PAR: Collection of Exercises

Eduard Ayguadé, Julita Corbalán, José Ramón Herrero
Daniel Jiménez and Gladys Utrera
Departament d'Arquitectura de Computadors

Course 2018-19 (Spring semester)

Index

1

Concurrency and parallelism

1. Assume we want to execute two different applications in our parallel machine with 4 processors: application *App1* is sequential; *App2* is parallelised defining 4 tasks, each task executing one fourth of the total application. The sequential time for both applications is 8 and 40 time units, respectively. Assuming: that *App1* starts its execution at time 4 and *App2* starts at time 0, draw a time line showing how they will be executed if the operating system:

   (a) Does not allow multiprogramming, i.e. only one application can be executed at the same time in the system.

   (b) Allows multiprogramming so that the system tries to have both applications running concurrently, each application making use of the number of processors is able to use.

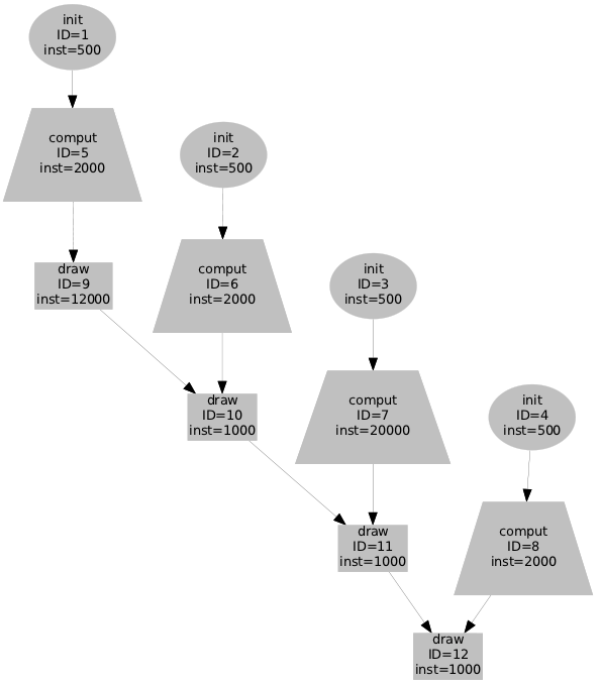
   (c) The same as in the second case, but now *App2* is parallelised defining 3 tasks, each task executing one third of the total application.

2. Assume we want to execute two different applications (*app1* and *app2*) in our parallel machine with p processors. Both applications can be (ideally) parallelised defining up to p tasks, each task executing one over p of the total application. The sequential time for *app1* and *app2* is 1200 and 2000 time units, respectively. Assuming that the operating system allows the multiprogrammed execution of parallel applications, for $p = 8$ decide the best allocation of processors to both applications in order to minimise the time the user has to wait for them to finish and giving the programmer the impression that both applications are running from the beginning.

2

Understanding parallelism

1. Given the following Task Dependence Graph (TDG), in which each node represents a task with a name, an instance number *ID* and the cost of executing it in terms of number of instructions *inst*. Edges represent dependences between pairs of tasks.



   We ask:

   (a) Which is the value for the T_1, T_∞ and parallelism metrics for this TDG?

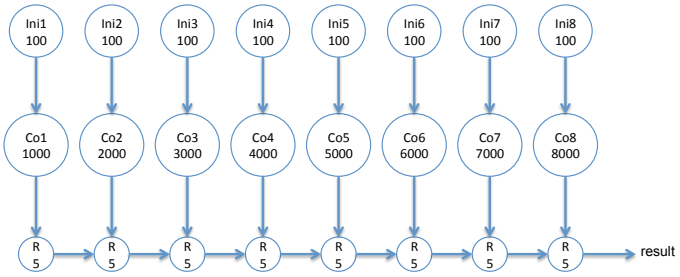
   (b) Which is the minimum number of processors P_{min} that are necessary to achieve the potential parallelism computed in the previous question?

2. We want to execute the tasks of the TDG in the previous problem on 4 processors, each processor executing a task sequence `init-compute-draw` (for example, tasks 1, 5 and 9 on processor 0, tasks 2, 6 and 10 on processor 1, ...).

   (a) Which is the speed-up that will be achieved when the tasks in this TDG are ideally executed on 4 processors?

   (b) Assume that we are able to balance the work among processors, which means that each node 1-4 weights 500, each node 5-8 weights 6500, and each node 9-12 weights 3750 instructions. Which is the speed-up that would be achieved with 4 processors, assuming the same task assignment as before?

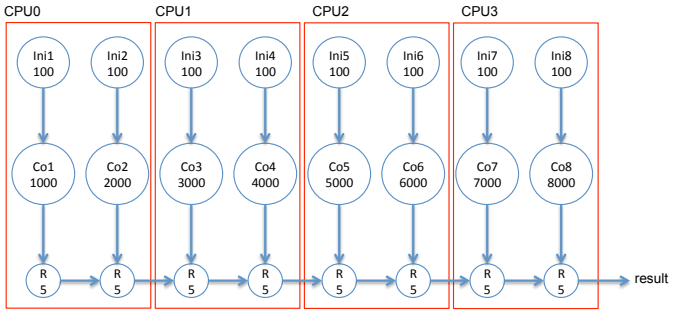
3. Given the following task dependence graph:



in which each node `IniX` runs for 100 time units, each node `CoY` runs for `Y*1000` time units and each node `R` runs for 5 time units. **We ask:**

(a) Calculate the values for T_1, T_∞ and the potential *Parallelism*.

(b) Calculate T_4 and the "speed–up" S_4 in an architecture with 4 processors if tasks are mapped (assigned) to processors as shown below:



(c) Determine the assignment of tasks to processors that would yield the best "speed-up" on 4 processors. Calculate such S_4.

4. Given the following C code in which two tasks have been identified using the *Tareador* API:

```
#define MAX 8
// initialization
for (outer = 0; outer < MAX; outer++) {
   tareador_start_task("for-initialize");
   for (inner = 0; inner < MAX; inner++)
      matrix[outer][inner] = inner;
   tareador_end_task("for-initialize");
   }

// computation
for (outer = 0; outer < MAX; outer++) {
   tareador_start_task("for-compute");
   for (inner = 0; inner <= outer; inner++)
      matrix[outer][inner] = matrix[outer][inner] + foo(outer,inner);
   tareador_end_task("for-compute");
   }
```

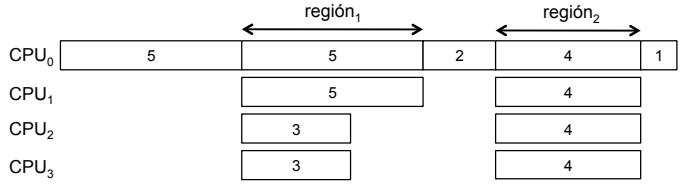
Assuming that: 1) in the initialization loop the execution of each iteration of the internal loop lasts 10 cycles; 2) in the computation loop the execution of each iteration of the internal loop lasts 100 cycles; and 3) the execution of the `foo` function does not cause any kind of dependence. **We ask**:

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

(b) Calculate the values for T_1 and T_∞ as well as the potential parallelism.

(c) Calculate which is the best value for the "speed-up" on 4 processors (S_4), indicating which would be the proper task mapping (assignment) to processors to achieve it.

5. Given the following time diagram for the execution of a parallel application on 4 processors:



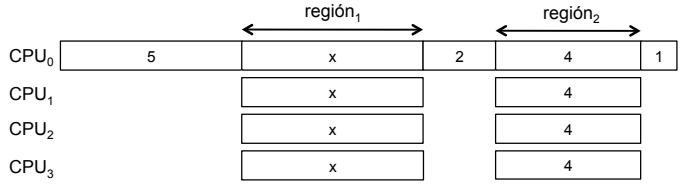
The application has two parallel regions and three sequential parts. In this diagram the numbers in the boxes represent the execution times of the different application bursts. **We ask**:

(a) Calculate the "speed-up" S_4 that is achieved in the execution with 4 processors.

(b) Calculate the parallel fraction (ϕ) for the application that is represented in the time diagram above. Which is the "speed-up" that could be achieved when using infinite processors (S_∞, assuming that the two parallel regions can be ideally decomposed into the ∞)?

(c) Imagine it is possible to remove the load imbalance that exists in "región1" at the expenses of adding an "overhead" that is proportional to the number of processors ($0.05 \times p$). Which would be the expression for S_p in this case?.

6. Given the following incomplete time diagram for the execution of a parallel application on 4 processors:



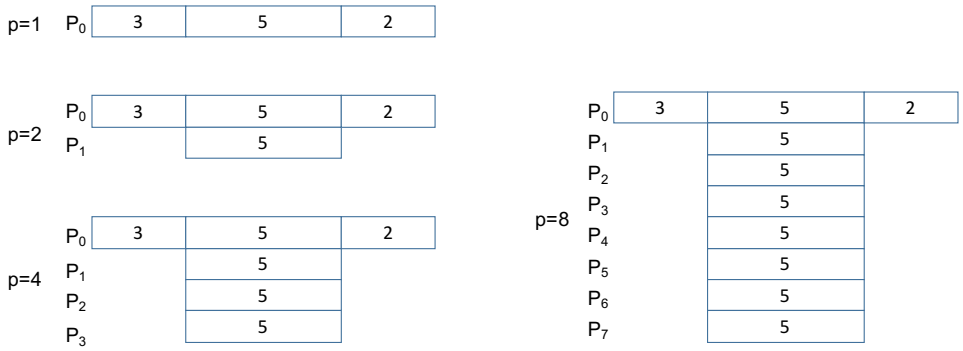
Numbers inside the boxes represent the execution time for the different execution bursts, being this value (x) unknown for the bursts in $region_1$. Knowing that a "speed-up" of 9 could be achieved when the application makes use of infinite processors ($S_\infty = 9$, assuming that the parallel regions can be decomposed into infinity ∞), **we ask:**

(a) What is the parallel fraction (ϕ) for the application represented in the time diagram above?

(b) Which is the "speedup" that is achieved in the execution with 4 processors (S_4)?

(c) Which is the value x in $region_1$?

7. Assume the following execution timelines to analyze the **weak scaling efficiency** for a parallel application, each timeline for a problem size proportional to the number of processors p:



(a) We define the parallel fraction of the application as $\varphi = T_{par} \div (T_{seq} + T_{par})$ when the application is executed with the original problem size (i.e. the one used when p=1). In the expression T_{par} is the time spent on code that can be parallelized and T_{seq} is the time spent

on code that cannot be parallelized. Compute the value of φ for the application according to this definition.

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

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

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

8. Given the following code in C:

```
int it_dot_product(int *X, int *Y, int n) {
   int i, sum=0;
   for (i=0; i<n; i++) sum += X[i]*Y[i];
   return sum;
}

int rec_dot_product(int *X, int *Y, int n) {
   int ndiv4 = n/4, sum1, sum2, sum3, sum4;

   if (n<=4) return it_dot_product(X,Y,n);
   sum1 = rec_dot_product(X, Y, ndiv4);
   sum2 = rec_dot_product(X+ndiv4, Y+ndiv4, ndiv4);
   sum3 = rec_dot_product(X+2*ndiv4, Y+2*ndiv4, ndiv4);
   sum4 = rec_dot_product(X+3*ndiv4, Y+3*ndiv4, n-3*ndiv4);
   return sum1+sum2+sum3+sum4;
}

void main() {
   int sum, X[N], Y[N];
   ...
   sum = rec_dot_product(X,Y,N);
   ...
}
```

Assume that there is not a parallelization strategy already defined for this application and that its sequential execution time is $T_{seq} = 6$ time units. Answer the following questions:

(a) Which should be the parallel fraction (ϕ) in order to achieve $S_\infty = 100$?

(b) Which should be the parallel fraction (ϕ) if we are looking for an $S_\infty = 100$ but we introduce a constant overhead of 0.01 time units due to task creation?

Independently of the previous questions, now assume a parallelization strategy is defined: each function invocation (i.e. call to `rec_dot_product` or `it_dot_product`) is a task. Assuming $N = 64$, answer the following questions:

(a) Draw the task data dependency graph (TDG). How many `rec_dot_product` tasks are created? How many `it_dot_product` tasks are created?

(b) Assume the execution time for `rec_dot_product` is t_{rec}. This cost includes the if statement, addition of results sum_i and the creation of the tasks corresponding to the 4 recursive calls and the base case. Each `it_dot_product` task has an execution cost equals to the number of iterations of the loop multiplied by t_c. Compute T_1, T_∞ and the potential *Parallelism*.

(c) What is the minimum number of processors P_{min} that are required in order to achieve the potential parallelism obtained in the previous question?

9. Given the following code computing matrix `u`, in which the k loop iterates from 1 to i:

```
for (i = 1; i < N-1; i++)
    for (k = 1; k <= i; k++) {
        tmp = 0.3 * u[i+1][k+1] + 0.7 * u[i-1][k-1];
        u[i][k] = (tmp * tmp) / 4;
    }
```

Assume that the execution of the innermost loop body takes one time unit. Answer the following questions:

(a) For $N = 16$, draw the iteration space that is traversed during the execution of the loop nest, shadowing the cells that correspond to iterations for which the loop body is actually executed.

(b) Draw the task dependence graph (TDG) if the iteration space above is decomposed into tasks according to the following block decomposition (shown for $N = 16$ and $B = 4$).

(c) For the TDG you obtained, compute the values for T_1 and T_∞.

(d) If these tasks are mapped to a machine with $P = B$ processors, so that each row of tasks is mapped to a different processor, and assuming that N is very large compared to P ($N >> P$). We ask to draw a temporal diagram with the parallel execution and obtain the expression for T_p as a function of N.

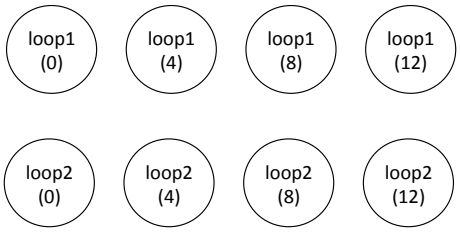
(e) If the cost for each synchronization between two processors takes t_{sync} time units, obtain the synchronization overhead that would be added to the previous expression for T_p.

10. Given the following two loops in a C program instrumented with *Tareador*:

```
#define N 16
#define BS 4
void main() {
char stringMessage[16];
tareador_ON();
for (int ii = 0; ii < N; ii=ii+BS) {
    sprintf(stringMessage,"loop1(%d)",ii);
    tareador_start_task(stringMessage);
    for (int i = ii;  i< ii+BS; i++)                  // BS perfectly divides N
        for (int j = 0; j < N; j++)
            b[i][j] = foo(a[i][j]);
    tareador_end_task(stringMessage);
    }
for (int ii = 0; ii < N; ii=ii+BS) {
    sprintf(stringMessage,"loop2(%d)",ii);
    tareador_start_task(stringMessage);
    for (int i = max(1,ii); i < min(ii+BS, N-1); i++) // min y max to ensure the access
                                                      // within the range of matrix b
        for (int j = 0; j < N; j++)
            c[i][j] = goo(b[i][j], b[i-1][j], b[i+1][j]);
    tareador_end_task(stringMessage);
    }
tareador_OFF();
}
```

We ask:

(a) Complete the following task dependence graph (TDG), assuming functions `foo` and `goo` only access to their arguments and do not update any other global variable.



(b) 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 two following assignments of tasks to processors:

Task	Assignment 1	Assignment 2
loop1(0)	0	0
loop1(4)	1	1
loop1(8)	2	2
loop1(12)	3	3
loop2(0)	0	0
loop2(4)	1	0
loop2(8)	2	0
loop2(12)	3	0

You can assume: 1) a distributed-memory architecture with 4 processors; 2) matrices `a`, `b` and `c` are initially distributed by rows (N/BS consecutive rows per processor); 3) once the second loop is finished, you don't need the return matrices to their original distribution; 4) 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 5) the execution time for a single iteration of the innermost loop body takes t_c.

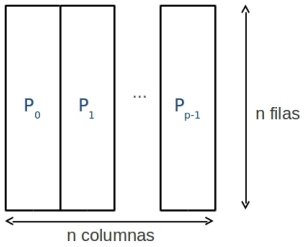
11. For each of these two loops:

```
1) for (i=1; i<n; i++)                          2) for (i=1; i<n; i++)
    for (j=1; j<n; j++) {                           for (j=1; j<n; j++) {
     B[i][j] = A[i][j-1]+A[i-1][j]+A[i][j];          A[i][j] = A[i][j-1]+A[i-1][j]+A[i][j];
    }                                               }
```

answer the following questions:

(a) Assuming that we define a task as the body of the innermost loop and that its execution time is t_c, calculate T_1 and T_∞.

(b) Assuming that we are using a distributed memory machine and that we are doing a distribution of the matrices A and B by columns, as shown below:



- Determine, for each code, which is the most suitable parallelization strategy for the data distribution indicated above.
- Calculate T_P (including both the computation time and data sharing overhead) for each code and the parallelization strategy you proposed, assuming that the cost for a message of B elements is $t_s + B \times t_w$.

12. Assume a distributed memory machine and a message passing model where the message cost is $t_{comm} = t_s + m \times t_w$, being t_s the "start-up" time and t_w the transfer time of a word. Answer the following questions related to the following code:

```
void smith-waterman(int h[N+1][N+1], char a[N], char b[N], int sim[20][20]) {
    int i,j;
    int diag, down, right;

    for (i=0;i<=N;i++) {
      h[0][i]=0;
      h[i][0]=0;
    }
    for (i=1;i<=N;i++)
      for (j=1;j<=N;j++) {
        diag    = h[i-1][j-1] + sim[a[i-1]][b[j-1]];
        down    = h[i-1][j] + 4;
        right   = h[i][j-1] + 4;
        h[i][j] =  MAX4(diag,down,right,0);
      }
}
```

(a) Draw matrix `h` and the data dependences between the computation of its elements (i.e. for each element indicate which elements of the same matrix need to be computed before).

(b) Assuming the following definitions for tasks: 1) each iteration of the first initializing to zero loop (cost of an iteration $t_{zero} = 1$ time unit) and 2) each iteration of the most internal loop in the nested loop (cost of an iteration $t_c = 100$ time units):

- Draw the task dependence graph (TDG) between tasks for $N = 4$.
- Obtain the expressions for T_1 and T_∞ as a function of N. How many processors P_{min} are needed to guarantee that T_∞ can be reached?

(c) Assuming that the vector `a`, the vector `b`, and the matrix `sim` have been replicated in P processors, and the matrix `h` is distributed by rows between these P processors ($\frac{N}{P}$ consecutive rows per processors). Which would be the most appropriate definition for the tasks for the computation loop in the program, with the objective of reducing T_p? Obtain the expression for T_p for that loop. You can assume that $N >> P$.

13. We want to find the expression that determines the parallel execution time in p processors (T_p) for the following loop:

```
for (i=1; i<n; i++) {
    for (k=0; k<n-1; k++) {
        u[i][k] = 0.8*u[i-1][k] + 0.5*u[i][k+1] - 0.2*u[i][k];
    }
}
```

using the data sharing model explained in class based on the distributed memory architecture with message passing: the access time to remote data is determined by $t_{comm} = t_s + m \times t_w$, being t_s and t_w the "start-up" and sending time of an element, respectively, and being m the size of the message. The execution time of the iteration of the body of the most internal loop is t_c.

Two different data distributions are considered: *Column distribution* (the matrix `u` is distributed so that each processor has n/p consecutive columns) and *Row distribution* (the matrix `u` is distributed so that each processor has n/p consecutive rows). For each data distribution **we ask** to 1) define the most appropriate definition of a task; and 2) complete the following table with the different contributions to T_p.

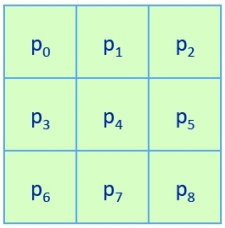
		Column distribution	Row distribution
Task definition			
Initial remote accesses	Total number of messages		
	Size of each message		
	Contribution to T_p		
Parallel computation	Total number of tasks		
	Size of each task		
	Contribution to T_p		
Remote accesses during parallel execution	Total number of messages		
	Size of each message		
	Contribution to T_p		

14. Assume a distributed memory architecture with message passing, where each message of m elements incurs an overhead of $t_{comm} = t_s + m \times t_w$.

(a) Given the following loop:

```
for (i=1; i<n-1; i++) {
   for (k=1; k<n-1; k++) {
     tmp = u[i+1][k] + u[i-1][k] + u[i][k+1] + u[i][k-1] - 4*u[i][k];
     f[i][k] = tmp/4;
   }
}
```

and a 2D block distribution for matrices u and f, as shown for the following case with 9 processors (3^2):



Assume 1) a task is defined as a block of $N \div P$ by $N \div P$ consecutive iterations of the `i` and `k` loop, respectively; 2) t_c the execution time of the innermost loop body; 3) P^2 the number of processors; and 4) n the size of the matrix (n rows by n columns), being n large compared to P. With this 2D block distribution, each processor will execute one task. Answer the following questions:

i. Indicate the data that each processor should receive from other processors, and when it has to receive such data.

ii. Each processor could start its computation when it has all the necessary data in its local memory but we will consider that the computation starts once all the communications in each step are completed. Obtain the model of execution time (computation and communication).

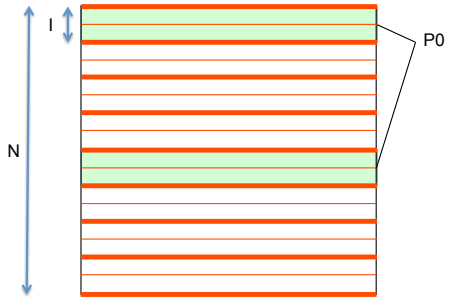
(b) Continuing with the same assumptions above and same task definition, consider next the following loop:

```
for (i=1; i<n-1; i++) {
   for (k=1; k<n-1; k++) {
     tmp = u[i+1][k] + u[i-1][k] + u[i][k+1] + u[i][k-1] - 4*u[i][k];
     u[i][k] = tmp/4;
   }
}
```

i. Indicate the data that each processor should receive from other processors, and when it has to receive such data.

ii. Each processor could start its computation when it has all the necessary data in its local memory but we will consider that the computation starts once all the communications in each step are completed. Obtain the model of execution time (computation and communication).

15. Given the following geometrical data decomposition for the matrices `a` and `b` on P processors (blocks of I consecutive rows separated by $I \times P$):



and the access pattern to the elements of the matrices caused by the following loop:

```
for (i=1; i<N-1; i++)
    for (k=1; k<N-1; k++) {
        tmp = a[i-1][k] + a[i][k-1] - 2*b[i-1][k-1];
        a[i][k] = tmp/4;
    }
```

Tasks are defined as blocks of $I \times B$ consecutive iterations of the `i` and `k` loops, respectively, where B is a divisor of N and bigger than the number of processors P. Obtain the expression for the data sharing overhead on a distributed memory architecture (t_s: start-up time and t_e: transfer time for a single element) according to the number of processors, P, and the values of I and B (assume that the division $N \div P$ is an integer).

3

Task decomposition

1. For each one of the two loops below:

   - Loop 1:
     ```
     #pragma omp parallel for schedule(dynamic, p) num_threads(p)
     for (i = 0; i < n; i++)
         b[i] = foo(a[i]);
     ```
   - Loop 2:
     ```
     #pragma omp parallel for schedule(static, p) num_threads(p)
     for (i = 0; i < n; i++)
         b[i] = foo(a[i]);
     ```

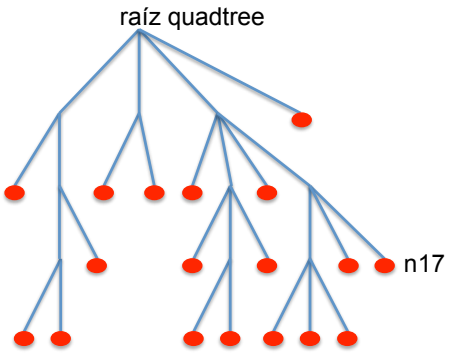
   Write an equivalent version (equivalent in terms of number of chunks and number of iterations per chunk, but not in terms of assignment of chunks to processors and execution order) in which you ONLY make use of the following OpenMP constructs: `parallel`, `single` and `taskloop`. Note: assume that `p` exactly divides `n`.

2. Given the following code:

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

   Indicate two ways to reduce the barrier synchronization `overheads` that also help to improve the performance of the code.

3. Suponer una arquitectura con ∞ procesadores y un overhead de creación de tareas de t unidades de tiempo. Dado el siguiente árbol *quadtree*



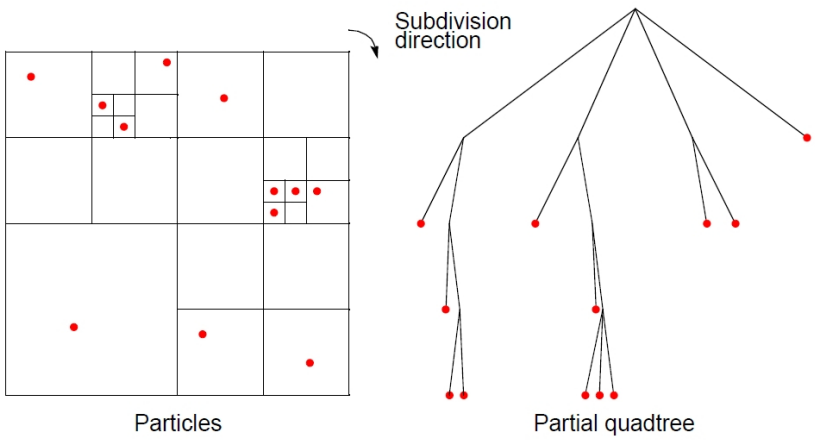
que representa la ejecución de una descomposición recursiva *Tree* en tareas totalmente independientes, en la que cada rama de un nodo interno del árbol corresponde a la creación de una tarea y en la que cada hoja del árbol corresponde con la ejecución de una función `doComp`, se pide calcular a partir de que instante de tiempo podrá iniciarse la ejecución del nodo `n17`, sabiendo que la ejecución de `doComp` tarda d unidades de tiempo y el resto de código tarda un tiempo despreciable.

Repetir el problema para el caso en que el àrbol represente una descomposición recursiva *Leaf* en tareas totalmente independientes, en la que cada rama de un nodo interno del árbol corresponda a la invocación recursiva y en la que cada hoja del árbol corresponda a la creación de una tarea que ejecuta la función `doComp`.

4. Given the following OpenMP parallelization for the traverse of a *quadtree* tree:

```
void traverse(TreeNode* subTree, int d) {
  int i, j;
  if(subTree) {
     if(!subTree->isLeaf) {
        if(d<3)
           #pragma omp parallel for collapse(2) num_threads(4)
           for(i=0; i<2; i++)
              for(j=0; j<2; j++)
                 traverse(subTree->quadrant[i][j], d+1);
        else
           for(i=0; i<2; i++)
              for(j=0; j<2; j++)
                 traverse(subTree->quadrant[i][j], d+1);
     }
     else // subtree is a leaf
        doComputation(subTree);
  }
  return;
}
void main() {
      ...
      traverse(root, 0);
      ...
}
```

where the `collapse(2)` clause indicates that we want to parallelize the iteration in the two following `for` loops. And given the following tree (pointed by `root`) on which the function will be called:



We ask:

(a) How many OpenMP threads are created during the complete tree traversal? How many of them execute `doComputation`?

(b) Assuming that the execution of `doComputation` uses 25 time units, the creation of threads in a parallel uses 4 time units and that we neglect the execution time of the rest of the code, calculate the time of parallel execution for an infinite number of processors in the system.

(c) Write a new parallelization based on OpenMP task decomposition in which the task creation is limited at the same level as recursivity.

(d) How many threads and OpenMP tasks are created during the complete tree traversal?

(e) Assuming that the creation of a task uses 2 time units, calculate the parallel execution time for an infinite number of processors in the system.

5. Given the following sequential code to calculate the sum of all the elements of a vector:

```
int sum_vector(int *X, int n) {
   int i, sum = 0;
   for (i=0; i< n; i++) sum += X[i];
   return sum;
}
void main() {
   sum = sum_vector(v,N);
}
```

(a) Write a parallel version in OpenMP implementing a linear or iterative task decomposition parallelisation strategy making use of *work–sharing* constructs. Your solution should not use explicit synchronizations during the execution of the `for` loop.

(b) Write a new parallel version that implements the same decomposition strategy and synchronization constraints, but now making use of the OpenMP tasking model, either with `task` or `taskloop`.

(c) Write a sequential recursive version of the `sum_vector` function (`recursive_sum_vector`) with the objective of achieving a $T_\infty = \log(n)$ in its parallelization (assuming that the cost of an iteration is 1). Indicate, without writing the sequential code, how would you change the code in order to obtain a $T_\infty = log_4(n)$. Justify your answers.

(d) Write a parallel version of the new recursive program that uses a "divide and conquer" task decomposition in which T_p (execution time with p processors) tends to $T_\infty = \log(n)$.

6. Given the following C function headers and OpenMP incomplete code to sort a vector `v` of `N` elements, based on a divide and conquer task decomposition:

```
void quicksort_base(int * v, int n); // sorts a vector v of n elements
int find_pivot(int *v, int n); // finds the index n of the pivot

void quicksort(int *v, int n) {
   int index;
   if (n<N_BASE)
       #pragma omp task
       quicksort_base(v,n);
   else {
     index = find_pivot(v,n);
     quicksort(v,index);
     quicksort(&v[index], n-index);
   }
}
void main() {
   quicksort(v,N);
}
```

We ask:

(a) Indicate at least two reasons why such a decomposition does not obtain a good speed-up, with respect to the sequential version (efficiency lower than 1), when executed with `OMP_NUM_THREADS=4`.

(b) Write an alternative version that obtains a better speed-up (and efficiency), without having to worry excessively about the task creation overheads.

(c) Modify the previous code with the objective of reducing the parallelization overheads, controlling the generation of tasks in two different ways, being the following: a) depth of the recursivity tree (`MAX_DEPTH`); b) size of the vector to sort (`VECTOR_SIZE`).

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

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

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

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

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

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

(a) Add the required pragmas to complete the parallelization of the `CXY` function following a tree strategy and briefly justify the pragmas inserted.

(b) Do you need to guarantee any dependence and/or protect any race condition between the execution of `CXY` and `CKJ`?. And during the execution of function `CKJ`? If affirmative in any of the two cases, how are you guaranteeing the dependence?

(c) Propose a modification in the code to introduce a cut-off based on the depth of the three, defining a maximum of 4 levels. We will consider positively the introduction of the minimum number of changes in the code.

(d) Propose an alternative parallelization for the CKJ function based on the use of `task` achieving an equivalent work distribution.

8. Given the following sequential code to count the number of times a value `key` appears in vector `a`

```
#define N 131072
long count_key(long Nlen, long *a, long key) {
    long count = 0;

    for (int i=0; i<Nlen; i++)
        if(a[i]==key) count++;
    return count;
}

int main() {
   long a[N], key = 42, nkey=0;
   // fill the array and make sure it has a few instances of the key
   for (long i=0;i<N;i++) a[i] = random()%N;
      a[N%43]=key; a[N%73]=key; a[N%3]=key;

   nkey = count_key(N, a, key);     // count key sequentially
   nkey = count_iter(N, a, key);    // count key in a using an iterative decomposition
   nkey = count_recur(N, a, key);   // count key in a with divide and conquer
}
```

(a) Write a parallel OpenMP version using an iterative task decomposition (`count_iter`)

(b) Write a parallel OpenMP version using a recursive task decomposition (divide and conquer, `count_recur`). The implementation should take into account the overhead due to task creation, limiting their creation once a certain level in the recursive tree is reached

9. Assume the following sequential C code that finds in a vector `DB` the first position in which a specific `key` appears:

```
int main() {
  unsigned long DBsize=(1<<24), position;
  double key, * DB;
  DB = (double *) malloc(sizeof(double) * DBsize);
  position = DBsize;
  // initialize elements in DB
  // read key to look for

  for (unsigned long i = 0; (i < DBsize) && (position == DBsize); i++)
      if (DB[i] == key) position = i;
  // write first position, if found
}
```

(a) In a first proposal of code parallelization using *OpenMP* we propose to add `#pragma omp parallel for` before the `for` loop. Which compilation error is going to return us the *OpenMP* compiler?

(b) Even assuming that we do not get the compilation error, identify a conceptual error in the adopted parallelization strategy and that leads to an incorrect result. What do we need to avoid the error? (It's not necessary to implement the solution now, we will ask for it later in the problem)

In a second attempt, we suggest to code a recursive version of the original algorithm that allows the application of a divide and conquer recursive parallelization strategy. We ask:

(c) Rewrite the sequential code in order to allow the application of a divide and conquer parallelization strategy.

(d) Add the needed *OpenMP* constructions to make the code execute in parallel, bearing in mind the overhead that the parallelization could introduce.

Finally we want to give a new chance to implement an iterative linear task decomposition making use of the OpenMP work–sharing model:

```
...
#pragma omp parallel
#pragma omp for
for (unsigned long i = 0; i < DBsize; i++) {
    ...
    if (DB[i] == key) position = i;
    ...
}
...
```

and the following two OpenMP pragmas (not explained in the slides) to force threads to leave the execution of the work-sharing as soon as they detect that the solution has been found:

- `#pragma omp cancel for`: this directive activates the cancellation of the innermost enclosing for region. The thread that finds the directive finishes the execution of the loop; the other threads continue their execution as normal.
- `#pragma omp cancellation point for`: introduces a point to check if cancellation has been activated. When found by a thread, if the innermost enclosing for region has been already cancelled, then it finishes the execution of the loop.

We ask you to complete the parallel code above making use of these two directives. Which would be the effect of using a different `schedule` than the default `static`?

10. Parallelise the following sequential code using *task dependences* in OpenMP, following a producer-consumer execution model. Producer and consumer code should be in two different tasks.

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

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

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

11. SAXPY stands for *Single-precision A times X Plus Y*. It is a function in the standard Basic Linear Algebra Subroutines (BLAS) library. SAXPY is a combination of scalar multiplication and vector addition. It takes as input two vectors of 32-bit floats X and Y with N elements each, and a scalar value A. It multiplies each element $X[i]$ by A and adds the result to $Y[i]$, as shown below:

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

Given the following main program that makes use of `saxpy`, we want to achieve the asynchronous execution of the initialization loop, the two SAXPY calls and the two loops writing the result vectors to files. Write a parallel version making use of the OpenMP task construct and task dependencies, creating the appropriate parallel context.

```
int main() {
        int N = 1 << 20;    /* 1 million floats */
        float  *fx = (float *) malloc(N * sizeof(float));
        float  *fy = (float *) malloc(N * sizeof(float));
        FILE *fpx = fopen("fx.out", "w");
        FILE *fpy = fopen("fy.out", "w");
        /* simple initialization just for testing */
        for (int k = 0; k < N; ++k)
                fx[k] = 2.0f + (float) k;
        for (int k = 0; k < N; ++k)
                fy[k] = 1.0f + (float) k;
        /* Run SAXPY TWICE */
        saxpy(N, 3.0f, fx, fy);
        saxpy(N, 5.0f, fy, fx);

        /* Save results */
        for (int k = 0; k < N; ++k)
                fprintf(fpx, " %f ", fx[k]);
        for (int k = 0; k < N; ++k)
                fprintf(fpy, " %f ", fy[k]);

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

12. We ask to parallelize the computation of the histogram of values appearing on a vector. The histogram is another vector in which each position counts the number of elements in the input vector that are in a certain value range. The following program shows a possible sequential implementation for computing the histogram (vector `frequency`) of the input vector `numbers`:

```
#define MAX_ELEM 1024*1024
#define HIST_SIZE 250
unsigned int numbers[MAX_ELEM];
unsigned int frequency[HIST_SIZE];

void ReadNumbers (int * input, int * size);
void FindBounds(int * input, int size, int * min, int * max) {
for (int i=0; i<size; i++)
      if (input[i]>(*max)) (*max)=input[i];

for (int i=0; i<size; i++)
      if (input[i]<(*min)) (*min)=input[i];
}

void FindFrequency(int * input, int size , int * histogram, int min, int max) {
int tmp;
for (int i=0; i<size; i++) {
      tmp = (input[i] - min) * (HIST_SIZE / (max - min - 1));
      histogram[tmp]++;
      }
}
```

```
void DrawHistogram(int * histogram, int minimum, int maximum);
void main() {
    int num_elem, max, min;

    ReadNumbers(numbers, &num_elem); // read input numbers
    max=min=numbers[0];
    FindBounds(numbers, num_elem, &min, &max); // returns the upper and lower
                                               // values for the histogram
    FindFrequency(numbers, num_elem, frequency, min, max); // compute histogram
    DrawHistogram(frequency, min, max); // print the histogram
}
```

We ask:

(a) Write a parallel version for function `FindBounds` ONLY using one OpenMP `#pragma` line, in such a way that you minimize the possible parallel execution and synchronization overheads.

(b) Write an alternative implementation for the following parallel version of function `FindFrequency`:

```
#pragma omp parallel for
for (int i=0; i<size; i++) {
      #pragma omp critical
            {
            tmp = (input[i] - min) * (HIST_SIZE / (max - min - 1));
            histogram[tmp]++;
            }
      }
```

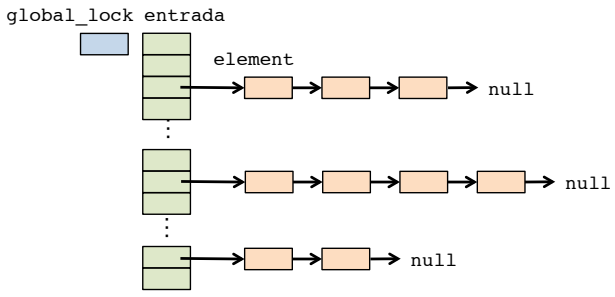
in which you improve the parallelism that is achieved, JUST using OpenMP `pragma`s.

(c) Write another alternative implementation for the same function based on the use of OpenMP `lock`s, in which you maximize the parallelism in the update of the histogram.

(d) Finally, write a parallel version for function `FindFrequency` based on a *divide_and_conquer* recursive strategy, making use of the synchronization and cut-off strategies that you consider the most appropriate.

13. Assume a hash table implemented as a vector of chained lists, such as shown in the next figure and type definition:

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

typedef struct {
    omp_lock_t global_lock;
    element *  entrada[SIZE_TABLE];
} HashTable;

HashTable table;
```



Inside each list, the elements are stored ordered by their `data` field value. Also assume the following code snippet to insert elements of the `ToInsert` vector of size `num_elem` into the mentioned hash table:

```
#define MAX_ELEM 1024
int main() {
    int ToInsert[MAX_ELEM], num_elem, index;
    ...
    omp_init_lock(&table.global_lock);
    #pragma omp parallel for private(index) schedule(static)
    for (i = 0; i < num_elem; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
```

```
        omp_set_lock (&table.global_lock);
        insert_elem (ToInsert[i], index);
        omp_unset_lock (&table.global_lock);
    }
    omp_destroy_lock(&table.global_lock);
    ...
}
```

where `hash_function` function returns the entry of the table (between 0 and `SIZE_TABLE`-1 where a specific element has to be inserted and the `insert_elem` function inserts the mentioned element in the corresponding position inside the chained list pointed by the `index` entry of the `HashTable`.

(a) Given the sequence `index`={5,10,14,10,25,25,10,8} returned by `hash_function` for a `ToInsert` vector with `num_elem`=8 elements, draw in a timing diagram the parallel execution with 4 threads, assuming that the `hash_function` function lasts 2 time units, `insert_elem` lasts 5 time units and the "set" and "unset" lock functions last 1 time unit each. The rest of the operations can be considered to use a negligible time.

(b) Modify the previous data structure and code to allow parallel insertions in different entries of the `HashTable`.

(c) For the same sequence of `index` values used previously, draw again the timing diagram of the parallel execution with 4 threads for the implementation proposed in section b). How would that diagram change if the loop planning was `(static,1)`? And `(dynamic,1)`? Draw the new timing diagram for each case.

(d) Modify the data structures to allow a higher degree of concurrency in the ordered insertion of elements inside the same chained list (it is NOT necessary neither to implement changes in the `insert_elem` function code nor to draw again the timing diagram).

14. Given the following code for population shuffle in a Genetic Algorithm:

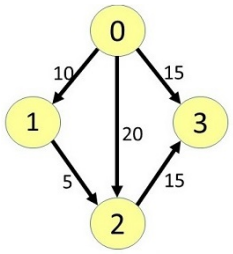
```
Int const NPOP    // number of chromosomes
Int const NCHROME // length of each chromosome

Array of Int :: iparent(NCHROME, NPOP)
Array of Int :: temp(NCHROME)
Real :: tempScalar
Array of Real :: fitness(NPOP)
Int :: j, iother

for (j=0; j<NPOP; j++) {
       iother = rand(j) // returns random value >= 0, != j, < NPOP
       // Swap Chromosomes
       temp(1:NCHROME) = iparent(1:NCHROME, iother);
       iparent(1:NCHROME, iother) = iparent(1:NCHROME, j);
       iparent(1:NCHROME, j) = temp(1:NCHROME);
       // Swap fitness metrics
       tempScalar = fitness(iother);
       fitness(iother) = fitness(j);
       fitness(j) = tempScalar;
} // end loop [j]
```

Determine the most appropriate task decomposition for a shared memory architecture, indicate the dependencies and data sharing between tasks, and describe the necessary synchronizations.

15. Describe two anomalies that make the parallel execution of the following loop incorrect and explain how to correct them:

```
#pragma omp parallel for
for (i=0; i<N; j++) {
     j = f(i); // returns any value between 0 and N-1, different than i
     omp_set_lock(&lck(i);
     omp_set_lock(&lck(j);
     int temp = vector[i];
     vector[i] = vector[j];
     vector[j] = tmp;
     omp_unset_lock(&lck(j);
     omp_unset_lock(&lck(i);
     }
```

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

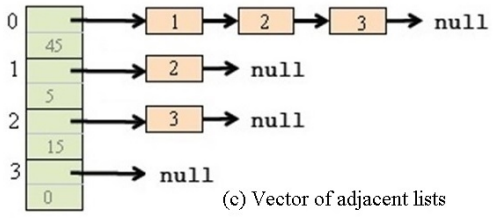


(a) Example graph

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

(b) Vector of edges

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

typedef tEdge *graphVec;
```



(c) Vector of adjacent lists

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

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

typedef list *graphList;
```

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

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

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

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

```
            // Accumulate total weight of edges coming from "from"
            g[from].totWeight += w;
            }
        }
    }
}
```

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

(b) The following code corresponds to an alternative sequential implementation of function `buildGraphList`. In this implementation, for each node `i` in the graph (first loop) the algorithm finds out which edges `k` in vector `v` (second loop), have a "from" node `i`. If so, it inserts the "to" node into the list of adjacent nodes of `i`.

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

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

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

4

Multiprocessor architectures

1. Given an SMP system with 3 CPUs, each with a cache memory initially empty, Snoopy, write-invalidate protocol and MESI coherence protocol, and assuming the following access sequence to the same memory direction: `r1, w1, r2, w3, r2, w1, w2, r3, r2, r1` (where `rx` indicates read by processor `x` and `wy` write by processor `y`), fill in the table indicating the CPU event (PrRd, PrWr), Bus transaction(s) (BusRd, BusRdX, Flush) and tate of the cache line (M, E, S, I) in each processor after each access to memory.

Memory Access	CPU event	Bus transaction(s)	Cache Line State		
			Mem1	Mem2	Mem3
r1					
w1					
r2					
w3					
r2					
w1					
w2					
r3					
r2					
r1					

2. For a SMP system with 3 CPUs with Snoopy, write-invalidate protocol and MSI coherence protocol, indicate:

   (a) A sequence of reads and writes by the CPUs that causes that one line of cache of the CPU1 change from (1) Invalidate to Modified, (2) from Modified to Shared and (3) from Shared to Invalidate. Moreover, for each operation indicate the operations that each processor does locally (PrWr,PrRd) and on the Bus (BusRd, BusRdX, Flush), and the states of the cache lines for all the CPUs.

   (b) Another sequence of reads and writes that provokes false sharing between processors CPU1 and CPU2. Explain your answer.

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

   Answer the following questions:

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

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

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

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

4. Describe the performance problem that occurs during the parallel execution of the following loop. Explain how you could fix it.

```
#pragma omp parallel for schedule(static,1) numthreads(NT)
for (i=0; i<N; j++)
     salida[i%NT] += entrada[i];
```

5. Given the following code excerpt including OpenMP directives:

```
int vector[N];
typedef struct {
    int pares = 0;
    int impares = 0;
    int dummy[PAD];
} contar[NUM_THREADS];
...
#pragma omp parallel
{
   int id = omp_get_thread_num();
   #pragma omp for schedule(static, CHUNK) private(i)
   for (i=0; i < N; i++)
       if (vector[i]%2) contar[id].pares++;
       else contar[id].impares++;
}
```

in which thread executes groups of consecutive `CHUNK` iterations in a round robin (cyclic) way. Assuming that each integer (`int`) variable occupies 4 bytes, a cache line occupies 32 bytes, and that the initial address of `vector` and `contar` are aligned with the start of a cache line, we ask:

(a) Compute the minimum value for constant `PAD` in the previous program in order to avoid *"false sharing"* during the execution of the parallel loop.

(b) Compute the minimum value for constant `CHUNK` in order to improve spatial locality, within each thread, when reading the elements of `vector`.

6. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction `t&s`:

```
lock: t&s r2, barr.lock          // acquire lock
      bnez r2, lock
      if (barr.counter == 0)
         barr.flag = 0           // reset flag if first
      mycount =  barr.counter++;
      if (mycount == P) {        // last to arrive?
         barr.counter =0         // reset for next barrier
         barr.flag = 1           // release waiting processors
      } else
         while (barr.flag == 0); // busy wait for release
      barr.lock = 0              // release lock
```

We ask:

(a) Identify a concurrency problem that exists in this code and propose a solution to solve it.

(b) Based on the solution proposed for the concurrency problem identified in the first question, implement a new version that reduces the synchronization overhead to acquire `lock`.

7. Propose two different versions for the program excerpt below that reduce the overheads related to *for/join*, *work-sharing* execution and *data sharing synchronisation*.

```
int i, j, sum=0;
for (j=0; j<M; j++) {
  #pragma omp parallel num_threads(NT)
  #pragma omp for
    for (i=0; i<N; i++)
       #pragma omp atomic
         sum+=foo(j,i);
}
```

The two proposed solutions should, at least, use different strategies to reduce the *data sharing synchronisation* without incurring new overheads, assuming that we are targeting the execution on a NUMA architecture. You can assume the execution of `foo` does not incur any data dependence.

8. Dado el siguiente código secuencial que calcula el histograma de los valores de los elementos de la variable `input` (en el rango `0..RANGE_OF_VALUES-1`):

```
void histogram_count(int *input, int *histogram, int n) {
   for (int i=0; i<RANGE_OF_VALUES; i++) histogram[i]=0;
   for (int i=0; i<n; i++) histogram[input[i]]++;
}
```

Se propone la siguiente solución paralela para memoria compartida:

```
#define CACHE_LINE_SIZE 128

typedef struct {
   int count;
   omp_lock_t lock;
   char tmp[***apartado a)***];
} element;

element histogram[RANGE_OF_VALUES];

void histogram_count(int *input, element * histogram, int n) {
   for (int i=0; i<RANGE_OF_VALUES; i++) {
      omp_init_lock(&histogram[i].lock);
      histogram[i].count=0;
   }

   // Punto A       ***apartado b)***

   #pragma omp parallel for schedule(static)
   for (int i=0; i<n; i++) { // ***apartado c)***
     omp_set_lock(&histogram[input[i]].lock);
     histogram[input[i]].count++;
     omp_unset_lock(&histogram[input[i]].lock);
   }

   for (int i=0; i<RANGE_OF_VALUES; i++) omp_destroy_lock(&histogram[i].lock);
}
```

Se pide:

(a) Completad la definición del tipo de dato `element` de manera que se reduzcan los *overheads* que pueda introducir el protocolo de coherencia en un sistema multiprocesador NUMA.

(b) Suponiendo que el multiprocesador está formado por 2 nodos NUMA, cada uno de ellos con un único procesador, y que ejecutamos el programa con 2 *threads* (es decir, *thread* i en nodo NUMA i), indicad en qué elemento(s) *hardware* del sistema, y en qué nodo(s) NUMA en concreto queda guardada la información que se utiliza para mantener la coherencia de la variable `histogram` una vez llegamos al *"Punto A"* del programa.

(c) Rellenad la tabla siguiente con las transacciones entre nodos NUMA que se realizan para mantener la coherencia de memoria durante la ejecución del bucle paralelo que sean provocados por el acceso a la variable `histogram`, así como el estado del directorio. Para ello considerad las siguientes condiciones: 1) `RANGE_OF_VALUES=64`; 2) vector `input` con 4 elementos: {0,0,0,4}; y 3) la ejecución de las iteraciones asignadas a los 2 *threads* se intercalan en el tiempo, tal como muestra la columna de la izquierda de la tabla a rellenar.

Tiempo	Iteración bucle i	NUMA transactions	Directory entry (state - sharers)
0	0		
1	2		
2	1		
3	3		

9. Given an SMP architecture with 2 processors, each one with a local cache memory and snoopy implementing a *write–invalidate MSI* coherence protocol, in which we execute the following code:

```
// processor P0                          // processor P1
count = 0;
flag = 1;                                ...
for (i = 0; i < 4; i++)            lock: while (test&set(flag));
    count++;                             ...
flag = 0;
```

Assuming that variable `i` is stored in one of the registers in the register file and initially empty cache local memories, **we ask:**

(a) Complete the following table with the actions that occur (*bus transactions* and changes in the state of cache lines to maintain memory coherency, assuming that there is no false sharing in the accesses to variables `count` and `flag`, and the following temporal ordering of memory instructions:

Tiempo	Procesador P0			Procesador P1		
	Instrucción ejecutada	Bus transaction	Estado cache	Instrucción ejecutada	Bus transaction	Estado cache
0	**count=0**					
1	**flag=1**					
2				**t&s flag**		
3	**count++**					
4				**t&s flag**		
5	**count++**					
6				**t&s flag**		
7	**count++**					
8				**t&s flag**		
9	**count++**					
10				**t&s flag**		
11	**flag=0**					
12				**t&s flag**		

You can also assume that the execution of the `test&set` instruction (abbreviated `t&s`) and the increment `++` imply a single transaction `BusRdX` when they miss in the cache memory.

(b) Repeat the previous question now assuming that there is false sharing when accessing variables `count` and `flag`, completing the following table:

Tiempo	Procesador P0			Procesador P1		
	Instrucción ejecutada	Bus transaction	Estado cache	Instrucción ejecutada	Bus transaction	Estado cache
0	**count=0**					
1	**flag=1**					
2				**t&s flag**		
3	**count++**					
4				**t&s flag**		
5	**count++**					
6				**t&s flag**		
7	**count++**					
8				**t&s flag**		
9	**count++**					
10				**t&s flag**		
11	**flag=0**					
12				**t&s flag**		

(c) With the objective of reducing coherency traffic, substitute the instruction *test&set* with a sequence of instructions that performs a *test-test&set* (write the proposed sequence of instructions).

(d) With the proposed sequence of instructions, complete the following table assuming that processor `P1` continues doing accesses to memory in the same cycles and that there is false sharing in the access to variables `count` and `flag`.

Tiempo	Procesador P0			Procesador P1		
	Instrucción ejecutada	Bus transaction	Estado cache	Instrucción ejecutada	Bus transaction	Estado cache
0	**count=0**					
1	**flag=1**					
2						
3	**count++**					
4						
5	**count++**					
6						
7	**count++**					
8						
9	**count++**					
10						
11	**flag=0**					
12						

5

Data decomposition

1. Write an OpenMP parallelization for the following sequential code that performs the dot product of two vectors:

```
void main (int argc, char *argv[]) {
int i;
float FinalAnswer=0.0;
float * Vector_A, *Vector_B;

Vector_A = (float *)malloc(N*sizeof(float));
Vector_B  = (float *) malloc(N*sizeof(float));

InitVector(Vector_A, N);
InitVector(Vector_B, N);

for (i=0 ;i<N; i++)
    FinalAnswer += Vector_A[i]*Vector_B[i];

free(Vector_A);
free(Vector_B);
}
```

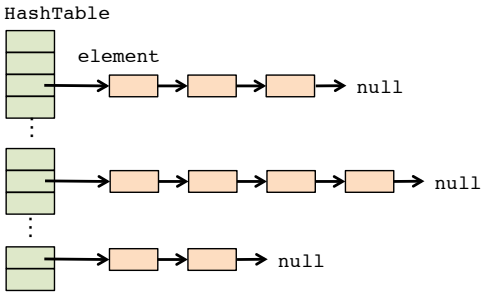
   assuming that vectors A and B are geometrically decomposed in blocks among all processors as indicated in the following data structure

```
typedef struct {
    int lower;
    int upper
} limits;

limits decomposition[num_threads];
```

   and that `num_threads` indicates the number of threads to be used.

2. Assume a hash table implemented as a chained list vector, such as the one shown in the next figure and type definition:

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

element * HashTable[SIZE_TABLE];
```

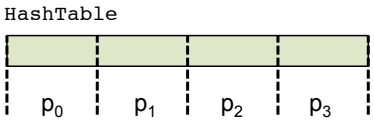


Taking as a starting point the following sequential code:

```
#define MAX_ELEM 1024
int main() {
    int ToInsert[MAX_ELEM], num_elem, index;
    ...
    for (i = 0; i < num_elem; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        insert_elem (ToInsert[i], index);
    }
    ...
```

we can see that the function `hash_function` returns the entry of the table (between 0 and `SIZE_TABLE`-1 where a determined element has to be inserted and the function `insert_elem` inserts that element in the corresponding position inside the chained list pointed by the entry `index` of the table `HashTable`.

We ask: Redefine the data structures, if necessary, and write an OpenMP parallel version that obeys a data decomposition, where each thread has to do all the insertions that have to be done in `SIZE_TABLE/P` consecutive entries of the table `HashTable`, as shown in the next figure (being P the number of threads).



3. Dado el siguiente código secuencial en C que encuentra en un vector `DB` la primera posición en la que aparece una determinada clave `key`:

```
int main() {
    double key = 1.25;
    double * DB = (double *) malloc(sizeof(double) * DBsize);
    initialize(DB, &DBsize);  // initialize elements in DB
    unsigned long position = DBsize;

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

y la siguiente solución incompleta para la paralelización del bucle `for`:

```
  #pragma omp parallel
      {
      unsigned long i, num_elems, lower;
      for (i = lower; (i < (lower + num_elems)) && (i < position); i++) {
      #pragma omp critical
          if ((DB[i] == key) && (i<position)) position = i;
      }
```

Se pide:

(a) Completad la solución incompleta anterior (con las sentencias y declaraciones de datos necesarias) para que cumpla con las siguientes condiciones, sin preocuparse por posibles problemas de rendimiento: 1) el reparto de iteraciones a procesadores obedezca a una descomposición de datos geométrica tipo BLOCK (es decir, a cada procesador se le asocian `DBsize/P` elementos consecutivos, siendo `P` el número de procesadores); 2) `P` no tiene por que dividir de forma entera `DBsize`, en cuyo caso se deberá maximizar el balanceo de carga; 3) la solución debe permitir que un procesador finalice su ejecución tan pronto encuentre `key` o detecte que no contribuirá a la solución final; y 4) no puede utilizarse el `#pragma omp for` para realizar el reparto de iteraciones.

(b) Modificad el código anterior para que se mejore el rendimiento de forma substancial, reduciendo al mínimo la sequencialización que introduce la sincronización actual.

(c) Proponed e implementad una descomposición de datos geométrica alternativa que reduzca el tiempo de ejecución requerido, en media, para encontrar la primera posición en la que aparece la clave `key`.

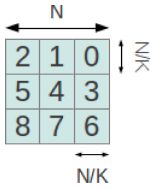
4. Dado el siguiente código incompleto:

```
typedef struct {
    int i_start, i_end, j_start, j_end;
} limits;
limits decomposition[num_threads];

void InitDecomposition(limits * decomposition, int N, int nt) { ...  }

void main (int argc, char *argv[]) {
  #pragma omp parallel
  #pragma omp single
     InitDecomposition(decomposition,N,omp_get_num_thread());
  #pragma omp parallel
  {
     ...
     int i_start = ...
     int i_end   = ...
     int j_start = ...
     int j_end   = ...
     foo(i_start,i_end,j_start,j_end);
  }
}
```

y suponiendo que `foo` realiza operaciones sobre todos los elementos de una matriz `NxN` desde la fila *i_start* hasta la fila *i_end* y la columna *j_start* y la columna *j_end*. Completa el código de la función `InitDecomposition` y del `main` para dos implementaciones que se quieren realizar:

- Implementación 1: La inicialización de la estructura `decomposition` en la función `InitDecomposition` da lugar a una `Block Geometric (Data) decomposition` por filas. En este caso NO puedes suponer que N sea múltiplo del número de `threads`. Además se quiere que el desbalanceo de carga entre threads sea no superior a una fila de cálculo.

- Implementación 2: La inicialización de la estructura `decomposition` en la función `InitDecomposition` da lugar a una descomposición geométrica en la que cada thread trata un bloque de la matriz tal y como muestra la figura.



  Sabemos que el número de threads es del tipo K^2 y que K divide perfectamente a N. Cada número en la figura indica el identificador del `thread`. Puedes suponer que existe una función `sqrt` que podemos usar en el código.

5. Se quiere paralelizar el cálculo del histograma de los valores que aparecen en un vector. El histograma es otro vector en el que cada posición almacena el número de valores en el vector de entrada que estan dentro de un determinado rango de valores. El siguiente código muestra una posible implementación para el cálculo del histograma (vector `frequency`) del vector de entrada `numbers`.

```
#define MAX_ELEM 1024*1024
#define HIST_SIZE 250
unsigned int numbers[MAX_ELEM];
unsigned int frequency[HIST_SIZE];

void ReadNumbers (int * input, int * size);

void FindBounds(int * input, int size, int * min, int * max) {
for (int i=0; i<size; i++)
      if (input[i]>(*max)) (*max)=input[i];

for (int i=0; i<size; i++)
      if (input[i]<(*min)) (*min)=input[i];
}

void FindFrequency(int * input, int size , int * histogram, int min, int max) {
int tmp;
for (int i=0; i<size; i++) {
      tmp = ((input[i] - min) * HIST_SIZE / (max - min - 1);
      histogram[tmp]++;
      }
}

void DrawHistogram(int * histogram, int minimum, int maximum);

void main() {
    int num_elem, max, min;

    ReadNumbers(numbers, &num_elem); // read input numbers
    max=min=numbers[0];
    FindBounds(numbers, num_elem, &min, &max); // returns the upper and lower
                                               // values for the histogram
    FindFrequency(numbers, num_elem, frequency, min, max); // compute histogram
    DrawHistogram(frequency, min, max); // print the histogram
}
```

Se pide:

(a) Escribir una versión paralela OpenMP para la función `FindBounds` que se base en una **descomposición de datos geométrica** `CYCLIC` **del vector** `input` (es decir, elementos con-

secutivos se asocian a procesadores consecutivos, de forma cíclica). **Nota:** no puede utilizarse el `pragma omp for`.

(b) Escribir una versión paralela OpenMP para la función `FindFrequency` que se base en una **descomposición de datos geométrica** `BLOCK` **del vector** `histogram` (es decir, a cada procesador se le asocian `HIST_SIZE/P` elementos consecutivos, siendo `P` el número de procesadores) y replicación del vector `input`. **Notas:** 1) no puede utilizarse el `pragma omp for`; 2) P no tiene por que dividir de forma entera a `HIST_SIZE`, debiéndose en este caso maximizar el balanceo de carga.

(c) Suponer que las funciones `ReadNumbers` y `DrawHistogram` se ejecutan en un único procesador (por ejemplo el `0`), y que el código resultante de aplicar la descomposición de datos de los apartados anteriores se quiere ejecutar en una arquitectura de memoria distribuida, indicar qué comunicaciones punto a punto y/o colectivas de MPI serían necesarias para realizar el movimiento entre procesadores del vector que almacena el histograma (`frequency`) y de la variable escalar `max` que almacena el valor máximo encontrado, concretando para cada una de dichas comunicaciones dónde se realizaría, los datos y los procesadores implicados (**No** es necesario escribir el código completo MPI).

6. Matrix multiplication can be expressed as multiplications and additions of submatrices. For example, the multiplication $C = A \times B$ can be seen as the calculation of their elements or submatrices $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$:

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

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

The following code (`matmul`) does the multiplication of matrices A and B using submatrices:

```
#define N  1024
#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]) {
  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_submatrix(N, BS, &A[i][k], &B[k][j], &C[i][j]);
}

void main() {
  ...  matmult(A,B,C); ...
}
```

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.

We ask you to provide a parallelization using OpenMP of the `matmul` function, using a *Geometric Data Decomposition*: we will do the Data Decomposition from the point of view of the output (C), with the idea of applying the *owner-computes rule*, i.e. each thread has to do all the necessary calculations to calculate their own submatrix of C. For instance, given the matrix partitioning into 4 submatrices shown above, once the Data Decomposition is done, the task allocation would be the following:

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

$$\text{Task 0: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$
$$\text{Task 1: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$
$$\text{Task 2: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$
$$\text{Task 3: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Hint: You have to do it in such a way that each thread will have one and only one Task assigned (for example: $task_0 \rightarrow thread_0$). Consequently, for the general case we need as many threads as $(N/BS) \times (N/BS)$, the total of submatrices of $BS \times BS$ elements of C.

7. A continuación se presenta un código que realiza la transposición a bloques de una matriz. Los bloques son de tamaño $BS \times BS$ y la matriz de tamaño $N \times N$, donde N es múltiplo de BS.

   Para realizar esta transposición a bloques se dispone de las subrutinas siguientes:

   - `void read_block(int Asrc[N][N], int i, int j, int Adest[BS][BS])`: subrutina que lee una submatriz (bloque) de `Asrc` (`Asrc[i..i+BS-1][j..j+BS-1]`) desde el elemento `Asrc[i][j]` y lo deja en la matriz `Adest`.
   - `void tranpose_block(int A[BS][BS])`: subrutina que realiza la transposición de la matriz `A`.
   - `void write_block_back(int Asrc[BS][BS], int Adest[N][N], int i, int j)`: subrutina que escribe la matriz `Asrc` en `Adest` (`Adest[i..i+BS-1][j..j+BS-1]`) a partir `Adest[i][j]`.

   El código de la transposición a bloques es:

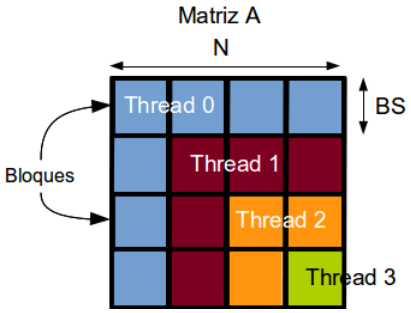
```
void tranpose(int A[N][N]) {
    int Ablock_ij[BS][BS], Ablock_ji[BS][BS];
    int i, j;

    for (i = 0; i < N; i += BS) {
      // Transpose de un bloque de la diagonal
      read_block(A, i, i, Ablock_ij);
      transpose_block(Ablock_ij);
      write_block_back(Ablock_ij, A, i, i);

      // Transpose de bloques que no son de la diagonal y swap de bloques (i,j) y (j,i)
      for (j=i+BS; j<N; j+=BS) {
         read_block(A, i, j, Ablock_ij);
         read_block(A, j, i, Ablock_ji);
         transpose_block(Ablock_ij);
         transpose_block(Ablock_ji);
         write_block_back(Ablock_ij, A, j, i);
         write_block_back(Ablock_ji, A, i, j);
      }
    }
}
```

a) Dada la siguiente solución incompleta para una *data decomposition* en la que cada thread se debería encargar de uno de los bloques de la matriz:

```
void tranpose(int A[N][N]) {
   #pragma omp parallel num_threads(N/BS*N/BS)
     {
     int Ablock_ij[BS][BS];

     int myid= omp_get_thread_num();
     int i = ...
     int j = ...

     read_block(A, i, j, Ablock_ij);
     #pragma omp barrier
     transpose_block(Ablock_ij);
     write_block_back(Ablock_ij, A, j, i);
     }
}
```

Se pide completar el cálculo de las variables `i` y `j` y justificar la necesidad de la sincronización `barrier` que se utiliza.

b) Con el objetivo de eliminar dicha sincronización **se pide** que implementéis una nueva solución paralela con una *data decomposition* como la que se indica en la figura siguiente, y sin usar `#pragma omp parallel for` ni `#pragma omp for`. Observar que cada thread tiene asignado todos los bloques del mismo color. Aunque en la figura se muestra para el caso $N/BS = 4$, vosotros lo tenéis que hacer para cualquier N/BS.



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

```
data * input;

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

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

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

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

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

(a) Write a parallel OpenMP version of the `rebinning` function following a *CYCLIC* data decomposition strategy for the input histogram `hist`. In a *CYCLIC* data decomposition consecutive elements are assigned to consecutive processors in a round–robin way, starting from processor 0. The output histogram `new_hist` is not decomposed.

(b) Write a parallel OpenMP version of the `rebinning` function following a *BLOCK* data decomposition strategy for the output histogram `new_hist`. In a *BLOCK* data decomposition each processor is assigned a single block of consecutive elements, trying to maximize load balancing (the number of processors does not necessarely divides `new_size`. The input histogram `hist` is not decomposed.

9. Given the following iterative task decomposition strategy implemented in OpenMP for a loop computing the evolution of the forces in a N-body gravitational problem.

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

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

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

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

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

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

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