (n-1)] ; H[0,..., (n-1)] ; J[0,..., (n-1)] ; K[0,..., (p-1)]

Local : i ; j

Begin

\[s = \frac{n}{p}\]

for k = 1 to (p - 1) do

FORK Comp_Q2k)

endfor

Comp_Q2 : For all Pi where 0 \(\leq\) i \(\leq\) (p - 1) do

K[my_id] = 0

for (j = 0 ; j < s ; j++) do

\[A[(\text{my}_\text{id}*s) + j] = B[(\text{my}_\text{id}*s) + j] + C[(\text{my}_\text{id}*s) + j]\]

endfor

K[my_id] = 1

for (j = 0 ; j < s ; j++) do

\[J[(\text{my}_\text{id}*s) + j] = A[(\text{my}_\text{id}*s) + j] / E[(\text{my}_\text{id}*s) + j]
\]

\[G[(\text{my}_\text{id}*s) + j] = A[(\text{my}_\text{id}*s) + j] + H[(\text{my}_\text{id}*s) + j]\]

endfor

for (j = 0 ; j < s ; j++) do

\[E[(\text{my}_\text{id}*s) + j] = B[(\text{my}_\text{id}*s) + j] - F[(\text{my}_\text{id}*s) + j]\]

endfor

While K[p - 1 - my_id] = 0 \{ \}

for (j = 0 ; j < s ; j++) do

If (B[(\text{my}_\text{id} * s) + j] \neq 0 then

\[D[(\text{my}_\text{id}*s) + j] = A[(n - 1) - (\text{my}_\text{id} * s) + j] + J[(\text{my}_\text{id}*s) + j]\]

Endif

endfor

Endfor

Join (p)

End
New values of vector A are needed by other vectors. There is also a false dependency on E. Therefore, two new vectors are used: Vector J with “n” elements and vector K with “p” elements. Vector J contains temporary values of A/E so that vector E is computed early. Vector K has all its elements 0 in the beginning and each element is set to 1 when the corresponding processor completes its work on A. Each processor checks if the portion of vector A it needs is ready. That is, the corresponding bit of K is set to 1. If yes, it uses those new elements of A. One can use a Barrier instead of vector K. However, each processor waits for only one other processor, not all the processors and so using vector K works. Finally, processors do considerable work (calculating A/E, G and E) to spend time until new A is calculated. That is, we overlap computations with communication.

All loops except the while loop are executed in O(n/p) time. The while loop is also O(n/p) in the worst case. However, due to overlapping of computations with communication, it may be very short. Even if it is O(n/p), it does not change the time complexity of the parallel algorithm which is O(n/p).

The cost: O(n/p) * O(p) = O(n). The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decompositions are not static for all the vectors:

Load balance is good in the computation loops. There can be idle time to wait for new vector A for some processors depending on their speed. Otherwise, all the processors are busy all the time.

There is communication in the last loop. The communication graph is as follows:

\[
0 \quad \circ \quad p-1 \\
1 \quad \circ \quad p-2 \\
2 \quad \circ \quad p-3 \\
\ldots \\
(p/2)-1 \quad \circ \quad p/2
\]

Q8) Consider the sequential algorithm below. All vectors above have “n” elements each, which means “D” has “n” elements from 0 to “n-1”.

a) What is the time complexity of the sequential algorithm? Explain.

b) Develop the corresponding UMA MIMD algorithm with “p” processors, named Q8.

Explain the time complexity of the algorithm. Is your algorithm cost efficient? Explain.
For (j = 0 ; j < n ; j++)
{  
  A[j] = (B[j] + C[j]) modulo n
  D[A[j]] = E[j] - F[j]
}

For (j = 0 ; j < n ; j++)
{  
  For (k = 0 ; k < n ; k++)
    M[j] = M[j] + (G[j] * N[k])
}

Make observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, etc.

A8) a) The sequential algorithm is given:

\[
\text{For (j = 0 ; j < n ; j++)} \\
\{ \\
  A[j] = (B[j] + C[j]) \text{ modulo } n \\
\} \\
\]

\[
\text{O(n)} \quad \text{O(n^2)} \\
\]

For “n” elements, the first loop takes O(n) times and the nested loops take O(n^2) times. For large “n” O(n^2) dominates. Therefore, the sequential time complexity is O(n^2). Polynomial time complexity.

b) The UMA MIMD algorithm is given below.

New values of vector A are needed by vector D. Each element of A is used as an index to access an element of D. In the best case, each A element is different, pointing at a distinct D element. Not only that A[j] = j so that a processor that computes A[j] can immediately compute D[j]. A worse situation is that all vector elements are distinct but A[j] generated by a processor results in an access to a D element in the domain of another processor. That means D is accessed completely by all the processors. For example, processor 0 works on A[6] and generates 157, i.e. A[6]=157. Assuming that each processor has 128 elements each, then processor 0 has to access element “157” of vector D to write to D[157] which is in the domain of processor 1. In the worst case, all vector A elements have the same value and so only one vector D element has to be computed. In this case, all “p” processors compute the same value for any A[j]. For example, if all A[j] elements are 204, then only D[204] will be written a new value and up to “n” times. Only the last value calculated by processor “p-1” will stay, all other values written to D[204] will be overwritten.

Anticipating these worst case scenarios, we decide to compute vector A and D elements by using one processor in O(n) time. This makes sense since the sequential time complexity is O(n^2) determined by a separate set of two nested loops that is completely parallel. Thus, while we use one processor to compute vector A and D elements, we use the remaining “p-1” processors to compute the nested loops in O(n^2/p) time. In summary, the nested loops section of the code takes O(n^2/p) time. For large “n”, it is larger than O(n) and so the parallel time complexity is O(n^2/p).
The cost: $O(n^2/p) \times O(p) = O(n^2)$

The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decompositions are **not** static for all the vectors:

- **B, G, H, M**
  - $P_1$:
    - $n/(p-1)$
  - $P_2$:
    - $n/(p-1)$
  - ...  
  - $P_{p-1}$:
    - $n/(p-1)$

- **A, B, C, D, E, F**
  - $P_0$:
    - $n$

Load balance is good in all the loops. However, Processor 0 may complete before the remaining processors. A more refined algorithm may allow Processor 0 to do the leftover work of the nested loops after it finishes the work on vectors A and D. That is vectors B, G, H and M are partitioned such that processor 0 is assigned some of the elements of these vectors plus all the elements of vector N to compute vectors G and M.

There is no communication in the algorithm! It is embarrassingly parallel.
Q9) Consider again the sequential algorithm in Question Q8. Develop the corresponding NORMA algorithm with “p” processors named Q9. The MIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “msg_recvd” in the algorithm. When a message arrives, the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement.

Explain the time complexity of the algorithm. Is your algorithm cost efficient? Explain.

Make observations relevant to the execution of your algorithm, including the data decomposition, load balancing, the communication graph, etc.

A9) The NORMA algorithm is as follows:

Q9 (2-D MESH NORMA)
Local : myid, s ; i ; j ; local_A[0, 1, .., (n-1)] ; local_B[0, 1, .., (n-1)] ; local_C[0, 1, .., (n-1)] ; local_D[0, 1, .., (n-1)] ; local_E[0, 1, .., (n-1)] ; local_F[0, 1, .., (n-1)] ; local_G[0, 1, .., ((n/p) - 1)] ; local_H[0, 1, .., ((n/p) - 1)] ; local_M[0, 1, .., ((n/p) - 1)] ; N[0, 1, .., (n-1)]

Begin
for all Pi where 0 < i < (p - 1) do
....
  s = n/(p-1)
  If (my_id = 0 then
    for (j = 0 ; j < s ; j++) do
      A[j] = (B[j] + C[j]) modulo n
      D[A[j]] = E[j] - F[j]
    endfor
  Else
    for (j = 0 ; j < s ; j++) do
      G[(my_id * s) + j] = B[(my_id * s) + j] * H[(my_id * s) + j]
      for (k = 0 ; k < n ; k++) do
        M[(my_id * s) + j] = M[(my_id * s) + j] + (G[(my_id * s) + j] * N[k])
      Endfor
    Endfor
  Endif
Endfor
End

The thinking about the NORMA algorithm is similar to the UMA MIMD. In the worst case there may be a large number of messages sent to one processor, causing long communication times. Thus, we assign computing vectors A and D to processor 0 and computing vectors G and M to the remaining “p-1” processors. The result is that there is no communication and so the parallel algorithm is embarrassingly parallel!

Similar to the UMA MIMD case, the parallel algorithm takes O(n^2/p) time for large “n” values.

The cost : O(n^2/p) * O(p) = O(n^2/p)

The cost is equal to the sequential time complexity. Therefore, the algorithm is cost efficient.

The data decompositions are not static for all the vectors and the same as the UMA MIMD case:
One peculiar point is the declaration of vector B for the processors. Vector B is completely used by processor 0. But, it is decomposed to “p-1” processors for the nested loops. How can one declare vector B so that the compiler and operating system know how it is used? We leave it the actual parallel programming language, compiler and operating system use. If there is no help from them, then one has to write two parallel programs: One for Processor 0 and one for all the remaining processors. This means that there is no SPMD style of programming then!

Load balance is good in all the loops as it is the case with the UMA MIMD algorithm. Again, Processor 0 may complete before the remaining processors and so a slightly modified algorithm may allow Processor 0 to do the leftover work of the nested loops after it finishes the work on vectors A and D as explained above.

There is no communication in the parallel algorithm. It is embarrassingly parallel!

Q10) Consider again the sequential algorithm in Question Q6 above. Develop the corresponding NORMA algorithm with “p” processors named QNORMA. The MIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “msg_recvd” in the algorithm. When a message arrives, the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement.

Explain the time complexity of the algorithm. Is your algorithm cost efficient? Explain.

Make observations relevant to the execution of your algorithm, including the data decomposition, load balancing, etc.

A10) The NORMA algorithm is below. The algorithm is similar to the UMA MIMD one. The two loops are executed in $O(n/p)$ time. However, in the second loop, all processors except processor 0 may wait for $A[0:3]$ if necessary, taking $O(p)$ time. Therefore, the time complexity of the parallel algorithm is $O(n/p)$.

The cost: $O(n/p) * O(p) = O(n)$. The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decompositions are not static for all the vectors:

Load balance is good in both loops, except that some processors may wait for messages to arrive depending on where
QNORMA: (2-D MESH NORMA)

Local: myid, s; i; j; local_A[0, 1; ..., ((n/p) - 1)]; local_B[0, 1; ..., ((n/p) - 1)]; local_C[0, 1; ..., ((n/p) - 1)];
local_D[0, 1; ..., ((n/p) - 1)]; local_E[0, 1; ..., ((n/p) - 1)]; local_F[0, 1; ..., ((n/p) - 1)]; TEMP[0, 1, 2, 3];
mesnum; my_row; my_column; south; east; west; north;

Begin

for all Pi where 0 <= i <= (p - 1) do

    s = n/p
    If my_id > 0 then mesnum = 0
    For (j = 0; j < s; j++)
        If my_id = 0 & j < 4 then
            local_A[j] = local_B[j] * local_C[j]
        Else If myid = 0 and j = 4
            Send (local_A[0:3], east)
            Send (local_A[0:3], south)
        Else If my_id > 0 & mesnum = 0 then
            If msg_recv = 1 then
                receive(TEMP)
                If my_column ≠ (l-1) then send (TEMP[0:3], east)
                If my_row ≠ (l-1) then send (TEMP[0:3], south)
                mesnum = 1
            Endif
        Endif
        local_D[j] = local_E[j] + local_F[j]
    Endfor

    For (j = 0; j < s; j++)
        If (my_id = 0 and j > 3) then
            local_A[j] = local_B[j] * local_A[j(modulo4)]
        Else
            While msg_recv = 0 & mesnum = 0{} \ O(p)
                receive(TEMP)
                If my_column ≠ (l-1) then send (TEMP[0:3], east)
                If my_row ≠ (l-1) then send (TEMP[0:3], south)
                mesnum = 1
            Endwhile
            local_A[j] = local_B[j] * TEMP[((i*s) + j)modulo4]
        Endif
    Endfor

End

they are located on the network and the network speed. Otherwise, all the processors are busy all the time.
There is communication among the processors and the communication graph is as follows:

Q11) Consider again the sequential algorithm in Question Q7 above. Develop the corresponding NORMA algorithm with “p” processors named Q11. The MIMD has a 2-d square mesh interconnection network. Again, assume that there are even number of processors.

Assume that the arrival of a message can be checked by testing a flag such as “msg_recvd” in the algorithm. When a message arrives, the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement.

Explain the time complexity of the algorithm. Is your algorithm cost efficient? Explain.

Make observations relevant to the execution of your algorithm, including the data decomposition, load balancing, the communication graph, etc.

A11) The NORMA algorithm is below.

The algorithm is similar to the UMA MIMD one. But, first, we do not need vector K. Second, unlike the UMA case processors generating new A vector must take charge and check to see if they have to send their new A element to the depending processors as soon as the new A element is calculated. A new A element is used if a certain B element is not zero. Thus, to determine if the new value of A has to be sent, each processor has a new vector with “n/p” elements to keep those elements of B checked by the consuming processor.

In order to make sure that we have the right order of A elements received, each processor sends not only the new A element calculated, but also its position in the local A vector. When received, the two values are stored in a 2-element vector TEMP. We make sure that we receive all the new A values and then use them. Thus, we will have a counter variable, “mesnum.” Each processor has a new vector with “n/p” elements (Opp_A) to keep the received values of new vector A. Again, the A/E, G and E are calculated early to overlap computations with communication. Similar to the UMA MIMD case all computation loops are executed in O(n/p) time. The while loop is also O(n/p) in the worst case. However, due to overlapping of computations with communication, it may be again very short. Even if it is O(n/p), it does not change the time complexity of the parallel algorithm which is O(n/p).

The cost: O(n/p) * O(p) = O(n)

The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

Load balance is good in the computation loops. There can be idle time to wait for new vector A for some processors depending on their speed. Otherwise, all the processors are busy all the time.
Q11 (2-D MESH NORMA)

Local : myid, s ; i ; j ; local_A[0, 1, ..., ((n/p) - 1)] ; local_B[0, 1, ..., ((n/p) - 1)] ; local_C[0, 1, ..., ((n/p) - 1)] ;
local_D[0, 1, ..., ((n/p) - 1)] ; local_E[0, 1, ..., ((n/p) - 1)] ; local_F[0, 1, ..., ((n/p) - 1)] ;
local_G[0, 1, ..., ((n/p) - 1)] ; local_H[0, 1, ..., ((n/p) - 1)] ; local_J[0, 1, ..., ((n/p) - 1)] ; local_Opp_A[0, 1, ..., ((n/p) - 1)] ;
local_Opp_B[0, 1, ..., ((n/p) - 1)] ; TEMP[0, 1] ; mesnum

Begin

for all Pi where 0 < i < (p - 1) do
    .... s = n/p
    mesnum = 0
    For (j = 0 ; j < s ; j++)
        If msg_recv = 1 then
            Receive (TEMP)
            local_Opp_A[TEMP[1]] = TEMP[0]
            mesnum = mesnum + 1
        Endif
        local_A[j] = local_B[j] * local_C[j]
        If (local_Opp_B[s - 1 - j] = 0 then
            Send (local_A[j], j, (p - my_id - 1))
        Endif
    Endfor

    For (j = 0 ; j < s ; j++)
        If msg_recv = 1 then
            Receive (TEMP)
            local_Opp_A[TEMP[1]] = TEMP[0]
            mesnum = mesnum + 1
        Endif
    Endfor

    For (j = 0 ; j < s ; j++)
        If msg_recv = 1 then
            Receive (TEMP)
            local_Opp_A[TEMP[1]] = TEMP[0]
            mesnum = mesnum + 1
        Endif
    Endfor

    While mesnum < s
        If msg_recv = 1 then
            Receive (TEMP)
            local_Opp_A[TEMP[1]] = TEMP[0]
            mesnum = mesnum + 1
        Endif
        Endwhile

    For (j = 0 ; j < s ; j++)
        If(local_B[j] = 0 then
        Endif
    Endfor

End
The data decompositions are **not** static for all the vectors:

![Data Decompositions Diagram]

There is communication in the last loop. The communication graph is below.

![Communication Graph]

The mapping of the communication graph to interconnection network does **not** result in a dilation-1 embedding. For example, processor 0 and processor (p-1) communicate by going through multiple links as shown on the left for a 4x4 mesh network.

This can be improved by using a torus network.

**Q12)** Develop an algorithm, called “NORMALIZE” for a 2-d square Mesh SIMD computer with “p” processing elements. The corresponding sequential NORMALIZE algorithm is as follows:

```plaintext
for (i = 0 ; i < n ; i = i + 1)
    {  if (i modulo 2 = 0 then A[i] = (A[i] + A[i - 1])/cons1
        else A[i] = (A[i])/cons1 
    }
```

The variable “**cons1**” is a constant and “**n**” is an even number. When “I” is zero, “i - 1” means “**n - 1**.” A node can send a value to another node by directly specifying the id number of the destination node:

```
send (25, var_d)
```

A node whose id number is contained in “myid” can send a value to another node the following way also:

```
send ((myid + 2), tmp_var)
```

- **Indicate the time complexity of your SIMD algorithm.**
- **Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc.**

**A12)** The code is shown below. We assume that variable “**s**” is even and so “**s - 1**” is odd.
The initial data decomposition is as follows:

During the computation, nodes whose id numbers differ by one communicate in one direction to send a single data element, forming a **ring**:

Therefore the communication graph is a **ring**! Any static direct interconnection network, other than the linear, tree, fat tree, perfect shuffle and star would be acceptable for the algorithm.

Since we have to use a square 2-D mesh, we have to consider how to embed the ring in the physical network. In parallel processing, there are the concepts of virtual and physical processor id numbers. A programmer would specify virtual processor id numbers in parallel programs. The mapping of virtual processor id numbers to physical processor id numbers must be done in the beginning of the computation. Systems typically have library programs to embed a regular graph to the physical interconnection network and also to map from virtual id numbers to physical id numbers. In the SIMD algorithm above, we use a system call before the “s = n/p” statement to embed the ring in the 2-d square mesh network with “p” nodes.

For the above ring graph, the embedding in the MESH for a 16-processor case is shown below. The ring is embedded **dilation-1** in the communication network, so that near neighbor communication on the communication graph is near
neighbor communication on the interconnection network.

There is a large number of elements per processor so that computations take a long time and can overlap with communication.

Load balancing is **good** since communication overlaps computation: every processor first sends the data element needed by the next processor on the ring then starts computing on its own elements. By the time a processor completes its computations on \(((n/p) - 1)\) elements, the element sent by its ring neighbor is received. Processors complete their execution by working on the received element.

The time complexity : \(O(n/p)\). This holds if computation overlaps with communication (as presumed above) or takes longer than the communication time, as the bulk of the work is done on \((n/p)\) local vector elements.

**Q13)** Consider the following **sequential** algorithm :

\[
k = 0 \\
For (j = 0 ; j < n ; j++)
{ A[j] = B[j] + C[j] ; /* S1 */
  k = k + A[j] + C[j] ; /* S3 */
}
\]

Vectors A, B, C, D, E and F have “\(n\)” elements each.

Develop the corresponding **SIMD** algorithm with “\(p\)” processing elements named Q13. The SIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “\texttt{msg\_recv}\_d” in the algorithm. When a message arrives the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement as done in class.

Indicate the time complexity of your algorithm. Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc. Is your algorithm cost efficient? Explain.
A13) The SIMD algorithm is below.

Q13 (2-D MESH SIMD)
Local : myid, s ; i ; j ; A[0, 1, .., ((n/p) - 1)] ; B[0, 1, .., ((n/p) - 1)] ; C[0, 1, .., ((n/p) - 1)] ; D[0, 1, .., ((n/p) - 1)] ; E[0, 1, .., ((n/p) - 1)] ; F[0, 1, .., ((n/p) - 1)] ; Local_k ; tmp ; i ; j ; s ; my_row ; my_column

Begin

for all P_i where 0 <= i <= (p - 1) do

s = n/p
For (j = 1 ; j < s ; j++)

\[ A[i*s + j] = B[i*s + j] + C[i*s + j] \]
\[ C[i*s + j] = A[i*s + j] / D[i*s + j] - (\log_{10}(E[i*s + j] * F[i*s + j])) \]
Local_k = Local_k + A[i*s + j] + C[i*s + j]
Endfor

If my_column = 1-1 then
If msg_sent = 0 then send(local_k, west)
Else
receive(tmp)
local_k = local_k + tmp
If my_column ≠ 0
send(local_k, west)
Endif
If my_column = 0
If my_row = l-1 then
send(local_k, north)
Else
receive(tmp)
local_k = local_k + tmp
If my_row ≠ 0 then
send(local_k, north)
Endif
Endif
Endif
If i = 0 then k = local_k
Endfor

End

The parallel time complexity is \( O(n/p) \), even though the communication time in the \( O(p) \) can be considerable. The algorithm cannot overlap computations and communication because of lack of loop-level parallelism. But, since addition is associative, we extract parallelism in a tree like fashion, with reduced parallelism as we move up the tree.

The cost : \( O(n/p) \times O(p) = O(n) \). The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decomposition is as follows:

The decompositions do not change during the execution:

<table>
<thead>
<tr>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>...</th>
<th>P_{p-1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/p</td>
<td>n/p</td>
<td>n/p</td>
<td>n/p</td>
<td>n/p</td>
</tr>
</tbody>
</table>

Local A, B, C, D, E and F vectors
Load balance is good in the first loop. But, during communication the load balance gets worse.

**Q14)** Consider the sequential algorithm below:

```
For (j = 0 ; j < n ; j++)
    A[j] = (B[j] + D[j]) ;
    E[j] = 1 / F[j] ;
Endfor

C = 0 ;
For (i = 0 ; i < n ; i++)
    C = C + A[i] ;
Endfor
```

Develop the corresponding **SIMD** algorithm with "p" processing elements named SIMDABCDEF. The SIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “msg_recvd” in the algorithm. When a message arrives the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement.

Indicate the time complexity of your algorithm. Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc. Is your algorithm cost efficient? Explain.

**A14)** The SIMD algorithm is shown below.

The parallel time complexity is O(n/p) + O(p) where the first component is due to computations and the second one is due to communication. The above algorithm overlaps computations and communication so that its processing elements perform computations while they also wait for messages to arrive. For large “n”, then the time complexity is...
SIMD ABCDEF (2-D MESH SIMD)

Local: myid, s, i, j; local_A[0, 1, .., ((n/p) - 1)]; local_B[0, 1, .., ((n/p) - 1)]; local_D[0, 1, .., ((n/p) - 1)]; local_E[0, 1, .., ((n/p) - 1)];
local_F[0, 1, .., ((n/p) - 1)]; west, north, tmp; i, j, s, k, l; local_sum; my_column; my_row; recvnum

Begin
for all Pi where 0 <= i <= (p - 1) do
    s = n/p
    l = recvnum = 0
    For (j=0; j<s; j++)
        local_sum = 0
        For (j=0; j<s; j++)
            local_sum = local_sum + local_A[j]
        Endfor
        If my_column = l - 1 and j = 0 then
            send(local_sum, west)
        Else
            If (msg_recvd)
                receive(tmp)
                local_sum = local_sum + tmp
                If my_column  0
                    send(local_sum, west)
                    recvnum = 2
                Else
                    recvnum = recvnum + 1
                    If my_row = l-1 then
                        send(local_sum, north)
                        recvnum = 2
                    Else
                        recvnum = recvnum + 1
                        If my_row  0 then
                            send(local_sum, north)
                        Endif
                    Endif
            Endif
        Endif
    Endfor
    If recvnum  2 then
        receive(tmp)
        local_sum = local_sum + tmp
        If my_column  0
            send(local_sum, west)
        Else
            If my_row = l-1 then
                send(local_sum, north)
            Else
                receive(tmp)
                If my_row  0 then
                    send(local_sum, north)
                Endif
            Endif
        Endif
    Endif
Endfor

If i = 0 then c = local_sum
Endfor
End
O(n). Then, the cost : O(n/p) * O(p) = O(n)

The data decompositions are as follows:

The decompositions do not change during the execution:

<table>
<thead>
<tr>
<th>Local A, B, D, E and F vectors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
</tr>
<tr>
<td>n/p</td>
</tr>
</tbody>
</table>

There is no communication among the processors in the beginning then they communicate in a tree like fashion:

There is communication on the bold links.

The direction of communication and so the structure of the tree is shown by the arrows:

Load balancing is good in the beginning since all processing elements are busy until the computation completes. During the communication sometimes the processing elements are idle.

Q15) Consider the following sequential algorithm:

```c
For (j = 1 ; j < n ; j++)
{
    K[j] = K[j] + K[j - 1];     /* S1 */
    D[j] = E[j] * B[j] - C[j];  /* S3 */
}
```

Vectors K, A, B, C, D and E have “n” elements each, but only last “n-1” elements are changed.

Develop the corresponding SIMD algorithm with “p” processing elements named Q15. The SIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “msg_recd” in the algorithm. When a message arrives the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement as done in class.

Indicate the time complexity of your algorithm. Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc. Is your algorithm cost efficient? Explain.

A15) The SIMD algorithm is shown below.

The parallel time complexity is O(n) since there is no loop-level parallelism. The algorithm overlaps computations and communication so that processing elements perform computations while they also wait for messages to
Q15 (2-D MESH SIMD)
Local : myid, s ; i ; j ; A[0, 1, .., ((n/p) - 1)] ; B[0, 1, .., ((n/p) - 1)] ; C[0, 1, .., ((n/p) - 1)] ; D[0, 1, .., ((n/p) - 1)] ; E[0, 1, .., ((n/p) - 1)] ; K[0, 1, .., ((n/p) - 1)]; temp ; resulstsent ; east ; i ; j ; s ; j

Begin
    for all Pi where 0 <= i <= (p - 1) do
        s = n/p
        For (j=1 ; j<s ; j++)
            O(n/p)
            If myid = 0 then
                send \( K[s - 1] \), mylineararrayneighbor
                resulstsent = 1
            Endif
            For (j=0 ; j<s ; j++)
                \( K[j] = K[j] + temp \)
            Endfor
            If myid < (p-1) then
                send \( K[s - 1] \), mylineararrayneighbor
                resulstsent = 1
            Endif
        Endfor
        If (myid = 0 and j > 1) or (myid > 0 and j  0) then
        Endfor
        While resulstsent = 0 and myid < (p - 1) then
            If msg_recvd then receive (tmp)
                K[s - 1] = K[s - 1] + temp
            Endfor
            For (j=0 ; j<s ; j++)
                If j = 0 then
                    K[j] = K[j] + temp
                else
                    K[j] = K[j] + K[j - 1]
                Endif
                \( K[j] = K[j] + temp \)
            Endfor
            If myid < (p-1) then
                send \( K[s - 1] \), mylineararrayneighbor
                resulstsent = 1
            Endif
        Endwhile
        For (j=0 ; j<s ; j++)
            If (myid = 0 and j = 1) or (myid > 0 and j  0) then
            Endfor
    Endfor
End

arrive. But, depending on the interconnection network, the overlapping may not be sufficient, then PEs would wait additional time whose complexity depends on the interconnection network and is shown as \( O(?) \) below.

The cost is \( (n) * O(p) = O(np) \). This is not equal to the sequential time complexity and so the algorithm is not cost efficient.
The data decomposition is such that groups of “s” elements are assigned to the PEs based on their position in the mesh network:

The decompositions above does not change during the execution, except for vector K. For vector K, in the middle loop, the last element of each domain vector K for PEs from 0 to (p-2) is used by PEs from 1 to (p-1).

In addition, the assignment of the domains to the PEs is not as done before. The assignment follows the position of each PE on the 2-d square mesh network. Above, the assignment is given for a 4x4 mesh network.

There is communication among the processors to pass new values of K. The processors communicate in a linear array fashion. Since we have to use a square 2-D mesh, we have to consider how to embed the linear array in the physical network. We will use virtual processor id numbers. The programmer specifies virtual processor id numbers. The mapping of virtual processor id numbers to physical processor id numbers is done in the beginning of the computation. Library programs embed the linear array to the 2-D square mesh network and also map from virtual id numbers to physical id numbers. In the SIMD algorithm above, we use a system call before the “s = n/p” statement to embed the linear array in the 2-d square mesh network with “p” nodes.

Load balance is good since processing elements are busy until the computation completes, with the exception that in the while loop they are idle if the interconnection network takes a long time.

Q16) Consider the following sequential algorithm:

For (j = 0 ; j < n ; j++)
  C[j] = D[j] * g /* S4 */
}

The linear array is mapped to the 2-d square mesh network such that the function “mylineararrayneighbor” returns the id number of the node. The mapping can be as shown on the right:

There is communication on the bold links. The direction of communication and so the structure of the linear array are shown by the arrows

The data decomposition is such that groups of “s” elements are assigned to the PEs based on their position in the mesh network:

Local A, B, C, D, E and K vectors for a 4x4 mesh network:

The decompositions above does not change during the execution, except for vector K. For vector K, in the middle loop, the last element of each domain vector K for PEs from 0 to (p-2) is used by PEs from 1 to (p-1).

In addition, the assignment of the domains to the PEs is not as done before. The assignment follows the position of each PE on the 2-d square mesh network. Above, the assignment is given for a 4x4 mesh mesh network.

There is communication among the processors to pass new values of K. The processors communicate in a linear array fashion. Since we have to use a square 2-D mesh, we have to consider how to embed the linear array in the physical network. We will use virtual processor id numbers. The programmer specifies virtual processor id numbers. The mapping of virtual processor id numbers to physical processor id numbers is done in the beginning of the computation. Library programs embed the linear array to the 2-D square mesh network and also map from virtual id numbers to physical id numbers. In the SIMD algorithm above, we use a system call before the “s = n/p” statement to embed the linear array in the 2-d square mesh network with “p” nodes.

Load balance is good since processing elements are busy until the computation completes, with the exception that in the while loop they are idle if the interconnection network takes a long time.
Vectors A, B, C and D have “n” elements each.

Develop the corresponding **SIMD** algorithm with “p” processing elements named QSIMD. The SIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “msg_recvd” in the algorithm. When a message arrives the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement as done in class.

Indicate the time complexity of your algorithm. Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc. Is your algorithm cost efficient? Explain.

A16) The SIMD algorithm is below.

QSIMD (2-D MESH SIMD)

Local : my_id, s ; i ; j ; A[0, 1, .., ((n/p) - 1)] ; B[0, 1, .., ((n/p) - 1)] ; C[0, 1, .., ((n/p) - 1)] ; D[0, 1, .., ((n/p) - 1)] ; e ; f; g ; tmp ; resultreceived

Begin

for all Pi where 0 <= i <= (p - 1) do

s = n/p

For (j = 0 ; j< s ; j++)


If my_id = 0 and (j = 0) then

send (A[j], mylineararrayneighbor)

Endif

C[j] = D[j] * g

If my_id = 0 & msg_rcvd then

receive (tmp)

resultreceived = 1

if my_id ≠ (lastprocessorinlineararray) then

send (tmp, mylineararrayneighbor)

Endfor

If my_id ≠ 0 & resultreceived = 0 then

while (msg_rcvd = 0)

receive (tmp)

resultreceived = 1

if my_id ≠ (lastprocessorinlineararray) then

send (tmp, mylineararrayneighbor)

Endif

Endfor

If my_id ≠ 0 and (j ≠ 0) then

If temp ≠ 0) then A[j] = A[j] + temp


Endfor

Endfor

End

Each processor, except processor 0, executes all three loops. In the first loop, processor 0 sends its A[0] value to its linear array neighbor. The other processors perform calculations on “C” and at the same time check if they received A[0]. If yes, they send it to their linear array neighbor. This way, communication is overlapped with calculations.
Processor 0 also calculates “C” elements in the first loop. If by the end of the first loop a processor, other than processor 0, has not received A[0], it waits until it receives it and sends it to its linear array neighbor if its id number is not lastprocessorinlineararray. The waiting time is highly interconnection network dependent. In the third loop, all processors are busy. Then, the parallel time complexity is O(n/p) + O(p) + O(n/p) = O(n/p).

The cost : O(n/p) * O(p) = O(n).

The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decompositions are as shown below. The decompositions do not change during the execution:

```
A, B, C, D :
```

![Diagram showing processor numbering and data decompositions](image)

There is communication among the processors to pass A[0]. The processors communicate in a linear array fashion. Since we have to use a square 2-D mesh, we have to consider how to embed the linear array in the physical network. We will use virtual processor id numbers. The programmer specifies virtual processor id numbers. The mapping of virtual processor id numbers to physical processor id numbers is done in the beginning of the computation. Library programs embed the linear array to the 2-D square mesh network and also map from virtual id numbers to physical id numbers. In the SIMD algorithm above, we use a system call before the “s = n/p” statement to embed the linear array in the 2-d square mesh network with “p” nodes.

Load balance is good in the first and third loops since processing elements are busy until the loops complete. In the second (while) loop some processors are idle if they have not received A[0]. The waiting time depends on the inter-connection network.