PAR: Collection of Exercises

Eduard Ayguadé, Josep Ramon Herrero and Daniel Jiménez
Departament d'Arquitectura de Computadors

Course 2014-15 (Q1)

Index

1

(1 & 2) Concurrency and parallelism

1. **Dining philosophers.** There are 5 philosophers sitting at a round table. Between each adjacent pair of philosophers there is a chopstick. Each philosopher thinks and eats, as follows:

   - The philosopher thinks for a while;
   - When the philosopher becomes hungry, he stops thinking;
     - Picks up left and right chopstick;
     - He cannot eat until he has both chopsticks, has to wait until both chopsticks are available;
     - When the philosopher gets the two chopsticks he eats;
   - When the philosopher is done eating he puts down the chopsticks and thinks again.

   For the following implementation in Java for the problem answer the questions below:

```
import java.io.*;
import java.util.*;

public class Philosopher extends Thread {
    static final int count = 5;
    Chopstick left;
    Chopstick right;
    int position;

    Philosopher(int position,
                Chopstick left,
                Chopstick right) {
        this.position = position;
        this.left = left;
        this.right = right;
    }
```

```
public static void main(String[] args) {
        Philosopher phil[] = new Philosopher[count];

        Chopstick last = new Chopstick();
        Chopstick left = last;
        for(int i=0; i<count; i++){
            Chopstick right = (i==count-1)?last :
                                    new Chopstick();
            phil[i] = new Philosopher(i, left, right);
            left = right;
        }

        for(int i=0; i<count; i++){
            phil[i].start();
        }
    }

    ...
}
```

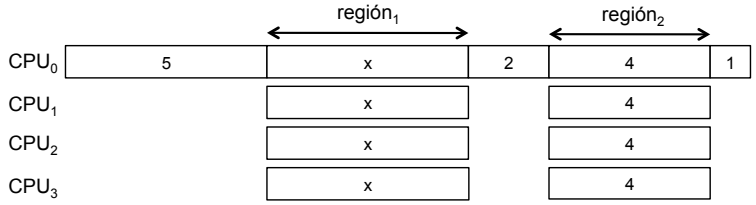
```
public void run() {
        try {
            while(true) {
1                   synchronized(left) {
2                       synchronized(right) {
3                           System.out.println(times + ": Philosopher " + position + " is done eating");
                        }
                    }
                }
            } catch (Exception e) {
                System.out.println("Philosopher " + position + "'s meal got disturbed");
            }
        }
```

   (a) Is it possible to have deadlock?

   (b) Would it be a solution if philosophers put down a fork after waiting five minutes and wait a further five minutes before making their next attempt?

   (c) Write a possible code solving the deadlock/livelock problems

2. Given the following incomplete execution timing diagram of a parallel application with 4 processors:



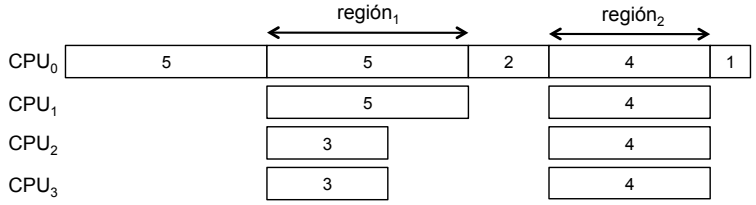
where the numbers inside the boxes represent the execution time of that application part, being this (x) unknown for the parts of the application in $region_1$. Knowing that a "speed-up" of 9 is achieved when the application is executed using infinite processors ($S_\infty = 9$, assuming that the parallel regions can be decomposed into infinity ∞), we ask:

(a) What is the parallel fraction (ϕ) of this application?

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

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

3. Given the following execution timing diagram of a parallel application with 4 processors:

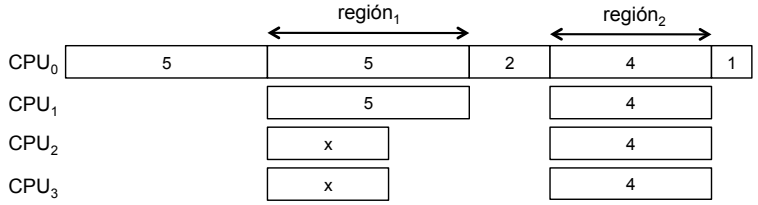


where the numbers in the boxes are the execution times of the application parts, we ask:

(a) Which "speed-up" is achieved in the execution with 4 processors (S_4)? Which is the parallel fraction (ϕ) of this application? Which is the "speed-up" that can be achieved using infinite processors (S_∞, assuming that the parallel regions can be decomposed into the ∞)?

(b) To remove the load imbalance that exists in "región1" a technique is applied that removes it at the expense of adding an "overhead" that is proportional to the number of processors ($0.05 \times p$). Which is the expression of S_p in this case?.

4. Given the following execution timing diagram of a parallel application with 4 processors:



where the numbers inside the boxes represent the execution time of that application part, being this unknown for two of the processors in $region_1$ (x can be smaller, equal or larger than 5):

(a) Knowing that the "speed-up" achieved is 2,5 when the application is executed using 4 processors ($S_4 = 2,5$), which of these statements is true?

i. `x = 4,25`

ii. `x` can have two values: 4,25 or 8.

iii. S_4 is independent of the value of `x`.

iv. None of the above.

(b) Knowing that the "speed-up" tends to 5 when the application is executed using infinite processors ($S_\infty = 5$) (assuming that, despite the imbalance in $region_1$, both parallel regions can be decomposed into the ∞). Which is the parallel fraction (ϕ) of this application?

i. 0.8
ii. 0.2
iii. Depending on the value `x` found in the previous question.
iv. None of the above.

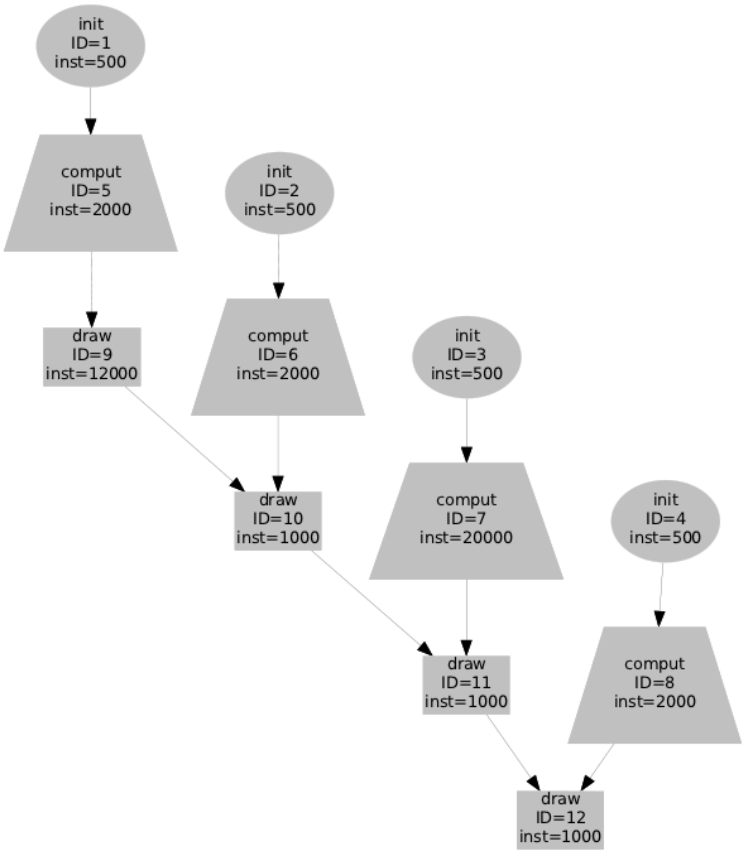
(c) What should be the value of `x` to obtain $S_\infty = 5$ in the previous question?

i. Any value equal or less than 5 (the duration of the other parts of the code from "región1").
ii. 3
iii. Is not possible to determine the value of `x` based on the information provided.
iv. None of the above.

(d) In case that `x=0`, meaning that it is not possible to distribute the workload in "región1" to more than two processors, What would be the S_∞ that could be achieved for this application?

i. 2, given that it can not be used more than two processors.
ii. 3, that is the value between 2 for the "región1" and 4 for the "región2".
iii. 2,61
iv. 4,25

5. Given the following task graph generated by *Tareador*:



We ask:

(a) Which are T_1, T_∞ and parallelism of this graph?

(b) Which is the minimum number of processors P_{min} that are necessary to achieve it?

(c) If we could perfectly balance the number of instructions executed in the 4 tasks where `comput` is done (that means that the nodes with ID=5, 6, 7 and 8 change their value and each of them has 6500 instructions), what is the new parallelism value for the resulting graph?

For an execution, using 4 processors, of the previous task graph (balanced) in which each processor executes a task sequence `init-comput-draw` (for example ID=2, 6, 10), and assuming the data sharing model explained in class based on the distributed memory architecture with the passing message where the access time to the remote data is determined by $t_{comm} = t_s + m \times t_w$, being $t_s = 1000$ and $t_w = 0.01$ the "start-up" time and the sending time of an element, respectively. We ask:

(d) Which is the total time of execution? Consider that the calculation time of an instruction is $t_c = 1$ and the number of elements m that are being communicated between processors when there is a dependency between tasks is equal to the number of instructions executed in the origin task of the dependency.

6. Given the following piece of C code in which two tasks have been identified using the tool *Tareador*:

```
#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");
   }
```

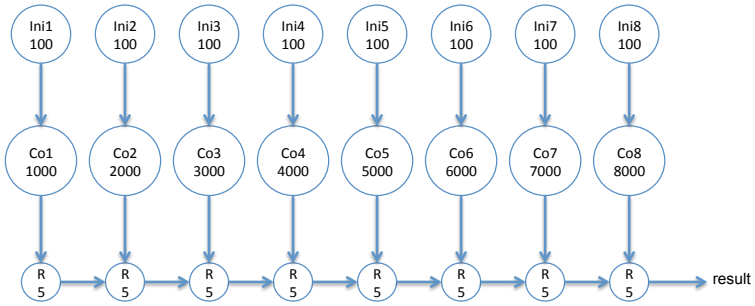
Assuming that: 1) in the initialization loop each executed iteration of the internal loop lasts 10 cycles; 2) in the calculation loop each executed iteration of the internal loop lasts 100 cycles; and 3) the execution of the `foo` function does not cause any kind of dependency, we ask:

(a) Draw the task graph, indicating for each node the number of iterations and its execution time.

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

(c) Calculate which is the best value for "speed-up" with 4 processors (S_4) that can be achieved and which would be the proper task assignment to processors to achieve it.

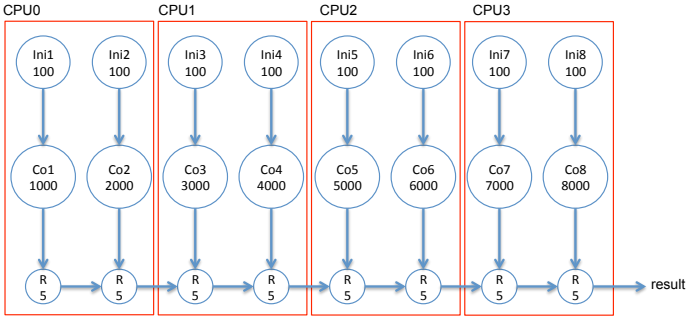
7. Given the following task graph:



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

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

(b) Calculate T_4 and the "speed–up" S_4 in an architecture with 4 processors and the assignment of tasks to processors shown below:



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

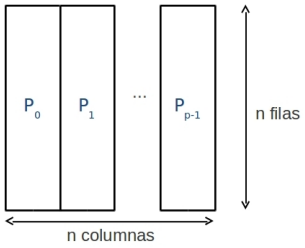
8. For each of these two codes:

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

answer these questions:

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

(b) Assuming that we are using a distributed memory machine and that we are doing a distribution of the matrices A and B by columns just as indicates the following figure:



- Determine, for each of these codes, which is the most suitable parallelization strategy: with or without blocking, the data that is necessary to communicate between processors and when they have to be communicated, and how many elements calculates each of the processors.
- Calculate T_P for each code and strategy used (the computational time and communication time model), assuming that the cost of a message of B elements is $t_s + B \times t_w$, that we have P processors and the resulting matrix is distributed between processors (there is no communication to a process).

9. Assuming that we are using 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 this code:

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

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

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

(a) Draw a matrix $N \times N$ ($N = 4$) and the existent dependences in the calculation of the elements of the matrix in the nested loop. The dependencies have to be indicated by arrows between the elements of the matrices indicating the dependence direction.

(b) Assuming that we define as tasks: 1) each iteration of the most internal loop in the nested loop (cost of an iteration: $O(100)$) and 2) each iteration of the first initializing to zero loop (cost of an iteration: $O(1)$):

- Draw the dependency graph between tasks for $N = 4$.
- Calculate T_1 and T_∞ as a function of N. How many processors guarantee that we can reach T_∞ for $N = 8$?

(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), calculate T_p applying *blocking* by columns in the rows that every processor needs to deal (B columns per block). Assume that the cost of each initialize to zero loop iteration is t_{zero} and the cost of each most internal loop iteration of the nested loops is t_c. Assume that at the end we do not have communication from all the matrix to a master process.

10. We want to find the expression that determines the parallel execution time in p processors (T_p) for the 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. Suppose also that the execution time of the iteration of the body of the most internal loop is t_c.

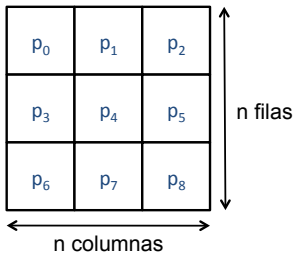
We ask complete the following table where we want to compare two different decomposition strategies of the `u` matrix: 1) *Column distribution*: the matrix `u` is distributed so that each processor has n/p consecutive columns and 2) *Row distribution*: the matrix `u` is distributed so that each processor has n/p consecutive rows. In case it is necessary to apply *blocking* for the parallel execution, consider that B is the block factor; consider that $B \approx N$ otherwise.

		Columns distribution	Rows distribution
Initial communication	Total number of messages		
	Size of each message		
	Contribution to T_p		
Blocks calculation (B)	Number of blocks		
	Number of elements in a block		
	Contribution to T_p		
Communication during the parallel calculus	Total number of messages		
	Size of each message		
	Contribution to T_p		

11. Given the following loop executed in a distributed memory architecture (message passing) where the communication time is $t_{comm} = t_s + n \times t_w$, being t_s and t_w the "start-up" and sending time of an element, and n the size of the message.

```
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;
    }
```

Suppose that the matrices `u` and `f` are distributed in blocks between the processors, just as observed in the next figure as an example using 9 processors (3^2):



And assuming that the execution time of the body of the loop is t_c and the number of processors is P^2. We ask:

(a) Which is the computation time for each processor?

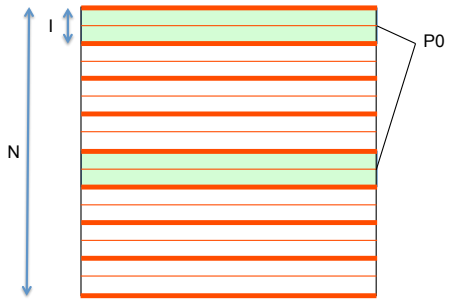
(b) Which data must be communicated between processors and when do they have to do these communications?

(c) Get the expression that determines the parallel execution time.

(d) Draw the execution timing diagram if in the previous code the matrix `f` would be the same matrix `u`, showing when the necessary communications are done:

```
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;
    }
```

12. Given the following geometrical decomposition for the matrices `a` and `b` (blocks of I consecutive rows separated by $I \times numprocs$):



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

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

To reduce the dependency effect the calculation of each processor is done in blocks of B columns (where B is a divisor of N and bigger than the number of processors). Calculate the communication cost of a parallel execution with distributed memory (start-up time t_s and element time t_e) according to the number of processors and the values of I and B (assume that the division $N/numprocs$ is an integer).

13. Given the following code:

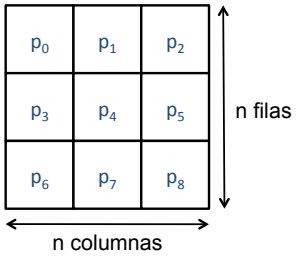
```
for (i=1; i<n-1; i+=BI) {
  for (j=1; j<n-1; j+=BJ) {
     tareador_start_task("task_control");
     for (ii=i; ii<i+BI; ii++) {
       for (jj=j; jj<j+BJ; jj++) {
          u[ii][jj] =u[ii-1][jj] + u[ii][jj-1] + u[ii+1][jj] + u[ii][jj+1] - 4*u[ii][jj];
       }
     }
     tareador_end_task("task_control");
  }
}
```

Supposing that the body of the most internal loop has a cost t_c, and BI and BJ are constants that divide the variable n, answer this questions:

(a) Draw the task graph that would be presented by the tool *Tareador*.

(b) Determine the value of T_1, T_∞ and P_{min}.

If each of the generated tasks is executed in a processor, assuming a matrix distribution between p^2 processors just like it's shown in the following figure:



Answer these questions:

(c) Assume that BJ and BI are equal to n/p:

i. Draw the execution time schedule for p^2 processors, indicating clearly what is communication time (also of what) and what is calculation time.

ii. Calculate T_p, using the data sharing model explained in class based on the shared 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 y t_w the start-up time and the sending time of an element, respectively, and being m the size of the message. Assume also that the execution time of an iteration in the body of the most internal loop is t_c.

(d) Assume now that each processor does blocking by columns in its part of the matrix, with a block size of $B = BJ/2$:

i. Draw the execution time schedule for p^2 processors, indicating clearly what is communication time (also of what) and what is calculation time.

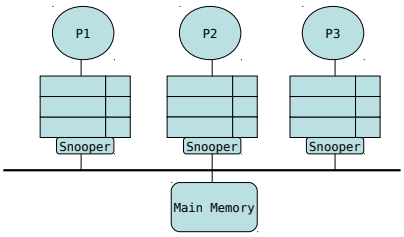
ii. Calculate T_p, using the data sharing model explained in class based on the shared memory architecture with message passing.

(e) Do you believe that a different distribution of the matrix would help to use in a better way the available processors of the machine?

3

Multiprocessor architectures

1. Given the following SMP system with 3 processors with `write-invalidate` policy.



   Let us assume all caches are initially empty and the following access sequence to the same memory direction is performed: r1, w1, r2, w3, r2, w1, w2, r3, r2, r1 (where rx: read from processor x; wy: write from processor y). Indicate

   - The *CPU event* and *Bus transaction* that are generated
   - The state of the cache line in each processor after each access to memory

   for each of the following cache coherence protocols: MSI and MESI.

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

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

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

3. Assume that a SMP multiprocessor with 4 processors, each of them with a cache memory with 4 lines, and coherence mechanism based on "snoop" with MSI protocol coherence. If you do the following sequence of reads (R) and writes (W) to variable A: $R_0, W_1, R_1, R_2, W_3, W_3, R_1, W_1, R_2$, where the sub-index indicates the processor that performs the access, and suppose that initially A is not loaded in any cache memory, explain:

   (a) The number of cache lines that have been invalidated during the sequence of operations:

   (b) The state, in each processor, of the line that contains the datum A at the end of the sequence:

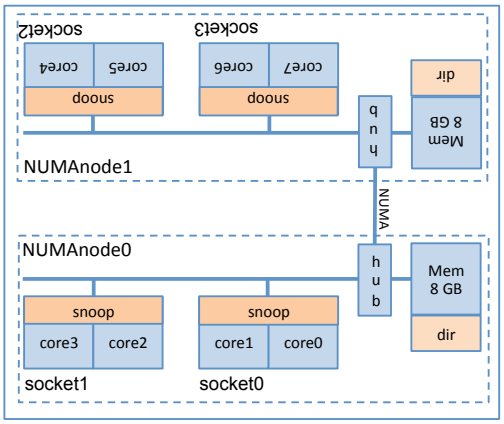
   (c) If the processor 2 performs a writing in the datum B, which is stored in a different main memory line but replaces the line where the datum A was, say the state of the cache lines that are affected:

4. Given the following fragment of a program written in *OpenMP*:

```
double vector[NTHREADS];
...
set_omp_num_threads(NTHREADS);
...
#pragma omp parallel for private(i)
    for (i=0; i<NTHREADS; i++)
#pragma omp critical
    vector[i] = foo(i);
...
```

We ask to enumerate and describe briefly the 3 most important overheads that would be observed during the execution of this fragment of the program. Indicate an order of magnitude (in time units) for each of them in the machine used to do the laboratory sessions.

5. Given the multiprocessor system architecture shown in the following figure, having coherent shared memory in a node and distributed memory between nodes:



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

Assuming that the synchronization between cores is done using atomic instructions like `test_and_set` inside each *NUMAnode* and using linked instructions like `load_linked store_conditional` between *NUMAnodes*. We ask you to indicate if these sentences are **true or false**:

(a) Inside a node the programmer does not have to worry about keeping the data coherency.

(b) The data sharing between different *NUMAnodes* that forms a node does not have any influence in the performance of the application.

(c) The directory structure present in a *NUMAnode* provides coherence information for the stored data in the memory of that *NUMAnode*.

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

(e) The atomic instructions cause a blockage of the processor that calls them in case the memory position they want to access is being accessed by another processor.

(f) The atomic instructions return in a register the value that existed in the memory position where they are writing a 1.

(g) The `store_conditional` instruction verifies that the value in a concrete memory position has not been changed regarding to the moment that the last `load_linked` has been done.

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

6. Indicate if each of these statements are **true or false**.

   (a) The access time to the data from a processor situated in the *NUMA* node that stores the coherence information of these data is always the same.

   (b) In a *Snoopy* system with *MSI* protocol, the false sharing is produced when there exists more than one modified copy of one memory line, in the caches of the processors.

   (c) In the *write-update* writing policy, to keep the coherence in the cache, a write provokes the actualization of each of the copies in the cache.

   (d) In a *NUMA* system, with the directory-based coherence protocol, the coherence information is maintained, with an amount of bits, for each of the cache lines.

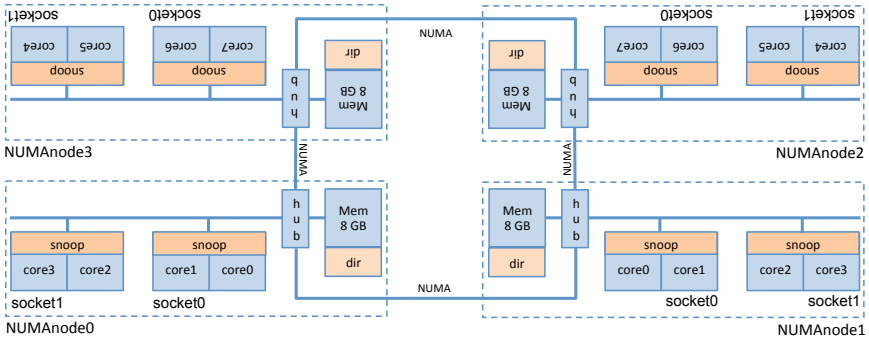
   (e) In an *SMP* system, using the coherence protocol with *write-invalidate* and MSI, the consistence in the writing is guaranteed.

   (f) In an *SMP* system, using the coherence protocol with *write-invalidate* and MSI, only one modified copy of the same memory line in the system can exist, but more than one copy can exist in a shared state inside the system.

   Finally, justify with a table that shows the actions performed in each processor, the bus transactions and the state of each cache for each operation done, if the following statement is **true or false**:

   (g) In a *SMP* system with 3 processors, (`Snoopy`, `write-invalidate` and `MSI`), the following access sequence in the same memory directions: w1, r1, w1, r2, r1, w2, w1, w3 (rx: reading of the processor x; wy: writing of the processor y), provokes 5 `BusRdX`, 3 `BusRd`, 3 `Flush` and 4 invalidations.

7. Dada la arquitectura de sistema multiprocesador mostrada en la figura siguiente, ofreciendo memoria compartida coherente entre todos los procesadores (cores) del sistema:



coreX: Núcleo.
socketY: Empaquetado con 2 núcleos, 1 MB de cache compartida entre ambos núcleos y snoopy.
NUMAnodeW: conjunto de 2 sockets conectados a un mismo "hub"/directorio NUMA con 8 GB de memoria principal.

   (a) En el sistema de la figura cada core ...

      i. ... sólo puede acceder a los datos que residen en la memoria de su *NUMAnode*
      ii. ... tiene acceso a 32 GB de memoria física
      iii. ... sólo puede acceder a los datos que residen en la memoria de los *NUMAnodes* directamente conectados a su *NUMAnode*
      iv. Todas las anteriores son ciertas

   (b) En un sistema "snoopy" con protocolo MSI (el explicado en clase) ...

      i. ... el comando de invalidación se genera cada vez que un core escribe un dato en su memoria cache
      ii. ... el comando de *flush* se genera para actualizar todas las copias de una linea de cache dentro del *NUMAnode*, incluyendo su memoria principal
      iii. ... un comando de *flush* dentro de un *NUMAnode* nunca dispara una acción de coherencia con el resto de *NUMAnodes*

iv. Ninguna de las anteriores es cierta

(c) El reparto de los datos entre las memorias de los distintos *NUMAnodes* ...

i. ... no tiene ninguna influencia en el rendimiento de la aplicación

ii. ... lo realiza en sistema operativo en base a una política predefinida

iii. ... va cambiando dinámicamente con el objetivo de mantener equilibrados el número de accesos a memoria que realizan los cores de cada *NUMAnode*

iv. Ninguna de las anteriores es cierta

(d) El número de entradas en el directorio de un *NUMAnode* ...

i. ... coincide con el número de lineas que caben en su memoria principal asociada

ii. ... coincide con el número total de lineas que caben en las memorias principales de todos los *NUMAnodes*

iii. ... queda determinado por el máximo número de cópias que se permiten de cada línea de cache

iv. Ninguna de las anteriores es cierta

(e) En cuanto a los mecanismos de sincronización entre cores es cierto que ...

i. ... las instrucciones atómicas bloquean al procesador que las invoca en caso de que la posición de memoria a la que se quiere acceder este siendo accedida por otro procesador

ii. ... la instrucción `store_conditional` verifica que el valor que hay en una posición de memoria concreta no haya cambiado respecto al momento que se hizo el ultimo `load_linked`

iii. ... son necesarias para garantizar la consistencia de memoria

iv. Todas las anteriores son ciertas

(f) Un problema de "false sharing" ...

i. ... se puede eliminar protegiendo el acceso a las variables con instrucciones de sincronización

ii. ... sólo aparece cuando se accede a elementos consecutivos de un vector o matriz

iii. ... se agrava cuando ocurre entre *NUMAnodes*, debido a la no uniformidad en el tiempo de acceso a memoria

iv. Todas las anteriores son ciertas

8. Marca cuál o cuáles de las siguientes afirmaciones son ciertas.

(a) El "false sharing" siempre aparece cuando dos threads acceden a dos elementos consecutivos de un vector.

(b) En un sistema *Snoopy* con protocolo *MESI*, la exclusividad de una línea de cache se obtiene por parte de un procesador cuando no hay copias modificadas de esa línea.

(c) `Store Conditional` en un procesador te garantiza que la operación de escritura a una dirección de memoria se hace de forma atómica con una operación `Load Linked`, en el mismo procesador, a la misma dirección de la memoria.

(d) En un sistema *NUMA*, la implementación de un `lock` usando `test&set` es más eficiente que usar un `test_and_test&set` o `load_linked` más `store_conditional`. Con `test&set`, a diferencia de con las otras dos estrategias, no se tiene que mantener la coherencia de cache.

(e) En un sistema *NUMA* con un protocolo de coherencia basado en directorios, cada *NUMAnode* guarda información de las lineas de memoria que hay en las caches de los procesadores del *NUMAnode*.

(f) En un sistema *Snoopy* con protocolo *MSI*, la consistencia de memoria se garantiza gracias a las invalidaciones que se realizan cuando un procesador intenta hacer una lectura para modificar (*BusRdX*).

(g) El *overhead* de creación de una región paralela es del orden de *nanoseconds*.

(h) En un sistema *UMA* con un sistema *Snoopy* con protocolo *MSI* sólo puede haber una linea de cache modificada en todo el sistema.

(i) *Data Race* se produce cuando uno o más threads acceden a un mismo dato en el mismo *NUMAnode*.

(j) El "true sharing" se puede evitar utilizando *padding*, de esta forma evitamos que dos threads accedan a la misma línea de cache.

4

Task decomposition

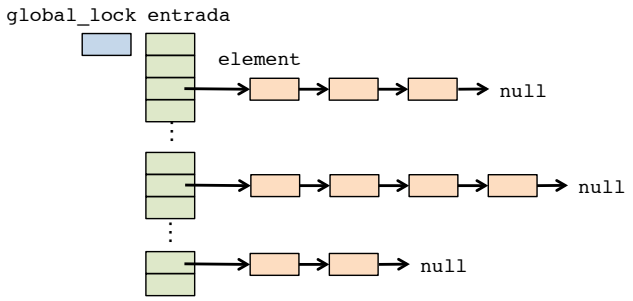
1. Given the following code:

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

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

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

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

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

HashTable table;
```



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

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

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

We ask:

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

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

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

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

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

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

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

for (j=0; j<NPOP; j++) {
       iother = rand(j) // returns random value >= 0, != j, < NPOP

       // Swap Chromosomes
       temp(1:NCHROME) = iparent(1:NCHROME, iother);
       iparent(1:NCHROME, iother) = iparent(1:NCHROME, j);
       iparent(1:NCHROME, j) = temp(1:NCHROME);

       // Swap fitness metrics
       tempScalar = fitness(iother);
       fitness(iother) = fitness(j);
       fitness(j) = tempScalar;
} // end loop [j]
```

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

4. Make an OpenMP parallelization of the following code having in mind the synchronization costs:

```
    for (i=1; i<N; i++)
      for (j=1; j<N; j++)
         A[i][j]=A[i-1][j]+A[i][j-1];
```

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

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

void main() {
   ....
   sum = sum_vector(v,N);
   ...
}
```

   (a) Write **two parallel versions** for the `sum_vector` function that fulfill the following conditions:
      i. Use a linear or iterative task decomposition parallelization strategy.
      ii. Do not do explicit synchronizations inside the `for` loop.
      iii. Don't cause false sharing.

      You can assume that you have a `define NTHREADS` that indicates the number of threads that will be used on the parallel execution.

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

   (c) Write **a parallel version** of the new function `recursive_sum_vector` that fulfills the following conditions:
      i. Use a task decomposition of the recursive type that uses the "divide and conquer" pattern.
      ii. The T_p when using p processor has to approximate to $T_\infty = \log(n)$.

6. We have a `to_lowercase` function, that converts to lower-case all the characters in a string with length `n`, and a `recursive_to_lowercase` function that searches sentences finished with a dot in a string,and converts them to lower-case using the `to_lowercase` function. Given the following "divide and conquer" implementation in *OpenMP*:

```
void to_lowercase(int n, char *cadena);
void recursive_to_lowercase (int n, char * cadena, int d) {
      if (d < cutoff) {
            int point = find_dot (n, cadena); // returns position of first dot in string
            if (point != (n-1))
                  #pragma omp task
                  recursive_to_lowercase(n-point-1, &cadena[point+1], d+1);
            #pragma omp task
            to_lowercase(point+1, cadena);
      } else
            to_lowercase(n, cadena);
}

int n, cutoff;
char * cadena;
void main() {
      ... // allocation and initialization of string
      cutoff = ...;

      #pragma omp parallel
      #pragma omp single
      recursive_to_lowercase(n, cadena, 0);
}
```

(a) Indicate which of the following sentences is true, assuming that `OMP_NUM_THREADS=4`:

i. The number of explicit tasks that will be created is at most 4, just as the environment variable `OMP_NUM_THREADS`.

ii. There is only one thread that creates and executes tasks because there is one "`#pragma omp single`".

iii. The code is not correct because there is no `taskwait` that guarantees the dependencies.

iv. None of the sentences above are correct.

(b) Knowing that the `to_lowercase(n,c)` execution lasts `n` time units, the `m=find_dot(n,c)` lasts `m` time units, and the creation of threads "threads" in a parallel lasts 50 time units and the creation of the tasks lasts 5 time units, what would be the parallel execution time with infinite processors for a string with 50 sentences with 10 characters (including the dot) each and `cutoff=10000`:

i. Less than 25 time units.

ii. Between 25 and 750 time units.

iii. More than 750 time units.

iv. None of the above.

(c) Knowing that the total number of tasks generated in the parallel execution is 16, what is the `cutoff` that has been used?

i. 8

ii. 16

iii. $cutoff = omp\_get\_num\_threads()$;

iv. $cutoff = 2 * omp\_get\_num\_threads()$;

7. 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);
     #pragma omp taskwait
   }
}


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 three different ways, being the following: a) depth of the recursivity tree (`MAX_DEPTH`); b) size of the vector to sort (`VECTOR_SIZE`); and c) number of pending tasks to be executed not higher than `MAX_TASKS` multiplied by the number of available threads.

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;
    int i;

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

    return count;
}

int main() {
   long a[N], key = 42, nkey=0;
   int  i;

   // fill the array and make sure it has a few instances of the key
   for (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 in a using an iterative decomposition
   nkey = count_iter(N, a, key);

   // count key in a with divide and conquer
   nkey = count_recur(N, a, key);
}
```

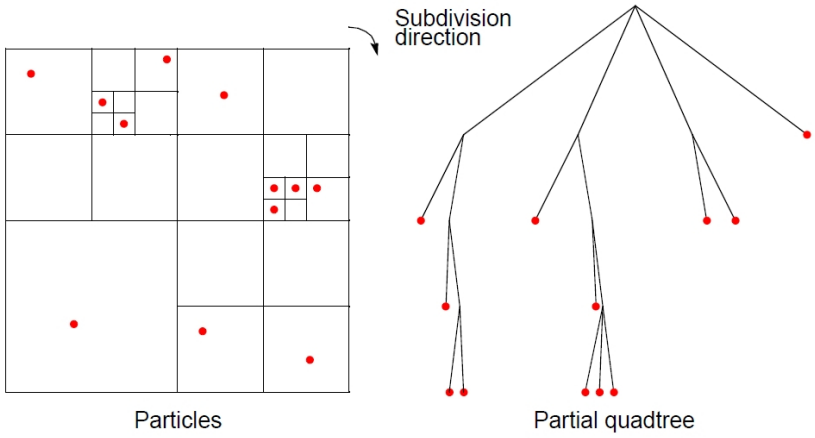
(a) Write a parallel OpenMP version using an iterative task decomposition (`count_iter`)

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

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

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

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



We ask:

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

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

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

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

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

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

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

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

  // write first position, if found
}
```

We ask:

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

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

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

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

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

5

Data decomposition

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

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

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

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

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

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

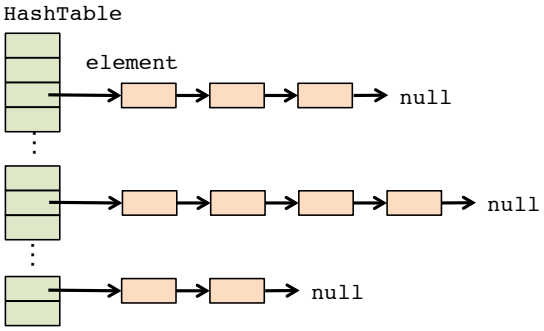
   assuming that vectors A and B are geometrically decomposed in blocks among all processors as indicated in the following data structure

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

limits decomposition[num_threads];
```

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

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

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

element * HashTable[SIZE_TABLE];
```

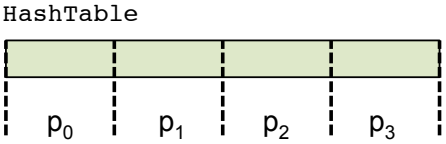


Taking as a starting point the following sequential code:

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

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

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



3. Assume the next sequential code written in C that finds in a vector `DB` the first position where a determined `key` appears:

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

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

  // write first position, if found
}
```

We propose a parallelization of the code using a schema where the distribution of iterations to threads obeys a geometric decomposition of the data to blocks, using `#pragma omp parallel` for the creation of threads:

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

We ask:

(a) Complete the previous *OpenMP* code (without modifying the `for` loop that accesses the vector) in a way that the parallel version gives the same result as the sequential one.

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

4. 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:

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

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

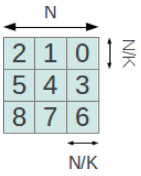
Hint: You have to do it in such a way that each thread will have one and only one Task assigned (for example: $task_0 \rightarrow thread_0$). Consequently, for the general case we need as many threads as $(N/BS) \times (N/BS)$, the total of submatrices of $BS \times BS$ elements of C.

5. Dado el siguiente código incompleto:

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

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

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

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

- Implementación 1: La inicialización de la estructura `decomposition` en la función `InitDecomposition` da lugar a una `Block Geometric (Data) decomposition` por filas. En este caso NO puedes suponer que N sea múltiplo del número de `threads`. Además se quiere que el desbalanceo de carga entre threads sea no superior a una fila de cálculo.
- Implementación 2: La inicialización de la estructura `decomposition` en la función `InitDecomposition` da lugar a una descomposición geométrica en la que cada thread trata un bloque de la matriz tal y como muestra la figura.



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

6. La función *MPI_Scatter* es una función colectiva de `MPI` que reparte equitativamente los elementos de un vector de un procesador al resto de procesadores. Por ejemplo el siguiente código:

```
MPI_Scatter(A,sizeToBeSent,MPI_DOUBLE,A,sizeToBeSent,MPI_DOUBLE,0,MPI_COMM_WORLD)
```

hace que el proceso 0 (parámetro con valor 0 de la llamada) distribuya los elementos de A (de tipo DOUBLE) de tal forma que cada proceso de los *nproc* procesos que estan en el contexto de comunicación `MPI_COMM_WORLD` reciba `sizeToBeSent` (el proceso 0 no se comunica con el mismo).

Suponiendo que implementáramos esta función con las funciones de comunicación `point to point` `MPI_Send` y `MPI_Recv`, cuyas cabeceras son:

```
int MPI_Send(buffer, count, datatype, dest, tag, comm);
int MPI_Recv(buffer, count, datatype, source, tag, comm, status);
```

Contestad a las siguientes preguntas:

- ¿Cúal sería el coste de comunicación de esa implementación con comunicaciones punto a punto? Realiza el cálculo suponiendo el modelo de compartición de datos explicado en clase basado en una arquitectura de memoria distribuida y con paso de mensajes en el que el tiempo de acceso a datos remotos viene determinado por $t_{comm} = t_s + m \times t_w$, siendo t_s y t_w los tiempos de "start-up" y de envío de un elemento, respectivamente, y m el tamaño del mensaje.
- ¿Cómo puede influir la red de interconexión en el coste real de estas comunicaciones?