DUE: NOT to be submitted

READ:
i) Part IV of the Sima book, except topics on dataflow systems and
ii) Chapter 6 and Appendix I of the Hennessy book

ASSIGNMENT: There are three problems.

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

1) Describe the Illinois cache coherence protocol, including its state diagram, as done in class.

2) 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.

3) Develop a dot-product algorithm on two vectors, A and B. Store the result in “k” for a 2-d toroidal-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.

RELEVANT QUESTIONS AND ANSWERS

Q1) One of the most common scientific operations is the transposition of a matrix of size nxn. 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:

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

The operation makes all rows columns (or all columns rows).
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)**

Transposing a matrix $A$ of size $n \times n$ using $p$ processors:

- **Constant**: $p$, $my\_id$
- **Global**: $A[0, 1, \ldots, (n - 1)][0, 1, \ldots, (n - 1)]$
- **Local**: $TARR[0, 1, \ldots, (n - 1)][0, 1, \ldots, ((n/p) - 1)]$, $k, j, l, i, m$

Begin

for $k = 1$ to $(p - 1)$ do

FORK Comp_transp($k$)

endfor

$m = n/p$

Comp_transp: For all $Pi$ where $0 \leq i \leq (p - 1)$ 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

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!

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.
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?

A2) 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 has to do actions in part (a) above in the order they are received. The whole process can be sped up if interconnection network switches are sophisticated such that they “combine” cache miss requests into one request. That is, 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).

Q3) A multiprocessor system consists of four processors on a single bus. The cache memories use the snoopy cache coherence protocol discussed in class (the protocol with three states). The processors have id numbers from 0 through 3.

Assume that a program will be run on the four processors and they will access main memory blocks “k” and “m.” These two blocks map to the same area in all cache memories. Processor P0 has already started the execution but not yet used blocks “k” and “m.” The other processors start after processor P0. Continue the following table that shows operations with respect to time:
### A3)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Operation</th>
<th>Block</th>
<th>Hit or miss</th>
<th><strong>Cache 0</strong> state changes</th>
<th>Any cache or memory action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td>Miss</td>
<td>Invalid to Exclusive Dirty</td>
<td>The memory supplies block k</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>k</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>k</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>k</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>m</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>m</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>W</td>
<td>m</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

### Q4) Consider the following sequential algorithm, named COMPARE:

```plaintext
k = 0
for (i = 0 ; i < n ; i++)
   if A[i] < B[i] 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 are suggested that you 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.

A4) Compare (UMA MIMD)

Constant : p
Local : i, local_k, j
Global : k, A[0, 1, ..., (n - 1)], B[0, 1, ..., (n - 1)]

Begin

k = 0
for j = 1 to (p -1) do

FORK Comp_compare(j) \(O(p)\)
endfor

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

for j = 0 to (n/p) do

if (A[i*n/p + j] < B[i*n/p + j] then

\{A[i*n/p + j] = B[i*n/p + j];
local_k = local_k + 1 \}
endif
endfor

F & A (k, local_k) \(O(1)\)
Join (p) \(O(1)\)
endfor

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

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:

\[ A \& B : \begin{array}{cccc}
\text{P}_0 & \text{P}_1 & \text{P}_2 & \ldots & \text{P}_{p-1} \\
n/p & n/p & n/p & \ldots & n/p
\end{array} \]

Communication is not required during the large for loop. The second loop requires communication which is handled by the interconnection network at its own speed.