PAR: Collection of Exercises

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

Course 2022-23

Index

Index **1**

1 Concurrency and parallelism **2**

2 Understanding parallelism **3**

3 Multiprocessor architectures **16**

4 Task decomposition **24**

5 Data-aware task decomposition **42**

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 the 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.

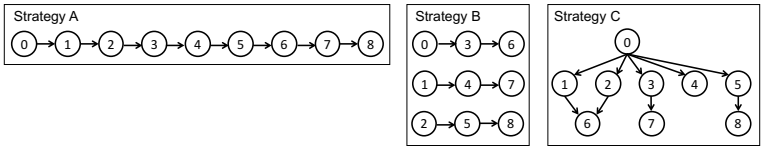
   (c) The same as in the second case, but now *App2* is parallelised defining 3 tasks, each task executing one third of the total application.

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 graphs for three different parallelization strategies of a sequential code:



   Answer the following questions:

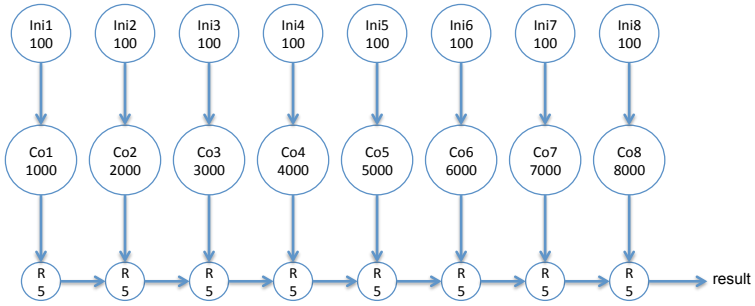
   (a) Compute the *Parallelism* and P_{min} metrics for each one of the three dependence graphs assuming that the cost of executing each task is t_c time units.

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

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

   Obtain the general expression for the speed–up S_3 for each strategy and associated mapping, assuming that there is an overhead related with task synchronisation of t_{synch} time units, i.e. the overhead that a task has to pay to signal to ALL its successor tasks that it has finished.

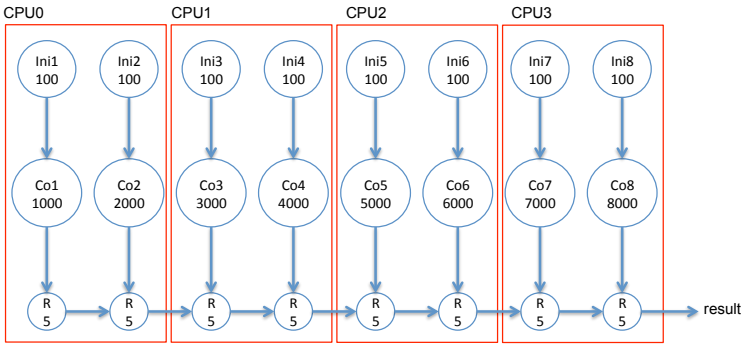
2. The following figure shows the Task Dependence Graph (TDG) representing the task decomposition of a program:



   In the figure there are three different kinds of nodes: Ini_X, Co_X and R_X (with X in the range 1 to 8). Each node Co_X depends of Ini_X and each node R_X depends of Co_X and R_{X-1} (notice that label X is omitted in the picture above for tasks R since their execution is commutative, i.e. you can execute them in any order); the last task R in the chain of dependences outputs the result of the computation. In terms of costs, each node Ini_X runs for 100 time units, each node Co_X runs for `X*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 in the following figure:



In this figure, tasks Ini_X, Co_X and R_X with $X = \{1, 2\}$ are assigned to CPU0, with $X = \{3, 4\}$ to CPU1, with $X = \{5, 6\}$ to CPU2 and with $X = \{7, 8\}$ to CPU3.

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

3. 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.

4. 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:**

(a) Draw the Task Dependence Graph (TDG), indicating for each node its cost in terms of execution time (in time units).

(b) Compute the values for T_1, T_∞ and the potential *Parallelism*, as well as the parallel fraction (ϕ).

(c) Indicate which would be the most appropriate task assignment on two processors in order to obtain the best possible "speed up". For that assignment, calculate T_2 and S_2.

5. Given the following Tareador–instrumented sequential 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, t;
   for (i=0; i<MAX_ITERS; i++) {
       tareador_start_task("seq_func");
       seq_func(i);
       tareador_end_task("seq_func");
       for (t=0; t<=i; t++) {
           tareador_start_task("execute_task");
           execute_task(i, t);
           tareador_start_task("execute_task");
       }
    }
}
```

The required specifications for the tasks defined above are the following ones:

- `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.

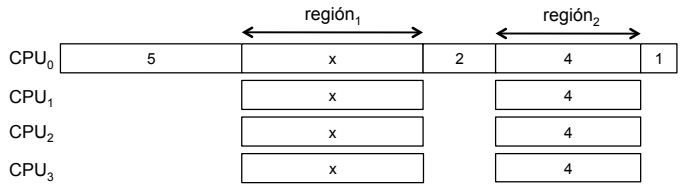
We ask you:

(a) Draw the Task Dependence Graph (TDG) that would be obtained when `MAX_ITERS` is set to 4. Each node should be labeled with the task name followed by the iteration number (i.e. a single value `i` for tasks `seq_func(i)` and a tuple `<i,t>` for tasks `execute_task(i,t)`).

(b) Compute T_1, T_∞ and *Parallelism* for the TDG obtained above (for `MAX_ITERS=4`). For that, 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).

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

6. The following figure shows an incomplete time diagram for the execution of a parallel application on 4 processors:



The figure has a set of rectangles, each rectangle represents the execution of a task with its associated cost in time units. In the timeline there are two regions (1 and 2) with 4 parallel tasks each. The execution cost for tasks in $region_1$ is unknown (x time units each); the cost for each task in $region_2$ is 4 time units. The computation starts with a sequential task (with cost 5), then all tasks in $region_1$ running in parallel, followed by another sequential task (with cost 2), then all tasks in $region_2$ running in parallel followed by a final sequential task (with cost 1).

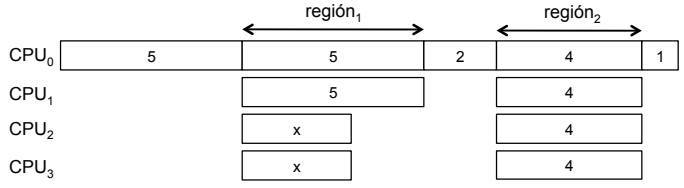
Knowing that an ideal speed-up of 9 could be achieved if the application could make use of infinite processors ($S_{p\to\infty} = 9$), and assuming that the two parallel regions can be decomposed ideally, with as many tasks as processors with the appropriate fraction of the original cost, **we ask:**

(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. The following figure shows an incomplete time diagram for the execution of a parallel application on 4 processors:



The figure has a set of rectangles, each rectangle represents the execution of a task with its associated cost in time units. In the timeline there are two regions (1 and 2) with 4 parallel tasks each. The execution cost for tasks in $region_1$ is not balanced, which means that 2 of the 4 tasks in $region_1$ have a cost of 5 time units and the other two tasks in the same region have an unknown cost x (which can be smaller, equal or larger than 5). The computation starts with a sequential task (with cost 5), then all tasks in $region_1$ running in parallel, followed by another sequential task

(with cost 2), then all tasks in $region_2$ running in parallel followed by a final sequential task (with cost 1).

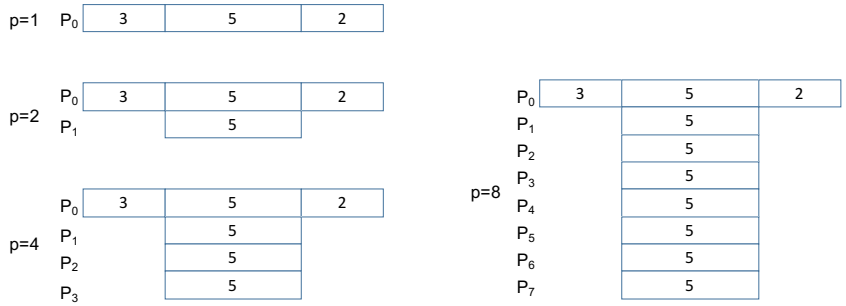
We ask you to answer the following questions:

(a) 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 (x) for the duration of the unknown tasks in $region_1$.

(b) Knowing that an ideal speed-up $S_{p\to\infty} = 5$ when the application is executed with infinite processors (assuming that, despite the imbalance in $region_1$, both parallel regions can be decomposed ideally, with as many tasks as processors with the appropriate fraction of the original cost, compute the value of the parallel fraction (ϕ) and the new value for x.

(c) In case that $x = 0$ and considering that it is not possible to distribute the workload in $region_1$ to more than two processors, which would be the $S_{p\to\infty}$ that could be achieved for this application?

8. The following figure shows 4 execution timelines to analyse the **weak scaling efficiency** of a certain parallel application executed on 1, 2, 4 and 8 processors:



Each timeline represents the execution of the application for a problem size that is proportional to the number of processors p that is used. Rectangles represents the execution of a task with its associated cost in time units. In all 4 timelines there is a single parallel region, using as many tasks as processors p; each one of these tasks has a constant cost of 5 time units. Also, in all 4 timelines the computation always starts with a sequential task (with cost 3) and finishes with another sequential task (with cost 2).

(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) Let T_x^y be the execution time for problem size y when using x processors. With this we define $S_p = T_1^p / T_p^p$ (i.e. the speed–up is computed with respect to the sequential execution time **for the problem size used for p processors**). Compute the values for the speed–up S_2, S_4 and S_8.

(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).

9. 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 parallelisation strategy already defined for this application and that its sequential execution time is $T_1 = 6$ time units. Answer the following questions:

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

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

Independently of the previous questions, now assume a parallelisation 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*.

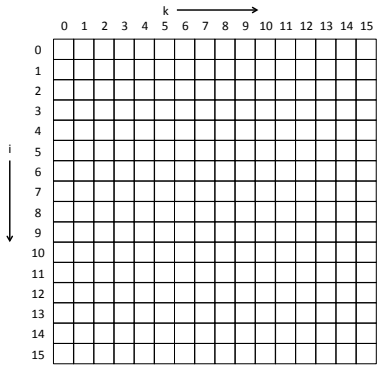
(c) What is the minimum number of processors P_{min} that are required in order to achieve the potential *Parallelism* obtained in the previous question?

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

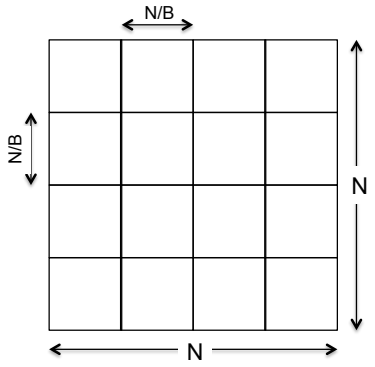
```
for (i = 1; i < N-1; i++)
    for (k = 1; k <= i; k++) {
        tmp = 0.3 * u[i+1][k+1] + 0.7 * u[i-1][k-1];
        u[i][k] = (tmp * tmp) / 4;
    }
```

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 (rectangle with $N \times N$ cells, as shown in the following figure), 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 blocks, each block executed by a task with N/B consecutive iterations of loop `i` and N/B consecutive iterations of loop `j` (considering that B perfectly divides N). The following figure shows a specific example 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 synchronisation between two processors takes t_{sync} time units, obtain the synchronisation overhead that would be added to the previous expression for T_P.

11. 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) Draw the Task Dependence Graph (TDG) that would be generated, 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.

12. Given the following C code with tasks identified using the *Tareador* API:

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

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

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

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

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

(a) Indicate which positions of matrix `m` are read and/or written by each task generated in `loop 1` and `loop 2` and in task `print_results`. You can fill in the table below to answer this question.

loop 1

task id	iteration (i, k)	Read	Written
t1	(0, 0)		
t1	(0, 1)		
t1	(0, 2)		
t1	(0, 3)		
t2	(1, 0)		
t2	(1, 1)		
t2	(1, 2)		
t2	(1, 3)		
t3	(2, 0)		
t3	(2, 1)		
t3	(2, 2)		
t3	(2, 3)		
t4	(3, 0)		
t4	(3, 1)		
t4	(3, 2)		
t4	(3, 3)		

loop2

task id	iteration (i, k)	Read	Written
t5	(0, 0)		
t5	(0, 1)		
t5	(0, 2)		
t5	(0, 3)		
t6	(1, 0)		
t6	(1, 1)		
t6	(1, 2)		
t6	(1, 3)		
t7	(2, 0)		
t7	(2, 1)		
t7	(2, 2)		
t7	(2, 3)		
t8	(3, 0)		
t8	(3, 1)		
t8	(3, 2)		
t8	(3, 3)		

print_results

task id	Read	Written
t9		

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

(c) Compute the values for T_1, T_∞ and the potential *Parallelism*.

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

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

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

13. For each of these two loops:

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

```
2) for (i=1; i<n; i++)
     for (j=1; j<n; 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) Consider that we are using a distributed memory machine with P processors and that matrices A and B are distributed by columns, each processor storing in its local memory the elements in a block of n rows by n/P consecutive columns.

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

14. 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]) {
    for (int i=0;i<=N;i++) {
      h[0][i]=0;
      h[i][0]=0;
    }

    for (int i=1;i<=N;i++)
      for (int j=1;j<=N;j++) {
        int diag    = h[i-1][j-1] + sim[a[i-1]][b[j-1]];
        int down    = h[i-1][j] + 4;
        int 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 initialising 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) 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 $N >> P$.

15. 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];
    }
}
```

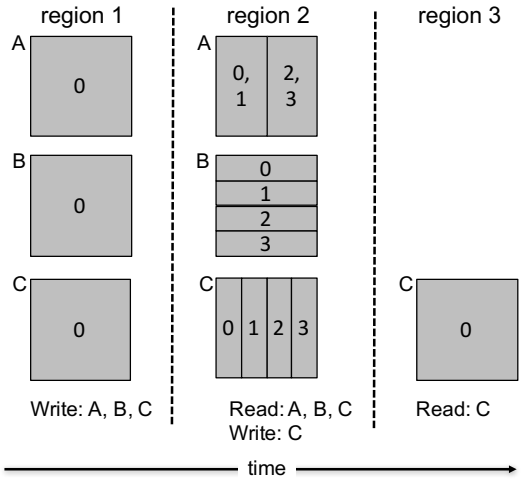
using the data sharing model explained in class based on the distributed memory architecture with message passing: the access time to remote data is determined by $t_{comm} = t_s + m \times t_w$, being t_s and t_w the "start-up" and sending time of an element, respectively, and being m the size of the 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**
a) Task definition:		
b) Initial remote accesses		
Total number of messages:		
Size of each message:		
Contribution to T_p:		
c) Parallel computation		
Total number of tasks:		
Size of each task:		
Contribution to T_p:		
d) Remote accesses during parallel execution		
Total number of messages:		
Size of each message:		
Contribution to T_p:		

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



In Region 1 processor 0 initialises (writes) all the elements of three matrices: A, B and C, all of size n *rows* $\times$ n *columns*). In Region 3 only matrix C is read by processor 0. In Region 2 the following accesses are done: 1) for matrix A processors 0 and 1 read its first half (n *rows* $\times n/2$ *columns*) while processors 2 and 3 read its second half (also n *rows* $\times n/2$ *columns*); 2) for matrix B each processor reads to a block of the matrix with $n/4$ *rows* $\times$ n *columns*, the first block read by processor 0, the second by processor1, and so on so forth; and 3), for matrix C each processor reads and writes to a block of the matrix with n *rows* $\times$ $n/4$ *columns*, the first block accessed by processor 0, the second by processor1, and so on so forth. 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.

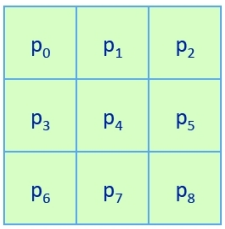
We ask you to write the expression for the data sharing (i.e. communication) overheads associated with the execution of the whole program, clearly indicating how each region contributes to that expression.

17. Assume a distributed memory architecture with P^2 processors and using message passing to interchange data, 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, with blocks of n/P consecutive rows times n/P consecutive columns. The following figure shows a specific example for 9 processors (3^2), with processors p_0, p_1 and p_2 assigned to the first row of blocks, p_3, p_4 and p_5 assigned to the second row of blocks and p_6, p_7 and p_8 assigned to the last row of blocks, in that order within each row of blocks from left to right.



    Assume 1) a task is defined as a block of n/P by n/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).

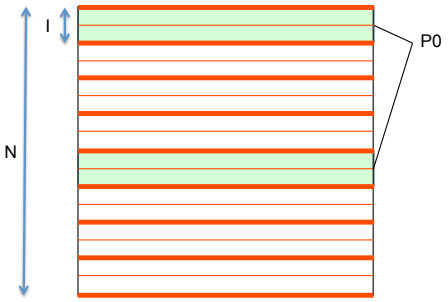
    (b) Continuing with the same assumptions above and same task definition, consider next the following loop:

```
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).

18. Given a geometrical **block–cyclic data distribution** for the two–dimensional matrices `a` and `b` that are used in a program on P processors: blocks of I consecutive rows cyclically assigned to processors in a round-robin way (i.e. blocks assigned to the same processor starting in $I \times P$ iterations apart). Each block has all the N columns. The following figure contains a specific example for $N = 16$, $I = 2$ and $P = 4$, showing that the first two rows are assigned to P_0, the next two to P_1, the next two to P_2 and the next two to P_3, repeating the same cyclic pattern for the rest of rows.



Consider 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_w: 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

Multiprocessor architectures

1. Given an SMP system with 3 CPUs, each with its own cache memory initially empty. To keep caches coherent the system uses a Snoopy–based write-invalidate *MSI* coherence protocol. Assuming the following sequence of memory instructions all accessing 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 including the CPU event (PrRd, PrWr), Bus transactions (BusRd, BusRdX, BusUpgr, Flush) and state of the cache line (M, S, I) in each cache memory after each access to memory. In the observations column indicate who is providing the line when a cache requires it and when main memory is updated.

Memory Access	CPU event	hit/miss	Bus transaction(s)	Cache Line State			Observations
				Cache1	Cache2	Cache3	
r1							
w1							
r2							
w3							
r2							
w1							
w2							
r3							
r2							
r1							

   What would change in the table if a *MESI* protocol is used instead of the original *MSI*?

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, BusUpgr, 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.

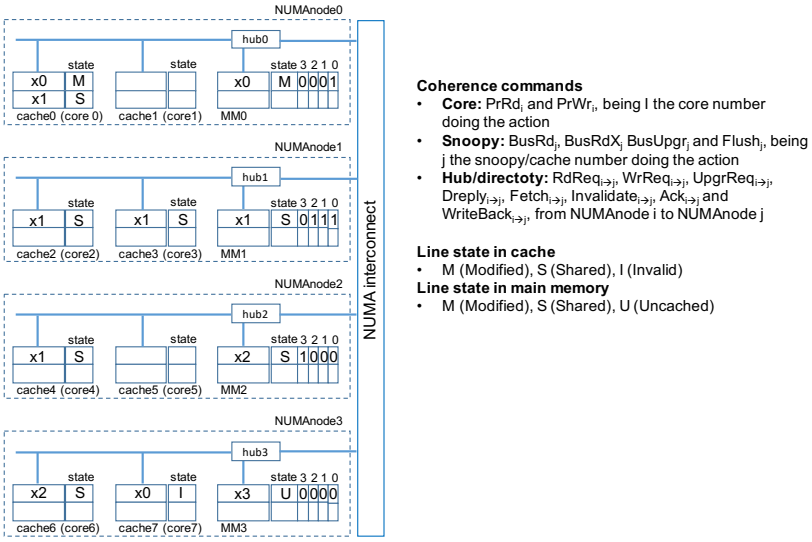
(b) How many bits are needed to maintain the coherence at the directory structures? What are their function/s? Please, give the number of bits per memory line and the total number of bits per directory.

(c) 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. Assume a multiprocessor system composed of four NUMA nodes ($NUMAnode_0$ to $NUMAnode_3$), 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. Assume also 4 variables, `x0`, `x1`, `x2` and `x3` that are stored in memory, 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. Assuming the following initial state of the memories (caches and MM):

   - The home node for the line that contains `x0` is NUMAnode 0. The directory has the presence bit associated to NUMA node 0 set to 1 and state M. The line is cached in the cache memory of the first core in NUMA node 0 (state M) and appears with invalid I state in the cache of the second core in NUMA node 3.
   - The home node for the line that contains `x1` is NUMAnode 1. The directory has the presence bits associated to NUMA nodes 0, 1 and 2 set to 1 and state S. The line is cached in the cache memory of the first core in NUMA node 0, in the first and second core of NUMA node 1 and in the first core of NUMA node 2, all in state S.
   - The home node for the line that contains `x2` is NUMAnode 2. The directory has the presence bit associated to NUMA node 3 set to 1 and state S. The line is only cached in the cache memory of the first core in NUMA node 3 with state S.
   - The home node for the line that contains `x3` is NUMAnode 3. The directory has all presence bits set to 0 and state U. There are no copies of that line in cache.

   The following figure represents the memory system and the initial status of the memories, as described above. It also recaps the list of possible coherence commands at the different levels and the state names for cache and memory lines, according to what is explained in the theory slides of the course.



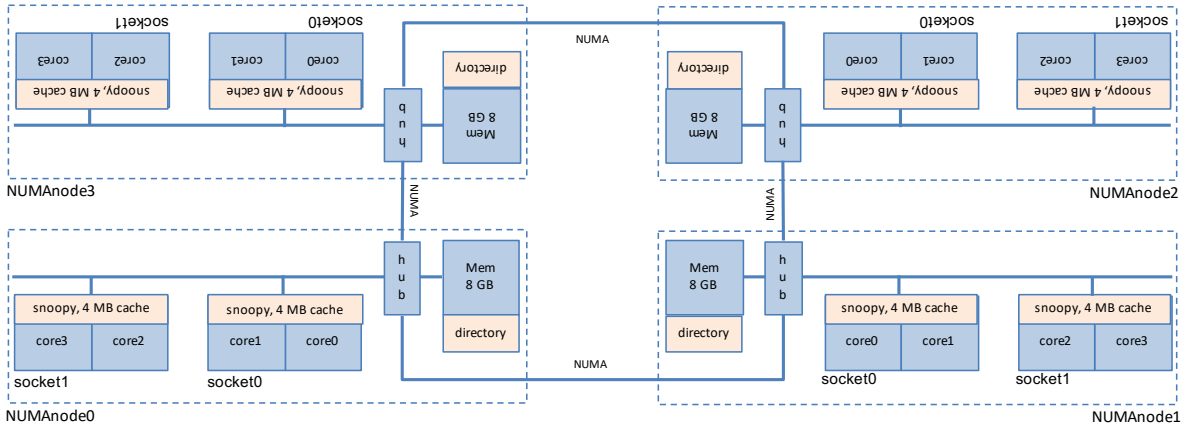
(a) Assuming that the multiprocessor system has 8 GB ($8 * 2^{30}$) of main memory, equally distributed in the four NUMA nodes and each processor has a cache memory of 4 MB ($4 * 2^{20}$), **we ask you** to compute the amount of bits taken by each snoopy to maintain the coherence between the caches inside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

(b) 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. Indicate which should be the correct state in the appropriate memories.

(c) If the second core in NUMA node 3 (`core7` in the diagram above) writes on variable `x0`, indicate 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.

(d) Finally, if the first core in NUMA node 1 (`core2` in the diagram above) writes on variable `x1`, indicate 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.

5. Given the following NUMA system with 4 dual-socket NUMA nodes, each NUMA node with 8 GB of main memory. Each socket has two cores sharing the access to a per-socket 4 MB cache. Coherence inside NUMA nodes is maintained with a snoopy–based mechanism implementing write–invalidate MSI. Coherence among NUMA nodes is maintained with a directory–based mechanism implementing write–invalidate MSU.



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

Assume that the home node for the line containing variable `var` is NUMAnode0, and at a given time there exist 3 clean copies of that line in cache memories: in socket0 in NUMAnode0, in socket0 in NUMAnode1 and in socket0 in NUMAnode2. Considering that the following memory accesses are done one after the other: 1) core2 in NUMAnode0 reads `var`; 2) core0 in NUMAnode3 reads `var`; 3) core0 in NUMAnode3 writes `var`; and 4) core0 in NUMAnode0 writes variable `other`. We ask you to select the sentences that are correct for each one of these 4 memory accesses (for each memory access at least one of the sentences is correct).

(a) When core2 in NUMAnode0 reads variable `var`, which of the following sentences are correct?

i. Core2 issues PrRd.

ii. The snoopy in socket1 issues BusRd on its local bus.

iii. The snoopy in socket0 observes the BusRd command and places the line on the bus (Flush).

iv. The hub associated to NUMAnode0 updates the directory for the line containing `var` to indicate that a new copy of the line exists inside NUMAnode0.

v. No coherence requests are sent to the rest of the NUMA nodes in the system.

(b) Then, when core0 in NUMAnode3 reads variable `var`, which of the following sentences are correct?

i. The snoopy in socket0 issues BusRd on its local bus.

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

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

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

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

(c) Then, when core0 in NUMAnode3 writes variable `var`, which of the following sentences are correct?

i. The snoopy in socket0 of NUMAnode3 issues an Invalidate command on its local bus.

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

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

iv. The state for the line in the caches of the remote nodes receiving the Invalidate command (as well as in the home node) changes from S to M to indicate that the line is modified somewhere else.

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

(d) Finally, core0 in NUMAnode0 writes variable `other` (observe that we are not giving any information about the home NUMA node for the line containing this variable). When this memory accesses finishes the state in the directory for the line containing variable `var` is kept to M, but the bit set to 1 in the sharers list moves from position 3 to position 0. Which of the following sentences are correct?

i. You should tell me which is the home NUMA node for variable `other` in order to decide if the following sentences are correct or not.

ii. This state in the directory is not possible since `var` and `other` are different variables.

iii. Since both variables reside in different cache lines, this state is possible if both variables are mapped in the same cache line entry (cache line replacement, i.e. the line containing `var` is replaced by the line containing `other`).

iv. This is the typical symptom of false sharing.

v. Since variables `var` and `other` are different, there is no need to interchange coherence commands between NUMAnode0 and NUMAnode3.

6. Assume a multiprocessor system composed of two NUMA nodes (labeled *NUMAnode0* and *NUMAnode1*), each with 8 GB of main memory shared by two sockets (labeled *socket0* and *socket1* for *NUMAnode0*, and *socket2* and *socket3* for *NUMAnode1*). Each socket labeled *socketX* has one core (labeled *coreX*) with a cache memory of 8 MB. **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**.

   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 = (N / nths) * myid;
    int 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 at the memory page level and that each memory page contains a single memory line; 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:**

   (a) 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.

   (b) 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.

   (c) Assuming that all cache memories are empty at the beginning of the program, fill in the following trable 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.

Vector x range	# Home NUMA node	Presence bits	State in MM	State in cache			
				socket0	socket1	socket2	socket3

   (d) Assuming the final previous state of the multiprocessor system with the presence bits and state for each cache and memory line, fill in the following table 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:

      i. $core_2$ reads the contents of `x[2]`

      ii. $core_2$ writes the contents of `x[2]`

Command	Coherence actions			Presence bits	State in MM	State in cache			
	Core	Snoopy	Directory			socket0	socket1	socket2	socket3
$core_2$ reads x[2]									
$core_2$ writes x[2]									
$core_1$ reads x[0]									

iii. $core_1$ reads the contents of `x[0]`

(e) Given a new code where we are using only two threads and the iterations are executed by the threads in an interleaved way:

```
#pragma omp parallel num_threads(2)
{
   int myid = omp_get_thread_num();
   int nths = omp_get_num_threads();

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

Assuming that threads 0 and 1 alternately execute the first 4 iterations in the time. What is the number of BusRdx that will be produced? Why is this happening?

7. Given the following code excerpt including OpenMP directives:

```
int vector[N];
typedef struct {
    int even = 0;
    int odd = 0;
    int dummy[PAD];
} count[NUM_THREADS];

...

#pragma omp parallel
{
   int id = omp_get_thread_num();
   int nt = omp_get_num_threads();
   for (int ii=id*CHUNK; ii < N; ii += nt*CHUNK)
       for (int i=ii; i < min(N, ii+CHUNK); i++)
           if (vector[i]%2) count[id].odd++;
           else count[id].even++;
}
```

in which each implicit task 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 `count` 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`.

8. 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 nThreads 8 // Number of threads
#define n 100000   // size of index vector, perfectly divides nThreads
#define m 5        // Number of bins in histogram

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

int main() {
    int iThread, BS;
    InitIndex(index, n); // initialization of vector index
    initHistograms(hist, hist_containers, m, nThreads);

    #pragma omp parallel private(iThread, BS) num_threads(nThreads)
    {
        iThread = omp_get_thread_num();
        BS = n / nThreads;
        for (int i = iThread * BS; i < (iThread + 1) * BS; 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:

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

(b) 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?

(c) 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.

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

4

Task decomposition

1. SAXPY (*Single-precision A times X Plus Y*) 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];
   }
   ```

   We ask you to write different parallel versions in OpenMP, all following a linear iterative task decomposition:

   (a) *version 1*: you are ONLY allowed to use the implicit tasks executed by threads created in the `parallel` region.

   (b) *version 2*: you are ONLY allowed to use explicit tasks, but NOT allowed to use `taskloop`, making explicit the data sharing clauses for the variables that are relevant.

   (c) *version 3*: you are ONLY allowed to use explicit tasks with `taskloop`, making explicit the data sharing clauses for the variables that are relevant.

   Important: In all code versions (implicit or explicit) tasks should execute $n \div P$ iterations of the loop, being P the number of threads in the parallel region; you can assume that P perfectly divides the number of iterations n and that $n \geq P$.

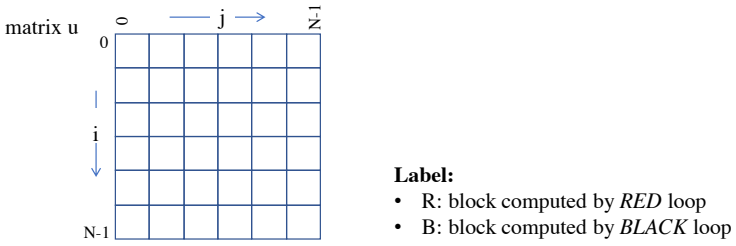
2. Given the following code computing matrix `u` by blocks of $BS \times BS$ elements, following the so called *Red-Black* algorithm:

   ```
   #define N 1024
   #define BS 128
   double u[N][N];

   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);
    ...
}
```

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:**

(a) The following figure represents the blocks of matrix `u` computed in the previous code (for the specific case of `N=6*BS` with 6×6 blocks). In this figure, indicate which blocks are computed by the *RED* and *BLACK* loops.

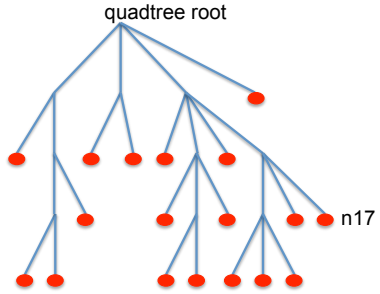


(b) If each invocation of function `compute_block` is defined as a task, are there any dependences precluding the parallel execution of the tasks generated during the execution of the *RED* loop? And for the tasks generated in the *BLACK* loop? Are there any dependences between the tasks generated in 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.

(c) 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`.

(d) Rewrite the parallel code so that you make use of the OpenMP `taskloop` construct to generate explicit tasks.

3. Given an ideal parallel architecture with an infinite number of processors with a task creation overhead of t time units and the following *quadtree* structure representing the execution of a recursive *Tree* task decomposition:



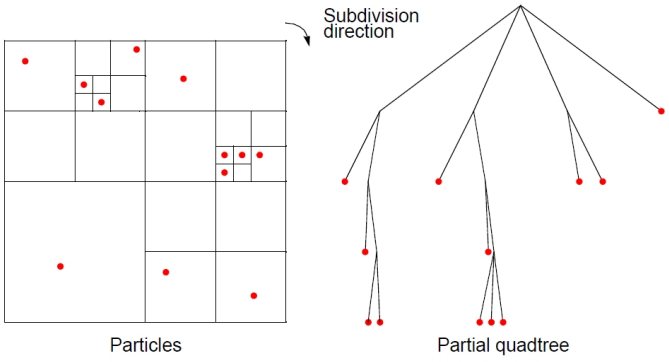
Each internal node in the tree represents a task that create new tasks, as many as branches going out of it. Each leaf in the tree (red dot) corresponds with the execution of function `doComp`; all instances of these computation tasks are totally independent. Assuming that the execution of the program starts at time 0, **we ask you** to determine when task labeled `n17` will be able to start its execution, assuming that the execution of each `doComp` instance takes d time units and the rest of code as a negligible execution time.

Repeat the problem considering that the *quadtree* above represents a recursive *Leaf* task depomposition, in which each internal node only performs recursive invocations, as many as branches going out of it (but not task creations). As before, each leaf in the tree corresponds with the execution of function `doComp`; all instances of these computation tasks are totally independent.

4. Given the following OpenMP parallelization for the traversal of a *quadtree* tree data structure:

```
void traverse(TreeNode* subTree, int d) {
  if(subTree) {
     if(!subTree->isLeaf) {
        if(d<3)
           #pragma omp parallel num_threads(4)
           #pragma omp single
           for(int i=0; i<2; i++)
              for(int j=0; j<2; j++)
                 #pragma omp task firstprivate(i, j, d)
                 traverse(subTree->quadrant[i][j], d+1);
        else
           for(int i=0; i<2; i++)
              for(int 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);
      ...
}
```

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?

(b) How many OpenMP (implicit and explicit) tasks are executed during the complete tree traversal? How many of them actually execute `doComputation`?

(c) Assuming that the execution of `doComputation` uses 25 time units, the creation of threads in a parallel takes 4 time units, the creation of a task takes 2 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.

(d) How would you reduce the overheads associated with the creation of OpenMP threads and improve the overall parallel execution time?

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

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

   (a) Write a parallel version in OpenMP implementing a linear or iterative task decomposition parallelisation strategy making use of the OpenMP tasking model, either with `task` or `taskloop`. Your parallel code should be parametric with the granularity of the tasks generated, so that each task executes `BS` iterations.

   (b) Write a new sequential version of the previous `sum_vector` function (`recursive_sum_vector`) that implements a "divide and conquer" recursive strategy over vector `v` with the objective of achieving an ideal $T_\infty = \log(n)$ in its parallelization. The recursion base case should execute the original `sum_vector` function with `BS` elements of the vector. 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.

   (c) Write a parallel version in OpenMP of the new recursive program.

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 parallelised 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;
    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
    #pragma omp single
        {
        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) Complete the parallelisation of the loop in function `CKJ` in order to implement an iterative task decomposition, with tasks of granularity `BS`.

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;
   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`), in which as many tasks as threads are generated.

   (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. 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);

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).

(a) 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.

(b) 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.

(c) 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.

10. 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;
}
```

(a) Write a first *OpenMP* parallelisation that implements an **iterative task decomposition strategy** of the outermost loop `i`, making use of the `taskloop` directive, 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.

(b) Write a second *OpenMP* parallelisation that also implements an **iterative task decomposition strategy**, but this time applied to the innermost loop `k`, again making use of the `taskloop` directive, 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.

(c) 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.

11. 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]);
    }
}
```

12. Given the following main program that makes use of `saxpy` function already defined in Problem 1 in this chapter. 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.

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

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);
}
```

13. As a continuation of problem 2 in this chapter, 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;

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 the new variable `residual` that is going to be used later in the third question. **We ask you:**

(a) Write a parallel version of the code making use of the following OpenMP constructs: `parallel`, `single` and `task` with `depend` clauses. No other task synchronization constructs are allowed.

(b) 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.

(c) 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.

14. 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` using an iterative task decomposition, in which you only generate as many tasks as threads in the parallel region, as you minimize the possible synchronization overheads.

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

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop private(tmp)
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.

15. Assume a hash table implemented as a vector of chained lists

```
#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
    #pragma omp single
    #pragma omp taskloop private(index) // default to as many tasks as threads
    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 task granularity is changed with `grainsize(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).

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

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop
for (i=0; i<N; i++) {
     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);
     }
```

17. Given the following sequential implementations for the computation of the histogram of the values in the vector `input` (values in the range `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]]++;
}
```

and the following parallel version in OpenMP:

```
#define CACHE_LINE_SIZE 128

typedef struct {
   int count;
   omp_lock_t lock;
   char tmp[*** to be computed in question (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;
   }
   // Point A       *** question (b) ***
   #pragma omp parallel
   #pragma omp single
   #pragma omp taskloop
```

```
   for (int i=0; i<n; i++) { // *** question (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);
}
```

(a) Complete the definition of data type `element` so that each element of the histogram occupies a whole cache line, minimizing in this way the overheads associated with the coherence protocol in a NUMA multiprocessor architecture.

(b) Assume that 1) the parallel system is composed of 2 NUMA nodes, each with a single processor and private cache; 2) the parallel program is executed with 2 threads (i.e. *thread* i in NUMA node i); and 3) the operating system makes use of "first touch" at page level to decide the allocation of memory addresses to NUMA nodes, one line per memory page. Once the execution of the program has reached *Point A*, explain where the elements of variable `histogram` will be stored in main memory and in cache memory.

(c) Assume that 1) `RANGE_OF_VALUES=64`; 2) vector `input` has 4 elements: {0,0,0,4}; 3) thread 0 executes the first task generated in the `taskloop` and thread 1 the second; and 4) the execution of the iterations assigned to the two threads are interleaved in time, as shown in the two leftmost columns in the table below. Complete the table with the contents of the directory for the line containing variable `histogram` during the execution of the parallel loop.

Time	Loop iteration i	Directory entry (sharers bit list)	Directory entry (status bits)
0	0		
1	2		
2	1		
3	3		

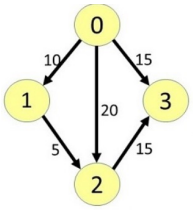
18. Given two possible representations for a graph: `graphVec` (graph stored as a vector of edges):

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

typedef tEdge *graphVec;
```

and `graphList` (graph stored as a vector of nodes, each with a list of its adjacent nodes)

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

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

typedef List *graphList;
```

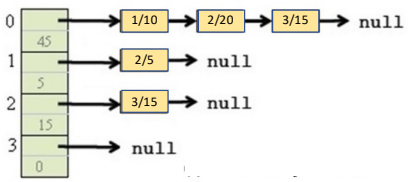
The following figure illustrates both representations for a simple example with only 4 nodes (0 to 3) with edges between (0,1) with weight 10, (0,2) with weight 20, (0,3) with weight 15, (1,2) with weight 5 and (2, 3) with weight 15. In the first representation the vector has as many elements as edges, in this case 5, each one indicating the source, destination and weight. In the second representation the vector has as many elements as nodes, in this case 4, each one with the total weight of all its outgoing edges and a list with these edges.



Example graph

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

Vector of edges



Vector of adjacency lists

(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, int weigth)`, which inserts node `n` into the list pointed by `l`.

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

19. Function `iter_distribute` copies vector S into vector D in such a way that all those elements $S[i]$ with the same value for $S[i]\%256$ are stored in consecutive positions of D. Therefore, at the end of the function,

there will be in D all the elements of S organized in 256 groups of elements: first all those with value %256 equal to 0, then those with value %256 equal to 1, ... up to those with value %256 equal to 255. In order to do this copy, the implementation provides a vector C which is initialized in function `preprocessing`; each element $C[value]$ indicates the initial position in D to store all those elements $S[i]$ whose $S[i]\%256 = value$.

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

void iter_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
    unsigned int i, value;

    for (i=0; i<n; i++) {
        value = S[i]%256;
        D[C[value]] = S[i];
        C[value]++;
    }
}

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

We have created a recursive divide–and–conquer sequential version of previous code for `iter_distribute` function.

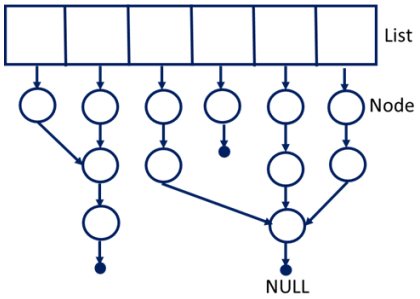
```
#define N 1024*1024*1024
unsigned int S[N], D[N], C[256];

void rec_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
    unsigned int i, value;
    unsigned int n2 = n/2;
    if (n==1) {
        value = S[0]%256;
        D[C[value]] = S[0];
        C[value]++;
    } else {
        rec_distribute(S, n2, C, D);
        rec_distribute(&S[n2], n-n2, C, D);
    }
}


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

Assuming that it is not important the order of the elements inside the same group in D (i.e. the elements of S can be written in different relative order in D in the parallel program than the order in the sequential code), **we ask you** to write an OpenMP parallel code for function `rec_distribute` and main program adding the necessary code and directives to implement a *recursive tree task decomposition*. Your parallel code should include a cut–off mechanism based on recursion depth, allowing parallel recursive calls for depths smaller than `MAX_DEPTH`. Your parallel code should minimize the serialization that is introduced by synchronization, if any.

20. Given the following C code that calculates the sum of all the elements in a structure of type List:

```
#define  N  ...

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

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

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

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

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



We ask you to:

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

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

```
float process_vector_rec (List *v[N], int n) {
     float res = 0.0;
     if (n <= MIN_SIZE)
         for (int i = 0; i < n; i++)
             res = res + compute_sequential (v[i]);
     else {
         int n2 = n / 2;
         float res1 = process_vector_rec (v, n2);
```

```
            float res2 = process_vector_rec (v+n2, n-n2);
            res = res1 + res2;
        }
        return res;
    }

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

21. 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`.

22. 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 thread x time units */
     } while (ret==1);
}
```

The function: `int test_and_set(int *lock, int value)` works as described in the course: atomically writes `value` into the memory address pointed by `lock`, returning the previous value on that memory address.

We ask you:

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

(b) Write an optimised 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.

23. `A ticket lock` is a lock implemented using two shared counters, `next_ticket` and `now_serving`, both initialised to 0. A thread wanting to acquire the lock uses an atomic operation to fetch the current value of `next_ticket` as its unique sequence number and increments it by 1 to generate the next sequence number. The thread then waits until `now_serving` is equal to its sequence number. Releasing the lock consists on incrementing `now_serving` in order to pass the lock to the next waiting thread. Given the following data structure and incomplete implementation of the primitives that support the ticket lock mechanism:

```
typedef struct {
   int next_ticket;
   int now_serving;
} tTicket_lock;

void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}

void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}

void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

We ask you to complete the code for the `ticket_lock_acquire` using:

(a) Low level synchronization function:

`int fetch_and_inc (int *addr);`

Recall that `fetch_and_inc (int *addr)` atomically returns the value of the memory pointed by `addr` pointer **and** increments by one this memory position.

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

5

Data-aware task decomposition

1. 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;
        }
    }
}
```

Notes: Function `ReadRowFromFile` is not provided: it reads one complete row from disk, (`Y_SIZE` elements) and stores it in row `i` of matrix `M`. Function `ComputePos` computes the position where element `(i, j)` has to be stored in vector `V` (it only depends on the values of `i` and `j`. Function `ComputeElement(M, i, j)` doesn't modify `M`.

We ask you:

(a) Create a OpenMP parallel version applying an input block geometric data decomposition strategy, so that each thread accesses to a block of consecutive rows of matrix `M`, for both initialization and main loop. 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.

(b) Create a new version applying an output block geometric data decomposition strategy, ensuring that each processor accesses to a block of consecutive positions of vector `V`. Reason if you need to include some synchronization.

(c) Finally create a new version applying an output block-cyclic geometric data decomposition strategy, ensuring that each processor accesses to blocks of `BS` elements of vector `V` cyclically distributed among processors. The value for `BS` should avoid the occurrence of false sharing, assuming that cache lines and doubles are `SIZE_OF_LINE` and `SIZE_OF_DOUBLE` bytes wide, respectively.

2. Assume a hash table implemented as a vector of chained lists of elements, as shown in the definition below:

```
#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 an *OUTPUT BLOCK* data decomposition for `HashTable`, where each thread has to do all the insertions that have to be done in `SIZE_TABLE/P` consecutive entries of the table `HashTable`, being P the number of threads.

3. Given the following sequential code in C that finds the first position in vector `DB` where the value stored in `key` appears:

```
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;
}
```

and the following incomplete parallelisation for the loop above:

```
  #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;
      }
```

We ask:

(a) Complete the previous incomplete solution making sure the following conditions are met, without taking care about synchronisation overheads: 1) the parallelisation obeys a *geometric block data decomposition* (i.e. each procesador is responsible for `DBsize/P` consecutive elements, being `P` the number of processors); 2) `DBsize` may not be a multiple of `P`, in which case you should take care of balancing the load assigned to processors; and 3) a processor should finish its execution as soon as it finds the value in `key` or knows that it will not contribute to the final solution.

(b) Modify your previous solution in order to substantially improve performance by minimising the serialisation introduced by the synchronisation based on `critical`.

(c) Finally, propose an alternative geometric data decomposition strategy that may reduce the execution time that is required, on average, to find the first position in which the value stored in `key` appears.

4. 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:

(a) 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).

(b) 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.

(c) 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`.

5. Given the following incomplete code:

```
static float Matrix[N][N];

void main (int argc, char *argv[]) {
  ...
  #pragma omp parallel
  {
     ...
     int i_start = ...
     int i_end   = ...
     int j_start = ...
     int j_end   = ...
     for(int i=i_start; i<i_end; i++)
        for (int j=j_start; j<j_end; j++)
           {
                ...Matrix[i][j]...
           }
  }
}
```

in which each iteration of the loop access (read access) to a `NxN` matrix, from row *i_start* to *i_end* and from column *j_start* to *j_end*. Complete the implementation of the code in `main` for two alternative data decompositions:

- Decompositon 1: `BLOCK geometric decomposition` by rows. In this case you can not assume that N is always a multiple of the number of threads. Your solution should then ensure that the load unbalance is limited to 1 row.
- Decomposition 2: `2D BLOCK geometric decomposition`, with each processor responsible for a block of N/K consecutive rows and N/K consecutive columns, being K^2 the number of processors. In this case you can assume that N is multiple of K. You can use function `sqrt` to compute the square root.

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

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

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

We ask you to:

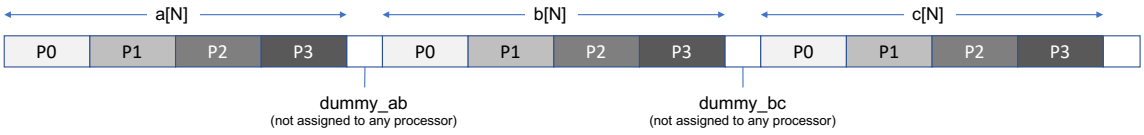
(a) Decide the most appropriate *geometric data decomposition strategy* for matrix `m` and write a parallel version of the code above using OpenMP that corresponds to it. Your solution should a) minimize the synchronization overhead among implicit tasks and b) guarantee that the load unbalance is limited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

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

8. Assume the following incomplete version for an OpenMP parallel program:

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

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

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



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

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

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

struct AoS {
     int a;
     int b;
     int c;
};
struct AoS dataElements[N];
...
```

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

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

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

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

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() {
  #pragma omp parallel
  #pragma omp single
  for (int timestep = 0; timestep < tmax; timestep++) {
    #pragma omp taskloop grainsize(1)
    for (int body = 0; body < NUM_BODIES; body++)
      forces[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 of the iterations using `taskloop` with `grainsize(1)`, which then leads to high task creation 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
  #pragma omp single
  #pragma omp taskloop grainsize(...) // 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 iteration assignment (and therefore 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).