PAR
Selection of Exams (with Solutions)

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

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

Course 2022-23 (Spring semester)

Contents

I In-term Exams **2**

II Final Exams **30**

Part I

In-term Exams

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

November 6^{th}, 2020

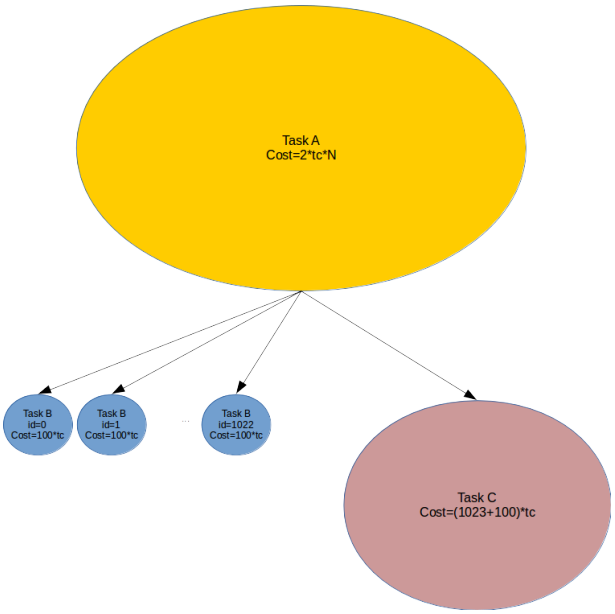
Problem 1 (2 points) Consider the following code excerpt (including *Tareador* task definitions):

```
#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}*:*

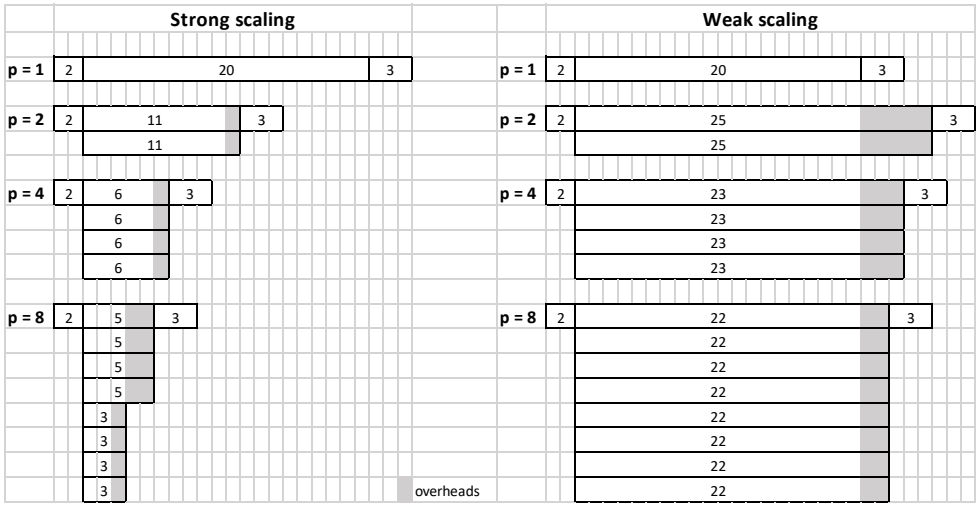
   To achieve T_∞ we need 1 processor to execute the critical path (task A and task C) and one or more processors to execute all instances of tasks B in parallel with task C.

   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 $S_2 = (T_{seq} + 2 \times T_{par}) \div T_2 = (5+2\times20) \div (2+25+3) = 1.5$; $S_4 = (T_{seq} + 4 \times T_{par}) \div T_4 = (5+4\times20) \div (2+23+3) = 3.03$; 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 interconnection 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, $T_P = T_P^{calculations} + T_P^{remote\_accesses} = (\frac{N}{BS} + P - 1) \times (\frac{N}{P} \times BS) \times t_c + (\frac{N}{BS} + P - 2) \times 2 \times (t_s + BS \times t_w)$

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

April 21^{st}, 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:

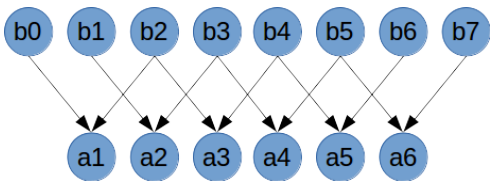
   The first loop iterates from 0 to $N-1$; one task is created per iteration. Therefore, N tasks of type `task-b` are created.

   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.

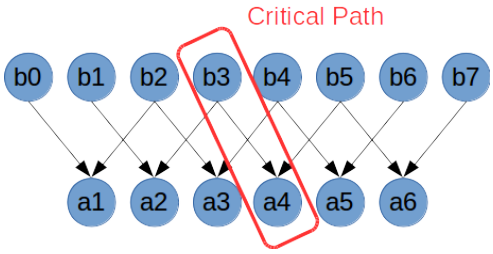


3. Compute T_1, T_∞ and P_{min}.

   Solution:

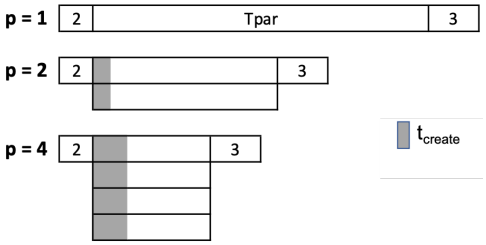
   Given the previous TDG:

   - $T_1 = N \times t_c + (N-2) \times t_c = (2N-2) \times t_c$.
   - $T_\infty = 2t_c$. 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 $S_2 = 1.65$.

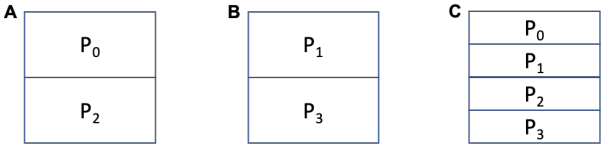
   Solution: $S_p = T_1 \div ((1 - \varphi) \times T_1 + (\varphi \times (T_1/p) + (\alpha \times p)))$. From this expression applied to $p = 2$ one can easily get $\alpha = 0.075$.

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:



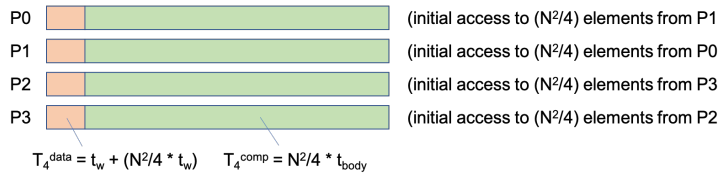
```
#define N 64
#define N_THREADS 4
int A[N][N], B[N][N], C[N][N];
...
#pragma omp parallel num_threads(N_THREADS)
{
    int thid = omp_get_thread_num();
    int chunk = N / N_THREADS;

    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.



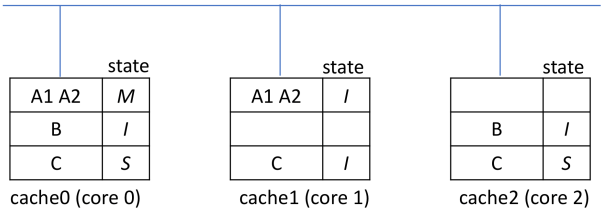
2. Obtain the expression for T_4 including both computation $T_{4(comp)}$ and data sharing $T_{4(data)}$.

   $T_4 = T_4^{comp} + T_4^{data}$

   $T_4^{comp} = (N^2 \div 4) \times t_c$

   $T_4^{data} = t_s + (N^2 \div 4) \times t_w$

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.

PAR – In-Term Exam – Course 2021/22-Q1

November 8^{th}, 2021

Problem 1 (2 points)

We have a sequential code that we want to parallelize. Our code has four disjoint parts executed one after the other, namely *Start*, *Init*, *Compute* and *End*, which take 1, 10, 100 and 2 time units, respectively.

1. If we only parallelize the *Compute* phase, which would be the value for the parallel fraction ϕ? Assuming that *Compute* can be perfectly parallelized, and there are no overheads resulting from its parallelization, which would be the value for the ideal speed–up $S_{p\to\infty}$?

   Solution:

   $\phi = \frac{100}{113} = 0.885$; $1 - \phi = 0.115$

   Then, according to Amdahl's law when the number of processors $P \to \infty$:

   $S_{p\to\infty} = \frac{1}{1-\phi} = \frac{1}{0.115} = 8.69$

2. Our system, however, has a parallelization overhead proportional to the number of processors being used. Which would be the value for the speed–up when using 10 processors (S_{10}) if the parallelization of *Compute* can be perfectly parallelized but incurs in an overhead of 0.001 time units per processor being used?

   Solution:

   Considering the overheads and Amdahl's law:

   $S_P = \frac{1}{1-\phi+\phi/P+0.001*P/T_1}$

   Thus,

   $S_{10} = \frac{1}{1-\phi+\phi/10+0.001*10/T_1} = 4.91$

   or, alternatively, just applying the definition of $S_p = T_1/T_p$:

   $S_{10} = \frac{T_1}{T_{10}} = \frac{113}{13+100/10+0.001*10} = \frac{113}{23.01} = 4.91$

3. Next, we parallelize the *Init* phase on two processors. Regrettably, in this *Init* phase the parallelization cannot be scaled beyond two processors. Assuming again that there are no overheads due to the parallelization, which would be the new value for S_{10} in this case?

   Solution:

   Since a part of the code cannot be perfectly parallelized we cannot use Amdahl's law here. Thus, our only choice in this case is using the definition $S_P = \frac{T_1}{T_P}$.

   $T_{10} = 1 + 5 + 10 + 2 = 18$

   Therefore: $S_{10} = \frac{T_1}{T_{10}} = \frac{113}{18} = 6.277$

Problem 2 (2.5 points)

Assume a program composed of two parallel regions: *Region A* and *Region B*; both regions scale ideally when executed with P processors. There is no code outside these two regions in the program. The first region, *Region A*, is basically a double nested loop reading matrix `Matrix_a` and writing into matrix `Matrix_b`:

```
for (int row = 0; row < N; row++)
  for (int col = 0; col < N; col++)
        Matrix_b[row][col] = foo(Matrix_a[row][col]);
```

For this region the programmer has implemented a task decomposition in which each task computes N/P consecutive iterations of the `row` loop, with tasks computing blocks of rows assigned to processors in ascending order. Once the execution of *Region A* is finished, the program proceeds with *Region B*, which is also a double nested loop reading matrix `Matrix_b` and writing into matrix `Matrix_c`:

```
for (int col = 0; col < N; col++)
  for (int row = 0; row < N; row++)
        Matrix_c[row][col] = goo(Matrix_b[row][col]);
```

For this region the programmer has implemented a task decomposition in which each task computes N/P consecutive iterations of the `col` loop, again with tasks computing blocks of columns assigned to processors in ascending order. After the execution of *Region B* the program terminates.

The three matrices have N rows and N columns, but are distributed in three different ways: `Matrix_a` is totally stored in the memory of processor 0; `Matrix_b` is distributed by rows among all P processors, so that each processor stores N/P consecutive rows in its memory (blocks of rows mapped to processors in ascending order); and `Matrix_c` is distributed by columns among all P processors, so that each processor stores N/P consecutive columns in its memory (blocks of columns mapped to processors in ascending order).

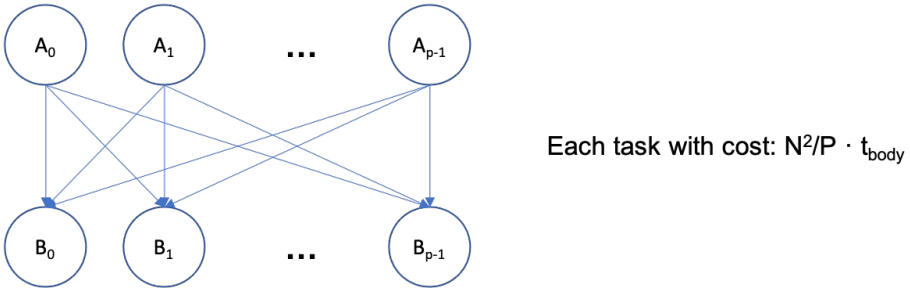
You can assume the data sharing model explained in class in which the overhead to perform a remote access is $t_s + t_w \times m$, being t_s the start-up time, t_w the time to transfer one element and m the number of elements to be transferred. At any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor. You can also assume that the execution of body of each innermost loop takes t_{body}.

We ask you to:

1. Draw the *Task Dependence Graph* (TDG) for the program described above, indicating the cost of each task.

   Solution:

   The TDG includes P nodes, each with a cost of $\frac{N^2}{P} \times t_{body}$, for *Region A* and P nodes, each also with a cost of $\frac{N^2}{P} \times t_{body}$, for *Region B*. Each task in *Region B* depends on all tasks in *Region A*, as shown in the TDG below.



2. Identify which remote accesses have to be performed during the execution of the parallel program, clearly identifying the processors involved in each remote access, the number of elements that need to be transferred and when the remote accesses should occur.

   Solution:

   Before the computation in *Region A* can start, each processor should access to those elements of `Matrix_a` that are needed, that is: each processor from 1 to $P-1$ should access to N/P consecutive rows (each one with N elements) from processor 0. Since this processor can only serve one remote request at a time, all these remote accesses are sequentialised. Before the computation in *Region B* can start, each processor should access to those elements of `Matrix_b` that are needed, that is: each processor (from 0 to $P-1$) should access to N/P consecutive rows (each one with N/P consecutive columns) from all other $P-1$ processors; and alternatively, each processor has to serve $P-1$ requests from the other processors which should be sequentialised.

3. Write the expression that determines the execution time with P processors, T_P, clearly identifying the contribution of the computation time and the overheads caused by data sharing.

Solution:

Remote accesses before starting the parallel execution in *Region A* are sequentialised because processor 0 can only serve 1 remote access at a time: this is why the cost of a remote access $(t_s + t_w \times N^2/P)$ is multiplied by P-1. Similarly, the P-1 remote accesses to the same processor before starting the parallel execution in *Region B* are also sequentialised because a processor can only serve 1 remote access at a time: this is why the cost of a remote access $(t_s + t_w \times (N/P)^2)$ is multiplied by P-1 too. Therefore, the expression for T_P is:

$T_p = T_P^{comp} + T_P^{data}$

$T_P^{comp} = N^2/P \times t_{body} + N^2/P \times t_{body} = 2 \times N^2/P \times t_{body}$

$T_P^{data} = (P-1) \times (t_s + t_w \times N^2/P) + (P-1) \times (t_s + t_w \times (N/P)^2)$.

Problem 3 (3 points) Assume a multiprocessor system with a hybrid NUMA/UMA architecture. The multiprocessor is composed of 2 identical NUMAnodes, each with 12 Gbytes of main memory. Each NUMAnode has 2 processors, each with its own private cache of 16 Mbytes. Memory and cache lines are 128 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. **First, we ask you to** answer the following two questions:

1. Compute the total number of bits that are necessary **in each cache memory** to maintain the coherence between caches **inside a NUMAnode**. Indicate also the function of those bits.

   Solution:

   We need 2 state bits (MSI) for each line of cache memory. Each cache memory has $(16 \times 2^{20}) \div 128$ lines, that is 2^{17} lines; therefore the number of bits per cache is $2^{17} \times 2 = 2^{18}$ bits.

2. Compute the total number of bits that are necessary **in each NUMAnode's directory** to maintain the coherence **among NUMAnodes**. Indicate also the function of those bits.

   Solution:

   We need 2 state bits (MSU) and 2 presence bits for each line of main memory. For the overall 12 GB, this is $(12 \times 2^{30}) \div 128$ lines, that is 12×2^{23} lines; therefore the number of bits in the directory is $12 \times 2^{23} \times (2+2) = 3 \times 2^{27}$ bits.

Now, given the following declaration for vector `x`:

```
#define N 1024
int x[N];
```

and assuming that: 1) the initial memory address of vector `x` is aligned to the start of a memory/cache line; 2) the size of an `int` data type is 4 bytes; and 3) processors 0 and 1 belong to NUMAnode0 and processors 2 and 3 belong to NUMAnode1. **We ask you to:**

3. Complete the table in the provided answer sheet with the necessary missing information: type of memory access (`read/write`), affected cache line (numbered from the first position where vector `x` is allocated), access in cache (hit/miss), CPU command for processor k ($PrRd_k/PrWr_k$), Bus transaction(s) from Snoopy in processor k ($BusRd_k/BusRdX_k/BusUpgr_k/Flush_k/Nothing$), cache line states (`I/S/M`), NUMA commands (yes/no), directory entry state (`U/S/M`) and presence bits (0/1, where the lowest ordered bit, the rightmost one, corresponds to NUMAnode0), to keep cache coherence, **AFTER the execution of each** memory access. **Note**: We are not asking for the coherence commands exchanged between NUMAnodes, we are only asking the Bus transactions within NUMAnodes to keep coherence resulted from local or remote memory access to NUMAnodes.

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state				NUMA commands	Directory entry	
					0	1	2	3		State	Presence bits
Processor 1 ..... x[4]	..	Miss	.....	.....	I	...	I	I	No	M	....
Processor 2 ..... x[16]	..	....	.....	.....	...	...	S	...	..	..	....
Processor 3 ..... x[32]	..	Hit	.....	$BusUpgr_3$	I	I	I	...	Yes	M	....
Processor 0 ..... x[32]	..	....	.....	.....	M	...	...	...	..	..	....
Processor 2 writes x[20]	..	....	.....	.....	...	...	...	...	..	..	....

Solution:

Observe that cache line size is 128 bytes and each integer is 4 bytes long. Therefore, 32 consecutive elements of vector x fit in one cache line. In the table below this means that `x[4]`, `x[16]` and `x[20]` belong to the same line (line 0) and `x[32]` is in a different line (line 1).

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state				NUMA commands	Directory entry	
					0	1	2	3		State	Presence bits
Processor 1 writes x[4]	0	Miss	PrWr1	BusRdX1	I	M	I	I	No	M	01
Processor 2 reads x[16]	0	Miss	PrRd2	BusRd2 / Flush1	I	S	S	I	Yes	S	11
Processor 3 writes x[32]	1	Hit	PrWr3	BusUpgr3	I	I	I	M	Yes	M	10
Processor 0 writes x[32]	1	Miss	PrWr0	BusRdX0 / Flush3	M	I	I	I	Yes	M	01
Processor 2 writes x[20]	0	Hit	PrWr2	BusUpgr2	I	I	M	I	Yes	M	10

Problem 4 (2.5 points) Given the following OpenMP code:

```
#define N          (1<<18)    /*  256*1024 */
#define N_THREADS (1<<4)    /*   16 */

int b[N];
int a[N];

#pragma omp parallel num_threads(N_THREADS)
{
  int thid = omp_get_thread_num();
  int chunk = 2;

  for (int ii = thid * chunk; ii < N; ii+= chunk*N_THREADS) {
    for (int i = ii; i < ii+chunk;  i++) {
           b[i] =  a[i];
    }
  }
}
```

Assume an SMP system with 16 CPUs, each with its own cache memory initially empty. To keep caches coherent the system uses a Snoopy–based write–invalidate MSI coherence protocol. Also assume cache lines of 128 bytes, that `int` size is 4 bytes, first element of vectors `a` and `b` are placed at the first position of memory line and rest of variables are stored in registers. Finally, we know that thread i runs on CPU i.

Although the code above is correct, its execution generates a lot of bus coherence transactions due to false sharing and bad exploitation of spatial locality, and as a consequence, high execution overheads. **We ask you:**

1. Indicate which threads are accessing to the first 6 elements of vector `a` and `b`.

   Solution:

   For both vectors `a` and `b`, elements 0 and 1 are accessed by thread 0, elements 2 and 3 are accessed by thread 1, elements 4 and 5 are accessed by thread 3, etc.

2. Assuming all processor caches are empty at the beginning of the code. Which types of coherence commands are generated by the snoopy controllers when the code is executed in parallel? Just indicate

the name of the coherence transactions and which accesses to variables provoke them. Would it be possible to count the number of coherence transactions of each type? Reason your answer.

Solution:

Note that first element of vectors `a` and `b` are placed at the first position of memory line and rest of variables are stored in registers. This means that first element of each vector is aligned to memory line (cache line). However, vector a and b are placed in different memory addresses and then, access to one vector doesn't provoke coherence commands affecting the other vector.

Therefore, accesses to vector b provoke BusRdX and Flush transactions[1]. Accesses to vector a only provoke BusRd transactions. For BusRdX and Flush transactions we cannot exactly count the number of bus transactions that will occur since we don't exactly know the order that each thread will access to the elements of vector b. For BusRd transactions we can say that there will be one per chunk of elements accessed ($N/2$ in total).

Details not needed in your answer to the question: In the case of the BusRdx and Flush transactions we can analyze it a little bit more:

- *For BusRdx:*
  - *the minimum number of transactions is one per chunk of elements accessed ($N/2$ in total).*
  - *the maximum number of transactions is the total number of accesses to vector b (N).*
- *For Flush: The number of Flush transactions will be smaller than the number of BusRdX transactions. Note that the first thread writing for the first time to a memory line does not provoke Flush transaction because there is not a dirty copy of this memory line in the rest of the caches. Later, when there exists a previous dirty copy of the memory line, any write to this memory line that provokes a new BusRdX will also provoke a Flush transaction.*

3. Briefly describe the reason why the execution of the code above leads to a a false sharing situation and a lack of spatial locality exploitation. Propose a modification in the code that avoids both situations at once.

   Solution:

   Cache line size is 128 bytes. Each integer is 4 bytes long. Therefore, every 32 consecutive elements of vector b can fit in one cache line (similarly for vector a). In the case of the code, 16 threads, running in different CPUs, access to consecutive elements (in this case each thread access 2 of the 32 elements that fit in a cache line) of vector b (similarly for vector a).

   - In the case of vector b this may provoke a false sharing situation if two or more threads alternatively write to the same cache line.
   - Lack of locality happens because each thread has a cache miss when accessing the first element of each `chunk`, and only can exploit spatial locality for the second element of the chunk.

   To solve both problems, we only need to set the value of `chunk` to 32. This provokes that threads access to different cache lines and fully exploit spatial locality.

[1] BusUpgr transactions are not possible because we are only writing to vector b

PAR – In-Term Exam – Course 2021/22-Q2

April 6^{th}, 2022

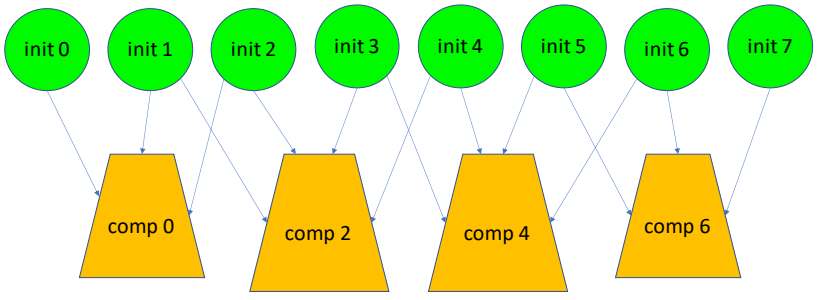
Problem 1 (4 points) Given the following code using *Tareador* for task annotations:

```
#define R  8
#define BS 2
int x[R][N], y[R][N];
...
// initialization phase
for (int i=0; i<R; i++) {
    tareador_start_task ("init");
    for (int k=0; k<N; k++)
        x[i][k] = foo (i); // no other memory accesses inside foo
    tareador_end_task ("init");
}
// computation phase
for (int i=0; i<R; i+=BS) {
    tareador_start_task ("comp");
    for (int ii=max(1,i); ii<min(R-1,i+BS); ii++)
        for (int k=0; k<N; k++)
            y[ii][k] = compute (x[ii][k], x[ii-1][k], x[ii+1][k]);
                       // no other memory accesses inside compute
    tareador_end_task ("comp");
}
```

Assume that the execution time for functions `foo` and `compute` is 2 and 20 time units, respectively, and that the execution cost in time for the rest of the code is negligible. **We ask you to**:

1. Draw the *Task Dependence Graph* (TDG) based on the above *Tareador* task definitions. Include for each task its name followed by the iteration number that generates it and its cost in time units (as a function of `N`).

   Solution:



Where task costs are:
cost (*init i*) = 2 x N where *i=0..7*
cost (*comp 0*) = cost (*comp 6*) = 20 x 1 x N
cost (*comp 2*) = cost(*comp 4*) = 20 x 2 x N

2. Calculate the values for T_1, T_∞ and amount of *Parallelism* (as a function of `N`).

   Solution:

   $T_1 = 8 \times 2 \times N + 2 \times 20 \times 1 \times N + 2 \times 40 \times N = 136 \times N$

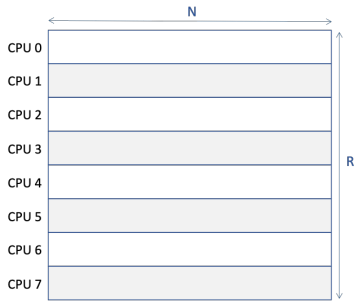
   $T_\infty = 2 \times N + 40 \times N = 42 \times N$, determined by any of the critical paths going through either `comp 2` or `comp 4`. For example, `init 2`–`comp 2`.

   $Parallelism = (136 \times N)/(42 \times N) = 3.2$.

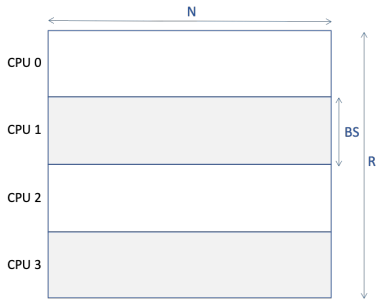
   Although it is not asked in the problem, observe that in this case $P_{min} = 6$. Why?

3. Given the following task allocation for $P = 8$ processors (CPUs) and mapping of matrices `x` and `y` by rows to processors ($R = 8$ and $BS = 2$):

Processor	Task assignment
0	init 0, comp 0
1	init 1, comp 2
2	init 2, comp 4
3	init 3, comp 6
4	init 4
5	init 5
6	init 6
7	init 7



Matrix x mapped to 8 processors



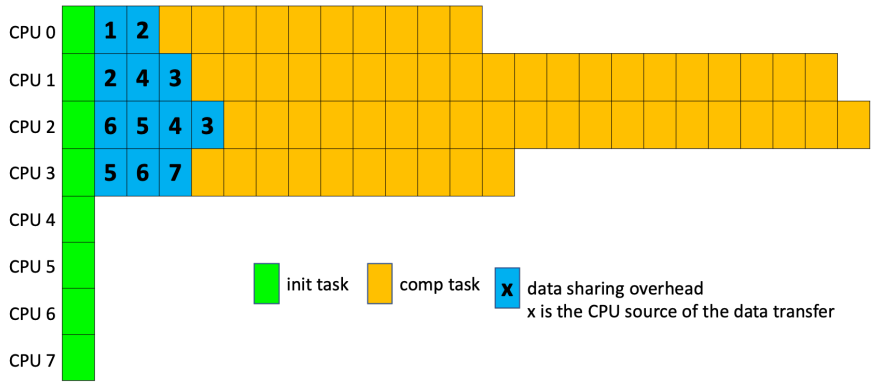
Matrix y mapped to 4 processors

Write the expression that determines the execution time, T_8 as a function of `N`, clearly identifying the contribution of the computation time and the data sharing overheads, assuming the data sharing model explained in class in which the overhead to perform a remote memory access is $t_s + t_w \times m$, being t_s the start-up time, t_w the time to transfer one element and m the number of elements to be transferred; at any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor.

Solution:

First let's find out the critical path considering data sharing overheads. Observe that `comp 0` executed by processor 0 has to perform two remote accesses (to get the row initialized in `init 1` by processor 1 and to get the row initialized in `init 2` by processor 2); `comp 2` executed by processor 1 has to perform three remote accesses (to get the row initialized in `init 2` by processor 2, the row initialized in `init 3` by processor 3 and the row initialized in `init 4` by processor 4); `comp 4` executed by processor 2 has to perform four remote accesses (to get the row initialized in `init 3` by processor 3, the row initialized in `init 4` by processor 4, the row initialized in `init 5` by processor 5, and the row initialized in `init 6` by processor 6); and finally, `comp 6` executed by processor 3 has to perform three remote accesses (to get the row initialized in `init 5` by processor 5, the row initialized in `init 6` by processor 6 and the row initialized in `init 7` by processor 7). Therefore it is clear that the critical path is determined by processor 3 executing `init 2` and `comp 4`,

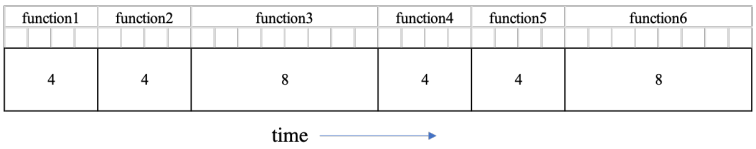
The following temporal diagram shows the execution taking into account the data sharing overheads. There is also indicated a possible scheduling of data transfers to show that it is possible to make it.



The data sharing overhead cost is given by the expression: $T_8^{datasharing} = 4 \times (t_s + t_w \times N)$

so $T_8 = T_8^{comp} + T_8^{datasharing} = (2 \times N + 40 \times N) + (4 \times (t_s + t_w \times N))$

Problem 2 (2 points) The following figure shows the timing diagram for the sequential execution of an application on 1 processor:



The figure has a set of rectangles, each rectangle representing the serial execution of a function with its associated cost in time units. In a first attempt the programmer has been able to parallelize functions 1, 2, 3, 4 and 5, which can be ideally decomposed; **function 6 remains sequential. We ask you to answer the following questions**:

1. What is the parallel fraction ϕ?

   Solution: $\phi = \frac{4+4+8+4+4}{4+4+8+4+4+8} = \frac{24}{32} = \frac{3}{4}$

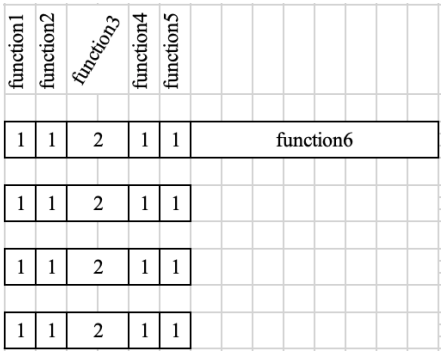
2. Which would be the maximum speedup that you could achieve based on Amdahl's law ($S_{p\to\infty}$)?

   Solution: $S_{p\to\infty} = \frac{1}{(1-\phi)} = 4$

3. Draw the time diagram of an ideal parallel execution of the application to obtain maximum speed-up with 4 processors. Which is the value for S_4 that is achieved?

   Solution:

   The proposed parallel execution is shown below, giving the ideal speed–up for 4 processors: $S_4 = \frac{T_1}{T_4} = \frac{32}{14} = \frac{16}{7} = 2.28$.



In a second attempt the programmer has been able to fully parallelize all funtions in the application (total workload of 32 time units), with ideal scalability. Please **answer the following two questions:**

4. Which should be the workload assigned to each processor when parallelized with $P = 8$ processors and strong scaling?

   Solution: $work\_load\_per\_thread = \frac{32}{8} = 4$ time units.

5. Which should be the workload assigned to each processor when parallelized with $P = 8$ processors and weak scaling?

   Solution: $work\_load\_per\_thread = 32$ time units.

Problem 3 (4 points) Given the following code excerpt, including OpenMP directives, to be executed on a UMA architecture with 8 processors, each one with a 1 MB fully-associative cache memory, and cache coherence maintained with the simplest write–invalidate MSI protocol explained in class, sharing the access to 16 GB of main memory:

```
#define NUM_THREADS 8
#define N 262144                  // N multiple of NUM_THREADS

int a[N], b[N];
struct {
    int zeros[NUM_THREADS];
    int positives;
} count;

#pragma omp parallel num_threads(NUM_THREADS)
{
    int id = omp_get_thread_num();
    int num_elems = N / NUM_THREADS;
    for (int i=id*num_elems; i < (id+1)*num_elems; i++) {
        a[i] = a[i] * b[i];
        if (a[i] == 0) count.zeros[id]++;
        if (a[i]>0)
            #pragma omp atomic
            count.positives++;
    }
}
```

Observe that each implicit task executes a chunk of `num_elems` consecutive iterations of the loop. Assume that processor i executes implicit task with $id = i$. Also assume that each integer (`int`) variable occupies 4 bytes, a memory (cache) line occupies 32 bytes, and that the initial address of vectors `a` and `b` as well as structure `count` are aligned with the start of a cache line. All other variables are stored in registers (not in memory). Processors have no cached copies of `a`, `b` and `count` in their private cache memories before starting the execution of the parallel region. We ask you to **answer the following questions:**

1. How many *BusRd*, *BusRdX*, *BusUpgr* and *Flush* commands will be placed BY EACH ONE of the 8 processors on the shared interconnection network (bus) due to accesses to **vectors `a` and `b`**.

   Solution: Each vector `a` and `b` occupies (262144 elements * 4 bytes/element) = 1 MB. Since elements are equally distributed among the 8 processors, each local cache will need 128 KB for each vector, which correspond to 128 KB / 32 bytes/line = 4096 lines. Therefore each processor will place on the bus 4096 *BusRd* command to read its elements of `a` and the same for `b`; since `a` is also written, 4096 *BusUpgr* commands (not *BusRdX* since the access results in a cache hit) will also be placed on the bus from each processor. Both vectors fit in the caches, so no *Flush* commands are generated for the lines of vector `a`. In total: 8192 BusRd and 4096 BusUpgr, no BusRdX and no Flush.

2. How many memory lines does **structure `count`** occupy? Does the access to `count` in the program causes *true* and/or *false* sharing? Briefly reason your answer.

   Solution: Structure `count` occupies (8 + 1) int * 4 bytes/int = 36 bytes. Therefore it occupies 2 cache lines. The access to `count.positives` causes *true* sharing, since that field is updated by all processors whenever a positive value is computed for `a`. The access to `count.zeros` causes *false* sharing, since each processor is accessing to a different element of the vector `zeros` but all of them reside in the same cache line.

3. In order to reduce the coherence traffic that is caused by the accesses to **vector `count.zeros`** the programmer has changed the definition of `count` and the access to it, as follows:

```
struct {
    int zeros[NUM_THREADS*PADDING];
    int positives;
} count;
...
        if (a[i] == 0) count.zeros[id*PADDING]++;
...
```

Which would be the most appropriate value for constant `PADDING`? How many *BusRd*, *BusRdX*, *BusUpgr* and *Flush* commands will be placed BY EACH ONE of the 8 processors on the shared interconnection network (bus) due to accesses to `count.zeros`?

Solution: Since this access is causing a *false* sharing problem, coherence traffic can only be eliminated if we use padding, so that each element of vector `zeros` occupies a different memory line. We achieve this with `PADDING=8`, so that the elements accessed by different processors are $8 \times 4 = 32$ bytes apart. With this each processor will place one *BusRd* and one *BusUpgr* command in the shared bus during the whole execution of the parallel region.

If the multiprocessor architecture is upgraded to a NUMA system with 8 nodes, each node with a single processor, a private cache memory of 1 MB, and a portion of main memory of 2 GB, and all variables (vectors `a` and `b` and structure `count`) are physically mapped to NUMA node 0 by the operating system (first touch policy) before starting the execution of the parallel region, with no cached copies in any of the nodes. We ask you to **answer the following questions:**

4. How many entries in the directory **of each NUMA node** will be used to store the coherence information for **vectors `a` and `b`**? After the execution of the parallel region, how many bits in the *sharers list* of each of these directory entries will be set to 1 and in which state will those memory lines be?

   Solution: As calculated before, each vector `a` and `b` occupies (262144 elements * 4 bytes/element)/32 bytes/line = 32768 lines. Since all elements are mapped in NUMA node 0, 32768*2=65536 entries will be used in that node. No entries will be used in the rest of nodes. Since each line is only accessed by the processor of a single NUMA node, only one bit in the *sharers list* will be set to one, with M state for vector `a` and S state for vector `b`.

5. After executing the program the programmer realized that the access to `count.positives` was creating a performance bottleneck, so she/he proposed the following program transformation for the parallel region, making use of a per–thread copy `tmppos` of the shared variable `count.positives`, which is accumulated into it before exiting the parallel region:

```
#pragma omp parallel num_threads(NUM_THREADS)
{
    int tmppos = 0;
    int id = omp_get_thread_num();
    int num_elems = N / NUM_THREADS;
    for (int i=id*num_elems; i < (id+1)*num_elems; i++) {
        a[i] = a[i] * b[i];
        if (a[i] == 0) count.zeros[id]++;
        if (a[i]>0) tmppos++;
    }
    #pragma omp atomic
    count.positives = count.positives + tmppos;
}
```

Assume that the processor in NUMA node 0 is the first executing `#pragma omp atomic` and that its execution ensures read/write atomicity in the access to `count.positives` (i.e. while one processor is accessing to it inside the `pragma` no other processors will be able to access it). Which of the following statements is/are true? **(selected wrong statements penalize the mark that you can obtain in this question)**

(a) The proposed transformation can never improve performance since the number of of positives found in vector `a` does not change.

(b) If the number of positive results in vector `a` is much larger than 1, the ONLY improvement in performance comes from the fact that `#pragma omp atomic` has been removed from the loop body.

(c) The last processor x executing `#pragma omp atomic` will first send a $RdReq_{x \to 0}$ command, which will provoke a $Fetch_{0 \to y}$ command to the remote node y that performed the previous update. As a consequence, $Dreply_{y \to 0}$ and $Dreply_{0 \to x}$ will be generated in order to get the updated line in processor x. The line is also updated in main memory of node 0.

(d) After that, the same processor x will send an $UpgrReq_{x \to 0}$ to change the state of the line in the home node and an $Invalidate_{x \to y}$ command to invalidate the copy in cache of node y, which will reply with an $Ack_{y \to x}$ to notify the completion of the command.

(e) At the end of the parallel region the directory entry associated to the memory line containing `count.positives` will be in state M with only the bit associated to node x in the sharers list active to 1.

Note: *$Command_{a \to b}$ refers to a NUMA command sent from node a to node b.*

Solution:

{False, False, True, False, True}

PAR – In-Term Exam – Course 2022/23-Q1

November 3^{rd}, 2022

Problem 1 (3.0 points) Given the following code:

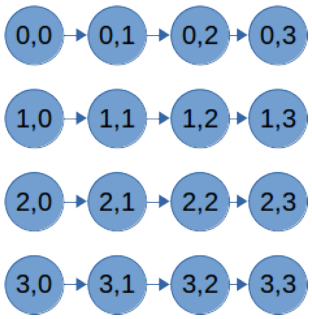
```
#define N  256
#define BS  64
int m[N][N];

for (int ii=0; ii<N/BS; ii++) {
   for (int jj=0; jj<N/BS; jj++) {
      tareador_start_task("task");

      // In general, a task processes a block of BSxBS elements
      // However, if the task is labeled with (ii,jj=0),
      //          this task processes BSx(BS-1) elements
      int i_start = ii*BS; int i_end = i_start+BS;
      int j_start = jj*BS; int j_end = j_start+BS;
      for (int i=i_start; i<i_end; i++)
        for (int j=max(j_start,1); j<j_end; j++)
          m[i][j] = compute (m[i][j], m[i][j-1]);  // tc t.u. (time units)

      tareador_end_task("task");
} }
```

Assume the innermost loop body takes t_c t.u., all variables but matrix m are in registers, function `compute` only reads the values received as arguments and does not modify other positions in the memory, BS perfectly divides N, being BS and N defined in the code above. **We ask you:**

1. (1.0 points) Draw the task dependence graph (TDG), indicating the cost of each task as a function of BS and t_c. Label each task with the values of ii and jj.

   Solution:

   Tasks (ii,jj) when jj!=0 have a cost of $BS \times BS \times t_c$. Tasks (ii,jj) when jj=0 have a cost $BS \times (BS-1) \times t_c$

2. (1.0 points) Compute T_1, T_∞, P_{min} as a function of BS, N and t_c.

   Solution:

   T_1 can be computed as:

   $T_1 = N \times (N-1) \times t_c$;

   T_∞ is the execution time of any of the rows of tasks of the TDG:

   $T_\infty = (BS-1) \times (BS) \times t_c + (\frac{N}{BS} - 1) \times (BS^2) \times t_c$;

This T_∞ is obtained when one processor is assigned to the computation of a row of tasks in the TDG. Therefore:

$P_{min} = \frac{N}{BS}$;

3. (1.0 points) Assuming the assignment of tasks to 4 processors of the table below, calculate T_4 and speedup S_4.

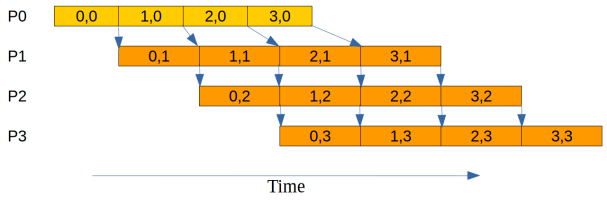
Processor	Tasks
P_0	$(0,0),(1,0),(2,0),(3,0)$
P_1	$(0,1),(1,1),(2,1),(3,1)$
P_2	$(0,2),(1,2),(2,2),(3,2)$
P_3	$(0,3),(1,3),(2,3),(3,3)$

Solution:

$S_4 = \frac{T_1}{T_4}$

$T_1 = N \times (N-1) \times t_c = 65280 \times t_c$ t.u.

Figure below shows the execution timeline considering the assignment of tasks to 4 processors above and their dependences. Each processor executes $\frac{N}{BS}$ tasks. Let's identify those that originate the critical path in the execution timeline. Processor 0 executes tasks with cost $BS \times (BS-1) \times t_c$. First task of Processor 0 contributes to T_4: $((BS-1) \times BS \times t_c)$. Then, first tasks of processors 1 an 2 also contribute to T_4: $((\frac{N}{BS} - 2) \times BS^2 \times t_c)$. Finally, all tasks executed in processor 3 contribute to T_4: $(\frac{N}{BS} \times BS^2 \times t_c)$.



$T_4 = ((BS-1) \times BS + (\frac{N}{BS} - 2) \times BS^2 + \frac{N}{BS} \times BS^2) \times t_c = 28608 \times t_c$ t.u.

$S_4 = \frac{T_1}{T_4} = 2.28\times$

Problem 2 (2.0 points) Given the same code and mapping of tasks to 4 processors as in the previous exercise, assume

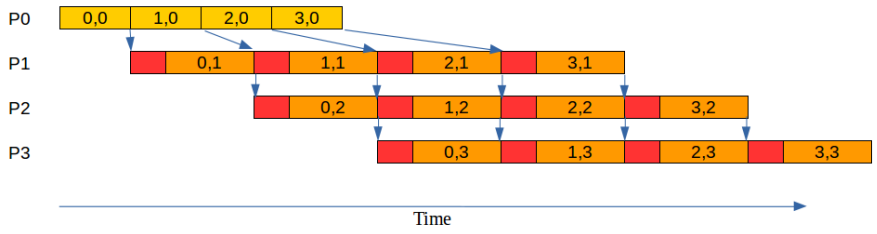
- A distributed-memory architecture with $P = 4$ processors;
- Matrix m is initially distributed by columns (BS consecutive columns per processor).
- Data sharing model with $t_{comm} = t_s + W \times t_w$, being W the number of elements to transfer, and t_s and t_w the start-up time and transfer time of one element, respectively;
- The execution time for a single iteration of the innermost loop body takes t_c t.u.

We ask you: Draw the execution timeline of the execution of tasks and write the expression that determines the execution time T_P, clearly indicating the contribution of the computation time $T_{P\,comp}$ and data sharing overhead $T_{P\,mov}$, as a function of N, BS, P, t_c, t_s and t_w.

Solution:

Figure below shows the execution timeline considering the assignment of tasks to 4 processors above, their dependences and the remote memory accesses. Processors 1, 2 and 3 have to wait for completness of tasks $ii, jj-1$ whose results are saved in the remote memory of the previous processor. As we are now considering data sharing overheads, processors 1, 2 and 3 have to perform a remote memory access of the BS elements of the left boundary before each task is executed. Therefore, the cost of each communication is $t_s + BS \times t_w$. Tasks in processor 0 perform local memory access with no data sharing overhead.

Each processor executes $\frac{N}{BS}$ tasks.



In particular, the contribution of computation and data sharing overheads is the following:

- Processor 0 executes tasks with cost $BS \times (BS-1) \times t_c$. First task of Processor 0 contributes to T_4: $(BS-1) \times BS \times t_c$.
- First tasks of processors 1 an 2 also contribute to T_4: $(\frac{N}{BS} - 2) \times ((t_s + BS \times t_w) + (BS^2 \times t_c))$.
- Finally, all tasks executed in processor 3 contribute to T_4: $(\frac{N}{BS}) \times ((t_s + BS \times t_w) + (BS^2 \times t_c))$.

$T_4 = T_{comp} + T_{comm}$, being:
$T_{comp} = ((BS-1) \times BS + (\frac{N}{BS} - 2) \times BS^2 + \frac{N}{BS} \times BS^2) \times t_c$
$T_{comm} = (\frac{N}{BS} - 2) \times (t_s + BS \times t_w) + \frac{N}{BS} \times (t_s + BS \times t_w) = (2 \times \frac{N}{BS} - 2) \times (t_s + BS \times t_w)$

Problem 3 (5.0 points) Consider a multiprocessor system with a hybrid NUMA/UMA architecture which is composed by 3 identical NUMAnodes. Each NUMAnode has 20 Gbytes of main memory and 2 processors with its own private cache of 8 Mbytes. The memory cache lines are 32 bytes wide, and data coherence is guaranteed 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. (1.0 points) Compute the total number of bits that are necessary **in each cache memory** to maintain the coherence, indicating the function of those bits.

   Solution:

   We need 2 state bits (MSI) for each line of cache memory. Each cache memory has $(8 \times 2^{20}) \div 32$ lines, that is 2^{18} lines; therefore the number of bits per cache is $2^{18} \times 2 = 2^{19}$ bits.

2. (1.0 points) Compute the total number of bits that are necessary **in each NUMAnode's directory** to maintain the coherence, indicating the function of those bits.

   Solution:

   We need 2 state bits (MSU) and 3 presence bits for each line of main memory. For the overall 20 GB, this is $(20 \times 2^{30}) \div 32$ lines, that is 20×2^{25} lines; therefore the number of bits in the directory is $20 \times 2^{25} \times (2 + 3) = 100 \times 2^{25}$ bits.

Given the following OpenMP code excerpt which is executed on processor 0 from the previous described multiprocessor system:

```
#define N (6*1024)
#define NUM_THREADS 6
int v[N], count[NUM_THREADS];
...
for (int k = 0; k < NUM_THREADS; k++)
   count[k]=0;

/*** POINT A ***/

#pragma omp parallel num_threads (NUM_THREADS)
{
   int id=omp_get_thread_num();
   int num_iter = N/NUM_THREADS;

   for (int k = id*num_iter; k < id*num_iter+num_iter; k++) {
       int value = v[k];
       if (is_prime(value)) /* returns true if "value" is a prime number */
           count[id]++;
   }
}
```

and assuming that: 1) the initial memory address of vector `count` is aligned to the start of a memory/cache line; 2) the size of an `int` data type is 4 bytes; and 3) processors 0 and 1 belong to NUMAnode0, processors 2 and 3 belong to NUMAnode1 and processors 4 and 5 belong to NUMAnode2; and 4) the Operating System applies the "first touch" policy for data allocation in memory. **We ask you to:**

3. (1.25 points) Complete the table in the provided answer sheet with the required information: affected cache line (numbered from the first position where vector `count` is allocated), cache line states (`I/S/M`) in processors 0 to 5, directory entry state (`U/S/M`) and presence bits (0/1, where the lowest ordered bit, the rightmost one, corresponds to NUMAnode0), to keep cache coherence, **when the execution of the previous code reaches POINT A**.

   Solution: The cache line size is 32 bytes, and each element of the vector occupies 4 bytes, so the entire vector `count` fits in a unique memory line.

	Affected line	Home NUMAnode	Cache line state						Directory entry	
			0	1	2	3	4	5	State	Presence bits
count[0]	0	0	M	-	-	-	-	-	M	001
count[1]	0	0	M	-	-	-	-	-	M	001
count[2]	0	0	M	-	-	-	-	-	M	001
count[3]	0	0	M	-	-	-	-	-	M	001
count[4]	0	0	M	-	-	-	-	-	M	001
count[5]	0	0	M	-	-	-	-	-	M	001

4. (1.25 points) Complete the table in the provided answer sheet with the required information: affected cache line, access in cache (hit/miss), CPU command for processor k ($PrRd_k/PrWr_k$), Bus transaction(s) from Snoopy in processor k ($BusRd_k$ / $BusRdX_k$ / $BusUpgr_k$ / $Flush_k$ / $Nothing$), cache line states in processors 0 to 5, directory entry state and presence bits, to keep cache coherence, **AFTER the execution of each** memory access in the table. Assume initially memory and cache states from previous question.

   Solution:

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state						Directory entry	
					0	1	2	3	4	5	State	Presence bits
Processor 1 reads count[1]	0	Miss	PrRd1	BusRd1 / Flush0	S	S	-	-	-	-	S	001
Processor 0 reads count[0]	0	Hit	PrRd0	Nothing	S	S	-	-	-	-	S	001
Processor 1 writes count[1]	0	Hit	PrWr1	BusUpgr1	I	M	-	-	-	-	M	001
Processor 2 reads count[2]	0	Miss	PrRd2	BusRd2 / Flush1	I	S	S	-	-	-	S	011
Processor 0 writes count[0]	0	Miss	PrWr0	BusRdX0	M	I	I	-	-	-	M	001

   Notice that, as the entirely vector `count` fits in a cache line, access to any element of the vector provokes access to the same cache line.

5. (0.5 points) Have you observed any potential efficiency problem in the previous code? Justify briefly your answer.

   Solution:

   False sharing: Vector `count` lies on a single memory line, then all the threads are eventually accessing for writing the same cache line during the concurrent execution of the program, but to different memory addresses, resulting in unnecessary coherence traffic, even between nodes.

   Possible memory bottleneck in the access to memory of processor 0: Vector count is entirely allocated in that memory, and all the threads have to access to it during execution.

Student name: .....

Answer for question 3.3

	Affected line	Home NUMAnode	Cache line state						Directory entry	
			0	1	2	3	4	5	State	Presence bits
count[0]										
count[1]										
count[2]										
count[3]										
count[4]										
count[5]										

Answer for question 3.4

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state						Directory entry	
					0	1	2	3	4	5	State	Presence bits
Processor 1 reads count[1]												
Processor 0 reads count[0]												
Processor 1 writes count[1]												
Processor 2 reads count[2]												
Processor 0 writes count[0]												

Part II

Final Exams

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

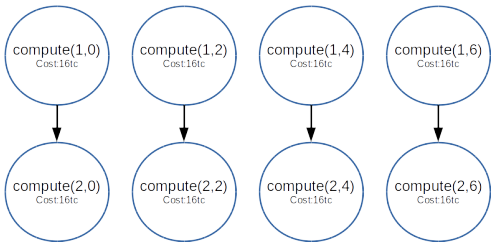
Problem 1 (5 points) Given the following nested loops in a C program instrumented with *Tareador*:

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

We ask you to answer the following questions:

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

   Solution: 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.

$$P_{min} = 4$$

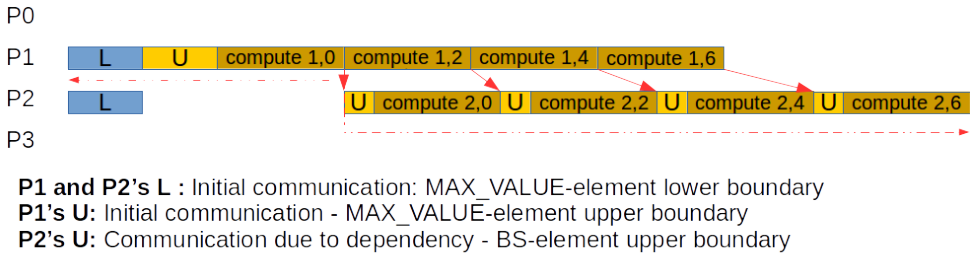
3. Write the expression that determines the execution time T_4 for the program, clearly indicating the contribution of the computation time T_4^{comp} and the data sharing overhead T_4^{mov}, for the following assignment of tasks to threads and processors: tasks `compute(1,jj)` are assigned to `thread 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 initial communication, 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

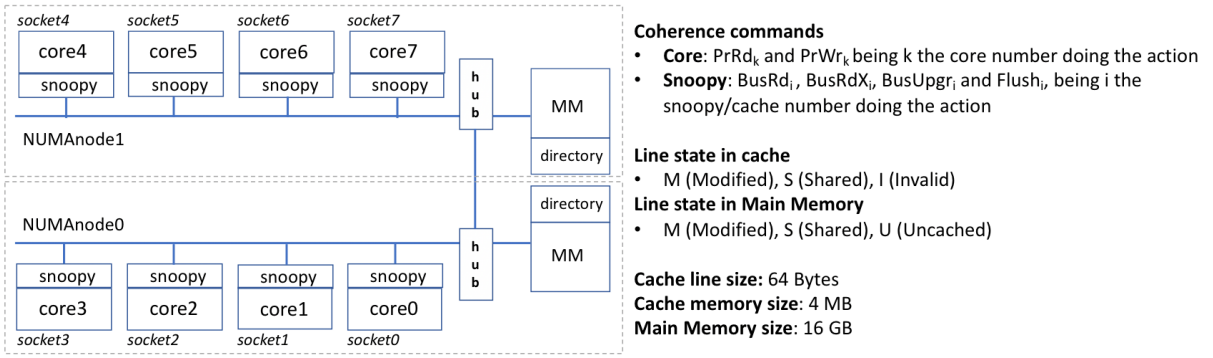
Note that we have added a dashed red line to indicate one possible way to compute the critical path. The critical path cost includes T_4^{comp} and T_4^{mov}:

$T_4^{comp} = 5 \times 2 \times 8 \times t_c$

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

$T_4 = T_4^{comp} + T_4^{mov}$

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.



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 18^{th}, 2021

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

version 1:

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

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

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

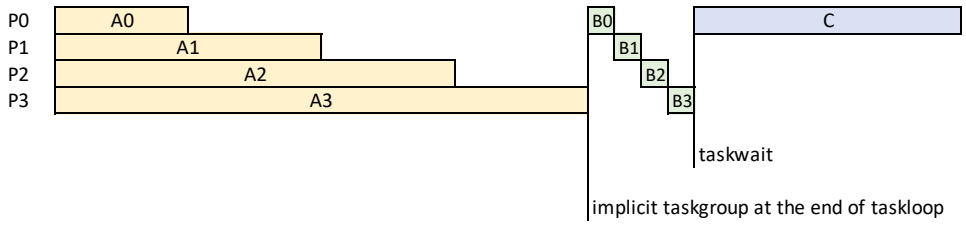
version 2:

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

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

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

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

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

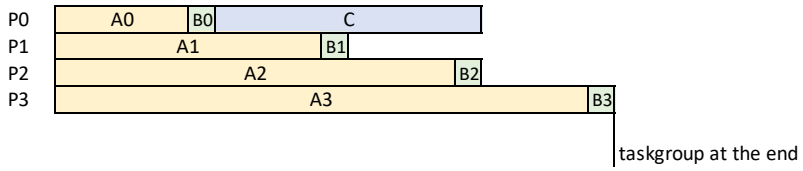
   In *version 1* dependences between tasks Ai and Bi are satisfied with the implicit `taskgroup` at the end of the `taskloop` region; dependences among tasks bi due to the sharing of global variable `x` are satisfied with task dependences using the `depend` clause; finally, the dependence between tasks 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 16^{th}, 2021

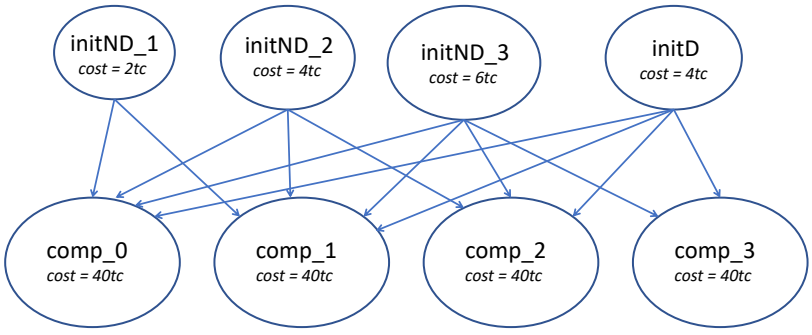
Problem 1 (2 points) Given the following nested loops in a C program instrumented with *Tareador*:

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

// 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:

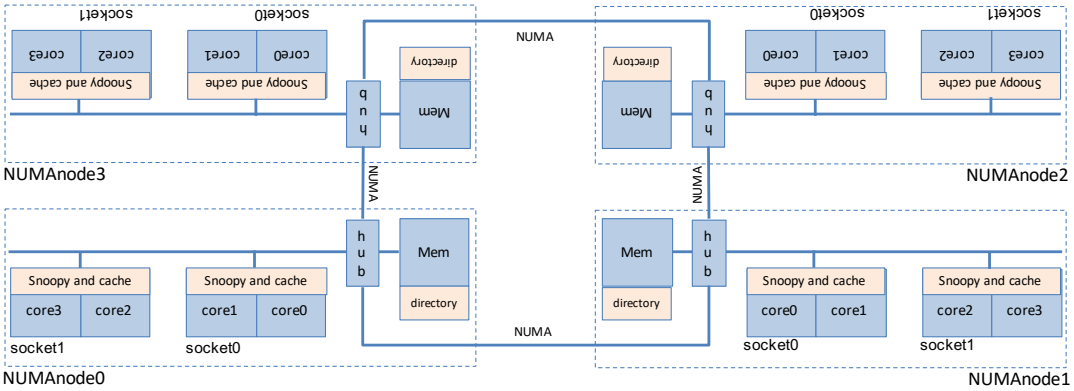
   P_0 = `initND_1, comp_0`

   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.

```
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 clause and omp_in_final intrinsic 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.

```
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
        if (!omp_in_final()) {
            #pragma omp task shared(min_max1) final(depth>=MAX_DEPTH)
            min_max1 = find_min_max_rec(v, n2, depth+1);
            #pragma omp task shared(min_max2) final(depth>=MAX_DEPTH)
            min_max2 = find_min_max_rec(v+n2, n2, depth+1);
            #pragma omp taskwait
        } else {
            min_max1 = find_min_max_rec(v, n2, depth+1);
            min_max2 = find_min_max_rec(v+n2, n2, depth+1);
        }

        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`.

PAR – Final Exam – Course 2021/22-Q1

January 17^{th}, 2022

Problem 1 (2.5 points)

The following code computes matrix `u[N][N]` by blocks of $BS \times N$ elements, with `N` very large:

```
double u[N][N];

// Compute elements in a block of BS x N elements
void compute_row_block(int ii) {
    for (int i=max(1, ii); i<min(ii+BS, N-1); i++)
        for (int j=1; j<N-1; j++) {
            double tmp = u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j];
            u[i][j] = tmp/4;
        }
}

void main() {
    int NB = 2*P;          // Total number of row blocks
    int BS = N/NB;         // Number of rows per block

    // EVEN loop: traversing all EVEN blocks
    for (int ii=0; ii<NB; ii+=2)
        compute_row_block(ii*BS);

    // ODD loop: traversing all ODD blocks
    for (int ii=1; ii<NB; ii+=2)
        compute_row_block(ii*BS);
}
```

In this code the computation is divided in two parts: the so called *EVEN* loop computing half of the blocks first (blocks 0, 2, ...), and the so called *ODD* loop computing the other half of the blocks later (blocks 1, 3, ...). Before answering the first question below, think about the parallel execution opportunities in this code when defining each iteration of the *EVEN* and *ODD* loops as a task. Could all the tasks in the *EVEN* loop be executed in parallel? Could all the tasks in the *ODD* loop be executed in parallel? And could the execution of tasks in the *ODD* loop be performed in parallel with the execution of tasks in the *EVEN* loop?. Having all that in mind, **we first ask you to:**

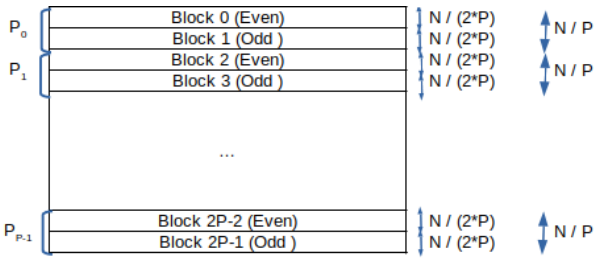
1. Write the expression for the contribution to the total parallel time T_P coming from the parallel computation time T_P^{comp} on an ideal machine with P processors, assuming that: 1) $NB = 2 * P$ perfectly divides the problem size `N`; 2) the execution time for a single iteration of the innermost loop body in routine `compute_row_block` takes t_c time units; and 3) no parallelisation overheads should be considered at this point.

   Solution:

   Since N is very large, we can consider $N - 2 \approx N$. Then,

   $T_P^{comp} = 2 \times BS \times N \times t_c = N^2/P \times t_c$

   Explanation:



All the tasks in the *EVEN* loop can be executed in parallel. All the tasks in the ODD loop can also be executed in parallel. However, tasks in the *ODD* loop can only be executed once the tasks in the EVEN loop computing the surrounding blocks have been completed. Each of the two loops iterate P times. Therefore, with P processors each processor will execute only 1 iteration of each loop. Since N is large, we can consider $N - 2 \approx N$. Thus, each processor computes N/P rows of size N, for a total of N^2/P executions of the innermost loop of routine `compute_row_block`. Note that half of them correspond to its *EVEN* block and the other half to its *ODD* block.

Next we consider that the ideal machine has the memory distributed among the P processors, In that machine, matrix `u` is divided row-wise in $NB = 2 * P$ blocks, where each block contains $BS = N/(2 * P)$ consecutive rows. Pairs of consecutive blocks are assigned to the same processor, so for example blocks 0 and 1 are owned by processor 0, blocks 2 and 3 are owned by processor 1; and so on and so forth. Each processor is in charge of computing all the rows in the blocks that are assigned to it, and therefore it will have to execute one iteration of the *EVEN* loop and one iteration of the *ODD* loop. Before answering the second question below, and having in mind the assignment of blocks and iterations, think about the need for processors to perform any remote access before starting the execution of tasks in the *EVEN* loop, or before starting the execution of tasks in the *ODD* loop; and, if affirmative, how many elements need to be transferred in each case and which processors to do them. Having all that in mind, next **we ask you to:**

2. Write the expression for the contribution to T_P coming from the data sharing overheads $T_P^{data\_sharing}$, if any, as part of the overall $T_P = T_P^{comp} + T_P^{data\_sharing}$. For that you should consider the data sharing model explained in class. Accesses to local memory are performed with zero overhead; accesses to remote memory take $t_{comm} = t_s + m \times t_w$, being t_s and t_w the start–up time for the remote access and transfer time of one element, respectively. At a given moment, a processor can only perform one remote memory access to another processor, and can only serve a remote memory access from another.

   Solution:

   $T_P^{data\_sharing} = 2 \times (t_s + N \times t_w)$

   Explanation:

   In general, before an EVEN block can be computed a processor needs to perform a remote access to the N elements in the last row of the block above, which is owned by the previous processor (except for processor 0). No remote access is required to access the first row of the next block, because it is an ODD block owned by the same processor. All the processors can perform such remote accesses in parallel.

   An ODD block can only be computed once its surrounding EVEN blocks are computed. Before an ODD block can be computed a processor needs to perform a remote access to the N elements in the first row of the block below, which is owned by the next processor. No remote access is required to access the last row of the previous block, because it is an EVEN block owned by the same processor. All the processors can perform such remote accesses in parallel.

Problem 2 (2.5 points)

Given the following code fragment:

```
#define CACHE_LINE_SIZE  128
#define DBSize           (32*1024*1024)
#define NTHREADS         3

int count[NTHREADS];
int DB[DBSize];
int key;

// Initialization loop
for (int i = 0; i < NTHREADS; i++)
    count[i] = 0;
```

```
#pragma omp parallel num_threads(NTHREADS)
{
    int my_id = omp_get_thread_num();
    for (int i = my_id; i < DBSize; i += NTHREADS) {
        // read access to DB[i]
        if (DB[i]==key)
            // read access to count[my_id] followed by a write access
            count[my_id] = count[my_id] + 1;
    }
}
```

Assume a shared–memory UMA parallel system with 3 processors, each one with its own (private) cache memory. Data coherence in the system is maintained using Write Invalidate `MSI` protocol, with a Snoopy attached to each cache memory. Also assume 1) empty caches at the beginning of the program; 2) `count[0]` and `DB[0]` are aligned to the beginning of different memory lines; 3) the rest of variables are stored in registers; and 4) the size for a variable of type `int` is 4 bytes and cache line size is 128 bytes. **We ask you:**

1. Assume thread 0, running on processor 0, starts executing the sequential part of the program until the parallel region starts (i.e. it executes the initialization loop). How many cache lines are used to store the full `count` vector after the execution of the initialization loop? What is the cache coherence state for each cache line storing elements of `count` and in which private cache it is located?

   Solution:

   We only require one cache line to store the three elements of the vector `count`. The coherence state of the cache line will be M (Modified) and the memory cache that will store the vector is MC_0.

2. Assume the cache states in previous question (after the execution of the initialization loop) and that each thread i runs on processor P_i of the UMA system within the parallel region. Complete the table in the provided answer sheet which represents the first sequence of actions (from top to bottom of the table) executed by the three threads in the parallel region for the first iterations of the loop. State MC_i in each row has to be filled with the state of the cache line after the memory access in the action. For the rest of columns, you should specify the processor event ($PrRd_i$, $PrWr_i$), if there is cache hit or miss, the bus command ($BusRd_i$, $BusRdX_i$, $BusUpgr_i$), and the flush transaction ($Flush_i$), where i is the number of the processor where it is generated. Note: Observe that if you want to leave a cell empty, you can write "-".

   Solution:

Parallel Region Execution							
Action	CPU event	Miss/Hit	Bus command	Flush?	State MC_0	State MC_1	State MC_2
P_0 read DB[0]	PrRd_0	miss	BusRd_0	-	S	-	-
P_0 read count[0]	PrRd_0	hit	-	-	M	-	-
P_1 read DB[1]	PrRd_1	miss	BusRd_1	-	S	S	-
P_1 read count[1]	PrRd_1	miss	BusRd_1	Flush_0	M->S	S	-
P_0 write count[0]	PrWr_0	hit	BusUpgr_0	-	S->M	S->I	-
P_1 write count[1]	PrWr_1	miss	BusRdX_1	Flush_0	M->I	I->M	-
P_2 read DB[2]	PrRd_2	miss	BusRd_2	-	S	S	S
P_2 read count[2]	PrRd_2	miss	BusRd_2	Flush_1	I	M->S	S
P_2 write count[2]	PrWr_2	hit	BusUpgr_2	-	I	S->I	S->M
P_0 read DB[3]	PrRd_0	hit	-	-	S	S	S
P_1 read DB[4]	PrRd_1	hit	-	-	S	S	S

3. Assuming the amount of data we are accessing in the program, a system with a main memory of 32 GBytes and 32 Kbytes of private cache memory per processor, what is the total number of bits that an UMA system would use to keep the cache coherence per processor and the overall system? Would this number of bits change if we change the protocol from `MSI` to `MESI`?

   Solution:

   Per processor/cache:
   $32kbytes \times \frac{1line}{128bytes} \times \frac{2bits}{1line} \rightarrow \frac{2^{15} \times 2}{2^7} bits \rightarrow 2^9 bits \rightarrow 512 bits$

   Overall:
   3×512 bits devoted to coherence

   The number of bits would not change because we can still use 2 bits to represent 4 states (M, E, S, I) as when we have 3 states (M, S, I).

Problem 3 (2.5 points)

Given the following sequential program that computes the number of times the value stored in variable `element` appears in a vector of lists `data`:

```
#define NUM_ELEMS 10000
typedef struct list {
    int elem;
    struct list * next;
} list; // the basic component of a list

list * data[NUM_ELEMS]; // vector of lists, each list with varying number of elements
int element, count = 0; // value to search within data and number of times it appears

// function that returns the number of times element appears in theList
int list_search(list * theList, int element);

void main() {
    ...
    for (int entry = 0; entry < NUM_ELEMS; entry++) {
         int tmp = list_search(data[entry], element);
         count += tmp;
    }
    ...
}
```

Each of the lists may have a different number of elements, so when parallelising the program one should take care of load balancing. A parallel version for the sequential program above is also available, in which the original `for` loop has been substituted by a parallel region that assigns individual iterations to explicit tasks in such a way that tasks are generated under certain circumstances. One new shared variable `active_tasks` and two functions to operate it have also been added:

```
...
int active_tasks = 0;
// Functions to atomically add or subtract 1 to memory location pointed by address.
// The operation is saturated to the max or min value (i.e. the result can not be
// greater than max and smaller than min, respectively). They return value in memory
// location before operation
int atomic_inc(int *address, int max);
int atomic_dec(int *address, int min);

void main() {
    #pragma omp parallel
    #pragma omp single
    {
```

```
        int workers = omp_get_num_threads() - 1; // one thread focuses on
                                                 // task creation, the rest execute tasks
        for (int entry = 0; entry < NUM_ELEMS; entry++) {
            while (atomic_inc(&active_tasks, workers) == workers);
            #pragma omp task depend(inout: count)
                {
                int tmp = list_search(data[entry], element);
                count += tmp;
                atomic_dec(&active_tasks, 0);
                }
        }
    }
}
```

After compiling the parallel program and executing it with P processors (with $P > 1$) we detect that the program **is not achieving any speed–up**, although it produces a correct result.

1. Assuming the functionality for functions `atomic_inc` and `atomic_dec` explained in the code itself, what are these two functions used for in the program? Which is the number of implicit and explicit tasks that are generated during the execution of the program?

   Solution:

   The program generates P implicit tasks, one for each thread in the `parallel` construct. Only one of them (the one entering in `single`) is in charge of generating the explicit tasks: one explicit task is generated for each iteration of the for loop, so in total `NUM_ELEMS` explicit tasks. Functions `int atomic_inc(int *address, int max)` and `int atomic_dec(int *address, int min)` are used to control the number of tasks generated (kind of cut-off mechanism), in such a way that no more than $P - 1$ explicit tasks are simultaneously pending to execute or executing.

2. In the parallel version provided above, how many of these explicit tasks can be simultaneously executing? Rewrite the program above making the minimum appropriate changes in order to increase this number and, as a consequence, achieve a much better speed–up. Make sure the program generates the correct result. After these minimal changes, which can be the maximum number of explicit tasks that could be simultaneously executing?

   Solution:

   The `depend(inout:count)` clause used in `task` forces explicit tasks to execute sequentially, in the order they are created. This is not the most appropriate way to guarantee the race condition in this program when updating variable `count`; instead the use of `atomic` would enable the parallelism in the execution of multiple invocations to `list_search` while guaranteeing the correct update of variable `count`. With this change, a maximum of $P - 1$ tasks could be executing simultaneously.

   ```
   #pragma omp task // shared(count) and firstprivate(entry) by default
       {
       int tmp = list_search(data[entry], element);
       #pragma omp atomic
       count += tmp;
       atomic_dec(&active_tasks, 0);
       }
   ```

   Alternatively, a solution based on task reductions would also be valid:

   ```
   #pragma omp taskgroup task_reduction(+: count)
   for (int entry = 0; entry < NUM_ELEMS; entry++) {
       while (atomic_inc(&active_tasks, workers) == workers);
   ```

```
            #pragma omp task in_reduction(+: count)
                {
                int tmp = list_search(data[entry], element);
                count += tmp;
                atomic_dec(&active_tasks, 0);
                }
        }
```

3. Do an implementation for function `atomic_inc` making use of load–linked (`ll`) and store–conditional (`sc`) operations, defined as follows:

```
int ll(int *address); // returns the value stored in address
int sc(int *address, int value); // stores value in address if atomicity with ll
                                 // has been accomplished, returning true (1);
                                 // retuns false (0) otherwise
```

Solution:

A possible solution could be:

```
int atomic_inc(int *address, int max) {
    int tmp = ll(address);
    while ((tmp < max) && !sc(address, tmp+1)) tmp = ll(address);
    return(tmp);
}
```

In this solution, if the value read from memory `address` is equal to `max`, then the while loop finishes, returning the value just read. If the value is smaller than `max`, then a `sc` of the incremented value on the same memory `address` is attempted; if it succeeds the while loop is finished, again returnin the original value read from memory. If `sc` fails, then the new value from memory `address` is read again.

An equivalent code written in a different (more explicit) way would be:

```
int atomic_inc(int *address, int max) {
    int retsc = 0;
    do {
        int tmp = ll(address);
        if (tmp == max) return(tmp);
        retsc = sc(address, tmp+1);
    } while (retsc == 0);
    return(tmp);
}
```

Another version that always writes to memory `address` would be:

```
int atomic_inc(int *address, int max) {
    int tmp, newvalue;
    do {
        tmp = newvalue = ll(address);
        if (tmp < max) newvalue++;
    } while (!sc(address, newvalue));
    return(tmp);
}
```

Observe that in this case if the value in memory `address` is already `max`, the same value `max` is unnecessarily written again to memory `address`.

4. Finally we found a recursive version alternative to the original sequential code:

```
...
void rec_list_search(list ** data, int size, int element) {
    int tmp = list_search(data[0], element);
    count += tmp;
    if (size > 1)
        rec_list_search(data+1, size-1, element);
}

void main() {
    rec_list_search(&data[0], NUM_ELEMS, element);
}
```

Write a parallel version for it that implements a *recursive* **leaf** *task decomposition*. This new version should not implement any cut-off mechanism to restrict the number of tasks that are generated.

Solution:

```
void rec_list_search(list ** data, int size, int element) {
    #pragma omp task shared(count)
    {
        int tmp = list_search(data[0], element);
        #pragma omp atomic
        count += tmp;
    }
    if (size > 1)
        rec_list_search(data+1, size-1, element);
}

void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    rec_list_search(&data[0], NUM_ELEMS, element);
    ...
}
```

Problem 4 (2.5 points)

Given the following code fragment computing matrix `m[N][N]`, with `N` much larger than the number of processors P to be used in the parallel execution, and with `N` not necessarily a multiple of P:

```
telem m[N][N];

for (int i=1; i<N-1; i++)
   for (int j=0; j<N; j++)
       m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
```

We ask you to:

1. Decide the most appropriate *geometric data decomposition strategy* for matrix `m` and write a parallel version of the code above using OpenMP that corresponds to it. Your solution should a) minimize the synchronization overhead among implicit tasks and b) guarantee that the load unbalance is limited to `N` elements (i.e. the number of elements in a row or column of the matrix).

   Solution:

   There are dependencies between iterations of the `for-i` loop: a RAW dependency given by $m[i][j]$ to $m[i-1][j]$ and a WAR dependency given by $m[i][j]$ to $m[i][j+1]$ for i in $[2..N-2]$. There are

no dependencies between iterations of the `for-j` loop, so we can fully parallelize it, with no need for synchronization.

Consequently, it's advisable to choose a *Geometric Block Data decomposition* by columns. We must take care of adjusting the block size (number of columns of N elements each) so that there is at most 1 column of difference in size between the different blocks. In order to apply the owner compute rule, the distribution of the matrix `m` is by columns with the resulting block size. A *Geometric Cyclic Data decomposition* by columns would also be correct, as the amount of work would be balanced automatically, without any extra calculation.

```
telem m[N][N];
...
#pragma omp parallel num_threads(P)
{
  int myid = omp_get_thread_num();
  int BS = N / P;
  int start = myid * BS;
  int end = start + BS;
  int mod = N % P;
  if (mod > 0) {
     if (myid < mod)
        start += myid;
        end = start + BS + 1;
     }
     else {
         start += mod;
         end += start + BS;
     }
  }
  for (int i=1; i<N-1; i++) {
    for (int j=start; j<end; j++) {
        m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
    }
  }
}
```

2. Now consider that the program is going to be executed on a parallel machine in which memory lines are 128 bytes long. The allocation in memory for matrix `m` is aligned to the start of a memory line and `sizeof(telem)` is 8 bytes (`N` is not necessarily multiple of `sizeof(telem)`). Decide the most appropriate *geometric data decomposition strategy* in this case and re-write the previous OpenMP parallel code and, if necessary, the definition of matrix `m`. Your solution should a) maximize parallelism among implicit tasks; and b) maximize data locality and reduce coherence traffic.

   Solution:

   In order to preserve data locality we must take care of the memory line size when accessing elements of the matrix. In this sense, given that `sizeof(telem) = 8 bytes`, a memory line of 128 bytes can hold up to 16 elements. For this reason we will consider only block sizes with values multiple of 16. Consequently we apply a *Geometric Block-cyclic data decomposition* by columns, with block size = 16. Given that N » P we can assume a load unbalance of one block (N x 16 elements). In addition, in case N is not multiple of `sizeof(telem)`, we must add padding at the end of the row to avoid generating false sharing with the beginning of the next row. In order to apply the owner compute rule, the distribution of the matrix `m` is by columns with the resulting block size.

```
#define MEMORYLINESIZE 128
#define BS MEMORYLINESIZE/sizeof(telem)
#define X (N%BS==0? 0: (BS - (N % BS)))
```

```
telem m[N][N+X];

int BS = MEMORYLINESIZE / sizeof(telem);
...
#pragma omp parallel num_threads(P)
{
  int myid = omp_get_thread_num();
  int start = myid * BS;
  int end = N;

  for (int i=1; i<N-1; i++) {
    for (int jj=start; jj<end; jj+=BS*P)
        for (int j=jj; j<j+BS; j++)
            m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
}
```

Student name: .....

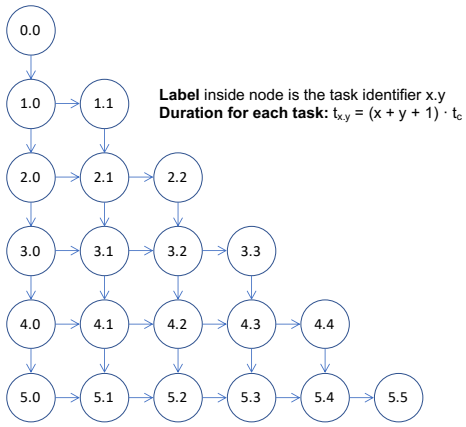
Answer sheet for **Question 2.2**.

Parallel Region Execution							
Action	CPU event	Cache Miss/Hit	Bus command	Flush?	State MC_0	State MC_1	State MC_2
P_0 reads DB[0]							
P_0 reads count[0]							
P_1 reads DB[1]							
P_1 reads count[1]							
P_0 writes count[0]							
P_1 writes count[1]							
P_2 reads DB[2]							
P_2 reads count[2]							
P_2 writes count[2]							
P_0 reads DB[3]							
P_1 reads DB[4]							

PAR – Final Exam – Course 2021/22-Q2

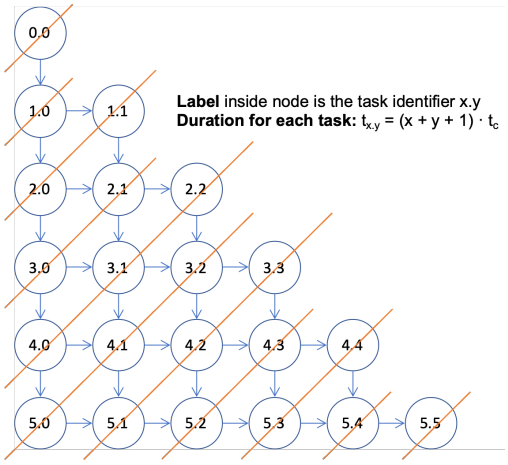
June 20^{th}, 2022

Problem 1 (1.5 points) Given the following Task Dependence Graph associated to the task decomposition applied to a certain program:

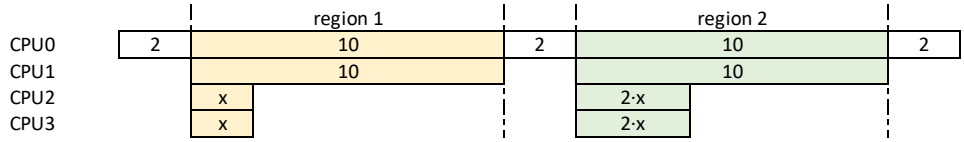


The duration, in time units, of each `compute_task` depends on the value of the row (`x`) and column (`y`) where it is placed (label `x.y`): $t_{x.y} = (x + y + 1) \times t_c$. **We ask you** to compute the values for T_1, T_∞, $Parallelism$ and P_{min}.

Solution: $T_1 = 126 \times t_c$ (sum of the cost of all nodes in the TDG above), $T_\infty = 66 \times t_c$ (defined by the critical path in the TDG, including all nodes in the column on the left and all nodes in the row at the bottom of the TDG above) and $Parallelism = T_1 \div T_\infty = 126 \div 66 = 1.\widehat{90}$. The minimum number of processors to achieve it is $P_{min} = 3$, since we need enough simultaneous processors to execute the widest anti–diagonal in the wavefront execution of the TDG, as shown in the picture below.



Problem 2 (1.5 points) The following diagram plots the timeline for the execution of a parallel application, with two parallel regions, on 4 processors (time evolves from left to right):



Each box represents a burst executed on a processor and the number inside its execution time. There are 3 sequential regions with constant execution time of 2 time units. The execution time is unknown for two of the bursts in $region1$ and in $region2$, both affected by the value x (the timeline shows the case for which

$(2 \times x) < 10$).

- Obtain all the possible values for value x that would lead to an speed-up $S_4 = 2.5$ when the application is executed on 4 processors. Observe that for values $0 < x \leq 5$, $5 < x \leq 10$ and $x > 10$ the two parallel regions have a different contribution in the timeline.

  Solution: In all cases, $T_1 = 2 + (10 + 10 + x + x) + 2 + (10 + 10 + 2 \times x + 2 \times x) + 2 = 46 + 6 \times x$.

  - For x smaller than 5, the execution time for both parallel regions is 10 units, so $T_4 = 2 + 10 + 2 + 10 + 2 = 26$. Therefore from $S_4 = T_1/T_4 = 2.5$ we get $x = 19/6 = 3.16$.
  - For x larger than 5 but smaller than 10, the execution time for the first parallel region is still 10 but for the second one is $2 \times x$. Therefore $T_4 = 16 + 2 \times x$. From $S_4 = T_1/T_4 = 2.5$ it is not possible to get a positive value for x.
  - Finally, for x larger than 10, both parallel regions are dominated by the value of x, being the execution time $T_4 = 6 + 3 \times x$. Again from $S_4 = T_1/T_4 = 2.5$ we obtain $x = 31/1.5 = 20.66$.

- For the values of x in the previous question, obtain the value for $S_{p \to \infty}$, assuming that parallel regions ideally scale with the number of processors.

  Solution: For the two cases above, $T_{p \to \infty} = 6$. Therefore:

  - For $x = 19/6 = 3.16$ we get $S_{p \to \infty} = 65/6 = 10.83$.
  - Similarly, for $x = 31/1.5 = 20.66$ we get $S_{p \to \infty} = 170/6 = 28.33$.

Problem 3 (3.0 points) Given the following data structures that models a graph in which each node can have up to 4 neighbour nodes. Vector `g` holds the information for `N` nodes; for each node the `telem` struct stores the weight of the node `w`, an integer to identify each of the 4 possible neighbours (`north`, `east`, `west` and `south`) and a field that is used to traverse the graph (`visited`) .

```
#define N ... /* large value */

typedef struct {
        int w;
        int north, east, west, south; // -1 value to indicate no neighbour
        char visited;
} telem;
telem g[N];
```

The following code processes all the nodes that are reachable from a given node (0 in the invocation from main):

```
int compute (int label, int weight); // heavy computation, does not access the graph

int traverse_reachable (int label) {
        int ret=0, ret1, ret2, ret3, ret4;
        if (label >= 0) {
            if (!g[label].visited) {
                g[label].visited=1;
                ret1 = traverse_reachable (g[label].north);
                ret2 = traverse_reachable (g[label].east);
                ret3 = traverse_reachable (g[label].west);
                ret4 = traverse_reachable (g[label].south);
                ret = compute(label, g[label].w) + ret1 + ret2 + ret3 + ret4;
            }
        }
        return ret;
}

int main() {
```

```
    ...
    int ret = traverse_reachable (0);
    ...
}
```

We ask you to write an *OpenMP* parallel version of the previous code using a *recursive tree task decomposition* strategy. The implementation should maximize parallelism and minimize the possible synchronization overheads. The implementation must also include a task generation control mechanism based on the recursion level. Use *MAX_ DEPTH* as the value for the maximum recursion level for which tasks must be generated.

Solution:

```
typedef struct {
        int w;
        int north, east, west, south;
        char visited;
        omp_lock_t lock;    // new field to protect the access to the node
} telem;
telem g[N];

int traverse_reachable (int label, int d) { // new argument d to control recursion level
        int ret=0, ret1, ret2, ret3, ret4, tmp;

        if (label >= 0) {
             if (!g[label].visited) {
                omp_set_lock (&g[label].lock);
                if (!g[label].visited) {
                    g[label].visited=1;
                    omp_unset_lock (&g[label].lock);

                    if (!omp_in_final()) {
                        #pragma omp task shared(ret1) final (d>= MAX_DEPTH)
                        ret1 = traverse_reachable (g[label].north, d+1);
                        #pragma omp task shared(ret2) final (d>= MAX_DEPTH)
                        ret2 = traverse_reachable (g[label].east, d+1);
                        #pragma omp task shared(ret3) final (d>= MAX_DEPTH)
                        ret3 = traverse_reachable (g[label].west, d+1);
                        #pragma omp task shared(ret4) final (d>= MAX_DEPTH)
                        ret4 = traverse_reachable (g[label].south, d+1);
                        tmp = compute(label, g[label].w);
                        #pragma omp taskwait     // compute does not need to wait for the graph t
                    } else {
                        ret1 = traverse_reachable (g[label].north, d+1);
                        ret2 = traverse_reachable (g[label].east, d+1);
                        ret3 = traverse_reachable (g[label].west, d+1);
                        ret4 = traverse_reachable (g[label].south, d+1);
                        int tmp = compute(label, g[label].w);
                    }
                    ret = tmp + ret1 + ret2 + ret3 + ret4;
                } else {
                    omp_unset_lock (&g[label].lock);
                }
            }
        }
        return ret;
}

int main() {
    ...
    #pragma omp parallel
```

```
    #pragma omp single
    int ret = traverse_reachable (0, 0);
    ...
}
```

Problem 4 (4.0 points) Given the following sequential code:

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

Assume parameters `x` and `y` point to two completely disjoint vectors, first element of each vector aligned to a memory/cache line boundary. Cache line size is 64 bytes and sizeof(float) is 4 bytes. **We ask you to:**

1. Write a first OpenMP parallel version for the previous sequential function that obeys to an *input block geometric data decomposition* strategy, so that each thread accesses to a single block of consecutive elements. The *block geometric decomposition* should minimise the possible load unbalance that occurs when the size of the vectors `n` is not a multiple value of the number of threads. Assume that `n` is large compared to the number of threads.

   Solution:

   ```
   void saxpy(int n, float a, float * x, float * y) {
     #pragma omp parallel
     {
        int nt;
        int my_id;
        int red;
        int i_start, i_end, n_elems;
        nt    = omp_get_num_threads();
        my_id = omp_get_thread_num();
        n_elems = n/nt;
        red     = n%nt;
        i_start = my_id*n_elems;
        i_start = i_start + (my_id<red)?my_id:red;
        i_end   = i_start + n_elems + (my_id<red);
        for (int i = i_start; i < i_end; ++i)
            y[i] = a*x[i] + y[i];
     }
   }
   ```

2. Write a new parallel version that obeys to an *input/output cyclic geometric data decomposition* strategy. Has the load balance improved with respect to the previous version?

   Solution:

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

   Both implementantions block and cyclic geometric data decomposition have the same load unbalance: maximum 1 element.

3. Assuming the *input/output cyclic geometric data decomposition strategy* in the previous implementation, and that 1) the parallel system is composed of 2 NUMA nodes, each with a single processor and private cache; 2)

the parallel program is executed with 2 threads (i.e. *thread* i in NUMA node i); 3) the operating system makes use of "first touch" at page level to decide the allocation of memory addresses to NUMA nodes, with one line per memory page; 4) main memory, directories and private caches are empty when the function above starts its execution; and 5) the execution of the iterations assigned to the two threads are interleaved in time, as shown in the two leftmost columns in the table in the answer sheet for the first 4 iterations of the loop. Data coherence across NUMA nodes is provided by a write-invalidate MSU directory-based system. Complete this table with the information of the directory entries where elements of vector `x` and `y` are stored.

4. Write a final parallel version that obeys to an *output block-cyclic geometric data decomposition* strategy, so that the overhead associated with the coherence protocol in a NUMA multiprocessor architecture is minimised, i.e. exploiting the data locality of both input and output data. Is the load balance better/worse than the one achieved in the parallel versions in question 1 and 2?

   Solution:

   We set BS to the number of float elements that fits in a cache/memory line. In this way we avoid extra cache misses accessing both vector `x` and `y` and false sharing accessing `y`.

```
#define BS (64/4)
void saxpy(int n, float a, float * x, float * y) {
  #pragma omp parallel
  {
    int nt;
    int my_id;
    nt    = omp_get_num_threads();
    my_id = omp_get_thread_num();
    for (int ii = my_id*BS; ii < n; ii+=nt*BS)
      for (int i = ii; i < max(n,ii+BS); i++)
         y[i] = a*x[i] + y[i];
  }
}
```

   Block cyclic geometric data decomposition may have `BS` elements of load unbalance. Therefore, it is worse than before.

Student name: .....

Answer sheet for **Problem 4**.

Time	thread	Loop iteration i - vector access	Home NUMA node	Directory entry (sharers bit list)	Directory entry (status)
0	1	1 - read x[1]			
		1 - read y[1]			
		1 - write y[1]			
1	0	0 - read x[0]			
		0 - read y[0]			
		0 - write y[0]			
2	0	2 - read x[2]			
		2 - read y[2]			
		2 - write y[2]			
3	1	3 - read x[3]			
		3 - read y[3]			
		3 - write y[3]			

Solution for **Problem 4**.

Time	thread	Loop iteration i - vector access	Home NUMA node	Directory entry (sharers bit list)	Directory entry (status bits)
0	1	1 - read x[1]	1	10	S
		1 - read y[1]	1	10	S
		1 - write y[1]	1	10	M
1	0	0 - read x[0]	1	11	S
		0 - read y[0]	1	11	S
		0 - write y[0]	1	01	M
2	0	2 - read x[2]	1	11	S
		2 - read y[2]	1	01	M
		2 - write y[2]	1	01	M
3	1	3 - read x[3]	1	11	S
		3 - read y[3]	1	11	S
		3 - write y[3]	1	10	M

PAR – Final Exam – Course 2022/23-Q1

January 18^{th}, 2023

Problem 1 (2.5 points)

Given the following code with *tareador* task annotations:

```
#define p ...   // Number of processors
#define NR ...  // Number of rows
#define NC ...  // Number of columns

int M[NR][NC];
int BS = NR/p;  // Assume p divides NR exactly

for (int ii=0; ii<NR; ii+=BS) {                    // Matrix initialization
    tareador_start_task ("init");
    for (int i=ii; i<ii+BS; i++)
       for (int j=0; j<NC; j++)
         M[i][j] = init(i,j);                      // <-- Cost ti
    tareador_end_task ("init");
}

tareador_start_task ("comp1");                     // Begin computations
for (int i=0; i<BS; i++)
    for (int j=0; j<NC; j++)
        M[i][j] = comp1(M[i][j]);                  // <-- Cost tb
tareador_end_task ("comp1");

for (int ii=BS; ii<NR; ii+=BS) {                   // Final computations
    tareador_start_task ("comp2");
    for (int i=ii, int i0=0;
         i<ii+BS;
         i++, i0++)
       for (int j=0; j<NC; j++)
         M[i][j] = comp2(M[i0][j], M[i][j]);       // <-- Cost tf
    tareador_end_task ("comp2");
}
```

Let us assume the data sharing model explained in class based on a distributed memory architecture in which we consider that local memory accesses have no cost, but an access to data in different processors introduces a data–sharing overhead: the access time to remote data is determined by $t_{comm} = t_s + m \times t_w$, being t_s and t_w the "start-up" and sending time of an element, respectively, and being m the size of the message. Also, according to the data sharing model: at any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor.

Assume that the number of processors p divides the number of rows `NR` exactly; routines `init`, `comp1` and `comp2` do not modify any memory position; the execution time of one iteration of the body of the most internal loops is t_i, t_b and t_f respectively; tasks are scheduled to processes following the *owner-computes rule* so that a task will be executed by the processor who owns the memory that holds the output of that task; the matrix is already distributed in the memory of each processor when the computation starts; the resulting matrix remains distributed and there is no final communication to a single processor; the strategy used for decomposing matrix `M` follows a *block row distribution*: the matrix `M` is distributed so that each processor has $\frac{NR}{p}$ consecutive rows. **We ask** you to:

1. Draw a Task Dependence Graph (TDG) for the case where $p = 4$;
2. Draw a timeline for the execution with $p = 4$;
3. Provide a general expression that determines the parallel execution time on p processors (T_p): express T_p as a function of p, `NR`, `NC`, t_i, t_b, t_f, t_s and t_w.

Solution:

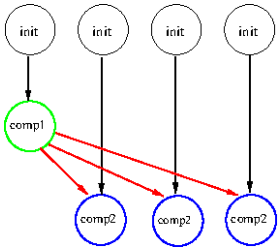


Figure 1: TDG

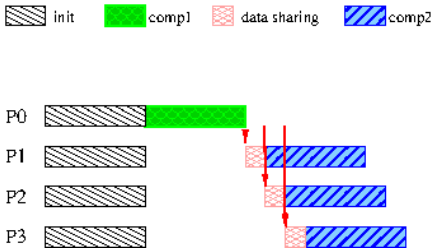


Figure 2: Timeline

$T_p = t_{data\_sharing} + t_{calc}$

$t_{data\_sharing} = (t_s + \frac{NR}{P} \times NC \times t_w) \times (P-1)$

$t_{calc} = \frac{NR}{P} \times NC \times t_i + \frac{NR}{P} \times NC \times t_b + \frac{NR}{P} \times NC \times t_f = \frac{NR}{P} \times NC \times (t_i + t_b + t_f)$

Problem 2 (2.5 points)

Assume a NUMA (non-uniform memory architecture) multiprocessor system composed of 4 NUMA-nodes ($node_{0:3}$), each node with 8 GBytes of main memory and 4 cores ($core_{0:3}$). Each core has only one level of cache memory of 2 MBytes (with cache lines of 32Bytes). The system includes all the necessary mechanisms (seen in class) to keep the memory coherence inside a NUMA-node and between NUMA-nodes.

1. (0.5 points) In order to support a write-invalidate MSI cache-coherence protocol within each NUMA-node, how many additional bits should be used in each cache line, and what are their role and possible values? How many bits in total per NUMA-node (only within each NUMA-node) are used for that purpose?

   Solution:

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

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

   The number of cache lines is calculated dividing the memory in a cache (2MB) by the cache line size (32B). Therefore: Number of cache lines $= \frac{2MB}{32B} = 2^{16} = 64K$ entries

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

   Thus, the total number of bits amounts to:

   $\frac{2MB}{32B}$ entries/cache $\times 2$ bits/entry $\times 4$ caches/NUMAnode $= 2^{19}$ bits $= 512K$ bits/NUMAnode

2. (0.5 points) In order to support a directory-based write-invalidate MSU cache-coherence protocol between NUMA-nodes, how many bits should be used in the directory for each line in main memory, and what are their role and possible values? How many bits in total per NUMA-node (directory) are used for that purpose?

   Solution:

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

Bits per directory entry:

- Presence bits (nodes currently having the line), 1 bit per NUMA-node: i.e. 4 bits
- State bits (to track the state of memory lines): 2 bits
  Those 2 bits can code up to four possible states:
  - U uncached, not valid in any cache
  - S shared (two or more nodes may have clean copies)
  - M modified (dirty)
  - Note that a fourth state can be coded with 2 bits. However, it is not necessary for the MSI protocol.

Total: 6 bits per directory entry.

The number of entries in the directory structure per NUMA-node is calculated dividing the memory in a NUMA-node (8GB) by the memoy line size (32B). Therefore:

Number of Directory Entries $= \frac{8GB}{32B} = 2^{28} = 256$ Mega entries

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

$\frac{8GB}{32B}$ entries/NUMAnode $\times 6$ bits/entry $= 3 \times 2^{29} = 1536$ Mega bits/NUMAnode

3. (1.5 points) Consider the following parallel program executed on only two cores (processors) inside $node_0$ of the previous multiprocessor system:

```
...
double A[M][N];
...
```

Core 0: Update even-numbered rows

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

Core 1: Update odd-numbered rows

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

Assume that matrix `A` is stored in main memory of $node_0$ (without copies of any of its elements in the caches of the NUMAnodes), that each element of matrix `A` is 8 Bytes and that the first element of `A` is aligned on a cache line boundary, $N = 2$ and M is a value multiple of 2.

(a) What would be the maximum number of invalidations that would be sent through the bus expressed as a function of M? What is causing such a memory coherence problem?

Solution:

For $N = 2$, every two rows fall in the same cache line, causing up to 3 invalidations due to false sharing for every pair of lines. Thus, the number of invalidations is $3 \times M/2$.

(b) Modify the declaration of matrix A so that you do not have to change the code and avoid the previous coherence problem?

Solution:

We add enough padding to the second dimension of matrix A to make the size (in bytes) of a row of A equal or multiple of the number of bytes of a cache line.

```
...
#define PADDING ((CACHE_LINE_SIZE - (N*sizeof(double) % CACHE_LINE_SIZE))/sizeof(double))
double A[M][N+PADDING];
...
```

Problem 3 (2.5 points)
Given the following code:

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

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

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

Write two different OpenMP parallel implementations of function `histogram` using the strategies presented below. You can modify the sequential code, add local variables and use any OpenMP pragma (except `omp for` worksharing-loop construct) and function you may need. Both implementations should minimize the use of synchronizations and load unbalance between threads during the processing of vector S.

1. (1 point) Cyclic Data Decomposition of the Output vector `index`.
   Solution:

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

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

   #pragma omp parallel private(i, tmp)
   {
      int id = omp_get_thread_num();
      int num_threads = omp_get_num_threads();

      for (i=0;i<n;i++) {
        tmp = S[i]%SIZE_INDEX;
        if ((tmp%num_threads)==myid)
          index[tmp]++;
    }
   }

   }

   ...
   ```

2. (1.5 points) Block Data Decomposition of the Input vector `S`.
   Solution:

   ```
   #define SIZE_INDEX 256
   ```

```
#define N 1024*1024*1024
...

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

 #pragma omp parallel private(i, tmp)
 {
  int myid        = omp_get_thread_num();
  int num_threads = omp_get_num_threads();
  int i_start     = myid * (N/num_threads);
  int i_end       = (myid+1) * (N/num_threads);
  int res         = N%num_threads;
  if (res)
  {
     i_start = i_start + (myid<res)?myid:res);
     i_end   = i_end   +   (myid<res)?myid+1:res);
  }

  unsigned int local_index[SIZE_INDEX];
  for (i=0;i<SIZE_INDEX;i++)
   local_index[i]=0;

  for (i=i_start;i<i_end;i++) {
     tmp = S[i]%SIZE_INDEX;
     local_index[tmp]++;
  }

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

 }
}
```

Problem 4 (2.5 points)

We ask you to write two additional parallel OpenMP implementations of the code that computes the `histogram`, this time using *task decomposition* strategies:

1. (1 point) Write an efficient OpenMP parallel version of the histogram program in Problem 3 using *explicit tasks*.

   Solution: A possible implementation:

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

 #pragma omp parallel
 #pragma omp single
 #pragma omp taskloop private(tmp)
 for (i=0; i<n; i++) {
   tmp = S[i]%SIZE_INDEX;
   #pragma omp atomic
   index[tmp]++;
 }
}
```

2. (1.5 points) Write a *recursive tree task decompostion* with *cutoff* based on the depth of the recursivity for the following code:

```
#define SIZE_INDEX 256
#define N 1024*1024*1024
#define BASE_SIZE 512
#define CUTOFF 3

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

void histogram_rec(unsigned int *S, int n, unsigned int *index) {
  unsigned int n2=n/2;

  if (n > BASE_SIZE) {
    histogram_rec(S, n2, index);
    histogram_rec(&S[n2], n-n2, index);
  } else {
      histogram(S, n, index);
 }
}

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

Solution: A possible implementation:

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

void histogram_rec(unsigned int *S, int n, unsigned int *index, int depth) {
  unsigned int i, tmp, n2;

  if (n > BASE_SIZE) {
     n2 = n/2;
     if ( !omp_in_final() ) {
       #pragma omp task final(depth>=CUTOFF)
       histogram_rec(S, n2, index, depth+1);
       #pragma omp task final(depth>=CUTOFF)
       histogram_rec(&S[n2], n-n2, index, depth+1);
     }
     else {
       histogram_rec(S, n2, index, depth+1);
       histogram_rec(&S[n2], n-n2, index, depth+1);
     }
  else {
      histogram(S, n, index);
 }
}

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