PAR
Selection of Exams (with Solutions)

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

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

Course 2021-22 (Fall semester)

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Contents

Part I

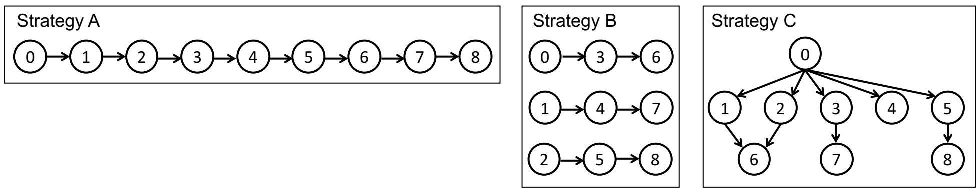
In-term Exams

Some of the exams in this part of this collection of exams are re-edited versions of the ones that originally took place when the course had two mid-term exams. They have been re-edited with the purpose of putting together the problems that correspond with the current mid-term exam.

PAR – In-Term Exam – Course 2018/19-Q2

April 3rd, 2019 and May 29th, 2019

Problem 1 (2.0 points) Given the following task dependence graphs for three different parallelization strategies of a sequential code:



Answer the following questions:

1. (1.0 point) Compute the *Parallelism* and P_{min} metrics for each one of the three dependence graphs assuming that the cost of executing each task is t_c time units.

   Solution: For all strategies $T_1 = 9 \times t_c$. Then for each strategy we have:

   (a) **Strategy A:** $T_\infty = 9 \times t_c$ so *Parallelism*$= T_1 \div T_\infty = 1$; the minimum number of processors to achieve this parallelism is $P_{min} = 1$.

   (b) **Strategy B:** $T_\infty = 3 \times t_c$, *Parallelism*$= 3$ and $P_{min} = 3$.

   (c) **Strategy C:** $T_\infty = 3 \times t_c$, *Parallelism*$= 3$ and $P_{min} = 4$.

2. (1.0 point) Assuming a multiprocessor with $P = 3$ processors and the following mapping of tasks to processors for each strategy:

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

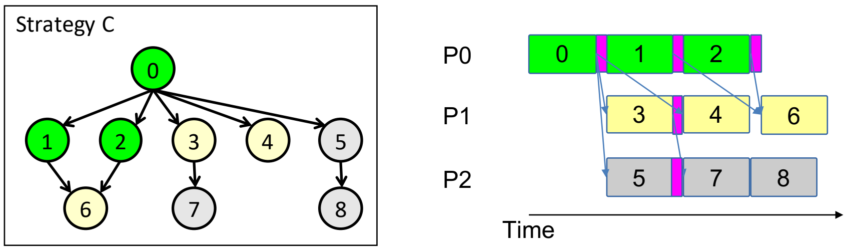
   Obtain the general expression for the speed–up $S_{P=3}$ for each strategy and associated mapping, assuming that there is an overhead related with task synchronisation of t_{synch} time units, i.e. the overhead that a task has to pay to signal to ALL its successor tasks that it has finished.

   Solution: For each parallel strategy we have:

   (a) **Strategy A:** $T_3 = 9 \times t_c + 8 \times t_{synchr}$; therefore $S_3 = (9 \times t_c) \div (9 \times t_c + 8 \times t_{synchr})$.

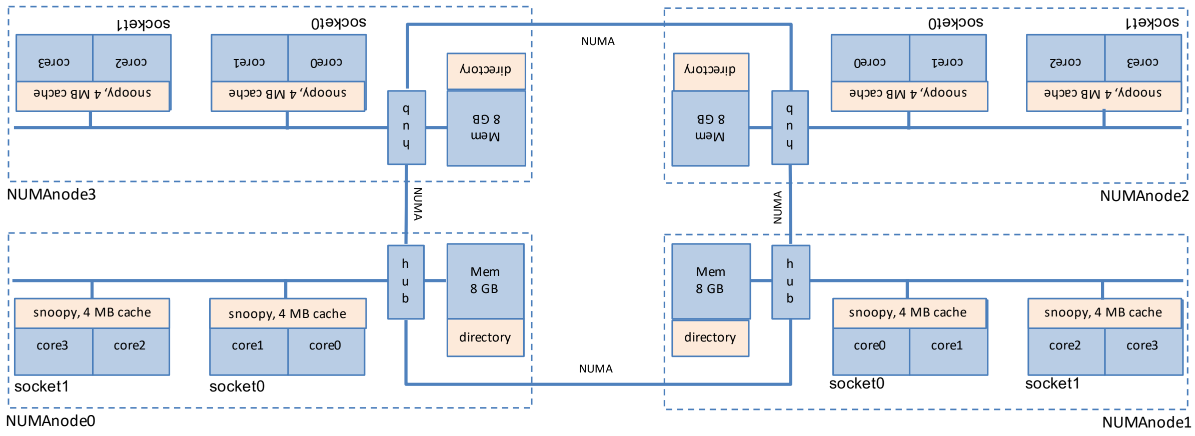
   (b) **Strategy B:** $T_3 = 3 \times t_c + 2 \times t_{synchr}$; therefore $S_3 = (9 \times t_c) \div (3 \times t_c + 2 \times t_{synchr})$.

   (c) This figure shows the timeline with the parallel execution of the tasks and the synchronisation overheads between tasks that are necessary.



Therefore, for **Strategy C:** $T_3 = 4 \times t_c + 3 \times t_{synchr}$; and therefore $S_3 = (9 \times t_c) \div (4 \times t_c + 3 \times t_{synchr})$.

Problem 2 (3.0 points) Given the following NUMA system with 4 dual-socket NUMA nodes, each NUMA node with 8 GB of main memory. Each socket has two cores sharing the access to a per-socket 4 MB 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.



coreX: core X within a socket. 16 cores in total.
socketY: socket Y within NUMA node: each socket with 2 cores, sharing 4 MB cache; MSI snoopy coherence protocol.
NUMAnodeW: NUMA node W; each node with two sockets sharing 8 GB of main memory and a hub/directory to keep coherence among nodes, MSU simplified protocol.
Size of memory and cache lines: 64 bytes.

Part I (1 point): Compute the total number of bits that are necessary **in the whole system** to:

1. Maintain the coherence among NUMA nodes.

   Solution: In each NUMA node the memory is able to store $8GB \div 64$ memory lines. This is $2^3 \times 2^{30} \div 2^6 = 2^{27}$ lines. For each line in memory the directory needs to store 2 bits to keep the state of the line (MSU) and 4 presence (sharers) bits (one per NUMA node); so 6 bits per memory line. In total for the whole system this is $4nodes \times 2^{27} lines/node \times 6bits/line = 3 \times 2^{30}$ bits.

2. Maintain the coherence within NUMA nodes.

   Solution: Each cache memory inside a NUMA node is able to store $4MB \div 64$ memory lines. This is $2^2 \times 2^{20} \div 2^6 = 2^{16}$ lines. For each line in the cache the snoopy needs to store 2 bits to keep the state of the line (MSI). In total for the whole system this is $4nodes \times 2caches/node \times 2^{16} lines/cache \times 2bits/line = 2^{20}$ bits.

Part II (2 points): 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`; 3) core0 in NUMAnode3 writes `var`; and 4) core0 in NUMAnode0 writes variable `other`. We ask you to select the sentences that are correct for each one of these 4 memory accesses (for each memory access at least one of the sentences is correct). Each correct selection adds 0.25 points; each wrong selection subtracts 0.17 points; the grade for this part of the problem is always in the range 0–2.

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: True, True, False, False, True

2. Then, when core0 in NUMAnode3 reads variable `var`, which of the following sentences are correct?
   (a) The snoopy in socket0 issues BusRd on its local bus.
   (b) 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.
   (c) The hub of the NUMA node receiving the RdReq reads the line from the cache memory that is storing it.
   (d) NUMAnode3 receives a Dreply command with the line containing variable `var` and stores a copy in its main memory.
   (e) 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: True, False, False, False, True

3. Then, when core0 in NUMAnode3 writes variable `var`, which of the following sentences are correct?
   (a) The snoopy in socket0 of NUMAnode3 issues an Invalidate command on its local bus.
   (b) The hub in NUMAnode3 issues an Invalidate command, going to the home NUMA node.
   (c) 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.
   (d) 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 M to indicate that the line is modified somewhere else.
   (e) 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: False, False, True, False, True

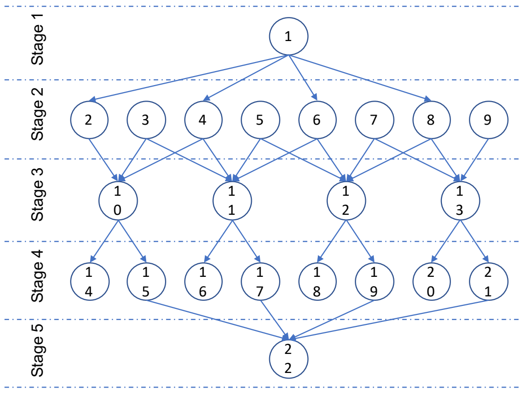
4. Finally, core0 in NUMAnode0 writes variable `other` (observe that we are not giving any information about the home NUMA node for the line containing this variable). When this memory accesses finishes the state in the directory for the line containing variable `var` is kept to M, but the bit set to 1 in the sharers list moves from position 3 to position 0. Which of the following sentences are correct?
   (a) You should tell me which is the home NUMA node for variable `other` in order to decide if the following sentences are correct or not.
   (b) This state in the directory is not possible since `var` and `other` are different variables.
   (c) Since both variables reside in different cache lines, this state is possible if both variables are mapped in the same cache line entry (cache line replacement, i.e. the line containing `var` is replaced by the line containing `other`).
   (d) This is the typical symptom of false sharing.
   (e) Since variables `var` and `other` are different, there is no need to interchange coherence commands between NUMAnode0 and NUMAnode3.

   Solution: False, False, False, True, False

PAR – In-Term Exam – Course 2019/20-Q1

November 6th, 2019 and December 18th, 2019

Problem 1 (5 points) Given the following Task Dependence Graph (TDG) representing the task decomposition of a program that has 5 computational stages:



Computation stage	Number of tasks	Weight for each task
Stage 1	1	N x t_c
Stage 2	2 x K	(N / (2 x K)) x t_c
Stage 3	K	(N / K) x t_c
Stage 4	2 x K	(N / (2 x K)) x t_c
Stage 5	1	N x t_c

In general the TDG has K tasks in stage 3 (each with a computational cost of $\frac{N}{K} \times t_c$), and $2 \times K$ tasks in stages 2 and 4 (each with a computational cost of $\frac{N}{2 \times K} \times t_c$). The TDG always has a single task in stages 1 and 5 (with computational cost $N \times t_c$). K is the degree of the TDG ($K \geq 1$) and the TDG on the left is the particular case when $K = 4$ (numbers inside nodes are used to enumerate tasks). The edges show the dependences between tasks. Observe that only K tasks in stage 2 depend on the task in stage 1; similarly, the task in the last stage only depends on K tasks in stage 4; dependences between tasks in stages 2 and 3, and between tasks in stages 3 and 4 always follow the pattern shown on the TDG above.

1. (1 point) Compute the expression for metrics T_1, T_∞ and $Parallelism$, as a function of K and N.

   Solution: T_1 is the sum of the computational costs of all tasks in the TDG. In each stage the cost is $N \times t_c$, so in total $T_1 = 5 \times N \times t_c$. T_∞ is determined by the sum of the computational costs of the nodes that belong to the critical path, which is this case traverses one of the tasks in each computational stage; therefore $T_\infty = (N + \frac{N}{2 \times K} + \frac{N}{K} + \frac{N}{2 \times K} + N) \times t_c = 2 \times N \times (1 + \frac{1}{K}) \times t_c$. Finally, $Parallelism = \frac{T_1}{T_\infty} = \frac{5 \times K}{2 \times (K+1)}$.

2. (0.5 points) Compute the minimum number of processors P_{min} that are necessary to achieve the $Parallelism$ obtained in the previous question, again as a function of K and N.

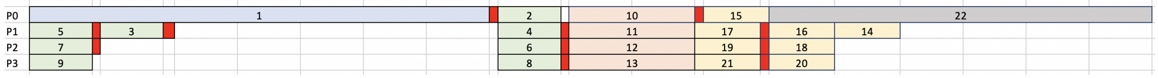
   Solution: For $K > 1$, the minimum number of processors is $P_{min} = K$. In order to not delay the critical path, we need 1 processor to execute the task in the first stage and 1 more processor to execute the K initially independent tasks in stage 2 one after the other; then K processors can execute the rest of tasks in stage 2, the K tasks in stage 3 and the K tasks in stage 4 that have a dependence with the last task in stage 5; finally 2 processors again needed to execute the rest of tasks in stage 4 (one after the other) and the last task in stage 5. For $K = 1$ the minimum number of processors is $P_{min} = 2$, one to execute the critical path and the other the rest of nodes outside it.

Assume that the synchronisation overhead associated to the edges in the TDG is 0 when the source and destination tasks are executed by the same processor, and t_{sync} when the source and destination tasks are executed by two different processors; also assume that if a task has several outgoing edges, the overhead t_{sync} is paid only once for all of them.

3. (1 point) For $K = 4$ (i.e. for the TDG shown above), indicate the assignment (mapping) of tasks to $P = K = 4$ processors that minimises the computation time and incurs the minimum synchronisation

overhead. Draw the temporal diagram that shows how the tasks assigned to each processor are executed along time.

Solution: Following the task numbering shown in the TDG, a possible assignment of tasks to 4 processors would be {1, 2, 10, 15, 22} to processor P0, {4, 5, 11, 16, 17} to P1, {6, 7, 12, 18, 19} to P2 and {8, 9, 13, 20, 21} to P3. There are two tasks pending to be assigned (3 and 14) that if we assign them to P0 then we increase the critical path; they should be assigned to any other processor, for example to P1, which then would have assigned {3, 4, 5, 11, 14, 16, 17}. The following picture shows then resulting execution timeline.



Observe that the following synchronisations affect the execution of the critical path in P0: after the execution of task 1 (to signal {4, 6, 8}), after tasks {4, 6, 8} to signal {10, 11, 12}, and after {17, 19, 21} to signal task 22. The other synchronisations are hidden by the execution of tasks and do not introduce any additional delay in the critical path.

4. (1.5 points) Obtain the expression for T_4 that would be obtained (for $K = 4$ as a function of N), including both computation as well as synchronisation overhead.

   Solution: Regarding the computation part, $T_4 = (2 \times N + 2 \times \frac{N}{8} + \frac{N}{4}) \times t_c = \frac{5}{2} \times N \times t_c$; that is the same expression for T_∞ above when $K = 4$. Regarding overheads, from the previous temporal diagram, three synchronisations between processors occur in the critical path. Therefore, the expression for T_4 is $T_4 = (\frac{5}{2} \times N \times t_c) + (3 \times t_{synch})$.

Problem 2 (5 points) Assume the following serial code computing a two–dimensional NxN matrix u:

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

The code is parallelised on three processors (P_{0-2}) with the assignment of iterations to processors shown on the left part of Figure 1.

Observe that each processor is assigned the computation of $N/3$ consecutive iterations of the i loop (except iterations 0 an N-1), starting with processor P_0 (the number of processors 3 perfectly divides the number of rows and columns N); each processor executes its assigned computation in 3 tasks, each one computing $N/3$ consecutive iterations of the j loop. Due to the dependences in this code, you should already know that for example $task_{11}$ can only be executed by P_1 once processor P_0 finishes with the execution of $task_{01}$ and the same processor (P_1) finishes with the execution of $task_{10}$.

The three processors compose a multiprocessor architecture with 3 NUMA nodes sharing the access to main memory, each NUMA node with a single processor P_p, a cache memory (of sufficient size to store all the lines required to execute all tasks) and portion of main memory M_p (p in the range 0–2). Each memory M_p has an associated directory to maintain the coherence at the NUMA level. Each entry in the directory uses 2 bits for the status (M, S and U) and 3 bits in the *sharers list*. The rows of matrix u are distributed among NUMA nodes as shown on the right part of Figure 1. In that figure rectangles represent the memory lines that are involved in the computation of $task_{11}$, for a specific case in which N=24 and each cache line is able to host 4 consecutive elements of matrix u.

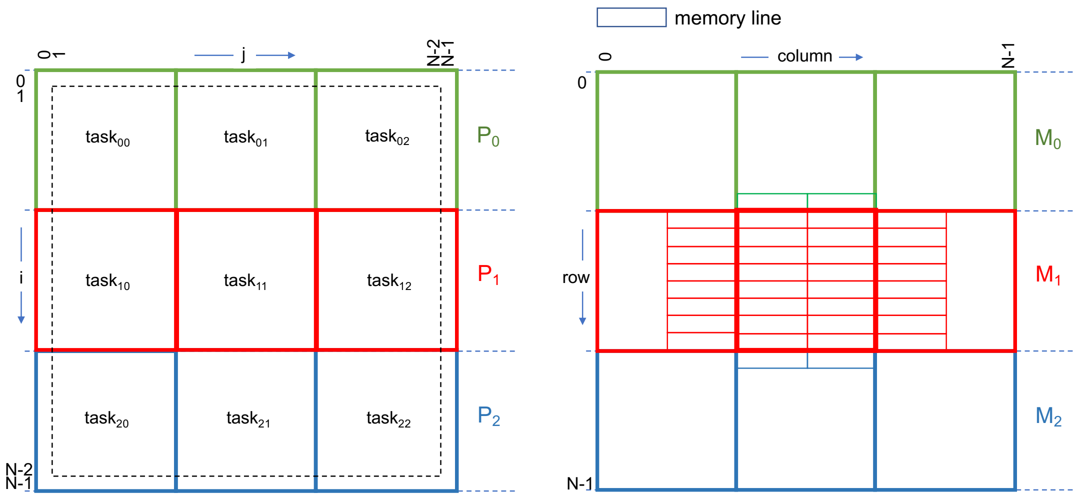


Figure 1: (Left) Assignment of iterations to processors. (Right) Mapping of array elements to memory modules in the NUMA system.

We ask you:

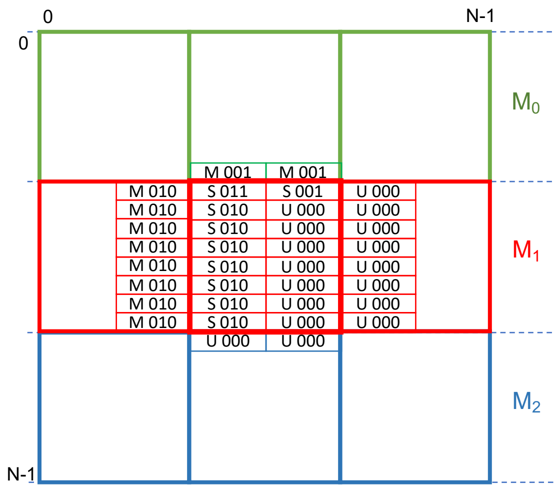
1. Assuming U status for all memory lines at the beginning of the parallel execution, which will be the contents in the directories for the lines shown on the right part of Figure 1 when $task_{11}$ is ready for execution? Use the provided answer sheet for this question, indicating for each memory line the contents of the directory (e.g. S011 meaning that the line is in S state with copies in NUMA nodes 1 and 0).

2. Indicate the sequence of coherence actions (*RdReq, WrReq, UpgrReq, Fetch, Invalidate, Dreply, Ack* or *WriteBack*) that will occur when processor P_1 executes the first iteration in $task_{11}$, i.e. the iteration in the upper left corner of $task_{11}$ on the left part of Figure 1.

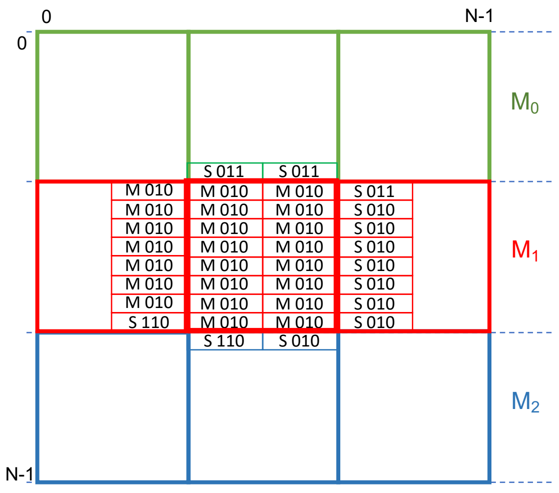
3. Which will be the contents in the directories for the same lines once $task_{11}$ finishes its execution? At that time you should assume that $task_{02}$ and $task_{20}$ have also finished their execution. Use the provided answer sheet for this question.

   Solution: the figure on the left represents the contents in the directories before the execution (question 1.1); the figure on the right after the execution (question 1.3).

Solution for question 1.1 (before execution)



Solution for question 1.3 (after execution)



Regarding question 1.2, processor P_1 first performs several read accesses, one of them to a memory position in state M in memory M_0; to read it, P_0 issues $RdReq_{1\rightarrow 0}$ to the home node M_0, which responds with the contents of the line and a $Dreply_{0\rightarrow 1}$. After the computation, P_1 has to write one element for which it is the home node; since the line containing that element is in S state, with copy in P_0's cache, M_1 has to send and $Invalidate_{1\rightarrow 0}$ command, which is acknowledged with $Ack_{0\rightarrow 1}$.

PAR – In-Term Exam – Course 2020/21-Q1

November 6th, 2020

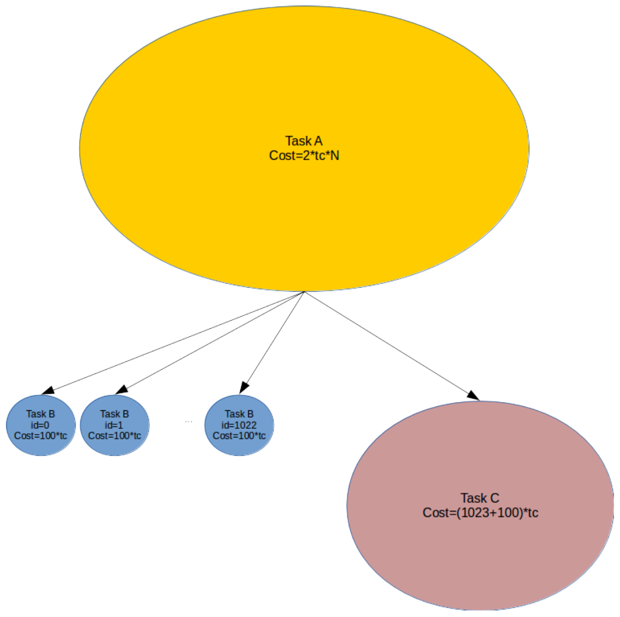
Problem 1 (2 points) Consider the following code excerpt (including *Tareador* task definitions):

```
   #define N         ((100*1024)+1023)
   #define NUM_TASKS 1024
   int A[N], B[N], C[N];
   int num_elems_task;
   ...
   tareador_start_task("TaskA");
   for (int i=0; i<N; i++) {
      A[i] = i * 10;
      B[i] = i * 10;
   }
   tareador_end_task("TaskA");

   // NOTE: integer division truncates the result to the nearest integer towards 0
   num_elems_task = N / NUM_TASKS;

   // all but last task
   for (int n_task=0; n_task<NUM_TASKS-1; n_task++) {
      tareador_start_task("TaskB");
      for (int ii=n_task*num_elems_task; ii<(n_task+1)*num_elems_task; ii++) {
         C[ii] = A[ii] * B[ii];
      }
      tareador_end_task("TaskB");
   }

   // last task doing all remaining iterations
   tareador_start_task("TaskC");
   for (int ii=(NUM_TASKS-1)*num_elems_task; ii<N; ii++) {
         C[ii] = A[ii] * B[ii];
   }
   tareador_end_task("TaskC")
   ...
}
```

Assume that the execution time for each iteration of the loop in `TaskA` is $2 \times t_c$ time units and the execution time for each iteration of the loops in `TaskB` and `TaskC` is t_c time units. **We ask you to answer the following questions:**

1. Draw the Task Dependence Graph (TDG) based on the above *Tareador* task definitions. **Notes:** 1) you can use "..." in case you have to draw too many identical tasks, clearly indicating the number of identical tasks; 2) identify each `TaskB` you draw with its n_task number; and 3) include for each task you draw its execution cost based on the number of iterations it executes and the cost of the loop body.

Solution:



2. Compute the expression for the T_1, T_∞ and P_{min} metrics for the task decomposition expressed above.

   Solution:

   $T_1 = 2 \times t_c \times N + t_c \times N = 3 \times t_c \times N$,

   $T_\infty = Task_A + Task_C = 2 \times t_c \times N + 1123 \times tc = (2 \times t_c \times (100 \times 1024 + 1023) + t_c \times (1023 + 100))$

   $P_{min} = 94$

   Detail of P_{min}:

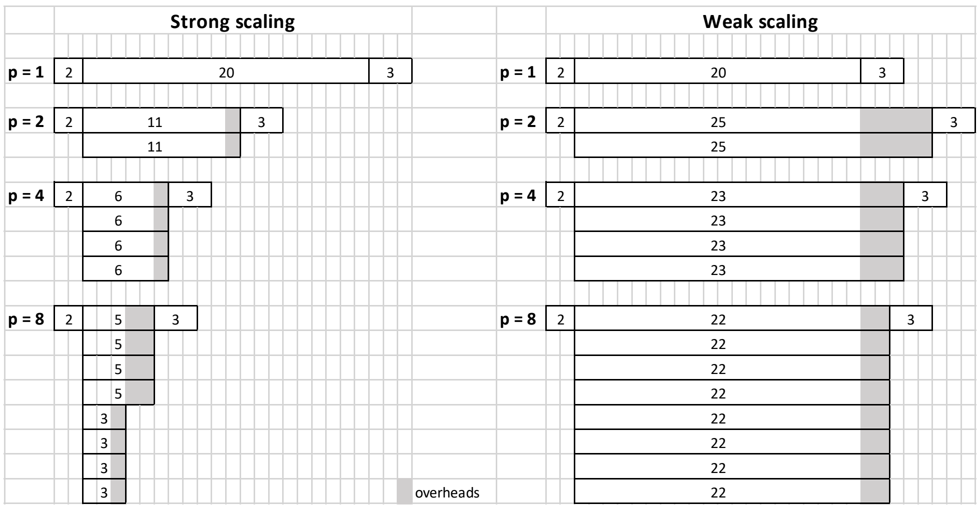
   To achieve T_∞ we need 1 processor to execute the critical path (task A and task C) and one or more processors to execute all instances of tasks B in parallel with task C.

   Let's define $N_{chunk\_of\_tasks\_B}$ as the number of tasks B that can be run in the time taken to execute task C. This number can be computed as the cost of task C divided by the cost of one task B: $N_{chunk\_of\_tasks\_B} = (1123 \times t_c \div (100 \times t_c)) = 11$. Therefore, while one processor is executing task C another processor can execute 11 tasks B. So how many processors do we need to execute 1023 task B's in parallel with task C? This number results of dividing the number of tasks B (1023) by $N_{chunk\_of\_tasks\_B}$: $1023 \div N_{chunk\_of\_tasks} = 1023 \div 11 = 93$. This means we need 93 processors to execute the 1023 tasks B while one processor executes task C. Therefore:

   $P_{min} = 1 + 1023 \div N_{chunk\_of\_tasks\_B}$;

   $P_{min} = 1 + 1023 \div (1123 \times t_c \div (100 \times tc)) = 1 + 1023 \div 11 = 1 + 93 = 94$

Problem 2 (2 points) Consider the following execution timelines to analyse the *strong* and *weak* scalability efficiencies for a parallel application. Each timeline, for a given number of processors p, shows the computation bursts executed by each processor and their duration (indicating in grey the overheads due to parallel execution, already included in the indicated burst duration):



We ask you to:

1. Calculate T_{seq}, T_{par} and the parallel fraction φ of the application.

   Solution: $T_{seq} = 5$, $T_{par} = 20$ and $\varphi = T_{par} \div (T_{seq} + T_{par}) = 20 \div 25 = 0.8$.

2. Assuming the usual definition for the speed–up for the strong scaling case ($S_p = T_1 \div T_p$) and the corresponding definition for weak scaling ($S_p = (T_{seq} + p \times T_{par}) \div T_p$), compute the values of the speed-up S_2, S_4 and S_8 for both strong and weak scaling.

   Solution: For strong scaling: $S_2 = T_1 \div T_2 = (2 + 20 + 3) \div (2 + 11 + 3) = 1.56$; $S_4 = T_1 \div T_4 = 25 \div (2+6+3) = 2.27$; and $S_8 = T_1 \div T_8 = 25 \div (2+5+3) = 2.5$. And for weak scaling $S_2 = (T_{seq} + 2 \times T_{par}) \div T_2 = (5+2\times 20) \div (2+25+3) = 1.5$; $S_4 = (T_{seq} + 4 \times T_{par}) \div T_4 = (5+4\times 20) \div (2+23+3) = 3.03$; and $S_8 = (T_{seq} + 8 \times T_{par}) \div T_8 = (5 + 8 \times 20) \div (2 + 22 + 3) = 6.11$.

3. Based on the previous numbers, reason what scalability type (weak and/or strong) would you suggest to use for this application in order to maximise its efficiency?

   Solution: Based on the results from the previous section one should suggest to use weak scaling.

Problem 3 (4 points) Assume a multiprocessor architecture with 24 GB (gigabytes, i.e. 2^{30} bytes) of main memory and 16 processors, each one with a private cache of 256 MB (megabytes, i.e. 2^{20} bytes); memory and cache lines are 64 bytes wide.

1. If the multiprocessor system is designed as a "pure" UMA architecture with snoopy–based write–invalidate MSI, calculate the **total number of MB** that are required **in the whole system** to store the information necessary to keep cache coherence.

   Solution: For the "pure" UMA configuration, one needs 2 state bits (MSI) for each line of cache memory. Each cache memory has $(256 \times 2^{20}) \div 64$ lines, that is 2^{22} lines; therefore the number of bits per cache is $2^{22} \times 2 = 2^{23}$ bits, that translated into MB is $(2^{23} \div 8) \div 2^{20} = 1$ MB. Since there are 16 processors, each one with its own private cache, the whole system needs 16 MB to keep cache coherence.

2. If the multiprocessor system is designed as a "hybrid" NUMA/UMA architecture with 4 identical nodes, each node with 4 processors sharing the access to $1/4^{th}$ of the total main memory, combining directory–based write–invalidate MSU among nodes and snoopy–based write–invalidate MSI within each node, calculate the **total number of MB** that are required **in the whole system** to store the necessary information to keep cache coherence.

   Solution: For the "hybrid" NUMA/UMA configuration, one needs 2 state bits (MSU) and only 4 presence bits for each line of main memory. For the overall 24 GB, this is $(24 \times 2^{30}) \div 64$ lines, that is 3×2^{27} lines; therefore the number of bits in the directory is $3 \times 2^{27} \times (4+2) = 9 \times 2^{28}$ bits, that translated into MB is $9 \times (2^{28} \div 8) \div 2^{20} = 9 \times 2^5 = 288$ MB. We also have to add the 16 MB required by the coherence in cache memories, which does not change in all system configurations. So in total $288 + 16 = 304$ MB.

For the hybrid UMA/NUMA architectural design with 4 nodes and 4 processors per node described just above, let's consider the execution of the following parallel region:

```
#define NUM_THREADS 16
#define N 1024*1024

char value[N];

typedef struct {
    int a[NUM_THREADS];
    int b[NUM_THREADS];
    int c[NUM_THREADS];
} statistics;

// initialisation of vector value omitted

#pragma omp parallel num_threads(NUM_THREADS)
{
    int whoamI = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    if (whoamI == 0)
        for (int i=0; i<NUM_THREADS; i++) {
            statistics.a[i] = 0; statistics.b[i] = 0; statistics.c[i] = 0;
         }
    #pragma omp barrier // all other threads wait here for thread 0 to finish

    for (int i=whoamI; i<N; i+=howmany) {
        if (value[i]=='a') statistics.a[whoamI]++;
        if (value[i]=='b') statistics.b[whoamI]++;
        if (value[i]=='c') statistics.c[whoamI]++;
    }
}
```

Assuming that 1) the integer data type occupies 4 bytes; 2) the initial addresses for both vector `value` and structure `statistics` are aligned with a cache line boundary (i.e. the first element is stored at the beginning of a cache line); and 3) $thread_i$ is executed by processor i:

3. **At the end of the parallel region**, how many entries in the directory will be used for variable `statistics`? Can you tell in which NUMA node(s) will they be stored? Briefly reason both answers.

   Solution: The `statistics` variable occupies $3 \times 16 \times 4$ bytes, that is 3 memory lines in total (64 bytes per cache line). In fact, each element of the struct occupies one cache line. Therefore `statistics` will use 3 entries in the directory. Since the initialization is performed by thread 0, the processor that executes it (processor 0) will have it in its slice of the directory.

4. **At the end of the parallel region**, how many valid copies in cache will exist for the cache lines holding variable `statistics`? Can you tell in which NUMA node(s) will they be cached? Briefly reason both answers.

   Solution: Since variable `statistics` is written there can only exist one valid copy of its 3 lines at the end of the parallel region. Each of these three lines will be stored in the cache memory of the last processor that updated its counter.

5. When the programmer compiled and executed the program, he/she detected a performance problem with the following diagnosis: "very large amount of cache coherence traffic caused by the way variable `statistics` is defined". In order to solve it the programmer decided to change the definition of the variable `statistics` from *SoA (Struct of Arrays)* to *AoS (Array of Structs)*, as follows:

   ```
   typedef struct {
       int a;
       int b;
       int c;
   } statistics[NUM_THREADS];
   ```

   with the appropriate changes in the access to it in the program. However the performance problem persisted. **Complete the definition of variable `statistics`, or propose an alternative one,** in order to reduce coherence traffic.

   Solution: The performance problem that is referred in the statement is *false sharing.* In order to avoid it, the programmer should add some padding to the definition of the `struct` so that each element of the vector occupies a complete cache line. Since a cache line can hold $64 \div 4 = 16$ integers, and 3 of them are taken by fields `a`, `b` and `c`, we need to insert 13 elements, for example in the form of a dummy vector, as follows:

   ```
   #define NUM_THREADS 16

   typedef struct {
       int a;
       int b;
       int c;
       int dummy[13];   // padding
   } statistics[NUM_THREADS];
   ```

   With that, variable statistics will use 16 complete lines instead of the original 3.

Problem 4 (2 points) We have a distributed memory architecture in which accessing the data stored in a remote processor through an interconnexion network implies an associated cost of $t_s + m \times t_w$ (being t_s the *start-up* time, m the number of elements being accessed, and t_w the *per-element* transfer time). A processor can only perform one remote access at a time, serve one remote access at a time, but can do both of them at the same time. In our model we assume that local accesses take zero overhead.

Given the following code:

```
for (i=2; i<N; i++)
  for (j=0; j<N-2; j++)
   {
     A[i][j] = ( A[i-1][j] + A[i-2][j] + A[i][j] + A[i][j+1] + A[i][j+2] ) / 5;
   }
```

and assuming that 1) we have P processors; 2) N is very large; 3) $N >> P$; 4) the execution of one iteration of the inner loop body takes t_c time units; 5) in the definition of tasks, please use BS as the block size when defining the granularity of tasks if necessary to avoid task serialization (*blocking*); and 6) N is divisible by both P and BS; **we ask you** to model the parallel execution time with P processors (T_p) for a *Row distribution* of matrix A (i.e. it is distributed so that each processor has N/P consecutive rows).

Solution:

Elements $A[i][j+1] + A[i][j+2]$ are in the same row. Thus, they do not require initial communication. Elements $A[i-1][j] + A[i-2][j]$ are in the previous two rows. When those rows belong to another processor we need to wait until the other processor has computed them. We define tasks which compute $\frac{N}{P} \times BS$ elements of the matrix ($\frac{N}{P}$ rows and BS columns) to avoid the serialization that would appear if a task was computing each row only after computing the whole previous one. Note that in each synchronization step we have to perform two remote accesses of size BS. Since N is very large we can consider $N-2 \simeq N$. Then, $T_P = T_{calculations} + T_{remote\_accesses} = (\frac{N}{BS} + P - 1) \times (\frac{N}{P} \times BS) \times t_c + (\frac{N}{BS} + P - 2) \times 2 \times (t_s + BS \times t_w)$

PAR – In-Term Exam – Course 2020/21-Q2

April 21st, 2021

Problem 1 (1.5 points) Given the following code:

```
#define N 8

for(i=0;i<N;i++) {
  tareador_start_task("task-b");
  b[i] += foo(i);
  tareador_end_task("task-b");
}

for(i=1;i<N-1;i++) {
  tareador_start_task("task-a");
  a[i] = b[i-1] + b[i+1];
  tareador_end_task("task-a");
}
```

Assume that 1) the cost of executing one instance of the loop body for the two loops is t_c; and 2) the parallel task decomposition strategy is the one indicated with *Tareador* annotations. **We ask you to answer** the following questions:

1. How many tasks of type `task-b` and `task-a` are created?

   Solution:

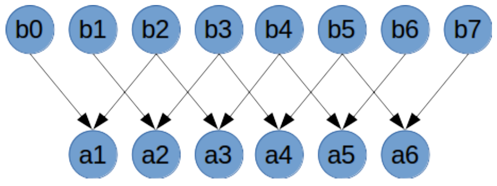
   The first loop iterates from 0 to $N-1$; one task is created per iteration. Therefore, N tasks of type `task-b` are created.

   The second loop iterates from 1 to $N-2$; one task is created per iteration. Therefore, $N-2$ tasks of type `task-a` are created.

2. Draw the TDG of the parallel code, indicating the cost for each task and dependences between tasks.

   Solution:

   Each `task-a` created in iteration i is shown in the TDG with a_i. Each `task-b` created in iteration i is shown in the TDG with b_i. Each task has a cost of t_c.

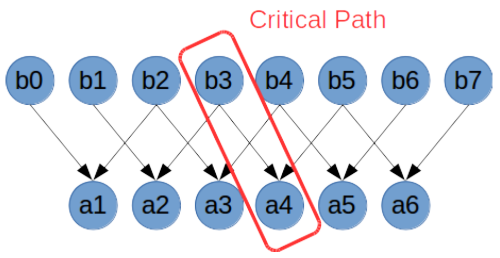


3. Compute T_1, T_∞ and P_{min}.

   Solution:

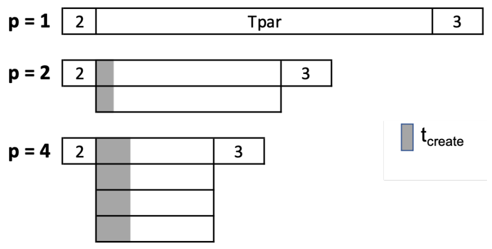
   Given the previous TDG:

   - $T_1 = N \times t_c + (N-2) \times t_c = (2N-2) \times t_c$.
   - $T_\infty = 2t_c$. One `task-b` plus one `task-a` as it is shown in the following figure with one of the possible critical paths.



- $P_{min} = N$. This is because all tasks (N) of type `task-b` should be able to be executed at the same time.

Problem 2 (1.5 points) Consider the following execution timelines that show the *strong scaling* behaviour of a simple parallel program composed of a single parallel region. The first timeline corresponds with the execution on a single processor, showing the sequential ($T_{seq} = 2 + 3$) and parallel (T_{par} unknown) execution bursts that define T_1. The second and third timeline correspond with the execution on two and four processors, respectively, in which we added the overhead of task creation $t_{create}(p) = \alpha \times p$ (which is proportional to the number of processors used in the parallel execution); therefore, $T_p = T_{seq} + \frac{T_{par}}{p} + t_{create}(p)$.



We ask you to:

1. Calculate the value for T_{par} assuming that $\varphi = \frac{4}{5}$.

   Solution: $T_1 = T_{seq} + T_{par}$ and $\varphi = T_{par} \div T_1$. Since $T_{seq} = 5$ and $\varphi = \frac{4}{5}$, one can easily obtain $T_{par} = 20$.

2. Obtain the value for the proportionality constant α assuming that $S_2 = 1.65$.

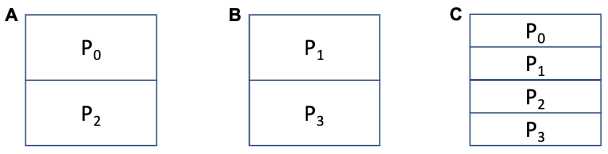
   Solution: $S_p = T_1 \div ((1 - \varphi) \times T_1 + (\varphi \times (T_1/p) + (\alpha \times p)))$. From this expression applied to $p = 2$ one can easily get $\alpha = 0.075$.

3. Obtain the value for S_4.

   Solution: From the same expression, by simply substituting the value for α just obtained one can get $S_4 = 2.42$.

Problem 3 (2 points) Consider a distributed memory architecture in which accessing the data stored in a remote processor through an interconnection network implies a data sharing overhead of $t_s + m \times t_w$ (being t_s the *start-up* time, m the number of elements being accessed, and t_w the *per-element* transfer time). At any time, a processor can simultaneously perform one remote access and serve one remote access, but only one of each kind. The data sharing model assumes that local accesses take zero overhead.

Assume the following simple parallel region to be executed with 4 processors and the distribution of matrices shown in the following figure:



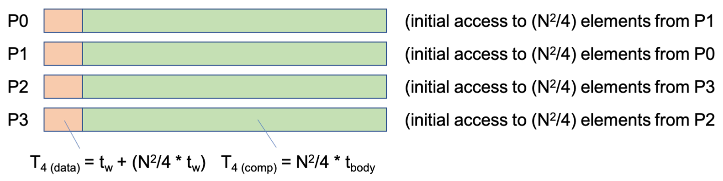
```
#define N 64
#define N_THREADS 4
int A[N][N], B[N][N], C[N][N];
...
#pragma omp parallel num_threads(N_THREADS)
{
    int thid = omp_get_thread_num();
    int chunk = N / N_THREADS;

    for (int i = thid * chunk; i < (thid + 1) * chunk; i++)
        for (int j = 0; j < N; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

with processor P_x executing the OpenMP thread with identifier x; each processor can start the execution of its thread as soon as it has all the data elements that are needed to execute the thread. The execution time for one iteration of the loop body is t_{body}.

We ask you:

1. Draw a temporal diagram (timeline) showing the execution of the threads by each processor and the data sharing overheads that occur.

   Solution: Each processors needs to access to half of the elements stored in another processor, incurring in a data sharing overhead. After that, each processor can proceed with the execution of its computation burst.



P0
P1
P2
P3
(initial access to (N²/4) elements from P1
(initial access to (N²/4) elements from P0
(initial access to (N²/4) elements from P3
(initial access to (N²/4) elements from P2
T4 (data) = tw + (N²/4 * tw)
T4 (comp) = N²/4 * tbody

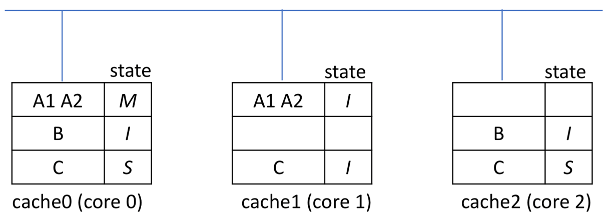
2. Obtain the expression for T_4 including both computation $T_{4(comp)}$ and data sharing $T_{4(data)}$.

   $T_4 = T_{4(comp)} + T_{4(data)}$

   $T_{4(comp)} = (N^2 \div 4) \times t_c$

   $T_{4(data)} = t_s + (N^2 \div 4) \times t_w$

Problem 4 (2 points) Given an SMP system with 3 processors (cores), each with its own cache memory and a shared main memory. Data coherence in the system is maintained using *Write-Invalidate MSI protocol*, with a Snoopy attached to each cache memory. The initial state of the cache memories is shown in the following representation of the system, in which only three cache lines are represented for each cache memory, with variables: `A1, A2, B, C`. Observe that variables `A1` and `A2` reside in the same cache line, while the other two reside in a different cache line each.



Assuming the following sequence of accesses to variables, **we ask you to** fill in the table indicating if each access makes a Hit or Miss in the affected cache, the Bus transactions ($BusRd_k$, $BusRdX_k$, $BusUpgr_k$, $Flush_k$, being k the core that provokes the transaction) that are generated and the state (M, S or I) of only the cache lines that contain the variable in each cache memory after each access. In the observations column indicate who is providing the line when a cache requires it and when main memory is updated.

Memory access	hit/miss	Bus transaction(s)	Cache line state			Observations
			Cache 0	Cache 1	Cache 2	
core 0 reads A1						
core 1 reads A2						
core 2 writes C						
core 2 reads C						
core 0 writes A1						
core 0 reads B						
core 2 reads B						
core 1 writes C						

Solution:

Memory Access	hit/miss	Bus transaction(s)	Cache Line State			Observations
			Cache 0	Cache 1	Cache 2	
core 0 reads A1	hit	–	M			(2)
core 1 reads A2	miss	BusRd/Flush	S	S		(2), (3)
core 2 writes C	hit	BusUpgr	I	I	M	(2)
core 2 reads C	hit	–	I	I	M	(2)
core 0 writes A1	hit	BusUpgr	M	I	I	(2)
core 0 reads B	miss	BusRd	S		I	(1)
core 2 reads B	miss	BusRd	S		S	(1)
core 1 writes C	miss	BusRdX/Flush	I	M	I	(2), (3)

Legend for column observations: (1) Main memory provides the line. (2) Cache provides the line. (3) Main memory is updated.

Problem 5 (1.5 points) Let's assume a multiprocessor system with a hybrid NUMA/UMA architecture composed of 6 identical NUMAnodes, each one with 8 GB (gigabytes, i.e. 2^{30} bytes) of main memory and 3 processors with their own private cache of 128 MB (megabytes, i.e. 2^{20}). Memory and cache lines are 16 bytes wide. Data coherence is maintained using *Write-Invalidate MSI protocol* within each NUMAnode and using a *Write-Invalidate MSU Directory-based* cache coherency protocol among NUMAnodes. **We ask you to** answer the following questions:

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:

   One needs 2 state bits (MSI) for each line of cache memory. Each cache memory has $(128 \times 2^{20}) \div 16$ lines, that is 2^{23} lines; therefore the number of bits per cache is $2^{23} \times 2 = 2^{24}$ bits, that translated into MB is $(2^{24} \div 8) \div 2^{20} = 1$ MB. Since there are 18 processors, each one with its own private cache, the whole system needs 36 MB to keep cache coherence.

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

   Solution:

   One needs 2 state bits (MSU) and 6 presence bits for each line of main memory. For the overall 48 GB, this is $(48 \times 2^{30}) \div 16$ lines, that is 48×2^{26} lines; therefore the number of bits in the directory is $48 \times 2^{26} \times (6 + 2) = 48 \times 2^{29}$ bits, that translated into MB is $((48 \times 2^{29}) \div 8) \div 2^{20} = 48 \times 2^{6} = 3072$ MB.

Problem 6 (1.5 points) Given the following OpenMP code:

```
#define N         (1<<8)   /*  256 */
#define N_THREADS (1<<6)   /*   64 */

int b[N];
int result=0;

#pragma omp parallel shared(result) num_threads(N_THREADS)
{
  int thid = omp_get_thread_num();
  int chunk = N / N_THREADS;

  for (int i = thid * chunk; i < (thid + 1) * chunk; i++) {
    b[i] += foo(i);
    result += b[i];
  }
}
```

Assume a cache-coherent system with 64 processors with write–invalidate coherence protocol and cache lines of 128 bytes. Also assume that the execution of `foo(i)` does not perform any memory accesses, `int` size is 4 bytes and variable `result` is placed at the first position of a memory line (not sharing any memory line with the elements of vector `b`). **We ask you to answer** the following questions:

1. Briefly describe which part of the code provokes a data race (true sharing) situation.

   Solution:

   All threads are **sharing and updating variable `result` with no synchronization** within the for loop. That provokes a possible data race condition reading and writing to result by all threads in the parallel region.

2. Briefly describe which part of the code provokes a false sharing situation (because of the given values for N and $N\_THREADS$).

   Solution:

   A `chunk` of consecutive iterations of the for loop is done in parallel by each thread, reading and updating consecutive elements of b vector ($b[i] += ...$). Although thoses `chunk`'s do not share iterations neither $b[i]$ elements, threads updating two consecutive `chunk`'s of elements of vector b share the same cache line, provoking a false sharing situation.

   Detail:

   In our case, one cache line is 128 bytes and each b element is an `int` of 4 bytes. Therefore, each cache line can contain $128 bytes \times \frac{1 element}{4 bytes} = 32 elements$. For the N and `N_THREAD` given, each thread processes a `chunk` of iterations equal to $N/$`N_THREADS`. That means that each thread processes $256/64 = 16$ consecutive elements of b of type int (half cache line). Therefore, two consecutive threads share the same cache line (half each) when accessing their `chunk` of elements.

3. For the given value of N, compute the maximum number of threads $N\_THREADS$ that can be used (larger than 1 and smaller than or equal to 64) to avoid the occurrence of the false sharing situation. Reason your answer.

   Solution:

   We need that each cache line, containing b elements, be accessed by only one thread. That means that each thread should process 32 elements (number of elements that a cache line can contain). Then, `chunk` is computed as $N/$`N_THREADS`. Therefore, we only need to solve this equation: $chunk = 32 = 256/N\_THREADS$.

   That means that `N_THREADS` should be 8.

Part II

Final Exams

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:

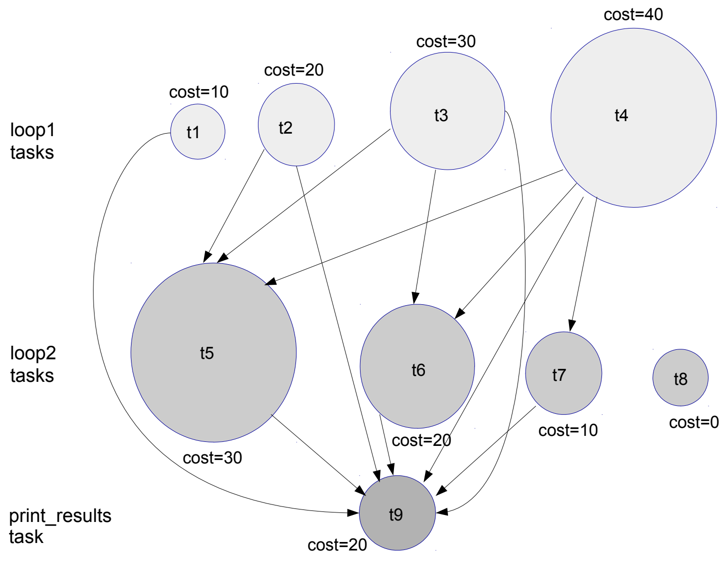
loop 1				
i	k	read	written	task id
0	0	-	m[0][0]	t1
	1	-	-	
	2	-	-	
	3	-	-	
1	0	-	m[1][0]	t2
	1	-	m[1][1]	
	2	-	-	
	3	-	-	
2	0	-	m[2][0]	t3
	1	-	m[2][1]	
	2	-	m[2][2]	
	3	-	-	
3	0	-	m[3][0]	t4
	1	-	m[3][1]	
	2	-	m[3][2]	
	3	-	m[3][3]	

loop 2				
i	k	read	written	task id
0	0	-	-	t5
	1	m[1][0]	m[0][1]	
	2	m[2][0]	m[0][2]	
	3	m[3][0]	m[0][3]	
1	0	-	-	t6
	1	-	-	
	2	m[2][1]	m[1][2]	
	3	m[3][1]	m[1][3]	
2	0	-	-	t7
	1	-	-	
	2	-	-	
	3	m[3][2]	m[2][3]	
3	0	-	-	t8
	1	-	-	
	2	-	-	
	3	-	-	

print results		
read	written	task id
m[0..3][0..3]	-	t9

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:

   $T_1 = (10 + 20 + 30 + 40) + (30 + 20 + 10 + 0) + 20 = 180$

   $T_\infty = 40 + 30 + 20 = 90$

   $Parallelism = 180/90 = 2.0$

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:

   $T_1 = 2 \times 9 + (10 + 20 + 30 + 40) + (30 + 20 + 10 + 0) + 20 = 198$

   $T_\infty = 2 \times 4 + 40 + 30 + 20 = 98$

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:

   $T_4 = t_{computation} + t_{comm}$

   Computation time ($t_{computation}$) on four processors coincides with T_∞ as can be observed in the TDG ($p_{min} = 4$). So $t_{computation} = 90$.

   $t_{comm} = t_{comm\_loop1} + t_{comm\_loop2} + t_{comm\_print\_results}$

$t_{comm\_loop1} = 0$
$t_{comm\_loop2} = 3 \times (t_s + t_w)$
The critical path is along t_5, which needs an element from each of the following tasks: t_2, t_3 and t_4. The other tasks in loop2, can perform the data sharing simultaneously with t_5 in this order:
P0: $t_5 \leftarrow t_2, t_5 \leftarrow t_3, t_5 \leftarrow t_4$
P1: $t_6 \leftarrow t_3, t_6 \leftarrow t_4$
P2: $t_7 \leftarrow t_4$

$t_{comm\_print\_results} = (t_s + 3 \times t_w) + (t_s + 3 \times t_w) + (t_s + 3 \times t_w)$

We obtain finally:
$T_4 = 90 + 3 \times (t_s + t_w) + 3 \times (t_s + 3 \times t_w)$

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:

```
/* A possible solution using omp taskloop grainsize [ or num_tasks ] */
...
#define NUM_ELEMENTS_PER_CACHE_LINE 8
#define BS NUM_ELEMENTS_PER_CACHE_LINE /* Or some multiple of this */

#pragma omp parallel
#pragma omp single
#pragma omp taskloop grainsize(BS) private(j)
for (i=0; i<R; i++) {
  double tmp = 0.0;                  /* Scalar replacement */
  for (j=0; j<C; j++)
    tmp += M[i][j] * X[j];
  Y[i] += tmp;
}
```

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:

```
...
#define NUM_ELEMENTS_PER_CACHE_LINE 8
#define BASE_SIZE NUM_ELEMENTS_PER_CACHE_LINE/2 /* Or some divisor of this */

void dgemv_rec (int iR, int fR, int C, double* M, double* X, double* Y)
{
 int i, j, rows, mR;

 rows = fR - iR;
 if( rows <= BASE_SIZE) {
    for (i = iR; i < fR; i++) {
      double tmp = 0.0;                  /* Scalar replacement */
      for (j=0; j<C; j++)
        tmp += M[i][j] * X[j];
      Y[i] += tmp;
    }
 } else {
    mR = (iR + fR ) / 2;
    #pragma omp task final( rows <= NUM_ELEMENTS_PER_CACHE_LINE) mergeable
    dgemv_rec(iR, mR, C, M, X, Y);
    #pragma omp task final( rows <= NUM_ELEMENTS_PER_CACHE_LINE) mergeable
    dgemv_rec(mR, fR, C, M, X, Y);
 }

}

void mydgemv (int R, int C, double* M, double* X, double* Y)
{

 #pragma omp parallel
 #pragma omp single
 dgemv_rec(0, R, C, M, X, Y);

}
```

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

   Solution:

   This data decomposition requires synchronization in the update to the output vector Y. In order to reduce syncronizations to only one at the very end of the computation of the partial result computed by each thread, we use a temporary scalar variable tmp to accumulate results within the inner loop. [This technique, which is oftentimes applied automatically by a compiler is know as *scalar replacement* and aims at replacing array references in the inner loop with a register reference.]

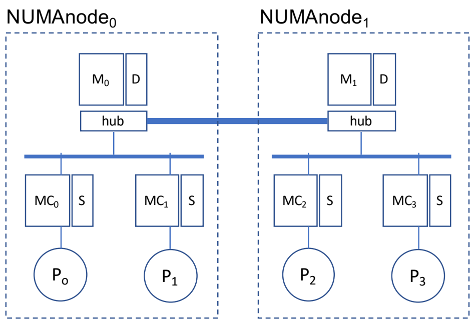
```
#define min(a,b)      ( (a) > (b) ? (b) : (a) )
...
#pragma omp parallel private(i,j)
{
 int id        = omp_get_thread_num();
 int howmany   = omp_get_num_threads();
 int basesize  = C / howmany;
 int reminder  = C % howmany;
 int extra     = id < reminder;
 int extraprev= extra ? id : reminder;
 int lb        = id * basesize + extraprev;   /* Inner loop lower bound */
 int ub        = lb + basesize + extra;       /* Inner loop upper bound */

 for (i = 0; i < R; i++) {
   double tmp = 0.0;                          /* Scalar replacement */
   for (j = lb; j < ub ; j++)
     tmp += M[i][j] * X[j];
   #pragma omp atomic
   Y[i] += tmp;
 }
}
...
```

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:

```
#define min(a,b)      ( (a) > (b) ? (b) : (a) )
#define NUM_ELEMENTS_PER_CACHE_LINE 8
...

#pragma omp parallel private(i,j)
{
 int id         = omp_get_thread_num();
 int howmany    = omp_get_num_threads();
 int block_size = NUM_ELEMENTS_PER_CACHE_LINE;
 int lb         = id*block_size;         /* Outer loop lower bound */
 int step       = howmany*block_size;    /* Outer loop step */

 /* Loop jumping blocks cyclically */
 for (int ii = lb; ii < R; ii+=step)
   /* Loop traversing each block */
   for (i = ii; i < min( R, ii+block_size ); i++) {
     double tmp = 0.0;                   /* Scalar replacement */
     for (j = 0; j < C; j++)
       tmp += M[i][j] * X[j];
     Y[i] += tmp;
   }
}
...
```

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.



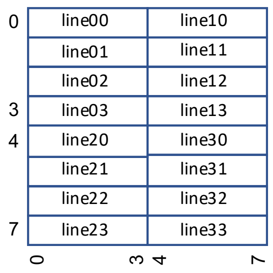
Legend:
NUMAnode$_i$: NUMA node i with two processors
M$_i$: main memory for NUMAnode$_i$
D: directory associated to M
hub: interconnect between NUMA nodes
P$_j$: processor j inside NUMA node
MC$_j$: cache memory local to processor P$_j$
S: snoopy associated to MC

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
```

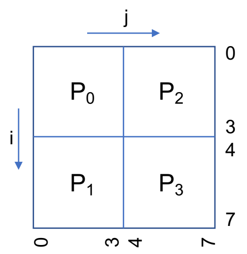
Matrix `a` is stored by rows in memory, with each memory line storing 4 consecutive integer values, Therefore matrix `a` occupies 16 memory lines, as shown (and labeled) in the following figure:



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: Lines are allocated in main memory of NUMAnode0 and they are cached in the caches of cores 0 and 1. This implies that in the initialisation loop only the `i` loop has been parallelised, assigning iterations $0..(n/2-1)$ to P_0 and the rest to P_1.

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	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		M			01 M					
	line1x for a	BusRd(hub), Flush(0) BusUpgr(hub)	(S) I		Dreply(0->1) Ack(0->1)	(11 S) 10 M	BusRd(2) BusUpgr(2)	(S) M		RdReq(1->0) UpgrReq(1->0)	
	line2x for a			M		01 M					
	line3x for a	BusRd(hub), Flush(1) BusUpgr(hub)		(S) I	Dreply(0->1) Ack(0->1)	(11 S) 10 M	BusRd(3) BusUpgr(3)		(S) M	RdReq(1->0) UpgrReq(1->0)	
compute 2 loop	line0x for a		M			01 M					
	line1x for a	BusRd (1)		S	Fetch(0->1)	11 S	BusRd(hub), Flush(2)	S		Dreply(1->0)	
	line2x for a	BusRd(hub), Flush(1)		S	Dreply(0->1)	11 S	BusRd(2)	S		RdReq(1->0)	
	line3x for a			I		10 M			M		

Student name: ....

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

loop 1

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

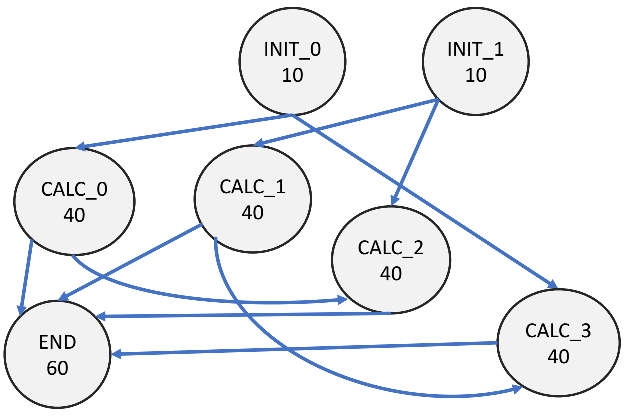
June 18th, 2019

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");
```

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:
   $T_1 = 2 \times 10 + 4 \times 40 + 60 = 240$
   critical path = INIT_0, CALC_0, CALC_2, END
   $T_\infty = 10 + 2 \times 40 + 60 = 150$
   $Par = T_1 \div T_\infty = 240/150 = 1.6$
   $P_{min} = 2$

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:
   Task allocation to P_{min} processors:
   $P_0 = INIT\_0, CALC\_0, CALC\_2, END$
   $P_1 = INIT\_1, CALC\_1, CALC\_3$
   $T_{calc} = 10 + 40 + 40 + 60 = 150$
   The critical path is in P_0 and the remote data access from P_0:
   task CALC_2 access remotely row i = 2 allocated in P_1
   task END access locally rows i = 0 and i = 2 and has to ask to P_1 row i = 1 and latest version of row i = 3 , so the expression for the communication time is:
   $T_{comm} = (t_s + 4.t_w) + (t_s + 8.t_w)$
   Finally the expression for the execution time on two processors:
   $T_2 = 150 + 2.t_s + 12.t_w$

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:

```
#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]++;
     omp_unset_lock(&locks_C[value]);
   }
 }

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

}
```

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:

```
#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 + C_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]++;
      }
    }
  }

}
```

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:

```
#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]++;
            omp_unset_lock(&locks_C[value]);
     } else {
        if (!omp_in_final()) {
            #pragma omp task final(depth>=MAX_DEPTH) // mergeable can not be used
            rec_distribute( S, n2, C, D, depth+1);
            #pragma omp task final(depth>=MAX_DEPTH) // mergeable can not be used
            rec_distribute( &S[n2], n-n2, C, D, depth+1);
        } else {
            rec_distribute( S, n2, C, D, depth+1);
            rec_distribute( &S[n2], n-n2, C, D, depth+1);
        }
     }
    }

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

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

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.

Solution: In each NUMA node the memory is able to store $8GB \div 32$ memory lines. This is $2^3 \times 2^{30} \div 2^5 = 2^{28}$ lines. For each line in memory the directory needs to store 2 bits to keep the state of the line (MSU) and X presence (sharers) bits (one per NUMA node). Since the total number of bits per NUMA node devoted to the directory is 2^{32} bits, we can formulate $2^{28} lines/node \times (2 + X) bits/line = 2^{32}$ bits; so $X = 14$.

Similarly, each cache memory inside a NUMA node is able to store $4MB \div 32$ memory lines. This is $2^2 \times 2^{20} \div 2^5 = 2^{17}$ lines. For each line in the cache the snoopy needs to store 2 bits to keep the state of the line (MSI). Since the total number of bits per NUMA node devoted to the snoopy coherence is 2^{20} bits, we can formulate $Y sockets/node \times 2^{17} lines/socket \times 2 bits/line = 2^{20}$ bits; so $Y = 4$.

Finally, there is not sufficient information to conclude the number of cores per socket, since they are all sharing the access to the same cache.

Assume that two different cores in the previous system, one in $NUMAnode_i$ and another in $NUMAnode_j$ are trying to atomically update variable `sum` with the following code:

```
#pragma omp atomic
sum += local_result;
```

Also assume that the home NUMA node for variable `sum` is $NUMAnode_k$ and that initially there are no copies of the memory line containing variable `sum` in any cache memory.

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: We simply need to chain the two low–level synchronization constructs and check whenever the store is successfully completed, as follow:

```
        int ret;
        do {
               int value = load_linked (&sum);
               value += local_value;
               ret = store_conditional (&sum, value);
        } while (ret == 0);
```

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.

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	0	0	1
2	**store_conditional**	M	1-BusUpgr	2-UpgrReq(i->k)					3-Ack(k->i)		M	0	0	1
3		S	4-BusRd, 5-Flush	6-Dreply(i->k)	**load_linked**	S	1-BusRd	2-RdReq(j->k)	3-Fetch(k->i), 7-Dreply(k->j)		S	0	1	1
4		I	4-BusUpgr	5-Ack(i->k)	**store_conditional**	M	1-BusUpgr	2-UpgrReq(j->k)	3-Invalidate(k->i) 6-Ack(k->j)		M	0	1	0

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**													
2	**store_conditional**													
3					**load_linked**									
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 16th, 2020

Problem 1 (2 points)

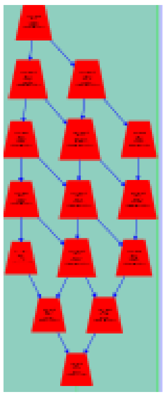
The following code excerpt instrumented with *Tareador* belongs to a program we want to parallelize. It simulates the heat diffusion equation using the *Gauss-Seidel* method. We have developed a blocked implementation that creates $N \times M$ blocks:

```
#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:



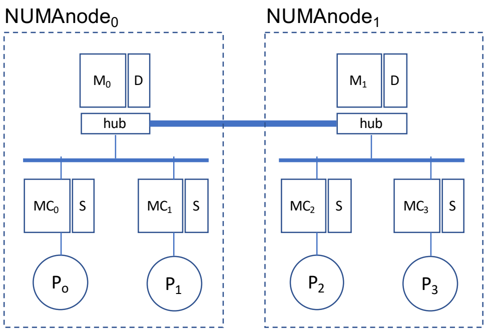
2. (0.5 points) In view of the TDG above, give a general expression for P_{min} as a function of N and M. Hint: Consider how the TDG would look like if N was larger than M.

   Solution: To schedule the tasks in the critical path without delays we need as many processors as the smallest of N and M. Thus, $P_{min} = min(N, M)$.

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: $T_{P_{min}} = (N + M - 1) \times T_c + (N + M - 2) \times T_{sync}$

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:

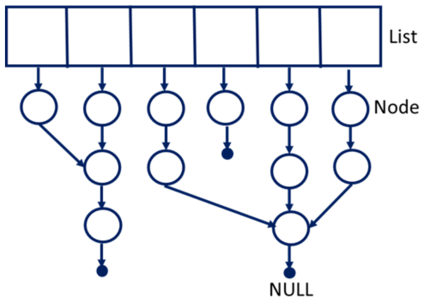
Time	NUMA Node 1					NUMA Node 0							
	Instruction	MC3 line state	MC2 line state	Bus transactions	NUMA transactions	Instruction	MC1 line state	MC0 line state	Bus transactions	NUMA transactions	Directory state	Sharers list	
		total	total				total	total			total	1	0
0		-	-				-	-			U	0	0
1	**load_linked (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-UpgrReq(1→0)					3-Ack(0→1)	M	1	0
3		S		3-BusRd(hub) 4-Flush(3)	5-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

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:

```
float process_vector (List *v[N]) {
     float res = 0.0;
     #pragma omp parallel
     #pragma omp single
     #pragma omp taskloop grainsize(1) reduction (+:res)
     for (int i = 0; i < N; i++)
         res = res + compute_sequential (v[i]);
     return res;
}
```

   In order to be correct, the update to footprint field in compute_sequential has to be done atomically:

```
float compute_sequential (struct Node *p) {
     float sum = 0.0;
     int end = 0;
     while (p != NULL && end == 0) {
          #pragma omp atomic
          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;
}
```

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:

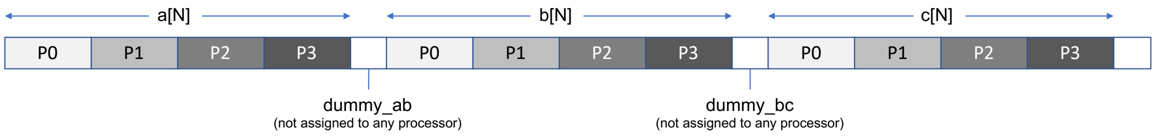
```
float process_vector_rec (List *v[N], int n) {
     float res = 0.0;
     if (n <= MIN_SIZE)
         for (int i = 0; i < n; i++)
             res = res + compute_sequential (v[i]);
     else {
         int n2 = n / 2;
         #pragma omp task shared(res1)
         float res1 = process_vector_rec (v, n2);
         #pragma omp task shared(res2)
         float res2 = process_vector_rec (v+n2, n-n2);
         #pragma omp taskwait
         res = res1 + res2;
     }
     return res;
}




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

Again, in order to be correct, the update to footprint field in compute_sequential has to be done atomically as shown in the previous part.

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:


a[N]
b[N]
c[N]
P0
P1
P2
P3
dummy_ab
(not assigned to any processor)
dummy_bc
(not assigned to any processor)

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: Each one of the vectors has $N = 1000$ elements and occupies $N \times INT\_SIZE$ bytes, that is 4000 bytes. This number of bytes fits in $4000 \div CACHE\_LINE = 62,5$ cache lines. To ensure that the next vector starts in a new cache line, we should add a padding of half a line, that is 32 bytes, or equivalently 8 integer elements in each padding vector.

Since the number of processors always divides N perfectly, we don't need to add code to balance the number of elements assigned to each thread.

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

struct SoA {
     int a[N];
     int dummy_ab[8]; // (CACHE_LINE - (N*INT_SIZE)%CACHE_LINE) / INT_SIZE
     int b[N];
     int dummy_bc[8];
     int c[N];
};
struct SoA dataElements;
...
#pragma omp parallel
{
     int myid = omp_get_thread_num();
     int howmany = omp_get_num_threads();
     int BS = N / howmany;
     int lower = myid * BS;
     int upper = lower + BS; // no need to add code to balance the number of iterations
     for (int i = lower, i < upper; i++)
          dataElements.c[i] = foo(i, dataElements.a[i], dataElements.b[i]);
}
...
```

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: Each element of vector `dataElements` occupies 12 bytes (3 integer elements). To avoid false sharing we should assign to each processor a number of consecutive elements that fit in a number of complete cache lines. In this case this is $mcm(12, 64) = 192$ (minimum common multiple), which corresponds to 16 elements. Other computations that also give as result 16, as for example the number of integers in a cache line ($64/4 = 16$), have not been considered correct answers.

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

   Solution:

```
#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 myid = omp_get_thread_num();
     int howmany = omp_get_num_threads();
     int BS = 16;  // result from previous question
     int lower = myid * BS;
     int upper = N;
     int step = howmany * BS;
     for (int ii = lower; ii < upper; ii += step)
          for (int i = ii; i < min(N, ii+BS); i++)
               dataElements[i].c = foo(i, dataElements[i].a, dataElements[i].b);
}
...
```

Solution sheet for Problem 2

SURNAME:

NAME:

	NUMA Node 1					NUMA Node 0							
Time	Instruction	MC3 line state	MC2 line state	Bus transactions	NUMA transactions	Instruction	MC1 line state	MC0 line state	Bus transactions	NUMA transactions	Directory state	Sharers list	
		total	total				total	total			total	1	0
0		-	-				-	-			U	0	0
1	**load_linked (in core 3)**												
2	**store_conditional (in core 3)**												
3						**load_linked (in core 1)**							
4						**store_conditional (in core 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 18th, 2021

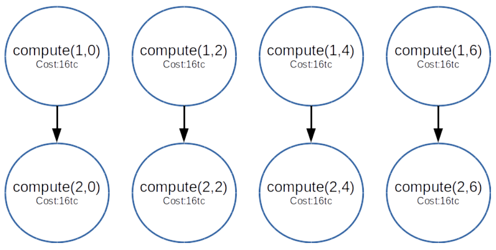
Problem 1 (5 points) Given the following nested loops in a C program instrumented with *Tareador*:

```
#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: The i loop only performs two iterations, generating 4 tasks (jj loop) in each of them. There are not dependences among `compute(i,jj)` task instances with the same value of i but there are true dependences due to the access to $A[i-1][j]$, between `compute(i,jj) - compute(i+1,jj)`, for jj in $\{0, 2, 4, 6\}$.



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

   Solution: The sum of the cost of all the tasks, executed in one only processor, determines T_1. Each task executes two iterations of the loop body (j loop), therefore:

$$T_1 = 8 \times 2 \times 8 \times t_c = 128 \times t_c$$

   T_∞ is defined by the minimum cost needed to execute all the tasks of the TDG when using infinite resources. In our case, the critical path is composed by any of the pair of tasks `compute(1,jj) - compute(2,jj)`, for jj in $\{0, 2, 4, 6\}$.

$$T_\infty = 2 \times 2 \times 8 \times t_c = 32 \times t_c$$

   For the given TDG, and the T_∞ obtained, we need 4 processors to execute in parallel all `compute(1,jj)` tasks, or all tasks `compute(2,jj)` in such a way we can achieve the T_∞ time.

$$P_{min} = 4$$

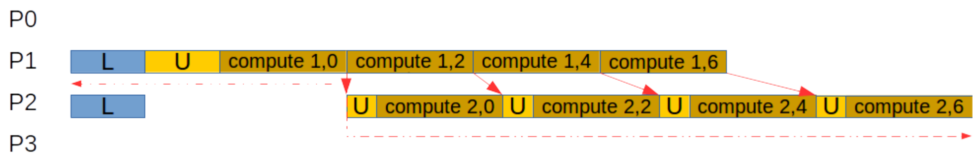
3. Write the expression that determines the execution time T_4 for the program, clearly indicating the contribution of the computation time T_4(comp) and the data sharing overhead T_4(mov), for the following assignment of tasks to threads and processors: tasks `compute(1,jj)` are assigned to `thread 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: In order to compute the model, we have done the initial communication at the beginning of the execution, as seen at class. Remember, initial communication is not due to true dependences but necessary due to the mapping of the data in a remote processor. However, there are other possibilities that may be more efficient to overlap computation and initial communication; all of them have been considered correct. On the other hand, due to the dependences, `thread 2` has to wait for task `compute(1,jj)` (executed by `thread 1`) to execute task `compute(2,jj)`. Once task `compute(1,jj)` is executed by `thread 1`, `thread 2` will read the upper boundary data of `BS` elements to execute task `compute(2,jj)`.

You can see the intial comunication, the executed tasks, the synchronization between tasks with solid red arrow, and the communication during the computation in the following time diagram:



P1 and P2's L : Initial communication: MAX_VALUE-element lower boundary
P1's U: Initial communication - MAX_VALUE-element upper boundary
P2's U: Communication due to dependency - BS-element upper boundary

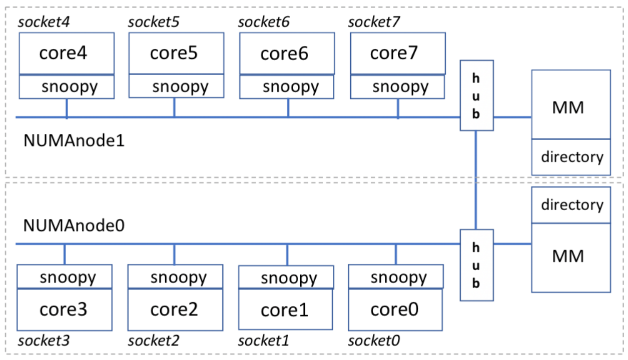
Note that we have added a dashed red line to indicate one possible way to compute the critical path. The critical path cost includes T_4(comp) and T_4(mov):

$T_4(\text{comp}) = 5 \times 2 \times 8 \times t_c$

$T_4(\text{mov}) = 2 \times (t_s + MAX\_VALUE \times t_w) + 4 \times (t_s + BS \times t_w)$

$T_4 = T_4(\text{comp}) + T_4(\text{mov})$

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: We need 2 bits for the cache line state. The possible values are: M (Modified), S (Shared) and I (Invalid). The total number of lines in a cache is: cache size / cache line size $= 4 * 2^{20}/2^6 = 2^{16}$, so the total number of bits per cache $= 2 * 2^{16}$.

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

   Solution: We need 4 bits per directory entry (that corresponds to each memory line: 2 bits for the line state with possible values: M (Modified), S (Shared), U (Uncached) and 2 bits for the data sharing (presence) bits (one bit per NUMA-node). Total number of lines in main memory per NUMA-node is : $8 * 2^{30}/2^6 = 2^{27}$, so the total number of bits per NUMA-node $= 4 * 2^{27}$ bits.

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]`

   Solution:

   (a) $core_0$ reads the contents of `v[0]`, which is not available in the associated cache (miss); $core_0$ generates a $PrRd$ event which activates the Snoopy protocol, placing a $BusRd$ transaction on the bus; since the local node and the home node are the same, and the initial status for the memory line in the directory is U (uncached), no coherence transactions are generated between NUMA nodes; the presence bits in the directory entry are set to 01 and state transitions to S (shared); a copy of the memory line is loaded into the local cache; and the cache line status in $core_0$ is updated from I (invalid) to S (shared) too.

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) $core_1$ reads the contents of `v[8]`, which is not available in the associated cache (miss); observe that `v[8]` resides in the same cache line as `v[0]`; $core_1$ generates a *PrRd* event which activates the Snoopy protocol, placing a *BusRd* transaction on the bus; again, the local node and the home node are the same, so, since no copies in other NUMA nodes exist, no coherence transactions between NUMA nodes are performed; presence bits and status in the directory entry do not change and a copy of the memory line is sent to the local cache of $core_1$, updating its status from I (invalid) to S (shared).

(c) $core_0$ writes the contents of `v[0]`, which is already loaded in the associated cache is S status (hit); $core_0$ generates a *PrWr* event which activates the Snoopy protocol, placing a *BusUpgr* transaction on the bus. Aa a result, the snoopy of $core_1$ invalidates its copy, setting its state to I; again, the local node and the home node are the same, so, since no copies in other NUMA nodes exist, no coherence transactions between NUMA nodes are performed; presence bits in the directory entry do not change but the status transitions to M (modified); similarly, the statues of the line in cache associated to $core_0$ also transitions to M.

(d) $core_4$ reads the contents of `v[16]`, which is not available in the associated cache (miss); observe that `v[16]` resides in a different memory line than `v[0]` and `v[8]`; $core_4$ generates a *PrRd* event which activates the Snoopy protocol, placing a *BusRd* transaction on the bus; now the local hub asks for a copy to the home node by sending a *RdReq* message; the hub in the home NUMA node updates the information associated to the directory entry of the memory line, updating the presence bits to 10 to indicate that a copy exists in NUMA nodes 1 and status set to S (shared); then the hub sends a copy of the line to the local hub through a *Dreply* message.; the state of the line in the cache associated to $core_4$ transitions from I (invalid) to S (shared).

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: We can find mainly two performance problems:

(a) As the whole vector `v` is entirely allocated in MM_0, all the accesses to elements in the vector will be managed by NUMA node 0, generating a bottleneck and consequently degrading performance.

(b) The code above shows a well-balanced distribution of the vector `v` among the NUMA nodes of the systems. This implies that each thread is in charge of updating 64 elements / 8 threads = 8 consecutive elements. However, as a the line cache size is 64 bytes and that `sizeof (int)` = 4, then each cache line holds 64 bytes / 4 bytes = 16 elements, which means that a cache line will be shared by two threads, generating the so called False sharing performance problem.

PAR – Final Exam: Part 2 – Course 2020/21-Q1
January 18th, 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);
}
```

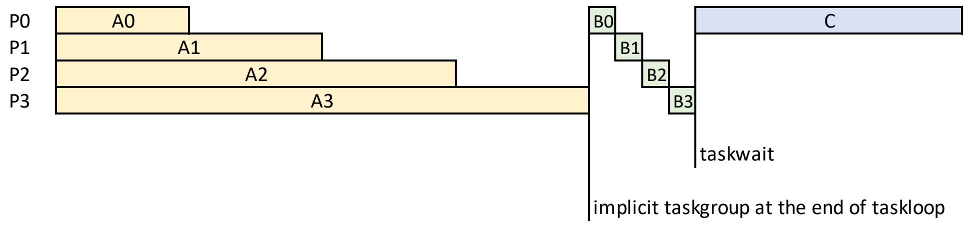
version 2:

```
// 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.

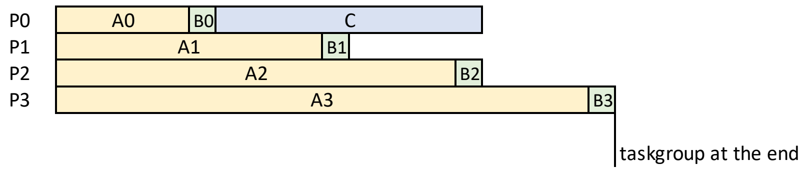
1. (2 points) Draw a temporal diagram showing a possible execution on 4 processors for each code version above, obtaining the expression for its execution time T_4.

   In *version 1* dependences between tasks *Ai* and *Bi* are satisfied with the implicit `taskgroup` at the end of the `taskloop` region; dependences among tasks *bi* due to the sharing of global variable `x` are satisfied with task dependences using the `depend` clause; finally, the dependence between tasks *Bi* and *C* is satisfied with an explicit `taskwait`. A possible diagram for the parallel execution on 4 processors for this code, assuming that tasks are executed as soon as dependences are satisfied, could be this:



   which results in an execution time of $T_4 = 20 + 4 + 10 = 34$ time units.

   In *version 2* dependences between tasks *Ai* and *Bi* are satisfied with task dependences using the `depend` clause on each element of vector `l`; dependences among tasks *Bi* and `C` due to the sharing of global variable `x` are satisfied with task reductions in the scope of a `taskgroup` region, using `in_reduction` clause to specify the tasks that contribute to the reduction operation specified in the `task_reduction` clause of the `taskgroup`. A possible diagram for the parallel execution on 4 processors for this code, assuming that tasks are executed as soon as dependences are satisfied, could be this:



which results in an execution time of $T_4 = 20 + 1 = 21$ time units; observe that the execution of task C is totally hidden by the execution of the longest task B_3.

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.

   The sequence of instructions that should be executed by each task to do the safe accumulation of its local variable `xlocal` into shared variable `x` could be:

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

   Atomicity in the accumulation to variable `x` is guaranteed by the paired execution of `load_linked` and `store_conditional`, which is repeated in case it fails (i.e. when `store_conditional` returns 0), reading again the current value for `x`.

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: An alternative synchronization with locks has also been considered correct. We assume that the initial address of vector `HashTable` is aligned to a memory line.

```
#pragma omp parallel private(p,value,bin)
#pragma omp single
{
 int NT = omp_get_num_threads();
 int GS = HT_SIZE / NT;
 int CACHE_SIZE_ELEMS = CACHE_LINE_BYTES / sizeof(int *);
 int resize = (GS%CACHE_SIZE_ELEMS) ? CACHE_SIZE_ELEMS - (GS%CACHE_SIZE_ELEMS) : 0;
 GS = GS + resize;
 #pragma omp taskloop grainsize(GS)
 for (i=0; i<HT_SIZE; i++) {
    p = HashTable[i];
    while ( p != NULL ) {
      value = p->value;
      bin = getbin(value,minval,maxval,NBINS);
      #pragma omp atomic
      Histogram[bin]++;
      p = p->next;
    }
 }
}
```

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

   Solution: The solution requires an *if* used to check if the thread has to update a range of positions in the histogram. No synchronizations are required. We assume that the initial address of vector `Histogram` is aligned to a memory line. Granularity must be properly defined to avoid false sharing in the access to `Histogram`.

```
#pragma omp parallel private(bin,i,p,value)
{
 int thid = omp_get_thread_num();
 int NT = omp_get_num_threads();
 int CACHE_SIZE_ELEMS = CACHE_LINE_BYTES / sizeof(int);
 for (i=0; i<HT_SIZE; i++) {
    p = HashTable[i];
    while ( p != NULL ) {
      value = p->value;
      bin = getbin(value,minval,maxval,NBINS);
      if ( (bin/CACHE_SIZE_ELEMS)%NT == thid )
        Histogram[bin]++;
      p = p->next;
    }
 }
}
```

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

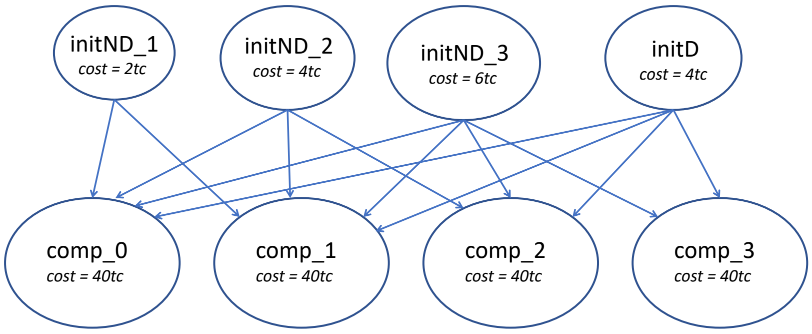
Problem 1 (2 points) Given the following nested loops in a C program instrumented with *Tareador*:

```
 #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);
}

// 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);
}
```

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:

   The sum up of the cost of all the tasks determines $T_1 = 176 \times tc$. The critical path is composed of nodes: `initND_3` and `comp_X`, where X is any number between 0 and 3. Its cost is $T_\infty = 46 \times tc$. The minimum number of processors to achieve T_∞ execution time is $P_{min} = 4$.

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:

   Since the number of requested preocessors is $P = P_{min} = 4$, then $T_4 = T_\infty = 46 \times tc$, and therefore $S_4 = 176\ /\ 46 = 3.8$ coincides with the *Parallelism* metric. The assignment of tasks to 4 processors is straightforward:

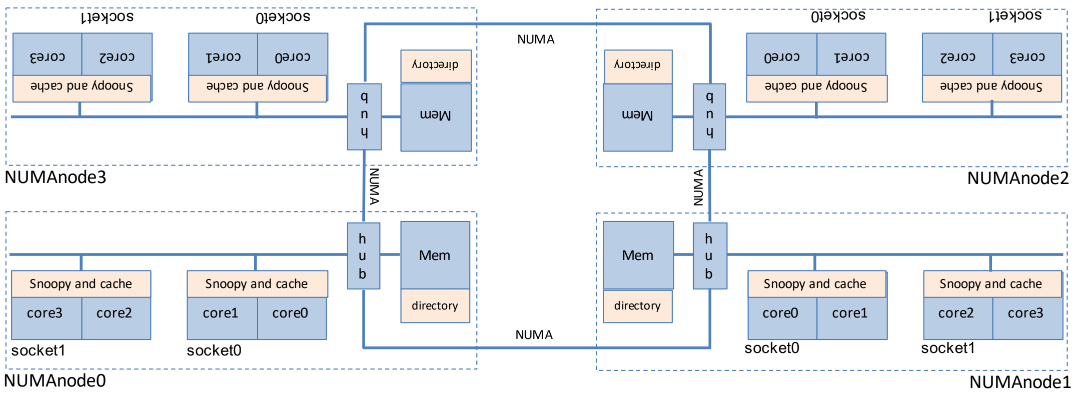
   $P_0 =$ `initND_1, comp_0`

   $P_1 =$ `initND_2, comp_1`

   $P_2 =$ `initND_3, comp_2`

   $P_3 =$ `initD, comp_3`

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: True, True, False, False, True

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: True, False, False, False, True

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: False, False, True, True, True

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:

   The key points are :

   - Usage of taskloop construct with a granularity bigger than 1, for instance, 2 (`grainsize(2)`) or evenly distributing the iterations among threads (`num_tasks(omp_get_num_threads())`).
   - Usage of reduction clause to avoid synchronizations to update the max and min durations.
   - Add constructs parallel and single in the main program.

```
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];

     /* Assuming the number of threads is smaller than n */
     #pragma omp taskloop num_tasks(omp_get_num_threads()) \
                          reduction(max:max_duration)      \
                          reduction(min:min_duration)
     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;
}
```

```
int main() {
    min_max_t min_max_V1, min_max_V2;
    float v1[N], v2[N];

    #pragma omp parallel
    #pragma omp single
    min_max_V1 = find_min_max_it(v1, N);

    min_max_V2 = find_min_max_rec(v2, N);
    }
}
```

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:

   The key points are :

   - Implement a recursive tree task decomposition
   - Add a new parameter to count the depth level
   - Usage of final and mergeable clauses to implement the cut-off based on the recursivity tree level. Note that it can be implemented using if statements controlling if the maximum depth is reached or not.
   - Force `min_max1` and `min_max2` to be shared, and a taskwait to wait for the two created tasks to be finished.
   - Add constructs parallel and single in the main program.

```
typedef struct {
  float max; float min;
} min_max_t;

min_max_t find_min_max_rec(float *v, int n, int depth) {
    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

        #pragma omp task shared(min_max1) final(depth>=MAX_DEPTH) mergeable
        min_max1 = find_min_max_rec(v, n2, depth+1);

        #pragma omp task shared(min_max2) final(depth>=MAX_DEPTH) mergeable
        min_max2 = find_min_max_rec(v+n2, n2, depth+1);

        #pragma omp taskwait

        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);

        #pragma omp parallel
        #pragma omp single
        min_max_V2 = find_min_max_rec(v2, N, 0);
    }
```

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:

   ```
   int thread_to_be_assigned(int row, int num_rows, int num_procs) {
       int num_elems = num_rows / num_procs;
       return (row / num_elems);
   }
   ```

   (b) *CYCLIC* data decomposition;
   Solution:

   ```
   int thread_to_be_assigned(int row, int num_rows, int num_procs) {
       return (row % num_procs);
   }
   ```

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

   ```
   #define BS ...
   int thread_to_be_assigned(int row, int num_rows, int num_procs) {
       return ((row / BS) % num_elems);
   }
   ```

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:

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

   Of course, solutions implementing a *test-test&set* solution have also been considered valid.

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:

   ```
   void wait4master (int num) {
       do {
           while (load_linked(&port[num].ready) == 0);
       } while (store_conditional(&port[num].ready, 0) == 0);
   }
   ```

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, respectively, and that cache and memory lines are 64 bytes long,

   Solution: False sharing may occur since several consecutive elements of vector `port` can reside in the same cache line and each of them read/written by a different processor. The solution would be to make sure each element occupies a complete cache line. Since the structure `Port` includes 2 integer and 1 double, in total 16 bytes, we can add padding for a total of $64-16$ bytes, that is 48 bytes (which are occupied for example by 12 integer elements):

   ```
   typedef struct {
       int ready;          // 4 bytes
       int row;            // 4 bytes
       double value;       // 8 bytes
       int padding[12];    // 48 bytes
   } Port;                 // 64 bytes

   Port port[P];           // 64 bytes per element
   ```

Another option would be to convert `port` to a matrix, so that each row occupies a complete cache line. To achieve that, since the structure occupies 16 bytes, we need 4 elements in each row:

```
typedef struct {
    int ready;           // 4 bytes
    int row;             // 4 bytes
    double value;        // 8 bytes
} Port;                  // 16 bytes

Port port[P][64/16];     // 64 bytes per row
```

and then modify all accesses in the code accordingly to only access to column 0, for example: `port[destination][0]` `row`.