PAR – Final Exam – Course 2018/19-Q1
January 16th, 2019

Problem 1 (2.5 points) Given the following C code with tasks identified using the *Tareador* API:

```
#define N 4
int m[N][N];

// loop 1
for (int i=0; i<N; i++) {
    tareador_start_task ("loop1");
    for (int k=0; k<=i; k++) {
        m[i][k] = comp1(i,k); // no access to m inside function comp1
    }
    tareador_end_task ("loop1");
}

// loop 2
for (int i=0; i<N; i++) {
    tareador_start_task ("loop2");
    for (int k=i+1; k<N; k++) {
        m[i][k] = comp2(m[k][i]); // no access to m inside function comp2
    }
    tareador_end_task ("loop2");
}

// print all the elements of m
tareador_start_task("print");
print_results(m);
tareador_end_task("print");
```

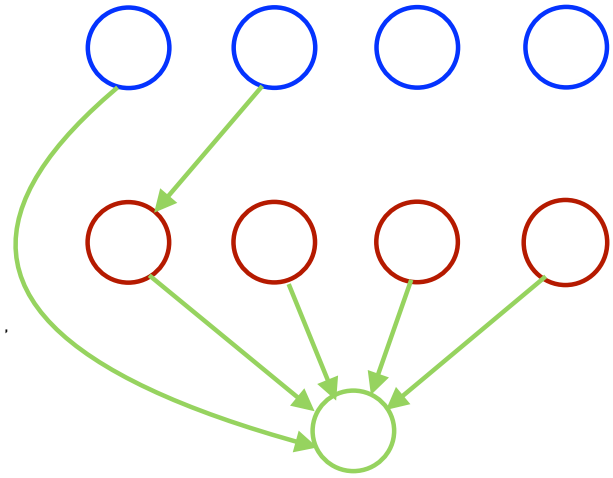
Assuming that: 1) the execution of the each invocation of functions `comp1` and `comp2` functions takes 10 time units; and 2) the execution of function `print_results` takes 20 time units; **we ask:**

1. Indicate which positions of matrix `m` are read and/or written by each task generated in `loop 1` and `loop 2` and in task `print_results`. Fill in the table in the provided answer sheet to answer this question.

   Solution:

2. Draw the task dependence graph (TDG), indicating for each node its cost in terms of execution time (in time units). Use the task identifiers that we provided in the previous table (i.e. $t1$ ... $t9$).

   Solution:



3. Compute the values for T_1, T_∞ and the potential parallelism.

   Solution:

4. Let's consider that each task creation has an associated overhead of 2 time units. Taking this overhead into account, and assuming that tasks are created in the order in which they are found in the sequential execution, compute the new values for T_1 and T_∞. Clearly identify to which tasks the overhead accounts for.

   Solution:

5. Assuming a distributed memory machine with a matrix distribution by rows on four processors and a message passing model where the transfer cost of a message of B elements is $t_{comm} = t_s + B \times t_w$ being t_s y t_w the start–up time and transfer time of one element, respectively; write the expression that determines the execution time T_4 of the program (taking into account computation time and data sharing overheads only) for the following data and task assignment to processors:

Processor	P0	P1	P2	P3
Row distribution	m[0][0..3]	m[1][0..3]	m[2][0..3]	m[3][0..3]
Task assignment	$t1, t5, t9$	$t2, t6$	$t3, t7$	$t4, t8$

   Solution:

Problem 2 (5 points) The following code excerpt implements the multiplication of a dense rectangular matrix `M` times a vector `X`, accumulating the result in vector `Y`:

```
/* Y += M * X  */
...
for (i=0; i<R; i++) {
  for (j=0; j<C; j++) {
    Y[i] += M[i][j] * X[j];
  }
}
...
```

where `M` is a matrix with `R` rows by `C` columns; `X` is a vector with `C` elements; and `Y` is a vector with `R` elements.

We want to parallelize the code using appropriate OpenMP pragmas and invocations to intrinsic functions, assuming the following constraints: 1) you **cannot** make use of the `for` work–sharing construct in OpenMP to distribute work among threads; 2) you **cannot** assume that the number of threads evenly divides neither `R` nor `C`; 3) parallelization overheads should always be kept as low as possible (due to thread/task creation, synchronization, false sharing, ...); 4) both the matrix and the vectors' initial addresses are conveniently aligned to cache boundaries; and 5) the number of consecutive elements of the data structures that fit into a cache line is `NUM_ELEMENTS_PER_CACHE_LINE`.

We ask you to write four independent versions for the parallel version of this code:

1. (1 point) Implement a first parallel version that follows an iterative task decomposition.

   Solution:

2. (1.5 points) Implement a second parallel version that follows a recursive *divide and conquer* task decomposition with a *tree* parallelization and a *cut-off* control which creates the maximum number of tasks while avoiding false sharing.

   Solution:

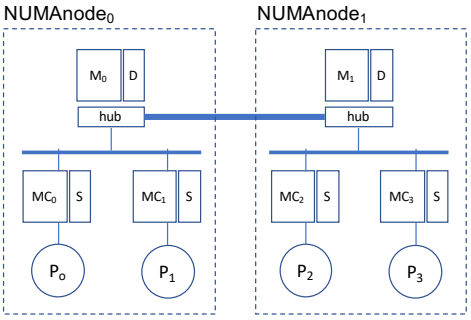
3. (1.25 points) Implement a third parallel version that obeys to an *input block geometric data decomposition* of the input vector `X`.

   Solution:

4. (1.25 points) Finally, implement a fourth parallel version that obeys to an *output block-cyclic geometric data decomposition* of the output vector `Y`.

   Solution:

Problem 3 (2.5 points) Consider the following parallel architecture composed of two NUMA nodes, each with its own main memory, directory to keep coherence between NUMA nodes (write-invalidate MSU) and two processors with their own cache memory and snoopy to keep coherence within each node (write-invalidate MSI). For the purposes of this problem, you can assume infinite sizes for both main and cache memories.



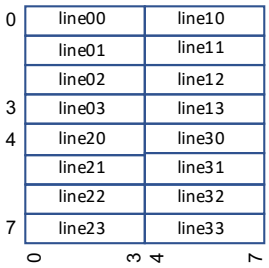
Legend:
NUMAnode$_i$: NUMA node i with two processors
M$_i$: main memory for NUMAnode$_i$
D: directory associated to M
hub: interconnect between NUMA nodes
P$_j$: processor j inside NUMA node
MC$_j$: cache memory local to processor P$_j$
S: snoopy associated to MC

Coherence commands:
- Snoopy: BusRd(j), BusRdX(j), BusUpgr(j) and Flush(j), being j the snoopy/cache number doing the action or hub
- Hub/directoty: RdReq(i→j), WrReq(i→j), UpgrReq(i→j), Dreply(i→j), Ack(i→j), Fetch(i→j), Invalidate(i→j) and WriteBack(i→j), from NUMAnode$_i$ to NUMAnode$_j$

Also consider the following skeleton for a parallel program only showing the accesses to matrix `a`:

```
#define CACHE_LINE 4 // number of integers in a cache line
#define n 8
int a[n][n];

// initialization loop
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        a[i][j] = ...;

// compute loop 1
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        a[i][j] = foo(a[i][j], ...);

// compute loop 2
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        ... = goo(a[j][i], ...);    // Transposed access to matrix a
```

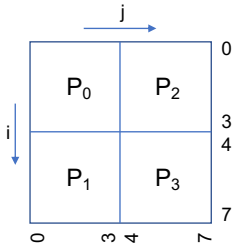
Matrix `a` is stored by rows in memory, with each memory line storing 4 consecutive integer values, Therefore matrix `a` occupies 16 memory lines, as shown (and labeled) in the following figure:



1. Assume that after the initialisation loop the state of the caches and directories is as shown in the upper part of the table in the provided answer sheet for this problem (for lines *line0x*, *line1x*, *line2x* and *line3x*, with $x = 0..3$). Indicate which iterations of the loops `i` and `j` were executed by each of the 4 processors P_{0-3} in the multiprocessor system.

   Solution:

2. Assume that both loops in *compute loop 1* can be parallelized using all processors in the system, as shown in the iteration space below:



   The middle part of the table in the provided answer sheet shows the coherence commands that are placed on the bus and exchanged between NUMA nodes during the execution of *compute loop 1* to maintain the coherence for the memory lines of matrix `a`. Complete the appropriate cells in the provided answer sheet to show the status for the lines of matrix `a` in the cache memories and in the directories after the execution of *compute loop 1*.

3. Assume the same parallelization for *compute loop 2*, again using all processors in the system. During the execution of *compute loop 2*, which coherence commands are placed on the bus by the snoopies and which coherence commands are exchanged between NUMA nodes to maintain the coherence for matrix `a`? After the execution of *compute loop 2*, which is going to be the status for the lines of matrix `a` both in the cache memories and in the directories? Fill in the lower part of the table in the provided answer sheet to answer this question.

Solution for questions 3.2 and 3.3:

		NUMA node 0					NUMA node 1				
		Intra-node bus commands	MC0 status	MC1 status	Inter-node NUMA commands	Directory entry sharers/state	Intra-node bus commands	MC2 status	MC3 status	Inter-node NUMA commands	Directory entry sharers/state
initialization loop	line0x for a										
	line1x for a										
	line2x for a										
	line3x for a										
compute 1 loop	line0x for a										
	line1x for a										
	line2x for a										
	line3x for a										
compute 2 loop	line0x for a										
	line1x for a										
	line2x for a										
	line3x for a										

Student name: .....

Tables to be used to deliver your solution to **Problem 1**

loop 1

i	k	read	written	task id
0	0		0 0	t1
	1			
	2			
	3			
1	0		1 0	t2
	1		1 1	
	2			
	3			
2	0			t3
	1			
	2			
	3			
3	0			t4
	1			
	2			
	3			

loop 2

i	k	read	written	task id
0	0			t5
	1	1 0	0 1	
	2			
	3			
1	0			t6
	1			
	2			
	3			
2	0			t7
	1			
	2			
	3			
3	0			t8
	1			
	2			
	3			

print results

read	written	task id
		t9

Student name: .....

Table to be used to deliver your solution to **Problem 3**

		NUMA node 0					NUMA node 1				
		Intra-node bus commands	MC0 status	MC1 status	Inter-node NUMA commands	Directory entry sharers/state	Intra-node bus commands	MC2 status	MC3 status	Inter-node NUMA commands	Directory entry sharers/state
initialization loop	line0x for a	BusRdX(0)	M			01 M					
	line1x for a	BusRdX(0)	M			01 M					
	line2x for a	BusRdX(1)		M		01 M					
	line3x for a	BusRdX(1)		M		01 M					
compute 1 loop	line0x for a										
	line1x for a	BusRd(hub), Flush(0) BusUpgr(hub)			Dreply(0->1) Ack(0->1)		BusRd(2) BusUpgr(2)			RdReq(1->0) UpgrReq(1->0)	
	line2x for a										
	line3x for a	BusRd(hub), Flush(1) BusUpgr(hub)			Dreply(0->1) Ack(0->1)		BusRd(3) BusUpgr(3)			RdReq(1->0) UpgrReq(1->0)	
compute 2 loop	line0x for a										
	line1x for a										
	line2x for a										
	line3x for a										

PAR – Final Exam – Course 2018/19-Q2

June 18^{th}, 2019

Problem 1 (3 points) Given the following sequential code with calls to *Tareador* to define tasks:

```
#define N 4
int m[N][N];
char taskname[8];

for (int i = 0; i < N/2; i++) {
     sprintf(taskname, "INIT_%d", i);
     tareador_start_task (taskname);
     for (int k = 0; k < N; k++) {
         m[i][k] = foo(i, k); // no access to m inside function foo
         m[N-i-1][k] = m[i][k];
     }
     tareador_end_task(taskname);
}

for (int i = 0; i < N; i++) {
    sprintf(taskname, "CALC_%d", i);
    tareador_start_task(taskname);
    for (int k = 0; k < N; k++) {
        if (i < N/2)
            m[i][k] += calculate_something (m[i][k]);
        else
            m[i][k] += calculate_something (m[i-N/2][k]);
    }
    tareador_end_task(taskname);
 }

 tareador_start_task("END");
 save_results(m);
 tareador_end_task("END");
```

0 0 ; 0 1 ; 0 2 ; 0 3 —> INIT_0
1 0 ; 1 1 ; 1 2 ; 1 3 —> INIT_1

3 0 ; 3 1 ; 3 2 ; 3 3 —> INIT_0
2 0 ; 2 1 ; 2 2 ; 2 3 —> INIT_1

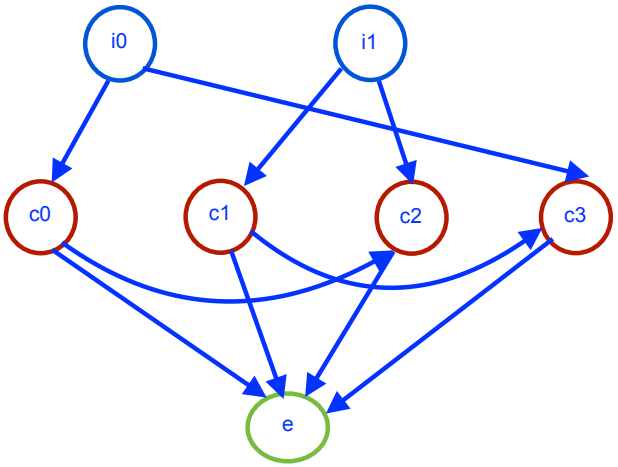
0 0 ; 0 1 ; 0 2 ; 0 3 —> CALC_0
1 0 ; 1 1 ; 1 2 ; 1 3 —> CALC_1

0 0 ; 0 1 ; 0 2 ; 0 3 —> CALC_0
1 0 ; 1 1 ; 1 2 ; 1 3 —> CALC_3

2 0 ; 2 1 ; 2 2 ; 2 3 —> CALC_2
3 0 ; 3 1 ; 3 2 ; 3 3 —> CALC_3

Assume: 1) the execution of functions `calculate_something` and `save_results` do not modify the input parameters; and 2) the execution of each `INIT_i` task takes 10 time units, of each `CALC_i` task takes 40 time units and of the `END` task takes 60 time units. **We ask you:**

1. (1 point) Draw the complete *Task Dependence Graph* (TDG) using the task identifiers dynamically set in the code.
   Solution:



2. (1 point) Compute the values for T_1, T_∞, potential parallelism Par and P_{min}.

   Solution: T1 = 240
   Tinf = 150
   Par = 1.6
   Pmin = 2

Now assume that 1) memory of the parallel system is physically distributed so that the access to data in different processors introduces a data–sharing overhead; this overhead follows the model explained in class: $t_{comm} = t_s + B \times t_w$ being t_s and t_w startup and transfer time, respectively, and B the number of elements accessed. And 2) a *first touch* allocation policy is used by the operation system to initially place data in distributed memory. **We ask you:**

3. (1 point) Define the assignment of tasks to the P_{min} processors calculated before, minimizing the data–sharing overhead, and find the expression for the execution time T_p for $p = P_{min}$ taking into account this overhead.

   Solution:

   T2 = Tcomputation + Tcomm
   T2 = 150 + Tcomm
   T2 = 150 + (ts + 4tw + ts + 8tw)

Problem 2 (4 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() {
    ...
    preprocessing(S, C, N); // initialization of S and C
    ...
    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. the 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 point) Write an OpenMP parallel implementation of function `iter_distribute` that follows a *Geometic Cyclic Data Decomposition* of the Input vector `S`. You should maximize the parallelism that your solution offers minimizing the serialization that is introduced by synchronization, if any.
   Solution:

2. (1 point) Write an alternative OpenMP parallel implementation of function `iter_distribute` that follows a *Geometric 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. Your parallel implementation has to ensure a proper load balance of the data distribution (i.e. no more than 1 element of difference) and maximize the parallelism that your solution offers minimizing the serialization that is introduced by synchronization, if any.
   Solution:

3. (2 points) Finally, we have created a recursive divide–and–conquer sequential version of previous code. Write an OpenMP parallel code for function `rec_distribute` and main program adding the necessary code and directives to implement a *recursive tree task decomposition*. Your parallel code should include a cut–off mechanism based on recursion depth, allowing parallel recursive calls for depths smaller than `MAX_DEPTH`. Your parallel code should not use the `mergeable` clause and should minimize the serialization that is introduced by synchronization, if any.

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

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);
    }
}


void main() {
    ...
    preprocessing(S, C, N); // initialization of S and C
    ...
    rec_distribute(S, N, C, D);
    ...
}
```

Solution:

Problem 3 (3 points) Given a NUMA muliprocessor system with X NUMA nodes, each NUMA node including Y sockets sharing the access to the node's main memory, each socket with Z cores sharing the access to a per-socket cache. Coherence inside NUMA nodes is maintained with a snoopy–based mechanism implementing write–invalidate MSI. Coherence among NUMA nodes is maintained with a directory–based mechanism implementing write–invalidate MSU. The following table summarises the main characteristics of the system, including the total number of bits devoted to keep coherence at the two levels (inside and among NUMA nodes).

System characteristic	**Size**
Number of NUMA nodes	X nodes
Size of main memory per NUMA node	8 GB (gigabytes)
Number of bits per NUMA node devoted to coherence in the directory (no data)	2^{32} bits
Number of sockets per NUMA node	Y sockets
Size of each per-socket cache memory	4 MB (megabytes)
Number of bits per NUMA node devoted to coherence among sockets (no data)	2^{20} bits
Number of cores per socket	Z cores
Cache and memory line size	32 B (bytes)

Question 3.1 (1 point) We ask you to compute the number of NUMA nodes X composing the system and the number of sockets Y per NUMA node. With the information provided, is it possible to determine the number of cores Z per socket? In case of affirmative answer, please compute that number.

(8*2^30 / 32) * (2+X) = 2^32;
(2^28) * (2+X) = 2^32;
2+x=2^4; X=14

Y*(4*2^20 / 32) * 2 = 2^20;
Y*2^18=2^20; Y = 4

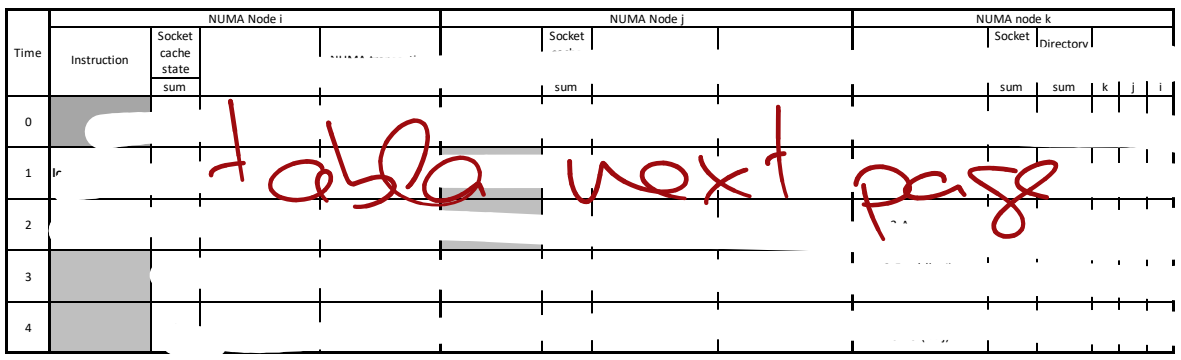
Solution:

Question 3.2 (1 point) Write an alternative code in C (with no OpenMP pragmas) to implement the atomic access that is done during the execution of previous code using the `atomic` pragma in OpenMP, making use of the following low–level *load-linked store-conditional* synchronisation primitives:

```
int load_linked (int *address);
int store_conditional (int *address, int value); // returns 1 in case of success
```

Solution:

Question 3.3 (1 point) Fill in the table in the solutions page with the sequence of coherence actions that will take place, both within and across NUMA nodes, if the two atomic updates by the core in $NUMAnode_i$ and by the core in $NUMAnode_j$ are **not overlapped in time**, so that first the core in $NUMAnode_i$ atomically updates variable `sum` followed by the core in $NUMAnode_j$ atomically updating variable `sum`. Clearly indicate, for each instruction, the order in which the coherence actions occur (e.g. `1-BusRd; 2-RdReq(k->j); 3-Dreply(j->k)`), how the state of the line in the directory will change and how the state of the lines holding the different copies in cache will change. For the execution of `load_linked` and `store_conditional` the processor issues a `PrRd` and `PrWr` command, respectively.



Solution sheet for Problem 3

SURNAME:

NAME:

Time	NUMA Node i				NUMA Node j				NUMA node k					
	Instruction	Socket cache state	Bus transaction	NUMA transaction	Instruction	Socket cache state	Bus transaction	NUMA transaction	NUMA transaction	Socket cache state	Directory state	Sharers list		
		sum				sum				sum	sum	k	j	i
0		-				-				-	U	0	0	0
1	**load_linked**	S	1-BusRd	2-RdReq(i->k)					3-Dreply(k->i)		S			
2	**store_conditional**	M	1-BusUpgr	2-UpgrReq(i->k)					3-ack(k->i)					
3					**load_linked**		1-BusRd	2-RdReq(j->k)	3-Dreply(k->i)		M			
4					**store_conditional**									

(1) Cache line state: M (modified), S (shared) or I (invalid)
(2) Coherence commands inside NUMA node: BusRd, BusRdX, BusUpgr and Flush
(3): Line state in node memory: M (modified), S (shared) or U (uncached)
(4) Coherence commands between NUMA nodes a and b: RdReq(a->b), WrReq(a->b), UpgrReq(a->b), Dreply(a->b), Ack(a->b), Fetch(a->b) and Invalidate(a->b)

PAR – Final Exam – Course 2019/20-Q1
January 16^{th}, 2020

Problem 1 (2 points)

The following code excerpt instrumented with *Tareador* belongs to a program we want to parallelize. It simulates the heat diffusion equation using the *Gauss-Seidel* method. We have developed a blocked implementation that creates $N \times M$ blocks:

```
#define N ...
#define M ...
#define lowerb(id, p, n)  ( id * (n/p) + (id < (n%p) ? id : n%p) )
#define upperb(id, p, n)  ( lowerb(id, p, n) + (n/p) + (id < (n%p)) - 1 )

/* Blocked Gauss-Seidel solver: Compute 1 subblock */
double compute_GS_block( double *u, unsigned sizex, unsigned sizey,
                         int i_lb, int i_ub, int j_lb, int j_ub) {
  double unew, diff, blocktotal=0.0;
  for (int i=max(1, i_lb); i<= min(sizex-2, i_ub); i++)
    for (int j=max(1, j_lb); j<= min(sizey-2, j_ub); j++) {
      unew= 0.25 * ( u[     i*sizey     + (j-1) ]+  // left
                     u[     i*sizey     + (j+1) ]+  // right
                     u[ (i-1)*sizey     + j     ]+  // top
                     u[ (i+1)*sizey     + j     ]); // bottom
      diff = unew - u[i*sizey+ j];
      blocktotal += diff * diff;
      u[i*sizey+j]=unew;
    }
  return(blocktotal);
}

/* Blocked Gauss-Seidel solver: one iteration step */
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
    double total=0.0;
    int howmanyX = N;
    int howmanyY= M;

    tareador_disable_object(&total);
    for (int blockidX = 0; blockidX < howmanyX; ++blockidX)
      for (int blockidY = 0; blockidY < howmanyY; ++blockidY) {
        int i_lb = lowerb(blockidX, howmanyX, sizex);
        int i_ub = upperb(blockidX, howmanyX, sizex);
        int j_lb = lowerb(blockidY, howmanyY, sizey);
        int j_ub = upperb(blockidY, howmanyY, sizey);
        tareador_start_task("GS Block");
        total+=compute_GS_block( u, sizex, sizey, i_lb, i_ub, j_lb, j_ub);
        tareador_end_task("GS Block");
      }
    tareador_enable_object(&total);
    return total;
}

int main() {
    init();
    tareador_ON();
    relax_gauss(...);
    tareador_OFF();
 }
```

We ask you to answer the following questions:

1. (0.5 points) Draw the Task Dependence Graph (TDG) that would be generated by *Tareador* for 1 invocation of routine `relax_gauss` considering $N = 3$ and $M = 5$ given the implementation shown above. Note: You can ommit the labels with details provided by *Tareador* within each node of the TDG.

Solution:

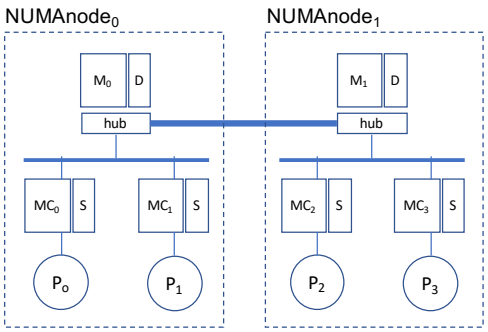
2. (0.5 points) In view of the TDG above, give a general expression for P_{min} as a function of N and M. Hint: Consider how the TDG would look like if N was larger than M.

   Solution:

3. (1 point) Give an expression for the parallel execution time as a function of N and M considering that: 1) the execution time of each invocation of routine `compute_GS_block` and update of variable `total` takes T_c time units; 2) there exists a cost for syncronization between tasks so that the overhead that a task has to pay to get notified that ALL its predecessor tasks have finished is T_{sync}; 3) the cost for task creation is negligible; 4) P_{min} processors are used.

   Solution: Tpmin = (N + M - 1)*Tc + (N + M - 2)*Tsynch

Problem 2 (2 points) Consider a parallel architecture composed of two NUMA nodes, each with its own main memory, directory to keep coherence between NUMA nodes (write-invalidate MSU) and two processors with their own cache memory and snoopy to keep coherence within each node (write-invalidate MSI). For the purposes of this problem, you can assume infinite sizes for both main and cache memories.



Legend:
NUMAnode$_i$: NUMA node i with two processors
M$_i$: main memory for NUMAnode$_i$
D: directory associated to M
hub: interconnect between NUMA nodes
P$_j$: processor j inside NUMA node
MC$_j$: cache memory local to processor P$_j$
S: snoopy associated to MC

Coherence commands:
- Snoopy: BusRd(j), BusRdX(j), BusUpgr(j) and Flush(j), being j the snoopy/cache number doing the action or hub
- Hub/directoty: RdReq(i→j), WrReq(i→j), UpgrReq(i→j), Dreply(i→j), Ack(i→j), Fetch(i→j), Invalidate(i→j) and WriteBack(i→j), from NUMAnode$_i$ to NUMAnode$_j$

Within each invocation of routine `relax_gauss` in the previous problem there is an update of a shared variable named `total`. Being aware of the existence of two NUMA nodes we have applyied low-level synchronizations using the `load_linked (ll)` and the `store_conditional (sc)` primitives seen during the course:

```
double load_linked (double *address);
int store_conditional (double *address, double value); // returns 0 if fails; 1 otherwise
```

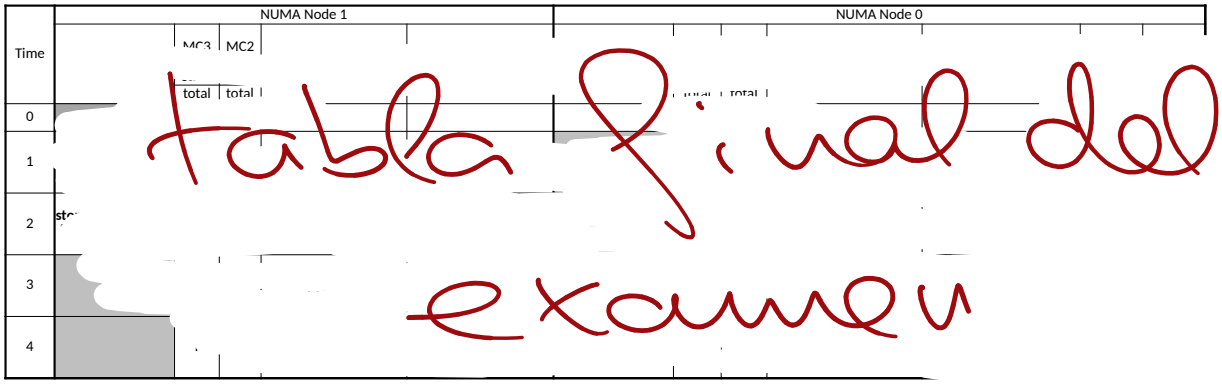
```
int ret;
double local_contribution, total=0.0;
...
local_contribution = compute_GS_block( u, sizex, sizey, i_lb, i_ub, j_lb, j_ub);
do {
     double value = load_linked (&total);
```

```
        value += local_contribution;
        ret = store_conditional (&total, value);
  } while (ret == 0);
  ...
```

Let us assume that: 1) the *home node* for variable `total` is NUMAnode$_0$; 2) the sequence of synchronization operations starts with the `ll` and the `sc` in core 3 within NUMAnode$_1$ and is later followed by the `ll` and the `sc` in core 1 within NUMAnode$_0$; 3) for the execution of `load_linked` and `store_conditional` the processor issues a `PrRd` and `PrWr` command, respectively.

We ask you to fill in the table provided in the answer sheet with the sequence of coherence actions that will take place, both within and across NUMA nodes. Indicate, for each instruction, the order in which the coherence actions occur and the processor or hub number performing the action (e.g. `1-BusRd(2); 2-RdReq(1->0); 3-Dreply(0->1); 4-...`), and the resulting state of the line in the directory as well as in any cache lines involved in storing the copies.

Solution:

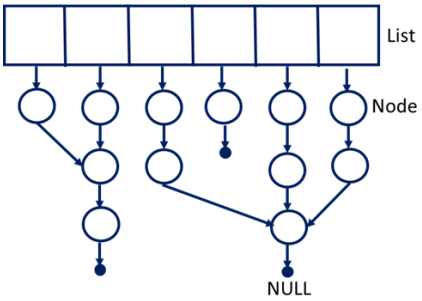


Problem 3 (3 points) Given the following C code that calculates the sum of all the elements in a structure of type List:

```
#define  N  ...

typedef struct Node {
    float value;
    int footprint; // initialized to value 0
    struct Node *next;
} List;

float compute_sequential (struct Node *p) {
     float sum = 0.0;
     int end = 0;
     while (p != NULL && end == 0) {
          int x = ++p->footprint;
          if (x == 1)  // to ensure value is accumulated only once
               sum = sum + heavy_calculation(p->value);
          else
               end = 1;
          p = p-> next;
     }
     return sum;
}

float process_vector (List *v[N]) {
     float res = 0.0;
     for (int i = 0; i < N; i++)
         res = res + compute_sequential (v[i]);
     return res;
}
```

```
void main() {
    List *v[N];
    // initialize footprint to value 0
    ...
    float total = process_vector (v);
    ...
}
```

where type `List` represents a vector of lists with the particularity that some lists are connected (two nodes from different lists can have the same following node in the list) as shown in the figure below:



We ask you to:

1. Write an OpenMP parallel version of the `process_vector` function using an iterative task decomposition strategy trying to maximize load balancing.

   Solution:

2. Write an OpenMP parallel version of `process_vector_rec` function following a *divide and conquer* task decomposition strategy and using the following sequential recursive version of `process_vector`:

```
float process_vector_rec (List *v[N], int n) {
     float res = 0.0;
```

```
    if (n <= MIN_SIZE)
        for (int i = 0; i < n; i++)
            res = res + compute_sequential (v[i]);
    else {
        int n2 = n / 2;
        float res1 = process_vector_rec (v, n2);
        float res2 = process_vector_rec (v+n2, n-n2);
        res = res1 + res2;
    }
    return res;
}

int main() {
    List *v[N];
    ...
    float total = process_vector_rec (v, N);
    ...
}
```

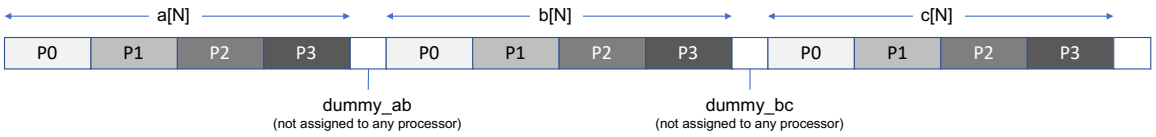
Solution:

Problem 4 (3 points) Assume the following incomplete version for an OpenMP parallel program:

```
#define N 1000
#define CACHE_LINE 64
#define INT_SIZE 4

struct SoA {
    int a[N];
    int dummy_ab[...];  // to complete
    int b[N];
    int dummy_bc[...];  // to complete
```

```
     int c[N];
};
struct SoA dataElements;
...
#pragma omp parallel
{
     int lower = ...;  // to complete
     int upper = ...;  // to complete
     for (int i = lower, i < upper; i++)
          dataElements.c[i] = foo(i, dataElements.a[i], dataElements.b[i]);
}
...
```

1. We ask you to complete the code above if we want to apply a *geometric block data decomposition* to each vector `a`, `b` and `c`, as shown in the figure below for the case of 4 processors:



   Your parallel code should make sure that vectors are properly aligned in memory (i.e. each one starts at the beginning of a cache line) in order to guarantee the proposed data decomposition; you can assume that vector `a` is already aligned in memory. To simplify the problem, you can also assume that the number of processors will always divide `N` perfectly. In particular you need to complete the lines indicated and add extra code, if necessary. As indicated in the code, each integer element occupies 4 bytes of memory and a cache line is 64 bytes long.

   Solution:

In order to avoid the need of introducing padding, the developer decided to change the definition of `dataElements` from *structure of arrays* to *array of structures*, as follows:

```
#define N 1000
#define CACHE_LINE 64
#define INT_SIZE 4

struct AoS {
     int a;
     int b;
     int c;
};
struct AoS dataElements[N];
...
#pragma omp parallel
{
     int lower = ...;  // to complete
     int upper = ...;  // to complete
     int step = ...;   // to complete
     for (int ii = lower; ii < upper; ii += ...)   // to complete
          for (int i = ii; ...; ...)               // to complete
               dataElements[i].c = foo(i, dataElements[i].a, dataElements[i].b);
}
...
```

In addition, the developer detected that function `foo` was introducing a monotonically increasing load unbalance in the computation (i.e. the computation time increases with the value of variable `i`), so he/she proposed to try a *geometric block-cyclic data decomposition* applied to vector `dataElements`. We ask you (the following two questions are independent, you can answer the third one assuming a generic answer from the second one):

2. Decide the number of consecutive elements of vector `dataElements` assigned to each processor in the *geometric block-cyclic data decomposition* that avoids false sharing and reduces load unbalance.

   Solution:

3. Complete the code above in order to implement a *geometric block-cyclic data decomposition.*

   Solution:

Solution sheet for Problem 2

SURNAME:

NAME:

NumaNode0 -> Home
Fetch -> dreply
Invalidate -> ack

NumaNode1 -> Local
RdReq -> dreply
WrReq -> dreply
UpgrReq -> ack

Time	NUMA Node 1					NUMA Node 0							
	Instruction	MC3 line state total	MC2 line state total	Bus transactions	NUMA transactions	Instruction	MC1 line state total	MC0 line state total	Bus transactions	NUMA transactions	Directory state total	Sharers list 1	Sharers list 0
0		-	-				-	-			U	0	0
1	**load_linked (in core 3)**	S		1-BusRd(3)	2-RdReq(1->0)					3-Dreply(0->1)	S	1	0
2	**store_conditional (in core 3)**	M		1-BusUpgr(3)	2-UpgrRq(1->0)					3-ack(0->1)	M	1	0
3		S		3-BusRd(hub) Flush(3)	4-Dreply(1->0)	**load_linked (in core 1)**	S		1-BusRd(1)	2-Fetch(0->1)	S	1	1
4		I		3-BusUpgr(hub)	4-ack(1->0)	**store_conditional (in core 1)**	M		1-BusUpgr(1)	2-Invalidate(0->1)	M	0	1

(1) Cache line state: M (modified), S (shared) or I (invalid)
(2) Coherence commands inside NUMA node: BusRd, BusRdX, BusUpgr and Flush
(3): Line state in node memory: M (modified), S (shared) or U (uncached)
(4) Coherence commands between NUMA nodes i and j: RdReq(i→j), WrReq(i→j), UpgrReq(i→j), Dreply(i→j), Ack(i→j), Fetch(i→j) and Invalidate(i→j)

PAR – Final Exam: Part 1 – Course 2020/21-Q1
January 18^{th}, 2021

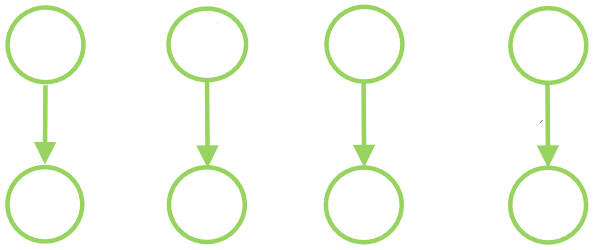
Problem 1 (5 points) Given the following nested loops in a C program instrumented with *Tareador*:

```
#define MAX_ROWS 4
#define MAX_VALUE 8
#define BS 2
   ...
   // compute loops
   for (i=1; i<MAX_ROWS-1; i++)
      for (jj=0; jj<MAX_VALUE; jj+=BS)
      {
          sprintf(stringMessage,"compute(%d,%d)",i,jj);
          tareador_start_task(stringMessage);
          for (j=jj; j<jj+BS; j++)
             A[i][j] = A[i][j] + 2*A[i-1][j] - 2*A[i+1][j]; // Cost 8*tc
          tareador_end_task(stringMessage);
      }
   ...
```

We ask you to answer the following questions:

1. Draw the Task Dependence Graph (TDG) assuming the value for the constants and the *Tareador* task definition in the program. In the TDG, annotate each node with the name of the corresponding task (`compute(i,jj)`) and its cost.

   Solution:



2. Compute the T_1, T_∞ and P_{min} metrics associated to the TDG obtained in the previous question.

   Solution:

   T1=16*8=128
   Tinf=32
   Pmin=4
   Par=4

3. Write the expression that determines the execution time T_4 for the program, clearly indicating the contribution of the computation time T_4(comp) and the data sharing overhead T_4(mov), for the following assignment of tasks to threads and processors: tasks `compute(1,jj)` are assigned to `thread 1` (which runs on processor 1) and tasks `compute(2,jj)` are assigned to `thread 2` (which runs on processor 2); threads 0 and 3, mapped to processors 0 and 3, respectively, have no tasks assigned to them.

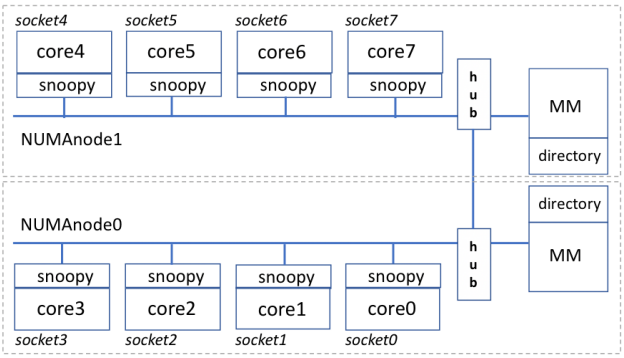
Tasks	thread	processor
none	0	0
compute(1,0), compute(1,2) compute(1,4), compute(1,6)	1	1
compute(2,0), compute(2,2) compute(2,4), compute(2,6)	2	2
none	3	3

You can assume: 1) a distributed-memory architecture with **4 processors**; 2) matrix A is initially distributed by rows (**row i to processor i**); 3) once the loop is finished, you don't need to return the matrix to their original distribution; 4) data sharing model with $t_{comm} = t_s + m \times t_w$, being t_s and t_w the start-up time and transfer time of one element, respectively; and 5) the execution time for a single iteration of the innermost loop body takes $8 \times t_c$.

Solution:

T4= Tcomp + Tmov
T4= 5*16+ Tmov
T4 = 80 + tcomm1 + tcomm2
T4 = + 4*(ts+BS*tw)

Problem 2 (5 points) Assume a multiprocessor system composed of two NUMA nodes, each with four sockets and a shared memory (MM) of 8 GB (total MM size is 16 GB). Each socket has one core with a cache memory of 4 MB. The cache line size is 64 bytes. Data coherence within each NUMA node is guaranteed by a Write-Invalidate MSI protocol with a Snoopy attached to each cache memory; data coherence between the two NUMA nodes is guaranteed by a directory-based MSU protocol.



Coherence commands
- **Core**: $PrRd_k$ and $PrWr_k$ being k the core number doing the action
- **Snoopy**: $BusRd_i$, $BusRdX_i$, $BusUpgr_i$ and $Flush_i$, being i the snoopy/cache number doing the action

Line state in cache
- M (Modified), S (Shared), I (Invalid)

Line state in Main Memory
- M (Modified), S (Shared), U (Uncached)

Cache line size: 64 Bytes
Cache memory size: 4 MB
Main Memory size: 16 GB

1. Compute the total number of bits that are necessary in each cache memory to maintain the coherence between the caches inside a NUMA node.

   Solution: 4*2^20 / 64 = 2^16
   2^17bits

2. Compute the total number of bits that are necessary in each node directory to maintain the coherence among NUMA nodes.

   Solution: 8*2^30 / 64 = 2^27
   2^27*(2+2) = 2^29

3. Given the following declaration for vector `v`:

   ```
   #define N 64
   int v[N];
   ```

   and assuming that: 1) the initial address of vector `v` is aligned with the start of a cache line; 2) vector `v` is entirely allocated in MM_0 (i.e. the portion of shared memory in **NUMA node 0**); 3) the size of an `int` data type is 4 bytes; and 4) all cache memories are empty at the beginning of the program. **We ask you to** fill in the table in the provided answer sheet with the sequence of processor commands (column *Core*), bus transactions within NUMA nodes (column *Snoopy*), *Yes* or *No* in column *Directory* to indicate if there are transactions between NUMA nodes, the presence bits and state in the directory entry associated to the accessed memory line (*Directory entry* columns) and the state for the cache line that keeps a copy of the accessed memory line in each core (last 8 columns), **AFTER the execution of each** of the following memory accesses:

   (a) $core_0$ reads the contents of `v[0]`
   (b) $core_1$ reads the contents of `v[8]`
   (c) $core_0$ writes the contents of `v[0]`
   (d) $core_4$ reads the contents of `v[16]`

   c0 -> N0
   c1 -> N0
   c4 -> N1

   Solution:

   (a)

Responder aquí, no dan la tabla

Command	Coherence actions			Directory entry		State in cache associated to core							
	Core	Snoopy	Directory (yes/no)	Presence bits	State	0	1	2	3	4	5	6	7
$core_0$ reads v[0]	PrRd	BusRd	no	01	S	S							
$core_1$ reads v[8]	PrRd	BusRd	no	01	S	S	S						
$core_0$ writes v[0]	PrWr	BusUpgr	no	01	M	M	I						
$core_4$ reads v[16]	PrRd	BusRd	yes	10	S					S			

(b)

(c)

(d)

4. Given the following code corresponding to a parallel region:

```
#pragma omp parallel num_threads(8)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int i_start = (N / howmany) * myid;
    int i_end = i_start + N / howmany;
    for (int i = i_start; i < i_end; i++)
        v[i] = comp (v[i]);  // comp does not perform any memory access
}
```

After executing the parallel region on the multiprocessor presented before, with all the assumptions in the previous question, the programmer observes that it does not scale as expected. We also know that: 1) $thread_i$ always executes on $core_i$, where $i = [0 - 7]$; 2) inside the function *comp* there are no memory accesses; and 3) vector `v` is the only variable that will be stored in memory (the rest of the variables will all be in registers of the cores). Which performance problem(s) does its execution have? Briefly justify your answer.

Solution:

(a)

(b)

PAR – Final Exam: Part 2 – Course 2020/21-Q1
January 18^{th}, 2021

Problem 1 Given two alternative versions to parallelise the execution of a code fragment:

version 1:

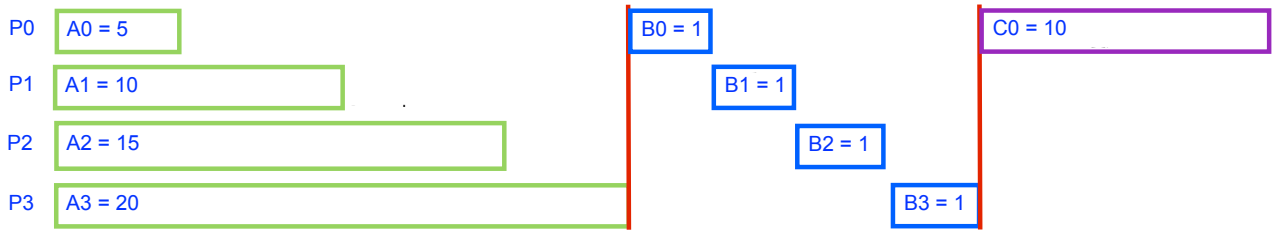
```
// x and l declared here
#pragma omp parallel num_threads(4)
#pragma omp single
{
#pragma omp taskloop grainsize(1)      // Ai
for (int i=0; i<4; i++) {
    l[i] = foo1(i);
    }

for (int i=0; i<4; i++) {
    #pragma omp task depend(inout: x) // Bi
    x += foo2(i, l[i]);
    }
#pragma omp taskwait

#pragma omp task                       // C
x += foo3(l);
}
```

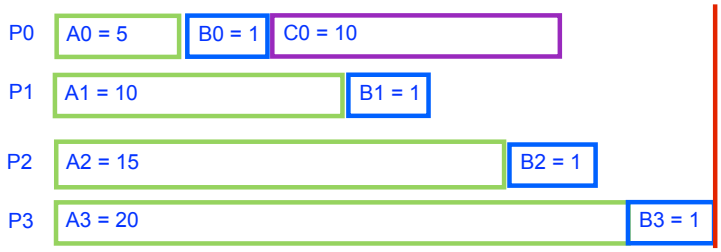
version 2:

```
// x and l declared here
#pragma omp parallel num_threads(4)
#pragma omp single
{
for (int i=0; i<4; i++) {
    #pragma omp task depend(out: l[i])      // Ai
    l[i] = foo1(i);
    }

#pragma omp taskgroup task_reduction(+: x)
    {
    for (int i=0; i<4; i++) {
        #pragma omp task depend(in: l[i])   // Bi
                         in_reduction(+: x)
        x += foo2(i, l[i]);
        }

    #pragma omp task in_reduction(+:x)      // C
    x += foo3(l);
    }
}
```

Assume that 1) tasks generated in the first loop (tasks `Ai`) take $((i+1)\times 5)$ time units to execute; 2) tasks generated in the second loop (tasks `Bi`) take a constant time of 1 time unit to execute; 3) task `C` takes 10 time units to execute; 4) task creation and synchronisation overheads are negligible; and 5) the execution of functions `foo1`, `foo2` and `foo3` do not modify any global variable, i.e. their execution does not produce any data dependence.

1. (2 points) Draw a temporal diagram showing a possible execution on 4 processors for each code version above, obtaining the expression for its execution time T_4.





2. (1.5 points) As you know, the implementation of the `task_reduction` clause in the `taskgroup` construct that is used in *version 2* above requires that each task annotated with `in_reduction`, at the end of its execution, accumulates its local value for variable `x` (let's name it `xlocal`) into shared variable `x`. Write a code excerpt that implements the safe accumulation of the private variable `xlocal` into the global variable `x` using load linked and store conditional instructions:

```
int load_linked (int *addr);
int store_conditional (int *addr, int value);
```

Recall that `store_conditional` returns 0 in case it fails or 1 otherwise. **Note:** You DON'T need to insert the sequence of instructions in *version 2* code above.

```
int success;
do {
int value = load_linked(&x);
value += xlocal;
success = store_conditional(&x, value);
} while (success == 0)
```

Problem 2 We have a code that explores the data stored in a hash table and creates a histogram:

```
#define CACHE_LINE_BYTES 64              // 64 bytes per cache line
#define HT_SIZE 1048576
#define NBINS 128

typedef struct {
  int value;
  elem * next;
} elem;

elem * HashTable[HT_SIZE], p;            // 8 bytes per pointer

int Histogram[NBINS], bin, i, value;     // 4 bytes per integer
...
for (i=0; i<HT_SIZE; i++) {
   p = HashTable[i];
   while ( p != NULL ) {
     value = p->value;
     bin = getbin(value, minval, maxval, NBINS);
     Histogram[bin]++;
     p = p->next;
   }
}
...
```

Let's assume that the minimum and maximum values stored in any element within the hash table are known and kept in variables `minval` and `maxval`, respectively. Routine `getbin` returns the *bin* where a `value` is classified according to the number of bins (`NBINS`) and `minval` and `maxval`. At the end of the computation each position in the histogram, a *bin* (or container), has to count the number of values found in the elements within the hash table for which function `getbin` returns the same value.

We ask you to write two parallel versions using OpenMP. **Note:** In both versions you just need to write the part of the code that shows clearly how the parallelization is done, writing "..." for the rest of the code.

1. (3 points) A *task decomposition* based on the use of **explicit tasks** which incurs low task management overhead. The solution must minimize synchronization overheads and maximize spatial locality in the accesses to vector `HashTable`.

   Solution:

2. (3.5 points) An *output block-cyclic geometric data decomposition* using **implicit tasks**. The solution should avoid *false sharing*.

   Solution:

PAR – Final Exam – Course 2020/21-Q2
June 16^{th}, 2021

Problem 1 (2 points) Given the following nested loops in a C program instrumented with *Tareador*:

```
#define N 4
int A[N][N], B[N][N];
...
// initialization of non-diagonal elements
for (i=1; i<N; i++) {
    sprintf(stringMessage, "initND_%d", i);
    tareador_start_task (stringMessage);
    for (k=0; k<i; k++) {
        A[i][k] = init(i,k); // inner loop body cost = 2*tc
        A[k][i] = A[i][k];
    }
    tareador_end_task (stringMessage);
}
```

1 0
0 1 <- 1 0
2 0
0 2 <- 2 0
2 1
1 2 <- 2 1
3 0
0 3 <- 3 0
3 1
1 3 <- 3 1
3 2
2 3 <- 3 2

```
// initialization of diagonal elements
tareador_start_task ("initD");
for (i=0; i<N; i++) A[i][i] = init (i,i); // inner loop body cost = 1*tc
tareador_end_task ("initD");

// computation phase
for (i=0; i<N; i++) {
    sprintf(stringMessage, "comp_%d", i);
    tareador_start_task (stringMessage);
    for (k=0; k<N; k++) B[i][k] = foo (A[i][k]);  // inner loop body cost = 10*tc
    tareador_end_task (stringMessage);
}
```

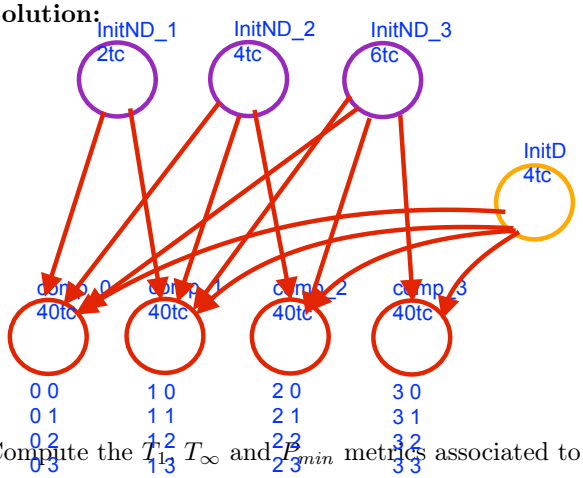
0 0, 1 1, 2 2, 3 3

B0 0 <- A0 0, B0 1 <- A0 1

We ask you to answer the following questions:

1. Draw the Task Dependence Graph (TDG) assuming the given value for constant `N` and the *Tareador* task definitions in the program. In the TDG, annotate each node with the name of the corresponding tasks (`initND_i`, `initD`, `comp_i`) and its cost.

   Solution:



2. Compute the T_1, T_∞ and P_{min} metrics associated to the TDG obtained in the previous question.

   Solution:

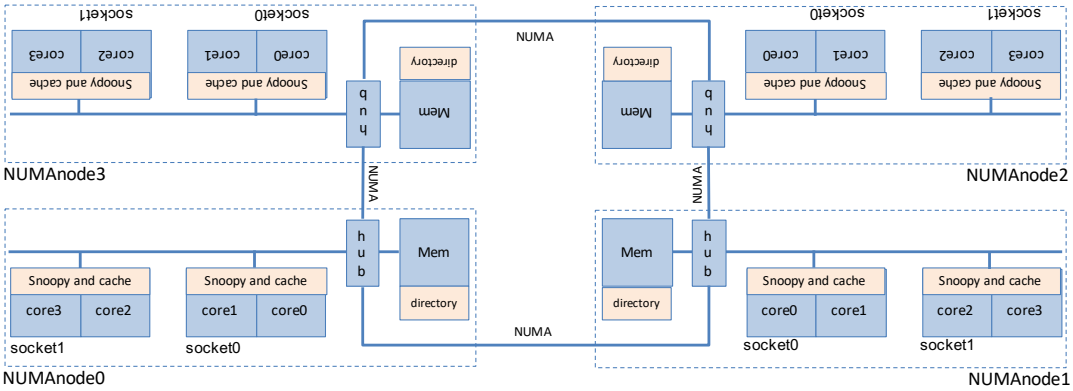
T1 = 176
Tinf = 46
Pmin = 4

3. Determine the assignment of tasks to processors that would yield the best *speed-up* on 4 processors. Calculate T_4 and S_4.

   Solution:

   T4 = Tinf =46
   S4 = T1/T4 = par = 3.83

Problem 2 (1 point) Given the following NUMA system with 4 NUMA nodes, each NUMA node with 2 sockets sharing the access to node memory, and each socket with two cores sharing the access to a per-socket cache. Coherence inside NUMA nodes is maintained with a snoopy–based mechanism implementing the simplest write–invalidate MSI explained in class. Coherence among NUMA nodes is maintained with a directory–based mechanism implementing the simplest write–invalidate MSU explained in class.



Assume that the home node for the line containing variable `var` is NUMAnode0, and at a given time there exist 3 clean copies of that line in cache memories: in socket0 in NUMAnode0, in socket0 in NUMAnode1 and in socket0 in NUMAnode2. Considering that the following memory accesses are done one after the other: 1) core2 in NUMAnode0 reads `var`; 2) core0 in NUMAnode3 reads `var`; and 3) core0 in NUMAnode3 writes `var`. **We ask you to select ONLY the eight sentences that you consider correct from the list below** (labeled from a) to o)). Each correct selection adds 0.125 points; each wrong selection subtracts 0.0625 points; if you select more than 8, only the first 8 will be considered; the grade for this problem is always in the range 0–1.

1. When core2 in NUMAnode0 reads variable `var`, which of the following sentences are correct?

   (a) Core2 issues PrRd.
   (b) The snoopy in socket1 issues BusRd on its local bus.
   (c) The snoopy in socket0 observes the BusRd command and places the line on the bus (Flush).
   (d) The hub associated to NUMAnode0 updates the directory for the line containing `var` to indicate that a new copy of the line exists inside NUMAnode0.
   (e) No coherence requests are sent to the rest of the NUMA nodes in the system.

   Solution:

RdReq -> Dreply
WrReq -> Dreply
UpgrRq -> Ack

Fetch -> Dreply
Invalite -> Ack

2. Then, when core0 in NUMAnode3 reads variable `var`, which of the following sentences are correct?

   (f) The snoopy in socket0 issues BusRd on its local bus.

   (g) The hub associated to NUMAnode3 finds the closest NUMA node that has a copy of the line and sends a RdReq to that NUMA node.

   (h) The hub of the NUMA node receiving the RdReq reads the line from the cache memory that is storing it.

   (i) NUMAnode3 receives a Dreply command with the line containing variable `var` and stores a copy in its main memory.

   (j) At the end the directory in the home NUMA node is updated so that all bits in the sharers list are set to 1 and the state is kept as S

   Solution:

3. Finally, when core0 in NUMAnode3 writes variable `var`, which of the following sentences are correct?

   (k) The snoopy in socket0 of NUMAnode3 issues an Invalidate command on its local bus.

   (l) The hub in NUMAnode3 issues an Invalidate command, going to the home NUMA node.

   (m) The home NUMA node checks if there are copies of the line in other NUMA nodes, sending to each one of them an Invalidate command.

   (n) The state for the line in the caches of the remote nodes receiving the Invalidate command (as well as in the home node) changes from S to I to indicate that the line is nor valid anymore.

   (o) At the end the directory in the home NUMA node only has bit 3 in the sharers list set to 1 and the state set to M.

   Solution:

Problem 3 (3 points) Assume the following sequential code and $N >= 2$ and power of two:

```
#define N (1<<29) // A power of 2 value
typedef struct {
  float max; float min;
} min_max_t;

min_max_t find_min_max_it(float *v, int n) {
     min_max_t min_max;
     float max_duration = v[1]-v[0];
     float min_duration = v[1]-v[0];

     for (int i=2; i<n; i++) {
        float d = v[i] - v[i-1];
        if (d > max_duration) max_duration = d;
        if (d < min_duration) min_duration = d;
     }

     min_max.max = max_duration;
     min_max.min = min_duration;
     return min_max;
}

min_max_t find_min_max_rec(float *v, int n) {
    min_max_t min_max, min_max1, min_max2;

    if (n==2) {
        min_max.max = v[1]-v[0]; min_max.min = v[1]-v[0];
```

```
    } else {
        int n2 = n/2; // n is power of 2
        min_max1 = find_min_max_rec(v, n2);
        min_max2 = find_min_max_rec(v+n2, n2);

        if (min_max1.min < min_max2.min) min_max.min = min_max1.min;
        else min_max.min = min_max2.min;
        if (min_max1.max > min_max2.max) min_max.max = min_max1.max;
        else min_max.max = min_max2.max;
    }
    return min_max;
}

int main() {
    min_max_t min_max_V1, min_max_V2;
    int v1[N], v2[N];
    min_max_V1 = find_min_max_it(v1, N);
    min_max_V2 = find_min_max_rec(v2, N);
}
```

We ask you to answer the following independent questions:

1. Implement an OpenMP parallel version of function `find_min_max_it` and modify the main program as you consider to create an efficient iterative linear task decomposition version of the code. This implementation should avoid synchronizations within the loop and exploit the parallelism with a grainsize bigger than one iteration per task. You are ONLY allowed to use explicit tasks.

   Solution:

2. Implement an OpenMP parallel version of function `find_min_max_rec` and modify the main program as you consider to create an efficient recursive task decomposition version of the code. This implementation should reduce the parallelization overheads due to the generation of tasks controlling it by the depth of the recursivity tree (MAX_DEPTH).

   Solution:

Problem 4 (4 points) Assume the following sequential code fragment implementing a certain computation with matrix `out_matrix` and vector `in_vector`:

```
#define N ... // number of elements in the input vector
#define M ... // number of rows and columns in the output matrix
double in_vector[N];
double out_matrix[M][M];
...
int i, row;
...
for (i = 0; i < N; i++) {
    row = random(M); // random returns a random number between 0 and M-1
    update_row(row, in_vector[i]);
}
```

The following code implements a parallel version for the above loop that uses the so called *"master–worker" paradigm*. In the *"master–worker" paradigm* the "master" thread (only one, thread P in the code below) is the only responsible for assigning work to the "worker" threads (P threads, numbered from 0 to P-1 in the code below, assuming a parallel region executed with P+1 processors). Communication between the "master" thread and a "worker" thread `k` is done through one element of vector `port`, in particular `port[k]`. Through this port `port[k]` the master sends to worker `k` the rows that it has to compute, one after the other, following a specific **output data decomposition** strategy:

```
#define N ... // number of elements in the input vector
#define M ... // number of rows and columns in the output matrix
#define P ... // number of worker threads
double in_vector[N];
double out_matrix[M][M];

typedef struct {
    int row;
    double value;
} Port;
Port port[P];

int i, row, destination;
...
#pragma omp parallel num_threads(P+1)
if (omp_get_thread_num() == P) {
    for (i = 0; i < N; i++) {
        row = random(M); // random returns a random number between 0 and M-1
        destination = thread_to_be_assigned(row, M, P);     // Question 4.1
        port[destination].row= row;
        port[destination].value= in_vector[i];
    }
} else {
```

```
    myid = omp_get_thread_num();
    for ( ; ; ) {
        update_row(port[myid].row, port[myid].value);
    }
}
```

The previous code is not complete since the master and worker threads need some sort of synchronization to ensure the proper assignment of work from master to worker threads. However, you SHOULD NOT WORRY about this issue by now and will address it later.

We ask you to:

1. Implement 3 versions of function `int thread_to_be_assigned(int row, int num_rows, int num_procs)` to implement a:
   (a) *BLOCK* data decompositon, assuming that M is a multiple of P;
   Solution:

   (b) *CYCLIC* data decomposition;
   Solution:

   (c) *BLOCK-CYCLIC* data decompositon, with block size `BS`;
   Solution:

To address the synchronization issue between master and worker threads mentioned before the programmer is proposing to add a new field `ready` to the definition of `Port`, initially set to 0, and two new functions `wait4worker` and `wait4master`, as follows:

```
typedef struct {
    int ready;
    int row;
    double value;
} Port;

Port port[P];

void wait4worker (int num) {
    while (port[num].ready == 1);
    port[num].ready = 1;
}

void wait4master (int num) {
    while (port[num].ready == 0);
    port[num].ready = 0;
}
```

2. Modify the implementation of function `wait4worker` so that its execution is performed atomically (i.e. the read/write of `port[num].ready` is performed atomically), making use of the following atomic primitive:

   - `int test_and_set(int *addr)`: returns the value stored at the memory address pointed by `addr` and sets it to 1;

   Solution:

3. Similarly, modify the implementation of function `wait4master` so that its execution is performed atomically (i.e. ensuring atomicity in the read/write of `port[num].ready`), making use of the following atomic primitives:

   - `int load_linked (int *addr)`: returns the value stored at the memory address pointed by `addr`;
   - `int store_conditional (int *addr, int value)`: tries to write `value` into the memory address pointed by `addr`, returning 1 if it succeeds (no intervening store to that address has taken place since the last call to `load_linked` with the same memory address) or 0 if it fails.

   Solution:

You can assume that functions `test_and_set`, `load_linked` and `store_conditional` are compatible in terms of atomicity. **You do not have to modify the original parallel code to make it correct using functions `wait4worker` and `wait4master`, you simply need to implement an atomic version of these two functions.**

Finally, the programmer wants to avoid the possibility of having false sharing when accessing vector `port`.

4. Why false sharing may happen when accessing to vector `port`? Redefine the last definition of data structure `Port` to ensure that false sharing will not occur, assuming that `int` and `double` data types occupy 4 and 8 bytes, respectivelv, and that cache and memory lines are 64 bytes long,

   Solution: