PAR
Selection of Exams (with Solutions)

Eduard Ayguadé, Julita Corbalán, José R. Herrero,
Daniel Jiménez and Gladys Utrera

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya, UPC, BarcelonaTech

Course 2016-17 (Fall semester)

Contents

Part I

1^{st} In-term Exams

PAR – 1^{st} In-Term Exam – Course 2014/15-Q1
October 22nd, 2014

Question 1 (2 points)
We would like to have a multiprocessor system based on a non-uniform memory architecture (NUMA). It should be composed of 2 NUMA-nodes, a total of 32 GB of physical memory and 8 uni-processors (4 per NUMA-node), each having only one level of cache of 2 MBytes (with cache lines of 64Bytes). The multiprocessor should have the required hardware (seen in class) to support write-invalidate MSI cache-coherence protocol.

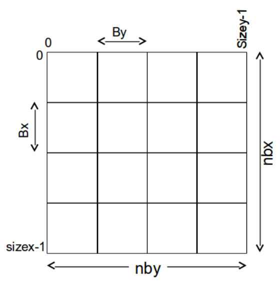
1. Draw a picture with all the components and interconnections of such NUMA system. Take into account the hardware support needed to keep the memory coherence inside a NUMA-node and between NUMA-nodes.
2. Focussing on the directory structure, how many bits should be used in the directory for each cache line in memory, and what are their role and possible values? How many entries does the directory structure have per NUMA-node?

Question 2 (4 points)
We have a code which has been instrumented using *Tareador*.

```
double u[SIZEX][SIZEY];

int sizex = SIZEX, sizey = SIZEY;
int nbx, Bx, nby, By;

nbx = 4; nby = 4;
Bx = sizex/nbx;
By = sizey/nby;

for (int ii=0; ii<nbx; ii++)
    for (int jj=0; jj<nby; jj++) {
        tareador_start_task("Block_Computation");
        for (int i=1+ii*Bx; i<=(ii+1)*Bx; i++)
            for (int j=1+jj*By; j<=(jj+1)*By; j++) {
                u[i][j]=  0.25 * (    u[ i   ][ j-1 ] +  // left
                                      u[ i   ][ j+1 ] +  // right
                                      u[ i-1 ][ j   ] +  // up
                                      u[ i+1 ][ j   ]);  // down
                }
        tareador_end_task("Block_Computation");
    }
```

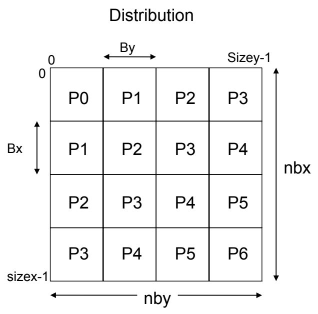
Assume that *sizey* and *sizex* are much larger than the block size and divisible by the number of blocks in each dimension. The code has been simplified to ease the analysis and does not treat the last block in a block row/column differently. The number of blocks in each dimension of the matrix is *nby* and *nbx*, being $Bx = sizex/nbx$ and $By = sizey/nby$ the dimensions of a block. In order to simplify the analysis, we will assume that each task executes $Bx \times By$ iterations (a block) of the inner loops. And each iteration takes time t_c.

1. Studying the loop carried data dependences which appear in the code, determine the order in which the blocks of the matrix can be computed. Do so by numbering the blocks in the figure below. Start numbering at 1, and increase the number in subsequent computation steps. If several blocks can be executed simultaneously then write the same number for all those blocks.



2. Draw a task graph such as the one *Tareador* would create.
3. Calculate T_1, T_∞ and S_∞.
4. Which is P_{Min}?
5. Which will be $T_{P_{Min}}$ and $S_{P_{Min}}$?
6. Draw a chronogram for the execution of the code above on P_{Min} processors.

Question 3 (4 points)
Given the code of the previous exercise executed in a distributed memory architecture (message passing) where the communication time is $t_{comm} = t_s + n \times t_w$, being t_s and t_w the "start-up" and sending time of an element, and n the size of the message.
Suppose that the matrix u is distributed in blocks among 7 processors, as shown in the next figure:



Using the data sharing model explained in class based on the distributed memory architecture with message passing and the rules for the remote load accesses, you should compute the T_p for $P = 7$. Assume that *sizey* and *sizex* are much larger than the number of processors and divisible by 4 (the number of blocks in each dimension), and that all the blocks are of the same size. In order to compute it we ask you to answer the following questions:

1. Which is the computation time for each processor? Complete the following table assuming that t_c is the cost of each most internal loop iteration of the nested loops.

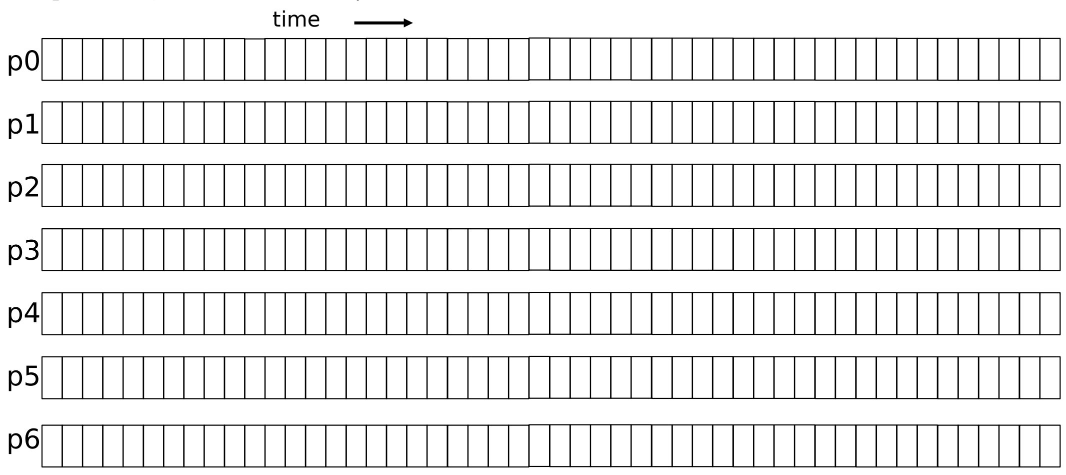
P0	P1	P2	P3	P4	P5	P6

2. Which data must be communicated between processors and when do they have to do these communications? Answer this question completing the following two tables. For instance, if P1 performed K remote load accesses, each of them of B_y elements, of its Upper bound (U) placed in P0 processor, you would have to write $K \times U_0(B_y)$ in the P1 slot of the table. Assume that at the end we do not have communication from all the matrix to a master process. Please, use U for Upper, D for Lower, R for Right, L for Left bound remote load accesses.

Initial communication of						
P0	P1	P2	P3	P4	P5	P6

Communication during the parallel computation of						
P0	P1	P2	P3	P4	P5	P6

3. Draw the execution timing diagram in the following empty diagram for the specific case of 7 processors and the distribution we have. In this question (and only this), in order to simplify the drawing of the timing diagram, suppose that each boundary segment communication lasts for one empty slot, and each block computation lasts for three empty slots. Please, in the diagram use U for Upper, D for Lower, R for Right, L for Left bound to identify the remote load accesses, C for computation, and arrows for synchronizations.



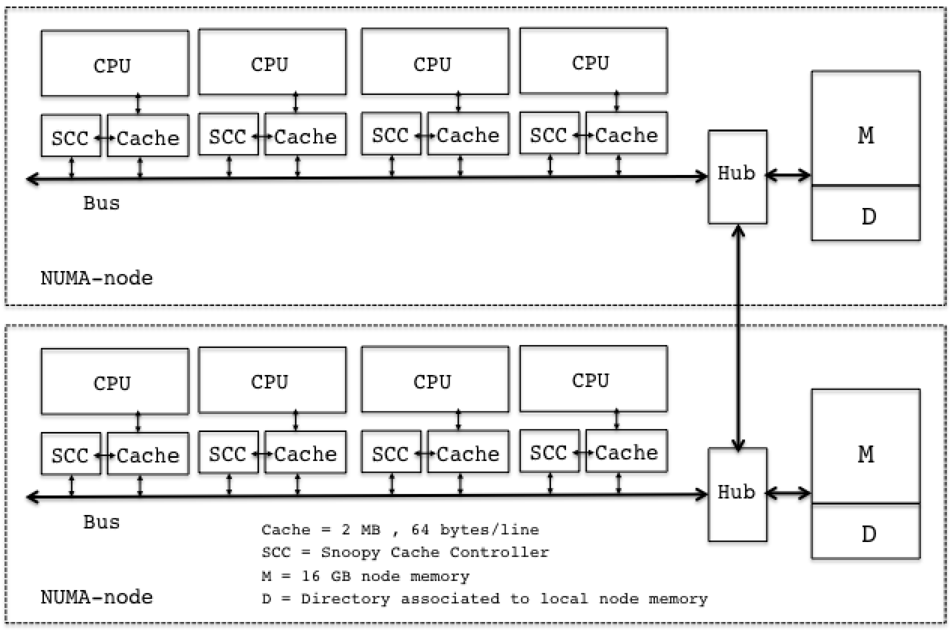
4. Calculate T_p for $P = 7$ as a function of B_x, B_y, t_s, t_w and t_c.

$T_p =$

Solution

Question 1 (2 points)

1.



CPU
SCC
Cache
Bus
Hub
M
D
NUMA-node
Cache = 2 MB , 64 bytes/line
SCC = Snoopy Cache Controller
M = 16 GB node memory
D = Directory associated to local node memory

2. The home NUMA-node is in charge of the coherence of its physical memory lines by means of the directory entries. A directory entry stores the line state and the identities of other NUMA-nodes sharing this memory line.

   Bits per directory entry:

   - Presence bits (nodes currently having the line), 1 bit per NUMA-node: 2 bits
   - State bits (to track the state of cache lines): 2 bits
     Those 2 bits can code up to four possible states:
     - I invalid (uncached, not valid in any cache)
     - S shared (two or more nodes may have copies)
     - M modified (dirty)
     - Note that a fourth state can be coded with 2 bits. This is useful for the additional state used in the MESI protocol: E exclusive (only this node has a copy, but not modified)

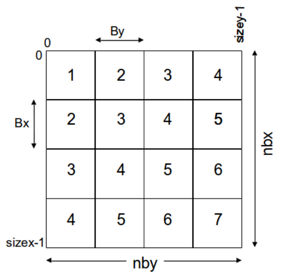
   Total: 4 bits per directory entry.

   The number of entries in the directory structure per NUMA-node is calculated dividing the memory in a NUMA-node (16GB) by the cache line size (64B). Therefore:

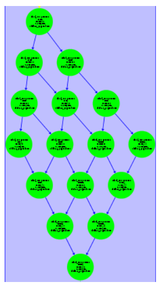
   Number of Directory Entries $= \frac{16GB}{64B} = 2^{28} = 256$ Mega entries

Question 2 (4 points)

1. The order of execution of the blocks is:



2. Task graph:



3. Let $T_{Block} = Bx \times By \times t_c$

   then:

   $T_1 = nbx \times nby \times Bx \times By \times t_c = 4 \times 4 \times T_{Block} = 16 \times T_{Block}$

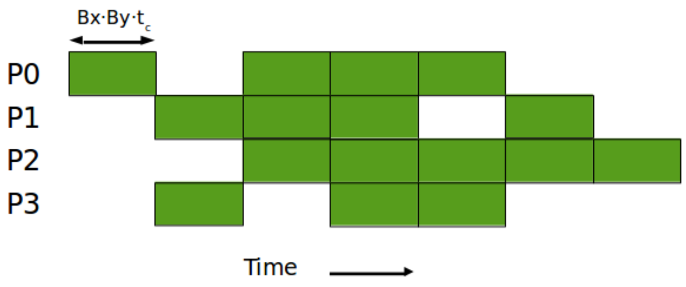
   $T_\infty = (nbx + nby - 1) \times T_{Block} = 7 \times T_{Block}$

   $S_\infty = \frac{1}{1-\phi}; \phi \rightarrow 1$

4. The minimum number of processors necessary to achieve the maximum parallelism in the execution of this code is related to the number of blocks in the counter-diagonal. In general, it would be $Min(nbx, nby)$. In this case, $P_{Min} = 4$.

5. If the scheduling is adequate $T_{P_{Min}}$ will be the same as T_∞. Then, $S_{P_{Min}} = \frac{T_1}{T_{P_{Min}}} = \frac{T_1}{T_\infty} = \frac{16 \times T_{Block}}{7 \times T_{Block}} = 2.28$

6. The tasks can be scheduled into different processors but must respect the dependencies. Disregarding the synchronization cost, a possible solution follows:



Question 3 (4 points)

1. Being C equal to the cost of a block: $B_x \times B_y \times t_c$:

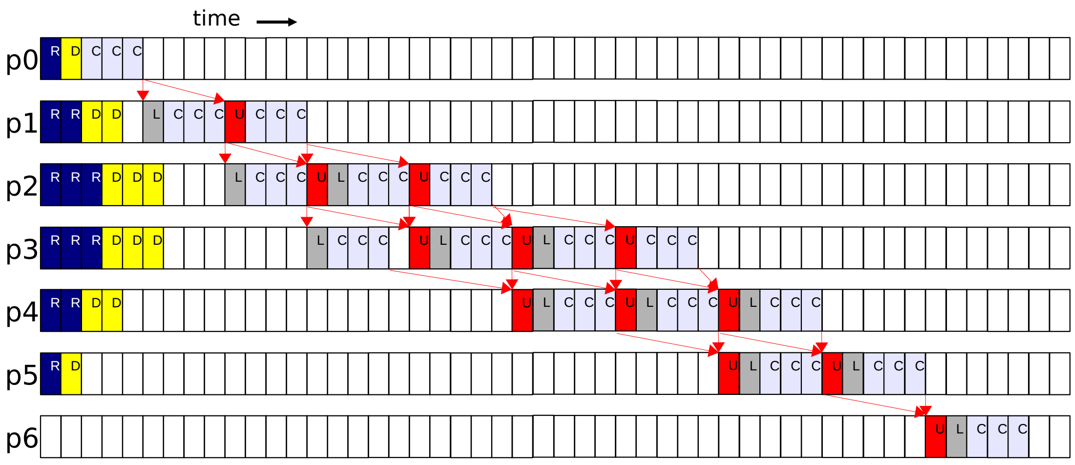
P0	P1	P2	P3	P4	P5	P6
C	$2C$	$3C$	$4C$	$3C$	$2C$	C

2. All communications D and U consist of B_y elements. All communications of L and R consist of B_x elements.

Initial Communication of						
P0	P1	P2	P3	P4	P5	P6
$1 \times R_1(B_x)$=1 R1	2 R2	3 R3	3 R4	2 R5	1 R6	-
$1 \times D_1(B_y)$=1 D1	2 D2	3 D3	3 D4	2 D5	1 D6	-

Communication during the parallel calculus of						
P0	P1	P2	P3	P4	P5	P6
-	$1 \times L_0(B_x)$= 1 L0	2 L1	3 L2	3 L3	2 L4	1 L5
-	$1 \times U_0(B_y)$= 1 U0	2 U1	3 U2	3 U3	2 U4	1 U5

3. The proposed solution has been simplified to ease the understanding of the diagram and the possible implementation of the parallel strategy. Other solutions, where some boundary communications are done as soon as possible, have also been accepted.



4. $T_p = t_s + B_x \times t_w + t_s + B_y \times t_w + 10 \times (B_x \times B_y \times t_c) + 7 \times (t_s + B_y \times t_w + t_s + B_x \times t_w) + 2 \times (t_s + B_x \times t_w)$

PAR – 1^{st} In-Term Exam – Course 2014/15-Q2
April 15th, 2015

Question 1: (3.0 points)
Given de following code in C:

```
int it_dot_product(int *X, int *Y, int n) {
   int i, sum=0;

   for (i=0; i<n; i++) sum += X[i]*Y[i];

   return sum;
}
int rec_dot_product(int *X, int *Y, int n) {

   int ndiv4 = n/4, sum1, sum2, sum3, sum4;

   if (n<=4) return it_dot_product(X,Y,n);
   sum1 = rec_dot_product(X, Y, ndiv4);
   sum2 = rec_dot_product(X+ndiv4, Y+ndiv4, ndiv4);
   sum3 = rec_dot_product(X+2*ndiv4, Y+2*ndiv4, ndiv4);
   sum4 = rec_dot_product(X+3*ndiv4, Y+3*ndiv4, n-3*ndiv4);
   return sum1+sum2+sum3+sum4;
}
void main() {
   int sum, X[N], Y[N];
   ...  sum = rec_dot_product(X,Y,N);  ...
}
```

Assume that there is not a parallelization strategy for this application yet and that its sequential execution time is $T_{seq} = 6$ units of time (u.t.). Answer the following questions:

1. Which is the minimum % execution time of the sequential application (T_{seq}) that we should parallelize if we are looking for an $S_\infty = 100$?
2. Which is the minimum % execution time of the sequential application (T_{seq}) that we should parallelize if we are looking for an $S_\infty = 100$ and there is a constant overhead of 0.01 u.t. to parallelize the application?

Independently of the previous questions and assuming that the parallelization strategy is that each function call is a task (i.e. any call to `rec_dot_product` or `it_dot_product`), and that $N = 64$. Answer the following questions:

3. Draw the task data dependency graph. How many `rec_dot_product` tasks are created? How many `it_dot_product` tasks are created?

4. Suppose that each `rec_dot_product` task execution has cost t_{rec}. This cost includes the if statement, addition of results sum_i and the creation of the tasks corresponding to each of the called functions, which will be executed like other tasks. Each `it_dot_product` execution has the same cost than the number of iterations of the loop multiplied by t_c. Compute:

   - $T_1 =$
   - $T_\infty =$
   - *Parallelism* =

5. What is the minimum number of processors that you would need in order to achieve the best performance (T_∞) for this parallelization strategy, considering that there is no overhead cost for creating threads, tasks, synchronization, communication, etc. other than the computational cost we have determined in the previous question?

Question 2: (3.5 points)
Assume a `NUMA` system with two `NUMAnodes`, two sockets per NUMAnode, six cores per socket, $24GB$ of main memory and a shared cache memory of $12MB$ (cache line size of 64 bytes) per socket. Data coherence is maintained using Write-Invalidate, Snoopy with MSI Protocol within each NUMAnode and using a Directory-based cache coherency protocol among NUMAnodes.
Answer the following questions:

1. How many bits are needed to maintain the coherence at each shared cache memory? What are their function/s? Please, give the number of bits per cache line and the total for each cache.

2. How many bits are needed to maintain the coherence at the directory structures? What are their function/s? Please, give the number of bits per memory line and the total number of bits per directory.

3. There is a case where the cache coherence protocol may affect the application performance although there is no memory position shared among the threads executing the application. What is that case? Explain it with an example and how you can avoid this issue.

4. The MESI protocol adds a new state to the MSI protocol. Explain which is the objective of this new state and how it helps to improve the performance of the MSI protocol. Justify your answer using an example, showing the protocol CPU events and BUS transactions with MSI and MESI. Do you need any additional bits in the coherence structures of MSI to include this new state of the MESI protocol?

Question 3 (3.5 points)
We want to find the expression that determines the parallel execution time in p processors (T_p) for the following code:

```
// Note that the loop induction variables: i and k are decremented.
for (i=N-2; i>=0; i--) {
    for (k=N-1; k>0; k--) {
        u[i][k] -= u[i+1][k] + u[i][k-1];
    }
}
```

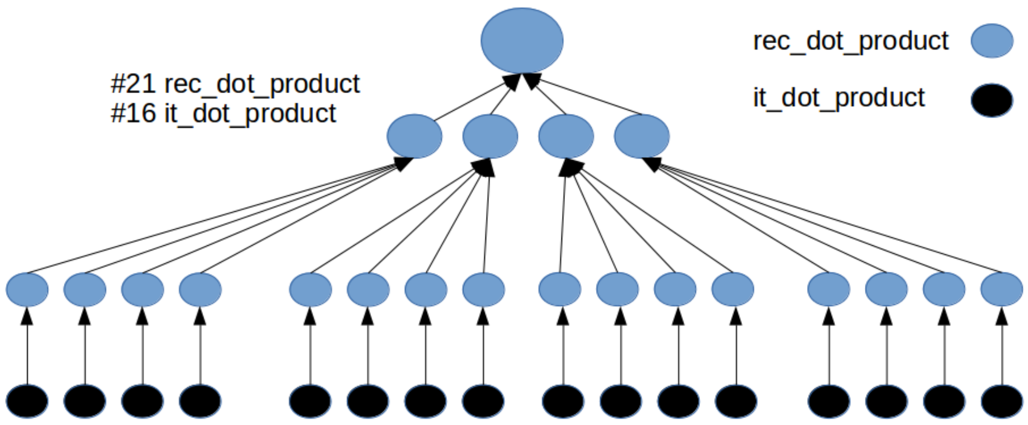
Let us assume the data sharing model explained in class based on the distributed memory architecture with message passing. The access time to remote data is determined by $t_{comm} = t_s + m \times t_w$, being t_s and t_w the "start-up" and sending time of an element, respectively, and being m the size of the message. Suppose also that the execution time of the iteration of the body of the most internal loop is t_c.
We ask you to complete the following table where we want to compare two different strategies for decomposing the `u` matrix: 1) *Column distribution*: the matrix `u` is distributed so that each processor has N/p consecutive columns and 2) *Row distribution*: the matrix `u` is distributed so that each processor has N/p consecutive rows. In case it is necessary to apply *blocking* for the parallel execution, consider that B is the block factor, i.e. the number or columns, or rows, taken at once in the dimension where blocking is applied; consider that $B \approx N$ otherwise. Assume that the resulting matrix remains distributed and there is no final communication to a single processor.

		Column distribution	Row distribution
Initial communication	Total number of messages		
	Size of each message		
	Contribution to T_p		
Computation of blocks (B)	Number of blocks		
	Number of elements in a block		
	Contribution to T_p		
Communication during the parallel computation	Total number of messages		
	Size of each message		
	Contribution to T_p		

Solution

Question 1: (3.0 points)

1. This should be 99% . We have to look for the ϕ that provides a $S_\infty = 100$.
   $S_\infty = 100 = \frac{1}{1-\phi}$
   $\phi = 0.99$
2. This should be more than 99%, in particular $\frac{595\times100}{600}$%. We have to look for the ϕ that provides a $S_\infty = 100$, considering that overhead.
   $S_\infty = 100 = \frac{T_{seq}}{T_{seq}(1-\phi)+0.01}$
   $\phi = \frac{595}{600}$
3. Task data dependency graph.



4. - $T_1 = 21 \times t_{rec} + 16 \times (4 \times t_c)$
   - $T_\infty = 3 \times t_{rec} + 4 \times t_c$
   - $Parallelism = \frac{T_1}{T_\infty} = \frac{21\times t_{rec}+16\times(4\times t_c)}{3\times t_{rec}+4\times t_c}$

5. This question is asking for P_{min}, which is 16 for this task data dependency graph.

Question 2: (3.5 points)

1. 2 bits per cache line. Their function is to keep the coherence state of the cache line, which can be Modified, Shared or Invalid.

   The total number per cache is the number of cache lines ($\frac{12\times 2^{20}}{2^6}$ lines) multiplied by the number of bits per line (2 bits/line). That is: 12×2^{15} bits

2. 4 bits per memory line: 2 bits for the coherence state information (Modified, Shared, Invalid) and 2 bits more for the presence information: 1 bit per NUMAnode multiplied by 2 NUMAnodes.

   The total number per Directory is the number of memory lines of the local memory in a NUMAnode ($\frac{12\times 2^{30}}{2^6}$ memory lines) multiplied by the number of bits per memory line (4 bits/line). That is: 12×2^{26} bits

3. This case can be false sharing. For instance, two threads (e.g. 0,1), updating several times two positions of a vector (e.g. thread 0 position $v[0]$ and thread 1 position $v[1]$) that are mapped to the same cache line. Those threads are not sharing the same memory position but they are sharing the same cache line, and then, one is invalidating the cache line of the other, and viceversa.

   The false sharing can be solved if we can add some padding between the two memory positions. For instance, thread 0 can update $v[0]$, and thread 1 can update $v[0 + CACHE\_LINE\_SIZE]$. In this case, those threads would not share the cache line, and then, they would not invalidate each other.

4. The new state helps to reduce the Bus traffic. In particular this state indicates that one cache line is the only CLEAN copy of a memory line. This state helps to improve the MSI performance for the case where a thread (e.g. thread 0) does a PrRd request of a memory line that any other cache has, and before any other thread requests it, thread 0 does a PrWr, provoking one BusRd for the PrRd CPU request and one BusRdx for the PrWr CPU request. In that case, MESI only provokes the first Bus Transaction (BusRd). The other one is not necessary.

   We don't need any additional bits in the coherence directory structure to keep the new Exclusive state. However, we may consider that it is necessary to have one bit of information in the snoopy hardware of each cache. This bit is the snoop state read from the system bus to know if there is any other copy in the system of a memory line when performing a BusRd within a NUMAnode. This information is necessary to know if the cache line will be the only clean copy in the system or not, and it is just one bit per snoopy hardware.

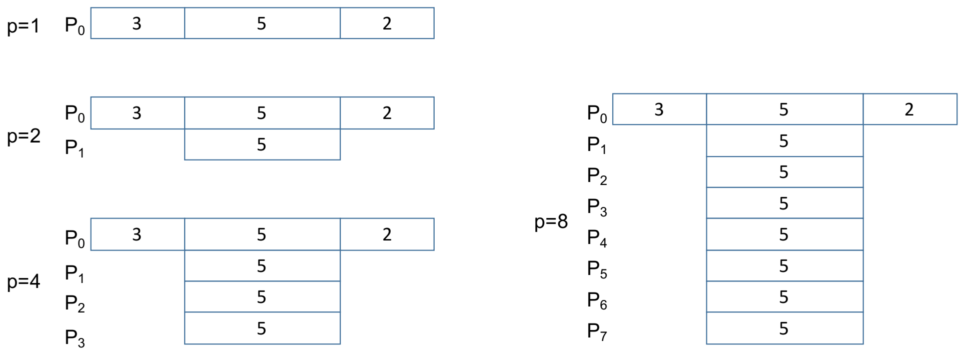
Question 3 (3.5 points)
The loops are traversed from high values to low values. This means the matrix accesses start close to the end of the matrix and evolve towards the beginning of the matrix.
With a distribution by columns:
1) the dependency created by the access `u[i+1][k]` remains internal within the execution of each processor and does not require communications.
2) The access `u[i][k-1]` does not create a dependency during the computations, but requires an initial communication: using the remote load model, each processor will initially do a remote load of the last column in the processor on its left (this is true for all processors but the leftmost one).

With a distribution by rows:
1) the dependency created by the access `u[i+1][k]` requires blocking in order to overlap the execution of blocks (of size B columns) in several processors. This implies the communication of B elements in the first row of the block, i.e. the row with lower index within the block. Thus, a processor will retrieve this first row in the block from the processor below (except for the last processor) once the block has been computed.
2) The access `u[i][k-1]` does not require communication since all the values in row `i` belong to the same processor.

		Column distribution	Row distribution
Initial communication	Total number of messages	$p-1$	–
	Size of each message	$\approx N$	–
	Contribution to T_p	$t_s + N \times t_w$	0
Computation of blocks (B)	Number of blocks	1 per processor	$N \div B$ per processor
	Number of elements in a block	$\approx (N^2 \div p)$	$\approx (N \div p) \times B$
	Contribution to T_p	$(N^2 \div p) \times t_c$	$((N \div B) + p - 1) \times ((N \div p) \times B) \times t_c$
Communication during the parallel computation	Total number of messages	–	$(p-1) \times (N \div B)$
	Size of each message	–	$\approx B$
	Contribution to T_p	0	$((N \div B) + p - 2) \times (t_s + B \times t_w)$

PAR – 1^{st} In-Term Exam – Course 2015/16-Q1
October 28th, 2015

Question 1 (3 points) Assume the following execution timelines to analyze the **weak scaling efficiency** for a parallel application, each timeline for a problem size proportional to the number of processors p:



1. We define the parallel fraction of the application as $\varphi = T_{par} \div (T_{seq} + T_{par})$ when the application is executed with the original problem size (i.e. the one used when p=1). In the expression T_{par} is the time spent on code that can be parallelized and T_{seq} is the time spent on code that cannot be parallelized. Compute the value of φ for the application according to this definition.

2. Compute the values for the speed–up $S(2)$, $S(4)$ and $S(8)$ considering that $S(p)$ is computed with respect to the sequential execution time **for the problem size used for p processors**.

3. Considering how the problem size is modified to evaluate weak scaling (i.e. T_{seq} does not change with the problem size and T_{par} increases linearly with the number of processors p), compute the general expression for $S(p)$, the speed-up when using p processors ($p > 1$) for the problem size associated to p processors. Compute it as a function of the parallel fraction φ defined above.

4. Compute the value or expression for $S(p \rightarrow \infty)$ if we add an overhead for fork/join proportional to the number of processors ($ovh = \beta \times p$, being β the fork/join overhead for one processor).

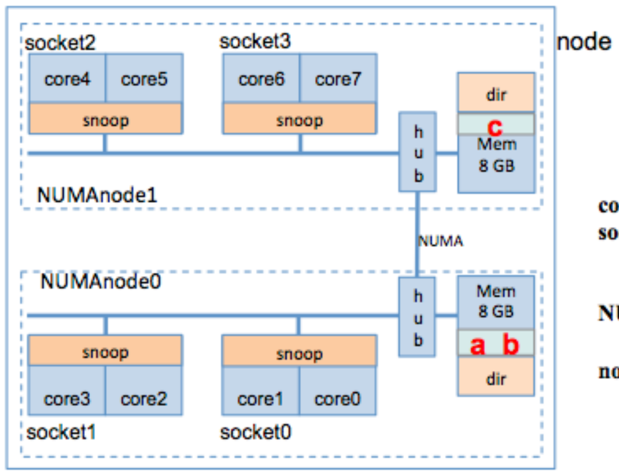
Question 2 (4 points) We want to find the expression that determines the parallel execution time in P processors (T_p) for the following program, considering there is a barrier synchronization between both nested loops **with no cost**:

```
#define N (1<<30)
float A[N][N], U[N][N];
int i,k;

// Loop1
for (i=0; i<N; i++)
    for (k=0; k<N-1; k++)
        A[i][k] = A[i][k] + 0.5*U[i][k+1];

// BARRIER SYNCHRONIZATION
// Loop2
for (i=1; i<N; i++)
    for (k=1; k<N; k++)
        U[i][k] = 0.8*A[i-1][k] + 0.5*U[i][k-1] - 0.2*U[i][k];
```

Assume the data sharing model explained in class based on the distributed-memory architecture with message passing, in which the access time to remote data is determined by $t_{comm} = t_s + m \times t_w$, being t_s and t_w the "start-up" and sending time of one element, respectively, and being m the size of the message. Assume also that the execution time of each iteration of the body in the most internal loops of Loop1 and Loop2 is t_c.

We ask: Complete the attached table where we want to compare two different decomposition strategies for matrices `U` and `A`: 1) *Column distribution*: matrices `U` and `A` are distributed so that each processor has N/P consecutive columns and 2) *Row distribution*: matrices `U` and `A` are distributed so that each processor has N/P consecutive rows. In case it is necessary to apply *blocking* for the parallel execution, consider that B is the width, for row distribution, or the height, for column distribution, of the block; consider that $B \approx N$ otherwise, which means that blocking is not applied or the number of blocks is 1. N is always multiple of P and B.

Question 3 (3 points) Assume a multiprocessor system with two NUMAnodes, each with two sockets and 8 GB of main memory. Each socket has two cores and a per-socket shared cache memory of 4MB. The cache line size is 64 bytes. Data coherence in the system is maintained using Write-Invalidate MSI protocols, with Snoopy cache coherency support within a NUMAnode and Directory-based cache coherency support between the two NUMAnodes.



coreX: Core.
socketY: Package with 2 cores,
4 MB cache shared between both cores
and snoopy coherence protocol.
NUMAnodeZ: set of 2 sockets connected to the same
NUMA "hub"/directory with 8 GB of main memory.
node: Node with 2 NUMAnodes.

1. How many bits are needed to maintain the coherence inside a NUMAnode in each per-socket shared cache memory? Which is the functionality for each one of these bits? Please, give the number of bits per cache line and the total for each cache.

2. How many bits are needed to maintain the coherence at the directory structures? Which is the functionality for each one of these bits? Please, give the number of bits per memory line. How many entries are needed at each NUMA Node Directory?

   (a) $R_3(a)$: core3 reads variable `a`;
   (b) $R_4(b)$: core4 reads variable `b`;
   (c) $W_5(b)$: core5 writes variable `b`;
   (d) $R_3(a)$: core3 reads again variable `a`;
   (e) $W_6(b)$: core6 writes variable `b`;
   (f) $R_6(c)$: core6 reads variable `c`, which is mapped in the same cache line as variable `a`.
   (g) $R_3(a)$: core3 reads again variable `a`;

Surname: **Name:** **Group:**

Table for Question 2

		Column distribution		Row distribution	
		Loop1	Loop2	Loop1	Loop2
Initial communication	Total number of messages				
	Size of each message				
	Contribution to T_p				
Blocks calculation (B)	Number of blocks				
	Number of elements in a block				
	Contribution to T_p				
Communication during parallel computation	Total number of messages				
	Size of each message				
	Contribution to T_p				

Surname: **Name:** **Group:**

Table for Question 3

Notation:
CPU events: $PrRd(cpu)$, $PrWr(cpu)$; **Snoopy bus transactions:** $BusRd(NUMAnode)$, $BusRdX(NUMAnode)$, $Flush(NUMAnode)$;
Cache line state: $M, S, I(socket)$; **NUMA transactions:** $RdReq$, $RdXReq$, $Invalidate$, $Intervention$;
State of memory line: $var\_name : M, S, I(NUMAnode)$; **Presence bits:** $var\_name(bitsequence)$, bit 0 on the right

	CPU event	cache hit/miss	Bus transaction	State of affected cache lines	NUMA transaction	State of affected line in directory	Presence bits of affected line in directory	Comments
$R_3(a)$								
$R_4(b)$								
$W_5(b)$								
$R_3(a)$								
$W_6(b)$								
$R_6(c)$								
$R_3(a)$								

Solution

Question 1

1. The parallel fraction is computed from the timeline for $p = 1$, so $\varphi = T_{par} \div (T_{seq} + T_{par}) = 5/10 = 0.5$.

2. Following the indications in the problem, the speed–up for the three problems sizes is:

   - $S(2) = 15/10 = 1.5$
   - $S(4) = 25/10 = 2.5$
   - $S(8) = 45/10 = 4.5$

3. As mentioned, $S(p)$ is computed as $S(p) = \frac{T_1(p)}{T_p(p)}$, being $T_1(p)$ and $T_p(p)$ the sequential and parallel time for the application executed with the problem size associated to p processors. For weak scaling the expressions for $T_1(p)$ and $T_p(p)$, as a function of the original T_{seq} and T_{par}, are

$$T_1(p) = T_{seq} + T_{par} \times p$$
$$T_p(p) = T_{seq} + T_{par}$$

   where $T_{seq} = (1 - \varphi) \times T_1(1)$ and $T_{par} = \varphi \times T_1(1)$ for the problem size for $p = 1$. By substituting in the previous expressions:

$$T_1(p) = (1 - \varphi) \times T_1(1) + \varphi \times T_1(1) \times p$$
$$T_p(p) = (1 - \varphi) \times T_1(1) + \varphi \times T_1(1) = T_1(1)$$

   and the quotient:

$$S(p) = \frac{T_1(p)}{T_p(p)} = (1 - \varphi) + \varphi \times p = 1 + (p - 1) \times \varphi$$

4. In this case, the expression for $T_p(p)$ includes an additional term to reflect the overhead

$$T_p(p) = (1 - \varphi) \times T_1(1) + \varphi \times T_1(1) + (\beta \times p)$$

   so when doing the limit for $p \to \infty$

$$S(p \to \infty) = (\varphi \times T_1) \div \beta$$

Question 2 See attached table.

Question 3

1. We need 2 bits per cache line in order to keep the coherence state of the cache line, which can be Modified, Shared or Invalid (MSI). The total number per 4MB cache is the number of cache lines ($\frac{4 \times 2^{20}}{2^6}$ lines) multiplied by the number of bits per line (2 bits/line). That is: 8×2^{14} bits for each cache.

2. The directory needs 4 bits per local memory line: 2 bits for the coherence state information (Modified, Shared, Invalid) and 2 bits more for the presence information: 1 bit per each NUMAnode. The total number of entries per Directory is the number of memory lines of the 8GB local memory in a NUMAnode ($\frac{8 \times 2^{30}}{2^6}$ memory lines). Multiplied by the number of bits per memory line (4 bits/line) gives a total of 8×2^{26} bits.

3. See attached table.

Table for Question 2 (solution)

		Column distribution		Row distribution	
		Loop1	Loop2	Loop1	Loop2
Initial communication	Total number of messages	$P-1$	-	-	$P-1$
	Size of each message	N	-	-	N
	Contribution to T_p	$t_s + N \times t_w$	-	-	$t_s + N \times t_w$
Blocks calculation (B)	Number of blocks	1 per thread	N/B per thread	1 per thread	1 per thread
	Number of elements in a block	$N \times N/P$	$B * N/P$	N^2/P	N^2/P
	Contribution to T_p	$N \times N/P \times t_c$	$(N/B + P - 1) \times (B * N/P) \times t_c$	$N^2/P \times t_c$	$N^2/P \times t_c$
Communication during the parallel calculus	Total number of messages	-	$N/B * (P-1)$	-	-
	Size of each message	-	B	-	-
	Contribution to T_p	-	$(N/B+P-2) \times (t_s + B \times t_w)$	-	-

Table for Question 3 (solution)

Notation:
CPU events: $PrRd(cpu)$, $PrWr(cpu)$; **Snoopy bus transactions:** $BusRd(NUMAnode)$, $BusRdX(NUMAnode)$, $Flush(NUMAnode)$;
Cache line state: $M, S, I(socket)$; **NUMA transactions:** $RdReq$, $RdXReq$, $Invalidate$, $Intervention$;
State of memory line: $var\_name : M, S, I(NUMAnode)$; **Presence bits:** $var\_name(bitsequence)$, bit 0 on the right

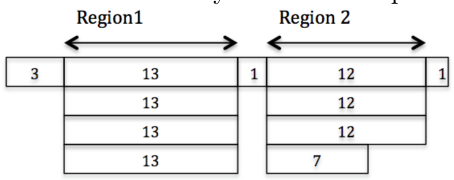
	CPU event	cache hit/miss	Bus transaction	State of affected cache lines	NUMA transaction	State of affected line in directory	Presence bits of affected line in directory	Comments
$R_3(a)$	PrRd(3)	Miss	$BusRd(0)$	$S(1)$	-	$ab : S(0)$	$ab(0,1)$	one single copy
$R_4(b)$	PrRd(4)	Miss	$BusRd(1)$	$S(2)$	RdReq	$ab : S(0)$	$ab(1,1)$	Snoopy cache sees request
$W_5(b)$	PrWr(5)	Hit	$BusRdX(1)$ $BusRdX(0)$	$M(2)$ $I(1)$	RdXReq Invalidate	$ab : M(0)$	$ab(1,0)$	False sharing
$R_3(a)$	PrRd(3)	Miss	BusRd(0)	$S(1)$ $S(2)$	RdReq Intervention	$ab : S(0)$	$ab(1,1)$	ab line sent to M in N0
$W_6(b)$	PrWr(6)	Miss	BusRdX(1)	$M(3)$ $I(1), I(2)$	RdXReq Invalidate	$ab : M(0)$	$ab(1,0)$	b modified, N0 gets invalidate

$R_6(c)$	PrRd(6)	Miss	Flush(1)
			
			BusRd(1)
				$S(3)$
				
				$I(1), I(2)$
					Intervention	$c : S(1)$	$c(1,0)$
							
							$ab(0,0)$
								conflict cache line, ab line sent to M in N0

$R_3(a)$	PrRd(3)	Miss	BusRd(0)	$S(1)$	-	$ab : S(0)$	$ab(0,1)$	Same cache line, c in Soc 3 should be contacted?

PAR – 1^{st} In-Term Exam – Course 2015/16-Q2
April 6th, 2016

Problem 1 (2 points)
Given the following timeline for the execution of a program with 4 processors, with two parallel regions, in which each horizontal line represents the activity of one of the processors:



1. Compute the value for T_1.

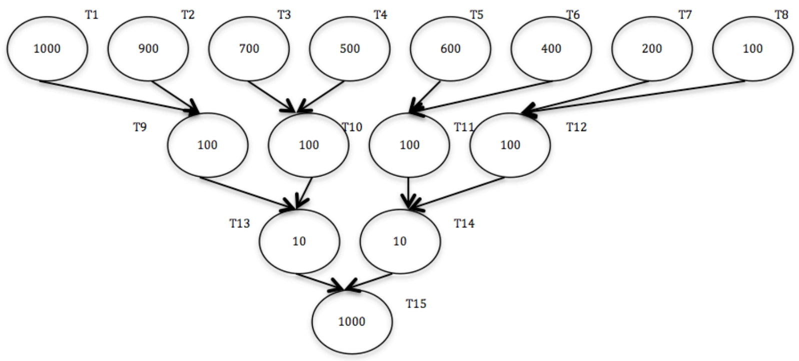
2. Compute the value for S_4.

3. Compute the parallel fraction φ for the application.

4. Compute S_∞ based on the previous φ, assuming that the two parallel regions in the program scale, i.e. can be ideally parallelized, with ∞ processors.

5. Please identify and briefly describe the source of overhead which affects the parallel performance of the program.

Problem 2 (2 points)
Given the following task dependence graph, in which the number inside each node represents its computational cost:



1. Compute T_1.

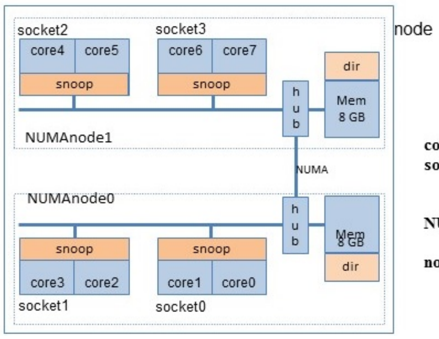
2. Compute T_∞.

3. Compute the Parallelism for an ideal execution of the task dependence graph with enough resources.

4. Propose an optimal task scheduling for the case of executing the tasks in this task dependence graph with 4 processors.

P0	
P1	
P2	
P3	

Problem 3 (3 points)
Assume a multiprocessor system with two NUMAnodes, each with two sockets and 8 GB of main memory. Each socket has two cores and a per-socket shared cache memory of 8MB. The cache line size is 64 bytes. Data coherence in the system is maintained using Write-Invalidate MSI protocols, with Snoopy cache coherency support within a NUMAnode and directory-based cache coherency support between the two NUMAnodes.



coreX: Core.
socketY: Package with 2 cores, 8 MB cache shared between both cores and snoopy coherence protocol.
NUMAnodeZ: set of 2 sockets connected to the same NUMA "hub"/directory with 8 GB of main memory.
node: Node with 2 NUMAnodes.

1. Which logic is necessary to maintain the cache coherence inside each NUMAnode and between the two NUMAnodes in a node? For each case, clearly indicate the number of bits needed to keep that information and their functionality, and compute the total number of bits that are required.

Given the following piece of C code:

```
#define SIZE 1000

typedef struct {
    double x, y; // Assume sizeof(double) = 8 bytes
} coord;

coord map[SIZE];

// code executed by thread0
for (int i=0; i<SIZE; i++)
    map[i].x = expensive_function(map[i].x);

// code executed by thread1
for (int i=0; i<SIZE; i++)
    map[i].y = expensive_function(map[i].y);
```

Assuming that 1) variable `map` is allocated in the memory of NUMAnode0; 2) there exists a single copy of all lines associated to variable `map` in the cache memory of socket1 (S state); and 3) thread0 executes on core0 and thread1 executes on core4, please answer the following questions:

2. Indicate the CPU events, cache hit/miss produced, bus and inter NUMAnode transactions and state for each cache and memory line after each memory access in the following sequence:
   (a) core0 reads the contents of map[0].x
   (b) core0 writes the contents of map[0].x
   (c) core4 writes the contents of map[0].y
3. Based on the analysis done and conclusions from the previous question, explain why this program is not able to reach the ideal speedup of 2X.
4. How would you change the data type definition of *map* in order to improve the parallel performance of the simultaneous execution of thread0 and thread1? In your solution, you cannot make use of data padding.

Problem 4 (3 points)
Given the following code:

```
#define N 16
#define BS 4
float b[N][N];
void main() {
char stringMessage[16];
...
tareador_ON();
for (int ii = 0; ii < N; ii=ii+BS) {
   for (int jj = 0; jj < N; jj=jj+BS) {
      sprintf(stringMessage,"task(%d,%d)",ii,jj);
      tareador_start_task(stringMessage);
      for (int i = max(1,ii);  i< min(ii+BS,N-1); i++)
         for (int j = max(1,jj); j < min(jj+BS,N-1); j++)
            b[i][j] += goo(b[i-1][j],b[i+1][j],b[i][j+1]);
      tareador_end_task(stringMessage);
   }
}
tareador_OFF();
}
```

1. Draw the task dependency graph assuming that `goo` only operates with the elements passed as parameters and does not modify (writes) any other global variable. You can assume that values of `N` and `BS` are those defined in the code.
2. For the following questions, assume: 1) execution on a distributed-memory architecture with $P = 2$ processors; 2) initial distribution of matrix `b` among the processors assigning N/P consecutive rows to each processor; 3) there is no need to communicate all rows to one of the processors at the end of process; 4) data sharing model explained at class, where the communication time of accessing m elements is $t_{comm} = t_s + m \times t_w$, being t_s and t_w the *start–up* time and the transfer time per element, respectively; and 5) the execution time for the body of the innermost loop body is t_c.
   (a) Draw the execution time diagram of the tasks in each of the processor assuming the task assignment shown in the table below, clearly indicating when remote memory accesses (communication), synchronizations between processors and task computations within each processor occur. You can assume that values of `N` and `BS` are those defined in the code. **Note that the task order in the list indicates the execution order within each processor that you have to respect.**

Processor	Task assignment
0	task(0,0), task(0,4), task(0,8), task(0,12), task(4,0), task(4,4), task(4,8), task(4,12)
1	task(8,0), task(8,4), task(8,8), task(8,12), task(12,0), task(12,4), task(12,8), task(12,12)

   (b) Obtain the expression that models the execution time T_p for 2 processors based on the time diagram of the previous question, clearly distinguishing the two components that contribute to T_2: computation $T_{2(comp)}$ and communication $T_{2(comm)}$.
   (c) Fill in the following table with a different task execution order of the SAME tasks assigned to each processor to reduce the parallel execution time of the application and obtain the new expression that models the execution time T_2. Reason about the task ordering proposed.

Processor	Task assignment
0	task(,), task(,), task(,), task(,), task(,), task(,), task(,), task(,)
1	task(,), task(,), task(,), task(,), task(,), task(,), task(,), task(,)

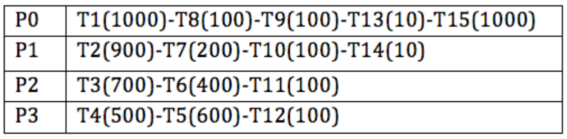
Solution

Problem 1 (2 points)

1. To compute T_1 (execution time in sequential) you must sum the execution time of all computation bursts in the execution timeline. In this case: 100 units of time.
2. The speedup is computed as $S_4 = T_1/T4$, in this case 3.33.
3. φ is the percentage of time that the program can execute in parallel with respect the the total execution time. The parallel fraction is computed from the timeline for $p = 1$, so $\varphi = T_{par} \div (T_{seq} + T_{par}) = 95 \div 100 = 0.95$.
4. S_∞ can be obtained from the general expression for the Amdahl's Law. $S_\infty = 1 \div (1 - \varphi)$. In that case: 20
5. In region 2 there is a clear load unbalance problem that does not allow an even distribution of the computation among the 4 processors.

Problem 2 (2 points)

1. To compute T_1 (execution time in sequential) you must sum the computational cost of each task in the task dependence graph. In this case: 5820 units of time.
2. To compute T_∞ the critical path needs to be identified. we must calculate the critical path. In this case, T1–T9–T13–T15, with a computational cost of 2110.
3. The parallelism is computed as $T_1 \div T_\infty$, 2.75 in this case.
4. An optimal task scheduling for the case of executing the tasks in this task dependence graph with 4 processors is:



P0	T1(1000)-T8(100)-T9(100)-T13(10)-T15(1000)
P1	T2(900)-T7(200)-T10(100)-T14(10)
P2	T3(700)-T6(400)-T11(100)
P3	T4(500)-T5(600)-T12(100)

Problem 3 (3 points)

1. Inside a NUMAnode we need a snoopy. The snoopy will maintiain 2 bits per cache line in order to keep the coherence state of the cache line, which can be Modified, Shared or Invalid (MSI). The total number per 8MB cache is the number of cache lines ($\frac{8\times 2^{20}}{2^6}$ lines) multiplied by the number of bits per line (2 bits/line). That is: 16×2^{14} bits for each cache.

   Between NUMAnodes in node, we need a directory structure. The directory needs 4 bits per local memory line: 2 bits for the coherence state information (Modified, Shared, Invalid) and 2 bits more for the presence information: 1 bit per each NUMAnode. The total number of entries per Directory is the number of memory lines of the 8GB local memory in a NUMAnode ($\frac{8\times 2^{30}}{2^6}$ memory lines). Multiplied by the number of bits per memory line (4 bits/line) gives a total of 8×2^{26} bits.

2. (a) The directory entry that corresponds to this variable has already the presence bits with value "01" and the state with value "S". There is a copy of the variable in the cache memory of socket1 with state value "S". Core0 generates PrRd, which provokes a BusRd transaction on the Bus. A copy of the variable is loaded into the cache memory of socket0 (the one that corresponds to core0) and the snoopy associated with this cache sets state value of this cache line to "S".

   (b) core0 generates PrWr, which provokes a BusRdX transaction on the bus. The snoopy associated to the cache memory of socket1 sets its state value to "I" (Invalidate). Local NUMAnode is the Home NUMAnode, so no request is generated to the hub of the other NUMAnode. The directory entry that corresponds to this variable is updated by setting the value state to "M". As there is no other hub NUMAnode that shares this variable no invalidation transaction is sent. The snoopy associated to cache of core0 sets the state value to "M".

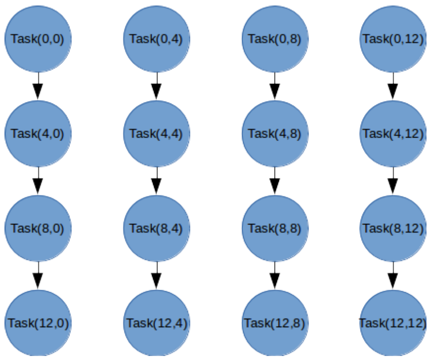
   (c) core4 generates PrWr, which provokes BusRdX transaction on the bus. The Local NUMAnode is not the Home NUMAnode, so the Local hub sends RdXReq to the Home Hub. The Home hub which is also the Owner Hub sends this information, so the Local responds sending Invalidate to Home/Owner Hub. Owner hub sends a bus transaction BusRdX inside NUMAnode0. Core0 snoop forces a flush of the cache memory copy and invalidates it. Owner hub sends dirty copy (Data) to Local hub indicating that it has invalidated the cache line. Local hub updates directory to indicate a dirty copy ("M" state) with presence bits "10". Core4 snoop updates the data and the coherence information on its cache memory.

3. Even both threads access to different fields in the `coord` struct for each position of *map*), they can be accessing the same cache line for reading and writing leading to false sharing. As the size of the double data type is 8 bytes, a cache line can allocate up to 4 elements. So even both threads were accessing different elements, they will be stored in the same cache line. This false sharing increases the access time to memory and thus limits the speedup of the parallel execution.
4. To avoid having the different values of x and y fields in the same cache line, we can separate them by allocating all the values of x together in contiguous address space and then all the values of y as in this data type definition:

```
typedef struct {
    double x[SIZE], y[SIZE];
} coord;

coord map;
```

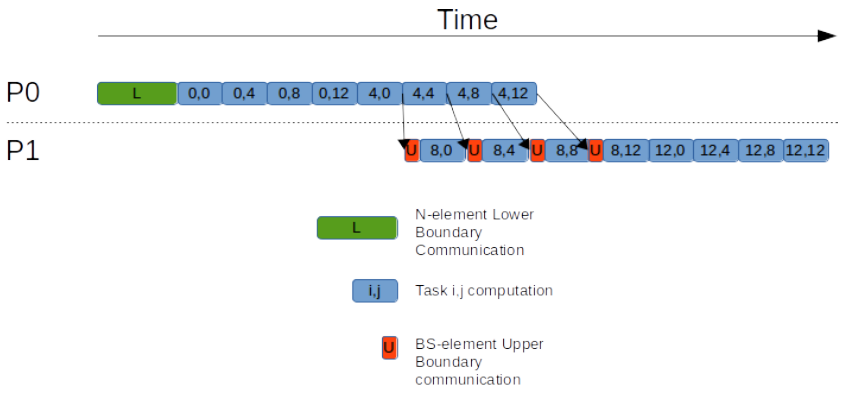
Problem 4 (3 points)

1. Task dependency graph assuming that `goo` only operates with the elements passed as parameters and does not modify (writes) any other global variable.



2. Answers to this exercise.

   (a) Execution time diagram of the tasks in each of the processor assuming the task assignment shown in the table:



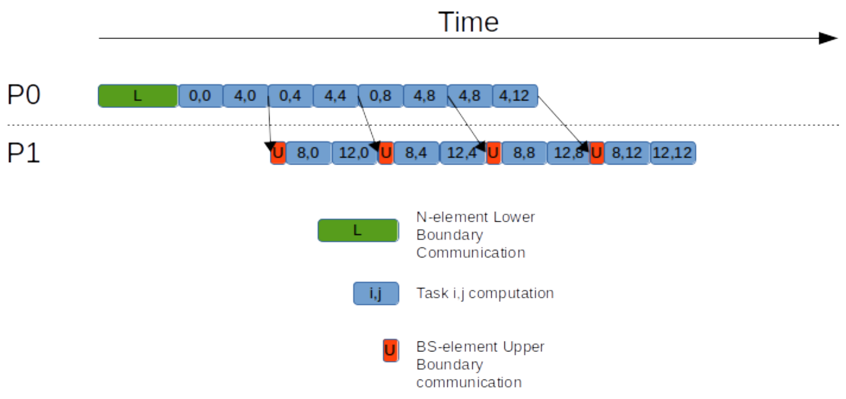
   (b) Expression that models the execution time T_p for 2 processors based on the time diagram of the previous question.

   $T_2 = T_{2(comp)} + T_{2(comm)}$
   $T_{2(comp)} = 13 \times BS^2 \times t_c$
   $T_{2(comm)} = t_s + N \times t_w + 4 \times (t_s + BS \times t_w)$

   (c) Assuming that the computational cost of a task is much bigger than the upper boundary communication of BS elements, we propose:

Processor	Task assignment
0	task(0,0), task(4,0), task(0,4), task(4,4), task(0,8), task(4,8), task(0,12), task(4,12)
1	task(8,0), task(12,0), task(8,4), task(12,4), task(8,8), task(12,8), task(8,12), task(12,12)

With that the dependences are resolved before than in previous task ordering assignment as can be seen in the following time diagram:



And then:

$T_2 = T_{2(comp)} + T_{2(comm)}$

$T_{2(comp)} = 10 \times BS^2 \times t_c$

$T_{2(comm)} = t_s + N \times t_w + 4 \times (t_s + BS \times t_w)$

Which is less than the previous T_2 ($3 \times BS^2 \times t_c$ units of time less).

Part II

2^{nd} In-term Exams

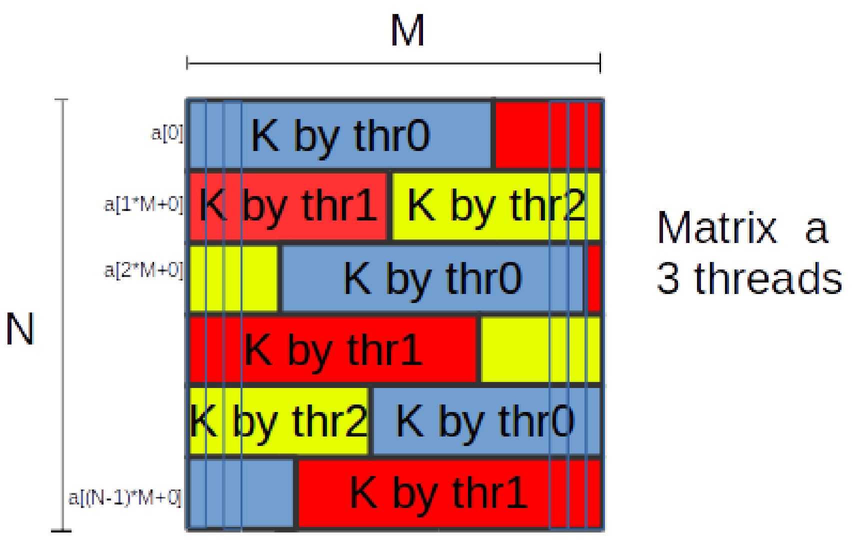
PAR – 2^{nd} In-Term Exam – Course 2014/15-Q1
December 17th, 2014

Question 1 (4 points)
Given a $N \times M$ matrix a that is updated by the following code:

```
...
for(i=0; i<N; i++)
  for (j=0; j<M; j++)
  {
     a[i*M+j] += foo(i,j);
  }
...
```

Assume that `foo(i,j)` does not have any memory access. `foo(i,j)` only computes an expression with no side effects. `NT` is the number of threads to be used in the parallel regions. Answer the following questions:

1. (1 point) Implement an OpenMP version of the code using Iterative Task Decomposition Strategy so that iterations are distributed as it is shown in the following figure for an example for $NT = 3$ threads: chunks of K consecutive elements of matrix `a` in memory, in the same row or not, are processed by one thread T (`thrT` in the figure). In this strategy, each thread is STATICALLY assigned chunks of K statements "`a[i*M+j] += foo(i,j)`" as it can be seen in the figure.



2. (1 point) Implement an OpenMP version of the code using Cyclic Data Geometric Decomposition by rows of the output matrix `a`. **Do not use `omp for`**.
3. (1 point) Implement an OpenMP version of the code that performs a Data Geometric Decomposition of matrix `a` in blocks of $K \times M$ elements of matrix `a` (a block of K rows of matrix `a`) in such a way that threads are dynamically assigned a new block of data of matrix `a` as soon as they finish their work, and there is still a block pending of being processed. **Do not use `omp for`**.
4. (1 point) Implement an OpenMP version of the code that performs a Data Geometric Decomposition of matrix `a` in blocks of $K \times K$ elements of matrix `a`. Blocks should be statically assigned between **ONLY TWO THREADS** in a chess board fashion. Assume that $N = M$ and $N\%K$ is zero. **Do not use `omp for`**.

Question 2 (2 points)
Assume you get this piece of OpenMP code:

```
void work(int w)
 {
 \\ Some real work here
 ...
 }

main() {
  omp_lock_t lock;
  omp_init_lock(&lock);
  #pragma omp parallel
  {
    int k;
    int nt = omp_get_num_threads();

    #pragma omp for
    for (k = 0; k < 32; k++) {
       #pragma omp task
       {
        int w = k%nt;
        omp_set_lock(&lock);
        work( w );
        omp_unset_lock(&lock);
       }
    }
  }
  omp_destroy_lock(&lock);

}
```

1. If OMP_NUM_THREADS=4, how many threads and tasks are launched during execution?
2. Is the code deadlock safe?
3. If the call to `work ( w )` has no dependences among calls for different values of `w`, could you improve the code in order to allow more parallelism?

Question 3 (4 points)
Matrix multiplication can be expressed as multiplications and additions of submatrices. For example, the multiplication $C = A * B$ can be seen as the calculation of their elements or submatrices $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} * \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Note, also, that it can be implemented in a recursive fashion. We have an application that stores matrices as quadtrees, with pointer matrices at the upper levels, and data submatrices (storing numerical values) at the leaves of the quadtree. We have recursive codes implementing matrix multiplication and addition using such quadtrees. We want to provide an efficient parallel implementation of the matrix multiplication based on the sequential versions shown below. The following excerpt of code shows the multiplication $C = A * B$ of matrices stored as quadtrees using routine `QTMat_RecMul`. Symbols a11, a12, ..., d21, d22, have been defined to ease the access to the 4 quadrants in a matrix stored as a quadtree (`qtmatrix` in the code).

```
/*
 * Square matrices of size <= BS are handled with the "classical
 * algorithms".  The shape of almost all functions is something like
 *
 *      if ( n <= BS )
 *          classical algorithms
 *      else
 *          n/= 2
 *          recursive call for 4 half-size submatrices
 *
 */

#define N  1024    /* Problem Size, i.e. multiply matrices of size NxN. */
#define BS   64    /* Data submatix Size. */

typedef union _qtmatrix {
    double **d;
    union _qtmatrix **p;
} *qtmatrix;

/* Let us define symbols a11, a12, ... , d21, d22, to ease the access to
   the 4 quadrants in a qtmatrix, i.e. QTM[i][j], with i,j in [1,2]. */

#define a11 A->p[0]
#define a12 A->p[1]
...
#define d22 D->p[3]

void matmul_submatrix(int N, double *A, double *B, double *C);
void matadd_submatrix(int N, double *A, double *B, double *C);

/* C = A + B */
void QTMat_RecAdd(int n, qtmatrix A, qtmatrix B, qtmatrix C) {
    if (n <= BS)
        matadd_submatrix(n, A, B, C); /* Add data submatrices. */
    else {
        n /= 2;
        QTMat_RecAdd(n, a11, b11, c11);      /* c11 = a11 + b11 */
        QTMat_RecAdd(n, a12, b12, c12);      /* c12 = a12 + b12 */
        QTMat_RecAdd(n, a21, b21, c21);      /* c21 = a21 + b21 */
        QTMat_RecAdd(n, a22, b22, c22);      /* c22 = a22 + b22 */
    }
}




/* C = A * B */
void QTMat_RecMul(int n, qtmatrix A, qtmatrix B, qtmatrix C)
{
    qtmatrix D; /* Auxiliary QuadTree Matrix. */

    if (n <= BS)
        matmul_submatrix(n, A, B, C); /* Multiply data submatrices. */
    else {
        D=new_qtmatrix(n); /* Dynamically allocate a QuadTree Matrix
                              for a problem of size n. */
        n2 = n/2;
        QTMat_RecMul(n2, a11, b11, d11);     /* d11 = a11 * b11 */
        QTMat_RecMul(n2, a12, b21, c11);     /* c11 = a12 * b21 */
        QTMat_RecAdd(n2, d11, c11, c11);      /* c11 += d11      */
        QTMat_RecMul(n2, a11, b12, d12);      /* d12 = a11 * b12 */
        QTMat_RecMul(n2, a12, b22, c12);      /* c12 = a12 * b22 */
        QTMat_RecAdd(n2, d12, c12, c12);      /* c12 += d12      */
        QTMat_RecMul(n2, a21, b11, d21);      /* d21 = a21 * b11 */
        QTMat_RecMul(n2, a22, b21, c21);      /* c21 = a22 * b21 */
        QTMat_RecAdd(n2, d21, c21, c21);      /* c21 += d21      */
        QTMat_RecMul(n2, a21, b12, d22);      /* d22 = a21 * b12 */
        QTMat_RecMul(n2, a22, b22, c22);      /* c22 = a22 * b22 */
        QTMat_RecAdd(n2, d22, c22, c22);      /* c22 += d22      */
        free_qtmatrix(D,n); /* Free space of Auxiliary QuadTree Matrix D. */
    }
}

void main() {
  ...
  /* Allocate and initialize quadtree matrices for a problem of size N. */
  ...
  QTMat_RecMul(N,A,B,C);    /* Compute C = A * B */
  ...
}
```

being `matmul_submatrix` a routine used to multiply two input data submatrices (A and B), updating the result matrix (C). Similarly, `matadd_submatrix` performs the addition of submatrices.

We ask you to provide pseudocode for the representative part of an efficient parallelization of the code above, using a *Divide and Conquer* approach and OpenMP explicit tasks.

1. (1.5 points) Write a parallel version using the `Tree` strategy. Do not use `OpenMP dependences`.
2. (1 point) Can the function calls in routine `QTMat_RecMul` above be rescheduled so that the `Tree` strategy in the previous exercise achieves better parallel performance? How would you change the code? Show the new parallelization of the resulting excerpt of code.
3. (1.5 points) Write a parallel version using `OpenMP dependences`.

Solution

Question 1 (4 points)

1. (1 point)

```
...
#pragma omp parallel num_threads(NT)
#pragma omp for collapse(2) schedule(static,K)
for(i=0; i<N; i++)
  for (j=0; j<M; j++)
  {
     a[i*M+j] += foo(i,j);
  }
...
```

2. (1 point)

```
...
#pragma omp parallel num_threads(NT) private(i,j)
{
  int id = omp_get_thread_num();
  for(i=id; i<N; i+=NT)
    for (j=0; j<M; j++)
    {
          a[i*M+j] += foo(i,j);
     }
}
...
```

3. (1 point)

```
...
#pragma omp parallel num_threads(NT)
#pragma omp single
{
  for(ii=0; ii<N; ii+=K)
  {
   #pragma omp task firstprivate(ii) private(j,i)
   for(i=ii; i<min(ii+K,N); i++)
    for (j=0; j<M; j++)
    {
       a[i*M+j] += foo(i,j);
    }
  }
}
...
```

4. (1 point)

```
...
#pragma omp parallel num_threads(2) private(ii,jj,i,j)
{
 int id= omp_get_thread_num();

 for(ii=0; ii<N/K; ii++)
 {
  for(jj=(ii%2)?(id+1)%2:id; jj<M/K; jj+=2)
  {
    for(i=ii*K; i<(ii+1)*K; i++)
    {
     for (j=jj*K; j<(jj+1)*K; j++)
     {
      a[i*M+j] += foo(i,j);
     }
    }
  }
 }
}
...
```

Question 2 (2 points)

1. The number of created threads is 4, as the environment variable OMP_NUM_THREADS states. Tasks are created at the K loop, which is splitted among the 4 existing threads, and each one creates 8 tasks. So 32 tasks total. Threads are responsible to run the tasks. As soon as they finish with one task they look for more and pick the next available one.
2. Yes, the code is deadlock safe. All threads that run tasks have to wait for the same lock so they call work() one after the other.
3. As calls to `work(w)` with different `w` value do not have conflicts, we could create an array of locks so tasks that generate a different `w` value can run the call to `work(w)` in parallel.

```
void work(int w)
 {
 \\ Some real work here
 ...
 }
```

```
int main() {
  int i;
  omp_lock_t lock[32];
  for (i = 0; i < 32; i++) omp_init_lock(&lock[i]);
  #pragma omp parallel
  {
    int k;
    int nt = omp_get_num_threads();

    #pragma omp for
    for (k = 0; k < 32; k++) {
       #pragma omp task
       {
        int w = k%nt;
        omp_set_lock(&lock[w]);
        work( w );
        omp_unset_lock(&lock[w]);
       }
    }
  }
  for (i = 0; i < 32; i++) omp_destroy_lock(&lock[i]);

}
```

Question 3 (4 points)

1. (1.5 points) Tree strategy not using OpenMP dependences.

```
#define CUTOFF ...

...

/* C = A + B */
void QTMat_RecAdd(int n, qtmatrix A, qtmatrix B, qtmatrix C) {
    if (n <= BS)
        matadd_submatrix(n, A, B, C); /* Add data submatrices. */
    else {
        n /= 2;
        #pragma omp task final( n < CUTOFF ) mergeable
        QTMat_RecAdd(n, a11, b11, c11);      /* c11 = a11 + b11 */

        ... /* Similarly for the other two quadrants. */

        #pragma omp task final( n < CUTOFF ) mergeable
        QTMat_RecAdd(n, a22, b22, c22);      /* c22 = a22 + b22 */
    }
}


/* C = A * B */
void QTMat_RecMul(int n, qtmatrix A, qtmatrix B, qtmatrix C)
{
    qtmatrix D; /* Auxiliary QuadTree Matrix. */

    if (n <= BS)
        matmul_submatrix(n, A, B, C); /* Multiply data submatrices. */
    else {
        D=new_qtmatrix(n); /* Dynamically allocate a QuadTree Matrix
                              for a problem of size n. */
        n2 = n/2;
        /* 1st quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a11, b11, d11);     /* d11 = a11 * b11 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a12, b21, c11);     /* c11 = a12 * b21 */
        #pragma omp taskwait
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecAdd(n2, d11, c11, c11);     /* c11 += d11       */

        /* 2nd quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a11, b12, d12);     /* d12 = a11 * b12 */

        ... /* Similarly for the other two quadrants. */

        /* 4th quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a21, b12, d22);     /* d22 = a21 * b12 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a22, b22, c22);     /* c22 = a22 * b22 */
        #pragma omp taskwait
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecAdd(n2, d22, c22, c22);     /* c22 += d22       */

        #pragma omp taskwait
        free_qtmatrix(D,n); /* Free space of Auxiliary QuadTree Matrix D. */
    }
}

void main() {
  ...
  #pragma omp parallel
  #pragma omp single
  QTMat_RecMul(N,A,B,C);    /* Compute C = A * B */
  ...
}
```

2. (1 point)

   We can move calls to `QTMat_RecAdd` to the end of the basic block which corresponds to the `else`. In this way, the number of synchronizations can be drastically reduced.

```
/* C = A * B */
void QTMat_RecMul(int n, qtmatrix A, qtmatrix B, qtmatrix C)
{
    qtmatrix D; /* Auxiliary QuadTree Matrix. */

    if (n <= BS)
        matmul_submatrix(n, A, B, C); /* Multiply data submatrices. */
    else {
        D=new_qtmatrix(n); /* Dynamically allocate a QuadTree Matrix
                              for a problem of size n. */
        n2 = n/2;
        /* Compute products. */
        /* 1st quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a11, b11, d11);     /* d11 = a11 * b11 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a12, b21, c11);     /* c11 = a12 * b21 */

        /* 2nd quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a11, b12, d12);     /* d12 = a11 * b12 */
        ... /* Similarly for the other two quadrants. */

        /* 4th quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a21, b12, d22);     /* d22 = a21 * b12 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a22, b22, c22);     /* c22 = a22 * b22 */

        #pragma omp taskwait

        /* Compute additions. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecAdd(n2, d11, c11, c11);     /* c11 += d11      */

        ... /* Similarly for the other two quadrants. */

        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecAdd(n2, d22, c22, c22);     /* c22 += d22      */

        #pragma omp taskwait
        free_qtmatrix(D,n); /* Free space of Auxiliary QuadTree Matrix D. */
    }
}
```

3. (1.5 points) Parallel version using `OpenMP dependences`.

   The recursive calls within routine `QTMat_RecAdd` are fully independent. Thus, we do not need to change that routine. The `main` function does not change either. With respect to routine QTMat_RecMul and for the sake of brevity we only include the data references which cause dependencies.

```
/* C = A * B */
void QTMat_RecMul(int n, qtmatrix A, qtmatrix B, qtmatrix C)
{
    qtmatrix D; /* Auxiliary QuadTree Matrix. */

    if (n <= BS)
        matmul_submatrix(n, A, B, C); /* Multiply data submatrices. */
    else {
        D=new_qtmatrix(n); /* Dynamically allocate a QuadTree Matrix
                              for a problem of size n. */
        n2 = n/2;
        /* 1st quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: d11)
        QTMat_RecMul(n2, a11, b11, d11);     /* d11 = a11 * b11 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: c11)
        QTMat_RecMul(n2, a12, b21, c11);     /* c11 = a12 * b21 */

        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(in: d11) depend(inout: c11)
        QTMat_RecAdd(n2, d11, c11, c11);     /* c11 += d11      */

        /* 2nd quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: d12)
        QTMat_RecMul(n2, a11, b12, d12);     /* d12 = a11 * b12 */

        ... /* Similarly for the other two quadrants. */

        /* 4th quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: d22)
        QTMat_RecMul(n2, a21, b12, d22);     /* d22 = a21 * b12 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: c22)
        QTMat_RecMul(n2, a22, b22, c22);     /* c22 = a22 * b22 */

        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(in: d22) depend(inout: c22)
        QTMat_RecAdd(n2, d22, c22, c22);     /* c22 += d22      */

        #pragma omp taskwait
        free_qtmatrix(D,n); /* Free space of Auxiliary QuadTree Matrix D. */
    }
}
```

PAR – 2^{nd} In-Term Exam – Course 2014/15-Q2
May 27th, 2015

Question 1: (5 points)
SAXPY stands for "Single-Precision A·X Plus Y". It is a function in the standard Basic Linear Algebra Subroutines (BLAS) library. SAXPY is a combination of scalar multiplication and vector addition. It takes as input two vectors of 32-bit floats X and Y with N elements each, and a scalar value A. It multiplies each element $X[i]$ by A and adds the result to $Y[i]$.
We already have this example of an implementation of the SAXPY function and a program that uses it:

```
void saxpy(int n, float a, float *x, float *y)
{
        for (int i = 0; i < n; ++i)
                y[i] = a * x[i] + y[i];
}

int main(int argc, char **argv)
{
        // Initialization of N, A, X, and Y
        ...
        saxpy(N, A, X, Y);
        return 0;
}
```

1. Implement an OpenMP parallel version of the SAXPY function code using a Linear (Iterative) Task Decomposition Strategy.
2. Create a new routine named *saxpy_rec* (recursive *saxpy*) with same parameters as *saxpy*, by means of splitting the vectors by half while they are larger than $MIN = 1 << 10$. The code must implement an OpenMP parallel version of the SAXPY function code following a Recursive Task Decomposition parallelization strategy (Divide and Conquer).
3. Modify the implementation of *saxpy_rec* from the previous question so that it stops creating tasks at a certain depth in the recursion tree (MAXDEPTH).
4. Independently of the previous implementations, and focusing now on the main program below, we want to achieve the asynchronous execution of the initialization loop, the two SAXPY calls and the two loops writing the result vectors to files. Create the contexts needed by using the OpenMP task construct and task dependencies.

```
#include <stdlib.h>
#include <stdio.h>

void saxpy(int n, float a, float *x, float *y)
{

        for (int i = 0; i < n; ++i)
                y[i] = a * x[i] + y[i];
}


int main(int argc, char **argv)
{
        int N = 1 << 20;    /* 1 million floats */

        if (argc > 1) N = atoi(argv[1]);

        float  *fx = (float *) malloc(N * sizeof(float));
        float  *fy = (float *) malloc(N * sizeof(float));
        FILE *fpx = fopen("fx.out", "w");
        FILE *fpy = fopen("fy.out", "w");
```

```
                /* simple initialization just for testing */
                for (int k = 0; k < N; ++k) {
                        fx[k] = 2.0f + (float) k ;
                        fy[k] = 1.0f + (float) k ;
                }

                /* Run SAXPY TWICE */
                saxpy(N, 3.0f, fx, fy);
                saxpy(N, 5.0f, fy, fx);

                for (int k = 0; k < N; ++k) {
                      fprintf(fpx, " %f ", fx[k]);
                }

                for (int k = 0; k < N; ++k) {
                        fprintf(fpy, " %f ", fy[k]);
                }

              free(fx); fclose(fpx);
              free(fy); fclose(fpy);
              return 0;
      }
```

Question 2: (5 points)
The transposition of a matrix A, written as A^T, consists in swapping all the rows for the columns, or all the columns for the rows. Thus, the result is a new matrix whose rows are the columns of the original.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^T = \begin{bmatrix} A & C \\ B & D \end{bmatrix}$$

The code excerpts below performs a transposition of a matrix of size $N \times N$ working on blocks of size $BS \times BS$. For simplicity, we consider N multiple of BS. The following routines are available:

- `void read_block(int Asrc[N][N], int i, int j, int Adest[BS][BS])`:
  subroutine which reads a block (submatrix) of size $BS \times BS$ of `Asrc` starting at `Asrc[i][j]` (`Asrc[i:i+BS-1][j:j+BS-1]`) and copies it in matrix `Adest`.
- `void transpose_block(int A[BS][BS])`:
  subroutine which performs the transposition of matrix `A` of size $BS \times BS$.
- `void write_block(int Asrc[BS][BS], int Adest[N][N], int i, int j)`:
  subroutine which copies matrix `Asrc` of size $BS \times BS$ into the block of `Adest` starting at `Adest[i][j]` (`Adest[i:i+BS-1][j:j+BS-1]`).

Using the building blocks above we can build some useful code to, for instance, copy a matrix transposed (left function: `copy_transposed`), or transposing a matrix in-place, i.e. changing the original matrix (right function: `transpose_inplace`).

```
void copy_transposed(int A[N][N],
                     int B[N][N]) {
  int block[BS][BS];
  int i, j;

  for (i = 0; i < N; i += BS) {
    // Transpose & copy diagonal block
    read_block(A, i, i, block);
    transpose_block(block);
    write_block(block, B, i, i);

    // Transpose & copy off-diagonal blocks
    // swapping blocks (i,j) and (j,i)
    for (j=i+BS; j<N; j+=BS) {
       read_block(A, i, j, block);
       transpose_block(block);
       write_block(block, B, j, i);
       read_block(A, j, i, block);
       transpose_block(block);
       write_block(block, B, i, j);
    }
  }
}
```

```
void transpose_inplace(int A[N][N]) {
  int Ablock_ij[BS][BS],
      Ablock_ji[BS][BS];
  int i, j;

  for (i = 0; i < N; i += BS) {
    // Transpose the diagonal block
    read_block(A, i, i, Ablock_ij);
    transpose_block(Ablock_ij);
    write_block(Ablock_ij, A, i, i);

    // Transpose off-diagonal blocks
    // and swap blocks (i,j) and (j,i)
    for (j=i+BS; j<N; j+=BS) {
       read_block(A, i, j, Ablock_ij);
       read_block(A, j, i, Ablock_ji);
       transpose_block(Ablock_ij);
       transpose_block(Ablock_ji);
       write_block(Ablock_ij, A, j, i);
       write_block(Ablock_ji, A, i, j);
    }
  }
}
```

a) We have some incomplete parallel codes to perform the two operations described above.

```
void par_copy_transposed(int A[N][N],
                         int B[N][N]) {
   #pragma omp parallel
            num_threads( ... )
     {
     int block[BS][BS];


     int i = ...
     int j = ...

     read_block(A, i, j, block);
     transpose_block(block);
     write_block(block, B, j, i);
     }
}
```

```
void par_transpose_inplace(int A[N][N]) {
   #pragma omp parallel
            num_threads( ... )
     {
     int Ablock_ij[BS][BS];


     int i = ...
     int j = ...

     read_block(A, i, j, Ablock_ij);
     transpose_block(Ablock_ij);
     write_block(Ablock_ij, A, j, i);
     }
}
```

We want to perform a data decomposition strategy by blocks where each matrix block is processed by one thread. **We ask** you to complete the parallel codes above so that they are correct.

a.1) Complete the pragma and the initialization of variables i and j so that, for both codes, each thread is responsible for the transposition of a single block.

```
#pragma omp parallel num_threads(.................)


int i =.................................
int j =.................................
```

a.2) Is there a need for synchonization in routine `par_copy_transposed`? In case of afirmative answer, change the code minimally to ensure correctness, showing only the representative code changes.

a.3) Is there a need for synchonization in routine `par_transpose_inplace`? In case of afirmative answer, change the code minimally to ensure correctness, showing only the representative code changes.

b) Using an additional data structure `already_read` to track accesses to blocks, complete the following parallel code so that it correctly transposes a matrix in-place:

```
#define N (1<<20)
#define BS (1<<12)
```

```
void par_transpose_inplace(int A[N][N]) {
    int block[BS][BS];
    int already_read[N/BS][N/BS];
    int i, j;

    // Perform initialization of already_read in parallel


    for(i=0;i<N/BS;i++) {
      for(j=0;j<N/BS;j++) {


      }
    }


    // Perform transposition of matrix A by blocks in parallel


    {


      i=                                        ;

      j=                                        ;



      read_block(A,i,j,block);


      transpose_block(block);


      write_block(block,A,j,i);

    }
}
```

Solution

Question 1: (5 points)

1.) The long loop (one million iterations in the example) will be splited among the actual threads, linearly assigning a static partition of an automatically privatized i counter. Each thread touches a disjoint part of vectors x and y, so they can run in parallel with no problem.

```
void saxpy(int n, float a, float *x, float *y)
{
#pragma omp parallel for
        for (int i = 0; i < n; ++i)
                y[i] = a * x[i] + y[i];
}
```

2.)

```
int main(int argc, char **argv)
{
        // Initialization of N, A, X, and Y
        ...
        #pragma omp parallel
        #pragma omp single
        saxpy(N, A, X, Y);
        return 0;
}

void saxpy_rec(int n, float a, float *x, float *y)
{
        int MIN = 1 << 10;    /* Min elements */

        if (n < MIN) {
         for (int i = 0; i < n; ++i)
                 y[i] = a * x[i] + y[i];
        }
        else {
         int nh = n/2;
         #pragma omp task
         saxpy_rec(nh, a, x, y);
         #pragma omp task
         saxpy_rec(n-nh, a, &x[nh], &y[nh]);
        }
}
```

3.)

```
void saxpy_rec(int n, float a, float *x, float *y, int d)
{
        int MIN = 1 << 10;    /* Min elements */
        int MAXDEPTH = 16 ;       /* Max number of created tasks inside saxpy */

        if (n < MIN) {
         for (int i = 0; i < n; ++i)
                 y[i] = a * x[i] + y[i];
        }
        else {
         int nh = n/2;
         #pragma omp task final (d > MAXDEPTH)   // can be mergeable, too
         saxpy_rec(nh, a, x, y, d++);
         #pragma omp task final (d > MAXDEPTH)   // can be mergeable, too
         saxpy_rec(n-nh, a, &x[nh], &y[nh], d++);
        }
}
```

4.)

```
int main(int argc, char **argv)
{
        int N = 1 << 20;    /* 1 million floats */

        if (argc > 1) N = atoi(argv[1]);

        float  *fx = (float *) malloc(N * sizeof(float));
        float  *fy = (float *) malloc(N * sizeof(float));
        FILE *fpx = fopen("fx3.out", "w");
        FILE *fpy = fopen("fy3.out", "w");

        #pragma omp parallel
        #pragma omp single
        {
        /* simple initialization just for testing */
                #pragma omp task depend(out:fx) // T1
```

```
                    for (int k = 0; k < N; ++k)
                            fx[k] = 2.0f + (float) k ;
                    #pragma omp task depend(out:fy) // T2
                    for (int k = 0; k < N; ++k)
                            fy[k] = 1.0f + (float) k ;


            /* Run SAXPY TWICE */
                    #pragma omp task depend(in:fx) depend(inout:fy)  // T3
                    saxpy(N, 3.0f, fx, fy);
                    #pragma omp task depend(in:fy) depend(inout:fx)  // T4
                    saxpy(N, 5.0f, fy, fx);

                    #pragma omp task depend(in:fx)  // T5
                    for (int k = 0; k < N; ++k) {
                            fprintf(fpx, " %f ", fx[k]);
                    }
                    #pragma omp task depend(in:fy)  // T6
                    for (int k = 0; k < N; ++k) {
                            fprintf(fpy, " %f ", fy[k]);
                    }
            }
            free(fx); fclose(fpx);
            free(fy); fclose(fpy);
            return 0;
    }
```

```
 - T1 and T2 can run in parallel
 - T1 and T2 have to be completed before T3 and T4 can be executed
 - T4 has to be completed before T5 can be executed
 - T3 has to be completed before T6 can be executed
 - T4 and T6 can run in parallel
 - T5 and T6 can run in parallel

 There is an implicit barrier at the end of the #single construct,
 before the calls to free() and fclose()
```

Question 2: (5 points)

a.1)
```
    #pragma omp parallel num_threads(N/BS*N/BS)
    ...
    int myid = omp_get_thread_num();
    int i = (myid / (N/BS)) * BS;
    int j = (myid % (N/BS)) * BS;
    ...
```

a.2) There is no need for syncronization since all the copies (plus transposition) can be done in parallel.

a.3) There is a need for syncronization since the writing of a block can only be performed after that block has already been read by the thread that uses it as input.

The minimum changes to the parallel code to produce correct parallel code consist in adding an OpenMP barrier before the call to the routine that writes the block:

```
    ...
    #pragma omp barrier
    write_block(Ablock_ij, A, j, i);
    ...
```

b)
```
#define N (1<<20)
#define BS (1<<12)

void par_transpose_inplace(int A[N][N]) {
    int block[BS][BS];
    int already_read[N/BS][N/BS];
    int i, j;

    // Perform initialization of already_read in parallel
    #pragma omp parallel for collapse(2)
    for(i=0;i<N/BS;i++) {
      for(j=0;j<N/BS;j++) {
        already_read[i][j] = 0;
      }
    }

    // Perform transposition of matrix A by blocks in parallel
    #pragma omp parallel private(i,j,block) num_threads(N/BS*N/BS)
    {

      int myid= omp_get_thread_num();
      int bi = myid / (N/BS);
      int bj = myid % (N/BS);
      i = bi * BS;
      j = bj * BS;

      read_block(A,i,j,block);

      already_read[bi][bj] = 1;
      #pragma omp flush

      transpose_block(block);

      while ( !already_read[bj][bi] ) {
        #pragma omp flush
      }

      write_block(block,A,j,i);

    }
}
```

PAR – 2^{nd} In-Term Exam – Course 2015/16-Q1
December 16th, 2015

Question 1 (2 points) Parallelise the following sequential code using *task dependences* in OpenMP, following a producer-consumer execution model. Producer and consumer code should be in two different tasks.

```
float sample[INPUT_SIZE+TAP1];
float coeff1[TAP1], coeff2[TAP2];
float data_out[INPUT_SIZE], final[INPUT_SIZE];

void main() {
    float sum;
    ...
    for (int i=0; i<INPUT_SIZE; i++) {
        // Producer: Finite Impulse Response (FIR) filter
        sum=0.0;
        for (int j=0; j<TAP1; j++)
            sum += sample[i+j] * coeff1[j];
        data_out[i] = sum;

        // Consumer: apply correction function (only depends on data_out[i] and coefficient)
        for (int j=0; j<TAP2; j++)
            final[i] += correction(data_out[i], coeff2[j]);
    }
    ...
}
```

Question 2 (4 points) Consider the following program doing some analytics on the information (`data`) available about bank clients. In particular, the program: 1) looks for the client with the highest `balance` (assuming there is a single one); and 2) counts how many clients receive an `interest` higher than 4.5% for the money they have in their account. The sequential iterative implementation of function `process_bank_data` below uses a global variable `best_client` to store the information of the richest client and returns the number of clients that benefit from a interest larger than 4.5%:

```
struct t_person{
   t_data data;      // personal information of bank client
   float balance;    // current balance for client
   float interest;   // interest for client
};
struct t_person best_client;

int process_bank_data(struct t_person *bank_info, int n) {
    int count=0;

    for(int i=0;i<n;i++) {
        if (bank_info[i].balance > best_client.balance) best_client = bank_info[i];
        if (bank_info[i].interest > 4.5) count++;
    }
    return count;
}

int main() {
    struct t_person bank_info[N_MAX];
    ...
    best_client.balance=0.0;
    int total_num = process_bank_data(bank_info, N_MAX);
    ...
}
```

Note: you can answer the three questions all in a single C program. **We ask** you:

1. Implement a sequential `recursive` function (`rec_process_bank_data`) that maintains the global variable `best_client`, has two recursive calls processing half of the vector and with base case $n = 1$.
2. Implement an OpenMP parallel version of the previous recursive function that implements a tree task decomposition parallel strategy with a cut-off mechanism based on recursion depth.
3. Review your OpenMP code and propose a modification of the code to avoid unnecessary synchronization overheads on updating global variables.

Question 3 (4 points) As you know, to construct a histogram, one has to "bin" the range of values (i.e. to divide the entire range of values into a series of intervals) and then count how many values fall into each interval. Consider the following (simplified) sequential program that "rebins" a histogram, i.e. builds a new histogram `new_hist` based on merging a certain number (`rebin`) of consecutive bins from the original histogram `hist`. Histograms have `size` and `new_size` bins, respectively, and `new_size` is always smaller than `size` and perfectly divides `size`.

```
data * input;

void build_histogram(data * input, int * size, int * hist);
void initialize_histogram(int size, int * hist);
void draw_histogram(int size, int * hist);

void rebinning(int size1, int * hist1, int size2, int * hist2) {
    int rebin = size1 / size2;
    for (int i = 0; i < size1; i++) {
        int tmp = i / rebin;
        hist2[tmp] += hist1[i];
    }
}

void main() {
    int * hist, size, * new_hist, new_size;

    build_histogram(input, hist, &size);
    new_size = size / 3;
    initialize_histogram(new_size, new_hist);
    rebinning (size, hist, new_size, new_hist);
    draw_histogram(new_size, new_hist);
}
```

Function `build_histogram` allocates memory to store a histogram and fills it with the information from `input`, also returning the number of bins (`size`) allocated. Function `initialize_histogram` simply allocates memory to store a histogram, initialising all bins to zero. Finally function `draw_histogram` plots the histogram. These functions are not parallelised in this exercise. **We ask**:

1. Write a parallel OpenMP version of the `rebinning` function following a *CYCLIC* data decomposition strategy for the input histogram `hist`. In a *CYCLIC* data decomposition consecutive elements are assigned to consecutive processors in a round–robin way, starting from processor 0. The output histogram `new_hist` is not decomposed.
2. Write a parallel OpenMP version of the `rebinning` function following a *BLOCK* data decomposition strategy for the output histogram `new_hist`. In a *BLOCK* data decomposition each processor is assigned a single block of consecutive elements, trying to maximize load balancing (the number of processors does not necessarely divides `new_size`. The input histogram `hist` is not decomposed.

Solution

Question 1
Using `depend` clause for `task`

```
float sample[INPUT_SIZE+TAP1];
float coeff1[TAP1], coeff2[TAP2];
float data_out[INPUT_SIZE], final[INPUT_SIZE];

void main() {
    float sum;
    ...

 #pragma omp parallel
 #pragma omp single
 for (int i=0; i<INPUT_SIZE; i++) {
    #pragma omp task depend(out: data_out[i]) private(sum) firstprivate(i)
        {
        // Producer: Finite Impulse Response (FIR) filter
        sum = 0.0;
        for (int j=0; j<TAP1; j++)
            sum += sample[i+j] * coeff1[j];
        data_out[i] = sum;
        }

    #pragma omp task depend(in: data_out[i]) firstprivate(i)
        {
        // Consumer: apply correction function (only depends on data_out[i] and coefficient
        for (int j=0; j<TAP2; j++)
            final[i] += correction(data_out[i], coeff2[j]);
        }
  }
}
```

Question 2

1.
```
struct t_person{
    t_data data;      // personal information of bank client
    float balance;    // current balance for client
    float interest;   // interest for client
};
struct t_person best_client;

int rec_process_bank_data(struct t_person *bank_info, int n) {
    int count1, count2;

    if (n==1) {
        if  (bank_info[0].balance > best_client.balance)
             best_client = bank_info[0];
        return (bank_info[0].interest > 4.5);
    }

    int ndiv2=n/2;
    count1 = rec_process_bank_data(bank_info, ndiv2)
    count2 = rec_process_bank_data(bank_info+ndiv2, n-ndiv2)
    return count1+count2;
}

int main() {
    struct t_person bank_info[N_MAX];
    ...
    best_client.balance=0.0;
    int total_num = rec_process_bank_data(bank_info, N_MAX);
    ...
}
```

2. In order to make it parallel as requested in the statement, we need to add one additional argument to control the recursion depth and add the necessary OpenMP pragmas

```
struct t_person{
   t_data data;       // personal information of bank client
   float balance;     // current balance for client
   float interest;    // interest for client
};
struct t_person best_client;

int rec_process_bank_data(struct t_person *bank_info, int n, int d) {
    int count1, count2;

    if (n==1) {
        #pragma omp critical
        if  (bank_info[0].balance > best_client.balance)
            best_client = bank_info[0];
        return (bank_info[0].interest > 4.5);
    }

    int ndiv2=n/2;

    #pragma omp task shared(count1) final(d>CUTOFF) mergeable
    count1 = rec_process_bank_data(bank_info, ndiv2, d+1)
    #pragma omp task shared(count2) final(d>CUTOFF) mergeable
    count2 = rec_process_bank_data(bank_info+ndiv2, n-ndiv2, d+1)
    #pragma omp taskwait

    return count1+count2;
}

int main() {
    struct t_person bank_info[N_MAX];
    ...
    best_client.balance=0.0;
    #pragma omp parallel
    #pragma omp single
    int total_num = rec_process_bank_data(bank_info, N_MAX, 0);
    ...
}
```

3. The idea is to acquire the synchronization just if there is a chance to need it. This is the test&test&set technique:

```
    ...
    if (n==1) {
        // We apply test&test&set in order to avoid unnecessary synchronizations
        if  (bank_info[0].balance > best_client.balance)    // TEST
            #pragma omp critical
            if (bank_info[0].balance > best_client.balance) // & TEST
                best_client = bank_info[0];                 // & SET
        return (bank_info[0].interest > 4.5);
    }
    ...
```

Question 3

1. One possible solution using atomic is:

```
void rebinning(int size1, int * hist1, int size2, int * hist2) {
    int rebin = size1 / size2;
    #pragma omp parallel
    {
        int myid = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        for (int i = myid; i < size1; i += howmany) {
            int tmp = i / rebin;
            #pragma omp atomic
            hist2[tmp] += hist1[i];
        }
    }
}
```

2. One possible solution in which we traverse the whole input vector is:

```
void rebinning(int size1, int * hist1, int size2, int * hist2) {
    int rebin = size1 / size2;
    #pragma omp parallel
    {
        int myid = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        int rem = size2 % howmany;
        int lower = (myid * size2/howmany) + (myid<rem ? myid : rem);
        int upper = lower + (size2/howmany) + (myid < rem);
        for (int i = 0; i < size1; i++) {
            int tmp = i / rebin;
            if ((tmp>=lower) && (tmp<upper))
                hist2[tmp] += hist1[i];
        }
    }
}
```

However, a much better solution is possible by computing the range of elements in hist that need to be traversed:

```
void rebinning(int size1, int * hist1, int size2, int * hist2) {
    int rebin = size1 / size2;
    #pragma omp parallel
    {
        int myid = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        int rem = size2 % howmany;
        int lower = (myid * size2/howmany) + (myid<rem ? myid : rem);
        int upper = lower + (size2/howmany) + (myid < rem);
        for (int i = rebin*lower; i < (upper+1)*rebin; i++) {
            int tmp = i / rebin;
            hist2[tmp] += hist1[i];
        }
    }
}
```

Or traversing `hist2` and make use of a second loop to traverse the corresponding `rebin` elements in `hist1`:

```
void rebinning(int size1, int * hist1, int size2, int * hist2) {
    int rebin = size1 / size2;
    #pragma omp parallel
    {
        int myid = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        int rem = size2 % howmany;
        int lower = (myid * size2/howmany) + (myid<rem ? myid : rem);
        int upper = lower + (size2/howmany) + (myid < rem);
        for (int i = lower; i < upper; i++)
            for (int j=0; j < rebin; j++) {
                int tmp = i*rebin + j;
                hist2[i] += hist1[tmp];
            }
    }
}
```

PAR – 2^{nd} In-Term Exam – Course 2015/16-Q2
June 1st, 2016

Problem 1 (3 points)
Given the following code, partially parallelized with OpenMP (assume that functions foo, foo2 and foo3 only read the values received as arguments and do not modify other positions in the memory):

```
#define N 1024
#define MIN_SIZE 16
void InitMatrix(long int *v,int size){...}

void CKJ(long int *v,int size) {
    int i, j;
    #pragma omp for
    for (i=0;i<size;i++)
        for(j=1;j<size;j++)
             v[i*size+j]=foo3(v[i*size+j])+foo3(v[i*size+(j-1)]);
}

void CXY_base(long int *v,int size) {
    for (int i=0;i<size;i++) v[i]=foo2(v[i]);
}

long int CXY(long int *v,int size) {
    if (size<=MIN_SIZE) {
        CXY_base(v,size);
    } else {
        CXY(v,size/2);
        CXY(&v[size/2],size/2);
    }
}
void main(int argh,char *argv[]) {
    long int *V;
    V=malloc(N*N*sizeof(long int));
    InitMatrix(V,N);
    #pragma omp parallel
        {
        CXY(V,N*N);
        CKJ(V,N);
        }
    fprintf(stdout, "End Computation\n");
}
```

We ask you to answer the following questions, briefly justifying (2-3 lines) your answers:

1. Add the required pragmas to complete the parallelization of the CXY function following a tree strategy and briefly justify the pragmas inserted.
2. Do you need to guarantee any dependence and/or protect any race condition between the execution of CXY and CKJ?. And during the execution of function CKJ? If affirmative in any of the two cases, how are you guaranteeing the dependence?
3. Propose a modification in the code to introduce a cut-off based on the depth of the three, defining a maximum of 4 levels. We will consider positively the introduction of the minimum number of changes in the code.

Problem 2 (4 points)

1. Implement a parallel version of the following code that follows a *Geometric Block Data Decomposition* by rows of output matrix `b`.

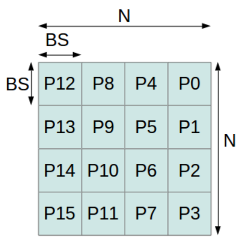
```
#define N A_VALUE_NOT_KNOWN_BIGGER_THAN_BS
#define BS A_VALUE_NOT_KNOWN
float b[N][N];

void main() {
...
int NB=((N+BS-1)/BS); // N may not be multiple of BS
for (int ii = 0; ii < NB ; ii++)
   for (int jj = 0; jj < NB ; jj++)
      for (int i = max(1,ii*BS);  i< min((ii+1)*BS,N-1); i++)
         for (int j = max(1,jj*BS); j < min((jj+1)*BS,N-1); j++)
            b[i][j] += goo(b[i-1][j],b[i+1][j],b[i][j+1]);
...
}
```

   In particular, each thread has to deal with a maximum of `BS` rows and there should be `NB` threads, where `NB` is computed in the code. You can use any OpenMP directive/clause that helps you to follow this data decomposition and take into account the possible data dependences. Note: You don't have to be worried about the load unbalance: N and `BS` are already defined, although those values are not known for this question.

2. We propose a modification of the code, introducing a new matrix variable: `a`. The new code is:

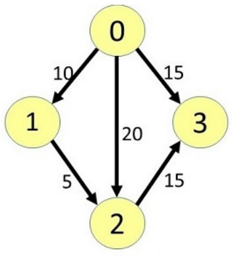
```
#define N A_VALUE_NOT_KNOWN_BIGGER_THAN_BS
#define BS A_VALUE_NOT_KNOWN
float b[N][N];

void main() {
...
int NB=(N/BS); // N is multiple of BS

for (int ii = 0; ii < NB ; ii++)
    for (int jj = 0; jj < NB ; jj++)
        for (int i = max(1,ii*BS);  i< min((ii+1)*BS,N-1); i++)
            for (int j = max(1,jj*BS); j < min((jj+1)*BS,N-1); j++)
                a[i][j] = b[i][j] + goo(b[i-1][j],b[i+1][j],b[i][j+1]);
...
}
```

   Implement a OpenMP parallel program that follows a *Data Decomposition* of output matrix `a` by blocks of $BS \times BS$ elements as shown in the following figure (example for $N/BS = 4$). In the figure, Pn corresponds to the thread number n that has to compute this block.



   Your program should follow the owner's compute rule, and the number of threads should be $NB \times NB$, where $NB = N/BS$ and N is multiple of BS. You can use any OpenMP directive/clause you want to BUT `omp for` to implement that data decomposition and take into account the possible data dependences. Note: You don't have to be worried about the load unbalance: N and BS are already defined, although those values are not known for this question.

Problem 3 (3 points)
Given two possible representations for a graph: `graphVec` (graph stored as a vector of edges, Figure (b)) and `graphList` (graph stored as a vector of nodes, each with a list of its adjacent nodes, Figure (c)) and a simple graph (Figure (a)) to illustrate them:

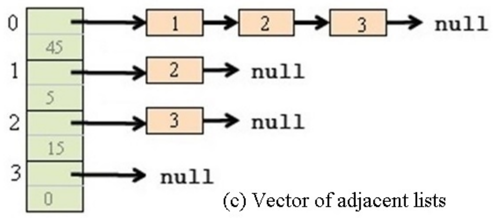


(a) Example graph

from	to	weight
0	1	10
0	2	20
0	3	15
1	2	5
2	3	15

(b) Vector of edges

```
typedef struct {
        int from, to, weight;
} tEdge;

typedef tEdge *graphVec;
```



(c) Vector of adjacent lists

```
typedef struct tnode {
        int id;
        struct tnode *next;
} node;

typedef struct {
        node *first;
        int totWeight;
} list;

typedef list *graphList;
```

We have two alternative implementations for a function `buildGraphList` that converts a graph (Figure (a)) stored using format `graphVec` (Figure (b)) into its equivalent graph stored in format `graphList` (Figure (c)).

1. The following code corresponds with the first implementation in OpenMP. The code simply iterates all edges in vector `v` and inserts each of them in the list of adjacent nodes in vector `g` using function `insList(list * l, int n)`, which inserts node `n` into the list pointed by `l`.

```
void buildGraphList (graphList g, int numnodes, graphVec v, int numedges)  {
     int i, from, to, w;

     #pragma omp parallel
     #pragma omp single
     for (i=0; i<numedges; i++) {
         #pragma omp task firstprivate (i) private(from, to, w)
         {
         from = v[i].from;
         to = v[i].to;
         w = v[i].weight;
         #pragma omp critical
             {
             // Insert node "to" in list of adjacent nodes to "from"
             insList(&g[from], to);
             // Accumulate total weight of edges coming from "from"
             g[from].totWeight += w;
             }
         }
     }
}
```

   We ask you to optimize the code in order to reduce as much as possible the overhead incurred in the parallelization and maximize the concurrency, consequently improving the overall parallel performance. For the optimization you can change the OpenMP constructs and synchronization mechanisms that are used, not the code and data structures in the baseline sequential program.

2. The following code corresponds to an alternative sequential implementation of function `buildGraphList`. In this implementation, for each node `i` in the graph (first loop) the algorithm finds out which edges `k` in

vector v (second loop), have a "from" node i. If so, it inserts the "to" node into the list of adjacent nodes of i.

We ask you to insert the proper OpenMP directives to parallelize this version providing the load is well balanced among threads. You don't need to change the code and data structures in the baseline sequential program in order to write the proper solution.

```
void buildGraphList (graphList g, int numnodes, graphVec v, int numedges) {
    int i, from, to, w;
    int k;

    for (i=0; i<numnodes; i++)
        for (k=0; k<numedges; k++) {
            from = v[k].from;
            if (from == i) {
                 to = v[k].to;
                 w = v[k].weight;
                 // Insert node "to" in list of adjacent nodes to "from"
                 insList(&g[i], to);
                 // Accumulate total weight of edges coming from "from"
                 g[i].totWeight += w;
             }
        }
}
```

Solution

Problem 1 (3 points)

1. We have to add a omp single in order to avoid the generation of tasks by all the threads. Since we are asking for a tree strategy, we must add the pragmas in the recursive part. We don't need to include a *taskwait*.

```
long int CXY(long int *v,int size){
        if (size<MIN_SIZE){
                CXY_base(v,size);
        }else{
#pragma omp task
            CXY(v,size/2);
#pragma omp task
            CXY(&v[size/2],size/2);
}}

void main(int argh,char *argv[]) {
...
#pragma omp parallel
{
#pragma omp single
CXY(..);
CKJ(...);
}}
```

2. Yes, CKJ can not start until CXY has finished its execution. The implicit barrier at the end of the single already enforces the dependence. During the execution of CKJ the dependence in the j loop is internalized in the execution of each thread, so no need to insert any synchronization for it. Variable j doesn't need to be privatized since it is declared inside the parallel and then it is already private.

3. A possible solution is to include a new argument, depth, in function CXY. The value for depth is checked in the final clause order to decide if the task is generated or not, together with the mergeable clause to minimize overheads.

```
long int CXY(long int *v,int size,int depth){
        if (size<MIN_SIZE) {
                CXY_base(v,size);
        } else {
            #pragma omp task final(depth>=4) mergeable
            CXY(v,size/2,depth+1);
            #pragma omp task final(depth>=4) mergeable
            CXY(&v[size/2],size/2,depth+1);
        }
}
void main(int argh,char *argv[]) {
...
#pragma omp parallel
{
#pragma omp single
CXY(..,0);
...
}
```

Problem 2 (4 points)

1. Parallel version that follows a *Geometric Block Data Decomposition* by rows of output matrix b.

```
#define N A_VALUE_NOT_KNOWN_BIGGER_THAN_BS
#define BS A_VALUE_NOT_KNOWN
float b[N][N];
unsigned short blocksfinished[(N+BS-1)/BS];

void main() {
...
int i;
int NB=((N+BS-1)/BS); // N is multiple of BS

#pragma omp parallel num_threads(NB)
for (i=0; i<NB; i++)
  blocksfinished[i]=0;

#pragma omp parallel for num_threads(NB)
for (int ii = 0; ii < NB ; ii++) {
   for (int jj = 0; jj < NB ; jj++) {
      if (ii!=0)
          while(jj>=blocksfinished[ii-1]) {
              #pragma omp flush
              ;
          }

      for (int i = max(1,ii*BS);  i< min((ii+1)*BS,N-1); i++)
         for (int j = max(1,jj*BS); j < min((jj+1)*BS,N-1); j++)
            a[i][j] = b[i][j] + goo(b[i-1][j],b[i+1][j],b[i][j+1]);

      blocksfinished[ii]++;
      #pragma omp flush
   }
} }
```

2. OpenMP parallel program that follows a *Data Decomposition* of output matrix a by blocks of $BS \times BS$ elements.

```
#define N A_VALUE_NOT_KNOWN_BIGGER_THAN_BS
#define BS A_VALUE_NOT_KNOWN
float b[N][N];
void main() {
...
int NB=(N/BS); // N is multiple of BS
  #pragma omp parallel num_threads(NB*NB)
  {
    int myid = omp_get_thread_num();
    int ii = (myid%NB);
    int jj = NB-1-(myid/NB);
    for (int i = max(1,ii*BS);  i< min((ii+1)*BS,N-1); i++)
        for (int j = max(1,jj*BS); j < min((jj+1)*BS,N-1); j++)
           a[i][j] = b[i][j] + goo(b[i-1][j],b[i+1][j],b[i][j+1]);
  }
}
```

Problem 3 (3 points)

1. Since load is well balanced for loop i, a for worksharing construct would be better. To maximize concurrency in the updates of the different lists, a vector or locks instead of a single critical construct should be used.

```
void buildGraphList (graphList g, int numnodes, graphVec v, int numedges) {
int i, from, to, w;
omp_lock_t graph_lock[numnodes];

        // intialize locks
        for (i=0; i<numnodes; i++) omp_init_lock(&graph_lock[i]);

        #pragma omp parallel for private(from, to, w)
        for (i=0; i<numedges; i++) {
                from = v[i].from;
                to = v[i].to;
                w = v[i].weight;

                omp_set_lock (&graph_lock[from]);
                insList(&g[from], to);
                g[from].totWeight+=w;
                #pragma omp flush
                omp_unset_lock (&graph_lock[from]);
        }

        // destroy locks
        for (i=0; i<numnodes; i++) omp_destroy_lock(&graph_lock[i]);
}
```

2. The iterations of loop `i` are totally independent and can be executed with no data sharing constraints on the list. To do a good load balancing we simply need to use a `dynamic` schedule for it.

```
void buildGraphList (graphList g, int numnodes, graphVec v, int numedges) {
int i, from, to, w;
int k;

       #pragma omp parallel for private(k, from, to, w) schedule(dynamic)
       for (i=0; i<numnodes; i++) {
            for (k=0; k<numedges; k++) {
                 from = v[k].from;
                 if (from == i) {
                     to = v[k].to;
                     w = v[k].weight;
                     insList(&g[i], to);
                     g[i].totWeight += w;
                 }
            }
       }
}
```

Part III

Final Exams

PAR: Final Exam – Course 2014/15-Q1
January 16th, 2015

Question 1 (3.5 points)
Assume a NUMA (non-uniform memory architecture) multiprocessor system composed of 4 NUMA-nodes ($node_{0:3}$), each node with 8 GBytes of main memory and 4 cores ($core_{0:3}$). Each core has only one level of cache memory of 2 MBytes (with cache lines of 64Bytes). The system includes all the necessary mechanisms (seen in class) to keep the memory coherence inside a NUMA-node and between NUMA-nodes.

1. (0.75 points) In order to support a write-invalidate MSI cache-coherence protocol within each NUMA-node, how many additional bits should be used in each cache line, and what are their role and possible values? How many bits in total per NUMA-node are used for that purpose?
2. (0.75 points) In order to support a directory-based write-invalidate MESI cache-coherence protocol between NUMA-nodes, how many bits should be used in the directory for each line in main memory, and what are their role and possible values? How many bits in total per NUMA-node are used for that purpose?
3. (1 point) Consider the following parallel program executed on only two cores (processors) inside $node_0$ of the previous multiprocessor system:

   Core 0: Update even-numbered rows

   ```
   for ( j = 0 ; j < N ; j += 2 )
      for ( k = 0 ; k < N ; k++ )
           A[j,k] = f(j,k);
   ```

   Core 1: Update odd-numbered rows

   ```
   for ( j = 1 ; j < N ; j += 2 )
      for ( k = 0 ; k < N ; k++ )
           A[j,k] = g(j,k);
   ```

   Assume that matrix `A` is stored in main memory of $node_0$, that each element of matrix `A` is 8 Bytes and that the first element of `A` is aligned on a cache line boundary. What would be the maximum number of invalidations that would be sent through the bus expressed as a function of N?
4. (1 point) If the previous program is now executed on two cores, one in $node_0$ and the other in $node_1$ and assuming the sequence of accesses that originates the maximum number of invalidation messages for $N = 4$, which coherence actions would take place during the execution of the first iteration of the outer loop in both cores?

Question 2 (1.5 points)
Given the following sequential code:

```
    for(i=0;i<N;i++)
        A[i] = i;
    for(i=0;i<N;i++)
        B[i] = A[i]*A[i];
```

and a possible parallel version:

```
#pragma omp parallel
{
    #pragma omp for schedule(runtime) nowait
    for(i=0;i<N;i++)
        A[i] = i;

    #pragma omp for schedule(runtime) nowait
    for(i=0;i<N;i++)
        B[i] = A[i]*A[i];
}
```

Answer the following questions:

1. (0.75 points) Is the parallel version correct if we define the environment variable `OMP_SCHEDULE` to `STATIC`, `(DYNAMIC,1)` or `(GUIDED,1)`? Please, justify your answer.
2. (0.75 points) Consider the situation where you don't know which schedule will be chosen by the user at runtime. Modify the code so that you can guarantee the correctness of your parallel code for any kind of schedule. Give two solutions that do not make use of explicit barriers.

Question 3 (2.5 points)
Given the following code:

```
#define SIZE_INDEX 256
#define N 1024*1024*1024
#define MAX_SIZE 32*1024

void histogram(unsigned int *S, int n, unsigned int *index) {
 unsigned int i, tmp;
 for (i=0; i<n; i++) {
   tmp = S[i]%SIZE_INDEX;
   index[tmp]++;
 }
}

void main() {
  unsigned int S[N];
  unsigned int index[SIZE_INDEX];
  ...
  // Here we have initialized S to random numbers and index to 0's
  histogram(S, N, index);
  ...
}
```

We ask you:

1. (1.5 points) Write the OpenMP parallel code for function `histogram` that follows a tree divide–and–conquer task decomposition with the objective of achieving a $T_\infty = log_4(n)$. Your parallel code should include a cut–off mechanism based on the current size of the vector `S`, i.e. the value of parameter n becoming lower than `MAX_SIZE`.
2. (1 point) Given the values defined for `MAX_SIZE` and `N`, considering that t_c is the computation time of a loop iteration in function `histogram`, and that the rest of the computation is negligible, determine the value of T_1, T_∞, `Parallelism` and P_{min} for the parallel strategy you have implemented (with cut–off).

Question 4 (2.5 points)
Write two different OpenMP parallel implementations of function `histogram` (previous question) using the following strategies:

1. (1.5 points) Block Data Decomposition of the Input vector `S`.
2. (1 point) Block Data Decomposition of the Output vector `index`.

Important: Both implementations should minimize the use of synchronizations. You can use any OpenMP pragma or function you may need.

Solution

Question 1 (3.5 points)

1. (0.75 points)

   We need 2 additional bits per cache line in order to store the state of the cache line. Those 2 bits can code up to four possible states:

   - I invalid (uncached, not valid in any cache)
   - S shared (two or more nodes may have copies)
   - M modified (dirty)
   - Note that a fourth state can be coded with 2 bits. However, it is not necessary for the MSI protocol.

   The number of cache lines is calculated dividing the memory in a cache (2MB) by the cache line size (64B). Therefore: Number of cache lines $= \frac{2MB}{64B} = 2^{15} = 32K$ entries

   We need 2 bits per entry. And there are a total of 4 cache memories per NUMAnode.

   Thus, the total number of bits amounts to:

   $\frac{2MB}{64B}$ entries/cache $\times 2$ bits/entry $\times 4$ caches/NUMAnode $= 2^{18}$ bits $= 256K$ bits/NUMAnode

2. (0.75 points)

   The home NUMA-node is in charge of the coherence of its physical memory lines by means of the directory entries. A directory entry stores the line state and the identities of other NUMA-nodes sharing this memory line.

   Bits per directory entry:

   - Presence bits (nodes currently having the line), 1 bit per NUMA-node: i.e. 4 bits
   - State bits (to track the state of cache lines): 2 bits

     Those 2 bits can code up to four possible states:

     – I invalid (uncached, not valid in any cache)
     – S shared (two or more nodes may have copies)
     – M modified (dirty)
     – E exclusive (only this node has a copy, but not modified)

   Total: 6 bits per directory entry.

   The number of entries in the directory structure per NUMA-node is calculated dividing the memory in a NUMA-node (8GB) by the cache line size (64B). Therefore:

   Number of Directory Entries $= \frac{8GB}{64B} = 2^{27} = 128$ Mega entries

   Thus, with 6 bits per directory entry, the total number of bits per NUMAnode will be:

   $\frac{8GB}{64B}$ entries/NUMAnode $\times 6$ bits/entry $= 3 \times 2^{28} = 768$ Mega bits/NUMAnode

3. (1 point)

   Let us consider the general case where N is large. The number of elements of A that fit in a cache line is 8. When $N\%8\ != 0$ the last few elements in a row can fall in the same cache line as those at the beginning of the next row, yielding false sharing and the corresponding invalidations. On average, the number of invalidations for every pair of lines will be 2. Having N rows, the number of invalidations will be upper bounded by $2N$. Note however that, for small N, the situation is different. For instance, for $N = 4$, every two rows fall in the same cache line, causing up to 7 invalidations for every pair of lines. Thus, for $N = 4$, the number of invalidations is $7N/2$.

4. (1 point)

   For $N = 4$ and with the first element of `A` aligned on a cache line boundary, the first two rows fall in the same cache line. Let us assume an alternate sequence of accesses from some processor (P0) in $node_0$, and some processor (P1) in $node_1$, starting from P0. Having alternate accesses to those rows from both cores will cause up to 7 invalidations.

   Matrix `A` is stored in main memory of $node_0$. Let us assume all caches are initially empty. P0 performs a PrWr, which generates a BusRdX transaction in the bus. Having no other copies of the cache line neither locally in the `home` node, i.e. $node_0$, nor in $node_1$, there are no further things to do other than the following: the directory entry in the `home` node is modified to account for the copy of the line in $node_0$ (presence bit is set), with a modified state (M); the line is sent through the local bus in $node_0$ and the snoop in P0 will keep a copy in its cache, with an associated modified state (M). It can then be written from P0.

If P1 then issues a PrWr to the same line, this will introduce a BusRdX in the bus. The hub in *node*$_1$ redirects the memory request (RdXReq) to the hub of the `home` node, i.e. *node*$_0$, which will return to *node*$_1$ the list of nodes which have a copy: *node*$_0$ in this case. Then, *node*$_1$ will send an invalidation request to the hub of *node*$_0$. The hub in *node*$_0$ will issue a BusRdX in the local bus, and the snoopy of P0 will detect such request and will flush the modified line updating memory in *node*$_0$. Next, it will invalidate the local copy in P0. The presence bit in the directory entry for this line corresponding to *node*$_0$ will be reset, and the one for *node*$_1$ will be set. The directory entry will continue to have the modified state (M). The line is then sent to the hub of *node*$_1$, which will write it in its local bus. The snoop in P1 will keep a copy in its cache, with an associated modified state (M), and can then be written from P1.

If P0 then issues a PrWr to the same line, this will introduce a BusRdX in the bus. The directory entry indicates that there is a modified copy in *node*$_1$. Thus, the hub in *node*$_0$ redirects the memory request (RdXReq) to the hub of *node*$_1$, which will issue a BusRdX in the local bus. The snoopy of P1 will detect such request and will flush the modified line and will mark as invalid the local copy in P1. The hub in *node*$_1$ will forward the modified line to the hub in *node*$_0$, which will in turn update memory in *node*$_0$. The presence bit in the directory entry for this line corresponding to *node*$_1$ will be reset, and the one for *node*$_0$ will be set. The directory entry will continue to have the modified state (M). The line is then sent through the local bus in *node*$_0$. The snoop in P0 will keep a copy in its cache, with an associated modified state (M), and can then be written from P0.

The sequence of coherence actions will be similar for the subsequent alternate accesses from P1 and P0.

Question 2 (1.5 points)

1. (0.75 points)

   `STATIC`: The parallel version using this policy is correct. The iterations of the loops are distributed among the threads in a way that a thread is in charge of the same range of values for `i`, so this ensures there will be no conflicts as each thread will update and read just the same elements of `A[]` at the first and second loop.

   `(DYNAMIC,1)` and `(GUIDED,1)` versions are NOT correct. Due to the fact that the loops are `no wait` and the assignment of work to threads is dynamic, the parallel execution cannot guarantee that same thread will update and later read the value of `A[i]` in sequence. Other threads can try to read `A[i]` at the second loop even before the first loop has written a value for it.

2. (0.75 points)

There are many ways to ensure that the three possible schedules are handled with correctness by modifying the code to make it predictable. As examples:

1. Remove the `nowait` from first loop (optionally at second loop, too)

```
#pragma omp parallel
{
    #pragma omp for schedule(runtime)
    for(i=0;i<N;i++)
        A[i] = i;

    #pragma omp for schedule(runtime)
    for(i=0;i<N;i++)
        B[i] = A[i]*A[i];
}
```

2. Collapse both parts

```
#pragma omp parallel
{
    #pragma omp for schedule(runtime) nowait
    for(i=0;i<N;i++)
    {
        A[i] = i;
        B[i] = A[i]*A[i];
    }
}
```

3. Realize that `A[i]` and `i` will be equivalent after first loop, so use it instead.

```
#pragma omp parallel
{
    #pragma omp for schedule(runtime) nowait
    for(i=0;i<N;i++)
        A[i] = i;

    #pragma omp for schedule(runtime) nowait
    for(i=0;i<N;i++)
        B[i] = i*i;
}
```

Question 3 (2.5 points)

1. (1.5 points)

```
#define SIZE_INDEX 256
#define N 1024*1024*1024
#define MAX_SIZE 32*1024

void histogram(unsigned int *S, int n, unsigned int *index) {
 unsigned int i, tmp;

 if (n<1) return;

 if (n==1) {
    tmp = S[0]%SIZE_INDEX;
    #pragma omp atomic
    index[tmp]++;
 }
 else {
    unsigned int n4=n/4;

    #pragma omp task final(n<MAX_SIZE) mergeable
    histogram(S, n4, index);

    #pragma omp task final(n<MAX_SIZE) mergeable
    histogram(S+n4, n4, index);

    #pragma omp task final(n<MAX_SIZE) mergeable
    histogram(S+n4+n4, n4, index);

    #pragma omp task final(n<MAX_SIZE) mergeable
    histogram(S+n4+n4+n4, n-(n4+n4+n4), index);
 }

}

void main() {
  unsigned int S[N];
  unsigned int index[SIZE_INDEX];
  ...
  // Here we have initialized S to random numbers and index to 0's

  #pragma omp parallel
  #pragma omp single
  histogram(S, N, index);
  ...
}
```

2. (1 point)

   - We have to do all the iterations one after the other:
     $T_1 = N \times t_c$

- For T_∞ only the cost of the leaves of the tree parallel execution has to be into acount; the rest is negligible. Indeed, as all the leaves can be done in parallel by definition of T_∞, only the cost of one leaf should be considered. Any of the leaves has $n = 2^{14}$ elements, after 8 levels of recursive calls to obtain a n smaller than MAX_SIZE.
  $T_\infty = \frac{2^{30}}{4^8} \times t_c = \frac{2^{30}}{2^{16}} \times t_c = 2^{14} \times t_c$
- Parallelism:
  $\mathit{Parallelism} = T_1/T_\infty = \frac{2^{30} \times t_c}{2^{14} \times t_c} = 2^{16}$
- Pmin. We need as many as number of leaves.
  $P_{min} = 4^8 = 2^{16}$

Question 4 (2.5 points)

1. (1.5 points) Block Data Decomposition of the Input vector S.

```
#define SIZE_INDEX 256
#define N 1024*1024*1024
#define MAX_SIZE 32*1024
...

void histogram(unsigned int *S, int n, unsigned int *index)
{
 unsigned int i, tmp;

 #pragma omp parallel private(i)
 {
  unsigned int local_index[SIZE_INDEX];
  for (i=0;i<SIZE_INDEX;i++)
   local_index[i]=0;

  #pragma omp for private(tmp)
  for (i=0;i<n;i++) {
     tmp = S[i]%SIZE_INDEX;
     local_index[tmp]++;
  }

  #pragma omp critical
  for (i=0;i<SIZE_INDEX;i++)
   index[i] += local_index[i];

 }
}
```

2. (1 point) Block Data Decomposition of the Output vector index.

```
#define SIZE_INDEX 256
#define N 1024*1024*1024
#define MAX_SIZE 32*1024
...

void histogram(unsigned int *S, int n, unsigned int *index)
{
 unsigned int i, tmp;

 #pragma omp parallel private(i, tmp)
 {
    int id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    int start_index = id * (SIZE_INDEX/num_threads);
    int end_index = (id+1) * (SIZE_INDEX/num_threads);
    if (id==num_threads-1) end_index = SIZE_INDEX;
 for (i=0;i<n;i++) {
   tmp = S[i]%SIZE_INDEX;
   if (tmp>=start_index && tmp<end_index)
     index[tmp]++;
 }
}

}

...
```

PAR – Final Exam – Course 2014/15-Q2
June 12th, 2015

Question 1: (3.0 points)
Answer the questions below, given de following code in C:

```
#define MIN_REC_SIZE (1<<10)

void it_min_max(int *X, int *min, int *max, int n) {
   int i, lmin=MAX_INT, lmax=MIN_INT;

   for (i=0; i<n; i++) {
     if (lmin>X[i]) lmin=X[i];
     if (lmax<X[i]) lmax=X[i];
   }
   *min=lmin; *max=lmax;
}

void rec_min_max(int *X, int *min, int *max, int n) {

   int ndiv2 = n/2, min1, min2, max1, max2;

   if (n<=MIN_REC_SIZE) it_min_max(X,min,max,n);
   else {
      rec_min_max(X, &min1, &max1, ndiv2);
      rec_min_max(X+ndiv2, &min2, &max2, n-ndiv2);

      if (min1<min2) *min=min1;
      else *min=min2;
      if (max1<max2) *max=max2;
      else *max=max1;
   }
}

int main() {
   int X[N];
   int min, max;
   ...
   ???_min_max(X,&min, &max, N);
   ...
}
```

1. (1 points) Implement a parallel version of function `it_min_max`, that avoids false sharing and any synchronization within the loop, **using a Linear (Iterative) Task Decomposition strategy**. Complete the main program assuming that you call `it_min_max` function.
2. (2 points) Implement a parallel version of function `rec_min_max` using a **Tree Recursive Task Decomposition strategy that takes into account the overhead of creating and synchronization of tasks**. Use a CUTOFF (`MAXDEPTH`) based on the recursion depth. Complete a different main program assuming that you call `rec_min_max` function.

Question 2: (1.0 point)
Assuming that a correct answer is (0.2 points) and an incorrect answer is (-0.1 points). Indicate if the following afirmations are true or false:

1. The NUMA architecture relies on load-linked and store-conditional operations in order to access data that is stored in remote NUMAnodes to avoid the consistency problem when accessing to different data values.
2. Test-and-set instructions can be used to protect a critical region using a waiting loop thanks to the atomicity between the test and the set accessing to main memory.
3. Using test-test-and-set technique to implement a lock reduces the memory and bus contention thanks to the exploitation of the temporal locality of the lock data.

4. Snoopy hardware can only support the MSI cache coherence protocol.
5. Memory directories in NUMA systems have as many entries as number of cache lines in the overall system, helping to identify which caches have a copy of a memory line.

Question 3 (4.0 points)
We want to implement a OpenMP parallel version of this code using Block Data Decomposition by rows having in mind the synchronization costs. Note that blocks are assigned starting by thread 0, and this thread 0 is assigned to block 0 of rows, which starts at row 0 of the matrix u. You are not allowed to use neither `#pragma omp parallel for` nor `#pragma omp for`. The number of threads will be defined using the `OMP_NUM_THREADS` and it is not guaranteed that it perfectly divides `N`. Please, ensure a proper load balance of the data distribution: no more than 1 row of extra-work.

```
#define N 1024
...
int i, k;
for (i=N-2; i>=0; i--) {
    for (k=N-1; k>0; k--) {
        u[i][k] -= u[i+1][k] + u[i][k-1];
    }
}
```

Question 4 (2.0 points)
Having in mind the strategy of parallelization of function `it_min_max` and the strategy of parallelization of `rec_min_max` with NO `CUTOFF` in the first question of the exam, and considering the folloging costs:

- Each level of parallel recursivity, except the last, of parallel function `rec_min_max` has a fixed cost of C_{rec}. Last level of the recursivity (base case) has a fixed cost of C_{if} plus the cost of function `it_min_max`.
- The cost of one loop iteration of either sequential or parallel version of `it_min_max`, with no synchronization inside the loop is C_{iter}
- The overhead of computing the global min/max of the parallel version of function `it_min_max`, which depends on the number of processors P used, is $K \log P$

Answer the following questions, for $n = (1 << 20)$ and p number of processors :

1. Draw a scheme of the task dependency graph and calculate T_1, T_∞ and P_{min} for the parallel strategy of function `rec_min_max`.
2. Which are T_1, S_p for the parallel implementation of the parallel strategy of function `it_min_max` in a system with p processors, considering that $n\%p = 0$?

Solution

Question 1: (3.0 points)

1. Linear Task Decomposition strategy of `it_min_max` function. Main program remains exactly the same, calling to `it_min_max` function.

```
#define MIN_REC_SIZE (1<<10)

void it_min_max(int *X, int *min, int *max, int n) {
   int i, lmin=MAX_INT, lmax=MIN_INT;

   #pragma omp parallel for reduction(min:lmin) reduction(max:lmax)
   for (i=0; i<n; i++) {
     if (lmin>X[i]) lmin=X[i];
     if (lmax<X[i]) lmax=X[i];
   }
   *min=lmin; *max=lmax;
}

int main() {
   int X[N];
   int min, max;
   ...
   it_min_max(X,&min, &max, N);
   ...
}
```

2. Tree Recursive Task Decomposition strategy, with CUTOFF based on the recursion depth.

```
#define MIN_REC_SIZE (1<<10)

void it_min_max(int *X, int *min, int *max, int n) {
   int i, lmin=MAX_INT, lmax=MIN_INT;

   for (i=0; i<n; i++) {
     if (lmin>X[i]) lmin=X[i];
     if (lmax<X[i]) lmax=X[i];
   }
   *min=lmin; *max=lmax;
}

void rec_min_max(int *X, int *min, int *max, int n, int d) {

   int ndiv2 = n/2, min1, min2, max1, max2;

   if (n<=MIN_REC_SIZE) it_min_max(X,min,max,n);
   else {

      #pragma omp task shared(min1, max1) final(d>MAXDEPTH) mergeable
      rec_min_max(X, &min1, &max1, ndiv2, d+1);

      #pragma omp task shared(min2, max2) final(d>MAXDEPTH) mergeable
      rec_min_max(X+ndiv2, &min2, &max2, n-ndiv2, d+1);

      #pragma omp taskwait

      if (min1<min2) *min=min1;
      else *min=min2;
      if (max1<max2) *max=max2;
      else *max=max1;
   }
```

```
    }

    int main() {
       int X[N];
       int min, max;
       ...
       #pragma omp parallel
       #pragam omp single
       rec_min_max(X,&min, &max, N, 0);
       ...
    }
```

Question 2: (1.0 point)

1. False
2. True
3. True
4. False
5. False

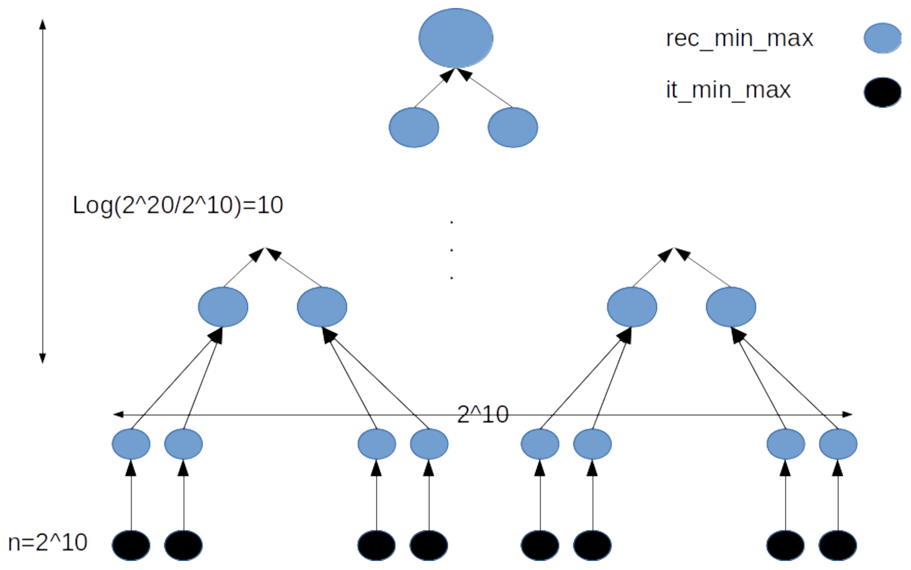
Question 3 (4.0 points)

```
#define N 1024
#define BS 128                  // Block size to split the rows
#define MT 512                  // MAX threads

  int u[N][N];
  int BR = N / BS;              // Blocks per row
  int blockfinished[MT];
  for (int i = 0; i < MT; i++)
    blockfinished[i] = BR-1;

  #pragma omp parallel
  {
    int myid = omp_get_thread_num ();
    int nt = omp_get_num_threads ();
    int nelem = N / nt;
    int rem = N % nt;

    int i_start = myid * nelem + (myid < rem ? myid : rem);     // First row current thread
    int i_end = i_start + nelem + (myid < rem) - 1;     // Last row current thread

    if (myid == nt - 1) i_end = i_end - 1;

    for (int block = BR - 1; block >= 0; block--)
      {
        int k_start = block * BS;       // First column actual block
        int k_end = (block < BR - 1) ? k_start + BS : N - 1; // Last column actual block

        if (myid < nt-1)
            while (blockfinished[myid + 1] >= block) {
            #pragma omp flush
            }

        for (int i = i_end; i >= i_start; i--) {
            for (int k = k_end; k > k_start; k--) {
                u[i][k] -= u[i + 1][k] + u[i][k - 1];
             }
          }

        blockfinished[myid]--;
```

```
            #pragma omp flush
        }
    }
```

Question 4 (2.0 points)

1. The scheme of the data dependency graph is the following:

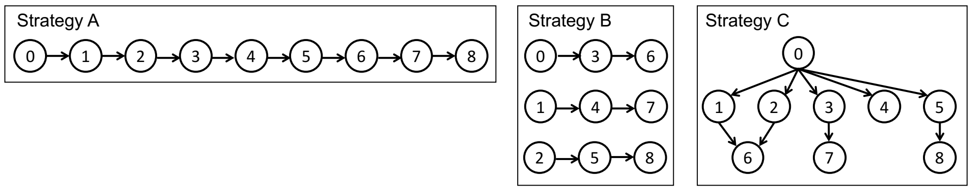


$T_1 = 1023 \times C_{rec} + 1024 \times (C_{if} + 1024 \times C_{iter})$
$T_\infty = 10 \times C_{rec} + (C_{if} + 1024 \times C_{iter})$
$P_{min} = 1024$

2. $T_1 = 2^{20} \times C_{iter}$, $T_p = T_1/p + K \log p$, $S_p = T_1/T_p$

PAR – Final Exam – Course 2015/16-Q1
January 15th, 2016

Problem 1 (1.5 points)
Given the following task dependency graphs for three different parallelization strategies of a sequential code:



Answer the following questions:

1. Compute T_1, T_∞ and `Parallelism` for each one of the three dependency graphs assuming that the cost of executing each task is 1 time unit.
2. Assuming a multiprocessor with $P = 3$ processors and the following mapping of tasks to processors for each strategy:

   - **Strategy A:** $P0 \leftarrow \{0, 1, 2, ..., 8\}; P1 \leftarrow \{\}; P2 \leftarrow \{\}$.
   - **Strategy B:** $P0 \leftarrow \{0, 3, 6\}; P1 \leftarrow \{1, 4, 7\}; P2 \leftarrow \{2, 5, 8\}$.
   - **Strategy C:** $P0 \leftarrow \{0, 1, 2\}, P1 \leftarrow \{3, 4, 6\}; P2 \leftarrow \{5, 7, 8\}$.

   Calculate the general expression for $T_{P=3}$ for each strategy and associated mapping, using the data sharing model explained in class based on the distributed memory architecture with message passing. For this model, assume that the computation time for each task is t_c, that each dependence represents the access to one element, and that the access time to remote data is determined by $t_{comm} = t_s + m \times tw$, being t_s and t_w the start-up time and the sending time of an element, respectively, and being m the size of the message measured as the number of elements. You can also assume that $t_c >> t_s + t_w$.

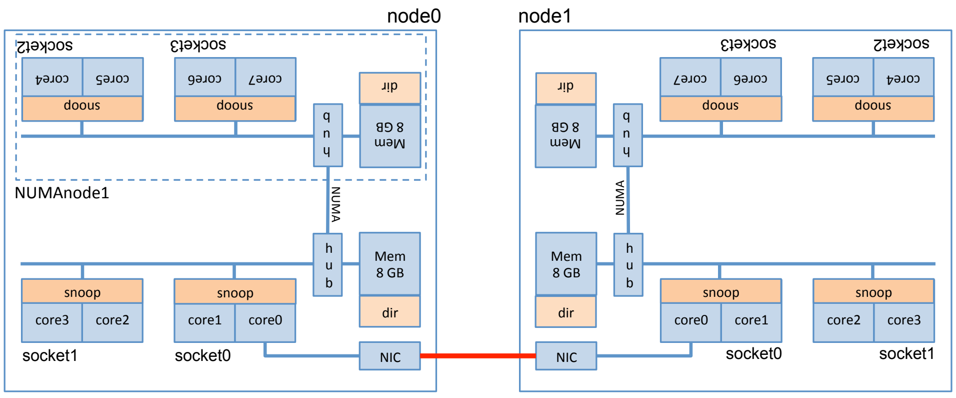
Problem 2 (1.5 points)
Assume a multiprocessor architecture with 4 processors. Consider the following `for` loop that initialises to zero the upper triangle of a 100x100 matrix `mat` (notice that the schedule clause has been left undefined):

```
float mat[100][100];
int i, j;
...
#pragma omp parallel for private(j) schedule(...) num_threads(4)
for (i = 0; i < 99; i++)
   for (j = i+1; j < 100; j++)
       mat[i][j] = 0.0;
```

We ask: Rank the following schedule clauses from slowest to fastest in terms of execution time and compute their ideal execution time: `(static)`, `(static, 10)` and `(dynamic, 20)`. In order to compute the ideal execution time please consider:

- Since a simple assignment is done in the body of the innermost loop, you can ideally estimate the execution time by counting how many assignments each thread does.
- To compute the number of iterations in a chunk of iterations you can use the expression that determines the sum of the members of a finite arithmetic progression: $n \times \frac{(a_1+a_n)}{2}$, being a_1 and a_n the first and last element of the progression and n the number of elements in the progression; for example, the total number of iterations when executing the two loops above is the sum of the sequence 99, 98, 97, ..., 2, 1 (i.e. the number of iterations of the `j` loop for each iteration of the `i` loop), that is $99 \times ((99 + 1)/2) = 4950$.

Problem 3 (1.5 points)
Given the following distributed-memory multiprocessor:



coreX: Núcleo.
socketY: Empaquetado con 2 núcleos, 1 MB de cache compartida entre ambos núcleos y snoopy.
NUMAnodeW: conjunto de 2 sockets conectados a un mismo "hub"/directorio NUMA con 8 GB de memoria principal.
nodeZ: Nodo con 2 NUMAnodes. Cada nodo incluye un NIC (Network Interface Card, Ethernet).

Answer the following questions, reasoning your answers appropriately:

1. Assume that variable A resides in a memory line that is mapped to the memory of *NUMAnode0* inside *node1*, and that at a certain moment there is an exclusive dirty copy of this memory line in the cache of *core7* of *NUMAnode1* inside *node1*. Describe the information (number of bits, type of information, and which information) is stored in both *node0* and *node1* to keep the coherence of variable A. Clearly indicate which is the coherence protocol used and the hardware support needed to keep the coherence.
2. Assuming the previous scenario, which will be the CPU operations and bus transactions that are needed to keep the coherence of variable A if *core2* of *NUMAnode0* inside *node1* wants to modify its value?.
3. Assuming the SAME INITIAL scenario, enumerate the CPU operations and bus transactions performed to keep the coherence of variable A if *core2* of *NUMAnode0* inside *node0* wants to write variable *A*.

Problem 4 (2.5 points)

In the *maximum subarray sum* problem you are given an array with integer (negative and positive) numbers. You need to compute the set of contiguous numbers that maximizes the sum of them. For example, given A = {-2,-5,7,-2,-3,1,5,-8}, the subarray A[2,6] = {7,-2,-3,1,5} has a sum of 8 and maximizes the sum of the elements of any other subarray. A divide-and-conquer algorithm that solves this problem is expressed in the following sequential recursive code:

```
// Utility functions to find maximum
int max2(int a, int b) { return (a > b)? a : b; }
int max3(int a, int b, int c) { return max2(max2(a, b), c); }

// Find the maximum possible sum in arr[left..right] such that arr[mid] is part of it
int maxCrossingSum(int * arr, int left, int mid, int right);

// Returns sum of maximum sum subarray in arr[left..right]
int maxSubArraySum(int arr[], int left, int right) {
   if (left==right) return arr[left]; // Base Case: Only one element

   // Find middle point
   int mid = (left + right)/2;
   /* Return maximum of following three possible cases
      a) Maximum subarray sum in left half
      b) Maximum subarray sum in right half
      c) Maximum subarray sum such that the subarray crosses the midpoint mid */
   return max3(maxSubArraySum(arr, left, mid), maxSubArraySum(arr, mid+1, right),
                     maxCrossingSum(arr, left, mid, right));
}

void main() {
   int * arr;
   int n = fill_vector(arr);
   int max_sum = maxSubArraySum(arr, 0, n-1);
   printf("Maximum contiguous sum is %d\n", max_sum);
}
```

We ask you to write a parallel OpenMP version of the sequential program above using a tree task decomposition parallel strategy, with a cut-off mechanism based on recursion depth.

Problem 5 (2 points)
Given the following iterative task decomposition strategy implemented in OpenMP for a loop computing the evolution of the forces in a N-body gravitational problem.

```
#define tmax 10000
#define NUM_BODIES 1048576
double forces[NUM_BODIES];
// definition of vector of struct bodies and implementation of the function to compute the
// force on a body b due to its neighbour bodies n are not relevant to do this exercise
double compute_force (theBody * b, neighbours * n);

void main() {
  ...
  for (int timestep = 0; timestep < tmax; timestep++) {
    #pragma omp parallel for schedule(dynamic,1)
    for (int body = 0; body < NUM_BODIES; body++)
      force[body] += compute_force (bodies[body].data, bodies[body].first);
  }
  ...
}
```

Due to the heavily unbalanced nature of the problem (number of neighbours for each body may differ a lot), the programmer has chosen a dynamic schedule for the execution of iterations in the parallel loop, which then leads to high scheduling overheads, lack of locality when accessing to the data associated to the problem in the repetitive instances of the parallel loop (`timestep` loop repeated `tmax` times) and false sharing when writing to vector `force`. To address these problems, the programmer has proposed the following not complete code implementing a data decomposition strategy:

```
#define tmax 10000
#define NUM_BODIES 1048576
double forces[NUM_BODIES];
// definition of vector of struct bodies and implementation of the function to compute the
// force on a body b due to its neighbour bodies n are not relevant to do this exercise
double compute_force (theBody * b, neighbours * n);

... // incomplete code a): define data structure to pass information
    // between inspector and executor

void main () {
  ...
  // INSPECTOR PHASE
  #pragma omp parallel for schedule(dynamic, ....) //incomplete chunk size b)
  for (int body = 0; body < NUM_BODIES; body++) {
    ... = omp_get_thread_num(); // incomplete code c)
    forces[body] += compute_force (bodies[body].data, bodies[body].first);
  }
  // EXECUTOR PHASE
  #pragma omp parallel
  {
    ... // incomplete code d)
    for (int timestep = 1; timestep < tmax; timestep++) {
      for (int body = 0; body < NUM_BODIES; body++) {
        if (...)  // incomplete code e)
          forces[body] += compute_force (bodies[body].data, bodies[body].first);
      }
      ... // incomplete code f)
    }
  }
  ...
}
```

in which a first iteration of the `timestep` loop is executed to dynamically obtain the assignment of iterations (and data) to threads that produces a balanced execution (inspector); after that follows the execution of the rest of iterations of the `timestep` loop using the same data decomposition obtained during the inspector phase. **We ask** you to complete the code sections (named a–f) in the code above in order to fully implement the proposed parallelization strategy (you don't need to repeat all the code above in your answer).

Problem 6 (1 point)
Assume the following OpenMP program to execute two tasks:

```
omp_lock_t lock_a, lock_b;
int ball_1=0, ball_2=1;

int main(){
  ... // lock initialization
  #pragma omp parallel num_threads(2)
  #pragma omp single
  {
    #pragma omp task // Task_1
    for (int i = 0 ; i< 10000 ; i++){
      omp_set_lock(&lock_a); omp_set_lock(&lock_b);
      if (ball_1==1) { printf("Hello! Task_1 throws ball to Task_2\n"); ball_2=1; ball_1=0;
      } else printf("Bye!, Task_1 has no ball to throw\n");
      #pragma omp flush
      omp_unset_lock(&lock_a); omp_unset_lock(&lock_b);
    }
    #pragma omp task // Task_2
    for (int i = 0 ; i< 10000 ; i++){
      omp_set_lock(&lock_b); omp_set_lock(&lock_a);
      if (ball_2==1) { printf("Hello! Task_2 throws ball to Task_1\n"); ball_1=1; ball_2=0;
      } else printf("Bye!, Task_2 has no ball to throw\n");
      #pragma omp flush
      omp_unset_lock(&lock_b); omp_unset_lock(&lock_a);
    }
  }
  ... // lock destruction
}
```

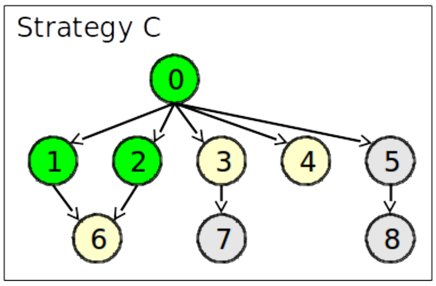
Answer the following questions, appropriately justifying your answers:

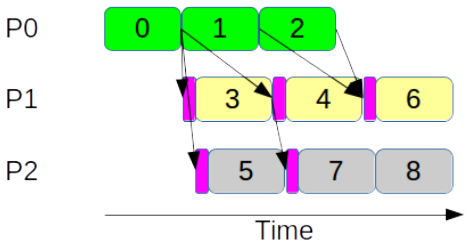
1. If the program is executed on a parallel architecture with 2 processors, is there any possible execution problem in this code? What is the minimum and the maximum number of times that each task can print out its "Hello ..." message? How would you solve the execution problem, if any? (You don't necessarily have to write the resulting code, just explain).
2. Why the programmer has introduced `#pragma omp flush` in the execution of each task?

Solution

Problem 1

1. For each parallel strategy we have:
   (a) **Strategy A:** $T_1 = 9$, $T_\infty = 9$, `Parallelism`$= 1$.
   (b) **Strategy B:** $T_1 = 9$, $T_\infty = 3$, `Parallelism`$= 3$.
   (c) **Strategy C:** $T_1 = 9$, $T_\infty = 3$, `Parallelism`$= 3$.
2. For each parallel strategy we have:
   (a) **Strategy A:** $T_P = 9 \times t_c$.
   (b) **Strategy B:** $T_P = 3 \times t_c$.
   (c) This figure shows the timing of the communications and task execution for the given mapping to processors. Therefore, for **Strategy C:** T_P is equal to $3 \times (t_s + t_w) + 4 \times t_c$.





Problem 2
The ranking is : `(static)`, `(dynamic, 20)` and `(static, 10)`
For the schedule(static) clause, each thread gets 25 iterates (except for one thread that gets 24). Thread 0 will get the first chunk of iterates, which has the most work to do. The execution time would be $25 \times ((99+75)/2) = 2175$.
For the schedule(dynamic, 20) clause, there are 5 chunks of iterates and each chunk gets 20 iterates (except for one chunk that gets 19). One thread (but we don't know which one) will get the first chunk of iterates, which has the most work to do; another thread will get two chunks of iterates, which are the two shortest chunks, which together $(20 \times ((39+20)/2) + 19 \times ((19+1)/2) = 780)$ take less time than the first one $(20 \times ((99+80)/2) = 1790)$.
For the schedule(static, 10) clause, there are 10 chunks of iterates and each chunk gets 10 iterates (except for one chunk that gets 9). Thread 0 will get three chunks of iterates, the first, fifth, and ninth chunks, and thread 0 will be the longest running thread. (Thread 1 will also get three chunks, and threads 2 and 3 will each get two chunks.) The execution time would be $(10\times((99+90)/2))+(10\times((59+50)/2))+(10\times((19+10)/2)) = 945+545+145 = 1635$.

Problem 3

1. The coherence protocol used is the `MSI` coherence protocol, inside and between NUMAnodes. Regarding to the hardware support we need, and the type and which information is stored for variable A we have:
   - Inter-NUMAnode Hardware support: Directory to keep de information of coherence information for each memory line at each NUMAnode memory. The information per memory line is: 2 bits for the coherence states ("M"odified,"S"hared,"I"nvalid) and 2 bits to keep the information of presence of copies in the two NUMAnodes in a node. There is only two bits (two NUMAnodes in a node) since the coherence information is only maintained inside one node (shared memory) and not between nodes (distributed memory). For the case of this variable A, the memory line entry mapped to the directory of NUMAnode0 will have state "M" stored in the coherence bits, and "1" value in the NUMAnode1 presence bit, and "0" in the NUMAnode0 presence bit; telling that there is a dirty copy in that NUMAnode.
   - Intra-NUMAnode Hardware support: a snoop hardware support per cache memory (socket) to mantain and update the coherence state bits (2 bits) for each cache line. For the case of this variable A, the cache memory associated to the core7 of NUMAnode1 will have state "M" for the coherence bits, telling that this line is modified (dirty).
2. Core2 does PrWr and its snoop performs a bus transaction: BusRdX. Local hub is the same as the home hub (NUMAnode 0). The local/home hub looks at the directory and figures out that NUMAnode1 is the Owner hub (it has the only dirty copy). Local hub sends Invalidate to Owner hub. Owner hub sends a

bus transaction BusRdX inside NUMAnode1. Core7 snoop forces a flush of the cache memory copy and invalidates it. Owner hub sends the dirty copy (Data) to Local/Home hub indicating that it has invalidated the cache line. Local/Home hub flushes data inside NUMAnode0, and updates directory to indicate a dirty copy ("M" state) with presence bit "1" in NUMANode0 (and therefore presence bit "0" in NUMANode1). Core2 snoop updates the data and the coherence information of its cache memory.

3. There is not operations neither bus transactions since there is not memory coherence maintenance between nodes that do not shared memory.

Problem 4

```
#define CUTOFF 3

// Returns sum of maximum sum subarray in arr[left..right]
int maxSubArraySum(int arr[], int left, int right, int depth) {
   int tmp1, tmp2, tmp3;
   if (left==right) return arr[left]; // Base Case: Only one element

   // Find middle point
   int mid = (left + right)/2;
   /* Return maximum of following three possible cases
      a) Maximum subarray sum in left half
      b) Maximum subarray sum in right half
      c) Maximum subarray sum such that the subarray crosses the midpoint mid */
   #pragma omp task shared(tmp1) final(depth>=CUTOFF) mergeable
   tmp1 = maxSubArraySum(arr, left, mid, depth+1);
   #pragma omp task shared(tmp2) final(depth>=CUTOFF) mergeable
   tmp2 = maxSubArraySum(arr, mid+1, right, depth+1);
   #pragma omp task shared(tmp3) final(depth>=CUTOFF) mergeable
   tmp3 = maxCrossingSum(arr, left, mid, right);
   #pragma omp taskwait
   return max3(tmp1, tmp2, tmp3);
}


void main() {
   int * arr;
   int n = fill_vector(arr);
   #pragma omp parallel
   #pragma omp single
   int max_sum = maxSubArraySum(arr, 0, n-1, 0);
   printf("Maximum contiguous sum is %d\n", max_sum);
}
```

Problem 5

```
// code a): vector to remember who has computed each body
int who[NUM_BODIES];
#define CACHE_LINE_SIZE 8 // Number of integers in a cache line

void main () {
  ...
  // INSPECTOR PHASE
  // b) chunk size for schedule clause, avoiding false sharing in access to who and forces
  #pragma omp parallel for schedule(dynamic, CACHE_LINE_SIZE)
  for (int body = 0; body < NUM_BODIES; body++) {
    // code c): remember which thread computed each element of forces
    who[body] = omp_get_thread_num();
    forces[body] += compute_force (bodies[body].data, bodies[body].first);
  }
  // EXECUTOR PHASE
  #pragma omp parallel
  {
    // code d): just once outside both loops to avoid overheads
    int whoamI = omp_get_thread_num();
    for (timestep = 1; timestep < tmax; timestep++) {
      for (int body = 0; body < NUM_BODIES; body++) {
        // code e): check if body owned by the thread
        if (who[body] == whoamI)
          forces[body] += compute_force (bodies[body].data, bodies[body].first);
      }
      // code f): barrier needed between consecutive iterations of timestep loop
      #pragma omp barrier
    }
  }
}
```

Note: the solution could be improved in order to store in vector `who` as many elements as chunks, that is `(NUM_BODIES/CACHE_LINE_SIZE)`. A modulo `body%NUM_BODIES` should be used when accessing vector `who`.

Problem 6

1. There may be a deadlock. The minimum and maximum number of times that each task can print out its "Hello ..." message is 0 and 10000 respectively. A way to avoid the deadlock problem is to acquire the two locks in the same order in both tasks (for example first `lock_a` and then `lock_b`). Another valid option would be to simply use one lock.
2. The programmer has introduced `#pragma omp flush` in the execution of each task to force the memory consitency, and then, be sure that `ball_1` and `ball_2` values are updated and seen by the another thread.

PAR – Final Exam – Course 2015/16-Q2

June 9th, 2016

Question 1 (2.0 points).
Assume a NUMA multiprocessor system composed of 2 NUMAnodes, each with two sockets and 8 GB of main memory. Each socket has two cores and a per-socket shared cache memory of 16 MB. The cache line size is 64 bytes. Data coherence in the system is maintened using Write-Invalidate MSI protocols, with Snoopy cache coherency support within a NUMAnode and directory-based coherency protocol support between the two NUMAnodes.

1. (0.5 points) Indicate the total number of bits needed per socket to maintain cache coherence inside a NUMAnode.
2. (0.5 points) Indicate how many bits should be used in the directory for each line in main memory and the total number of bits per NUMAnode to maintain cache coherence between them.
3. (1.0 point) Given the following OpenMP code executed using 8 threads running on each of the 8 cores of the previous multiprocessor system:

```
#define NTHREADS 8
...
#pragma omp parallel num_threads(NTHREADS)
{
 ...
 #pragma omp critical /* (1) */
  do_work();
 ...
}
```

Let's assume that acquiring the lock to enter the critical region can be implemented with any of these two alternatives:

```
/* test-test&set code */
      ...
lock: ld r2, barrier.lock
      bnez r2, lock
      t&s r2, barrier.lock
      bnez r2, lock
      ...
```

```
/* test&set code */
      ...
lock: t&s r2, barrier.lock
      bnez r2, lock
      ...
```

Assuming a moment where none of the threads are inside the critical region (and no copy of the variable on which the lock is performed (*barrier.lock*) is present in any cache), what would be the number of invalidations sent through the bus and number of invalidated cache lines if all the threads arrive at the same time to (1) (i.e. try to enter the critical region) in the following situations: Important: You have to count just UNTIL the first thread acquires the lock.

(a) The lock implementation of the critical region is done using the atomic instruction $test - test\&set$ with the synchronization algorithm shown above (justify your answer in no more than 2-3 lines).

(b) The lock implementation of the critical region is done using the atomic instruction *test&set* with the synchronization algorithm shown above (justify your answer in no more than 2-3 lines).

Question 2 (3.0 points).
Given the following code for which we want to evaluate two alternative data decompositions strategies (by rows and by columns):

```
void compute( int N, double *u) {
  for ( int i = 1; i < N-1; i++ ) {
    for ( int k = 1; k < N-1; k++ ) {
        double tmp = u[i+1][k] + u[i-1][ k] + u[i] [k+1] - 4 * u[i][k];
        u[i][k] = tmp/4;
    }
  }
}
```

In order to evaluate the parallel behaviour of these two strategies we will assume the data sharing model explained in class, in which the time to access remote data is determined by an startup time t_s and a transfer time per element t_w. To model the computation part, we will assume that the cost of computing one iteration of the most internal loop is t_c.

1. (0.5 points) Clearly explain (and draw) the data access pattern during the computation of the loop, indicating which access(es) can create a potential data dependence and which access(es) may require remote data accesses before starting the parallel execution of the loop and/or during the parallel execution of the loop. In case of creating a data dependence, how can you reduce the serialisation caused by that dependence, i.e. which program transformation should you apply?
2. **Column decomposition for matrix u**. Answer the following questions:
   (a) (0.25 points) Draw the task dependence graph for N=16, assuming each column is executed by a task.
   (b) (0.75 points) Obtain the expression for the speedup $S(p)$, clearly indicating the overheads introduced by the access to remote data.
   (c) (0.25 points) This computation is part of program with an initialization phase that must be executed in sequential. The cost of this initialization phase is 1000 time units. Assuming t_c is 1 time unit (the cost of executing one iteration of the most internal loop), compute the parallel fraction φ and the S_∞ for N=512.
3. **Row decomposition for matrix u**. Answer the following questions:
   (a) (0.25 points) Draw the task dependence graph for N=16, assuming each row is executed by a task.
   (b) (1.0 point) Obtain the expression for the speedup $S(p)$, clearly indicating the overheads introduced by the access to remote data.

Question 3 (3.0 points).
In this problem you have to parallelize function `iter_distribute`. This function copies vector S into vector D in such a way that all those elements ($S[i]$) with the same value for $S[i]\%256$ are stored in consecutive positions of D. Therefore, at the end of the function, there will be in D all the elements of S organized in 256 groups of elements: first all those with value %256 equal to 0, then those with value %256 equal to 1, ... up to those with value %256 equal to 255. In order to do this copy, the implementation provides a vector C which is initialized in function `preprocessing`; each element $C[value]$ indicates the initial position in D to store all those elements $S[i]$ whose $S[i]\%256 = value$.

```
#define N 1024*1024*1024
unsigned int S[N], D[N], C[256];

void iter_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
 unsigned int i, value;

 for (i=0; i<n; i++) {
   value = S[i]%256;
   D[C[value]] = S[i];
   C[value]++;
 }
}

void main() {
  ...
  // Here we have initialized S and C
  preprocessing(S,C,N);
  ...
  iter_distribute(S, N, C, D);
  ...
}
```

Assuming that it is not important the order of the elements inside the same group in D (i.e. those elements of S can be written in different relative order in D in the parallel program than the order in the sequential code), we ask you:

1. (1.5 points) Write an OpenMP parallel code for function `iter_distribute` that follows an iterative task decomposition. Your solution should exploit parallelism among threads.

2. (1.5 points) We have created a recursive sequential version of previous code and re-written the main program. Write an OpenMP parallel code for function `rec_distribute` and add the necessary code and directives to the main program to follow a tree divide–and–conquer task decomposition. Your parallel code should include a cut–off mechanism based on parallel recursive tree depth; allowing parallel recursive calls for depths smaller than `4`.

```
#define N 1024*1024*1024
// ... As above
void rec_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
 unsigned int i, value;
 unsigned int n2 = n/2;
 if (n==1) {
   value = S[0]%256;
   D[C[value]] = S[0];
   C[value]++;
 }
 else {
    rec_distribute( S, n2, C, D);
    rec_distribute( &S[n2], n-n2, C, D);
 }
}

unsigned int S[N], D[N], C[256];

void main() {
  // Here we have initialized S and C
  preprocessing(S,C,N);
  ...
  rec_distribute(S, N, C, D);
}
```

Question 4 (2.0 points).
Write two different OpenMP parallel implementations of function `iter_distribute` (previous question) using the following strategies. As above, assume that it is not important the order of the elements inside the same group in D (i.e. those elements of S can be written in different relative order in D in the parallel program than the order in the sequential code).

1. (1.0 point) Cyclic Data Decomposition of the Input vector `S`. You cannot use the `omp for` worksharing in order to implement this parallel version.
2. (1.0 point) Block Data Decomposition of the Output vector `C`. Consequently, each thread uses and updates only a part of C, and then, will only write to a part of D, depending of the part of C the thread works with. You cannot use the `omp for` worksharing in order to implement this parallel version. Please, ensure a proper load balance of the data distribution: no more than 1 element of `C` of extra-work.

Important: Both implementations should try to minimize the use of synchronizations.

Solution

Question 1 (2.0 points).

1. Inside a NUMAnode we need a snoopy. The snoopy requires 2 bits per cache line in order to keep the coherence state, which can be Modified, Shared or Invalid (MSI). The total number for a 16MB cache is the number of cache lines ($\frac{16\times2^{20}}{2^6}$ lines) multiplied by the number of bits per line (2 bits/line). That is: 32×2^{14} bits for each cache.
2. Between NUMAnodes in node, we need a directory structure. The directory needs 4 bits per local memory line: 2 bits for the coherence state information (Modified, Shared, Invalid) and 2 bits more for the presence information: 1 bit per each NUMAnode. The total number of entries in the directory is the number of memory lines of the 8GB local memory in a NUMAnode ($\frac{8\times2^{30}}{2^6}$ memory lines). Multiplied by the number of bits per memory line (4 bits/line) gives a total of 8×2^{26} bits.
3. Answers to this question:
   - (a) The maximum value of invalidations occurs when all the threads try to enter the critical region at the same time. Every thread perform the first test and loads the lock variable, so there will be a copy of it in every cache. After that every thread executes test&set instruction. The first thread that succeeds provoking a BusRdX transaction on the bus is the one that acquires the lock. The other copy of the lock variable inside the NUMAnode (related with the cache associated to the other socket) is invalidated. The hub, which has listened the BusRdX transaction sends an invalidation request (RdRqX) to the hub of the other NUMAnode. After that the other two copies are invalidated (related to the caches associated with the two sockets of that NUMAnode). In total we have 3 cache lines invalidations.
   - (b) The maximum value of invalidations occurs when all the threads try to enter the critical region at the same time. The execution of test&set provokes a BusRdX transaction on the bus, so the first thread that succeeds doing it first, is the one that acquires the lock. As there is no copy of the variable in any cache, on which the lock is performed, no invalidation is sent and consequently, there are no invalidated caches.

Question 2 (3.0 points).

1. To compute one element of the matrix, the algorithm needs to access one element in the previous row $(i-1)$ (same column), one element in the next row $(i+1)$ (same column) and one element in the same row but previous column $(k-1)$. Accesses with $+1$ might require initial data transfers (before the computation) and accesses -1 might require data transfers during the computation. In the case of row decomposition, we will have to apply blocking and use explicit synchronization to guarantee the correct execution of the blocks.
2. **Column decomposition for matrix u**. Answers of the questions:
   - (a) Graph with N=16 tasks, totally independent.
   - (b) In that case, data transfers must be done before the computation. Each processor will compute a block of N/P columns and N rows (we will assume that N is sufficiently large, so we approximate N-1 with N).
     $S(p) = T(1)/T(p)$
     $T(1) = N * N * t_c$
     $T(p) = DataTransfer + Computation$
     $DataTransfer = t_s + N * t_w$ (1 column)
     $Computation = (N/P) * N * t_c$
   - (c) Parallel fraction φ and the S_∞ for N=512.
     $\varphi = (512x512)/(1000 + (512 * 512))$
     $S_\infty = 1/(1-\varphi)$
3. **Row decomposition for matrix u**. Answers of the questions:
   - (a) Graph with N=16 tasks, but in this case with dependences between consecutive tasks. It will requiere initial data transfer (1 column) and also during the computation.

(b) In that case, data transfers must be done before (1 row) and during the computation. Each processor will compute N/P columns and N rows (for the computation N is equivalent to N-1).We will apply blocking to extract a certain parallelism. Each block will compute N/P columns and B rows. Each processor will compute then N/B blocks.
$S(p) = T(1)/T(p)$
$T(1) = N * N * T_c$
$T(p) = DataTransfer + Computation$
$DataTransfer = Initialtransfer + DuringComputationTransfers$
$Initialtransfer = T_s + N * T_w$ (1 row)
$DuringComputationTransfers = ((P-2) * (T_s + B * T_w)) + ((N/B) * (T_s + B * T_w))$
Each block needs $BLOCK\_TIME = (N/P) * B * T_c$
$Computation = BLOCK\_TIME * ((N/B) + (P-1))$

Question 3 (3.0 points).

1. (1.5 points) An OpenMP parallel code for function `iter_distribute` that follows an iterative task decomposition.

```
#include <omp.h>
#define N 1024*1024*1024

void iter_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
 unsigned int i, value;

 omp_lock_t locks_C[256];

 for(i=0;i<256;i++)
   omp_init_lock(&locks_C[i]);

 #pragma omp parallel for private(value)
 for (i=0; i<n; i++) {
   value = S[i]%256;

   omp_set_lock(&locks_C[value]);
      D[C[value]] = S[i];
      C[value]++;
      #pragma omp flush
   omp_unset_lock(&locks_C[value]);
 }

 for(i=0;i<256;i++)
   omp_destroy_lock(&locks_C[i]);

}

unsigned int S[N], D[N], C[256];
void main() {
  ...
  // Here we have initialized S and C
  preprocessing(S,C,N);
  ...
  iter_distribute(S, N, C, D);

  ...
}
```

2. (1.5 points) An OpenMP parallel code for function `rec_distribute` that follows a tree divide–and–conquer task decomposition and includes a cut–off mechanism based on parallel recursive tree depth; allowing parallel recursive calls for depths smaller than `4`.

```
#include <omp.h>
```

```
#define N 1024*1024*1024
omp_lock_t locks_C[256];

// ... As above
void rec_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D,
                    unsigned int depth) {
 unsigned int i, value;
 unsigned int n2 = n/2;
 if (n==1) {
   value = S[0]%256;
   omp_set_lock(&locks_C[value]);
      D[C[value]] = S[0];
      C[value]++;
      #pragma omp flush
   omp_unset_lock(&locks_C[value]);
 }
 else {
    #pragma omp task final(depth>=4) mergeable
    rec_distribute( S, n2, C, D, depth+1);
    #pragma omp task final(depth>=4) mergeable
    rec_distribute( &S[n2], n-n2, C, D, depth+1);
 }
}

unsigned int S[N], D[N], C[256];

void main() {
  // Here we have initialized S and C
  preprocessing(S,C,N);
  ...

 for(i=0;i<256;i++)
   omp_init_lock(&locks_C[i]);

  #pragma omp parallel
  #pragma omp single
  rec_distribute(S, N, C, D, 0);

 for(i=0;i<256;i++)
   omp_destroy_lock(&locks_C[i]);
}
```

Question 4 (2.0 points).
Different OpenMP parallel implementations of function `iter_distribute` (previous question) using the following strategies.

1. (1.0 point) Cyclic Data Decomposition of the Input vector `S`.

```
#include <omp.h>
#define N 1024*1024*1024

void iter_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
 unsigned int i, value;

 omp_lock_t locks_C[256];

 for(i=0;i<256;i++)
   omp_init_lock(&locks_C[i]);

 #pragma omp parallel private(value, i)
 {
   unsigned int myid = omp_get_thread_num();
```

```
    unsigned int nt   = omp_get_num_threads();

    for (i=myid; i<n; i+=nt) {
      value = S[i]%256;

      omp_set_lock(&locks_C[value]);
         D[C[value]] = S[i];
         C[value]++;
         #pragma omp flush
      omp_unset_lock(&locks_C[value]);
    }
  }

  for(i=0;i<256;i++)
    omp_destroy_lock(&locks_C[i]);

}
```

2. (1.0 point) Block Data Decomposition of the Output vector C.

```
#include <omp.h>
#define N 1024*1024*1024

void iter_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
 unsigned int i, value;

 #pragma omp parallel private(value, i)
 {
   unsigned int myid = omp_get_thread_num();
   unsigned int nt   = omp_get_num_threads();
   unsigned int C_nt= 256/nt;
   unsigned int rest_nt = 256%nt;
   unsigned int C_start = myid*C_nt + ((rest_nt>myid)?myid:rest_nt);
   unsigned int C_end   = C_start + index_nt + (myid<rest_nt);

   for (i=0; i<n; i++) {
     value = S[i]%256;
     if ((value >= C_start) && (value < C_end))
     {
        D[C[value]] = S[i];
        C[value]++;
     }
   }
 }

}
```