PAR
Selection of Exams (with Solutions)

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

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

Course 2018-19 (Fall semester)

Contents

Part I

1^{st} In-term Exams

PAR – 1^{st} In-Term Exam – Course 2016/17-Q1
November 2^{nd}, 2016

Problem 1 Consider the following main program that calculates the power of any number using an iterative function and a recursive function.

```
#include <stdio.h>
#define MIN_POWER 10

long int getPower_iterative(int b, int p) {
   long int result=1;
   for(int i=0; i<p; i++)
      result = result * b;
   return result;
}

long int getPower_recursive(int b,int p) {
    long int result;
    if(p<MIN_POWER)
       result = getPower_iterative(b,p);
    else
       result = getPower_recursive(b ,p/2) *
                getPower_recursive(b ,p-p/2);

    return result;
}

int main() {
    int base, power;
    long int result;
    ...
    result = getPower_recursive(base, power);
    ...
    result = getPower_iterative(base, power);
    ...
    return 0;
}
```

Note: Each question is independent. **We ask** you:

1. Write two significative different OpenMP versions that implement an iterative task decomposition for `getPower_iterative`, avoiding any synchronization inside the loop and the appearance of false sharing.

   Solution 1:

   ```
   long int getPower_iterative(int b, int p) {
      long int result=1;
      #pragma omp parallel for reduction(*:result)
      for(int i=0; i<p; i++)
         result = result * b;
      return result;
   }
   ```

Solution 2:

```
long int getPower_iterative(int b, int p) {
   long int result=1;
   long int result_vect[MAX_NUM_THREADS][CACHE_SIZE_LINE/sizeof(long int)];

   #pragma omp parallel
   {
     int my_id=omp_get_num_thread();
     result_vect[my_id][0]=1;
     #pragma omp for
       for(int i=0; i<p; i++)
          result_vect[my_id][0] = result_vect[my_id][0] * b;

     #pragma omp atomic
        result *= result_vect[my_id][0];
   }
   return result;
}
```

2. Write two different OpenMP versions that implement a recursive (divide and conquer) task decomposition strategy for the recursive function (`getPower_recursive`). The two implementations should ideally exploit the parallelism in such a way that $T\infty \rightarrow \log(p)$ (if `MIN_POWER` was 1 and there wasn't any task creation parallel recursive control). The difference between the two versions is how they control task creation overheads:

   - *v*ersion 1: no more tasks are created once a specific parallel recursion depth (`MAX_DEPTH`) is reached.
   - *v*ersion 2: a task is created if and only if the total number created tasks so far is less than `MAX_TOTAL_CREATED_TASKS`.

   Solution version 1:

```
long int getPower_recursive(int b,int p, int depth) {
    long int result;
    long int result1;
    long int result2;
    if(p<MIN_POWER)
       result = getPower_iterative(b,p);
    else
    {
       #pragma omp task shared(result1) final(depth>=MAX_DEPTH) mergeable
       result1 = getPower_recursive(b ,p/2, depth+1);

       #pragma omp task shared(result2) final(depth>=MAX_DEPTH) mergeable
       result2 = getPower_recursive(b ,p-p/2, depth+1);

       #pragma omp taskwait
       result = result1 * result2;
    }
    return result;
}

int main() {
    int base, power;
    long int result;
    ...
    #pragma omp parallel
```

```
    #pragma omp single
    result = getPower_recursive(base, power,0);
    ...
    result = getPower_iterative(base, power);
    ...
    return 0;
}
```

Solution version 2:

Note: To simplify the code, we check (control) if it is possible or not to create the two tasks of a recursive level at the same time (test&test&set). Checking that before each possible task will be perfectly correct and more precise.

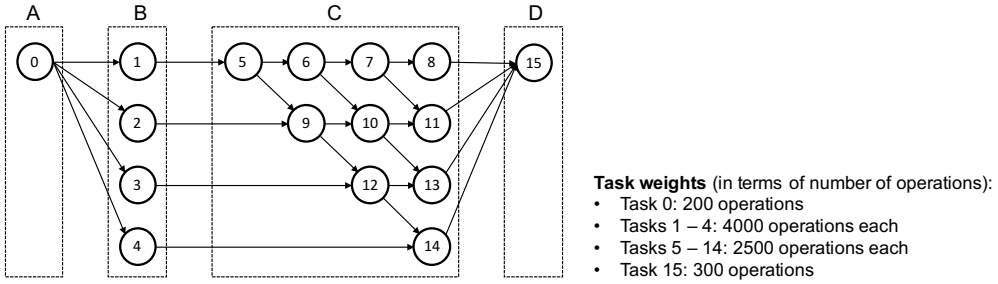
```
unsigned int n_tasks=0;

long int getPower_recursive(int b,int p) {
    long int result;
    long int result1;
    long int result2;
    unsigned int flag_create=0;

    if(p<MIN_POWER)
       result = getPower_iterative(b,p);
    else
    {
       if (n_tasks+2<MAX_TOTAL_CREATED_TASKS) // TEST
        #pragma omp critical
         if(n_tasks+2<MAX_TOTAL_CREATED_TASKS) // TEST&SET
         {
              n_tasks+=2;
              flag_create=1;
         }

       #pragma omp task shared(result1) final(!flag_create) mergeable
       result1 = getPower_recursive(b ,p/2);

       #pragma omp task shared(result2) final(!flag_create) mergeable
       result2 = getPower_recursive(b ,p-p/2);

       #pragma omp taskwait
       result = result1 * result2;
    }
    return result;
}

int main() {
    int base, power;
    long int result;
    ...
    #pragma omp parallel
    #pragma omp single
    result = getPower_recursive(base, power);
    ...
    result = getPower_iterative(base, power);
    ...
    return 0;
}
```

Problem 2 Consider a program with four code regions (A, B, C and D) executed one after the other in its initial sequential version. Code regions A and D can not be parallelised, i.e. they must be executed sequentially, performing 200 and 300 operations, respectively. On the contrary, code regions B and C can be parallelised, performing 16000 and 25000 operations, respectively.

1. Compute the parallel fraction φ of the program and, using Amdahl's Law, compute the ideal speed–up S_∞ when using infinite processors, assuming that both regions B and C ideally scale up to that number of processors.

   Solution: φ is the quotient between the time spent in tasks that belong to the parallel regions B and C divided by the total execution time, so in this case $\varphi = (4 \times 4000 + 10 \times 2500) \div (200 + 4 \times 4000 + 10 \times 2500 + 300) = 0.98$; from here and applying the expression of Amdalh Law we obtain $S_\infty = 1 \div (1 - \varphi) = 83$.

After applying task decomposition to regions B and C we realised that region B can be executed fully in parallel (i.e. no dependences among tasks) and region C has dependences among tasks. The following figure shows the task dependence graph that is obtained, in which we represented both sequential regions A and D as a single task each (tasks 0 and 15, respectively). The legend on the right of the graph indicates the computation weight for each of the tasks in the graph.



2. Compute T_1, T_∞ and Par (parallelism) for this task dependence graph. Is $Par \neq S_\infty$? If affirmative, give the reason or reasons that justifiy the difference.

   Solution: $T_1 = 200 + 4 \times 4000 + 10 \times 2500 + 300 = 41500$; T_∞ is determined by the critical path in the graph; in this graph there are several alternatives, for example {0, 1, 5, 9, 12, 14, 15}. $T_\infty = 200 + 4000 + 4 \times 2500 + 300 = 14500$; finally $Par = 41500 \div 14500 = 2.86$. $Par \neq S_\infty$ mainly because two reasons: the task decomposition does not contain enough tasks to feed ∞ processors and there are dependences among tasks.

3. If tasks were assigned to two processors as follows: P0={0, 1, 4, 5, 6, 7, 8, 14, 15} and P1={2, 3, 9, 10,11, 12, 13}, which would be the speed–up S_2 that could be achieved?

   Solution: The speed–up that would be achieved is the quotient between T_1 and T_2. With the proposed assignment, $T_2 = 200 + 2 \times 4000 + 5 \times 2500 + 300 = 21000$ determined by P0. Therefore $S_2 = 41500 \div 21000 = 1.97$. The first temporal diagram below shows the schedule of tasks to processors and their execution order.

4. If tasks were assigned to four processors as follows: P0={0, 1, 5, 6, 7, 8, 15}, P1={2, 9, 10,11}, P2={3, 12, 13} and P3={4, 14}, which would be the speed–up S_4 that could be achieved?

   Solution: The speed–up that would be achieved is the quotient between T_1 and T_4. With the proposed assignment, $T_4 = 200+4000+4\times2500+300 = 14500$ determined by P0. Therefore $S_4 = 41500 \div 14500 = 2.86$. The second temporal diagram below shows the schedule of tasks to processors and their execution order.

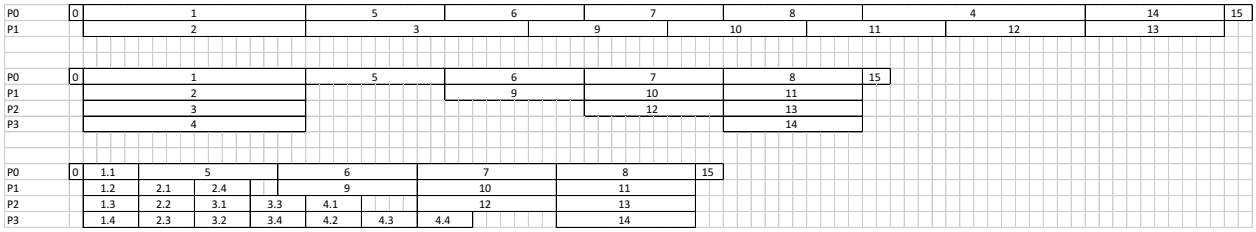
5. In order to improve the speed–up that is obtained with four processors, someone proposed us to further divide EACH ONE of the tasks in region B into four totally independent tasks. In other words, Task 1 to be replaced by Task 1.1, Task 1.2, Task 1.3 and Task 1.4, each one with 1000 operations and with Task 5 depending on these 4 new tasks; and the same for Task 2, Task 3 and Task 4. Is this suggestion

improving the potential parallelism Par in the program? Compute the new value for Par. Suggest the best mapping of tasks to processors and compute the achievable speed–up S_4.

Solution: Each new task in region B weights 1000 operations, reducing the weight of the new critical in the graph $\{0, 1.1, 5, 9, 12, 14, 15\}$ to $T_\infty = 200 + 1000 + 4 \times 2500 + 300 = 11500$; finally $Par = 41500 \div 11500 = 3,61$. A mapping of these tasks to processors could be:

```
P0 = { 0, 1.1, 5, 6, 7, 8, 15}
P1 = { 1.2, 2.1, 2.4, 9, 10, 11}
P2 = { 1.3, 2.2, 3.1, 3.3, 4.1, 12, 13}
P4 = { 1.4, 2.3, 3.2, 3.4, 4.2, 4.3, 4.4, 14}
```

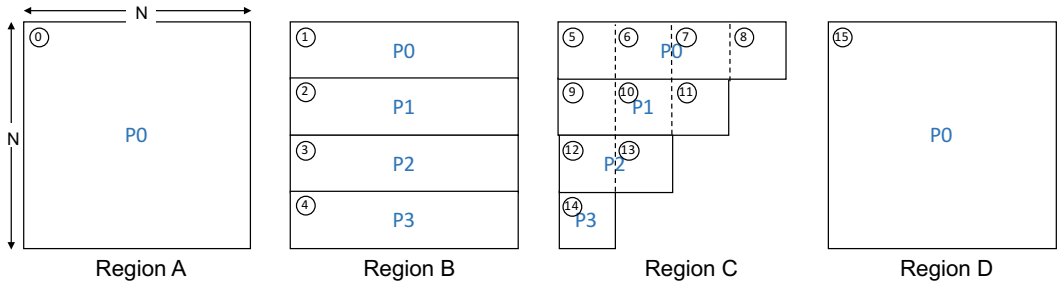
so that we compensate the lack of work in region C with work from B, without delaying in any case processor P0. This results in $T_4 = 200 + 1000 + 4 \times 2500 + 300 = 11500$ and speed–up of $S_4 = 41500 \div 11500 = 3.61$ determined by P0. The third temporal diagram below shows the schedule of tasks to processors and their execution order.



Problem 3 As a continuation of the previous problem, assume that the four regions above compute A SINGLE two–dimensional matrix of N by N elements according to the following code skeletons:

```
Region A
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      init(A[i,j]);

Region B
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      A[i,j] = comp1(A[i,j]);

Region C
for (i=1; i<N; i++)
   for (j=1; j<(N-i); j++)
      A[i,j] += comp2(A[i-1,j], A[i,j-1]);

Region D
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      save(A[i,j]);
```

According to the original task decomposition (the one shown in the task dependence graph plotted in the previous page) and mapping of tasks ro processors suggested (also in question 4 of the previous problem), processors compute (and write) the elements of the matrix shown in the following picture (assuming that N is sufficiently large):



For each region in the program (A, B, C and D), the picture shows which data is written by each processor (P0, P1, P2 and P3) and also which data is written by each task (small numbered circles).

Assuming 4 processors, **we ask:**

1. Before the execution of region B, is it necessary to move data among processors? If yes, indicate what each processor needs to access from other processors. During the execution of region B, is it necessary to move data among processors? If yes, indicate what each processor needs to access from other processors. Write the expression (see note below) that determines the data movement overhead for region B, as a function of N.

   Solution:

   Before: Yes, it is necessary to perform remote accesses: processors P1, P2 and P3 need to access to the set of consecutive rows of A in processor P0 that need to be updated in region B. Since all remote accesses go to P0, and P0 can only accept one remote access at a time, the three accesses will be sequentialised. Each processor needs to access $(N \div 4) \times N)$, so in total $t_{ovh} = 3 \times (t_s + ((N \div 4) \times N) \times t_w)$.

   During: No, there are no remote accesses during the parallel execution in region B.

2. Before the execution of region C, is it necessary to move data among processors? If yes, indicate what each processor needs to access from other processors. During the execution of region C, is it necessary to move data among processors? If yes, indicate what each processor needs to access from other processors. Write the expression (see note below) that determines the data movement overhead for region C, as a function of N.

   Solution:

   Before: No, there is no need to access remote data before starting parallel execution of region C. All necessary data is already in place.

   During: Due to the access pattern in region C, the computation of each task in processor i requires a piece of row of $N \div 4$ elements from the previous processor (i-1); all except processor P0. There will be 3 (i.e. the number of processors minus 1) of such accesses in the critical path. So in total $t_{ovh} = 3 \times (t_s + (N \div 4) \times t_w)$.

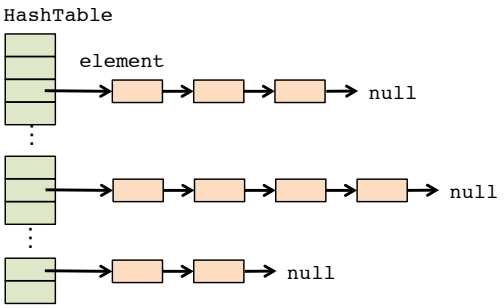
3. Before the execution of region D, is it necessary to move data among processors? If yes, indicate what each processor needs to access from other processors. Write the expression (see note below) that determines the data movement overhead for region D, as a function of N.

   Solution: Yes, processor P0 needs to collect all rows from the other processors (P1, P2 and P3) one after the other (because processor P0 can only perform one remote access at a given time). P0 needs to access $(N \div 4) \times N$ from the other processors, so in total $t_{ovh} = 3 \times (t_s + ((N \div 4) \times N) \times t_w)$.

Note: In order to write the expressions for the overheads, consider: 1) the data sharing model explained in class, where the time spent to access m elements in a different processor is $t_{comm} = t_s + m \times t_w$, being t_s and t_w the *start–up* time and the transfer time per element, respectively; 2) each processor can serve one remote access and perform one remote access, both simultaneously; and 3) N is sufficiently large so you can approximate N-1 $\approx$ N in all your expressions.

Problem 4 (Optional) Assume the following definition for a hash table used to store the elements of a list:

```
#define SIZE_TABLE 1048576
typedef struct {
    int data;
    element * next;
} element;

element * HashTable[SIZE_TABLE];
```



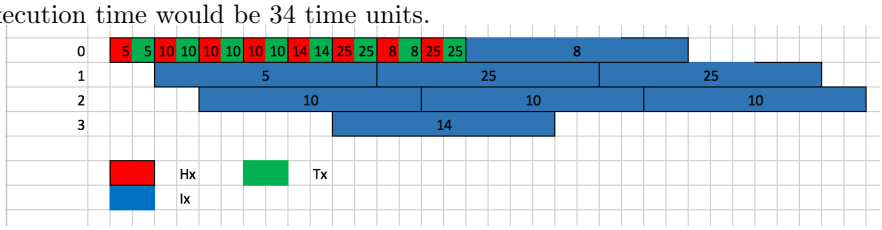
And the following parallel version of a code to insert the `MAX_ELEM` elements in vector `ToInsert`:

```
#define MAX_ELEM 1024
int ToInsert[MAX_ELEM];
...
int main() {
    int i, index, num_elem;
    ...
    #pragma omp parallel private(index) num_threads(4)
    #pragma omp single
    for (i = 0; i < num_elem; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        #pragma omp task firstprivate(i, index) depend(inout : HashTable[index])
        insert_elem (ToInsert[i], index);
    }
    ...
}
```

Function `hash_function` returns the entry of the table (between 0 and `SIZE_TABLE`-1) where each element in vector `ToInsert` has to be inserted. Function `insert_elem` inserts that element in the corresponding position inside the chained list pointed by `HashTable[index]`.

Given the sequence `index={5,10,10,10,14,25,8,25}` returned by `hash_function` for a given vector `ToInsert` with `num_elem`=8 elements, **complete the timing diagram in the answer sheet** with the parallel execution with 4 threads, assuming that the execution of `hash_function` lasts 1 time unit, task creation takes 1 time unit, the execution of `insert_elem` lasts 10 time units and the rest of the operations (including selecting the next task to be executed) can be considered to use a negligible time. **Compute the total execution time** for the parallel region. Note: If more than one task is ready for execution, a thread will choose the one that was first created.

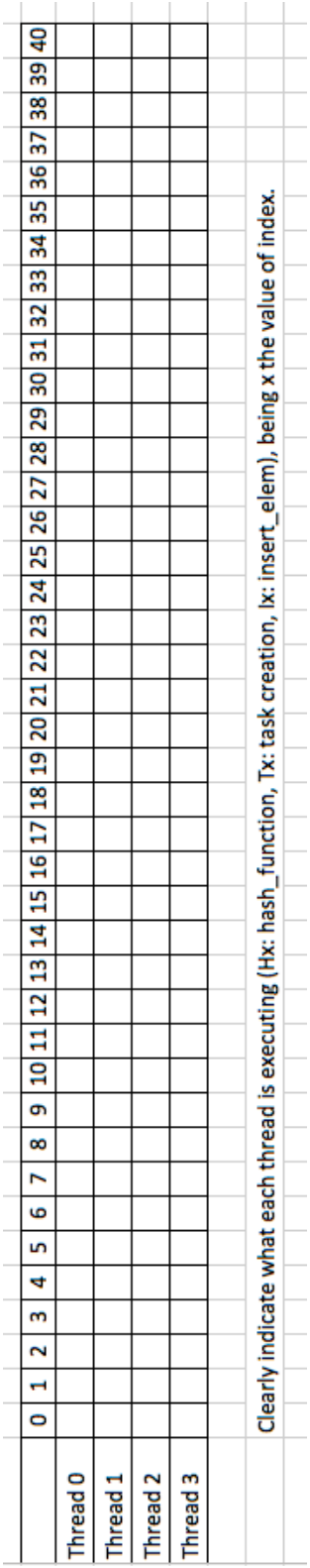
Solution: The following diagram shows the execution timeline for the previous program and sequence of indexes. The execution time would be 34 time units.



Only one thread creates all the tasks (in the diagram thread 0, but anyone could do). Tasks are executed out of order, as soon as they are created and free of dependences. If more than one task is ready for execution at the same time, the one that was first created is executed.

Student name: .....

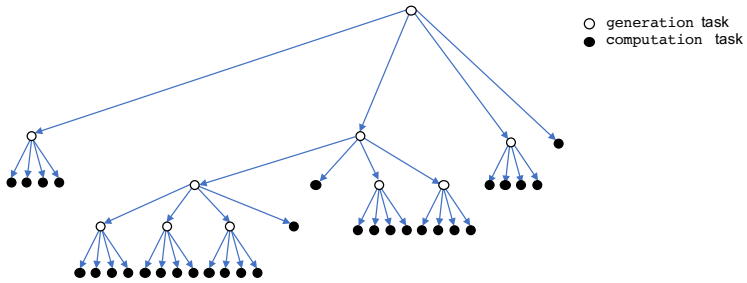
Timeline diagram to be used to deliver your solution to Problem 4.



PAR – 1^{st} In-Term Exam – Course 2016/17-Q2

April 19^{th}, 2017

Problem 1 (1.5 points) Given the following task dependence graph for the **parallelizable part** of a program:



in which there are two different kind of tasks: 1) `generation` tasks in charge of generating more tasks by traversing a recursive program; and 2) `computation` tasks in charge of doing the actual computation.

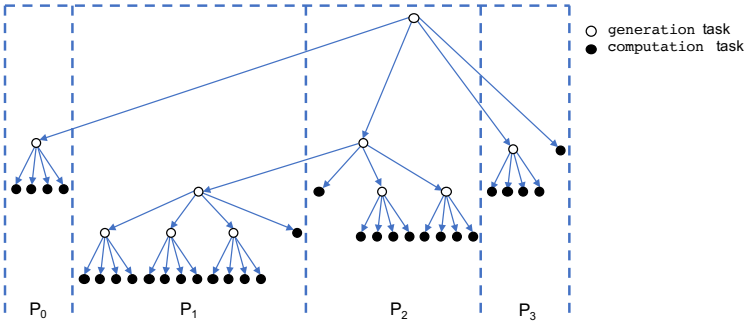
1. Assuming that the execution of each `generation` and `computation` task takes 3 and 12 time units, respectively, **we ask** to compute the values of T_1, T_∞ and $Parallelism$ for the parallelizable part of the program. Consider that both kind of tasks are also executed in the sequential version of the program.

   Solution: In the task graph there are 10 `generation` and 31 `computation` tasks. So in total $T_1 = 10 \times 3 + 31 \times 12 = 402$. The critical path in the graph has 4 `generation` and 1 `computation` tasks, so $T_\infty = 4 \times 3 + 1 \times 12 = 24$, and therefore $Parallelism = 16.75$.

2. If the execution of the **non-parallelizable** part of the program takes 38 time units, **we ask** to compute the value of the parallel fraction ϕ for the whole program and the ideal speed–up S_∞ that could be achieved if we are able to scale the parallelizable part to infinite processors.

   Solution: Since the parallelizable part takes 402 time units, the parallel fraction can be computed as $\phi = 402 \div (38 + 402) = 0.91$. According to Amdalh's law the ideal speed–up would be $S_\infty = 1 \div (1 - \phi) = 11.58$.

Problem 2 (1.5 points) Assume that the tasks in the previous task graph are assigned to 4 processors as shown in the following picture:



For the same task durations in the previous problem and duration of the non–parallelizable part, **we ask** to compute the values for T_4 and S_4 **for the whole program**.

Solution: From the assignment of tasks to processors, it is clear that processor P_1 is the one that determines the execution time, who can start its execution after P_2 has executed 2 `generation` tasks. Therefore, $T_4 = 38 + (2 \times 3) + (4 \times 3 + 13 \times 12) = 212$ and $S_4 = T_1/T_4 = 440/212 = 2.07$.

Problem 3 (2.5 points) Assume that each `computation` task performs the computation specified in the following code:

```
void computation (double *A, double *B, int size) {
    for (int i=0; i<size; i++) A[i] += foo(B[i]);
}
```

Observe that during the execution of the task each element of vector `A` accumulates the value of the corresponding element in vector `B` after invoking `foo`.

In order to improve the parallelization efficiency someone has proposed us the following idea to dynamically balance the load (number of tasks) assigned to each processor: *"as soon a processor finishes with all the tasks initially assigned to it, it steals a `computation` task from the most loaded processor at that moment; the process is repeated until all tasks are executed"*. However, stealing the execution of a task incurs an overhead that is associated with 1) the cost of moving the data that is needed to execute that task from the memory of the owner processor to the new processor, and 2) returning the computed result to the memory of the owner processor once the computation is finished. Assuming that each processor has all data associated to the initially assigned tasks and the data sharing model explained in class ($t_{overhead} = t_s + m \times t_w$, being $t_s = 1$ and $t_w = 0.1$ the start-up time and transfer time for each data element, respectively), **we ask:**

1. Compute the value of the overhead associated with the **remote execution of one `computation` task**, assuming that `size=20` for all `computation` tasks and that all the elements of a vector are transferred in a single message, each vector in a different message.

   Solution: Since `size=20`, then the overhead of each individual remote access takes $t_{overhead} = 1 + 20 \times 0.1 = 3$ time units. For each remote task execution there are two reads and one write, so the total overhead is 9 time units.

2. Using the attached solution sheet, draw a possible temporal diagram for the execution of the previous task graph, considering the initial assignment of tasks to processors shown in **problem 2** and the use of the stealing mechanism that has been proposed to dynamically balance the load initially assigned. **Important:** 1) assume that tasks are extracted from the task pool first choosing the one that is at a lower recursion level (i.e. close to the root of the tree), and second, if more than one is at the same recursion level, choosing from left to right; 2) if several threads attempt to steal tasks from the same processor at the same time, the steals will be sequentialized; and 3) according to the data sharing model, a processor can only serve one remote transfer from other processors at a time.

   Solution: See attached solution sheet.

Problem 4 (3 points) Assume that the task graph shown in **problem 1** is generated from the parallel OpenMP version of the following sequential recursive program:

```
#define SIZE_VECTOR 992     // always larger than 4 times MIN_SIZE
#define MIN_SIZE 32

void generation(double * A, double * B, int size) {
    int assigned = 0;
    for (int i=0; i<4; i++) {
        int m = partition(size-assigned, i);
        if (m > MIN_SIZE) generation(&A[assigned], &B[assigned], m);
        else computation(&A[assigned], &B[assigned], m);
        assigned += m;
    }
}
void main() {
      generation(vectorA, vectorB, SIZE_VECTOR);
}
```

In this program, function `partition` randomly returns a value that is multiple of `MIN_SIZE` (i.e. between `MIN_SIZE` and the remaining number of elements `size-assigned`), ensuring that all the `size` elements are partitioned after the four invocations in the loop. **We ask** to write an OpenMP parallelization that could potentially generate the tasks shown in the task graph in **problem 1**. In addition, in order to minimize the overheads of task creation, the code will have to include a cut–off mechanism to ensure that no additional `generation` tasks are created after a certain recursion level (`MAX_LEVEL`) (**important:** the cut-off mechanism should only apply to `generation` tasks, not to `computation` tasks).

Solution: It is not possible to make use of the clause `final` since this would also cut-off the generation of `computation` tasks at the leaves. So the cut-off is implemented with a conditional statement.

```
#define SIZE_VECTOR = 992
#define MIN_SIZE = 32

void generation(double * A, double * B, int size, int depth) {
    int assigned = 0;
    for (int i=0; i<4; i++) {
        int m = partition(size, assigned, i);
        if (m > MIN_SIZE)
            if (depth < MAX_LEVEL)
                #pragma omp task
                generation(&A[assigned], &B[assigned], m, depth+1);
            else
                generation(&A[assigned], &B[assigned], m, depth+1);
        else
            #pragma omp task
            computation(&A[assigned], &B[assigned], m);
        assigned += m;
    }
}

void main() {
    #pragma omp parallel
    #pragma omp single
    // the following initial task could be avoided if we consider the implicit task
    // executing the single
    #pragma omp task
    generation(vectorA, vectorB, SIZE_VECTOR, 0);
}
```

Problem 5 (1.5 points) Function `int foo(int input)` invoked in **problem 3** performs a certain computation that takes a considerable amount of time. Since the `input` may appear multiple times during program execution, we have decided to do an implementation that memorizes previously computed values, assuming that `input` values are always in a range between 0 and `MAX-1`:

```
struct entrada {
    int computed = 0; // 0 to indicate that the result is not already computed
    int result;
} table[MAX]

int foo(int input) {
if (table[input].computed == 0) {
    table[input].result = expensive_comp(input);
    table[input].computed = 1;
    }
return(table[input].result);
}
```

We ask you to complete the implementation of function `foo`, the main program in **problem 4** and the definition of data type `entrada` to guarantee the correct access to `table` while maximizing the parallelism.

Solution: In the solution below one thread waits if another thread is computing the result for a particular `input` vaule at that moment; as soon as the computing thread finishes, the one waiting will get the already computed value. This is better than re-computing the value, which may take longer than the waiting time for the result.

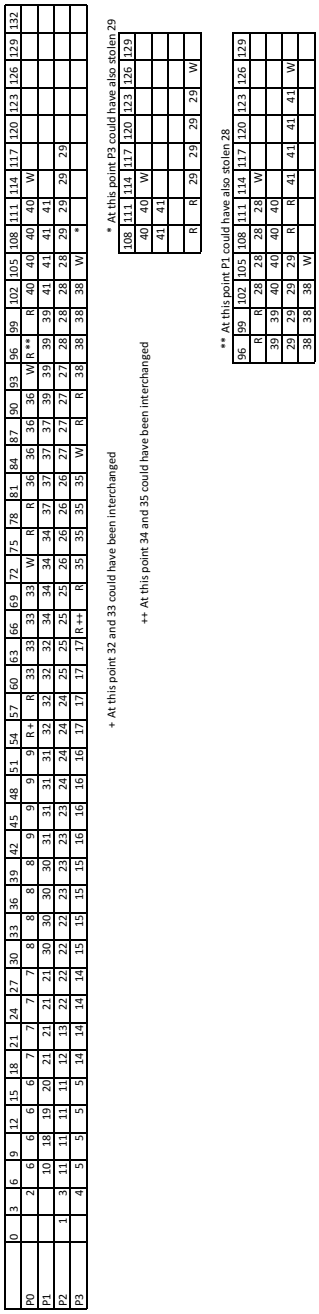
```
struct entrada {
    omp_lock_t locke;
    int computed = 0; // 0 to indicate that the result s not already computed
    int result;
} table[MAX]
```

```
int foo(int input) {
omp_set_lock(table[input].locke);
if (table[input].computed == 0) {
    table[input].result = expensive_comp(input);
    table[input].computed = 1;
    }
omp_unset_lock(table[input].locke);
return(table[input].result);
}
...
void main() {
      for (int i=0; i< MAX; i++)  omp_init_lock(&table[i].locke);
      #pragma omp parallel
      #pragma omp single
      generation(vectorA, vectorB, SIZE_VECTOR);
      for (int i=0; i< MAX; i++)  omp_destroy_lock(&table[i].locke);
}
```

In order to be the execution correct, one should include a `#pragma omp flush` in order to enforce memory consistency among threads. However this is an issue not explained in class, so it has not been considered when evaluating this problem.

Student name: .....

Task numbering and empty temporal diagram for **Problem 3**. Fill each cell with: 1) the number of the task that is executed; 2) R if a read data transfer is occurring; or 3) W if a write data transfer is occurring. **Important:** each slot in the temporal diagram corresponds to 3 time units.



PAR – 1^{st} In-Term Exam – Course 2017/18-Q1

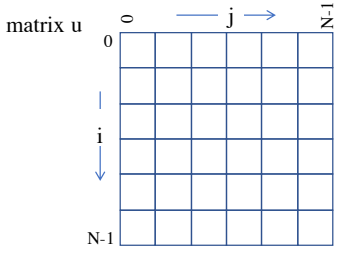
November 15^{th}, 2017

Problem 1 (4 points) Given the following code computing matrix u by blocks of $BS \times BS$ elements, following the so called *Red-Black* algorithm:

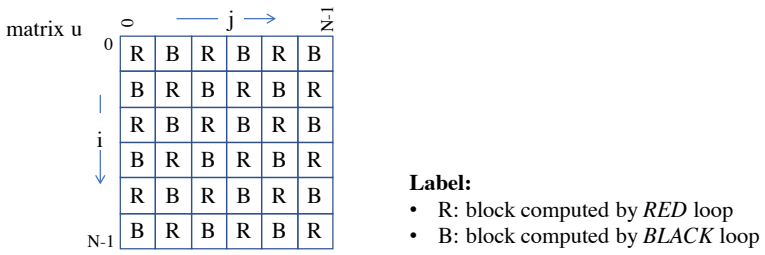
```
#define N 1024
#define BS 128
double u[N][N], residual=0.0; // residual variable only used in question 2.6

void compute_block(int ii, int jj) {
    double tmp;

    for (int i=max(1, ii); i<min(ii+BS, N-1); i++)
        for (int j=max(1, jj); j<min(jj+BS, N-1); j++) {
            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() {
    ...
    // RED loop: traversing all RED blocks
    for (int ii=0; ii<N; ii+=BS)
        for (int jj=((ii/BS)%2)*BS; jj<N; jj+=2*BS)
            // Computing a RED block of BSxBS elements
            compute_block(ii, jj);

    // BLACK loop: traversing all BLACK blocks
    for (int ii=0; ii<N; ii+=BS)
        for (int jj=(((ii/BS)+1)%2)*BS; jj<N; jj+=2*BS)
            // Computing a BLACK block of BSxBS elements
            compute_block(ii, jj);

    printf("Value for residual = %f and central element %f \n", residual, u[N/2][n/2]);
    ...
}
```

Observe that the so called *RED* loop computes half of the blocks while the so called *BLACK* loop computes the other half of the blocks. Based on this observation and the assumption that `2*BS` perfectly divides `N`, **we ask you:**

1. In the following diagram representing the blocks of matrix u computed in the previous code (assuming `N=6*BS`), indicate which blocks are computed by the *RED* and *BLACK* loops.



Label:
- R: block computed by *RED* loop
- B: block computed by *BLACK* loop



Label:
- R: block computed by *RED* loop
- B: block computed by *BLACK* loop

Solution:

2. If each invocation of function `compute_block` is defined as a task, are there any dependences precluding the parallel execution of the tasks generated during the execution of the *RED* loop? And for the tasks generated in the *BLACK* loop? Are there any dependences between the tasks generated in the *RED* loop and the tasks generated in the *BLACK* loop? If your answer to any of the previous questions is affirmative, please clearly indicate what is causing those dependences.

   Solution:

   During the execution of the *RED*, all tasks are independent since they compute non-contiguous blocks. The same occurs during the execution of the *BLACK* loop. However, there are dependences between tasks in the *RED* and *BLACK* loops. Specifically, a task generated in the *BLACK* loop depends on the 4 tasks computing the blocks above, below, right and left, due to the `[i-1,j]`, `[i+1,j]`, `[i,j+1]` and `[i,j-1]` accesses, respectively.

3. Assuming the definition of task above, write a first parallel version of the code making use of the following OpenMP constructs: `parallel`, `single`, `task` without `depend` clauses, `taskwait` and/or `taskgroup`.

   Solution:

```
void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    {
    // RED loop: traversing all RED blocks
    for (int ii=0; ii<N; ii+=BS)
        for (int jj=(ii%2)*BS; jj<N; jj+=2*BS)
            #pragma omp task
            // Computing a RED block of BSxBS elements
            compute_block(ii, jj);

    #pragma omp taskwait

    // BLACK loop: traversing all BLACK blocks
    for (int ii=0; ii<N; ii+=BS)
        for (int jj=((ii+1)%2)*BS; jj<N; jj+=2*BS)
            #pragma omp task
            // Computing a BLACK block of BSxBS elements
            compute_block(ii, jj);
    }
    printf("Value for residual = %f and central element %f \n", residual, u[N/2][n/2]);
    ...
}
```

   To guarantee the dependences between tasks in the *RED* and *BLACK* loops we need to insert `taskwait` before starting the *BLACK* loop.

4. Write a second parallel version of the code making use of the following OpenMP constructs: `parallel`, `single` and `task` with `depend` clauses.

Solution:

```
void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    {
    // RED loop: traversing all RED blocks
    for (int ii=0; ii<N; ii+=BS)
        for (int jj=(ii%2)*BS; jj<N; jj+=2*BS)
            #pragma omp task depend(out: u[ii][jj])
            // Computing a RED block of BSxBS elements
            compute_block(ii, jj);

    // BLACK loop: traversing all BLACK blocks
    for (int ii=0; ii<N; ii+=BS)
        for (int jj=((ii+1)%2)*BS; jj<N; jj+=2*BS)
            #pragma omp task
                    depend(in: u[ii-BS][jj], u[ii+BS][jj], u[ii][jj-BS], u[ii][jj+BS])
            // Computing a BLACK block of BSxBS elements
            compute_block(ii, jj);
    }
    printf("Value for residual = %f and central element %f \n", residual, u[N/2][n/2]);
    ...
}
```

Observe that the specification of depend clauses in the *RED* loop is not specifying dependences caused by the `[j-1]` and `[j+1]` accesses.

5. If the previous *RED* and *BLACK* loops are surrounded by an iterative loop, as follows:

```
void main() {
    ...
    residual = 0.0;

    for (int iter=0; iter < MAXITER; iter++) {
        // RED loop: traversing all RED blocks
        ...
        // BLACK loop: traversing all BLACK blocks
        ...
    }

    printf("Value for residual = %f and central element %f \n", residual, u[N/2][n/2]);
    ...
}
```

Do you have to do any modification to your second parallel version (tasks with `depend` clauses) in order to make the parallel execution correct. If your answer is negative justify why; if affirmative, do the necessary changes.

Solution: In order to allow parallelism across iterations of the outer `iter` loop, we need to do two changes. First, move the `parallel-single` directive pair outside that loop. Second, since now the *RED* loop also depends on blocks computed in the *BLACK* loop, we need to complete the specification of dependences in the *RED* loop.

```
void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    for (int iter=0; iter < MAXITER; iter++) {
        // RED loop: traversing all RED blocks
```

```
        for (int ii=0; ii<N; ii+=BS)
            for (int jj=(ii%2)*BS; jj<N; jj+=2*BS)
                #pragma omp task depend(out: u[ii][jj])
                        depend(in: u[ii-BS][jj], u[ii+BS][jj], u[ii][jj-BS], u[ii][jj+BS])
                // Computing a RED block of BSxBS elements
                compute_block(ii, jj);

        // BLACK loop: traversing all BLACK blocks
        for (int ii=0; ii<N; ii+=BS)
            for (int jj=((ii+1)%2)*BS; jj<N; jj+=2*BS)
                #pragma omp task depend(out: u[ii][jj])
                        depend(in: u[ii-BS][jj], u[ii+BS][jj], u[ii][jj-BS], u[ii][jj+BS])
                // Computing a BLACK block of BSxBS elements
                compute_block(ii, jj);
        }
        printf("Value for residual = %f and central element %f \n", residual, u[N/2][n/2]);
        ...
}
```

6. If function `compute_block` is redefined as follows:

```
void compute_block(int ii, int jj) {
    double tmp, diff;

    for (int i=max(1, ii); i<min(ii+BS, N-1); i++)
        for (int j=max(1, jj); j<min(jj+BS, N-1); j++) {
            tmp = u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j];
            diff = tmp - u[i][j];
            residual += diff * diff;
            u[i][j] = tmp/4;
        }
}
```

Add the required synchronization or data sharing constructs to guarantee the proper update of variable `residual`, **with the following constraint:** the overhead due to synchronization or data sharing should be kept to the MINIMUM, if possible to be executed ONLY once per task instantiation.

Solution: In order to reduce syncrhronizations, and since the specification of reduction operations is not possible with `task`, the best option is to privatize variable `residual` and do a single update at the end using an `atomic` operation:

```
void compute_block(int ii, int jj) {
    double tmp, diff;
    double local_residual = 0.0;

    for (int i=max(1, ii); i<min(ii+BS, N-1); i++)
        for (int j=max(1, jj); j<min(jj+BS, N-1); j++) {
            tmp = u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j];
            diff = tmp - u[i][j];
            local_residual += diff * diff;
            u[i][j] = tmp/4;
        }
    #pragma omp atomic
    residual += local_residual;
}
```

Problem 2 (1 point) For the loop below executed inside the scope of an OpenMP parallel region:

```
#pragma omp for schedule(dynamic, p)
for (i = 0; i < n; i++)
    b[i] = foo(a[i]);
```

Write two equivalent versions (equivalent in terms of number of chunks and number of iterations per chunk) in which:

- **version 1**: You make use of the `taskloop` construct.
- **version 2**: You make use of the `task` construct.

Note: assume that `p` exactly divides `n`.

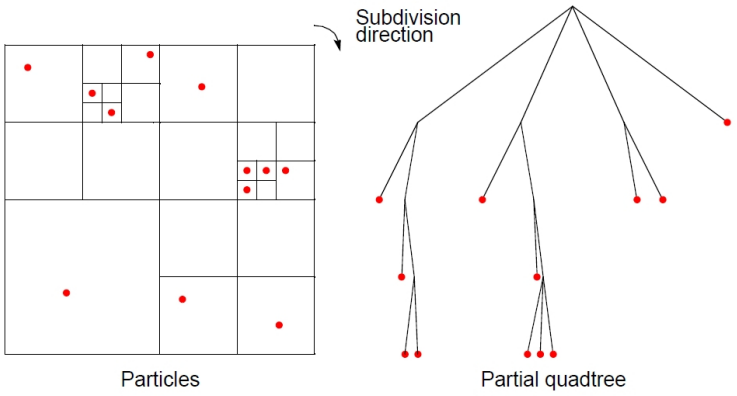
Solucion: Version 1:

```
#pragma omp single
#pragma omp taskloop grainsize(p) // or num_tasks(n/p)
for (i = 0; i < n; i++)
    b[i] = foo(a[i]);
```

Version 2:

```
#pragma omp single
for (int ii = 0; ii < n; ii +=p)
    #pragma omp task firstprivate(ii) private(i)
    for (i=ii; i<ii+p; i++)
        b[i] = foo(a[i]);
```

Problem 3 (5 points) Let us consider the use of a *quadtree* data structure to store information about particles in a 2-dimensional space. The space is divided into four quadrants, and a subtree is used to store the information corresponding to the particles in each quadrant. The *quadtree* is recursively defined by dividing each quadrant in four sub-quadrants recursively until the leaves are reached (which store the information of a single particle). To save space and computations, a branch of a subtree at any level is cut when there are no particles in the corresponding sub-quadrant. Thus, we deal with *partial quadtrees* as the one shown in the example below which has to be used to answer the questions in this problem:



We are given the following OpenMP parallel code which traverses a partial *quadtree* and updates the data items (particles) at the leaves of the *quadtree*. For each leaf, the update is the result of computing the force between the particle represented by that leaf and the reference particle provided as the second parameter in subroutine `QT_Traversal`:

```
TreeNode *root, *ref_particle;

void QT_Traversal(TreeNode *subTree, TreeNode *particle) {
                                    // particle is not included in subTree
  TreeNode *TreeQuadrant;
  for(i=0; i<4; i++) {
      TreeQuadrant = subTree.quadrant[i];
      if (TreeQuadrant != NULL)       // if this branch is not cut
          if (TreeQuadrant.isLeaf)    // subtree in this branch is a leaf
              #pragma omp task
```

```
            ComputeForces(TreeQuadrant, particle);
        else
            #pragma omp task
            QT_Traversal(TreeQuadrant, particle);
  }
}
void main() {
  Initialization (root, ref_particle); // Inherently SEQUENTIAL code
  #pragma omp parallel
  #pragma omp single
  #pragma omp task
  QT_Traversal(root, ref_particle);    // Inherently PARALLELIZABLE code
  QT_Write(root);                      // Inherently SEQUENTIAL code
}
```

The `Initialization` and `QT_Write` functions constitute the non-parallelizable part of the program. In the parallelizable part of the program, `QT_Traversal` tasks recursively traverse the *quadtree* and `ComputeForces` tasks are in charge of doing the actual computation.

1. Assuming that the execution of each `QT_Traversal` and each `ComputeForces` invocation takes 3 and 9 time units, respectively, **we ask you** to compute the values of T_1, T_∞ and $Parallelism$ for the Task Dependence Graph (TDG) shown in the last page of the exam (which would be generated by the program when traversing the *quadtree* shown above). **Notes:** 1) to answer this question you should NOT consider the assignment of tasks to processors shown in that TDG; and 2) you should NOT consider any overhead due to task creation/synchronization.

   Solution:

   In the task graph there are 8 `QT_Traversal` and 12 `ComputeForces` tasks. So in total $T_1 = 8 \times 3 + 12 \times 9 = 132$. The critical path in the graph has 4 `QT_Traversal` and 1 `ComputeForces` tasks, so $T_\infty = 4 \times 3 + 1 \times 9 = 21$, and therefore $Parallelism = 6.28$.

2. If the execution of the non-parallelizable part of the program takes 33 time units, **we ask you** to compute the value of the parallel fraction ϕ for the whole program and the ideal speed–up S_∞ that would be achieved for the whole program if we were able to scale the parallelizable part to infinite processors.

   Solution:

   Since the parallelizable part takes 132 time units, the parallel fraction can be computed as $\phi = 132 \div (33 + 132) = 0.8$. According to Amdalh's law the ideal speed–up would be $S_\infty = 1 \div (1 - \phi) = 5$.

3. Assume that tasks are assigned to 4 processors ($P_0 \ldots P_3$) as shown in the last page of the exam. The task executed at the root of the *quadtree* (i.e. the initial call to `QT_Traversal` in the main program) is assigned to P_0. Then the 4 `QT_Traversal` tasks that are generated at the first level of recursion are assigned in such a way that P_i gets assigned the task generated at iteration i (quadrants traversed in clockwise order). After that, the whole subtree of tasks generated from this first-level task is assigned to the same processor P_i. Considering the same task durations and execution time of the non–parallelizable part as in the previous questions, **we ask you** to compute the values for T_4 and S_4 for the whole program.

   Solution:

   The execution time with 4 processors will include 33 time units corresponding to the execution of the non-parallelizable part of the code; 3 time units for the execution of the initial `QT_Traversal` task in P_0; and $3 \times 3 = 9$ time units for the execution of `QT_Traversal` tasks and $5 \times 9 = 45$ time units for the execution of `ComputeForces` tasks in P_1.

   $T_4 = 33 + 3 + 3 \times 3 + 5 \times 9 = 90$ Time Units

   In order to calculate S_4 we need to recompute T_1, this time taking into account both the parallelizable and non-parallelizable parts of the program. Thus

$T_1 = 33 + 132 = 165$ Time Units

Then, $S_4 = T_1/T_4 = 165/90 = 1.83$

4. Assume that the information for all particles in the leaf nodes of the *quadtree* are stored in the memory of processor P_0. This means that all other processors will have to access to this information remotely in order to execute `ComputeForces` tasks. More specifically, function `ComputeForces` has to first read the information associated to the particle in the leaf, which occupies 20 bytes in total, do the computation, and finally write the updated leaf particle, as shown below:

```
void ComputeForces(TreeNode *subTree, TreeNode *particle) {
  TreeNode p = ReadParticle(subTree);
  Compute(&p, particle);
  WriteParticle(&p, subTree);
}
```

Observe that remote accesses are performed during the execution of the task, neither before nor after. Assuming that 1) the access to data stored in a remote processor adds an overhead of $t_{overhead} = t_s + m \times t_w$ (being $t_s = 1$ and $t_w = 0.1$ the start-up time and per-byte transfer time, respectively); 2) 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; and 3) if 2 or more processors want to simultaneously perform a remote access to the same destination processor, the request of the processor P_i with lower identifier (i=1..3) will be served first. **We ask:**

(a) Compute the minimum remote data access overhead for any of the `ComputeForces` tasks assigned to P_i (i=1..3).

Solution: Since `size=20`, then the overhead of each individual remote access takes $t_{overhead} = 1 + 20 \times 0.1 = 3$ time units. For each remote task execution there are one read and one write, so the total overhead is 6 time units.

(b) Assuming the restrictions in the number of remote accesses simultaneously supported by a processor; that the execution of each `QT_Traversal` takes 3 time units, and each `ComputeForces` invocation takes 9 time units plus the time necessary for remote data transfers (according to the remote data access overheads you just computed) while the rest of the work has negligible execution time; and considering the assignment of tasks to processors described in question 3 and the task identifiers given in the picture shown in the last page of the exam, **we ask you** to draw the temporal diagram with a scheduling of tasks in each processor that results in the BEST possible execution time of the previous OpenMP parallel program. Answer this question by filling in the temporal diagram provided in the last page of this exam.

Solution: One possible scheduling of tasks that results in the BEST possible execution time is

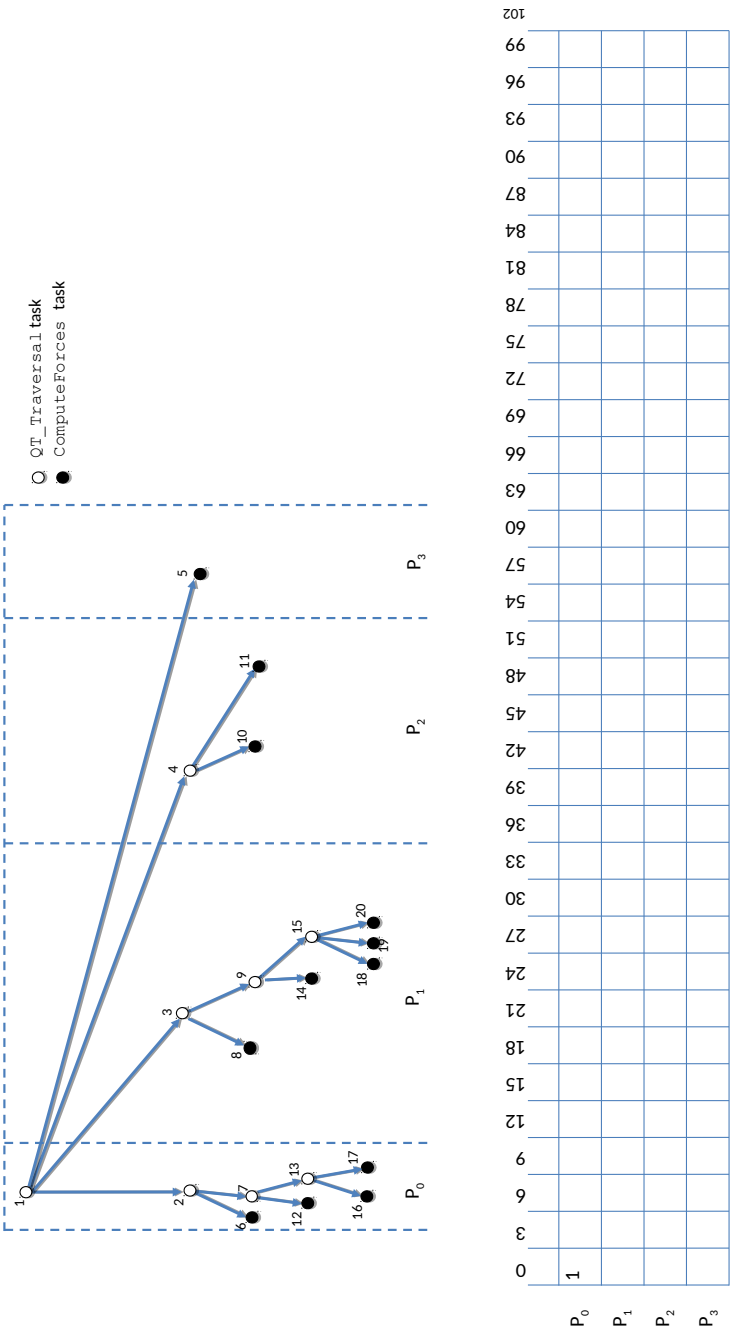
	0	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60	63	66	69	72	75	78	81	84	87
P_0	1	2	7	13	16	16	16	17	17	17	12	12	12	6	6	6														
P_1		3	9	15	R	18	18	18	W	R	19	19	19	W	R	20	20	20	W	R	14	14	14	W	R	8	8	8	W	
P_2		4	R	10	10	10	W	R	11	11	11	W																		
P_3		R	5	5	5	W																								

5. Starting from the OpenMP parallel program provided above, **we ask** you to do all the necessary changes to include a cut–off mechanism that ensures that no additional `QT_Traversal` tasks are created after a certain recursion level (`MAX_LEVEL`) (**important:** the cut-off mechanism should only apply to `QT_Traversal` tasks, not to `ComputeForces` tasks that should always be generated).

Solution: It is not possible to make use of the clause `final` since this would also cut-off the generation of `ComputeForces` tasks at the leaves. So the cut-off is implemented with a conditional statement.

```
#define MAX_LEVEL 3 // Set to 3 just as an example
#define CUT_OFF MAX_LEVEL

TreeNode *root, *ref_particle;

void QT_Traversal(TreeNode *subTree, TreeNode *particle, int d) {
                                        // particle is not included in subTree
  TreeNode *TreeQuadrant;
  for(i=0; i<4; i++) {
      TreeQuadrant = subTree.quadrant[i];
      if (TreeQuadrant != NULL)       // if this branch is not cut
          if (TreeQuadrant.isLeaf)    // subtree in this branch is a leaf
              #pragma omp task
              ComputeForces(TreeQuadrant, particle);
          else
              if(d<CUT_OFF)
                #pragma omp task
                QT_Traversal(TreeQuadrant, particle, d+1);
              else
                QT_Traversal(TreeQuadrant, particle, d+1);
  }
}
void main() {
  Initialization (root, ref_particle); // Inherently SEQUENTIAL code
  #pragma omp parallel
  #pragma omp single
  #pragma omp task
  QT_Traversal(root, ref_particle, 0);    // Inherently PARALLELIZABLE code
  QT_Write(root);                       // Inherently SEQUENTIAL code
}
```

TDG (with task numbering) and empty temporal diagram for **Problem 3**. Use the temporal diagram to answer to **question 4.b**, filling each cell with: 1) the number of the task that is executed; 2) R if a remote read data transfer is occurring; or 3) W if a remote write data transfer is occurring. **Important:** each slot in the temporal diagram corresponds to 3 time units.



PAR – 1^{st} In-Term Exam – Course 2017/18-Q2

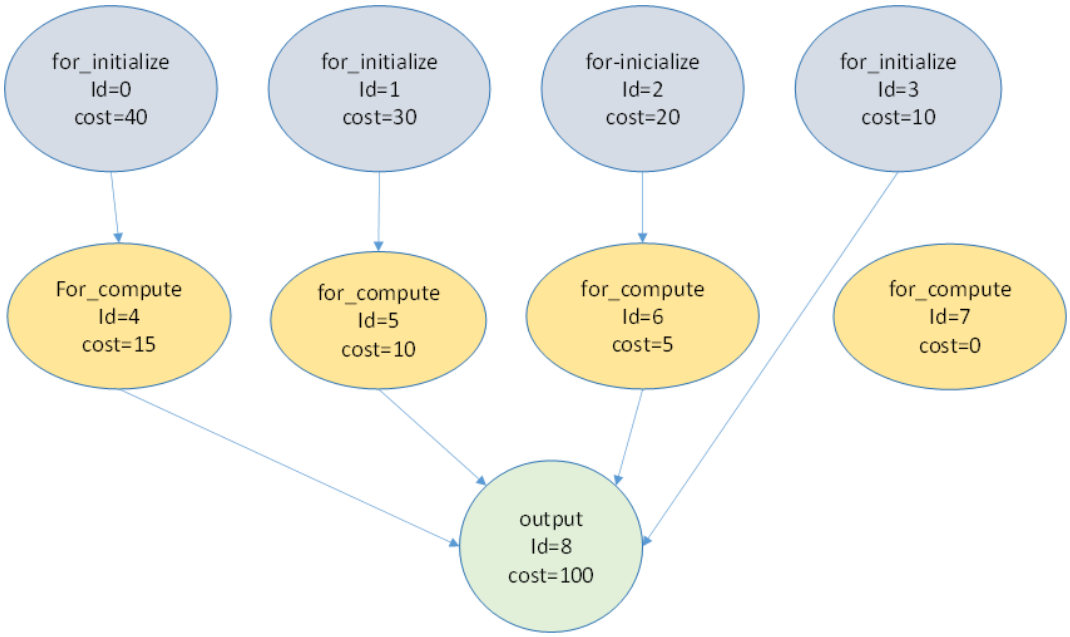
April 18^{th}, 2018

Problem 1 (3 points) Given the following C code with tasks identified using the *Tareador* API:

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

// initialization
for (int i=0; i<N; i++) {
    tareador_start_task ("for_initialize");
    for (int k=i; k<N; k++) {
        if (k == i) modify_d(&m[i][i], i, i);
        else {
            modify_nd (&m[i][k], i, k);
            modify_nd (&m[k][i], k, i);
        }
    }
    tareador_end_task ("for-initialize");
}

// computation
for (int i=0; i<N; i++) {
    tareador_start_task ("for_compute");
    for (int k=i+1; k<N; k++) {
        int tmp = m[i][k];
        m[i][k] = m[k][i];
        m[k][i] = tmp;
    }
    tareador_end_task ("for-compute");
}

// print results
tareador_star_task ("output");
print_results(m);
tareador_end_task ("output");
```

Assuming that: 1) the execution of the `modify_d` routine takes 10 time units and the execution of the `modify_nd` routines takes 5 time units; 2) each internal iteration of the computation loop (i.e. each internal iteration of the *for_ compute* task) takes 5 time units; and 3) the execution of the *output* task takes 100 time units, **we ask:**

1. Draw the task dependence graph (TDG), indicating for each node its cost in terms of execution time (in time units).



Solution:

2. Compute the values for T_1, T_∞, the parallel fraction (phi) as well as the potential parallelism.

   Solution:

   $T_1 = (40+30+20+10)+(15+10+5)+100 = 230$

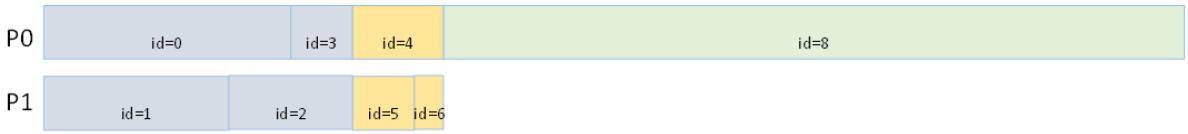
   $T_\infty = 40+15+100 = 155$

   $phi = ((40+30+20+10)+(15+10+5))/230 = 130/230 = 0.57$

   $Parallelism = 230/155 = 1.48$

3. Indicate which would be the most appropriate task assignment on two processors in order to obtain the best possible "speed up". Calculate T_2 and S_2.

   Solution:



   $T_2 = 40+10+15+100 = 165$

   $S_2 = 230 \ / \ 165 = 1.39$

Problem 2 (4 points) Given the following C code:

```
#define VEC_SIZE        1000000000
#define MIN_SIZE        10
#define MAX_TASKS       200
#define MAX_DEPTH       50

int compute_basic(int size, int *V) {
        int ret = 0;
        for (int i=0; i<size; i++) {
                ret+=foo(V[i]);
        }
        return ret;
}

int compute_rec(int size, int *V) {
        int ret = 0;
        int ret1, ret2;
        if (size > MIN_SIZE) {
                ret1 = compute_rec(size/2,V);
                ret2 = compute_rec(size/2,&V[size-size/2]);
                ret = ret1 + ret2;
        } else ret = compute_basic(size, V);
        return ret;
}

void init_vec(int size, int *V) {
        for (int i=0; i<5; i++) V[i] = i;
        // main loop
        for (int i=5; i<size; i++) V[i] = foo2(V[i-5]);
}

void main(int argc, char *argv[]) {
        int ret;
        int my_vec[VEC_SIZE];
        init_vec(VEC_SIZE, my_vec);
        ret = compute_rec(VEC_SIZE, my_vec);
        printf("%d\n", ret);
}
```

Comment: the execution of functions `foo` and `foo2` return some values based on their input arguments (which are not modified).

1. Create a parallel version in OpenMP using a recursive task decomposition for the `compute_rec` function. (In this first version you don't have to include any cut-off mechanism). Select the most appropriate strategy (tree or leaf) that will maximize the processor utilisation assuming a system with a high number of processors.

   Solution: We have selected a tree strategy, Otherwise threads will be waiting for work

   ```
   int compute_rec(int size,int *V) {
           int ret=0;
           int ret1,ret2;
           if (size>MIN_SIZE){
                   #pragma omp task shared(ret1)
                   ret1=compute_rec(size/2,V);
                   #pragma omp task shared(ret2)
                   ret2=compute_rec(size/2,&V[size-size/2]);
   ```

```
                #pragma omp taskwait
                ret=ret1+ret2;
        }else{
                ret=compute_basic(size,V);
        }
        return ret;
}

void main(int argc,char *argv[]) {
        int ret;
        int my_vec[VEC_SIZE];
        init_vec(VEC_SIZE,my_vec);
        #pragma omp parallel
        #pragma omp single
        ret=compute_rec(VEC_SIZE,my_vec);
        printf("compute_rec returns %d\n",ret);
}
```

2. Implement a task generation control mechanism based on the depth level, making use of the appropriate clauses for the OpenMP `task` construct. Use `MAX_DEPTH` as the maximum depth level to decide if tasks must be created or not.

   Solution:

   We must include a new argument to control the depth level.

```
int compute_rec(int size,int *V, int d) {
        int ret=0;
        int ret1,ret2;
        if (size>MIN_SIZE){
                #pragma omp task shared(ret1) final(d>=MAX_DEPTH) mergeable
                ret1=compute_rec(size/2,V,d+1);
                #pragma omp task shared(ret2) final(d>=MAX_DEPTH) mergeable
                ret2=compute_rec(size/2,&V[size-size/2],d+1);
                #pragma omp taskwait
                ret=ret1+ret2;
        }else{
                ret=compute_basic(size,V);
        }
        return ret;
}

void main(int argc,char *argv[]) {
        int ret;
        int my_vec[VEC_SIZE];
        init_vec(VEC_SIZE,my_vec);
        #pragma omp parallel
        #pragma omp single
        ret=compute_rec(VEC_SIZE,my_vec,0);
        printf("compute_rec returns %d\n",ret);
}
```

3. Implement a task generation control mechanism based on the number of pending tasks to be executed. Use `MAX_TASKS` as the maximum number of tasks pending to be executed to decide if we have to create a new task or not.

Solution: we need to add a new variable to control the number of tasks. In that case, it is a dynamic cut-off and we must evaluate the condition at each task creation.

```
int tasks_pending=0;
int compute_rec(int size,int *V) {
        int ret=0;
        int ret1,ret2;
        int in_parallel=0;
        if (size>MIN_SIZE){
                #pragma omp critical
                if ((tasks_pending+2)<MAX_TASKS){
                        tasks_pending=tasks_pending+2;
                        in_parallel=1;
                }
                #pragma omp task shared(ret1) if (in_parallel) mergeable
                ret1=compute_rec(size/2,V);
                #pragma omp task shared(ret2) if (in_parallel) mergeable
                ret2=compute_rec(size/2,&V[size-size/2]);
                #pragma omp taskwait
                #pragma omp critical
                if (in_parallel) tasks_pending=tasks_pending-2;
                ret=ret1+ret2;
        }else{
                ret=compute_basic(size,V);
        }
        return ret;
}

void main(int argc,char *argv[]) {
        int ret;
        int my_vec[VEC_SIZE];
        init_vec(VEC_SIZE,my_vec);
        #pragma omp parallel
        #pragma omp single
        ret=compute_rec(VEC_SIZE,my_vec);
        printf("compute_rec returns %d\n",ret);
}
```

4. After evaluating the performance obtained, we detect the function `init_vec` consumes a lot of time and we decide to paralelize it. Propose a paralelization for the main loop in `init_vec` using work-sharing constructs to extract the maximum parallelism. Justify the selected loop scheduling.

   Solution: To extract some parallelism we can not use a static "default" scheduling. We can use static with a chunk or dynamic/guided, but not the static with chuck N/num_threads (default). In that case we don't have any information concerning load balance, so we can select static,1.

```
        // main loop
        #pragma omp parallel for ordered(1) scheduling(static,1)
        for (i=5;i<size;i++){
                #pragma omp ordered depend(sink:i-5)
                V[i]=foo2(V[i-5]);
                #pragma omp ordered depend(source)
        }
```

Problem 3 (3 points) Given the following C code:

```
#define N 1024
#define BS 256
#define N_ITER 2

double u1[N][N];
double u2[N][N];

void compute_block(double A[N][N], double B[N][N], int jj) {
   double tmp;
      for (int j=max(1, jj); j<min(jj+BS, N-1); j++)
       for (int i=1; i<N-1; i++) {
         tmp = A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1] - 4*A[i][j];
         B[i][j] = tmp/4;
       }
}

void my_print(double A[N][N]) {
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
       printf("(%d,%d)=%e\n",i,j,A[i][j]);
}

void main() {
...
 for (i=0; i<N_ITER; i++) {
  // loop 1
  for (int jj=0; jj<N; jj+=BS)
    compute_block(u1, u2, jj);
  // loop 2
  for (int jj=0; jj<N; jj+=BS)
    compute_block(u2, u1, jj);
 }

 // Only processor 0 should execute this function
 my_print(u1);
...
}
```

Consider that each call to `compute_block` is a task, `my_print` is also a task only executed by processor 0 and matrices are initially distributed as indicated below. **We ask:** Write the expression that determines the execution time T_4, clearly indicating the contribution of the computation time $T_{4(comp)}$ and data sharing overhead $T_{4(mov)}$, for the following assignment of tasks to processors in each of the two iterations of `loop i` of the main program:

Task	Processor
loop1 jj=0, loop2 jj=0 and my_print(u1)	0
loop1 jj=256 and loop2 jj=256	1
loop1 jj=512 and loop2 jj=512	2
loop1 jj=768 and loop2 jj=768	3

You can assume: 1) a distributed-memory architecture with 4 processors; 2) matrices `u1` and `u2` are initially distributed by columns (BS consecutive columns per processor, where $BS = N/P$ and $P = 4$); 3) data sharing model with $t_{comm} = t_s + m \times t_w$, being t_s y t_w the start–up time and transfer time of one element, respectively; and 4) the execution time for a single iteration of the innermost loop body takes t_c time units (invocation to `compute_block`) and each call to the `printf` library function takes $2 \times t_c$ time units.

Solution:

Here, we analyze the timing corresponding to initial communication, communication during the execution of the tasks and tasks computation.

Initial Communication: At the beginning Loop 1 tasks of iteration $i = 0$ need to read the right and left boundary (P0 only right, P3 only left) from remote memories. Note that matrices are distributed by columns and `compute_block` reads `A[i][j-1]` and `A[i][j+1]`.

Communication during the execution (Dependences): Each `compute_block` reads from one matrix (`A`) and writes to another matrix (`B`). Therefore, for each iteration i, there are not dependences between tasks of Loop 1, neither between tasks of Loop 2.

However, from one Loop 1 execution to the next Loop 2 (or viceversa), there are dependences between `compute_block` tasks. In particular, there are dependences between Loop 2 tasks and Loop 1 tasks of iteration $i = 0$, Loop 1 tasks of iteration $i = 0$ and Loop 2 tasks of iteration $i = 1$, and Loop 2 tasks and Loop 1 tasks of iteration $i = 1$.

For those dependences that correspond to the same jj, there is not need of remote reads since those `compute_block` tasks (executed in Loop 1 or 2) are executed in the same processor. For those `compute_block` tasks executed in different jj (and then mapped in different processors), two remote reads of $\simeq N$ elements each will be required: right and left boundaries (due to `A[i][j-1]` and `A[i][j+1]`). Right and left boundaries consist on the $\simeq N$ elements of a column. Each message requires $t_s + N \times t_w$ u.t.

Finally, P0 processor needs to remotely read all the data of $u1$ processed in processors P1, P2 and P3 during Loop 2 of iteration $i = 1$. Once this data is read, `my_print` task can be executed. The cost of the three messages (one per remote memory) of $\simeq N \times BS\times$ elements each is $3 \times (t_s + (N \times BS\times)t_w)$ u.t.

Computation: Each `compute_block` task requires $t_{compute\_block} \simeq N \times BS \times t_c$ u.t. (t_c u.t., if you have considered the full `compute_block` invocation to be this cost), and `my_print` task requires $t_{my\_print} \simeq N \times N \times 2t_c$ u.t. ($2t_c$ u.t., if you have considered the full `my_print` invocation to be this cost).

Timing: Therefore, T_4 can be calculated as the time lasted by tasks and synchronizations done by P1 (or P2), plus the time corresponding to the communication necessary for `my_print` task and the `my_print` task execution, both in P0.

$T_4 = 2 \times (t_s + N \times t_w)$	//Initial communication (right and left boundary)
$+t_{compute\_block}$	//Loop 1 iteration i=0
$+2 \times (t_s + N \times t_w)$	//Data Dependency between Loop 1 and Loop 2 iteration i=0 (right and left boundary)
$+t_{compute\_block}$	//Loop 2 iteration i=0
$+2 \times (t_s + N \times t_w)$	//Data Dependency between Loop 2 i=0 and Loop 1 i=1 (right and left boundary)
$+t_{compute\_block}$	//Loop 1 iteration i=1
$+2 \times (t_s + N \times t_w)$	//Data Dependency between Loop 1 and Loop 2 iteration i=1 (right and left boundary)
$+t_{compute\_block}$	//Loop 2 iteration i=1
$+3 \times (t_s + (N \times BS\times)t_w)$	//Data read from P1, P2 and P3 by P0 to do the print
$+t_{my\_print}$	//Print computation

Therefore:

$T_{4(comp)} = 4 \times t_{compute\_block} + t_{my\_print}$

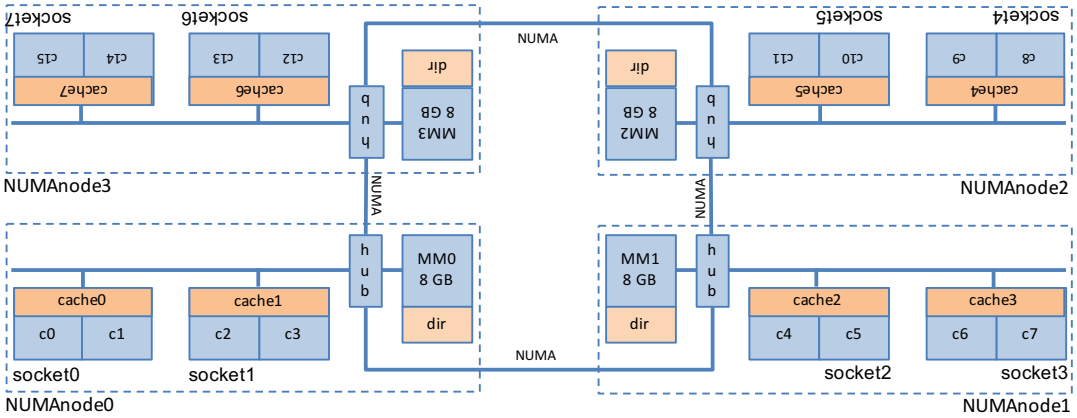
$T_{4(mov)} = 8 \times (t_s + N \times t_w) + 3 \times (t_s + (N \times BS)t_w)$

Part II

2^{nd} In-term Exams

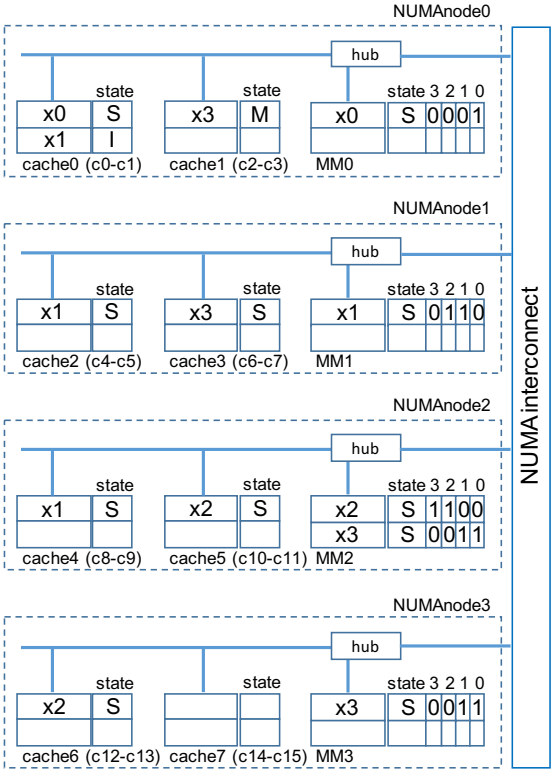
PAR – 2^{nd} In-Term Exam – Course 2016/17-Q1
December 21st, 2016

Problem 1 Assume a multiprocessor system composed of four NUMA nodes, each with two sockets and main memory (MM). Each socket has two cores and a cache memory shared by the two cores in each socket. Data coherence in the system is maintained using Write-Invalidate MSI protocols, with Snoopy cache coherency support within a NUMA node and directory-based cache coherency support between the four NUMA nodes.



coreX: Core.
socketY: Package with 2 cores, 1 MB cache shared between both cores and snoopy coherence protocol.
NUMAnodeW: set of 2 sockets conected to the same NUMA "hub"/directory with 8 GB of main memory (MM).

Assuming the initial state of the memories (caches and MM) shown in the following representation of the memory system, in which only two lines are represented for each memory (cache or MM), **we ask**:



Variables `x0`, `x1`, `x2` and `x3` are 4 bytes wide, each residing in a different 16-byte wide line

1. Who has decided the allocation of the lines containing variables `x0`, `x1`, `x2` and `x3` on the different NUMA nodes? ("the professor" and similar answers are not correct). Which of the lines containing these four variables is not in a correct state? Indicate all reasons why its state in the memory system (caches and/or MM) is not correct.

   Solution: The Operating System, using a given policy (such as "first touch"), decides the allocation. Variable `x3` is not in a correct state for two reasons: first because it has been allocated in two different NUMA nodes (2 and 3). Second, because the directory says it is S in main memory of the home NUMA node with copies in two different remote NUMA nodes (0 and 1), one of them in M state (in NUMANode0).

2. If core c2 performs PrRd on variable x0 (assuming that it is mapped in the available cache line in cache1) followed by core c0 performing PrWr on the same variable, indicate in the provided answer sheet which will be the state of the affected memories (caches and MM) after these two consecutive memory accesses.

   Solution: The first access (PrRd by core c2) will bring a new copy of x0 to cache1, also in S state. The second access (PrWr by core c0) will invalidate the line in MM0 and the recently brought line in cache1. The list of sharers in the directory of MM0 is unaffected.

3. If core c14 performs PrRd on variable x1, which memory (cache or MM) will provide to core c14 the line containing variable x1? indicate in the provided answer sheet which will be the state of the affected memories (caches and MM) after this memory access.

   Solution: For this memory access from core c14, the line containing variable x1 has to be provided by MM1. In this case the list of sharers will be updated to reflect the fact that NUMAnode3 has a new copy of that line, and cache7 will have it in S state.

4. If core c4 performs PrWr on variable x2 (and assuming that it is mapped in the available line in cache2), does someone (cache or MM) provide the line containing variable x2 to core c4? If your answer is affirmative, which memory (cache or MM) has to provide it? Indicate in the provided answer sheet which will be the state of the affected memories (caches and MM) after this memory access.

   Solution: For this memory access from core c4, the line containing variable x2 has to be provided by MM2; although core c4 will be writing x2, the other bytes in the same line need to be brought from their current location. After the memory access, the directory in MM2 will indicate that the line is invalid with the list of sharers pointing to NUMAnode1 as the only owner for that line. The other copies of the line in NUMAnode2 and NUMAnode3 will change to invalid state.

Problem 2 Consider the following data types and function definitions:

```
typedef struct {
    int key;
    int status; // -1 invalid
} tElem;
typedef tElem tVector[MAXELEM];

typedef struct tNode {
    int key;
    struct tNode *next;
} node;
typedef node *tHashTable[MAXHASH];

int valid (tElem e)
    return (!(e.status<0));

int hash_function (int key, int size)
    return (key%size);
```

```
void vector2hashTable (tVector v,  tHashTable h)
{
    int index;
    for (int i=0; i<MAXELEM; i++) {
        if (valid (v[i])) {
            index = hash_function (i, MAXHASH);
            insert_elem (v[i], index, h);
        }
    }
}
```

Function `vector2hashTable` inserts the elements from vector `v` into the hash table `h`, invoking function `hash_function` for which we provide a specific implementation in the code above. Notice that the **hash function is applied to the index** of vector `v`. Only valid elements from vector `v` (i.e. those with the `status` field $>= 0$) will be inserted into the hash table.

We ask:

1. Write a parallel OpenMP version of function `vector2hashTable` that implements a CYCLIC data decomposition strategy for the OpenMP input (i.e. consecutive elements are mapped to consecutive threads, in a round robin way). Important: a) you must NOT use the OpenMP `for` directive; and b) the number of available threads is `p`, being `MAXHASH` multiple of `p`. Is it necessary to use any kind of thread synchronization during the execution of the loop? Reason your answer.

   Solution: As a result of the proposed hash function that is applied to the index of vector v (index of vector v module MAXHASH) and as MAXHASH is multiple of p, in a cyclic distribution of the vector v, each thread is assigned a different set of index (each set contains all the index from v that are multiple of the assigned entries in the hash table). In this way, there is no conflict between threads, and consequently there is no need for synchronization inside the loop.

   ```
   void vector2hashTable (tVector v, tHashTable h, int p)
   {
    int index;
    #pragma omp parallel num_threads(p) private(index)
    {
     int myid = omp_get_thread_num();
     for (int i=myid; i<MAXELEM; i+=p) {
           if (valid (v[i])) {
                   index = hash_function (i, MAXHASH);
                   insert_elem (v[i], index, h);
           }
     }
    }
   }
   ```

2. Write a parallel OpenMP version of function `vector2hashTable` that implements a BLOCK CYCLIC data decomposition strategy for the input (i.e. blocks of consecutive `MAXHASH/p` iterations are cyclically assigned to threads, in a round robin way). Important: a) you must NOT use the OpenMP `for` directive; and b) the number of available threads is `p`, assuming that `MAXELEM` is multiple of `MAXHASH` and `MAXHASH` is multiple of `p`. Is it necessary to use any kind of thread synchronization during the execution of the loop? Reason your answer.

   Solution: As a result of the proposed hash function that is applied to the index of vector v (index of vector v module MAXHASH) and as MAXHASH is multiple of p, in a block cyclic distribution of the vector v, each thread is assigned a different set of index, some of them consecutive (each set contains all the index from v that are multiple of the assigned entries in the hash table). In this way, there is no conflict between threads, and consequently there is no need for synchronization inside the loop. In

addition, as MAXELEM is multiple of MAXHASH, there is no need of making any additional load balancing betweeen threads.

```
void vector2hashTable (tVector v, tHashTable h, int p)
{
int index;
#pragma omp parallel num_threads(p) private(index)
{
 int myid = omp_get_thread_num();
 int chunk = MAXHASH / p;

 for (int i=myid*chunk; i<MAXELEM; i+=p*chunk) {
     for (int k=i; k<min(MAXELEM, i+chunk); k++) {
        if (valid (v[i])) {
                index = hash_function (i, MAXHASH);
                insert_elem (v[i], index, h);
        }
     }
 }
}
}
```

3. Let's consider now that **the hash function is applied to the `key` field of the `tElem` structure** (i.e. `hash_function(v[i].key, MAXHASH)`). Is the OpenMP parallel code that you have written in the first question still correct?. If not, rewrite the code to make it work correctly.

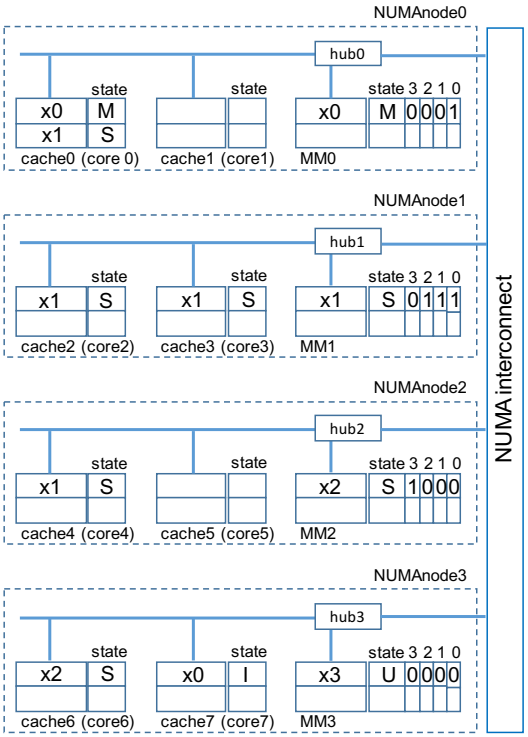
   Solution: The new proposed hash function does not guarantee any predictable distribution of index from v to the hash table. In consequence, as we cannot prevent conflicts between threads, to ensure correctness in the execution, a synchronization mechanism inside the loop has to be added. Parallel insertions at different entries in the hash table must be allowed, since no race conditions arise in that case. For this reason, an array of locks of size MAXHASH is proposed.

```
void vector2hashTable (tVector v, tHashTable h, int p)
{
int index;
omp_lock_t locks[MAXHASH];
for (int i=0; i<MAXHASH; i++) omp_init_lock(&locks[i]);

#pragma omp parallel num_threads(p) private(index)
{
    int myid = omp_get_thread_num();
    for (int i=myid; i<MAXELEM; i+=p) {
        if (valid (v[i])) {
                index = hash_function (i, MAXHASH);
                omp_set_lock(&locks[index]);
                insert_elem (v[i], index, h);
                #pragma omp flush
                omp_unset_lock(&locks[index]);
        }
    }
}
for (int i=0; i<MAXHASH; i++) omp_destroy_lock(&locks[i]);
}
```

PAR – 2^{nd} In-Term Exam – Course 2016/17-Q2
May 31st, 2017

Problem 1 (3 points) Assume a multiprocessor system composed of four NUMA nodes, each with two processors (cores) with their own cache memory and a shared main memory (MM). Data coherence in the system is maintained using Write-Invalidate MSI protocols, with a Snoopy attached to each cache memory to provide coherency within each NUMA node and directory-based coherence among the four NUMA nodes.

Assuming the initial state of the memories (caches and MM) shown in the following representation of the memory system, in which only two lines are represented for each memory (cache or MM), with 4 variables `x0`, `x1`, `x2` and `x3`, 4 bytes wide each. Variables x2 and x3 reside in the same cache line (cache lines are 32 bytes wide) while the other two reside in a different cache line each.



Coherence commands
- **Core:** $PrRd_i$ and $PrWr_i$, being I the core number doing the action
- **Snoopy:** $BusRd_j$, $BusRdX_j$ and $Flush_j$, being j the snoopy/cache number doing the action
- **Hub/directoty:** $RdReq_{i\to j}$, $WrReq_{i\to j}$, $Dreply_{i\to j}$, $Fetch_{i\to j}$, $Invalidate_{i\to j}$ and $WriteBack_{i\to j}$, from NUMAnode i to NUMAnode j

Line state in cache
- M (modified), S (shared), I (invalid)

Line state in main memory
- M (Modified), S (shared), U (Uncached)

In the representation above we also include the list of possible coherence commands at the different levels and the state names for cache and memory lines; please use this notation whenever necessary.

1. Assuming that the multiprocessor system has 8 GB ($8 * 2^{30}$) of main memory, equally distributed in the four NUMA nodes and each processor has a cache memory of 4 MB ($4 * 2^{20}$), **we ask you** to compute the amount of bits taken by each snoopy to maintain the coherence between the caches inside a NUMA node and the amount of bits used in each node directory to maintain the coherence among NUMA nodes.

   Solution: (1 point) Each node has 2 GB of main memory organized in lines 32 bytes long. Therefore each NUMA node has $2 * 2^{30}/32 = 2^{26}$ lines. For each line the directory needs to store 2 bits for the state and 4 bits for the presence bits. In total $6 * 2^{26}$ bits. Each snoopy is associated to a cache memory with $4 * 2^{20}/32 = 2^{17}$ lines. For each line only the state needs to be maintained, which again for MSI is 2 bits; therefore $2 * 2^{17} = 2^{18}$ bits.

2. Indicate which one of the above mentioned variables (only one) is not in a correct state either in a cache or in main memory and the reason for that. Based on you answer, write that variable in the correct state in the appropriate memories (cache and/or MM) in the provided answer sheet.

Solution: (0.5 points) Variables `x2` and `x3` live in the same memory line, so it is not correct that they are mapped in two different NUMA nodes; in other words, each memory line can only exist in one home node. Then we can consider that variable `x3` does not exist in NUMAnode3 and just add it in the same line as the one containing `x2` in NUMAnode2.

3. If core `c7` writes on variable `x0`, indicate in the provided answer sheet which will be the state of the affected memories and directories after this memory access. You DON'T need to enumerate the sequence of coherence actions that happen.

   Solution: (0.5 points) The Snoopy in cache0 will change its line to *Invalid* state. The directory will not change the status for that line but will change the list of sharers removing NUMAnode0 and adding NUMAnode3. The Snoopy of cache 7 will update the state of the line from Invalid to Modified.

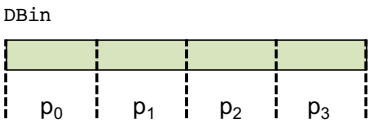
4. Finally, if core `c2` writes on variable `x1`, enumerate the sequence of coherence actions that will occur (in order of occurrence) and indicate in the provided answer sheet which will be the state of the affected memories and directories after this memory access.

   Solution: (1 point) In this case, core `c2` will issue $PrWr_2$; although the access results in a hit in cache2, the associated Snoopy will place $BusRdX_2$ in order to invalidate all possible copies of that line in other cache memories, either inside the same NUMA node or in other NUMAnodes. The state for that line in cache2 will transition from S to M. As a response to $BusRdX_2$, the Snoopy associated to cache3 will invalidate the line containing `x1` and the directory in NUMAnode1 will inform the hub that there are additional copies of that line in NUMAnodes 0 and 2, sending to them the corresponding invalidation commands $Invalidate_{1-0}$ and $Invalidate_{1-2}$. The hubs in these two NUMAnodes will transfer the invalidations to their respective buses by placing a $BusRdX$; as a response, the Snoopies for caches 0 and 4 will invalidate their lines containing `x1`. In addition, the state of that line in the memory of NUMAnode1 will change its state from S to M and the list of sharers updated in order to remove NUMAnodes 0 and 2 from the list.

Problem 2 (3,5 points) Given the following sequential code in C that looks for the position of all instances of a key (contained in variable `key`) in a portion of vector `DBin` (previously initialized randomly) storing the positions where key appears in vector `DBout`:

```
#define DBsize 1048576
int main() {
    double key=0.21;
    double * DBin = (double *) malloc(sizeof(double) * DBsize);
    double * DBout = (double *) malloc(sizeof(double) * DBsize);
    unsigned long counter = 0, min_pos, max_pos;

    for (unsigned long i = 0; i < DBsize; i++) // LOOP1
        DBin[i] = init(i);

    find_limits(DBin, &min_pos, &max_pos);

    for (unsigned long i = min_pos; i < max_pos; i++) // LOOP2
        if (DBin[i] == key) {
            DBout[counter] = i;
            counter++;
        }
}
```

1. Write a parallel version in OpenMP for the loop initializing `DBin` (LOOP1), assuming a *block data decomposition* for the input vector `DBin`, so that each thread is responsible for a block of $DBsize/P$ consecutive elements, being P the number of threads, as shown in the following figure.



Solution: (1 point)

```
#pragma omp parallel
    {
    unsigned long i, num_elems, lower;
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    lower = myid * (DBsize / howmany) +
            (myid < (DBsize % howmany) ? myid : DBsize % howmany);
    num_elems = (DBsize / howmany) + (myid < (DBsize % howmany));
    for (i = lower; i < lower + num_elems; i++)
        DBin[i] init(i);
    }
```

2. Write a parallel version in OpenMP for the second loop (LOOP2), assuming the same data decomposition strategy. You can reuse code from the previous question if needed.

   Solution: (1 point)

```
#pragma omp parallel
    {
    unsigned long i, num_elems, lower;
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    lower = myid * (DBsize / howmany) +
            (myid < (DBsize % howmany) ? myid : DBsize % howmany);
    num_elems = (DBsize / howmany) + (myid < (DBsize % howmany));
    for (i = max(lower, min_pos); i < min(lower + num_elems, max_pos); i++)
        if (DBin[i] == key) {
            DBout[counter] = i;
            counter++;
        }
    }
```

3. Insert the necessary synchronization in LOOP2 to avoid data races, minimizing the overhead that is introduced by that synchronization. Is the order of elements in `DBout` deterministic (i.e. is always the same for different executions of the parallel program)?

   Solution: (0,5 points)

```
        if (DBin[i] == key) {
            #pragma omp critical
            {
                DBout[counter] = i;
                counter++;
            }
        }
```

   The insertion order is not deterministic.

4. The originally proposed data decomposition has an important performance problem in LOOP2. Which problem are we talking about? Propose an alternative data decomposition for `DBin` that solves the performance problem and also maximizes locality in the access to `DBin`. You DON'T have to write the corresponding parallel code, just clearly describe the data decomposition proposed.

Solution: (1 point) There is a load balancing problem because the loop iterates from min_pos to max_pos, not traversing the whole vector DBin. We can solve it using a CYCLIC data decomposition. However, if we want to use all the elements in a cache line for `DBin`, we have to use a *block–cyclic decomposition* with as many elements per block as the cache line size divided by the size of a double. If the size of the cache line is S bytes and the size of a double is D bytes, the block size should be equal to S/D consecutive elements.

Problem 3 (1.5 points) The following piece of code shows the implementation of the `spin_lock` synchronization function, which works in the following way: the thread executing it tries to acquire a lock at the memory address `lock`; if the lock is already acquired, it waits for a while (`x` time units) and then tries again; the process is repeated until the thread succeeds acquiring the lock.

```
void spin_lock (int *lock, int x) {
     int ret;
     do {
          ret=test_and_set(lock, 1);
          if (ret==1)
               pause(x); /* Pause the program x time units */
     } while (ret==1);
}
```

We ask you:

1. Re-implement the `spin_lock` function for a new platform in which the `test_and_set` function is not available; you have to use `load_linked/store_conditional` instead.

   Solution: (0,5 points)

   ```
   void spin_lock (int *lock) {
        int ret;
        int value;
        do {
             value=load_linked(lock);
             ret=store_conditional(lock, 1);
             if (value==1 || ret==0)
                  pause(x);
        } while (value==1 || ret==0);
   }
   ```

2. Write an optimized version for each of the two previous implementations of `spin_lock` functions (the one with `test_and_set` and the one with `load_linked/store_conditional`) with the objective of reducing coherency traffic.

   Solution: (1 point)

   The optimization applied is based on the test-test-and-set technique to reduce the number of coherence protocol operations.

   ```
   void spin_lock (int *lock, int x) {
        int ret;
        do {
             /* test */
             while (*lock==1) {
                  pause(x);
             }
             /* test-and-set */
             ret=test_and_set(lock, 1);
             if (ret==1)
   ```

```
                pause(x); /* Pause the program x time units */
          } while (ret==1);
}
```

```
void spin_lock (int *lock) {
     int ret;
     do {
          while (load_linked(lock)==1) {
               pause(x);
          }
          ret=store_conditional(lock, 1);
          if (ret==0)
               pause(x);
          } while (ret==0);
}
```

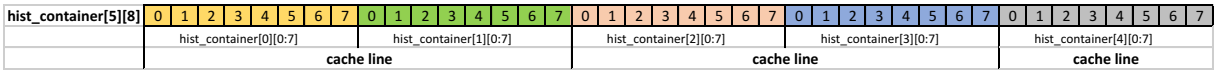
Problem 4 (2 points) Given the following parallel code in OpenMP, prepared to execute on a given number of threads (`nThreads`), that computes the histogram of vector `index` of `n` elements:

```
#define n 100000   // size of index vector
#define m 5        // Number of bins in histogram
#define nThreads 8 // Number of threads

int index[n];                     // input vector
int hist[m];                      // histogram
int hist_containers[m][nThreads]; // per-thread histogram

int main() {
    int iThread;

    InitIndex(index, n); // initialization of vector index
    initHistograms(hist, hist_containers, m, nThreads);

    #pragma omp parallel private(iThread) num_threads(nThreads)
    {
        iThread = omp_get_thread_num();
        #pragma omp for
        for (int i = 0; i < n; i++)
             hist_containers[index[i]%m][iThread]++;
    }
    for (iThread = 0; iThread < nThreads; iThread++)
        for (int i = 0; i < m; i++)
            hist[i] += hist_containers[i][iThread];
}
```

We ask:

1. Identify the main performance bottleneck (problem) that occurs during the execution of the parallel region.

   Solution: (0.5 points) The main performance bottleneck in this parallel region is false sharing. During its execution different threads are continuously WRITING to elements of `hist_containers` that reside in the same cache line, invalidating each other copies. Only reading would not cause the performance bottleneck. In particular, since the line size is 64 bytes and each integer element occupies 4 bytes, a cache line holds 16 consecutive elements of `hist_containers`, which corresponds to 2 rows of the matrix. In total the whole structure `hist_containers` occupies 2 and a half cache lines.



2. If you change the definition of the per-thread histogram to `hist_containers[nThreads][m]` (and all the accesses to it in the code accordingly), will this solve the performance bottleneck identified?

   Solution: (0.5 points) The main performance bottleneck remains, since the whole data structure still occupies 2 and a half cache lines, although shared in a different way:

   Now the same line is shared by less processors, but the false sharing problem still remains.

3. If the previous change did not solve the performance problem, do the necessary changes in the definition of `hist_containers` and/or code, without introducing other performance problems.

   Solution: (1 point) In order to avoid the FALSE SHARING problem to the data structure, the best solution is to add padding to the last definition. In particular, since cache lines are 64 bytes and each integer takes 4 bytes, we need to pad each row with 11 dummy elements. In this way each row occupies 64 bytes, a complete cache line.

```
#define m 5         // Number of bins in histogram
#define nThreads 8 // Number of threads

int hist_containers[nThreads][m+11]; // per-thread histogram
```

   This will also require to transpose all the accesses to `hist_containers` in the code.

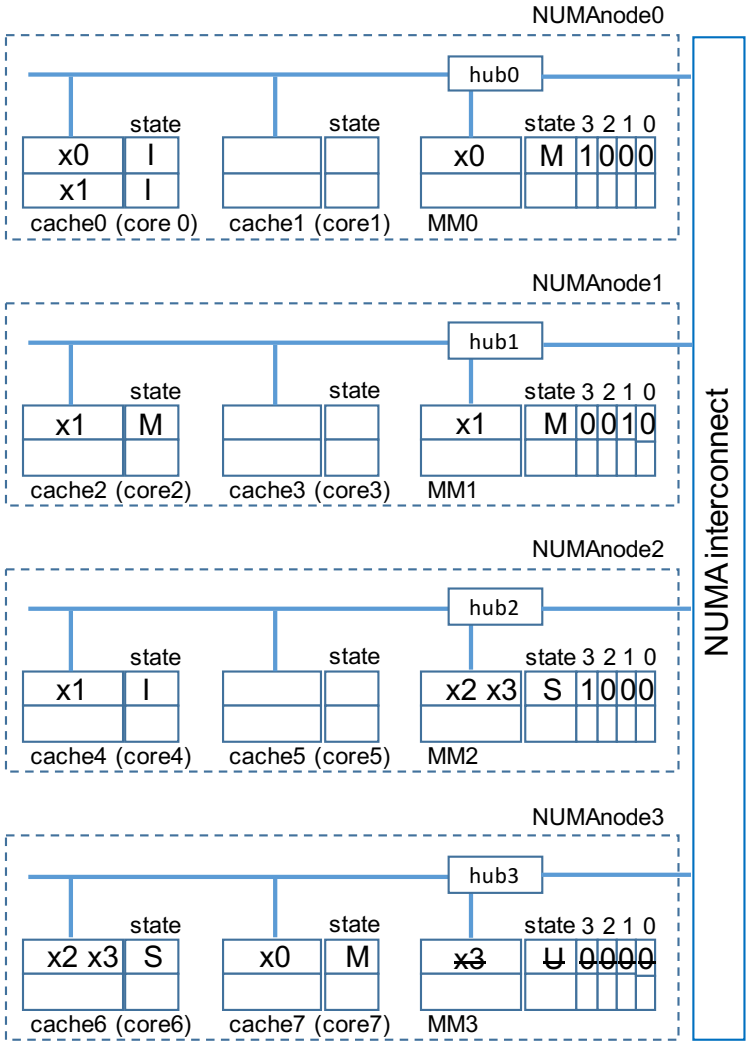
```
...
#pragma omp parallel private(iThread) num_threads(nThreads)
{
    iThread = omp_get_thread_num();
    #pragma omp for
    for (int i = 0; i < n; i++)
         hist_containers[iThread][index[i]%m]++;
}
for (iThread = 0; iThread < nThreads; iThread++)
    for (int i = 0; i < m; i++)
        hist[i] += hist_containers[iThread][i];
...
```

Note: You can assume that each integer occupies 4 bytes and cache lines are 64 bytes wide.

Student name: .....

Empty representation of the memory system, to be filled in with the final state for all the memories (caches and MM) after all the memory accesses in **Problem 1**.

Solution:



PAR – 2^{nd} In-Term Exam – Course 2017/18-Q1
January 10th, 2018

Problem 1 (4 points) Consider the following sequential C code for reversing the order of a vector of data items and doing some computation on them:

```
int * array; // pointer to start of array
int N; // length of array, assumed to be multiple of the number of processors

void swap (int i, int j) {
    int tmp = array[i]; array[i] = array[j]; array[j] = tmp;
}

void reverse ( ) {
    for (int i = 0; i < N/2; i++) swap (i, N - 1 - i);
}

void compute ( ) {
    for (int i = 0; i < N; i++) array[i] = foo(array[i], i);
}

void main( ) {
    reverse( );
    compute( );
}
```

We ask you:

1. Implement, using OpenMP, a parallel version for function `compute` that follows a *block-cyclic geometric data decomposition* for the *output* vector `array`. Decide the minimum value for the block size that better exploits data locality (assuming that cache lines are 32 bytes long and integers occupy 4 bytes). **Important:** You are **NOT ALLOWED to use the `for`** clause in your parallel version.

   Solution:

   The block size should allow the processor to use all the elements in a cache line, benefiting from spatial locality and avoiding false sharing. To implement the block-cyclic decomposition we need a nested loop, with an outer loop jumping the blocks cyclically and an inner loop traversing all the elements in each block, as follows:

```
#define CACHE_LINE_SIZE 32

void compute ( ) {
    #pragma omp parallel
    {
        int P = omp_get_num_threads();
        int id = omp_get_thread_num();
        // computation of the block size for the block-cyclic decomposition
        int block_size = CACHE_LINE_SIZE / sizeof(int)

        // loop jumping blocks cyclically
        for (int ii = id*block_size; ii < N; ii+=(P*block_size))
            // loop traversing each block
            for (int i = ii; ii < min(N, ii+block_size); i++)
                array[i] = foo(array[i], i);
    }
}
```

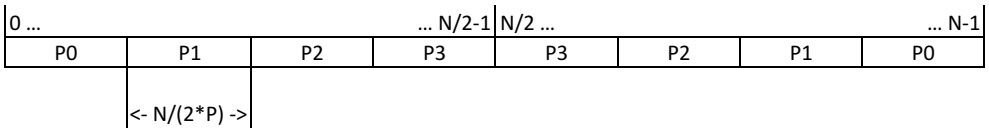
2. Draw the *geometric data decomposition* for the *output* vector `array` that would be implemented with the (incomplete) parallel version for function `reverse` that is given below:

```
void reverse ( ) {
    int P = ...;  // number of threads executing this function
    int id = ...; // identifier of the thread executing this instance (0 .. P-1)

    int segmentLength = N / ( 2 * P );
    int segmentStart = id * segmentLength;
    for (int i = segmentStart; i < segmentStart + segmentLength; i++)
        swap (i , N - 1 - i);
}
```

Complete the parallel code with the appropriate OpenMP pragmas and invocations to intrinsic functions.

Solution:

According to the owner computes rule (i.e. the output is computed/updated by the thread to which the output data is assigned) the data decomposition is:



The complete parallel code is the following:

```
void reverse ( ) {
    #pragma omp parallel
    {
        int P = omp_get_mum_threads();
        int id = omp_get_thread_num();

        int segmentLength = N / ( 2 * P );
        int segmentStart = id * segmentLength;
        for (int i = segmentStart; i < segmentStart + segmentLength; i++)
            swap (i , N - 1 - i);
    }
}
```

3. Finally, implement a new parallel version for function `compute` that follows the same *output geometric data decomposition* that has been specified above in function `reverse`.

Solution:

In order to implement the above data decomposition, the easiest way is to split the iteration space in two halves: the first assigning half of the iterations to threads 0..P-1 and the second assigning the other half of the iterations to the threads in reverse order (P-1..0).

```
void compute ( ) {
    #pragma omp parallel
    {
        int P = omp_get_mum_threads();
        int id = omp_get_thread_num();
        int segmentLength = N / ( 2 * P );

        // Loop traversing the first half of the vector
        int segmentStart = id * segmentLength;
```

```
        for (int i = segmentStart; i < segmentStart + segmentLength; i++)
                array[i] = foo(array[i], i);

        // Loop traversing the second half of the vector
        int segmentStart = N - ((id+1) * segmentLength);
                          // or also valid N/2 + (P-1-id) * segmentLength
        for (int i = segmentStart; i < segmentStart + segmentLength; i++)
                array[i] = foo(array[i], i);
    }
}
```

A simpler solution to write fuses both loops into a single one, with two calls to function `foo`, following the same pattern of invocations that was done in `traverse`.

```
void compute ( ) {
    #pragma omp parallel
    {
        int P = omp_get_mum_threads();
        int id = omp_get_thread_num();
        int segmentLength = N / ( 2 * P );

        // Loop traversing the first half of the vector
        int segmentStart = id * segmentLength;
        for (int i = segmentStart; i < segmentStart + segmentLength; i++) {
                array[i] = foo(array[i], i);
                array[N - 1- i] = foo(array[N - 1- i], N - 1- i);
        }
    }
}
```

Problem 2 (2 points) Given the following sequential code in C that counts the number of times each key in a set of keys (contained in vector `keys`) appears in a vector `DBin`:

```
#define DBsize 1048576
#define nkeys 128 // much larger than the number of processors

int main() {
    double DBin[DBsize];
    double keys[nkeys];
    unsigned int counter[nkeys];

    getkeys(keys, nkeys);      // get keys
    initialize(DBin, DBsize);  // initialize elements in DBin

    #pragma omp parallel
    for (unsigned int i = 0; i < DBsize; i++)
        #pragma omp for schedule(static, 1)
        for (unsigned int k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) counter[k]++;
}
```

Does the proposed parallelisation strategy suffer from false sharing? Justify your answer and in affirmative case propose two alternative solutions that eliminate the false sharing issue:

1. Solution 1: only changing the `schedule` clause.
2. Solution 2: changing the declaration(s) of data structure(s) and its/their access in the program.

Notes: 1) Assume that vectors `keys` and `counter` start at the beginning of a cache line; 2) cache lines are 32 bytes long; and 3) `double` and `int` data elements occupy 8 and 4 bytes, respectively.

Solution: Yes, it suffers false sharing since a chunk size equal to 1 implies that several threads can access consecutive positions of vector `counter`. Since several consecutive positions fall in the same cache line, and several threads may write into it, this can cause false sharing.

Being 32 bytes the cache line size, and 4 bytes the size of an `unsigned integer`, this implies that up to 8 elements of vector `counter` can fall in the same cache line.

```
#define NUM_COUNTER_ELEMS_PER_CACHE_LINE 8
```

- Solution 1: Change the *chunk* size.

```
... schedule(static,NUM_COUNTER_ELEMS_PER_CACHE_LINE)
```

- Solution 2: Introduce *padding* in `counter`.

```
...
unsigned int counter[nkeys][NUM_COUNTER_ELEMS_PER_CACHE_LINE];
...
if (DBin[i] == keys[k]) counter[k][0]++;
...
```

Problem 3 (4 points) Given a NUMA system with 2 nodes, each node with 2 cores (each with its own local cache memory), which implements a write–invalidate coherence scheme based on directories among NUMA Nodes and MSI snoopy within each NUMA Node. Assume that the following code is executed on 3 cores of that system:

```
// In cores 0 and 1 (NUMA Node 0)          // In core 2 (NUMA Node 1)
count = 0;
flag = 1;                                  ...
#pragma omp parallel for                   tmp = ll(flag);
for (i = 0; i < 4; i++)              lock: while (!(sc(1, flag) && tmp==0))
    #pragma omp atomic                         tmp = ll(flag);
    count++;                               ...
flag = 0;
```

Assumptions: 1) variables `i` and `tmp` are stored in two registers in the register file of each core; variables `count` and `flag` are stored in the same memory line, mapped in NUMA Node 1; 2) local cache memories are initially empty; 3) the atomic pragma does not imply additional memory accesses; and 4) the execution of `ll` (load linked) implies a $BusRd$ and $RdReq$, `sc` (store conditional) implies a single $BusRdX$ and $WrReq$, and the increment ++ implies a $BusRdX$ and $WrReq$, if any of them is necessary.

We ask you to complete the following table with the actions that occur (*NUMA transactions* and *Bus transactions*) and changes in the state of caches and directories to maintain memory coherency, assuming the temporal ordering of memory instructions shown in the same table.

Solution: Since both variables count and flag reside in the same line, there is a false sharing problem:

- the state of the line switches between M and I alternatively in the two caches; the line is only in S state when a read in one thread is done immediately after the write on the other.
- the state of the line in the directory is kept M all the time switching the list of sharers between 01 and 10, depending on which node has the line in M state in its cache; the line is only in S state in the directory when two caches have the copy also in S state.
- the coherence commands between NUMA nodes is shown in the table below, with the pattern WrRq-Dreply followed by Dreply-Fetch/invalidate that is symptom of the false sharing problem that is happening.

Time	NUMA Node 0						NUMA Node 1						
	Core 0		Core 1		Bus transactions	NUMA transactions	Core 2		Core 3	Bus transactions	NUMA transactions		
	Inst.	Cache state	Inst.	Cache state			Inst.	Cache state	Not used			Directory state	Sharers list
		flag/count		flag/count				flag/count				flag / count	1 0
0	**count=0**	M		--	BusRdX	WrReq		--		--	Dreply	M	0 1
1	**Flag=1**	M		--	--	--		--		--	--	M	0 1
2	**count++**	M		--	--	--		--		--	--	M	0 1
3		I	**count++**	M	BusRdX/Flush	--		--		--	--	M	0 1
4		I		S	BusRd/Flush	Dreply	**ll flag**	S		BusRd	Fetch	S	1 1
5	**count++**	M		I	BusRdX	WrReq		I		BusRdX	Dreply (*)	M	0 1
6		I		I	a) BusRd/Flush + b) BusRdX	a) Dreply + b) --	**sc 1, flag**	M		BusRdX	a) Fetch + b) Invalidate	M	1 0
7		I		I	--	--	**ll flag**	M		--	--	M	1 0
8		I	**count++**	M	BusRdX	WrReq		I		BusRdX/Flush	Dreply	M	0 1
9		I		I	a) BusRd/Flush + b) BusRdX	a) Dreply + b) --	**sc 1, flag**	M		BusRdX	a) Fetch + b) Invalidate	M	1 0
10	**flag=0**	M		I	BusRdX	WrReq		I		BusRdX/Flush	Dreply	M	0 1
11		S		I	BusRd/Flush	Dreply	**ll flag**	S		BusRd	Fetch	S	1 1
12		I		I	BusRdX	--	**sc 1, flag**	M		BusRdX	Invalidate	M	1 0

(*) Assuming that memory always provides line in S

SURNAME:

NAME:

Time	NUMA Node 0						NUMA Node 1								
	Core 0		Core 1		Bus transactions	NUMA transactions	Core 2		Core 3	Bus transactions	NUMA transactions				
	Inst.	Cache state flag/count	Inst.	Cache state flag/count			Inst.	Cache state flag/count	Not used			Directory state flag / count	Sharers list 1	0	
0	**count=0**														
1	**Flag=1**														
2	**count++**														
3			**count++**												
4							**ll flag**								
5	**count++**														
6							**sc 1, flag**								
7							**ll flag**								
8			**count++**												
9							**sc 1, flag**								
10	**flag=0**														
11							**ll flag**								
12							**sc 1, flag**								

(1) Cache line state: M (modified), S (shared) or I (invalid)

(2) Coherence commands inside NUMA node: BusRd, BusRdX and Flush

(3): Line state in node memory: M (modified), S (shared) or U (uncached)

(4) Coherence commands between NUMA nodes: RdReq, WrReq, Dreply, Fetch and Invalidate

PAR – Second In-Term Exam – Course 2017/18-Q2

June 6^{th}, 2018

Problem 1 (4 points) Given the following code (sequential version):

```
#define X_SIZE        40960
#define Y_SIZE        40960
#define V_SIZE    128
double *M,*V;
void main(int argc,char *argv[])
{
int i,j,aux,pos;
M=malloc(sizeof(double)*X_SIZE*Y_SIZE);
V=malloc(sizeof(double)*V_SIZE);
for (i=0;i<V_SIZE;i++) V[i]=0;
for (i=0;i<X_SIZE;i++) ReadRowFromFile(M,i);
// Main loop
for (i=0;i<X_SIZE;i++){
   for (j=0;j<Y_SIZE;i++){
                  aux=ComputeElement(M,i,j);
                  pos=ComputePos(i,j);
                  V[pos]=+aux;
   }
}
}
```

Comment: Function ReadRowFromFile is not provided . It reads one row (i) from disk , assuming each row has Y_SIZE elements, and stores the result in M (row=i). ComputePos is a function that computes the position to be inserted on V. It only depends on i,j. The function ComputeElement(M,i,j) doesn't modify M.

1. We ask you to create a OpenMP parallel version applying a block geometric data decomposition strategy per rows on matrix M for both initialization and main loop. You cannot use the `#pragma omp parallel for` neither `#pragma omp for` constructs. The block geometric decomposition should minimize the load unbalance on the distribution of the rows. Also, reason if you need to include any synchronization in the main loop.

   Solution: variables `i`, `j`, `aux` and `pos` must be declared as `private`. It has been considered as correct solution the use of `critical` or `atomic` constructs to protect the access to vector `V`. However, the most efficient solution is using a vector of lock variables with as many elements as `V_SIZE`.

   ```
   #define X_SIZE        40960
   #define Y_SIZE        40960
   #define V_SIZE    128
   double *M,*V;
   void main(int argc,char *argv[])
   {
       int i,j,aux,pos;
       M=malloc(sizeof(double)*X_SIZE*Y_SIZE);
       V=malloc(sizeof(double)*V_SIZE);
       for (i=0;i<V_SIZE;i++) V[i]=0;
       #pragma omp parallel private(i,j,aux,pos)
       {
   ```

```
      int id=omp_get_thread_num();
      int nump=omp_get_num_threads();
      int lower,upper;
      // compute the lower and upper limits
      lower=id*(X_SIZE/nump);
      if (id<(X_SIZE%nump)) lower+=id;
      upper=lower+(X_SIZE/nump)+(id<(X_SIZE%nump));
      for (i=lower;i<upper;i++) ReadRowFromFile(M,i);
      // Main loop
      for (i=lower;i<upper;i++){
        for (j=0;j<Y_SIZE;i++){
         aux=ComputeElement(M,i,j);
         pos=ComputePos(i,j);
         #pragma omp atomic
         V[pos]=+aux;
        }
      }
    }
}
```

2. Create a new version were we will implement an output data decomposition to be sure each processor is accessing to N consecutive positions of vector V. Reason if you need to include some synchronization.

   Solution: We must compute specific limits for vector `V` and update only those positions assigned to the current thread. With this strategy there is no need to use synchronisation since matrix `M` is not modified and each thread is accessing different positions of vector `V`.

```
#define X_SIZE        40960
#define Y_SIZE        40960
#define V_SIZE    128
double *M,*V;
void main(int argc,char *argv[])
{
    int i,j,aux,pos;
    M=malloc(sizeof(double)*X_SIZE*Y_SIZE);
    V=malloc(sizeof(double)*V_SIZE);


    for (i=0;i<X_SIZE;i++) ReadRowFromFile(M,i);
    // Main loop
    #pragma omp parallel private(i,j,aux,pos)
    {
     // Computing lower and upper limits
     int id=omp_get_thread_num();
     int nump=omp_get_num_threads();
     int pos_lower,pos_upper;

     pos_lower=id*(V_SIZE/nump);
     if (id<(V_SIZE%nump)) pos_lower+=id;
     pos_upper=pos_lower+(V_SIZE/nump)+(id<(V_SIZE%nump));
     for (i=pos_lower;i<pos_upper;i++) V[i]=0;
     for (i=0;i<X_SIZE;i++){
       for (j=0;j<Y_SIZE;i++){
          pos=ComputePos(i,j);
          if ((pos>=pos_lower) && (pos<pos_upper)){
```

```
                aux=ComputeElement(M,i,j);
                V[pos]=+aux;
              }
            }
          }
        }
    }
```

Problem 2 (3 points) Given the following OpenMP code:

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

struct t_person best_client;

void find_best_client(struct t_person *people, int n) {
   #pragma omp parallel for
   for(int i=0; i<n; i++)
      if (best_client.balance< people[i].balance)
           best_client = people[i];
}

int main() {
   struct t_person bank_info[N_MAX];
   ...
   best_client.balance=0.0;

   find_best_client(bank_info, N_MAX);
   ...
}
```

Considering that `reduction` clause cannot be used on variables of type struct, **we ask you:**

1. There is a concurrency problem in the proposed OpenMP code for `find_best_client`. What is this concurrency problem? Reason your answer.

   Solution:

   There is a data race condition when updating global variable `best_client`: two threads may concurrently compare (read) the value in `best_client.balance` with value in the `people[i].balance` they have read, and decide to update `best_client`. The read and update must be done in mutual exclusion to ensure that the global `best_client` variable is updated appropriately.

2. Propose a modification of the code to avoid this concurrency problem just inserting a `omp` construct.

   Solution:

   We only need to insert a `#pragma omp critical` to force mutual exclusion, including the read (if statement) and update (assignment) of the `best_client` global variable. We cannot use `#pragma omp atomic` since this only works for some basic operations.

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

```
   struct t_person best_client;

   void find_best_client(struct t_person *people, int n) {
      #pragma omp parallel for
      for(int i=0; i<n; i++)
         #pragma omp critical
         if (best_client.balance< people[i].balance)
              best_client = people[i];
   }

   int main() {
      struct t_person bank_info[N_MAX];
      ...
      best_client.balance=0.0;

      find_best_client(bank_info, N_MAX);
      ...
}
```

3. Propose an improvement of your previous modification to significantly reduce unnecessary synchronization overheads. This solution should also avoid false sharing problems.

   Solution:

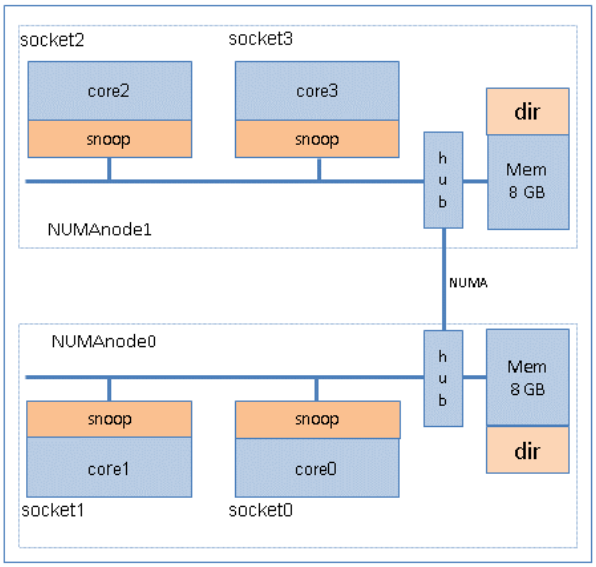
   In order to significantly reduce the amount of unnecessary synchronization we can use the `test&test&set` technique. We will check `best_client.balance` value before forcing a mutual exclusion area. If there is a chance of updating the `best_client` variable, we force a mutual exclusion and check and update the `best_client` global variable if necessary. With this solution there is not a false sharing situation to avoid.

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

   struct t_person best_client;

   void find_best_client(struct t_person *people, int n) {
      #pragma omp parallel for
      for(int i=0; i<n; i++)
         if (best_client.balance< people[i].balance)
           #pragma omp critical
             if (best_client.balance< people[i].balance)
              best_client = people[i];
   }

   int main() {
      struct t_person bank_info[N_MAX];
      ...
      best_client.balance=0.0;

      find_best_client(bank_info, N_MAX);
      ...
}
```

   Other possible solutions are:

- To use a private and local `local_best_client` for each thread, and then, after the for loop and before finishing the `parallel` region, use a `critical` construct to update `best_client` global variable.
- To use a global vector variable of as many entries as number of threads. Then, each thread will update one entry of the vector with the `best_client` they find. However, this last version may have a false sharing problem because two consecutive entries of the vector of `best_client`, updated by two different threads, may share a cache line. In this case, padding the elements of the vector or create a table with the enough padding to provoke that two elements of the vector doesn't share any cache line will avoid the false sharing problem.

Problem 3 (3 points) Assume a multiprocessor system composed of two NUMA nodes, each with two sockets. Each socket has one core with a cache memory of 8MB. **The cache line size is 16 bytes**. Data coherence in the system is maintained using **Write-Invalidate MSI protocols**, with a **Snoopy attached to each cache memory** to provide coherency within each NUMA node and **directory-based coherence among the two NUMA nodes**.



coreX: Core.
socketY: Package with 1 core,
8 MB cache and snoopy coherence protocol
The cache line size is 16 bytes
NUMAnodeZ: set of 2 sockets connected to the same NUMA "hub"/directory with 8 GB of main memory.
node: Node with 2 NUMAnodes.

Coherence commands
• **Core:** PrRdi and PrWri, being i the core number doing the action
• **Snoopy:** BusRdj, BusRdXj and Flushj, being j the snoopy/cache number doing the action
• **Hub/directoty:** RdReqij, WrReqij, Dreplyij, Fetchij, Invalidateij and WriteBackij, from NUMAnode i to NUMAnode j

Line state in cache
• M (modified), S (shared), I (invalid)
Line state in main memory (MM)
• M (Modified), S (shared), U (Uncached)

Given the following C code:

```
#define N 16
int x[N];
...
#pragma omp parallel num_threads(4)
{
    int myid = omp_get_thread_num();
    int nths = omp_get_num_threads();
    int i_start, i_end;

    i_start = (N / nths) * myid;
    i_end = i_start + N/nths;
    // FOR loop
    for (int i=i_start; i<i_end; i++) x[i]=init();
}
```

and assuming that: 1) the Operating System decides the data allocation using a "first touch" policy; 2) vector `x` is the only variable that will be stored in memory (the rest of variables will be all in registers of the processors); 3) the initial address of vector `x` is aligned with the start of a cache line; 4) the size of an `int` data type is 4 bytes; and 5) $thread_i$ always executes on $core_i$, where $i = [0 - 3]$, **we ask you:**

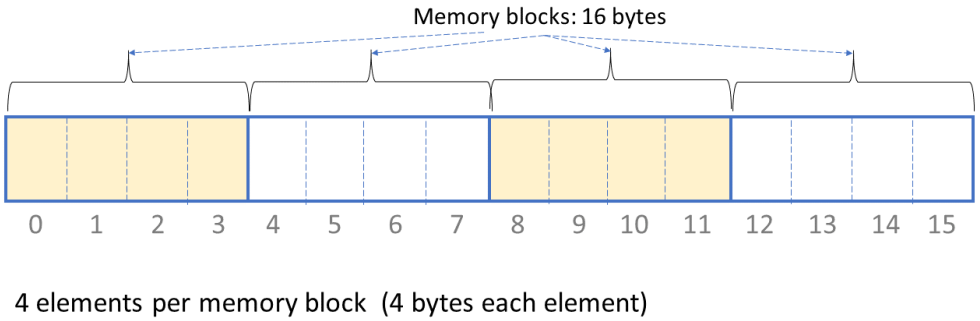
1. Compute the amount of bits taken by each snoopy to maintain the coherence between caches inside a NUMA node and, the amount of bits in each node directory to maintain the coherence among NUMA nodes.

   Solution:

   Each node has 8 GB of main memory organized in lines 16 bytes long. Therefore each NUMA node has $8 * 2^{30}/16 = 2^{29}$ lines. For each line the directory needs to store 2 bits for the state and 2 bits for the presence bits; therefore $4 * 2^{29} = 2^{31}$ bits. Each snoopy is associated to a cache memory with $8 * 2^{20}/16 = 2^{19}$ lines. For each line only the state needs to be maintained, which again for MSI is 2 bits; therefore $2 * 2^{19} = 2^{20}$ bits.

2. Draw a picture that shows vector `x` and how many memory lines are necessary to store its elements, identifying the range of elements per memory line.

   Solution:



3. Assuming that all cache memories are empty at the beginning of the program, fill in the attached `Table 1` with the information corresponding to each range of elements allocated per memory line once all threads arrive to the end of the parallel region: vector range, the Home node number, the presence bits, main memory line state (`State in MM`) corresponding to accesses to vector `x`, and the state of any copy (`State in cache socket0-3` in the table) of those memory blocks in one or more caches of sockets 0 to 3.

   Solution: Table 1: to be used to deliver your solution to Problem 3.3

Vector x range	# Home NUMA node	Presence bits	State in MM	State in cache			
				socket0	socket1	socket2	socket3
0-3	0	01	M	*M*	I	I	I
4-7	0	01	M	I	*M*	I	I
8-11	1	10	M	I	I	*M*	I
12-15	1	10	M	I	I	I	*M*

4. Assuming the final previous state of the multiprocessor system with the presence bits and state for each cache and memory line, fill in the attached `Table 2` with the sequence of processor commands (Core), bus transactions within NUMA nodes (Snoopy), transactions between NUMA nodes (Directory), the presence bits, state for each cache and memory line, to keep cache coherence, **AFTER the execution of each** of the following sequence of commands:

   (a) $core_2$ reads the contents of `x[2]`

   (b) $core_2$ writes the contents of `x[2]`

   (c) $core_1$ reads the contents of `x[0]`

Solution: Table 2: to be used to deliver your solution to Problem 3.4

Command	Coherence actions			Presence bits	State in MM	State in cache			
	Core	Snoopy	Directory			socket0	socket1	socket2	socket3
$core_2$ reads x[2]	$PrRd_2$	$BusRd_2$ $BusRd/Flush_0$	$RdReq_{10}$ $Dreply_{01}$	11	S	S	I	S	I
$core_2$ writes x[2]	$PrWr_2$	$BusRdX_2$ $BusRdX$	$WrReq_{10}$	10	M	I	I	M	I
$core_1$ reads x[0]	$PrRd_1$	$BusRd_1$ $BusRd/Flush_2$	$Fetch_{01}$ $Dreply_{10}$	11	S	I	S	S	I

1. $core_2$ reads the contents of `x[2]`: $core_2$ triggers a $PrRd$ event; the line that holds `x[2]` is not loaded in the local cache, so the associated snoopy generates a $BusRd$ transaction to notify about the reading operation. The Local Hub is not the Home Hub, so the Local Hub sends a $RdReq$ operation to the Home Hub (in Numa Node 1) to ask for a copy of the value. The Home Hub knows because of the presence bits that there is a modified copy in its Numa node. A $BusRd$ transaction on the bus generates a $Flush$ operation from $core_0$ and the cache line state is modified to Shared. The line state in the directory of the Home Node is also changed to $Shared$ and in the list of sharers Numa Node 1 is added. The Home Hub sends back with a $Dreply$ message to the Local Hub the requested line.

2. $core_2$ writes the contents of `x[2]`: $core_2$ triggers a $PrWr$ event; although the element `x[2]` is already in the local cache (hit), the state is shared so the associated snoopy generates a $BusRdX$ to notify other caches that the line is going to be modified. The local Hub sends a $WrReq$ message to the Home Hub which owns the list of sharers that have a copy of this element. The Home Hub launches a $BusRdX$ transaction (presence bits indicate there is a copy in Numa Node 0), then cache line state where the copy resides is invalidated. Line state in the directory in the Home Node is changed to $Modified$, presence bits are changed to 10 expressing that just Numa Node 1 has a valid copy of the line.

3. $core_1$ reads the contents of `x[0]`: $core_1$ generates a $PrRd$ event; the line where `x[0]` resides is not loaded in the local cache, so this activates the snoopy protocol generating a $BusRd$ transaction. As the variable `x[0]` shares memory line with `x[2]`, the element is already loaded in the cache associated to $core_2$, in the other Numa Node (this information is in the list of sharers). The Local Hub, which in this case is the Home Hub, gets the current value of `x[0]` by asking the line with a $Fetch$ operation to the Owner Hub. The Owner Hub activates the snoopy protocol, sending a $BusRd$ operation, which makes $core_2$ to Flush the cache line and changes its cache line state to $Shared$. The Owner Hub flushes the line and responds to the Home Hub with a $Dreply$ operation. The line state in memory is updated to $Shared$, the list of sharers is updated by adding Home Hub, and the cache line state associated to the cache of $core_1$ is changed to $Shared$.

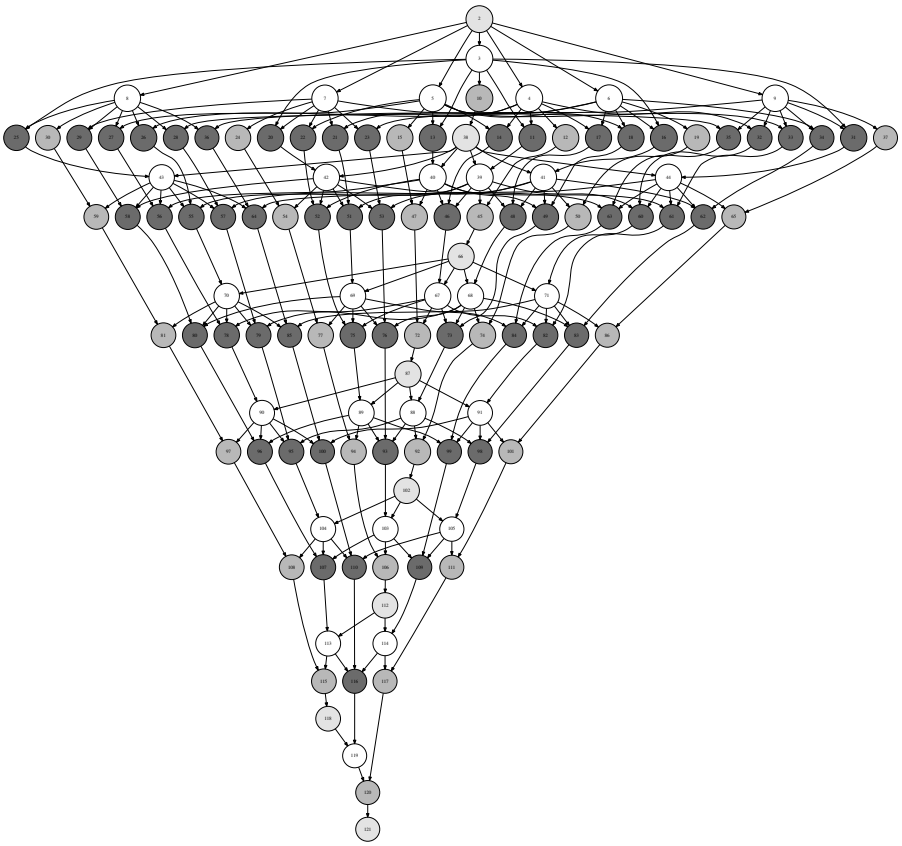
Part III

Final Exams

PAR – Final Exam – Course 2016/17-Q1

January 13, 2017

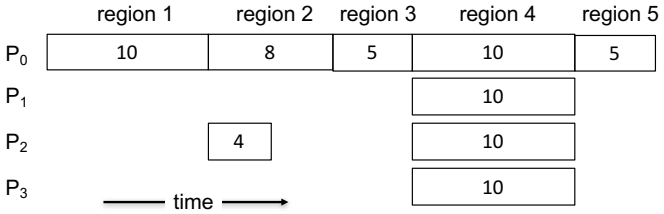
Problem 1 (0.5 points): Given the following task dependence graph, in which each node represents the execution of a given task and arrows represent data dependences between tasks:



Assuming that the execution time for each node (task) is one time unit, **we ask you to compute** the potential parallelism in an ideal parallel machine with infinite number of identical processors.

Solution:
$T_1 = 120$, which corresponds with the total number of nodes in the graph.
$T_\infty = 22$, which corresponds with the number of nodes in the critical path in the graph.
Therefore, the parallelism is $Par = T_1/T_\infty = 120/22 = 5,45$

Problem 2 (1 point): Given the following temporal execution timeline for a task decomposition on 4 processors:

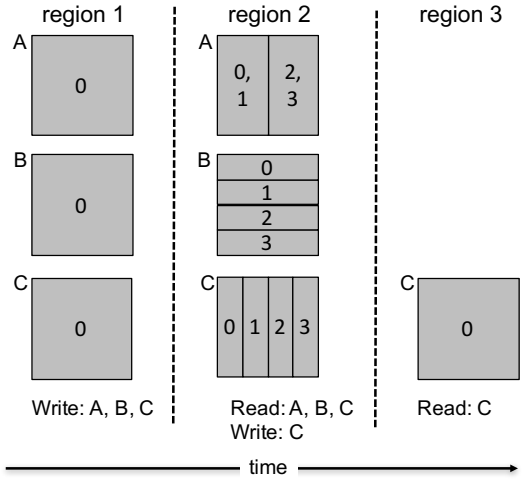


in which each burst corresponds with the execution of one task assigned to one of the available processors (P_0 ... P_3). The number inside each burst represents the execution time of the associated task. Assuming that

code regions 1, 3 and 5 can not be parallelized, region 2 can only make use of 2 processors as shown above and that region 4 scales perfectly with any number of processors, **we ask you to compute** the speed–up obtained when using 4 processors (S_4) and the potential speed–up if infinite processors were available (S_∞).

Solution: We need to compute T_1 as a reference to compute both speed–up metrics: $T_1 = 10 + (8 + 4) + 5 + (4 \times 10) + 5 = 72$.
In order to compute S_4 we need to compute the execution time with 4 processors: $T_4 = 10+8+5+10+5 = 38$, and therefore $S_4 = 72/38 = 1.89$.
To compute the ideal speed–up with infinite processors we consider that only region 4 perfectly scales. So in this case $T_\infty = 10 + 8 + 5 + 0 + 5 = 28$, and therefore $S_\infty = 72/28 = 2.57$ (region 2 does not scale beyond 2 processors).

Problem 3 (1.5 points): Assume a program with 3 code regions executed one after the other (with a barrier between them), as shown in the following execution timeline. Regions 1 and 3 are executed sequentially (by processor 0) while region 2 is parallelized using 4 processors:



In this timeline we also show for each region how the three matrices used in the program (A, B and C, all of size $n \times n$) are accessed (read or write) and which part of the matrix is actually used (numbers inside indicate processor number). Assuming 1) the data sharing model explained in class in which the overhead of a remote access is determined by $t_{comm} = t_s + m \times t_w$, being t_s the start-up time and t_w the time to transfer one element; 2) regions 2 and 3 start their computation once all data needed is available; 3) in region 2 matrix C is first read and then written; and 4) there is no data sharing during the parallel execution in region 2, **we ask you to write** the expression for the data sharing (i.e. communication) overheads associated with the execution of the whole program, clearly indicating what is contributiong to that expression.

Solution: Remote accesses occur before starting Region 2 and Region 3. For region 2 all processors except processor 0 (i.e. processors 1 to 3) need to access processor 0 to get the part of matrices A, B and C that are needed. Since all of them are accessing the processor 0, and only one remote access can be served at any time, all accesses will be serialised. For matrix A, each of the three processors will have to access $N^2/2$ elements; for matrices B and C each of the three processors need to access $N^2/4$ elements.
For region 3, only processor 0 has to do remote access to collect matrix C. Since a processor can only request one request one remote access at any time, the accesses will be again serialised, each with $N^2/4$ elements.
So in total, 9 accesses of $N^2/4$ and 3 accesses of $N^2/2$, which contribute with an overhead of $t_{comm} = (9 \times (t_s + (N^2/4) \times t_w) + 3 \times (t_s + (N^2/2) \times t_w))$ time units.

Problem 4 (1 point): Given a generic distributed-memory architecture composed of several cluster nodes. Each cluster node provides shared-memory to several NUMA nodes, each one including a number of sockets with a certain number of cores per socket. **Indicate if the following statements are true or false**. For each correct answer you will obtain 0.2 points; if the answer is not correct, we will apply a penalty of 0.07 points):

a) With write-invalidate MSI inside a NUMA node, the invalidation command is generated every time a core performs PrWr on a valid line in its cache memory. **Solution: False**

b) A given physical memory address can only be stored in the memory of a single NUMA node inside a cluster node, although multiple copies of that address may be temporarily stored in the cache memories of other NUMA nodes in the same cluster node. **Solution: True**

c) The directory structure in a NUMA node provides information that allows a core in that NUMA node to find the nearest NUMA node in the same cluster node where to find a given memory address, in order to minimize the access time. **Solution: False**

d) The continuous execution of a `count++` instruction in two different cores inside a NUMA node or in different NUMA nodes of the same cluster node, protected with the appropriate synchronization mechanism, originates false sharing. **Solution: False**

e) Load (ld) and store (st) instructions are no longer used to access data residing in a remote cluster node; message exchange (point–to–point and collective) has to be used instead with the appropriate support of a NIC (Network Interface Card). **Solution: True**

Problem 5 (3 points): Given the following code:

```
#define MAX_ITER 10000
#define NUM_ELEM 8192

// data array aligned to a cache line
double data[NUM_ELEM];
double compute( int elem, ... );

for (timestep = 0; timestep < MAX_ITER; timestep++)
    for (int elem = 0; elem < NUM_ELEM; elem++)
        data[elem] += compute (elem, ... );
```

being `compute` a function with the following characteristics: 1) does not generate dependencies between iterations in the loop `elem` and 2) its execution time varies depending on the value of `elem`, which may originate a load imbalance problem. In order to alleviate this problem an array of lists is defined: this array will be used to record the list of iterations of the loop `elem` that is assigned to each thread.

```
#define NUM_THREADS 8
typedef struct {
    int elem;
    iteration * next;
} iteration;

iteration * decomposition[NUM_THREADS];

// Inserts elem at the beginning of list. Last element of the list points to NULL
void insert(int elem, iteration * list);
// Initializes all the num_elems elements of an array of lists to NULL
void initialize(int num_elems, iteration * array);
```

The task decomposition strategy that we want to apply has an *Inspector/Executor* scheme. In the *Inspector* phase only ONE iteration of the `timestep` loop is executed; this iteration is used to dynamically determine the mapping of iterations of the `elem` loop to *threads*. During the *Executor* phase THE REST OF iterations of the `timestep` loop (`MAX_ITER - 1` iterations) are executed, following the decomposition found in the *Inspector* phase, as shown in the following incomplete code:

```
initialize(NUM_THREADS, decomposition);

// INSPECTOR PHASE
#pragma omp parallel for schedule( ... /* incomplete code (a) */ )
for (int elem = 0; elem < NUM_ELEM; elem++) {
    ... /* incomplete code (b) */
    data[elem] += compute (elem, ... );
}

// EXECUTOR PHASE
for (timestep = 1; timestep < MAX_ITER; timestep++) {
    ... /* incomplete code (c) */
}
```

We ask:

a) Complete the schedule clause (code section (a)) that has to be used in the *Inspector* phase so that the mapping of iterations to threads minimizes the load unbalance problem and avoids *false sharing* in the access to the `data` structure (you can NOT modify the definition of the `data` structure). Note: you can assume that the constant `CACHE_LINE_SIZE` defines the size of a cache line in bytes and remember that the operator `sizeof(data_type)` returns the size in bytes of `data_type`.

Solution: Given that the execution time of each loop iteration depends on the value of `elem`, the more appropiate iteration distribution is a `dynamic` one. In addition, to avoid *false sharing* accessing to `data` without using any type of *padding*, the best solution is to use a dynamic chunk that assures that elements of the same cache line are written by the same thread. Therefore, the schedule should be: `schedule(dynamic, CACHE_LINE_SIZE / sizeof(double))`.

b) Complete the code section (b) in the *Inspector* phase, using the `decomposition` structure and its associated functions to implement the functionality mentioned above for the *Inspector* phase.

Solution: Each thread should insert in its list the iterations assigned by the schedule policy `(dynamic, CACHE_LINE_SIZE / sizeof(double))`.

```
// INSPECTOR PHASE
#pragma omp parallel for schedule(dynamic, CACHE_LINE_SIZE / sizeof(double))
for (int elem = 0; elem < NUM_ELEM; elem++) {
    insert(elem, decomposition[omp_get_thread_num()]);
    data[elem] += compute (elem, ... );
}
```

c) Complete the code section (c) in the *Executor* phase, so that the task decomposition determined during the *Inspector* phase is applied, making use of the information stored in the `decomposition` structure. also

Solution: Each thread should traverse its list to obtain the list of iterations that were assigned to it during the inspector phase.

```
// EXECUTOR PHASE
#pragma omp parallel private(timestep)
for (timestep = 1; timestep < MAX_ITER; timestep++) {
    iteration * iter = decomposition[omp_get_thread_num()];
    for ( ; iter != NULL; iter = iter.next) {
```

```
            data[iter.elem] += compute (iter.elem, ... );
        }
    }
```

d) Write an alternative parallel code for the *Inspector* phase based on the use of `#pragma omp task` that follows a recursive *divide-and-conquer tree* strategy. As in section a), the implementation must minimize load unbalance and avoid *false sharing*. Regarding to this, we ask you to explain, with a comment in your code, how you recursively divide the work to be done by each task (task granularity) so that *false sharing* does not appear between different threads executing different tasks.

Solution:

```
// Define the elements per cache line
#define CACHE_LINE_ELEMS (CACHE_LINE_SIZE / sizeof(double))

void inspect_rec(int iter, int elements) {
    // base case when number of elements fits in a cache line
    if (elements <= CACHE_LINE_ELEMS ) {
        for (int elem = iter; iter < elements; elem++) {
            insert(elem, decomposition[omp_get_thread_num()]);
            data[elem] += compute (elem, ... );
        }
        return;
    }
    // divide and conquer

    // We should divide it in a cache line boundary

    // First, compute number of blocks of cache line elements
    // that goes to first partition
    int blocks_div_2= (elements/CACHE_LINE_ELEMS)/2;

    // Second, translate it to number of elements
    int elemdiv2 = blocks_div_2 * CACHE_LINE_ELEMS;

    #pragma omp task   // first private by default
    inspect_rec(iter, elemdiv2);
    #pragma omp task
    inspect_rec(iter+elemdiv2, elements-elemdiv2);
}

// INSPECTOR PHASE
#pragma omp parallel
#pragma omp single
inspect_rec (0, NUM_ELEM);
```

Problem 6 (3 points): The following parallel code for (`matmul`) does the multiplication of matrices A and B using submatrices:

```
#define N  128
#define BS 32

void matmult_submatrix(int N, int BS, float *A, float *B, float *C);

void matmult(float A[N][N], float B[N][N], float C[N][N]) {
#pragma omp parallel num_threads(N/BS)
#pragma omp for collapse(2)
  for (int i=0;i<N;i+=BS)
```

```
    for (int j=0 ;j<N; j+=BS)
      for (int k=0 ;k<N; k+=BS)
         matmul_submatriz(N, BS, &A[i][k], &B[k][j], &C[i][j]);
}
```

being the `matmul_submatrix` function the one that multiplies a submatrix of $BS \times BS$ elements of A by one of B and updates the result on the submatrix of $BS \times BS$ elements of C. This implementation is based on the fact that $C = A \times B$ can be seen as the calculation of their submatrices $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$, as follows:

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

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$
$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$
$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$
$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

We ask: (see footnote below[1])

a) Draw matrices A, B and C, divided in blocks (submatrices) clearly indicating which thread is accessing (reading and/or writing) each block. Based on that, indicate for each matrix A, B and C which data decomposition has been applied.

Solution: Matrices A and C: Block Data Decomposition by rows, as follows:

0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3

For B there is no data decomposition.

b) Write an alternative parallel code to the proposed one that implements the same data decomposition strategy in which you DON'T make use of OpenMP `for` work–sharing construct.

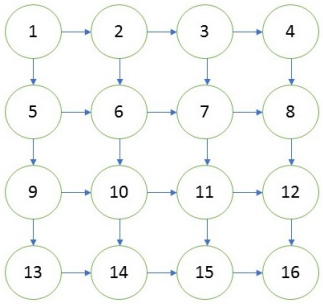
Solution:

```
#define N  128
#define BS 32

void matmult_submatriz(int N, int BS, float *A, float *B, float *C);

void matmult(float A[N][N], float B[N][N], float C[N][N]) {
#pragma omp parallel num_threads(N/BS)
 {
    int my_id = omp_get_thread_num();
    int i     = my_id * BS;
    for (int j=0 ;j<N; j+=BS)
      for (int k=0 ;k<N; k+=BS)
         matmul_submatriz(N, BS, &A[i][k], &B[k][j], &C[i][j]);
 }
}
```

[1] From the OpenMP specification, about the `collapse` clause: "If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space."

PAR – Final Exam – Course 2016/17-Q2
June 19th, 2017

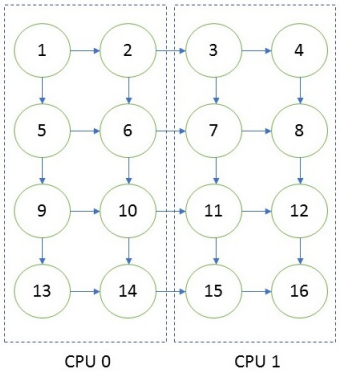
Problem 1 (2 points) Given the following **task dependence graph** in which each node represents a task, with an execution of 10 time units, and each edge represents a dependence relationship between two tasks:

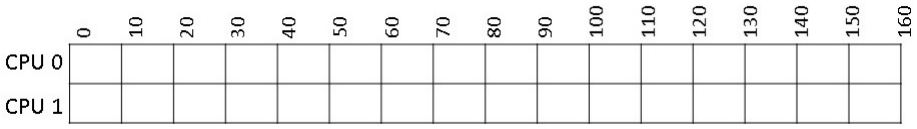


1. Calculate the values for the T_1, T_∞ and potential *Parallelism* metrics.

   Solution: In the task graph there are 16 tasks. So in total $T_1 = 10 \times 16 = 160$. The critical path in the graph has 7 tasks (i.e. *{1,2,3,4,8,12,16}*), so $T_\infty = 10 \times 7 = 70$, and therefore $Parallelism = 2.29$.

2. Fill in the temporal diagram below (by writing in each box the number of the task that is executed), for the execution of the previous task graph in an **ideal machine** with two processors, that 1) considers the following assignment of tasks to processors and 2) minimizes the parallel execution time:





	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160
CPU 0	1	2	5	6	9	10	13	14									
CPU 1			3	4	7	8	11	12	15	16							

Solution:

3. From the previous temporal diagram, calculate T_2 and the "speed–up" S_2 that would be ideally obtained.

   Solution: As can be seen in the temporal diagram $T_2 = 100$, so the speedup $S_2 = T_1/T_2 = 160/100 = 1.6$

4. Re-calculate T_2 and S_2 when **data sharing overheads** are taken into account. To do that, use the data sharing model explained in class based on the distributed memory architecture and messages model. In that model, the access time to remote data is determined by $t_{remote} = t_s + m \times t_w$, being $t_s = 0.1$ the "start-up" time for the message, $t_w = 0.001$ the transfer time for each element in the message and $m = 100$ the number of elements of the message that is being communicated between processors when there is a dependence between tasks. **Clearly enumerate** the tasks that participate in each remote data transfer (source and target task).

   Solution: There are four dependencies between tasks mapped in CPU0 and CPU1, as indicated in the task graph; therefore we have four data transfers: $2 \rightarrow 3$, $6 \rightarrow 7$, $10 \rightarrow 11$ and $14 \rightarrow 15$, which account for a total data sharing overhead of $4 \times (t_s + m \times t_w)$. Substituting by the given values we have: $T_2 = 100 + 4 \times (0.1 + 100 \times 0.001) = 100 + 4 \times 0.2 = 100.8$ and $S_2 = T_1/T_2 = 160/100.8 = 1.59$.

Problem 2 (3 points) Given the following sequential code in C that computes vector `count`, which counts the number of times a set of keys (contained in vector `keys`) appear in a vector `DBin`:

```
#define DBsize 1048576
#define nkeys 128

int main() {
    double keys[nkeys], DBin[DBsize];
    unsigned long counter[nkeys], i, k;

    getkeys(keys, nkeys);            // get keys
    init_DBin(DBin, DBsize);         // initialize elements in DBin
    init_counter(counter, nkeys);    // initialize vector counter

    #pragma omp parallel for private(k)
    for (i = 0; i < DBsize; i++)
        for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) {
                #pragma omp atomic
                counter[k]++;
            }
```

1. Write an alternative OpenMP parallelization that implements an **iterative task decomposition strategy** in which there is no need for synchronization during the computation of the vector `count` and in which you minimize the overheads due to parallelism.

   Solution: Parallelize the `k` loop and interchange the loops to minimize overheads.

```
    #pragma omp parallel for private(i)
    for (k = 0; k < nkeys; k++)
        for (i = 0; i < DBsize; i++)
            if (DBin[i] == keys[k])
                counter[k]++;
}
```

2. Write an OpenMP parallelization to the original code that takes into account all the following requirements: 1) implements a **recursive divide–and–conquer task decomposition strategy**; 2) uses a **cut–off strategy based on the size of the input vector**, so that tasks are only generated when that size is larger than `CUT_SIZE`; 3) only uses OpenMP pragmas and clauses for the implementation of the cut–off strategy; and 4) **makes use of locks** instead of `atomic` to guarantee exclusive access.

   Solution:

```
#define DBsize 1048576
#define nkeys 128
#define CUT_SIZE 4
```

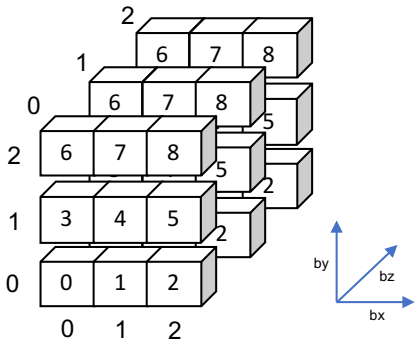
```
omp_lock_t locks[nkeys];

void recursion (double * DBin, double *keys, unsigned long *counter,
                int left, int right) {
    if ((left - right) == 1) { // base case
        for (k = 0; k < nkeys; k++)
            if (DBin[left] == keys[k]) {
                // use of individual locks to maximize concurrency
                omp_set_lock(&locks[k]);
                counter[k]++;
                omp_unset_lock(&locks[k]);
            }
    } else {
        int middle = (left+right) / 2;
        #pragma omp task final((left-right) < CUT_SIZE) mergeable
        recursion(DBin, keys, counter, left, middle);
        #pragma omp task final((left-right) < CUT_SIZE) mergeable
        recursion(DBin, keys, counter, middle+1, right);
    }
}

int main() {
    double keys[nkeys], DBin[DBsize];
    unsigned long counter[nkeys], i, k;

    ...

    for (k=0; k < nkeys; k++) omp_init_lock(&locks[k]);

    #pragma omp parallel
    #pragma omp single
    recursion(DBin, keys, counter, 0, DBsize);

    for (k=0; k < nkeys; k++) omp_destroy_lock(&locks[k]);
}
```

Problem 3 (3 points) The following sequential code computes $C = compute(A, B)$, being A, B and C three dimensional matrices, all of them divided in a number of blocks in each dimension (defined by constants $BLOCK\_X$, $BLOCK\_Y$ and $BLOCK\_Z$.

```
#define BLOCK_X 2
#define BLOCK_Y 4
#define BLOCK_Z 8
#define MATRIX_SIZE 1000

double *A,*B,*C;

// computes a single block, iterating in each dimension xbegin <= x < xend
void compute_block(int xbegin, int xend, int ybegin, int yend,
                   int zbegin, int zend);

// Computes indexes to compute based on bx, by, bz
void compute_index(int *xbegin, int *xend, int *ybegin, int *yend,
                   int *zbegin,int *zend, int bx, int by, int bz) {
...
}

void main(int argc, char *argv[]) {
```

```
        int bx, by, bz;
        int x_start, x_end, y_start, y_end, z_start, z_end;
        A = malloc(MATRIX_SIZE*MATRIX_SIZE*MATRIX_SIZE*sizeof(double));
        B = malloc(MATRIX_SIZE*MATRIX_SIZE*MATRIX_SIZE*sizeof(double));
        C = malloc(MATRIX_SIZE*MATRIX_SIZE*MATRIX_SIZE*sizeof(double));

        ReadData();    // Initialize A and B

        for (bz=0; bz<BLOCK_Z; bz++)
            for (bx=0; bx<BLOCK_X; bx++)
                for (by=0; by<BLOCK_Y; by++) {
                    compute_index(&x_start, &x_end, &y_start, &y_end,
                                  &z_start, &z_end, bx, by, bz);
                    compute_block(x_start, x_end, y_start, y_end, z_start, z_end);
                }

        PrintResults(); // Prints C
}
```

We want to implement a parallel version using OpenMP directives using $BLOCK\_X * BLOCK\_Y$ processors and the **data decomposition** for matrices A, B and C shown below:



for the case of $BLOCK\_X = BLOCK\_Y = BLOCK\_Z = 3$. The number inside each block corresponds with the processor number that has assigned the block.

In order to implement the parallel code, we provide you with the following skeleton in which at each iteration of the `bz` loop each thread computes one specific block determined by the values of variables `bx`, `by` and `bz`, as shown below:

```
        ...
        #pragma omp parallel
        for (bz=0; bz<BLOCK_Z; bz++) {
                bx = ...
                by = ...
                compute_index(&x_start, &x_end, &y_start, &y_end,
                              &z_start, &z_end, bx, by, bz);
                compute_block(x_start, x_end, y_start, y_end, z_start, z_end);
        }
        ...
```

We ask (you can not add any other directive but you can use calls to the OpenMP intrinsic functions if needed):

1. Include the appropriate clauses for the `parallel` directive
2. Include the instructions to compute the value for `bx` and `by` following the owner-computes rule.

3. Complete the definition of function `compute_index` for a general case that the number of blocks in each dimension does not perfectly divides $MATRIX\_SIZE$.

Solution:

```
void compute_index(int *xbegin,int *xend,int *ybegin,int *yend,
                   int *zbegin,int *zend,int bx,int by,int bz)
{
    int bsize_x,bsize_y,bsize_z;
    int rem_x,rem_y,rem_z;

    bsize_x=MATRIX_SIXE/BLOCK_X;
    bsize_y=MATRIX_SIXE/BLOCK_Y;
    bsize_z=MATRIX_SIXE/BLOCK_Z;

    rem_x=MATRIX_SIXE%BLOCK_X;
    rem_y=MATRIX_SIXE%BLOCK_Y;
    rem_z=MATRIX_SIXE%BLOCK_Z;
    *xbegin=bsize_x*bx+(bx<rem_x?bx:rem_x);
    *xend=*xbegin+bsize_x+(bx<rem_x?1:0);
    *ybegin=bsize_y*by+(by<rem_y?by:rem_y);
    *yend=*ybegin+bsize_y+(by<rem_y?1:0);
    *zbegin=bsize_z*bz+(bz<rem_z?bz:rem_z);
    *zend=*zbegin+bsize_z+(bz<rem_z?1:0);
}

void main(int argc,char *argv[])
{
    int bx,by,bz;
    #pragma omp parallel num_threads(BLOCK_X*BLOCK_Y) private(bx,by,bz)
    for (bz=0;bz<BLOCK_Z;bz++) {
        int x_start,x_end,y_start,y_end,z_start,z_end;
        int who=omp_get_thread_num();
        *bx=who%BLOCK_X;
        *by=who/BLOCK_X;
        compute_index(&x_start,&x_end,&y_start,&y_end,&z_start,&z_end,bx,by,bz);
        compute_block(x_start,x_end,y_start,y_end,z_start,z_end);
    }
}
```

Problem 4 (2 points) Given a NUMA architecture with 2 nodes, each one with just one core (with its own local cache memory), using directory write-invalidate coherence, in which we execute the following code:

```
// processor P0 (node 0)              // processor P1 (node 1)
count = 0;                                  ...
flag = 1;                                   tmp = ll(flag);
for (i = 0; i < 4; i++)             lock:   while (!(sc(1, flag) && tmp==0))
    count++;                                    tmp = ll(flag);
flag = 0;                                   ...
```

Assume that variables `i` and `tmp` are stored in two registers in the register file, variables `count` and `flag` are stored in the same memory line, mapped in node 1 and assuming initially empty cache local memories.

We ask to complete the following table with the actions that occur (*NUMA transactions*) and changes in the state of caches and directories to maintain memory coherency assuming the following temporal ordering of memory instructions:

Time	Node with P0			Node with P1					
	Instruction	P0 cache status (1) flag/count	NUMA transaction (2)	Instruction	P1 cache status (1) flag/count	NUMA transaction (2)	Directory state (3) flag / count	Sharers list 1	0
0	**count=0**								
1	**flag=1**								
2				**ll flag**					
3	**count++**								
4				**sc 1, flag**					
5				**ll flag**					
6	**count++**								
7				**sc 1, flag**					
8				**ll flag**					
9	**count++**								
10				**sc 1, flag**					
11				**ll flag**					
12	**count++**								
13				**sc 1, flag**					
14	**flag=0**								
15				**ll flag**					
16				**sc 1, flag**					
			(1) M (modified), S (shared) or I (invalid)						
			(2): RdReq, WrReq, Dreply, Fetch, Invalidate, WriteBack						
			(3): M (modified), S (shared) or U (uncached)						

Important: 1) the execution of the `ll` implies a `BusRd` command inside the node and a `RdReq` between nodes, if any of the two is necessary; 2) the execution of the `sc` implies a `BusRdX` command inside the node and a `WrReq` between nodes, if any of the two is necessary; 3) and the increment ++ implies a single `BusRdX` command inside the node and a single `WrReq` command between nodes, if any of the two is necessary.

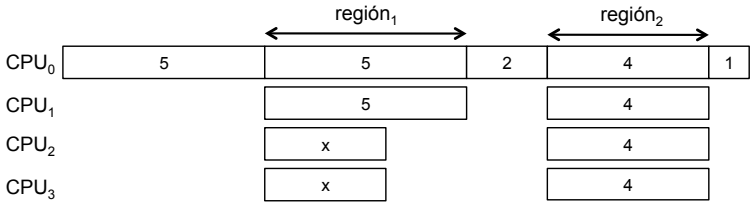
Solution: SInce both variables count and flag reside in the same line, there is a false sharing problem:

- the state of the line switches between M and I alternatively in the two caches; the line is only in S state when a read in one thread is done immediately after the write on the other.
- the state of the line in the directory is kept M all the time switching the list of sharers between 01 and 10, depending on which node has the line in M state in its cache; the line is only in S state in the directory when both caches have the copy also in S state.
- the coherence commands between NUMA nodes is shown in the table below, with the pattern WrRq-Dreply followed by Dreply-Fetch/invalidate that is symptom of the false sharing problem that is happening.

Time	Node with P0			Node with P1					
	Instruction	P0 cache status (1)	NUMA transaction (2)	Instruction	P1 cache status (1)	NUMA transaction (2)	Directory state (3)	Sharers list	
		flag/count			flag/count		flag / count	1	0
0	**count=0**	M	WrReq		--	Dreply	M	0	1
1	**flag=1**	M	--		--	--	M	0	1
2		S	Dreply	**ll flag**	S	Fetch	S	1	1
3	**count++**	M	WrReq		I	--	M	0	1
4		I	Dreply/--	**sc 1, flag**	M	Fetch/Invalidate	M	1	0
5		I	--	**ll flag**	M	--	M	1	0
6	**count++**	M	WrReq		I	Dreply	M	0	1
7		I	Dreply/--	**sc 1, flag**	M	Fetch/Invalidate	M	1	0
8		I	--	**ll flag**	M	--	M	1	0
9	**count++**	M	WrReq		I	Dreply	M	0	1
10		I	Dreply/--	**sc 1, flag**	M	Fetch/Invalidate	M	1	0
11		I	--	**ll flag**	M	--	M	1	0
12	**count++**	M	WrReq		I	Dreply	M	0	1
13		I	Dreply/--	**sc 1, flag**	M	Fetch/Invalidate	M	1	0
14	**flag=0**	M	WrReq		I	Dreply	M	0	1
15		S	--	**ll flag**	S	Fetch	S	1	1
16		I	--	**sc 1, flag**	M	Invalidate	M	1	0

PAR – Final Exam – Course 2017/18-Q1
January 19th, 2018

Problem 1 (2 points) The following diagram plots the timeline for the execution of a parallel application, with two parallel regions, on 4 processors:



Each box represents a burst executed on a processor and the number inside its execution time. This value is unknown for two of the bursts in $region_1$ (same value x for the two bursts, which can be smaller, equal or larger than 5). **Answer the following questions:**

1. Knowing that the speed-up that is achieved when the application is executed on 4 processors is $S_4 = 2.5$, obtain the two possible values for the duration of the unknown bursts (x).

   Solution: x can have two values: 4,25 or 8. If x is larger than 5, then $S_4 = (34 + 2 \times x)/(12 + x)$; from here you can obtain $x = 8$. If x is smaller than 5, then $S_4 = (34 + 2 \times x)/17$; from here you can obtain $x = 4, 25$.

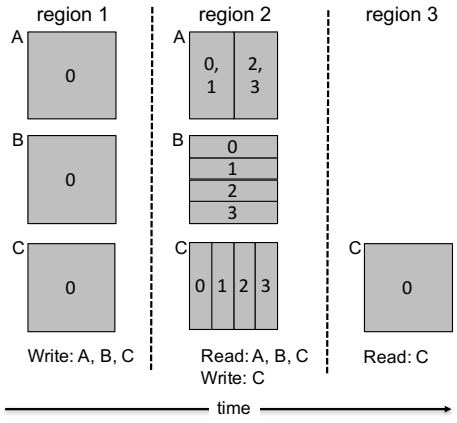
2. Knowing that the ideal speed-up $S_\infty = 5$ when the application is executed on infinite processors (assuming that, despite the imbalance in $region_1$, both parallel regions can be ideally decomposed), compute the value of the parallel fraction (ϕ) and the new value for x.

   Solution: For the ideal expression $S_\infty = 1/(1 - \phi)$ we obtain $\phi = 0.8$. From this value and the expression for $\phi = (26 + 2 \times x)/(34 + 2 \times x))$ we obtain $x = 3$.

3. In case that $x = 0$, meaning that it is not possible to distribute the workload in "región1" to more than two processors, what would be the ideal S_∞ that could be achieved for this application?

   Solution: In this case, $S_\infty = T_1/T_\infty = 34/13 = 2, 61$ (13 because only region 2 tends to 0 when the number of processors tend to ∞.

Problem 2 (2 points) Assume a program with 3 code regions executed one after the other (with a barrier between them), as shown in the following execution timeline. Regions 1 and 3 are executed sequentially (by processor 0) while region 2 is executed in parallel by 4 processors:



In this timeline we also show for each region how the three matrices used in the program (A, B and C, all of size $n \times n$) are accessed (read or write) and which part of the matrix is actually used (numbers inside indicate

processor number). Assumptions: 1) the data sharing model explained in class in which the overhead of a remote access is determined by $t_{comm} = t_s + m \times t_w$, being t_s the start-up time and t_w the time to transfer one element; 2) due to the barrier synchronisation between regions, regions 2 and 3 start their computation once all data needed is available; 3) in region 2 matrix C is first read and then written; 4) there is no data sharing between processors during the parallel execution of region 2; and 5) at a given moment, a processor can only execute one remote memory access, and can only serve a remote memory access from another processor. **We ask you to write** the expression for the data sharing (i.e. communication) overheads associated with the execution of the whole program, clearly indicating how each region contributes to that expression.

Solution: Remote accesses occur before starting Region 2 and Region 3. For region 2 all processors except processor 0 (i.e. processors 1 to 3) need to access processor 0 to get the part of matrices A, B and C that are needed. Since all of them are accessing the processor 0, and only one remote access can be served at any time, all accesses will be serialised. For matrix A, each of the three processors will have to access $N^2/2$ elements; for matrices B and C each of the three processors need to access $N^2/4$ elements.
For region 3, only processor 0 has to do remote access to collect matrix C. Since a processor can only request one remote access at any time, the accesses will be again serialised, each with $N^2/4$ elements.
So in total, 9 accesses of $N^2/4$ and 3 accesses of $N^2/2$, which contribute with an overhead of $t_{comm} = (9 \times (t_s + (N^2/4) \times t_w) + 3 \times (t_s + (N^2/2) \times t_w))$ time units.

Problem 3 (3 points) The following sequential code in C finds all positions in vector `DBin` in which a set of keys (contained in vector `keys`) appear. Positions where keys appear are stored in a new vector `DBout` (the order in `DBout` of the positions found is irrelevant).

```
#define DBsize 1048576
#define nkeys 16 // the number of processors can be larger than the number of keys

int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int i, k, counter[nkeys];

    getkeys(keys, nkeys);              // get keys
    init_DBin(DBin, DBsize);           // initialize elements in DBin
    clear_DBout(DBout, nkeys, DBsize); // initialize elements in DBout
    clear_counter(counter, nkeys);     // initialize counter to zero

    for (i = 0; i < DBsize; i++)
        for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) DBout[k][counter[k]++] = i;
}
```

1. Write a first *OpenMP* parallelisation that implements an **iterative task decomposition strategy** of the outermost loop `i`, making use of the **`for`** work–sharing construct, in which you minimise the serialisation introduced by the synchronisation that you may introduce. **Note:** you are not allowed to change the structure of the two loops.

   Solution: We could simply use `critical` to update `DBout` and `counter`, as follows:

   ```
   #pragma omp parallel for private(k)
   for (i = 0; i < DBsize; i++)
       for (k = 0; k < nkeys; k++)
               if (DBin[i] == keys[k]) {
                   #pragma omp critical
                   DBout[k][counter[k]++] = i;
               }
   ```

   However, a solution that minimises the serialisation introduced by that `critical` region should make use of looks, as many as possible keys, as follows:

```
omp_lock_t lock[nkeys];
for (k = 0; k < nkeys; k++) omp_init_lock(&lock[k]);
...
#pragma omp parallel for private(k)
for (i = 0; i < DBsize; i++)
    for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) {
                omp_set_lock(&lock[k]);
                DBout[k][counter[k]++] = i;
                omp_unset_lock(&lock[k]);
                #pragma omp flush
            }
...
for (k = 0; k < nkeys; k++) omp_destroy_lock(&lock[k]);
```

2. Write a second *OpenMP* parallelisation that also implements an **iterative task decomposition strategy**, but this time applied to the innermost loop `k` and making use of the **`taskloop`** directive (i.e. `task` or `for` are not allowed in your solution), in which you maximise the parallelism that can be exploited. **Notes:** 1) `taskloop` has an implicit `taskgroup` synchronisation that you can omit with the `nogroup` clause; 2) observe that the number of keys is not large when compared to the possible number of processors to use; and 3) you are not allowed to change the structure of the two loops.

   Solution: The first naive solution below is not appropriate:

```
#pragma omp parallel private(i)
#pragma omp single
for (i = 0; i < DBsize; i++)
    #pragma omp taskloop num_tasks(4)
    for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) {
                DBout[k][counter[k]++] = i;
            }
```

   In this solution we don't need to introduce any kind of synchronisation between tasks since, for each iteration `i`, each task updates different set of rows of `DBout` (the implicit `taskgroup` at the end of `taskloop` ensures this). However the correct solution should consider that the number of keys is not large compared to the number of threads executing the parallel region, so we create a small number of tasks in each `taskloop` construct, insufficient to feed all threads. For this reason, we simply remove the implicit task barrier at the end of the `taskloop` so that multiple `taskloop` constructs for different iterations of the `i` loop can be active, thus generating enough tasks to feed all the threads available. However, now multiple tasks may update the same set of rows of `DBout`, forcing us to protect the update of that variable, as before, with locks to minimise serialisation.

```
omp_lock_t lock[nkeys];
for (k = 0; k < nkeys; k++) omp_init_lock(&lock[k]);
...
#pragma omp parallel private(i)
#pragma omp single
for (i = 0; i < DBsize; i++)
    #pragma omp taskloop num_tasks(4) nogroup
    for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) {
                omp_set_lock(&lock[k]);
                DBout[k][counter[k]++] = i;
                omp_unset_lock(&lock[k]);
                #pragma omp flush
            }
```

```
    ...
    for (k = 0; k < nkeys; k++) omp_destroy_lock(&lock[k]);
```

3. Finally, write a third *OpenMP* parallelisation that implements a **`task`–based recursive divide–and–conquer decomposition strategy**, with the following requirements: 1) the recursion splits the input vector `DBin` in two almost identical halves, with a base case that corresponds to checking a single element of `DBin`; 2) uses a **cut–off strategy based on the size of the input vector**, so that tasks are only generated while that size is larger than `CUT_SIZE`; 3) only uses *OpenMP* pragmas and clauses for the implementation of the cut–off strategy; and 4) you have to use the synchronisation mechanism, if needed, that maximises the parallelism in the program.

   Solution:

```
#include <omp.h>

#define DBsize 1048576
#define nkeys 16 // the number of processors can be larger than the number of keys

#define CUT_SIZE ... /* Set the desired value here */

omp_lock_t lock[nkeys];

void find_keys( double *DBin, double *keys, double** DBout,
                unsigned int *counter, unsigned int ini, unsigned int end)
{

  unsigned int k;

  if ((end-ini) == 0) { /* Base case */
    for (k = 0; k < nkeys; k++)
      if (DBin[ini] == keys[k]) {
          omp_set_lock(&lock[k]);
          DBout[k][counter[k]++] = ini;
          omp_unset_lock(&lock[k]);
          #pragma omp flush
      }
  } else {
    unsigned int half = (end-ini+1) / 2;
    #pragma omp task final(half <= CUT_SIZE) mergeable
    find_keys(DBin, keys, DBout, counter, ini, ini+half-1);
    #pragma omp task final(half <= CUT_SIZE) mergeable
    find_keys(DBin, keys, DBout, counter, ini+half, end);
  }
}

int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int k, counter[nkeys];

    for (k = 0; k < nkeys; k++) omp_init_lock(&lock[k]);
    ...

    getkeys(keys, nkeys);              // get keys
    init_DBin(DBin, DBsize);           // initialize elements in DBin
    clear_DBout(DBout, nkeys, DBsize); // initialize elements in DBout
    clear_counter(counter, nkeys);     // initialize counter to zero

    #pragma omp parallel
```

```
    #pragma omp single
    find_keys(DBin, keys, DBout, counter, 0, DBsize-1);
    ...
    for (k = 0; k < nkeys; k++) omp_destroy_lock(&lock[k]);

    // Use results
}
```

Problem 4 (2 points) Assume the following simplified version for the previous code, which just finds in a vector `DBin` the first position where a single `key` appears:

```
#define DBsize 1048576

int main() {
  double key, DBin[DBsize];
  unsigned int i, position;

  getkey(&key);               // get keys
  init_DBin(DBin, DBsize);    // initialize elements in DBin
  position = DBsize;    // initialize position to the first position outside DBin

  for (i = 0; (i < DBsize) && (position == DBsize); i++)
    if (DBin[i] == key) position = i;
}
```

We propose a parallelization of the code using a strategy where the distribution of iterations to threads obeys a *geometric block data decomposition* for the input vector `DBin`, using `#pragma omp parallel` for the creation of threads:

```
int main() {
  // Variable declaration and initialization like the one in the sequential code
  #pragma omp parallel ...
  {
    numElements = ...
    for (i = 0; (i < numElements) && (position == DBsize); i++) {
      if ( ... == key) position = ...;
    }
  }
}
```

1. Complete the previous *OpenMP* code (without modifying the `for` loop that accesses the vector) in a way that the parallel version gives the same result as the sequential one. **Notes:** 1) if necessary, you can change the code in the innermost loop body; 2) for simplicity, you can assume that `DBsize` is multiple of the number of processors used to execute the parallel region.

   Solution:

   Since the loop goes from 0 to `numElements` we have to compute the index to `DBin` by adding the offset to the starting element on which each thread has to work. In addition, we need to protect the update of variable `position` ensuring that the first position where key appears is returned. We do so with a reduction where the reduction operator is `min`. Finally, variables `numElements` and `i` need to be private to each thread.

   ```
   int main() {
     // Variable declaration and initialization as in the sequential code
     ...
     #pragma omp parallel private(numElements, i) reduction(min:position)
     {
   ```

```
    int myId = omp_get_thread_num();
    int howMany = omp_get_num_threads();
    numElements = DBsize / howMany;
    int first = (numElements * myId);
    for (i = 0; (i < numElements) && (position == DBsize); i++) {
      if (DB[first+i] == key) position = first + i;
    }
  }
  ...
}
```

2. For the mentioned parallelization strategy, and assuming that the vector does not have repetitions of the same key, obtain the expression that determines the *speed-up* ($S = T_{seq}/T_{par}$) knowing that the execution time of an iteration in the original sequential loop and in the parallel loop are approximately the same (t_{iter}).

   Solution:

   We disregard the sequential code and focus only in the loop being parallelized.

   If the first occurrence of `key` is found in `position`, then the sequential execution time is

   $$T_{seq} = (position + 1) \times t_{iter}$$

   In order to compute the parallel time, we need to take into account the assumption that the vector does not have repetitions. This means that most threads will explore `numElements` positions of the vector, being `numElements = DBsize / omp_get_num_threads()`. The parallel execution time will be given by the time needed by the slowest thread(s). Then,

   $$T_{par} = (DBsize/omp\_get\_num\_threads()) \times t_{iter}$$

   The speedup is therefore:

   $$S = (position + 1)/(DBsize/omp\_get\_num\_threads())$$

   Note that the best possible speedup would be obtained if the key was found in the very last position of the vector. In that case, the speedup would equal the number of threads used in the parallel execution.

Problem 5 (1 point) Assume a shared–memory multiprocessor system composed of several *NUMAnodes* with directory-based coherence, each *NUMAnode* with several processors (cores) with its own cache memory and *Snoopy* to maintain coherence inside the node. Synchronisation is done using atomic instructions, such as `test_and_set`, between cores inside a *NUMAnode*; and using linked instructions, such as `load_linked store_conditional`, between *NUMAnodes*. We ask you to indicate if these sentences are **true or false**. **Important:** Each correct answer adds 0.1 points, each incorrect answer penalises 0.05 points.

1. Memory consistency is closely related to the protocols and mechanisms used to implement memory coherence, both inside and outside the *NUMAnode*, in such a way that it is not a problem if they are appropriately designed.

   False

2. When using *write-update* each write to a cache line provokes the update of all copies of the same line in the other caches inside the *NUMAnode*.

   True

3. Inside a *NUMAnode* using write–invalidate *MSI*, false sharing occurs when there exists several copies of the same memory line in the caches of the cores in M state.

   False

4. Inside a *NUMAnode* using write–invalidate *MSI*, only one modified copy of the same memory line in the system can exist, but more than one copy can exist in S state inside the system.

   True

5. The directory structure in a *NUMAnode* provides coherence information for the data allocated in the main memory of that *NUMAnode*.

   True

6. The directory structure in a *NUMAnode* provides information that allows one to find the data allocated in other *NUMAnodes*.

   False

7. The time to access a memory line from a core in the *owner NUMAnode* is always the same.

   False

8. Synchronisation instructions are used to ensure the appropriate data sharing and task ordering constraints and to avoid the appearance of false sharing.

   False

9. The atomic `test_and_set` instruction returns in a register the value that existed in the memory position where it is writing a 1.

   True

10. The `load_linked` instruction returns the value in a concrete memory position, ensuring that no other processor will be able to read again until the corresponding `store_conditional` has been executed.

    False

PAR – Final Exam – Course 2017/18-Q2

June 19^{th}, 2018

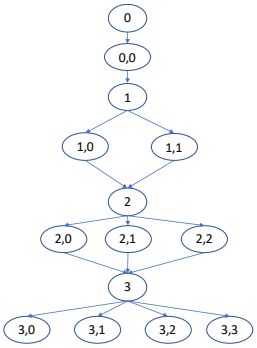
Problem 1 (4.0 points)
Given the following incomplete OpenMP parallel code:

```
#define MAX_ITERS XXX
void execute_task(int i, int t) {
   ...
}
void seq_func(int i) {
   ...
}
void main(int argc, char *argv[]) {
   int i;
   #pragma omp parallel
   #pragma omp single
   {
    int t;
    for (i=0; i<MAX_ITERS; i++) {
       #pragma omp task firstprivate(i)
       seq_func(i);
       for (t=0; t<=i; t++) {
          #pragma omp task firstprivate(i)
          execute_task(i, t);
       }
    }
   }
}
```

The required specifications for the parallel OpenMP implementation are the followings:

- `seq_func(i)` task should be executed before any `execute_task(i,t)` task, for any i.
- `execute_task(i,t)` task should be executed before `seq_func(`$i\prime$`)` task, for any $i\prime > i$.
- `execute_task(i,t)` tasks, for a given i, do not have dependences among them, and can be executed in parallel.
- The value of `MAX_ITERS` has been left undefined intentionally.

The following Task Dependence Graph represents the expected program behaviour when `MAX_ITERS` is set to 4. Each node represents the above defined tasks: nodes with a single label `i` representing `seq_func(i)` tasks and nodes labeled with a tuple `<i,t>` representing `execute_task(i,t)` tasks.



We ask you to complete the previous parallelization:

1. Complete the parallelization of the code above with the required synchronization to guarantee that instances of `seq_func(i)` and `execute_task(i,t)` are executed according to the required specifications. You can only add/remove OpenMP clauses and/or directive(s), not modify the C code.

   Possible Solution: We can include a taskwait directive after the for creating the `execute_task(`i, t`)` tasks, and remove the task definition of `seq_func(`i`)`.

```
void main(int argc,char *argv[])
{
   int i;
   #pragma omp parallel
   #pragma omp single
   {
    int t;
    for (i=0;i<MAX_ITERS;i++)
    {
       seq_func(i);
       for (t=0;t<=i;t++)
       {
          #pragma omp task firstprivate(i)
          execute_task(i,t);
       }
       #pragma omp taskwait
    }
   }
}
```

   Alternative Possible Solution: We can include depend clauses after task definition. No taskwait is necessary

```
void main(int argc,char *argv[])
{
   int i;
   #pragma omp parallel
   #pragma omp single
   {
    int t;
    for (i=0;i<MAX_ITERS;i++)
    {
       #pragma omp task depend(out:i) firstprivate(i)
       seq_func(i);
```

```
        for (t=0;t<=i;t++)
        {
           #pragma omp task depend(in:i) firstprivate(i)
           execute_task(i,t);
        }
     }
    }
}
```

2. Modify your previous solution to limit the number of `execute_task(i,t)` tasks generated in each execution of loop `t` to `MAX_TASKS_PER_LOOP` (already defined). Note that this limit is not a global limit. This control has to be done by only adding OpenMP clauses, and not modifying the C code.

   Solution: We must add the if clause

```
void main(int argc,char *argv[])
{
   int i;
   #pragma omp parallel
   #pragma omp single
   {
    int t;
    for (i=0;i<MAX_ITERS;i++)
    {
       seq_func(i);
       for (t=0;t<=i;t++)
       {
          #pragma omp task firstprivate(i) if (t<MAX_TASKS_PER_LOOP) mergeable
          execute_task(i,t);
       }
       #pragma omp taskwait
    }
   }
}
```

3. Compute T_1, T_∞ and *Parallelism* for the previous task dependency graph (for `MAX_ITERS=4`). Assume that the execution cost of each `seq_func(i)` task and each `execute_task(i,t)` task is 2 and 10 seconds, respectively. Also, consider the cost of creating tasks and other parts of the program negligible (not significant).

   Solution: T_1 is the cost of adding all the tasks, T_{inf} is the longest critical path when having infinite resources. The parallelism is the ratio between the T_1 and T_{inf}

   $T_1 = 4 * seq\_func\_cost + 10 * execute\_task = 4 * 2 + 10 * 10 = 108 seconds$

   $T_\infty = 4 * seq\_func\_cost + 4 * execute\_task = 4 * 2 + 4 * 10 = 48 seconds$

   $Parallelism = 108/48 = 2,25$

4. `For MAX_ITERS=4`, assuming the execution costs of previous exercise, calculate the parallel fraction of this application φ.

   Solution: φ is the percentage of time of the application that can be accelerated over the T_1

   $\varphi = 100/108 = 0.92$

Problem 2 (3.0 points)
Given the following OpenMP code:

```
// SIZE is left undefined on purpose
#define SIZE XXXX

unsigned int computation(unsigned int* X, unsigned int *Y, int n, int nthreads) {
   unsigned int sum=0;

   #pragma omp parallel for schedule(static, SIZE) reduction(+: sum) num_threads(nthreads)
   for (int i=0; i<n; i++) sum += X[i] * Y[i];

   return sum;
}
```

We ask you:

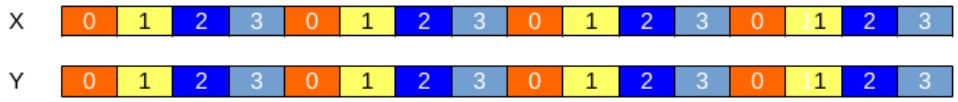
1. Assuming that `SIZE` has a value larger than 1 and smaller than `n`, indicate which data decomposition strategies are applied for each variable (`X`, `Y`, and `sum`) in the previous code (i.e. you should specify if the variable is decomposed as input, output or input/output data, if it is distributed or replicated among threads, appropriately naming the data decomposition that is applied, as seen in class: *block*, *cyclic*, *block-cyclic*, *replicated*, by rows, columns, blocks, etc.

   Solution:

   It is a block-cyclic geometric data decomposition of X and Y (inputs) . Variable `sum` (output) is locally replicated, and then reduced at the end.

2. Draw a picture (for `n` = 64, `SIZE` = 4 and `nthreads` = 4) to clarify the data decomposition, indicating which elements of the vectors are accessed by each thread. In the case of replication, specify that each thread has a copy.

   Solution:



Each color box corresponds to four elements of the vector
Number within a box indicates which thread takes care of the block of 4 elements

thread	0		1		2		3
replicated	sum		sum		sum		sum

3. Write an alternative implementation for the previous code that meets the following requirements:
   (a) It implements the same data decomposition strategies.
   (b) It cannot use any of the following constructs and clauses: `parallel for`, `for` and `reduction`.
   (c) It is assumed that `n`%`SIZE` = 0.
   (d) It must avoid any synchronisation within a for loop.
   (e) It should reduce any possible false sharing coherence problem. For this, you can assume that `unsigned int` size is 4 bytes and that exists a define in the `C` program for `CACHE_LINE_SIZE`, indicating the number of bytes of a cache line.

```
// SIZE is undefined in purpose
#define SIZE XXXX

unsigned int computation(unsigned int*X, unsigned int *Y, int n, int nthreads)
{
    unsigned int sum=0
    unsigned int sum_vector[nthreads][CACHE_LINE_SIZE/sizeof(unsigned int))];

    #pragma omp parallel num_threads(nthreads)
    {
       int i,ii;
       int myid=omp_get_thread_num();

       sum_vector[myid][0]=0;
       for (ii=myid*SIZE; ii<n; ii+=nthreads*SIZE)
          for(i=ii; i<ii+SIZE; i++)
            sum_vector[myid][0]+=X[i]*Y[i];

       #pragma omp atomic
          sum += sum_vector[myid][0];
    }
    return sum;
 }
```

Problem 3 (3.0 points)

Assume a NUMA (non-uniform memory architecture) multiprocessor system composed of 4 NUMA-nodes (NUMA-node 0 to 3). Each NUMA node has 16 GBytes of main memory and 2 cores. Each core has only one level of cache memory of 4 MBytes (with cache lines of 32 Bytes). The system includes all the necessary mechanisms (seen in class) to keep the memory and cache coherence inside a NUMA-node and between NUMA-nodes. Given the following OpenMP code executed on the previous multiprocessor system:

```
int x[N];
#pragma omp parallel for schedule(static, 1)
for (int i=0; i<N; i++)
    if (x[i]<0) x[i]=foo(i);
```

and assuming that: 1) vector `x` is allocated in main memory of NUMA $node_0$ and already initialized, 2) there are not copies of the vector elements in any cache; 3) the initial address of vector `x` is aligned with the start of a cache line; 4) the size of an `int` data type is 4 bytes; and 5) each thread executes on a different core; **we ask:**

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

   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 $= 4 * 2^{20}/2^5 = 2^{17}$, so the total number of bits per cache $= 2 * 2^{17}$ and the total number of bits per NUMA-node $= 2 * (2 * 2^{17}) = 2^{19}$ bits

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

Solution: We need 6 bits per memory line: 2 bits for the line state with possible values: M (Modified), S (Shared), U (Uncached) and 4 bits for the presence bits (one bit per NUMA-node). Total number of lines in main memory per NUMA-node is : $2^4 * 2^{30}/2^5 = 2^{29}$, so the total number of bits per NUMA-node $= 6 * 2^{29}$ bits

3. Describe, for each of the following operations, the sequence of processor commands, bus transactions within NUMA-nodes, transactions between NUMA-nodes (indicating source and destination of the transaction between hubs), the presence bit values of the Home Node Memory entry affected after the operation, and the state of the cache line/s and memory line affected after the operation, in order to keep cache and memory coherence. Assume that all caches are empty before the execution of the first command and recall that vector `x` is allocated in the memory of NUMA-node 0. Core 0 and 3 are located at Numa Node 0 and 1, respectively.

   (a) $core_0$ reads x[0]

   (b) $core_3$ reads x[3], considering the state after doing previous operation.

   (c) $core_3$ writes x[3], considering the state after doing previous operations.

   Solution:

   (a) $core_0$ reads x[0] : The variable `x[0]` is not loaded in the associated cache, so $core_0$ generates a $PrRd$ event which activates the Snoopy protocol, sending a $BusRd$ transaction on the bus. The local node is also the home node, and the line state indicate that the variable is uncached. A copy is loaded to local cache from the main memory of the Home Node. The memory line state in the directorythat holds the variable is updated to state Shared and the cache line state of $core_0$ is updated to Shared too.

   (b) $core_3$ reads x[3] : The variable `x[3]` is not loaded in the associated cache, so $core_3$ generates a $PrRd$ event which activates the Snoopy protocol, sending a $BusRd$ transaction on the bus. The Home node is not the local node, so the Hub sends a RdReq to the Home Node (NUMA-node 0). The variable `x[3]` belongs to the same memory line that `x[0]`, so the line has Shared state. The home hub sends a Dreply message to the local node with a copy of the value of the requested variable. The sharers list is updated (adding numa node 1) and also the presence bits (0011), and the cache line state associated to $core_3$ is updated to Shared.

   (c) $core_3$ writes x[3] : The variable `x[3]` is already loaded in the associated cache with Shared state, so $core_3$ generates a $PrWr$ event which activates the Snoopy protocol, sending a $BusRdX$ transaction on the bus. The local hub ask to modify the variable by sending a WrReq message to the home hub. The home hub invalidates (BusRx in the bus) the copy of the cache line in $core_0$ of the numa node 0 (home hub) and updates the information associated to this directory entry of the memory line: presence bits (0010) and state to Modified. The state of the line in cache associated to $core_3$ is updated to Modified.