DUE: April 23, 2014

READ:
- Related portions of Chapters 3, 5 and Appendix I of the Hennessy book
- Related portions of Chapter 1, 2, 4, 5 and 7 of the Jordan book

ASSIGNMENT: There are six problems.

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

1) Define the following terms related to parallelism and computational methods:
   a) Degree of parallelism
   b) Computational granularity

2) Consider the following piece of high-level language loop:
   
   ```
   for (i=1 ; i < 100 ; i = i + 1) {
     a[i] = b[i] + c[i] ; /* S1 */
     b[i] = a[i] + d[i] ; /* S2 */
     a[i+1] = a[i] + e[i] ; /* S3 */
   }
   ```

   i) List the dependencies.

   ii) Rewrite the high-level language loop based on your dependence list so that there is loop-level parallelism. That is, all iterations are independent of each other such that loop body statements for all iterations can be performed in parallel. Therefore, it would not matter whether iteration 28 is done first or iteration 43, the result will be correct. Note that you will NOT compile the loop to an assembly code.

3) Develop a sequential Binary Search algorithm to search element “k” on one-dimensional array A whose “n” elements are already ordered. If the search is successful, you will return the “index” of the array, i.e. A[index] = k. Otherwise, you will return -1. Specify the time complexity of your algorithm. What is the time complexity of your algorithm?

4) Consider the sequential Binary Search algorithm worked on in Question 3 above. Convert the sequential binary search algorithm to a parallel binary search PRAM algorithm to search element “k” on one-dimensional vector “A” with “n” ordered elements on “p” processors. If a processor finds “k,” it returns “index” where A[index] = k. Note that “index” is a global variable. If “k” is not found by any processor, then, “index” contains -1.
• Indicate the time complexity of the algorithm. Can it be cost efficient? Explain.
• Make observations relevant to the execution of your PRAM algorithm, including the
data decomposition, load balancing, the communication graph, etc.

5) Develop a cost efficient dot-product PRAM algorithm on two vectors, “A” and “B” with ”p“processors. Store the result in “k” which is a global variable. In order to keep the result of each
processor’s dot product result, use vector “D” with “p” elements. To store the result in “k,” pro-
cessor 0 copies D[0] to k.

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

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

b) Assume that a multiprocessor system consists of two processors on a single bus. Cache mem-
ories use the Illinois cache coherence protocol. The processors have id numbers 0 and 1.

Assume that a program is run on these two processors and they access main memory blocks “k”
and “m.” The two blocks map to the same area in the two cache memories. Processor 0 has
already started the execution but not yet used blocks “k” and “m.” Processor 1 starts after proces-
sor 0. Continue the following table that shows operations with respect to time:

<table>
<thead>
<tr>
<th>P</th>
<th>Op</th>
<th>Block</th>
<th>Hit/ Miss</th>
<th>Cache 0 state changes</th>
<th>Cache 1 state changes</th>
<th>Any cache or memory action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RELEVANT QUESTIONS AND ANSWERS

Q1) Consider the following piece of old code:

```
loop: L.D F2, 0(R1) ; Statement S1. R1 points at vector X & is initially (16)_{10}
ADD.D F0, F0, F2 ; S2. F0 is variable k & is initially 0
DADDI R1, R1, #(-8)_{10} ; S3
BNEZ R1, loop ; S4
S.D F0, 0(R2) ; S4
```

R1 and F0 are already initialized. Assume that R1 is initially (16)_{10}.

This code is written for the unpipelined processor, since, for example, the ADD.D is right after the L.D., even though the ADD.D depends on the L.D. Therefore, this code is written for machine model number 0. The corresponding algorithm is as follows:

```
for (i = 1 ; i <= 2 ; i++)
    k = k + X[i];
```

Determine the time complexity of the algorithm.

A1)

```
for (i = 1 ; i <= 2 ; i++)
    k = k + X[i];
```

The algorithm is a sequential algorithm.

We see that for n elements, the loop body is executed n times. The time complexity is O(n), a linear time complexity.

Q2) Consider the old code written for the unpipelined CPU (machine model number 0) below:

```
L.D F8, #0 ; Load value zero (0) to F8
DADDI R1, R0, #(64)_{10} ; Memory accesses are commented below:
loop: L.D F0, 0(RA) ; RA points at vector A
L.D F2, 0(RB) ; RB points at vector B
L.D F4, 0(RC) ; RC points at vector C
MUL.D F6, F0, F2
ADD.D F8, F8, F6
MUL.D F10, F4, F8
S.D F10, 0(RC) ; Stores to vector C
DADDI R1, R1, #(-1)_{10}
DADDI RA, RA, #8 ; RA is advanced
DADDI RB, RB, #8 ; RB is advanced
DADDI RC, RC, #8 ; RC is advanced
BNEZ R1, loop
S.D F8, 0(Rk) ; Stores to scalar k out of the loop.
```

L.D F8, #0 ; Load value zero (0) to F8
DADDI R1, R0, #(64)_{10} ; Memory accesses are commented below:
loop: L.D F0, 0(RA) ; RA points at vector A
L.D F2, 0(RB) ; RB points at vector B
L.D F4, 0(RC) ; RC points at vector C
MUL.D F6, F0, F2
ADD.D F8, F8, F6
MUL.D F10, F4, F8
S.D F10, 0(RC) ; Stores to vector C
DADDI R1, R1, #(-1)_{10}
DADDI RA, RA, #8 ; RA is advanced
DADDI RB, RB, #8 ; RB is advanced
DADDI RC, RC, #8 ; RC is advanced
BNEZ R1, loop
S.D F8, 0(Rk) ; Stores to scalar k out of the loop.
The corresponding algorithm is as follows:

```c
k = 0;
for (i = 0; i < 64; i = i + 1)
{
    k = k + (A[i] * B[i]); /* S1: Implements the dot product */
    C[i] = k * C[i]; /* S2: Implements the other operation */
}
```

a) Determine the time complexity of the algorithm.

b) List the dependencies

A2) a)

```
k = 0;
for (i = 0; i < 64; i = i + 1)
{
    k = k + (A[i] * B[i]); /* S1 */
    C[i] = k * C[i]; /* S2 */
}
```

O(n)

The algorithm is a **sequential** algorithm. We see that for n elements, the loop body is executed n times. The time complexity is O(n), a **linear** time complexity.

b) From S1 to S2 there is true dependence on k.
From S1 to S1 there is a loop-carried true dependence on k.
From S1 to S1 loop-carried output dependence on k
From S2 to S1, there is a loop-carried false dependence on k

Q3) Consider the following **old** MIPS code for the **unpipelined** MIPS processor (**machine model number 0**):

```
loop :  L.D   F0, 0(R1) ; Load from vector A
       ADD.D F0, F0, F1 ; F1 is already initialized with the constant value “b”
       MUL.D F0, F0, F2 ; F2 is already initialized with the constant value “c”
       DIV.D F0, F0, F3 ; F3 is already initialized with the constant value “d”
       S.D   F0, 0(R1) ; Store to vector A
       DADDI R1, R1, #8 ; Advance the vector A pointer
       DADDI R2, R2, #(-1)10 ; Decrement the loop counter
       BNEZ R2, loop ; Branch back if not the end
```

This code is without delayed loads, without delayed branches, and without any consideration for the latencies of functional units, etc.

The corresponding algorithm is as follows:

```
For (i = 0; i < n; i++)
    A[i] = ((A[i] + b) * c) / d
```

O(n)
a) Determine the time complexity of the algorithm.

b) Is there a loop-level parallelism in the high-level code? Explain.

A3) a) 
\[
\text{For } (i = 0 ; i < n ; i++)
\]
\[
\text{A}[i] = ((\text{A}[i] + b) \times c) / d 
\]
\[
\text{O}(n)
\]

The algorithm is a \textbf{sequential} algorithm. We see that for n elements, the loop body is executed n times. The time complexity is \text{O}(n), a \textbf{linear} time complexity.

b) The code has loop-level parallelism since \text{A}[i] does not depend on any other \text{A}[i], but itself. Thus \textbf{all} n iterations can run in parallel. The code is embarrassingly parallel!

Q4) Consider the following \textit{old} MIPS code for the \textit{unpipelined} MIPS processor (\textit{machine model number 0}) :

\begin{verbatim}
loop : 
L.D F0, 0(R2) ; Load from vector B
L.D F1, 0(R3) ; Load from vector C
L.D F2, 0(R4) ; Load from vector D
MUL.D F3, F1, F2
ADD.D F4, F0, F2
DIV.D F5, F0, F2
ADD.D F6, F3, F4
SUB.D F7, F6, F5
S.D F7, 0(R1) ; Store to vector A
DADDI R1, R1, #8 ; Advance the A pointer
DADDI R2, R2, #8 ; Advance the B pointer
DADDI R3, R3, #8 ; Advance the C pointer
DADDI R4, R4, #8 ; Advance the D pointer
DADDI R5, R5, #-110 ; Decrement loop counter which has (64)_{10} initially
BNEZ R5, loop ; Branch back if not the end
\end{verbatim}

The corresponding algorithm is as follows:

\[
\text{For } (i = 0 ; i < 64 ; i++)
\]
\[
\text{A}[i] = (\text{C}[i] \times \text{D}[i]) + (\text{B}[i] + \text{D}[i]) - (\text{B}[i] / \text{D}[i]) ;
\]

a) Determine the time complexity of the algorithm.

b) Is there a loop-level parallelism in the high-level code? Explain.

A4) a) 
\[
\text{For } (i = 0 ; i < 64 ; i++)
\]
\[
\text{A}[i] = (\text{C}[i] \times \text{D}[i]) + (\text{B}[i] + \text{D}[i]) - (\text{B}[i] / \text{D}[i]) ;
\]

b) The code has loop-level parallelism that \textbf{all} 64 loop iterations can be run in parallel. For n elements, the loop body is executed n times. The time complexity is \text{O}(n), a \textbf{linear} time complexity.
Q5) Consider the following algorithm:

For (i = 0 ; i < 64 ; i = i + 1)
{   A[i] = (B[i] * c) + D[i] ;         /* S1 */
    D[i] = A[i] / e   }                /* S2 */

List the dependencies. Explain clearly whether there is loop-level parallelism or not.

A5) The dependencies are as follows:

=> From S1 to S2 true dependency on A[i]
=> From S1 to S2 antidependence on D[i]

The code has loop-level parallelism since iterations on A[i] and D[i] are independent of each other. Thus all n iterations can run in parallel. It does not matter which iteration is performed when, the result will be correct.

Q6) 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.

a) List the dependencies among S1, S2 and S3. Explain clearly whether there is loop-level parallelism or not.

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

A6) a) The sequential algorithm is given:

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

The dependencies are as follows:

=> From S1 to S2 true dependency on A[i]
=> From S1 to S2 antidependence on C[i]
=> From S1 to S3 true dependence on A[i]
=> From S2 to S3 loop-carried true dependence on k
=> From S3 to S3 loop-carried output dependence on k

There is a loop-carried true dependence and loop carried outputs dependence on k. Therefore, the code does not have loop-level parallelism.
b) For “n” elements, the loop body is executed “n” times. These sequential time complexity is $O(n)$, linear time complexity.

Q7) Consider the following **sequential** algorithm:

For ($j = 1 ; j < n ; j++$)
{
    K[$j$] = K[$j$] + K[$j - 1$] ; /* S1 */
}

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

a) List the dependencies. Explain clearly whether there is loop-level parallelism or not.

b) What is the time complexity of the high-level code? Explain.

c) Develop the corresponding **PRAM** algorithm with “p” processors, named PRAMABD.

**Explain** the time complexity of the PRAM algorithm.

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

A7) a) The sequential algorithm is given:

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

There is no loop-level parallelism! Iterations on A[$i$] and D[$i$] are independent of each other. But there is a loop-carried true dependence on K[$i$]. Thus, one can generate two loops, one with S1 and the other one with S2 and S3. In the second loop, all n iterations can run in parallel. That is, in the second loop, it does not matter which iteration is performed when, the result will be correct.

b) For “n” elements, the loop body is executed “n-1” times. The time complexity is $O(n)$, linear time complexity.

c) The PRAM algorithm is below.

The parallel time complexity is $O(n)$ since there is no loop-level parallelism. The middle loop takes $O(n)$ time : $O(p) * O(n/p) = O(n)$ ! (p-1) processors update their K vector elements taking $O(n)$ time. Tjis time complexity is expected as the algorithm has a completely sequential part.

The cost : $O(n) * O(p) = O(np)$. The cost is not equal to the sequential time complexity. Since the cost is not equal to the sequential time complexity, the algorithm is not cost efficient.
The initial data decompositions are as shown below. The decompositions change lightly during the execution.

\[
\begin{align*}
\text{A, B, C,} & \quad \text{P, P, P, \ldots, P} \\
\text{D, E, K} & \quad \text{n/p, n/p, n/p, \ldots, n/p}
\end{align*}
\]

In the middle loop, processors 1 to p-1 use the last element of vector K in the domain of the processor whose id number is one less.

There is communication among the processors to pass new values of K in a linear array fashion:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & \ldots & p-1
\end{array}
\]

Load balancing is good except in the O(p) loop where gradually most of the processors become idle.
Q8) Develop a cost efficient PRAM algorithm that corresponds to the following sequential algorithm:

```plaintext
for (i = 0 ; i < n ; i++) do
    k = A[i]
    if (k = 0) then
        for (j = 0 ; j < 63 ; j++) do
            C[i][j] = B[i][j] + q
        endfor
    endif
endfor
```

The algorithm is called “CONDSCALE” standing for “Conditional Scaling.” Matrices “B” and “C” have “n” rows and 64 columns. Also, note that “n” can be quite large.

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

A8) The sequential time complexity is O(n) :

```plaintext
for (i = 0 ; i < n ; i++) do
    k = A[i]
    if (k = 0) then
        for (j = 0 ; j < 63 ; j++) do
            C[i][j] = B[i][j] + q
        endfor
    endif
endfor
```

The outer “for” loop is executed “n” times while the inner “for” loop is always executed 64 times. Since 64 is small compared with a large n, the time complexity of the inner loop is considered constant time, O(1). Thus, 64 is ignored for large “n.”

The PRAM algorithm :

```plaintext
CONDSCALE (CREW PRAM)
Global : n ; p ; A[0,...(n-1)] ; B[0,...(n-1)][0,...63] ; C[0,...(n-1)][0,...63] ; q
Local : j, k, l ; i
Begin
---
Spawn (P1, P2, ..., P(p-1))
For all Pi where 0 ≤ i ≤ (p-1) do
    for (j=0 ; j ≤ (n/p)-1 ; j++) do
        k = A[(i*n/p) + j]
        if (k = 0) then do
            for (l = 0 ; l ≤ 63 ; l++) do
                C[(i*n/p) + j][l] = B[(i*n/p) + j][l] + q
            endfor
        endif
    endfor
End
```

NYU School of Engineering  Page 9 of 24  CS6143  Handout No : 13  April 9, 2014
The parallel time complexity is \( O(n/p) \) since each processor executes the outer loop pointed by the above two parallel lines \( n/p \) times. The cost is the multiplication of the parallel time complexity and the number of processors: \( O(n/p) \times O(p) = O(n) \). Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decomposition: the A vector decomposition is as shown below. The decomposition does not change during the execution:

The B and C matrices are decomposed to the processors rowwise and the decomposition does not change during the execution:

The communication graph: The processors do not communicate with each other at all during the execution. Thus, the algorithm is embarrassingly parallel! There is no communication graph!

The load balance: All the processors have the same amount of work and do not wait for results from other processors (since it is embarrassingly parallel). The processors are busy all the time during the execution. Thus, the load balance is very good.

**Q9)** Develop a cost efficient DAXPY algorithm for two \( n \)-element vectors, "X" and "Y," for a PRAM. Indicate the time complexity of your algorithm. Make observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, etc.

**A9)** The algorithm is shown below.

The time complexity: \( O(\text{local\_domain\_size}) = O(n/p) \), a \textbf{linear} time complexity.

The algorithm is cost efficient since

\[
O(n/p) \times O(p) = O(n)
\]

Sequential algorithm time complexity: \textbf{linear} time complexity
The data decomposition throughout the calculation is static and straightforward as follows:

There is no communication among the processors: an **embarrassingly parallel** algorithm! There is **no communication graph**!

Load balancing is good since **all** processors are busy until the computation completes.

Q10) Consider the following sequential algorithm:

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

It works on two **n x 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 on page 2.

Develop the corresponding **cost-efficient PRAM** algorithm with “**p**” processors, named **PRAMABC**. **Explain** the time complexity of the PRAM algorithm.

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

A10) The PRAM algorithm is shown below.
The parallel time complexity is $O(n^2/p)$ as 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.

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

```
The initial data decomposition of matrices A and B is rowwise:

<table>
<thead>
<tr>
<th>m = n/p rows for P_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = n/p rows for P_1</td>
</tr>
<tr>
<td>m = n/p rows for P_2</td>
</tr>
<tr>
<td>\ldots \ldots</td>
</tr>
<tr>
<td>m = n/p rows for P_{p-1}</td>
</tr>
</tbody>
</table>
```

There is no communication among the processors: an **embarrassingly parallel** algorithm! There is no communication graph!

Load balancing is good since all processors are busy until the computation completes.

**Q11)** Consider the following sequential algorithm:

```c
For (i = 0 ; i < n ; i++)
    { A[i] = (C[i] * D[i]) + (B[i] + D[i]) - (B[i] / D[i]) ;
        E[i] = F[i] + G[i] ; }
```

PRAMABC (CREW PRAM)
Global : A[0,...,(n-1)][0,...,(n-1)]; B[0,...,(n-1)][0,...,(n-1)]; C[0,...,(n-1)]; n ; p ; s
Local : i; j; r
Begin
```
\[
\begin{align*}
    s &= n/p \\
    \text{Spawn} & (P_1, P_2, ..., P_{(p-1)}) \\
    \text{For all } P_i & \text{ where } 0 \leq i \leq (p-1) \text{ do} \\
    & \text{for } (j = 0 ; j < s ; j++) \text{ } C[i*s+j] = 0 \\
    & \text{for } (r = 0 ; r \leq n - 1 ; r++) \text{ do} \\
    & \quad C[i*s+j] = C[i*s+j] + (A[i*s+j, r] + B[i*s+j, r]) \\
\end{align*}
\]
```
```
Endfor
Endfor
End
```

The parallel time complexity is $O(n^2/p)$ as 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.
Develop the corresponding cost-efficient PRAM algorithm, named **PRAMABCDEFG**.

**Explain** the time complexity of the PRAM algorithm.

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

**A11)** The PRAM algorithm is as follows:

```
PRAMABCDEFG (CREW PRAM)
Global : 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)] ; n ; p ; s
Local : i; j
Begin
  s = n/p
  Spawn (P1, P2, ..., P(p-1))
  For all Pi where 0 ≤ i ≤ (p-1) do
    for (j=0 ; j ≤ s - 1 ; j++) do
      A[i*s+j] = (C[i*s+j] * D[i*s+j]) + (B[i*s+j] + D[i*s+j]) - (B[i*s+j] / D[i*s+j])
      E[i*s+j] = F[i*s+j] + G[i*s+j]
    enfor
  endfor
End
```

The parallel time complexity is \( O(n/p) \) since each processor executes the loop pointed by the two parallel lines \( n/p \) times.

The cost : \( O(n/p) \times O(p) = O(n) \)

The time complexity of the sequential algorithm is \( O(n) \) since the loop in the algorithm is executed “\( n \)” times. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decomposition : all the A, B, C, D, E, F and G vector decompositions are as shown below. The decompositions do not change during the execution:

```
For all seven vectors
the data decomposition is the same :
```

<table>
<thead>
<tr>
<th></th>
<th>A, B, C, D, E, F and G vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>n/p</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>n/p</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>n/p</td>
</tr>
<tr>
<td>( \ldots )</td>
<td></td>
</tr>
<tr>
<td>( P_{p-1} )</td>
<td>n/p</td>
</tr>
</tbody>
</table>

There is no communication among the processors : an **embarrassingly parallel** algorithm ! There is **no communication graph** !

Load balancing is good since all processors are busy until the computation completes.
Q12) Consider the following **sequential** algorithm:

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

E[j] = 1 / F[j] ;
```

Develop the corresponding **cost-efficient** PRAM algorithm with “p” processors, named PRAMABDF.

A12) The PRAM algorithm is as follows:

```
PRAMABDF (CREW PRAM)
Global : c ; n ; p ; s ; A[0,...(n-1)] ; B[0,...(n-1)] ; D[0,...(n-1)] ; E[0,...(n-1)] ; F[0,...(n-1)] ; TEMP[0,...(p-1)]
Local : i ; j ; my_id
Begin
    ---
    s = n/p
    Spawn (P1, P2, ..., P(p-1))
    For all Pi where 0 ≤ i ≤ (p-1) do
        For (j=0 ; j<s ; j++)
            A[i*s + j] = B[i*s + j] + D[i*s + j]
        Endfor
        TEMP[i] = 0
        For (j = 0 ; j < ((n/p) - 1) ; j++)
            TEMP[i] = TEMP[i] + A[i*s + j]
        Endfor
        For (j=0 ; j<(logp) ; j++)
            If imod2^j = 0 and 2i + 2^j < p) then
                TEMP[2i] = TEMP[2i] + TEMP[2i + 2^j]
            Endif
        Endfor
        If my_id = 0 then c = TEMP[0]
        For (j=0 ; j<s ; j++)
            E[i*s + j] = 1/F[i*s + j]
        Endfor
    Endfor
End
```

The parallel time complexity is O(n/p) since each processor executes three large loops n/p times.

The time complexity of the sequential algorithm is O(n) since the two loops in the algorithm are executed “n” times.

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:
There is no communication among the processors in all the loops except the one with $O(\log p)$ time complexity. There the communication is tree like. For an 8-processor system it looks like as follows:

Load balancing is good except in the $O(p)$ loop where gradually most of the processors become idle.

Q13) Consider the following sequential algorithm:

Develop the corresponding cost-efficient PRAM algorithm, named PRAMABCDEFGH.

Explain the time complexity of the PRAM algorithm.

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

A13) The PRAM algorithm is as follows:

```
PRAMABCDEFGH (CREW PRAM)
Global : 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)] ; n ; p ; s
Local : i; j
Begin
---
s = n/p
Spawn (P1, P2, ..., P(p-1))
For all Pi where 0 ≤ i ≤ (p-1) do
    for (j=0 ; j ≤ s - 1 ; j++) do
        A[i*s+j] = B[i*s+j] * C[i*s+j])
        D[i*s+j] = D[i*s+j] + E[i*s+j])
        F[i*s+j] = F[i*s+j] + G[i*s+j] ;
    endfor
endfor
End
```

The parallel time complexity is $O(n/p)$ since each processor executes the loop pointed by the two parallel lines $n/p$ times.

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

The time complexity of the sequential algorithm is $O(n)$ since the loop in the algorithm is executed “n” times. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.
The data decomposition: all the A, B, C, D, E, F and G vector decompositions are as shown below. The decompositions do not change during the execution:

For all seven vectors the data decomposition is the same:

<table>
<thead>
<tr>
<th></th>
<th>A, B, C, D, E, F, G vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_0</td>
<td>n/p</td>
</tr>
<tr>
<td>P_1</td>
<td>n/p</td>
</tr>
<tr>
<td>P_2</td>
<td>n/p</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>P_{p-1}</td>
<td>n/p</td>
</tr>
</tbody>
</table>

There is no communication among the processors: an embarrassingly parallel algorithm! There is no communication graph!

Load balancing is good since all processors are busy until the computation completes.

Q14) Consider the sequential algorithm in Question Q6 above. Develop the corresponding cost-efficient PRAM algorithm with “p” processors, named Q16. Explain the time complexity of the PRAM algorithm.

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

A14) The PRAM algorithm is as follows:

Q16 (CREW PRAM)
Global: n; p; 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,...(p-1)]; k
Local: i; j; my_id
Begin
---
s = n/p
Spawn (P_1, P_2, ..., P_{p-1})
For all Pi where 0 ≤ i < (p-1) do
    G[i] = 0
    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])
    Endfor
    For (j = 0; j < (log_2(p)); j++)
        If i mod 2^j = 0 and 2i + 2^j < p then
        Endif
    Endfor
    If my_id = 0 then k = G[0]
Endfor
End

The parallel time complexity is O(n/p) since each processor executes one large loop and contributes to another loop based on its id number no more than log_2(p) times: O(n/p) + O(log(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:
Load balancing is good except in the O(logp) loop where gradually most of the processors become idle.

Q15) Consider the following sequential algorithm:

\[
\begin{align*}
\text{For } (j = 0 : j < n : j++) \\
&\quad \text{If } (A[0] \neq 0 \& j \neq 0) \text{ then } A[j] = A[j] + A[0] \\
&\quad C[j] = D[j] * g \\
&\} \\
\end{align*}
\]

Vectors A, B, C and D have “n” elements each.

a) List the data dependencies among S1, S2, S3 and S4. Explain clearly whether there is loop-level parallelism or not.

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

c) Develop the corresponding cost-efficient PRAM algorithm with “p” processors, named Q3. Explain the time complexity of the PRAM algorithm.

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

A15) The sequential algorithm is given:

\[
\begin{align*}
\text{For } (j = 0 : j < n : j++) \\
&\quad \text{If } (A[0] \neq 0 \& j \neq 0) \text{ then } A[j] = A[j] + A[0] \\
&\quad C[j] = D[j] * g \\
&\} \\
\end{align*}
\]

a) The dependencies are as follows:

\[
\begin{align*}
\Rightarrow \text{From S1 to S2 true dependency on A[i]} \\
\Rightarrow \text{From S1 to S2 output dependency on A[i]} \\
\Rightarrow \text{From S1 to S2 antidependency on A[i]} \\
\Rightarrow \text{From S1 to S3 true dependence on A[i]} \\
\Rightarrow \text{From S2 to S3 true dependence on A[i]} \\
\Rightarrow \text{From S1 to S2 loop-carried true dependence on A[i]} \\
\end{align*}
\]

There is no loop-level parallelism due to the loop-carried dependency.
b) For “n” elements, the loop body is executed “n” times. The sequential time complexity is $O(n)$, linear time complexity.

c) The PRAM algorithm is below.

Q3 (CREW PRAM)
Global : $n$ ; $p$ ; $s$ ; $A[0,....(n-1)]$ ; $B[0,....(n-1)]$ ; $C[0,....(n-1)]$ ; $D[0,....(n-1)]$ ; $e$ ; $f$ ; $g$
Local : $i$ ; $j$ ; my_id

Begin
---
$s = \frac{n}{p}$
$A[0] = A[0] + e$
Spawn ($P_1$, $P_2$, ..., $P_{(p-1)}$)
For all $P_i$ where $0 \leq i \leq (p-1)$ do
    For ($j = 0$ ; $j < s$ ; $j++$) do
        If my_id $\neq 0$ or $j \neq 0$ then $A[i*s + j] = A[i*s + j] + e + A[0]$
        $C[i*s + j] = D[i*s + j] * g$
    Endfor
Endfor
End

The parallel time complexity is $O(n/p)$ since each processor executes the loop. Processor 0 also executes the first computation which is negligible. Therefore, the time complexity 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 as shown below. The decompositions do not change during the execution:

```
<table>
<thead>
<tr>
<th>A, B, C, D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
</tr>
<tr>
<td>$P_1$</td>
</tr>
<tr>
<td>$P_2$</td>
</tr>
<tr>
<td>$P_{(p-1)}$</td>
</tr>
</tbody>
</table>

n/p  
n/p  
n/p  
n/p
```

Load balancing is good. In the loop, processor 0 is less busy than the other processors for only one iteration which is negligible.

There is communication in the loop whose graph is a tree:

```
      0
     /|
    / |
   /  |
  1   2
     /|
    / |
   /  |
  ... p-1
```

Q16) A multiprocessor system consists of two 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 0 and 1.

Assume that a program will be run on the two processors and they will access main memory blocks “k” and “m.” These two blocks map to the same area in the two cache memories. Processor $P_0$ has already started the execution
but not yet used blocks “k” and “m.” The other processor, P1, starts after processor P0. Continue on the following table that shows operations with respect to time:

<table>
<thead>
<tr>
<th>P</th>
<th>Op</th>
<th>Block</th>
<th>H/M</th>
<th>Cache 0 state changes</th>
<th>Cache 1 state changes</th>
<th>Any cache or memory action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>Invalid to Shared</td>
<td>None</td>
<td>The memory supplies block k</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>m</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>m</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>m</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

A16) The table is as follows:

<table>
<thead>
<tr>
<th>P</th>
<th>Op</th>
<th>Block</th>
<th>H/M</th>
<th>Cache 0 state changes</th>
<th>Cache 1 state changes</th>
<th>Any cache or memory action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>Invalid to Shared</td>
<td>None</td>
<td>The memory supplies block k</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td>Hit</td>
<td>Shared to exclusive</td>
<td>None</td>
<td>Cache 0 broadcasts invalidate</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>m</td>
<td>Miss</td>
<td>Exclusive to shared</td>
<td>None</td>
<td>Memory is written k, memory supplies m</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>None</td>
<td>Invalid to shared</td>
<td>Memory supplies k</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td>Hit</td>
<td>Shared to exclusive</td>
<td>None</td>
<td>Cache 0 broadcasts invalidate</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>m</td>
<td>Miss</td>
<td>Exclusive to shared</td>
<td>Shared to shared</td>
<td>Cache 0 supplies m, memory is written m</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>m</td>
<td>Hit</td>
<td>Shared to invalid</td>
<td>Shared to exclusive</td>
<td>Cache 1 broadcasts invalidate &amp; Cache 0 invalidates m</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>Invalid to exclusive</td>
<td>Exclusive to invalid</td>
<td>Cache 0 broadcasts invalidate &amp; Cache 1 supplies m &amp; memory is written m</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>Exclusive to shared</td>
<td>None</td>
<td>Memory is written m, memory supplies k</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>None</td>
<td>Invalid to shared</td>
<td>Cache 0 supplies k</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td>Hit</td>
<td>Shared to exclusive</td>
<td>Shared to invalid</td>
<td>Cache 0broadcasts invalidate &amp; cache 1 invalidates k</td>
</tr>
</tbody>
</table>

Q17) A multiprocessor system consists of two processors on a single bus. Cache memories use the Illinois cache coherence protocol studied in a homework problem. The processors have id numbers 0 and 1.
Assume that a program is run on these two processors and they access main memory blocks “k” and “m.” The two blocks map to the same area in the two cache memories. Processor 0 has already started the execution but not yet used blocks “k” and “m.” Processor 1 starts after processor 0. Continue the following table that shows operations with respect to time:

<table>
<thead>
<tr>
<th>P</th>
<th>Op</th>
<th>Block</th>
<th>Hit/Miss</th>
<th>Cache 0 state changes</th>
<th>Cache 1 state changes</th>
<th>Any cache or memory action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A17)

<table>
<thead>
<tr>
<th>P</th>
<th>Op</th>
<th>Block</th>
<th>Hit/Miss</th>
<th>Cache 0 state changes</th>
<th>Cache 1 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 Modified</td>
<td>None</td>
<td>Cache 0 broadcasts invalidate for k Memory sends k to Cache 0</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>k</td>
<td>Miss</td>
<td>Modified to Invalid</td>
<td>Invalid to Modified</td>
<td>Cache 1 broadcasts invalidate for k Cache 0 sends k to Cache 1 and memory</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>Invalid to Shared</td>
<td>Modified to Shared</td>
<td>Cache 1 sends k to Cache 0 and memory</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td>Hit</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>Shared to Modified</td>
<td>None</td>
<td>Cache 0 broadcasts invalidate for m Memory sends m to Cache 0</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>Modified to Invalid</td>
<td>Shared to Modified</td>
<td>Cache 1 broadcasts invalidate for m Cache 0 sends m to Cache 1 and memory</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>m</td>
<td>Miss</td>
<td>Invalid to Shared</td>
<td>Modified to Shared</td>
<td>Cache 1 sends m to Cache 0 and memory</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>m</td>
<td>Hit</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>Shared to Exclusive</td>
<td>None</td>
<td>Memory sends k to Cache 0</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>Exclusive to Shared</td>
<td>Shared to Shared</td>
<td>Cache 0 sends k to Cache 1</td>
</tr>
</tbody>
</table>
Q18) A multiprocessor system consists of four processors on a single bus. The cache memories use the snoopy cache coherence protocol with three states discussed in class. 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:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Operation</th>
<th>Block</th>
<th>Hit or miss</th>
<th>Cache 0 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>

A18)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Operation</th>
<th>Block</th>
<th>Hit/Miss</th>
<th>Cache 0 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>Hit</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>k</td>
<td>Miss</td>
<td>Exc Dirty to Invalid</td>
<td>Cache 0 sends k, memory writes</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td>Hit</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>Invalid to Shared</td>
<td>Cache 1 sends k, memory writes</td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>None</td>
<td>Cache 0/1 sends k</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>None</td>
<td>Cache 0/1/2 sends k</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td>Hit</td>
<td>Shared to Exc Dirty</td>
<td>Cache 0 broadcasts invalidate &amp; caches 1, 2, 3 invalidate k</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>m</td>
<td>Miss</td>
<td>Exc Dirty to Shared</td>
<td>Cache 0 sends k to memory, memory sends m</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>m</td>
<td>Miss</td>
<td>None</td>
<td>Cache 0 sends m</td>
</tr>
<tr>
<td>2</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>Shared to Invalid</td>
<td>Cache 2 broadcasts invalidate &amp; Cache 0/1 supplies m &amp; Caches 0 and 1 invalidate m</td>
</tr>
</tbody>
</table>
Q19) A multiprocessor system consists of four processors on a single bus. Cache memories use the snoopy cache coherence protocol discussed in class (the protocol with three states). The processors have id numbers 0, 1, 2 and 3.

Assume that a program is run on these four processors and they access main memory blocks “k” and “m.” The two blocks map to the same area in the four cache memories. Processor 0 has already started the execution but not yet used blocks “k” and “m.” The other processors start after processor 0. Continue the following table that shows operations with respect to time:

<table>
<thead>
<tr>
<th>P</th>
<th>Op</th>
<th>Block</th>
<th>Hit/Miss</th>
<th>Cache 0 state changes</th>
<th>Any cache or memory action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>Invalid to Shared</td>
<td>Memory supplies k</td>
</tr>
<tr>
<td>1</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>1</td>
<td>W</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</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>
<tr>
<td>3</td>
<td>W</td>
<td>m</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>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>m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A19)
Q20) A multiprocessor system consists of three processors on a single bus. Cache memories use the snoopy cache coherence protocol discussed in class (the protocol with three states). The processors have id numbers 0, 1, and 2.

Assume that a program is run on these three processors and they access main memory blocks “k” and “m.” The two blocks map to the same area in the four cache memories. Processor 0 has already started the execution but not yet used blocks “k” and “m.” The other processors start after processor 0.

Continue the following table that shows operations with respect to time:

<table>
<thead>
<tr>
<th>P</th>
<th>Op</th>
<th>Block</th>
<th>Hit/Miss</th>
<th>Cache 0 state changes</th>
<th>Any cache or memory action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R</td>
<td>m</td>
<td>Miss</td>
<td>None</td>
<td>Cache 1 writes k to memory, C0 or C3 supplies m</td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>None</td>
<td>Memory supplies k</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>None</td>
<td>Cache 2 supplies k</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td>Hit</td>
<td>Shared to Exclusive</td>
<td>Cache 0 broadcasts invalid</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>Exclusive to Invalid</td>
<td>Cache 1 broadcasts invalid, Cache 0 supplies m to C1 and memory</td>
</tr>
<tr>
<td>P</td>
<td>Op</td>
<td>Block</td>
<td>Hit/Miss</td>
<td>Cache 0 state changes</td>
<td>Any cache or memory action</td>
</tr>
<tr>
<td>---</td>
<td>----</td>
<td>-------</td>
<td>---------</td>
<td>-----------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>Invalid to Shared</td>
<td>Memory supplies k</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>None</td>
<td>Cache 0 supplies k</td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>k</td>
<td>Miss</td>
<td>None</td>
<td>Cache 0/1 supplies k</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>Shared to Exclusive</td>
<td>Cache 0 broadcasts invalid, memory supplies m</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>Exclusive to Invalid</td>
<td>Cache 1 broadcasts invalid, Cache 0 supplies m to C1 and memory</td>
</tr>
<tr>
<td>2</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>None</td>
<td>Cache 2 broadcasts invalid, Cache 1 supplies m to C2 and memory</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>k</td>
<td>Miss</td>
<td>Invalid to Exclusive</td>
<td>Cache 0 broadcasts invalid, memory supplies k</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>k</td>
<td>Miss</td>
<td>Exclusive to Invalid</td>
<td>Cache 1 broadcasts invalid, Cache 0 supplies k to C1 and memory</td>
</tr>
<tr>
<td>2</td>
<td>W</td>
<td>k</td>
<td>Miss</td>
<td>None</td>
<td>Cache 2 broadcasts invalid, Cache 2 writes m to memory, Cache 1 supplies k to C2 and memory</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>m</td>
<td>Miss</td>
<td>Invalid to Shared</td>
<td>Memory supplies m</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>m</td>
<td>Miss</td>
<td>None</td>
<td>Cache 0 supplies m</td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>m</td>
<td>Miss</td>
<td>None</td>
<td>Cache 0/1 supplies k</td>
</tr>
<tr>
<td>0</td>
<td>W</td>
<td>m</td>
<td>Hit</td>
<td>Shared to Exclusive</td>
<td>Cache 0 broadcasts invalid</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>Exclusive to Invalid</td>
<td>Cache 1 broadcasts invalid, Cache 0 supplies m to C1 and memory</td>
</tr>
<tr>
<td>2</td>
<td>W</td>
<td>m</td>
<td>Miss</td>
<td>None</td>
<td>Cache 2 broadcasts invalid, Cache 1 supplies m to C2 and memory</td>
</tr>
</tbody>
</table>