PAR
Selection of Exams and their Solution

Eduard Ayguadé, José R. Herrero and Daniel Jiménez

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

Course 2014-15 (Q1)

Contents

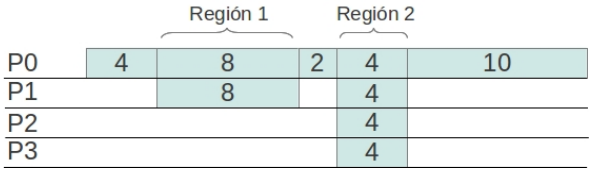
Part I

1^{st} In-term Exams

Primer Control de PAR – Curso 2013/14-Q1

28 de Octubre de 2013

1. (2 puntos) Dado el siguiente diagrama de ejecución de una aplicación paralela en 4 procesadores, donde el número en cada ráfaga indica su coste:



Un estudiante, observando las dos regiones paralelas, ha calculado S_4 aplicando $S_p = \frac{1}{(1-\phi)+\phi/p}$, donde p es el número de procesadores. Responde a las siguientes preguntas:

   (a) Calcula el valor ϕ en función de los datos del diagrama.

   (b) Indica si la forma en que el estudiante ha calculado S_p es correcta. En el caso de que no sea correcta, indica en qué circunstancias lo sería.

   (c) Calcula el valor de S_∞ suponiendo que las dos regiones pueden escalar hasta el infinito.

   (d) Suponiendo que cada ráfaga del diagrama es una tarea, dibuja el grafo de tareas y dependencias entre ellas que sea coherente con el diagrama de ejecución de la figura.

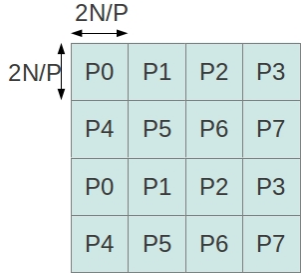
2. (5 puntos) Dado el siguiente código:

```
for (i=1; i<N-1; i++) {
    for (k=1; k<N-1; k++) {
        u[i][k] = 0.4*u[i][k-1]
                + 0.8*u[i][k+1]
                + 0.5*u[i+1][k]
                - 0.2*u[i][k];
    }
}
```

Responde a las siguientes preguntas:

   (a) Suponiendo que definimos tarea como el cuerpo del bucle más interno y que ésta tiene un coste de t_c. Calcula T_1 y T_∞.

   (b) **Se quiere calcular** T_8 utilizando el modelo de compartición de datos explicado en clase basado en una arquitectura de memoria distribuida y con paso de mensajes. Para ello **se pide** que completéis la siguiente tabla. 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 siendo m el tamaño del mensaje. Suponed también que el tiempo de ejecución de una iteración del cuerpo del bucle más interno es t_c, y que la matriz u está distribuida según la siguiente figura entre los $P = 8$ procesadores (P divide perfectamente N). En el cálculo en los procesadores no se realiza ningún tipo de *blocking*, de tal forma que cada procesador tiene asignadas dos submatrices y las procesa sin hacer bloques:



Comunicación inicial	Número total de mensajes	
	Tamaño de cada mensaje	
	Contribución a T_p	
Cálculo	Coste de procesar una submatriz	
	Contribución a T_p	
Comunicación durante cálculo paralelo	Número total de mensajes	
	Tamaño de cada mensaje	
	Contribución a T_p	

3. (3 puntos) Dado el siguiente fragmento de código en C con directivas OpenMP, y suponiendo que estamos en una máquina *UMA* con 16 procesadores, con protocolo de coherencia *MSI* y política *Write-Invalidate*:

```
int sum,i;
int sum_vector[NUM_THREADS};
...
sum=0;
#pragma omp parallel
{
  int id = omp_get_thread_num(); // thread_id
  sum_vector[id]=0;
  #pragma omp for
  for (i=0; i<N; i++)
  {
    #pragma omp critical
    {
      if (sum+foo(i)<MAX_SATURACION)
         sum+=foo(i);
      else sum_vector[id]+=foo(i);
    }
  }
}
```

Se sabe que hay dos líneas que provocan dos *overhead* significativos para N muy grandes: la del *critical* y la del *else*.

(a) Suponed que *foo(i)* no hace accesos a memoria y que el primer elemento del vector *sum_vector* se guarda en una dirección de memoria que es inicio de línea de memoria (cache). Muestra los eventos de procesador y transacciones de bus debidos a accesos a memoria, e indica si el dato

está en cache o no de los procesadores involucrados para la siguiente secuencia de ejecuciones de la línea del código *else*: t0, t1, t0, t1, t1, t0. El orden en la secuencia indica orden en el tiempo, y tX indica thread X ejecuta la línea *else*, y cada thread se ejecuta en un procesador diferente con su cache asociada.

(b) Proponed una código alternativo que reduzca significativamente el *overhead* innecesario en las dos líneas indicadas. Pista para una de ellas: pensad en la técnica de test-test-and-set utilizada para reducir el coste del test-and-set.

Solution

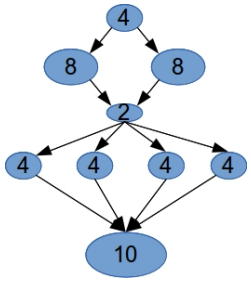
1. (2 puntos)

   (a) $\phi = \frac{8+8+4+4+4+4}{4+8+8+2+4+4+4+4+10} = \frac{32}{48} = \frac{2}{3}$

   (b) No es correcta. Seria correcta si la repartición de trabajo del trabajo de la región 1 fuera igual para los $p = 4$ procesadores. En este caso sólo se distribuye el trabajo entre dos de los cuatro procesadores.

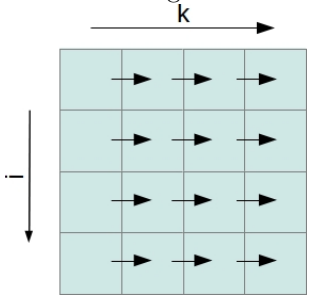
   (c) $S_p = \frac{1}{(1-\phi)+\phi/p}$
   $S_\infty = \frac{1}{(1-\phi)} = \frac{1}{1-2/3} = \frac{1}{1/3} = 3$

   (d) Grafo de tareas y dependencias coherente con el diagrama de ejecución de la figura:



2. (5 puntos)

   (a) Solución: ($\simeq$ para N muy muy grandes)
   Las dependencias entre iteraciones es la siguiente.

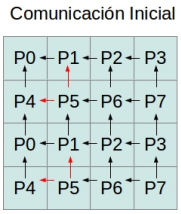


   $T_1 = (N-2) \times (N-2) \times t_c \simeq N^2 \times t_c$
   $T_\infty = (N-2) \times t_c \simeq N \times t_c$

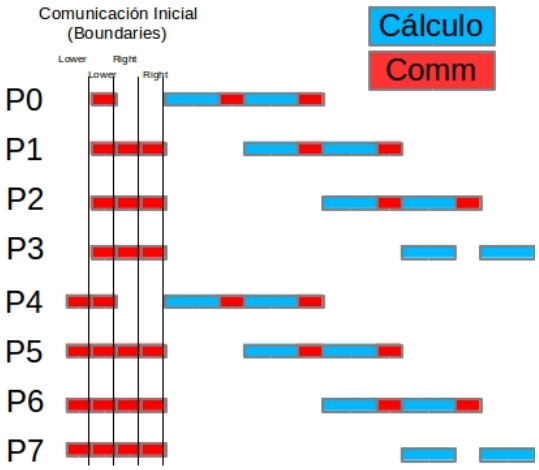
(b) Solución: (consideramos N muy muy grandes)

Comunicación inicial	Número total de mensajes	24
	Tamaño de cada mensaje	$2\frac{N}{P}$
	Contribución a T_p	$4 \times (t_s + 2\frac{N}{P}t_w)$ debido a P_5 o P_6 o P_7
Cálculo	Coste de procesar una submatriz	$2\frac{N}{P} \times 2\frac{N}{P} \times t_c$ (en general)
	Contribución a T_p	$(P/2) \times (2\frac{N}{P})^2 \times t_c + (2\frac{N}{P})^2 \times t_c$
Comunicación durante cálculo paralelo	Número total de mensajes	$(P-2) \times 2 = 12$
	Tamaño de cada mensaje	$2\frac{N}{P}$
	Contribución a T_p	$((P/2)-1) \times (t_s + 2\frac{N}{P}t_w) + (t_s + 2\frac{N}{P}t_w)$

Los datos de esta tabla se han basado en esta comunicación inicial:



y en este diagrama de tiempo:



3. (3 puntos)

(a) Suponemos que thread 0 tiene los datos en la cache. Y que la línea en la cache del thread 0 está en estado **modified**.

i. t0: `PrRd` y `PrWr` thread 0. No hay transacciones en el bus. El estado de la línea de cache se mantiene: `modified`

ii. t1: `PrRd` procesador t1, `BusRd` procesador t1. Snoopy t0 realiza `Flush`. Línea de cache pasa a estado `shared` en ambos procesadores de t0 y t1. `PrWr` procesador t1, `BurRdX` procesador t1. Snoopy t0 invalida línea de cache en procesador t0. Linea de cache en t1 se pone a `modified`.

iii. t0: `PrRd` procesador t0, `BusRd` procesador t0. Snoopy t1 realiza `Flush`. Línea de cache pasa a estado `shared` en ambos procesadores de t0 y t1. `PrWr` procesador t0, `BurRdX` procesador t0. Snoopy t1 invalida línea de cache en procesador t1. Linea de cache en t0 se pone a `modified`.

iv. t1: Lo mismo que paso ii.

v. t1: `PrRd` y `PrWr` thread 1. No hay transacciones en el bus. El estado de la línea de cache se mantiene a `modified` para el procesador del t1.

vi. t0: Lo mismo que paso iv.

(b)
```
int sum,i;
#define PADDING (CACHE_LINE_SIZE/sizeof(int))
int sum_vector[NUM_THREADS][PADDING];
...
sum=0;
#pragma omp parallel
{
  int id = omp_get_thread_num(); // thread_id
  sum_vector[id][0]=0;
  #pragma omp for
  for (i=0; i<N; i++)
  {
    if (sum+foo(i)<MAX_SATURACION) /* First test */
    {
      #pragma omp critical
      {
        if (sum+foo(i)<MAX_SATURACION) /* Second test */
           sum+=foo(i);
        else sum_vector[id][0]+=foo(i);
      }
    }
    else sum_vector[id][0]+=foo(i);
  }
}
```

Parallelism - 1^{st} In-term Exam - 2013/14 -Q1

October 28th, 2013

1. (2 points)

   (a) If we only optimize one routine that amounts to 90% of the execution time of a program, which is the maximum speed-up that we can get?

   (b) Assuming no parallelization overheads, which would be the efficiency using 10 processors?

   (c) Which is the difference between strong and weak scalability?

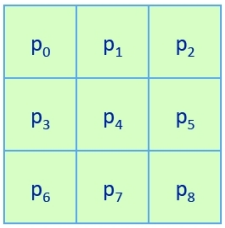
   (d) Explain the Load-linked Store-conditional (ll-sc) pair of instructions and the motivation to provide them in a shared memory system.

2. (5 points) We have a distributed memory architecture with message passing, where each message of m elements has a communication time cost of $t_{comm} = t_s + m \times t_w$.

   (a) (2 points) 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;
   }
}
```

   let us assume that matrices u and f are distributed in a 2D block partition among the processors. As an example, consider the case with 9 processors (3^2) as shown in the following figure:



   Assuming that the execution time of the loop body is t_c, the number of processors is P^2 and the matrix has n rows and n columns, being n large, 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) (3 points) Continuing with the same assumptions above, 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).

iii. Given a fixed number of processors, would you chose this 2D distribution or, instead, the 1D row-wise distribution seen in the examples in the course slides where each group of n/P consecutive rows (segment) assigned to a processor is logically divided in blocks of B columns? Justify your answer.

3. (3 points) Given an SMP system with 3 CPUs, each with a cache memory initially empty, `Snoopy`, `write-invalidate` protocol and `MESI` coherence protocol, and assuming the following access sequence to the same memory direction: r1, w1, r2, w3, r2, w1, w2, r3, r2, r1 (rx: reading of the processor x; wy: writing of the processor y), fill in the table indicating the

   (a) *CPU event* (PrRd,PrWr)

   (b) *Bus transaction(s)* (BusRd, BusRdX, flush)

   (c) State of the cache line (M, E, S, I) in each processor after each access to memory

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

Solution

1. (2 points)

   (a) The ideal speed-up is $S_\infty = \frac{1}{1-\phi} = \frac{1}{0.1} = 10$

   (b) $Eff_P = \frac{S_P}{P} = \frac{\frac{1}{1-0.9+\frac{0.9}{P}}}{P} = \frac{\frac{1}{0.19}}{10} = 0.52$

   (c) Strong scaling: How the solution time varies with the number of processors for a fixed total problem size

   Weak scaling: how the solution time varies with the number of processors for a fixed problem size per processor (the task granularity is constant)

(d) Load-Link and Store-Conditional (LL/SC) are a pair of instructions used in multi-threading to achieve synchronization. Load-link returns the current value of a memory location, while a subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-link.

(Read) Load Linked: Track reads and writes to a memory location

(Modify) Instruction(s) that create value to store

(Write) Store Conditional: Store fails if tracked memory address was read or written and the sequence is executed again

Atomicity is difficult or inefficient in large systems. Together, LL/SC implement a lock-free atomic read-modify-write operation.

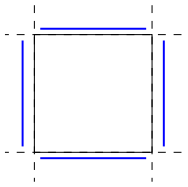
2. (5 points)

In all questions: Since n is large we consider $n - 2 \approx n$

(a) (2 points)

i. Data received:

Each processor will need to receive the boundary elements from its neighbor processors. The boundary consists of n/P elements in the first—last column—row in the block. Since the source matrix is not modified, all the boundaries can be communicated initially.



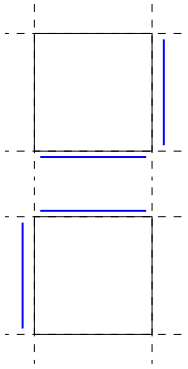
ii. $T_{P^2} = \underbrace{4 \times (t_s + \frac{n}{P} \times t_w)}_{T_{Initial\_Comm}} + \underbrace{\frac{n^2}{P^2} \times t_c}_{T_{Comp}}$

(b) (3 points)

i. Each processor will need to receive the boundary elements from its neighbor processors. The boundary consists of n/P elements in the first—last column—row in the block. Since the source matrix is modified, we need to distinguish the boundaries that can be communicated initially from those that need to be sent only once they have been modified, respecting data dependencies.

Initially:
the boundaries received from the right and the bottom can be communicated.

During the execution of the loops:
the boundaries received from the left and the top can be communicated after the corresponding blocks have been updated.



ii. $T_{P^2} = \underbrace{2 \times (t_s + \frac{n}{P} \times t_w)}_{T_{Initial\_Comm}} + \underbrace{(2P-1) \times \frac{n^2}{P^2} \times t_c}_{T_{Comp}} + \underbrace{(2P-2) \times 2 \times (t_s + \frac{n}{P} \times t_w)}_{T_{Other\_Comm}}$

iii. The 1D row-wise distribution would be preferable for two reasons:

- Higher Parallelism: with the 1D distribution we can reach a maximum of P^2 processors working simultaneously, while with the 2D distribution proposed we can only reach at most P.
- Lower interprocessor communication: the 1D distribution only requires communication of rows while the 2D distribution requires communication of rows and columns.

3. (3 points)

Memory Access	CPU event	Bus transaction(s)	Cache Line State		
			Mem1	Mem2	Mem3
r1	PrRd	BusRd	E	I	I
w1	PrWr	-	M	I	I
r2	PrRd	BusRd + Flush	S	S	I
w3	PrWr	BusRdX	I	I	M
r2	PrRd	BusRd + Flush	I	S	S
w1	PrWr	BusRdX	M	I	I
w2	PrWr	BusRdX + Flush	I	M	I
r3	PrRd	BusRd + Flush	I	S	S
r2	PrRd	-	I	S	S
r1	PrRd	BusRd	S	S	S

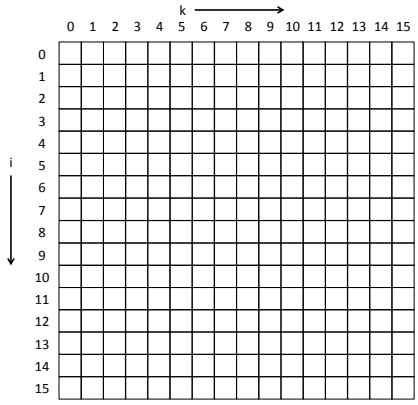
Primer Control de PAR – Curso 2013/14-Q1

28 de Octubre de 2013

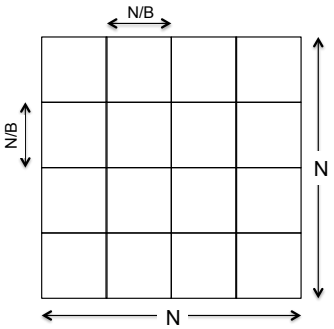
Pregunta 1 (6 puntos) Dado el código secuencial que realiza el cáculo de los elementos de la matriz `u`, en el que el bucle k va de 1 hasta la diagonal principal ($k = i$):

```
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;
    }
```

1. Para el caso concreto de N=16, dibujar el espacio de iteraciones que se recorre para la realización del cáculo, sombreando aquellas casillas para las cuales se ejecuta el cuerpo del bucle.



2. Dibujar a la derecha el grafo de dependencias entre tareas que resultará si se aplica una descomposición del espacio de iteraciones en tareas según el patrón mostrado a continuación (para N=16 y B=4),

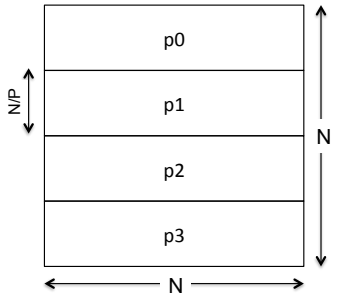


   indicando para cada nodo el número de iteraciones que ejecuta la tarea asociada.

3. Para el grafo de tareas obtenido, el valor de T_1 es:
   (a) 105 (c) 196
   (b) 136 (d) Ninguno de los anteriores

4. Y el valor de T_∞ es:
   (a) 64 (c) 46
   (b) 60 (d) Ninguno de los anteriores

Suponiendo que la asignación de tareas a procesadores se realiza según el patrón mostrado a continuación, para $P = 4$,

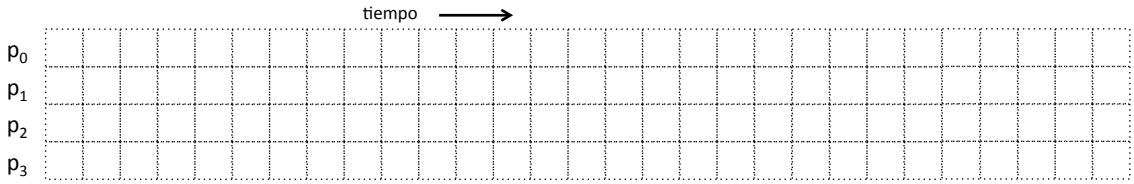


y suponiendo también que:

- 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 siendo m el tamaño del mensaje.
- El tiempo de ejecución de una iteración del cuerpo del bucle más interno es t_c.
- En caso de requerir *"blocking"* (para mitigar el efecto de posibles dependencias de datos) el número de bloques es $B = P$.
- N es un valor muy grande (en general, $N >> P$).

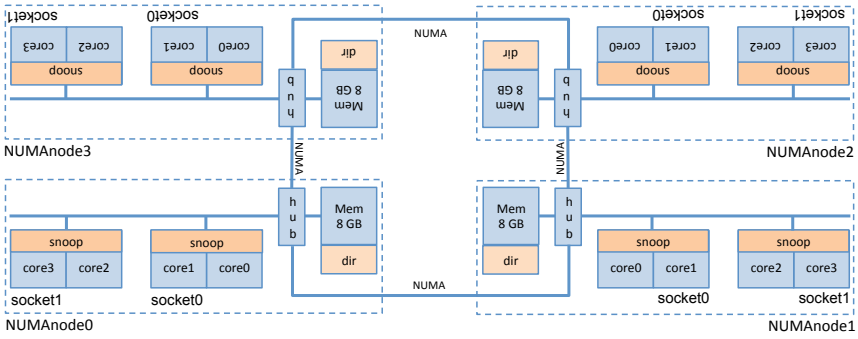
Se pide:

5. Dibujar el cronograma de ejecución que os permita calcular T_4.



6. Obtener la contribución a T_4 del tiempo de cálculo, en función del tamaño de la matriz N.
7. Obtener la contribución a T_4 del *"overhead"* provocado por la comunicación que sea necesaria realizar antes de iniciar el cálculo paralelo, también en función del tamaño de la matriz N.
8. Obtener la contribución a T_4 del *"overhead"* provocado por la comunicación que sea necesaria realizar durante el cálculo paralelo, de nuevo en función del tamaño de la matriz N.

Pregunta 2 (3 puntos) 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 (protocolo write-invalidate MSI).
NUMAnodeW: conjunto de 2 sockets conectados a un mismo "hub"/directorio NUMA con 8 GB de memoria principal.

Indica la respuesta correcta para cada una de las siguientes 6 cuestiones *(Nota: cada cuestión bien contestada suma 0.5 puntos, pero resta 0.15 puntos en caso de estar mal contestada*

1. La memoria cache de un *socket* ...
   (a) ... se mantiene coherente sólo con la memoria cache del otro *socket* en el mismo *NUMAnode*
   (b) ... sólo podrá tener en estado M aquellas líneas para las cuales su *NUMAnode* es el nodo *Home*
   (c) ... sólo puede acceder a los datos en líneas de cache que residen en sus dos *NUMAnodes* adyacentes
   (d) Ninguna de las anteriores es cierta
2. El reparto de los datos entre las memorias de los distintos *NUMAnode* ...
   (a) ... se determina en tiempo compilación, en base a los datos que utilizan las tareas paralelas
   (b) ... se determina durante la ejecución del programa paralelo, en base a algún parametro de configuración del sistema operativo
   (c) ... va cambiando dinámicamente con el objetivo de mantener equilibrados el número de accesos a memoria que se realizan desde los distintos *NUMAnode*
   (d) ... no tiene ninguna influencia en el tiempo de ejecución del programa paralelo dado que el hardware se encarga, en cualquier momento, de asegurar el acceso coherente a los mismos
3. El número de entradas en el directorio de cada *NUMAnode* ...
   (a) ... coincide con la suma del número de líneas que caben en las memorias de todos los *NUMAnodes*
   (b) ....coincide con el número de líneas que caben en cada una de las memorias cache de un *socket*
   (c) ... coincide con el número de líneas que caben en la memoria principal del *NUMAnode*
   (d) Ninguna de las anteriores es cierta
4. Un problema de "false sharing" ...
   (a) ... aparece cuando *core*s en *socket*s distintos en un *NUMAnode* acceden a una misma variable
   (b) ... aparece cuando *core*s en *socket*s distintos en un *NUMAnode* acceden a dos variables cuyas direcciones de memoria están separadas un número de bytes inferior al tamaño de línea de cache
   (c) ... no puede aparecer entre accesos provocados en el seno de *NUMAnode* distintos dado que el carácter NUMA de la arquitectura hace que se utilicen políticas de escritura "write-through"
   (d) Todas las anteriores son falsas
5. Suponiendo que en un momento dado existen cópias de la variable `var` en las memorias cache de los *socket0* y *socket1* del *NUMAnode0*, cuando el *core1* de dicho *socket0* realiza una escritura en la variable `var`, ¿cuál de las siguientes acciones NO ocurre?
   (a) El *core1* provoca un `PrWr`
   (b) El snoopy del *socket0* provoca un `BusRdX`
   (c) El snoopy del *socket1* hace un `flush` de la línea que contiene `var` provocando que el estado de la línea pase de S a I
   (d) El *hub* asociado al *NUMAnode0* asegurará la coherencia con otros posibles *NUMAnode*, si hubiera cópia de `var` en los mismos

6. Si a continuación el *core0* del *NUMAnode3* realiza una lectura de la misma variable `var`, ¿cuál de las siguientes acciones NO ocurre?

   (a) El snoopy del *socket0* en el *NUMAnode3* provoca un BusRd

   (b) El *hub* en el *NUMAnode3* envia un `RdReq` al resto de *NUMAnodes* para localizar la ubicación más cercana de la línea que contiene la variable `var`

   (c) El hub en el *NUMAnode0* genera un `BusRd`

   (d) El snoopy del *socket0* en el *NUMAnode0* hace un `flush` de la línea correspondiente, provocando que tanto el estado de la línea en el *socket0* como en el directorio del *NUMAnode0* se actualice pasando de M a S

Pregunta 3 (1 punto) Dado el siguiente fragmento de código en C con directivas OpenMP y suponiendo que cada variable de tipo *int* ocupa 4 bytes, que una línea de *cache* ocupa 32 bytes: y que la dirección inicial de `vector` y `contar` están alineadas con el inicio de linea de cache:

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

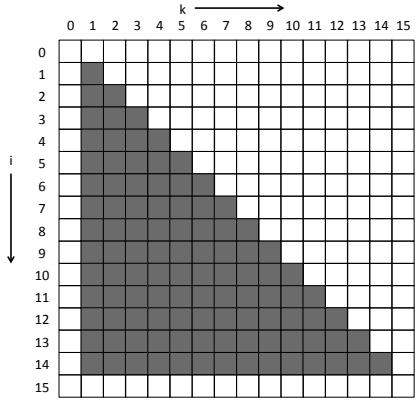
1. Calcula el valor mínimo con el que debería de definirse la constante `PAD` en el programa anterior para evitar *"false sharing"* durante la ejecución del bucle paralelo.

2. Calcula el valor mínimo con el que debería definirse la constante `CHUNK` para mejorar la localidad espacial, dentro de cada thread, en los accesos de lectura a `vector`.

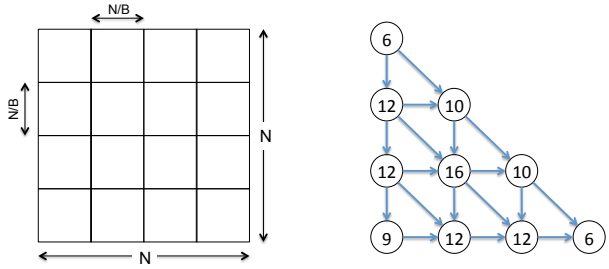
Solution

Pregunta 1 (6 puntos)

1. N=16



2. N=16 y B=4

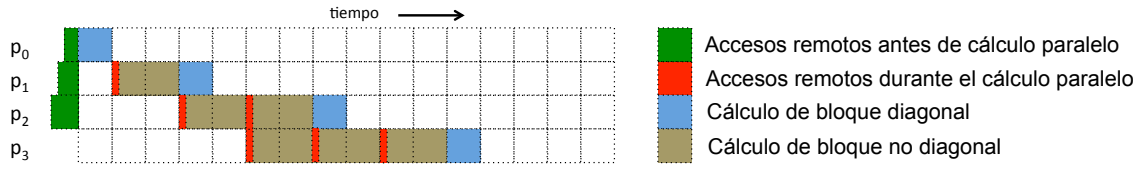


donde cada nodo indica el número de iteraciones que ejecuta la tarea asociada.

3. (a) $T_1 = 105$

4. (d) $T_\infty = 76$, definido por 6/12/12/16/12/12/6)

5. Cronograma de ejecución que os permite calcular T_4.



6. En el camino crítico se incluye el cálculo de 1 bloque "diagonal" y $(B-1)$ bloques "no diagonal" por parte del procesador p_3, además de 2 $(P-2$ en general) bloques "no diagonal" por parte de p_1 y p_2 y un bloque "diagonal" por parte de p_0. Dado que $N >> P$ y N es muy grande, podemos aproximar el tiempo de cálculo de un bloque "no diagonal" por $t_{nd} = (N \div P) \times (N \div B)$ y el de un bloque diagonal por $t_d = (t_{nd} \div 2)$. Por lo tanto la contribución a T_4 es $((((B-1)+(P-2)) \times t_{nd}) + (2 \times t_d)) \times t_c$. Dado que $B = P$, entonces aproximamos la contribución a T_4 por $6 \times (N^2 \div 16) \times t_c$.

7. Todos los procesadores p_x, excepto $x = 3$, acceden a datos remotos en el procesador $x+1$. La cantidad de datos que se acceden depende de x, siendo ésta mayor para $x = 2$ (necesita acceder a $(B-1) \times (N \div B)$ elementos, aproximadamente). Supondremos que todos estos accesos ocurren en paralelo y que ademas no se inicia el cálculo hasta que todos ellos finalizan. De esta manera, la contribución a T_4 es $t_s + ((B-1) \times (N \div B)) \times t_w$, que para $B = P$ quedaría $t_s + (3/4 \times N) \times t_w$. Si no consideramos la sincronización de estos accesos remotos, entonces podrian solaparse con el cálculo, reduciendo su contribución a T_4, en el mejor de los casos a $t_s + (N \div B) \times t_w$.

8. En el camino crítico se realizan $(B-1)$ accesos remotos por parte del procesador p_3 a una fila de N/B elementos, aproximadamente, del procesador p_2. Y además 2 accesos remotos $(P-2$ en general) por parte de p_2 y p_1 a datos en p_1 y p_0, respectivamente, también del mismo tamaño, aproximadamente. Por lo tanto, la contribución de estos accesos a T_4 es $((B-1)+(P-2)) \times (t_s + (N \div B) \times t_w)$. Para el caso de $B = P$, la contribución es $5 \times (t_s + N/4 \times t_w)$.

Pregunta 2 (3 puntos)

1. (d)
2. (b)
3. (c)
4. (d)
5. (c)
6. (b)

Pregunta 3 (1 punto)

1. Cada elemento `struct` del vector `contar` debería de ocupar 1 línea completa de cache si queremos evitar que se produzca compartición falsa (al realizar la actualización de elementos distintos por parte de procesadores distintos). Dado que una línea son 32 bytes, debe de cumplirse que $(2 + PAD) * 4 = 32$, o sea, $PAD = 6$.

2. $CHUNK = 8$ es el valor mínimo con el que se garantiza que cada procesador accede a todos los elementos de `vector` que se almacenan en una linea de cache (si cada `int` ocupa 4 bytes, $8 * 4 = 32$). De esta manera se aprovecha al máximo la capacidad de la memoria cache en cada procesador gracias a la locaclidad espacial del acceso que se realiza.

Primer Control de PAR – Curso 2013/14-Q2

31 de Marzo de 2014

1. (2 puntos) A continuación se muestra el código de un barrier, implementado para una máquina `UMA` con protocolo de coherencia `MSI`, utilizando la instrucción atómica de sincronización `t&s`.

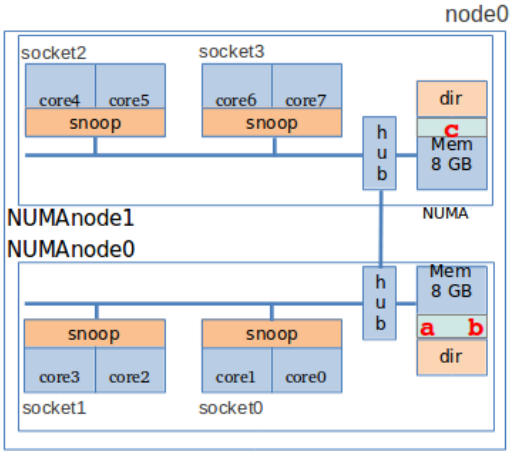
```
lock: t&s r2, barr.lock        // acquire lock
      bnez r2, lock
      if (barr.counter == 0)
         barr.flag = 0            // reset flag if first
      mycount =  barr.counter++;
      if (mycount == P) {         // last to arrive?
         barr.counter =0          // reset for next barrier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

Se pide

   (a) Identifica un problema de concurrencia existente en este código y propon la forma de solucionarlo.

   (b) A partir de la solución propuesta al problema de concurrencia, realiza una versión que reduzca el `overhead` de sincronización para adquirir el `lock`. Justifica qué aspectos mejora tu solución.

2. (3 puntos) Dada la arquitectura de un sistema NUMA mostrada en la figura siguiente:



coreX: Núcleo.
socketY: Empaquetado con 2 núcleos, 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.
nodeZ: Nodo con 2 NUMAnodes.

Y suponiendo que en un momento dado se dispone de las copias coherentes de la variables `a,b y c` mostradas también en la figura, se pide que determinéis qué acciones/efectos, relacionados con el acceso a las variables `a,b y c`, se producirán en los elementos del nodo en caso de que:

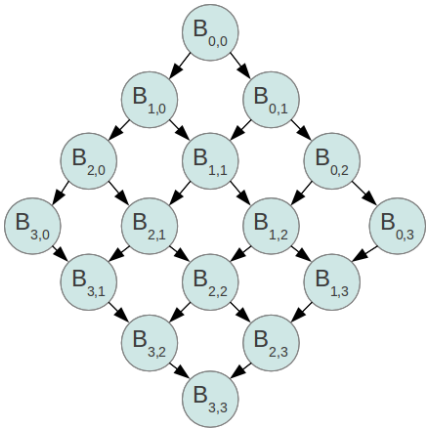
   (a) el core3 solicite acceder a la variable `a` para lectura.

   (b) Si a continuación el core4 solicita acceder a la variable `b` para lectura.

   (c) Si a continuación el core4 modifica (escribe) la variable `b`.

   (d) Si finalmente el core5 modifica (escribe) la variable `c`.

3. (2 puntos) Dado el siguiente grafo de dependencias, donde cada círculo se corresponde con una tarea que computa un bloque/submatriz $B_{i,j}$ con granularidad BI filas $\times BJ$ columnas,



contesta a las siguientes preguntas sabiendo que la matriz tiene un tamaño de $N \times N$, que el coste de operar con un bloque $B_{i,j}$ de la matriz es $1 \times BI \times BJ$ y que disponemos de P procesadores:

(a) Aunque el grafo de dependencias es para el caso particular de $N/B = 4$, $B = BJ = BI$, calcula T_1, T_∞ y P_{min} en función de cualquier N, BI, BJ y P, sabiendo que BI y BJ dividen N.

(b) Sabemos que el código que corresponde a este grafo de dependencias es parte de una aplicación más grande. ¿Cuál debería ser la proporción de tiempo invertida en este código respecto a toda la aplicación para tener un $S_\infty = 5$, entendiendo que esta parte se puede paralelizar hasta el infinito?

4. (3 puntos) Suponed el grafo del ejercicio anterior y el modelo de compartición de datos visto en clase donde los accesos a la memoria local tienen coste 0 y los accesos remotos $t_s + m \times t_w$. La sincronización entre procesadores también tiene coste 0. Las dependencias entre dos tareas se traduce en un acceso remoto a datos si las tareas no estan en el mismo nodo. La cantidad de datos accedidos son BJ elementos entre tareas del tipo $B_{i,Y}$ y $B_{i+1,Y}$, y BI elementos entre tareas $B_{X,j}$ y $B_{X,j+1}$. **Responde a la siguientes preguntas (independientes entre ellas):**

(a) Suponed que trabajamos con $P = 4 = N/B$ procesadores, que $B = BI = BJ$ (igual que en la figura), que forman parte de una máquina `UMA`, y que tenemos una aplicación paralela que garantiza la sincronización adecuada entre tareas. Dibuja el diagrama de tiempo de ejecución de tareas indicando claramente lo que corresponde a comunicación (acceso remoto) y a cómputo (indica el bloque $B_{i,j}$ que se calcula en el diagrama), y calcula T_P sabiendo que en esta aplicación la tarea $B_{i,j}$ se asigna al procesador $(i + j)\%P$. Notad que un procesador puede tener asignadas más de una tarea y que puede empezar a ejecutar una tarea tan pronto la/s tarea/s de las que depende se hayan finalizado.

(b) Suponed que trabajamos con P procesadores ($P = N/B$) que forman parte de una máquina `UMA`, que $B = BI = BJ$ y que tenemos una aplicación paralela que garantiza la sincronización adecuada entre tareas. Calcula T_P sabiendo que en esta aplicación a cada procesador se le asigna una tarea del grafo de dependencias cada vez que este procesador está libre y hay tareas listas (las tareas de las que dependen ya han finalizado) para ejecutar. Suponed que el coste de asignación de tarea a procesador es 0. Notad que no pedimos el diagrama de tiempo.

(c) Suponed que trabajamos con una máquina con P nodos con memoria distribuida, donde cada nodo tiene un procesador y su memoria local. También disponemos de una aplicación que garantiza la sincronización adecuada entre tareas. Aquí supondremos que $BI = N/P$ y $N = 4 \times BJ$, que cada tarea $B_{i,x}$ es realizada por el procesador i y que los datos originales asignados a una tarea se mapean, al iniciarse el programa, en la memoria local del procesador que la ejecuta. Dibuja el diagrama de tiempo de ejecución de tareas indicando claramente lo que corresponde a comunicación y a cómputo, y calcula T_P.

Solution

1. (2 puntos)

   (a) Existe un problema de deadlock debido a que el release del lock se hace al final. Esto puede provocar, para $P > 1$, que un thread se quede esperando en el bucle del $flag == 0$ sin liberar el lock, y el resto de threads en el bucle para intentar obtener el lock. La solución propuesta consiste en avanzar ese release hasta antes del segundo if.

```
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++;
      barr.lock = 0               // release lock
      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
```

   (b) Realizamos un *test and test and set*

```
      ...
lock: ld r2, barr.lock         // test and test and set
      bnez r2, lock
      t&s r2, barrier.lock        // acquire lock
      bnez r2, lock
      if (barr.counter == 0)
         barr.flag = 0            // reset flag if first
      mycount =  barr.counter++;
      barr.lock = 0               // release lock
      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
      ...
```

   De esta forma reducimos el número de accesos a la memoria principal de forma atómica, que puede ser costoso. En particular, al realizar un `ld` sobre `barr.lock` conseguimos traernos la línea de memoria a la cache, y mientras que no hay actualizaciones del lock, ésta se mantendrá en cache, reduciendo los accesos a memoria y al bus. Sólo se realizará un `t&s` si hay alguna probabilidad de adquirir el `lock`

2. (3 puntos)

   (a) core3 genera PrRd, lo que provoca una transacción BusRd en el bus por parte del snoopy. Este BusRd no afecta a ninguna cache ya que no existen copias en la cache de la línea de memoria donde está la variable `a`. core3 está en el hub `home` de la línea de memoria donde está `a`, por lo que no haría falta generar una petición de memoria (RdReq) a otro hub. Se actualizará la entrada correspondiente del directorio activando el bit para indicar que este hub tendrá una copia de la línea de memoria en la cache del socket1, además de indicar que esta línea de memoria es shared (S). No hace falta enviar la línea de cache donde está la variable (RdResp) a otro hub, pero se pone en el bus del hub. El snoop del core3 guardará una copia de la línea de memoria en su cache e indicará que está en estado S.

(b) core4 genera PrRd, lo que provoca una transacción BusRd en el bus. Tampoco existen copias de la línea de memoria donde está la variable `b` en las caches de dicho bus. El hub redirecciona la petición de memoria (RdReq) al otro hub donde se encuentra almacenada la línea de memoria que contiene la variable. Se actualizará la entrada correspondiente del directorio. Se activará el bit de la línea de memoria indicando que el nuevo hub contendrá una copia y se envia la línea de memoria (RdResp). De hecho, como las variables `a y b` comparten línea de memoria, los bits del directorio para esta línea de memoria reflejaran que hay dos hubs que la comparten (el estado se mantiene en shared). La cache del core4 guardará una copia de la línea de memoria en su cache, e indicará que el estado de la línea de cache que contiene la copia tiene tiene estado shared (S).

(c) core4 genera PrWr, lo que provoca una transacción BusRdX en el bus. El hub redirecciona la petición de memoria (RdXReq) al otro hub donde se encuentra almacenada la línea de memoria donde está la variable (hub `home`). Éste deberá indicar la lista de hubs que comparten esta línea de memoria (hub del NumaNode0 y hub del NumaNode1). NumaNode1 recibirá la lista y enviará una transacción de invalidación a los hubs que tienen copia, en este caso sólo lo enviará al hub del NumaNode0[1]. El hub del NumaNode0 enviará un BusRdX en el bus y el snoopy del socket1 (core3 con copia en cache de línea de memoria donde está la `a`) escuchará dicha transacción e invalidará su copia, pasando a estado inválido (I). Finalmente, indicará al NumaNode1 que ya se ha invalidado, y actualizará el directorio para indicar que sólo hay un hub con una copia, y en estado modified. El hub del NumaNode1 indicará al snoop del socket2 que ya puede actualizar el bit de coherencia de esa copia de la línea de memoria. Éste cambiará el estado de la copia para indicar que está modificada (modified - M).

Aquí hemos visto un caso de False Sharing ya que el core3 ha tenido que invalidar su copia en cache de la línea de memoria donde está la variable de `a` debido a que el core4 ha actualizado su copia de la línea de memoria, donde está la variable `b`.

En caso de que no se hubiera considerado que a y b estan en la misma línea de memoria entonces no haría falta invalidar la línea de cache de a en el core3 y tampoco se le enviaría nada al NumaNode1.

(d) core5 genera PrWr, lo que provoca una transacción BusRdX en el bus. A su vez, el hub de dicho bus consultará la lista de hubs con copia de la línea de memoria y verá que está vacía. Modificará la información del directorio de esa línea para indicar que el NumaNode1 tendrá una copia modificada (M). Además, avisará al socket2 que ya se han invalidado todas las copias (ninguna) en cache de la línea de memoria. La línea de cache del socket2, con copia de la línea de memoria donde está la variable, pasa a estado modified (M).

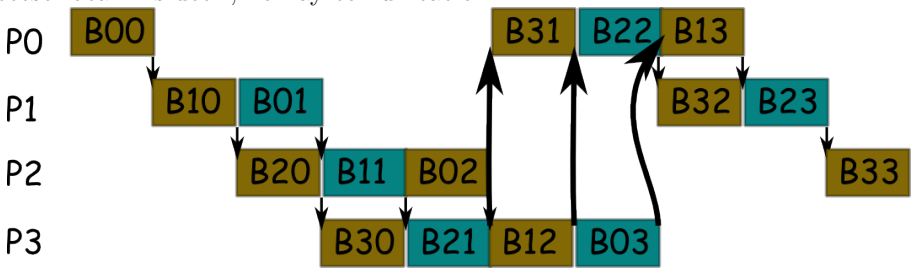
3. (2 puntos)

(a)
- $T_1 = 1 \times N \times N$
- $T_\infty = (N/BI + N/BJ - 1) \times BI \times BJ \times 1$
- $P_{min} = \min(N/BI, N/BJ)$

(b) $S_\infty = 5 \Rightarrow S_\infty = \frac{1}{1-\phi} \Rightarrow \phi = 4/5$

4. (3 puntos)

(a) Al estar en una máquina UMA todos los accesos a memoria tienen el mismo coste que un acceso local. Es decir, no hay comunicación.



[1] seguramente todo esto no será necesario ya que el hub `home` podría optimizar este proceso de invalidar

El cálculo de T_P para $P = 4$ es:

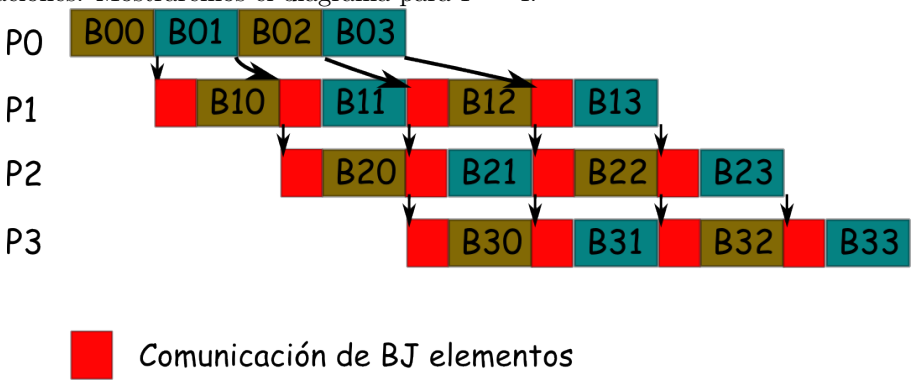
$T_4 = 10B^2$

(b) La solución básicamente consiste en seguir el camino crítico del grafo de dependencias entre tareas ya que tenemos tantos procesadores disponibles como el P_{min}. Por consiguiente, a medida que se van cumpliendo las dependencias se van asignando a procesadores y se van ejecutando.

Al estar en una máquina UMA todos los accesos a memoria tienen el mismo coste que un acceso local. Es decir, no hay comunicación.

$T_P = (N/B + N/B - 1) \times B^2 \Rightarrow (\frac{2N}{B} - 1) \times B^2$

(c) En este caso se especifíca que estamos en una máquina con memoria distribuida y con una distribución de los datos inicial. Por consiguiente, en esta ejecución sí se realizaran comunicaciones. Mostraremos el diagrama para $P = 4$:



El cálculo de T_P es:

$T_P = \frac{N}{BJ}(BJ \times BI + t_s + BJ \times t_w) + (P-1) \times BJ \times BI + (P-2)(t_s + BJ \times t_w)$

simplificando:

$T_P = (\frac{N}{BJ} + P - 1)(BJ \times BI) + (\frac{N}{BJ} + P - 2)(t_s + BJ \times t_w)$

Parallelism - 1^{st} Mid-term Exam - 2013/14 -Q2

March 31st, 2014

1. (2 Points) Given the following code and a matrix $u[N+2][N+2]$:

```
for (i=1; i<N+1; i+=NB) {
  for (j=1; j<N+1; j+=NB) {
     Tareador_start_task("task_code");
     for (ii=i; ii<i+NB; ii++) {
       for (jj=j; jj<j+NB; 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();
  }
}
```

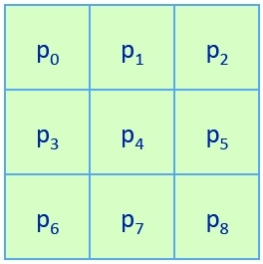
   Suppose that the body of the most internal loop has a cost t_c, and NB is a constant that divides N. Answer the following questions:

   (a) (0.5 Points) Draw the task graph that would be displayed by the tool *Tareador* if there were 4 blocks in each dimension.

   (b) (1.5 Points) Determine the general expression for T_1, T_∞ and P_{min}, as a function of N and NB.

2. (5 Points) Continuing with the code in the previous exercise and considering in all cases that the parts of the external halo are already stored in the processor that needs them:

   (a) (2.5 Points) If each of the generated tasks is executed in a processor, assuming a matrix distribution between p^2 processors as shown in the following figure for $p^2 = 9$:

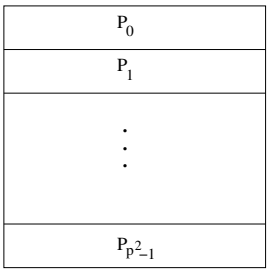


   Answer these questions assuming NB is equal to N/p:

      i. (1 Point) Draw the execution time schedule for $p^2 = 9$ processors, indicating clearly what is communication time (also of what) and what is calculation time.

      ii. (1.5 Points) Calculate the general expression for T_p as a function of p, N and NB, 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 time and the sending time of an element, respectively, and being m the size of the message measured as the number of elements. Assume also that the execution time of an iteration in the body of the most internal loop is t_c.

   (b) (2.5 Points) If we assume a rowwise matrix distribution among p^2 processors as shown in the following figure:

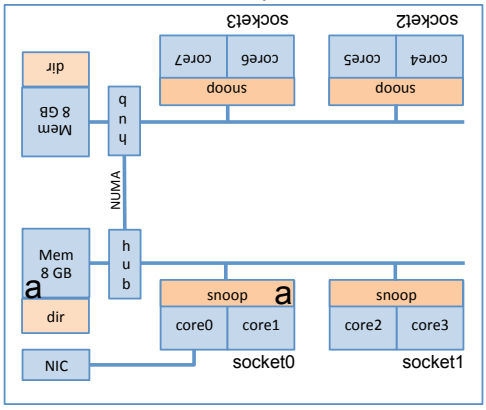


and we assume that each processor does blocking by columns in its part of the matrix, with a block size of B:

i. (1 Point) Supposing $B = N/p^2$, draw the execution time schedule for $p^2 = 9$ processors, indicating clearly what is communication time (also of what) and what is calculation time.

ii. (1.5 Points) Calculate the general expression for T_p as a function of p, N, B, and NB, using the same data sharing model as in the previous question.

3. (3 Points) 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.

Assuming that the synchronization between cores is done using atomic instructions such as `test_and_set` inside each *NUMAnode* and using linked instructions such as `load_linked store_conditional` between *NUMAnodes*.

(a) (2 Points) We ask you to indicate if these sentences are **true or false**:

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

ii. The atomic instructions return in a register the value that existed in the memory position where they are writing a new value.

iii. The `store_conditional` instruction verifies that the value in a concrete memory position has not been changed with respect to the moment when the last `load_linked` has been done.

iv. 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.

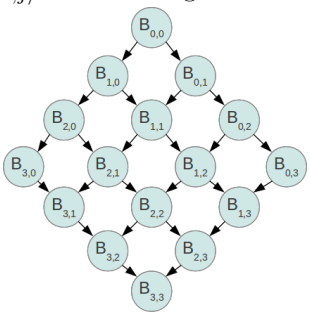
(b) (1 Point) If there exists an unmodified copy of variable a in the cache of socket0, indicate the coherence actions that will take place when a write on a is performed from core4 in socket2.

Solution

1. (2 Points)

   (a) (0.5 Points)

   Each task is labeled as $B_{i/NB,j/NB}$ indicating the block it updates:



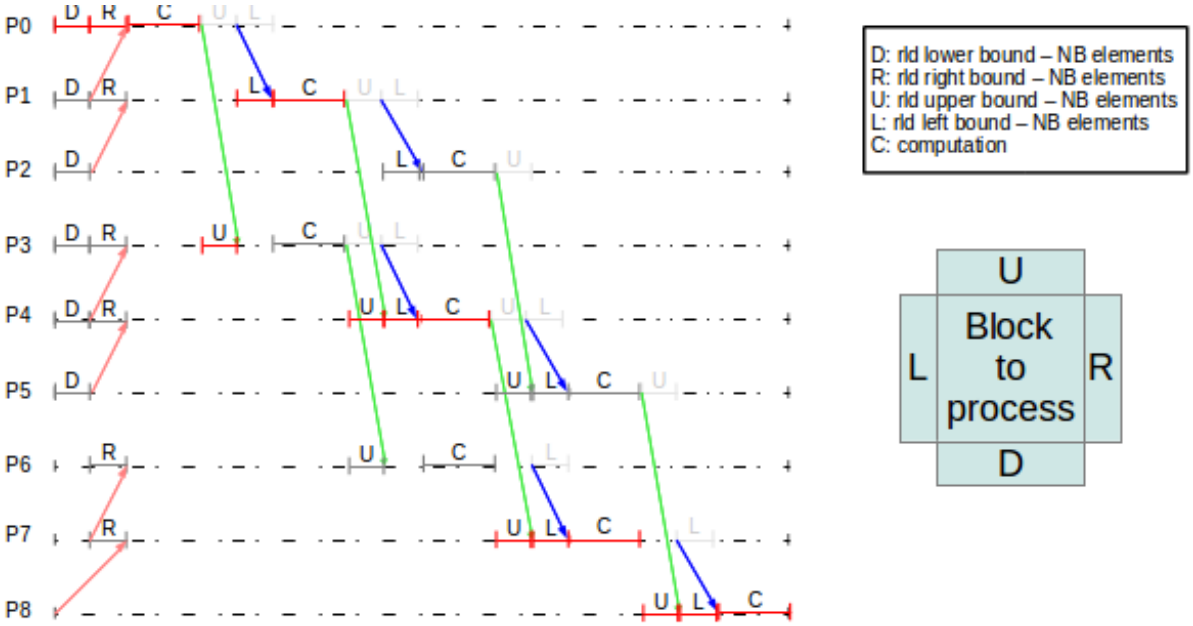
   (b) (1.5 Points)

   - $T_1 = N \times N \times t_c = N^2 \times t_c$
   - $T_\infty = (\frac{N}{NB} + \frac{N}{NB} - 1) \times (NB \times NB \times t_c) = (2\frac{N}{NB} - 1) \times NB^2 \times t_c$
   - $P_{min} = \frac{N}{NB}$

2. (5 Points)

   (a) (2.5 Points)

   i. (1 Point)
   The external halo includes the first and last row, and the first and last column. We do not consider any communication for the halo as indicated in the statement of the exercise. Also, we do not consider possible overlapping between communications and computations. Thus, a simplified chronogram showing how the computations and communications would proceed is as follows:



   The red segments indicate the critical path. We show in light grey the time while the network interface of a processor is serving a remote-load from another processor. As it can be observed, P1 can only have its remote-load of its *left boundary (L)* served from

P0 once P3 has its remote-load of the *upper boundary (U)* served from P0. The reason for this is because, in our model, a processor can only serve one remote-load at a time. Notice, that this is one possible chronogram but there exist some variants. For instance, processor P_3 could start earlier, as soon as it has remote-loaded U from P_0. However, P_4 would still have to wait to do the remote-load corresponding to L from P_3 until P_4 had finished the remote-load corresponding to U from P_1. The reason for this is because, in our model, a processor can only a perform one remote-load at a time.

ii. (1.5 Points)

- Communication time:
  - $T_{comm\_initial} = 2 \times (t_s + NB \times t_w)$
  - $T_{comm\_during\_computation} = (\frac{N}{NB} - 1 + \frac{N}{NB} - 1) \times 2 \times (t_s + NB \times t_w)$
    $= 2(p-1) \times 2 \times (t_s + NB \times t_w)$
- Computation time:
  - $T_{comp} = (\frac{N}{NB} + \frac{N}{NB} - 1) \times (NB \times NB \times t_c) = (2p-1) \times (NB^2 \times t_c)$
- Total time:
  - $T_p = T_{comm\_initial} + T_{comm\_during\_computation} + T_{comp}$

(b) (2.5 Points)

i. (1 Point)

The red segments indicate the critical path.

ii. (1.5 Points)

- Communication time:
  - $T_{comm\_initial} = t_s + N \times t_w$
  - $T_{comm\_during\_computation} = (\frac{N}{B} + p^2 - 2) \times (t_s + B \times t_w) = (2p^2 - 2) \times (t_s + B \times t_w)$
- Computation time:
  - $T_{comp} = (\frac{N}{B} + p^2 - 1) \times B \times B \times t_c = (2p^2 - 1) \times B^2 \times t_c$
- Total time:
  - $T_p = T_{comm\_initial} + T_{comm\_during\_computation} + T_{comp}$

3. (3 Points)

(a) (2 Points)

i. **False**
ii. **True**
iii. **False**
iv. **False**

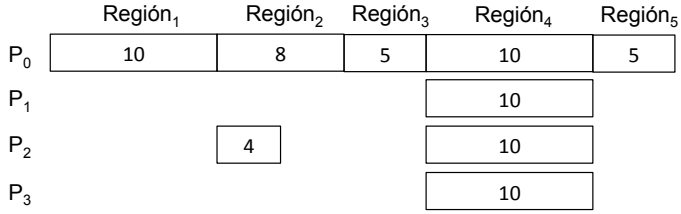
(b) (1 Point)

Let us call NUMAnode0 the one containing sockets 0 and 1; and NUMAnode1 the one containing sockets 2 and 3. When a write on a is performed from core4 in socket2, core4 generates a PrWr. This, in turn, generates a BusRdX transaction in the bus. There are no copies of the memory line which contains a (*line_a*) in NUMAnode1. Thus, this BusRdX does not affect any other cache in NUMAnode1. However, the hub in NUMAnode1 (`hub1`) redirects the memory request (RdXReq) for *line_a* to `hub0`, the hub of the *home* node of *line_a*, i.e. NUMAnode0. `hub0` will check the directory entry corresponding to *line_a*. Since *line_a* is not marked as modified in the directory there is no need to update it with a new value. `hub0` will return to `hub1` the contents of *line_a* and a list of hubs which have a copy of this memory line: `hub0` in this case. Then, the bit corresponding to NUMAnode1 will be set in the directory of `hub0` for *line_a*. `hub1` will send an invalidation request to `hub0` (this invalidation process could be optimized by having `hub0` invalidate any copies in NUMAnode0 directly). `hub0` will send a BusRdX through the local bus in NUMAnode0. The Snoopy mechanism in socket0 will listen to such transaction and will invalidate the copy of *line_a*, changing its state to I. Next, `hub0` will inform `hub1` that the invalidation has been performed. `hub0` will update its directory entry for *line_a* to reflect that only `hub1` has a copy, and it is in Modified (M) state. Finally, the copy of the memory line is kept in the cache of socket2, and `hub1` will indicate that it can already be modified. The Snoopy in socket2 will change the state of the memory line from Invalid (I) into Modified (M) and core4 will proceed to modify variable a.

Primer Control de PAR – Curso 2013/14-Q2

31 de Marzo de 2014

Pregunta 1 (4 puntos) Dado el siguiente diagrama de ejecución temporal (cronograma):

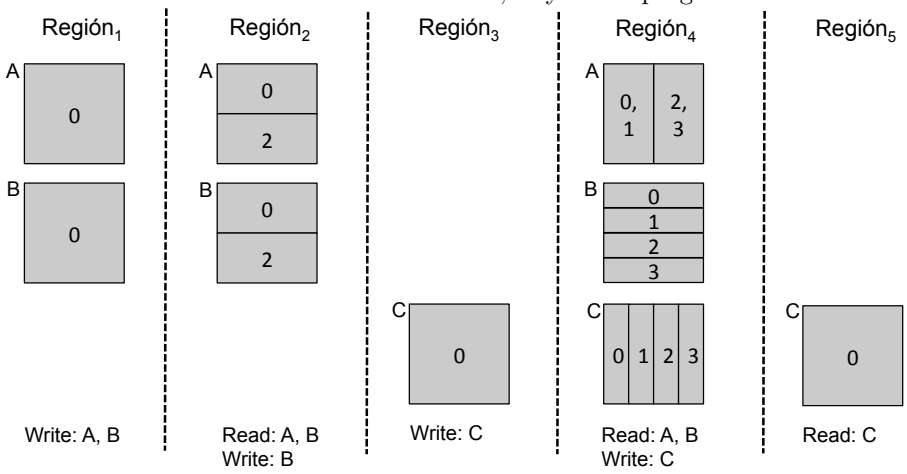


en el que se muestra la actividad (ráfagas) de cada uno de los 4 procesadores de un sistema multiprocesador. El número en cada ráfaga representa el tiempo de ejecución de la misma. Suponiendo que las regiones 1, 3 y 5 son secuenciales, que la región 2 no permite usar más de dos procesadores (y sólo permite la distribución de carga mostrada) y que la región 4 escala de forma ideal con el número de procesadores hasta el infinito, **se pide:**

1. Calcular los valores de T_1, T_4, T_∞, S_4 y S_∞.

2. Calcular el valor de la fracción paralela ϕ.

Con el objetivo de modelar los *overheads* causados por la compartición de datos, supondremos una arquitectura de memoria distribuida. La figura siguiente nos muestra como cada una de las regiones mencionadas anteriormente accede a las tres matrices A, B y C del programa:



En esta figura, para cada región se indica que matrices se acceden y el tipo de acceso (lectura o escritura) que se realiza sobre las mismas; los números indican qué parte de las matrices es accedida por cada procesador. Durante la ejecución de las regiones paralelas 2 y 4 no hay ningún tipo de dependencia entre procesadores. Al finalizar estas dos regiones se realiza un *barrier* (barrera) antes de continuar con la parte secuencial que les sigue. **Se pide:**

3. Dibujar el diagrama de ejecución temporal (cronograma) con 4 procesadores suponiendo el modelo de acceso a datos remotos explicado en clase de teoría, indicando que representa cada una de las ráfagas que se añadan en el mismo. Considerad que antes de ejecutar una región, cada procesador deberá tener los datos que necesita en su memoria, accediendo a los mismos con el menor número posible de accesos remotos. Los tiempos de cálculo de cada ráfaga son los indicados en el cronograma inicial.

4. Obtener la expresión que determina el T_P sabiendo que las matrices son de tamaño $N \times N$, que el número de procesadores P divide de forma entera a N y que el "overhead" de los accesos remotos sigue el modelo $t_s + m \times t_w$, siendo m el número de elementos accedidos.

Pregunta 2 (4 puntos) Dado un sistema multiprocesador SMP con 2 procesadores, cada uno de ellos con una memoria *cache* y *snoopy* que implementa un protocolo de coherencia *write–invalidate MSI*, en el que se ejecuta el código mostrado a continuación:

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

suponiendo que la variable `i` se almacena en un registro y que inicialmente las *caches* estan vacias.
Se pide:

1. Suponiendo que no se produce *false sharing* en los accesos a las variables `count` y `flag` y el siguiente orden temporal de ejecución de las instrucciones que acceden a memoria:

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

   completar la tabla anterior con las acciones se realizan (*bus transactions* y cambios de estado en las *caches*) para mantener la coherencia de memoria (podéis suponer que la ejecución de la instrucción `test&set` (abreviada `t&s`) y el incremento `++` provocan una única transacción `BusRdX` en caso de hacer *miss* en la cache).

2. Repetir el apartado anterior suponiendo ahora que se produce *false sharing* cuando se accede a las variables `count` y `flag`, utilizando para ello la tabla siguiente:

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

3. Con el objetivo de reducir el tráfico de coherencia que se provoca, substituir la instrucción *test&set* por un una secuencia de instrucciones que realice *test-test&set* (escribir dicha secuencia de instrucciones).

4. Con la secuencia de instrucciones del apartado anterior, completad de nuevo la tabla asumiendo que el procesador `P1` sigue realizando accesos a memoria en los mismos ciclos de antes y que se provoca *false sharing* cuando se accede a las variables `count` y `flag`.

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

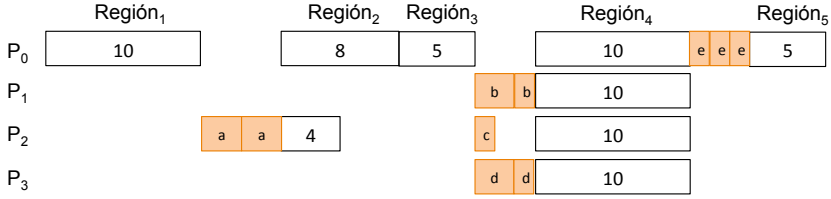
Pregunta 3 (2 puntos) Indicar cuál o cuáles de las siguientes afirmaciones son ciertas:

1. En una arquitectura NUMA se requieren instrucciones de acceso a memoria distintas a las convencionales (load/store) para acceder a variables almacenadas en otros *NUMAnodes*, para que pueda activarse correctamente el protocolo de coherencia.
2. El directorio que hay en un *NUMAnode* indica donde se encuentran las variables que necesita el procesador en dicho *NUMAnode* para ejecutar el programa.
3. En un sistema NUMA el *home node* para cada dirección de memoria lo determina el sistema operativo en base a alguna política de acceso a los datos.
4. En un sistema NUMA el *home node* proporciona la información que necesita el *local node* para realizar las intervenciones necesarias en los *remote nodes*, si los hay.

Solution

Pregunta 1 (4 puntos)

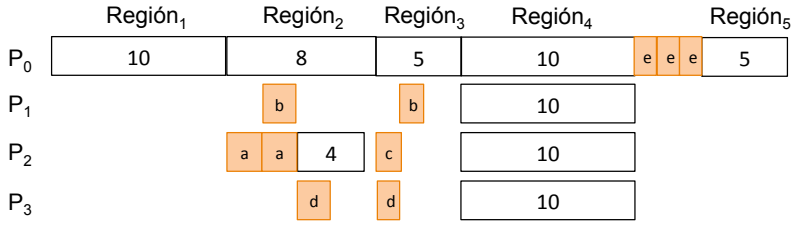
1. $T_1 = 10 + (8 + 4) + 5 + (4 \times 10) + 5 = 72; T_4 = 10 + 8 + 5 + 10 + 5 = 38;$
   $T_\infty = 10 + 8 + 5 + 0 + 5 = 28; S_4 = 72/38 = 1.89; S_\infty = 72/28 = 2.57$ (este último calculado a partir de un diagrama de tiempo donde la region 4 desaparece y la 2 queda igual pués no se puede escalar más allá de 2 procesadores).
2. $\phi = ((8 + 4) + (4 \times 10)) \div (10 + (8 + 4) + 5 + (4 \times 10) + 5) = 52/72 = 0.72$
3. Un cronograma en el que se muestran los accesos remotos de lectura a realizar antes de iniciar las regiones de cálculo, ignorando conflictos en el acceso a memoria durante los mismos seria el siguiente:



Se necesitan los siguientes accesos remotos:

- a: acceso a las mitades inferior de A y B.
- b: acceso a la mitad izquierda de A, cuarto central-superior de B. El acceso remoto para C no es necesario pues no se lee y se reescribe.
- c: acceso al cuarto superior derecha de A. El acceso remoto para C no es necesario.
- d: acceso a la mitad derecha de A, cuarto inferior de B. El acceso remoto para C no es necesario.
- e: acceso a los cuartos de C que están en los otros 3 procesadores.

Sin embargo, un cronograma más realista en el que suponemos que se solapan los accesos remotos sería el siguiente (cada acceso se inicia tan pronto como se puede, considerando conflictos en los accesos remotos):



4. Suponiendo el caso más sencillo en el que se pueden solapar los accesos remotos, la expresión de T_p consta de dos partes:

   - $T_p(calculo) = 10 + 8 + 5 + (40 \div p) + 5$
   - Segun el cronograma anterior, $T_p(accesos\_remotos) = (p-1) \times (t_s + (n^2/p) \times t_w)$ (acceso a C por parte de P_0 necesario para la región 5).

   Para ello debe de cumplirse que las ráfagas de cálculo solapan totalmente los *overheads*. Por ejemplo que $8 > 4 + 2 \times (t_s + (n^2/2) \times t_w)$ para los accesos a A y B (calculados en la región 1) necesarios en P_2 para la región 2 o que $5 > (t_s + (n^2/p) \times t_w)$ para el acceso a B (calculado en la región 2) necesario en la región 4. Los accesos a A que requieren P_1 y P_3 tienen más oportunidades de solape durante las regiones 2 y 3, dado que se calcularon en la región 0 por parte de P_0.

Pregunta 2 (4 puntos)

1. **Nota:** En la tabla se muestra el nombre de la variable y entre paréntesis, el estado en que se encuentra la línea de cache que contiene esa variable en la cache del procesador.

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

2. **Nota:** En la tabla se muestra el nombre de las variables y entre paréntesis, el estado en que se encuentra la línea de cache que las contiene (debido al false sharing comparten la misma línea).

Tiempo	Procesador P0			Procesador P1		
	Instrucción ejecutada	Bus transaction	Estado cache	Instrucción ejecutada	Bus transaction	Estado cache
0	**count=0**	BusRdX	count(M)			
1	**flag=1**		(misma linea)			
2		flush	flag/count(I)	**t&s flag**	BusRdX	flag/count(M)
3	**count++**	BusRdX	flag/count(M)		flush	flag/count(I)
4		flush	flag/count(I)	**t&s flag**	BusRdX	flag/count(M)
5	**count++**	BusRdX	flag/count(M)		flush	flag/count(I)
6		flush	flag/count(I)	**t&s flag**	BusRdX	flag/count(M)
7	**count++**	BusRdX	flag/count(M)		flush	flag/count(I)
8		flush	flag/count(I)	**t&s flag**	BusRdX	flag/count(M)
9	**count++**	BusRdX	flag/count(M)		flush	flag/count(I)
10		flush	flag/count(I)	**t&s flag**	BusRdX	flag/count(M)
11	**flag=0**	BusRdX	flag/count(M)		flush	flag/count(I)
12		flush	flag/count(I)	**t&s flag**	BusRdX	flag/count(M)

3. La secuencia de instrucciones que realiza el *test-test&set* es:

```
lock: ld r2, flag
      bnez r2, lock
      test&set r2, flag
      bnez r2, lock
```

4. Asumiendo que el procesador `P1` sigue realizando accesos a memoria en los mismos ciclos de antes pero usando el *test-test&set* y que se provoca *false sharing* cuando se accede a las variables `count` y `flag`:

Tiempo	Procesador P0			Procesador P1		
	Instrucción ejecutada	Bus transaction	Estado cache	Instrucción ejecutada	Bus transaction	Estado cache
0	**count=0**	BusRdX	count(M)			
1	**flag=1**		(misma linea)			
2		flush	flag/count(S)	**ld r2, flag**	BusRd	flag/count(S)
3	**count++**	BusRdX	flag/count(M)			flag/count(I)
4		flush	flag/count(S)	**ld r2, flag**	BusRd	flag/count(S)
5	**count++**	BusRdX	flag/count(M)			flag/count(I)
6		flush	flag/count(S)	**ld r2, flag**	BusRd	flag/count(S)
7	**count++**	BusRdX	flag/count(M)			flag/count(I)
8		flush	flag/count(S)	**ld r2, flag**	BusRd	flag/count(S)
9	**count++**	BusRdX	flag/count(M)			flag/count(I)
10		flush	flag/count(S)	**ld r2, flag**	BusRd	flag/count(S)
11	**flag=0**	BusRdX	flag/count(M)			flag/count(I)
12		flush	flag/count(S)	**ld r2, flag**	BusRd	flag/count(S)

Pregunta 3 (2 puntos)

1. **Falsa**
2. **Falsa**
3. **Cierta**
4. **Cierta**

Part II

2^{nd} In-term Exams

Segundo Control de PAR, Curso 2012/13-Q2
30 de Mayo de 2013

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

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

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

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

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

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

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

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

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

1. Task decomposition (4.5 puntos: 0.75/0.75/1/2):

1.1 Escribe una versión paralela para la función `FindBounds`, utilizando **un único** `#pragma` de OpenMP, de manera que se minimicen los overheads asociados con su ejecución paralela y la sincronización.

1.2 Escribe una versión alternativa a la siguiente paralelización de la función `FindFrequency`:

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

en la que se mejore el paralelismo que se puede obtener, **sólo** utilizando `pragma`s de OpenMP.

1.3 Suponiendo que se desea reducir al máximo la secuencialización que provoca la sincronización en el apartado anterior, escribir una solución basada en el uso de `lock`s de OpenMP.

1.4 Escribe una versión paralela OpenMP para la función `FindFrequency` basada en una estrategia de descomposición en tareas *divide_and_conquer*, utilizando la sincronización y la estrategia de control de *overhead* en la creación de tareas que os parezcan más adecuadas.

2. Data decomposition (3 puntos: 0.5/1.5/1):

2.1 Escribir una versión paralela OpenMP para la función `FindBounds` que se base en una **descomposición de datos geométrica `CYCLIC` del vector `input`** (es decir, elementos consecutivos se asocian a procesadores consecutivos, de forma cíclica). **Nota:** no puede utilizarse el `pragma omp for`.

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

2.3 Suponer que las funciones `ReadNumbers` y `DrawHistogram` se ejecutan en un único procesador (por ejemplo el `0`), y que el código resultante de aplicar la descomposición de datos de los apartados 1.4 y 1.5 se quiere ejecutar en una arquitectura de memoria distribuida, indicar que comunicaciones punto a punto y/o colectivas de MPI serian necesarias para realizar el movimiento entre procesadores del vector que almacena el histograma (`frequency`) y de la variable escalar `max` que almacena el valor máximo encontrado, concretando para cada una de dichas comunicaciones donde se realizaría, los datos y los procesadores implicados (**No** es necesario escribir el código completo MPI).

3. Cuestiones (2.5 puntos: 0.5/0.5/0.75/0.75). Responder a cada una de las cuestiones siguientes:

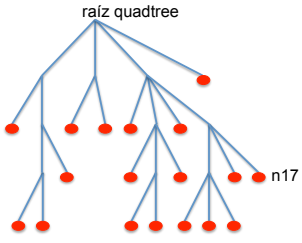
3.1 Describe en el espacio disponible el problema de rendimiento que introduce la siguiente paralelización del bucle y como puede corregirse:

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

3.2 Describe en el espacio disponible la anomalía que puede ocurrir en la ejecución paralela del siguiente bucle y como puede corregirse:

```
#pragma omp parallel for
for (i=0; i<N; j++) {
     j = f(i); // devuelve cualquier valor entre 0 y N-1, distinto de i
     set_omp_lock(&lck(i);
     set_omp_lock(&lck(j);
     int temp = vector[i];
     vector[i] = vector[j];
     vector[j] = tmp;
     unset_omp_lock(&lck(j);
     unset_omp_lock(&lck(i);
     }
```

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



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

3.4 Idem en el caso de una descomposición recursiva *Leaf* en tareas totalmente independientes, en la que cada rama de un nodo interno del árbol corresponde a la invocación recursiva y en la que cada hoja del árbol corresponde a la creación de una tarea que ejecuta la función `doComp`.

Solution

1. Task decomposition (4.5 puntos: 0.75/0.75/1/2)

1.1
```
int tmin=*min, tmax=*max;
#pragma omp parallel for reduction(max: tmax) reduction(min: tmin)
for (int i=0; i<size; i++) {
      if (input[i]>(*max)) (*max)=input[i];
      if (input[i]<(*min)) (*min)=input[i];
      }
*min=tmin; *max=tmax;
```

1.2
```
#pragma omp parallel for private(tmp)
for (int i=0; i<size; i++) {
     tmp = (input[i] - min) * HIST_SIZE / (max - min - 1);
     #pragma omp atomic
     histogram[tmp]++;
     }
```

1.3
```
omp_lock_t histlock[HIST_SIZE];
for (int i=0; i<HIST_SIZE; i++) omp_init_lock(&histlock[i]);
#pragma omp parallel for private(tmp)
for (int i=0; i<size; i++) {
     tmp = (input[i] - min) * HIST_SIZE / (max - min - 1);
     omp_set_lock(&histlock[tmp]);
     histogram[tmp]++;
     omp_unset_lock(&histlock[tmp]);
     }
for (int i=0; i<HIST_SIZE; i++) omp_destroy_lock(&histlock[i]);
```

1.4
```
omp_lock_t histlock[HIST_SIZE];
#define MAX_LEVEL 4

void FindFrequency_rec(int * input, int size , int * histogram, int min, int max, int level) {
int tmp;
if ((size==1) || (level > MAX_LEVEL))
     for (int i=0; i<size; i++) {
          tmp = ((input[i] - min) * HIST_SIZE / (max - min - 1);
          omp_set_lock(&histlock[tmp]);
          histogram[tmp]++;
          omp_unset_lock(&histlock[tmp]);
          }
else {
      int size2 = size/2;
      #pragma omp task
      FindFrequency_rec(input, size2, histogram, min, max, level+1);
      #pragma omp task
      FindFrequency_rec(input+size2, size-size2, histogram, min, max, level+1);
}}
```

```
void FindFrequency(int * input, int size , int * histogram, int min, int max) {
for (int i=0; i<HIST_SIZE; i++) omp_init_lock(&histlock[i]);
#pragma omp parallel
#pragma omp single
FindFrequency_rec(input, size , histogram, min, max, 0);
for (int i=0; i<HIST_SIZE; i++) omp_destroy_lock(&histlock[i]);
}
```

2. Data decomposition (3 puntos: 0.5/1.5/1)

2.1
```
int tmin=*min, tmax=*max;
#pragma omp parallel reduction(max: tmax) reduction(min: tmin)
    {
    int who = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    for (int i=who; i<size; i = i + howmany) {
        if (input[i]>(*max)) (*max)=input[i];
        if (input[i]<(*min)) (*min)=input[i];
        }
    }
*min=tmin; *max=tmax;
```

2.2
```
#pragma omp parallel private(tmp)
    {
    int who = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int psize = (HIST_SIZE/howmany) + (who<(HIST_SIZE%howmany));
    int lower = who*(HIST_SIZE/howmany) +
                (who<(HIST_SIZE%howmany) ? who : (HIST_SIZE%howmany));
    for (int i=0; i<size; i++) {
        tmp = (input[i] - min) * HIST_SIZE / (max - min - 1);
        if ((tmp>=lower) && (tmp<lower+psize))
            histogram[tmp]++;
        }
    }
```

2.3 Para `frequency` sería necesario una comunicación colectiva tipo *scatter* desde el procesador 0, con el objetivo de distribuirlo entre todos los procesadores. Dado que no se inicializa, también se ha considerado válido si no se realiza esta comunicación. Sin embargo, es necesaria una comunicación tipo *gather* despues de la ejecución de `FindFrequency`.

Para la variable `max` es necesaria una comunicación colectiva tipo *broadcast* antes de `FindBounds` (dado que esta inicializada), una colectiva tipo *reduce* despues de `FindBounds` para combinar los máximos parciales en cada procesador y una colectiva tipo *broadcast* de nuevo antes de iniciar `FindFrequency` con el objetivo de distribuir a todos los procesadores el valor máximo encontrando.

3. Cuestiones (2.5 puntos: 0.5/0.5/0.75/0.75)

3.1 False sharing, se puede corregir utilizando padding (por ejemplo extendiendo a matriz el vector salida, dimensionando la segunda dimensión de manera que el numero de elementos ocupe una íinea de cache entera).

3.2 Compartición de variable j –¿ requiere privatización dentro de la región paralela. Deadlock, se puede evitar adquiriendo los locks siempre en el mismo orden.

3.3 El enunciado permite varias respuestas válidas, en función del orden de creación de las tareas en cada nivel de recursividad. Si estas se crean de izquierda a derecha, entonces la respuesta es $(3 + 4 + 3) \times t = 10 \times t$. Si es de derecha a izquierda, entonces la respuesta correcta es $(2 + 1 + 1) \times t = 4 \times t$.

3.4 El enunciado permite varias respuestas válidas, en función del orden de invocación de las llamadas en cada nivel de recursividad. Si estas se crean de izquierda a derecha, entonces la respuesta es $17 \times t$ (número de hojas a su izquierda). Si es de derecha a izquierda, entonces la respuesta correcta es $2 \times t$ (número de hojas a su derecha).

Segundo Control de PAR – Curso 2012/13-Q2

27 de Mayo de 2013

1. (3 puntos) Proponed dos códigos que reduzcan los `overheads` de sincronización `scheduling for/join` y de sincronización para compartición de datos. Los dos códigos propuestos se deben diferenciar, al menos, en la forma de reducir la sincronización por compartición de datos sin incorrurir en otros overheads, sabiendo que el código se ejecutará en una máquina `NUMA`:

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

2. (3 puntos) Suponiendo el siguiente código:

```
int compute_base(int *v, int n); // process n elements and return a integer value

int compute_rec(int *v, int n) {
   int n2;
   if (n<N_BASE) return compute_base(v,n);
   else {
     n2=n/2;
     return compute_rec(v,n/2) +
            compute_rec(v+n/2,n-n2);
   }
}
void main() {
   int result;
   compute_rec(v,N);
}
```

   (a) Proponed un código secuencial para poder utilizar una estrategia de paralelización que tienda a $T_\infty = log_4(n)$.

   (b) Escribe una versión paralela OpenMP del código del apartado anterior, que aplique `Task Decomposition` y el patrón de paralelización `Divide & Conquer`, sabiendo que nuestro objetivo es un T_p que tienda a $T_\infty = log_4(n)$.

   (c) Ahora añade lo necesario para ser consciente del coste de creación de tareas al código OpenMP propuesto en el apartado anterior.

   Nota: podéis proponer directamente la solución que englobe los tres apartados anteriores.

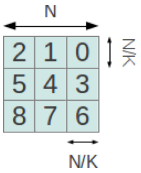
3. (3 puntos) Dado el siguiente código incompleto:

```
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.

4. (1 punto) 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?

Solution

1. (3 puntos)

   Una posible solución 1:

```
int i;
int nit;
int sum=0;
#pragma omp parallel num_threads(NT) reduction(+:sum)
{
 #pragma omp for collapse(2)
 for (nit=0; nit<NITER; nit++)
    for (i=0; i<N; i++)
         sum+=foo(nit,i);
}
```

   Una posible solución 2:

```
int sum=0;
int sumvector[NT][CACHE_LINE_SIZE/sizeof(int)];
#pragma omp parallel num_threads(NT)
{
 int nit;
 int id = omp_get_thread_num();
 sumvector[id][0]=0;
 #pragma omp for collapse(2)
 for (nit=0; nit<NITER; nit++)
    for (i=0; i<N; i++)
         sumvector[id][0]+=foo(nit,i);

 #pragma omp atomic
    sum+=sumvector[id][0];
}
```

2. (3 puntos)

```
int compute_base(int *v, int n); // process n elements and return a integer value

int compute_rec(int *v, int n, int d) {
   int n4;
   int c1,c2,c3,c4;
   if (n<N_BASE)
       return compute_base(v,n);
   else {
     n4=n/4;
     if (d<CUTOFF) // para b) no limitamos y N_BASE
                   // deberia tender a 1
     {
       #pragma omp task shared(c1)
       c1 = compute_rec(v,n4,d+1);
       #pragma omp task shared(c2)
       c2 = compute_rec(v+n4,n4,d+1);
       #pragma omp task shared(c3)
       c3 = compute_rec(v+2*n4,n4,d+1);
       #pragma omp task shared(c4)
       c4 = compute_rec(v+3*n4,n4+n%4,d+1);
       #pragma omp taskwait
       return  c1+c2+c3+c4;
     }
     else {
```

```
       return compute_rec(v,n4,d) +
              compute_rec(v+n4,n4,d) +
              compute_rec(v+2*n4,n4,d) +
              compute_rec(v+3*n4,n4+n%4,d);
     }
   }
}

void main() {
   int result;
   #pragma omp parallel
   #pragma omp single
   result = compute_rec(v,N,0);
}
```

3. (3 puntos)

Una posible solución caso 1

```
typedef struct {
    int i_start,i_end;
    int j_start,j_end;
} limits;

limits decomposition[num_threads];


void InitDecomposition(limits * decomposition, int N, int nt)
{
   int i;
   int remain = N%nt;
   int nelems = N/nt;
   for (i=0;i<remain;i++)
   {
      decomposition[i].i_start= i*(nelems+1);
      decomposition[i].i_end  = (i+1)*(nelems+1);
      decomposition[i].j_start= 0;
      decomposition[i].j_end= N;

   }

   for (; i<nt;i++)
   {
      decomposition[i].i_start= i*(nelems)+remain;
      decomposition[i].i_end  = (i+1)*(nelems)+remain;
      decomposition[i].j_start= 0;
      decomposition[i].j_end= N;
   }
}

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

  #pragma omp parallel
  #pragma omp single
  {
    InitDecomposition(decomposition,N,omp_get_num_thread());
  }

  #pragma omp parallel
  {
```

```
    int id = omp_get_thread_num();
    int i_start = decomposition[id].i_start;
    int i_end   = decomposition[id].i_end;
    int j_start = decomposition[id].j_start;
    int j_end   = decomposition[id].j_end;

    foo(i_start,i_end,j_start,j_end);
  }

}
```

Una posible solución caso 2

```
typedef struct {
    int i_start,i_end;
    int j_start,j_end;
} limits;

limits decomposition[num_threads];


void InitDecomposition(limits * decomposition, int N, int nt)
{
   int i;
   int k=sqrt(nt);
   int nelems = N/k;
   for (i=0;i<nt;i++)
   {
     decomposition[i].i_start = (i/k)*nelems;
     decomposition[i].i_end   = ((i/k)+1)*nelems;
     decomposition[i].j_start = ((k-1)-(i%k))*nelems;
     decomposition[i].j_end   = (k-(i%k))*nelems;
   }

}

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

#pragma omp parallel
#pragma omp single
{
  InitDecomposition(decomposition,N,omp_get_num_thread())
}

#pragma omp parallel
{
  int id = omp_get_thread_num();
  int i_start = decomposition[id].i_start;
  int i_end   = decomposition[id].i_end;
  int j_start = decomposition[id].j_start;
  int j_end   = decomposition[id].j_end;

  foo(i_start,i_end,j_start,j_end)
}

}
```

4. (1 punto)

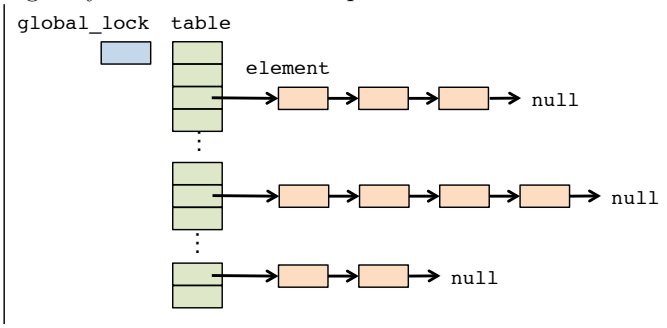
   (a) $t_{comm} = (nproc - 1) * (t_s + sizeToBeSent \times t_w)$

   (b) La topología de la red de interconexión puede afectar a la latencia de comunicación, además de favorecer o no la existencia de contención en las red según el patrón de comunicación. Por otro lado el tipo de interconexiones (ethernet, infiniband, etc.) determinarán el `bandwidth` en la transferencias entre cada una de las componentes de la red.

Segundo Control de PAR – Curso 2013/14-Q1
16 de Diciembre de 2013

1. (3.0 puntos) Suponed una tabla de "hash" implementada como un vector de listas encadenadas, tal como se muestra en la siguiente figura y en la definicion de tipos:

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

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

HashTable table;
```



Dentro de cada lista los elementos se guardan de forma ordenada por el valor del campo `data`. Suponed tambien el siguiente fragmento de código para insertar los elementos del vector `ToInsert` de tamaño `num_elem` en dicha tabla de "hash":

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

en el que la función `hash_function` retorna la entrada de la tabla (entre 0 y `SIZE_TABLE`-1) en la que debe de insertarse un determinado elemento y que la función `insert_elem` inserta dicho elemento en la posición correspondiente dentro de la lista encadenada apuntada por la entrada `index` de la `HashTable`.

Se pide:

(a) (0.5 puntos) Dada la secuencia de `index`={`5,10,14,10,25,25,10,8`} retornada por `hash_function` para un vector `ToInsert` con `num_elem`=8 elementos, dibujar en un diagrama temporal la ejecución paralela con 4 threads, asumiendo que la función `hash_function` tarda 2 unidades de tiempo, `insert_elem` tarda 5 unidades de tiempo y las funciones de "set" y "unset" del lock tardan 1 unidad de tiempo. El resto de operaciones podéis considerar que tardan un tiempo despreciable.

(b) (1.0 punto) Modificar las estructuras de datos y el código anterior para que se permitan inserciones en paralelo en distintas entradas de la `HashTable`.

(c) (0.75 puntos) Para la misma secuencia de valores de `index` anterior, dibujar de nuevo el diagrama temporal de la ejecución paralela con 4 threads para la implementación propuesta en el apartado b). ¿Cómo cambiaría dicho diagrama si la planificación del bucle fuera `(static,1)`? Dibujad el nuevo diagrama temporal para `(static,1)`.

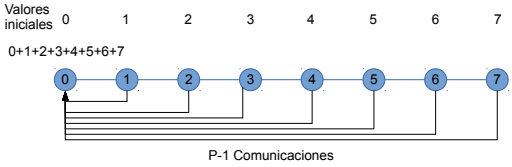
(d) (0.75 puntos) Modificar las estructuras de datos para permitir un mayor grado de concurrencia en la inserción ordenada de elementos dentro de una misma lista encadenada (NO es necesario implementar cambios en el código de la función `insert_elem` ni realizar el diagrama temporal de nuevo).

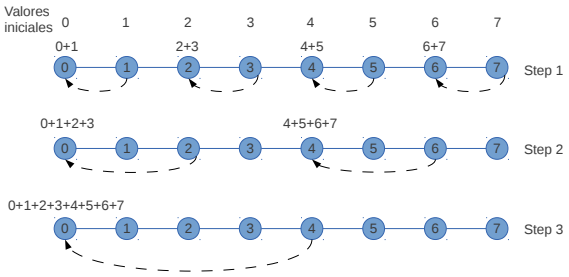
2. (2.0 puntos) Siguiendo con el código original del anterior ejercicio, y sin utilizar `omp for` o `omp parallel for`, **se pide:**

   (a) Redefinir las estructuras de datos, si fuera necesario, y escribir una versión paralela OpenMP que obedezca a una estrategia *block geometric data decomposition* de los datos de salida (tabla de hash, `table`) y que cumpla la regla *owner-computes*.

   (b) Redefinir las estructuras de datos, si fuera necesario, y escribir una versión paralela OpenMP que obedezca a una estrategia *cyclic geometric data decomposition* de los datos de entrada (`ToInsert`) y que cumpla la regla *owner-computes*.

3. (2.0 puntos) En la figura siguiente se muestran dos implementaciones diferentes de un MPI_Reduce (la operación de reducción es la suma), utilizando únicamente comunicaciones punto a punto, para una red de interconnexión de tipo linear para 8 procesadores. Cada punto es un procesador y en ambos esquemas de comunicación se parte de los mismos valores iniciales. En el esquema de la izquierda el proceso 0 recibe una comunicación directa de cada uno de los demás procesadores. En el esquema de la derecha se precisan tres pasos de comunicaciones con reducciones parciales, hasta llegar a la reducción final.





Responde las siguientes preguntas:

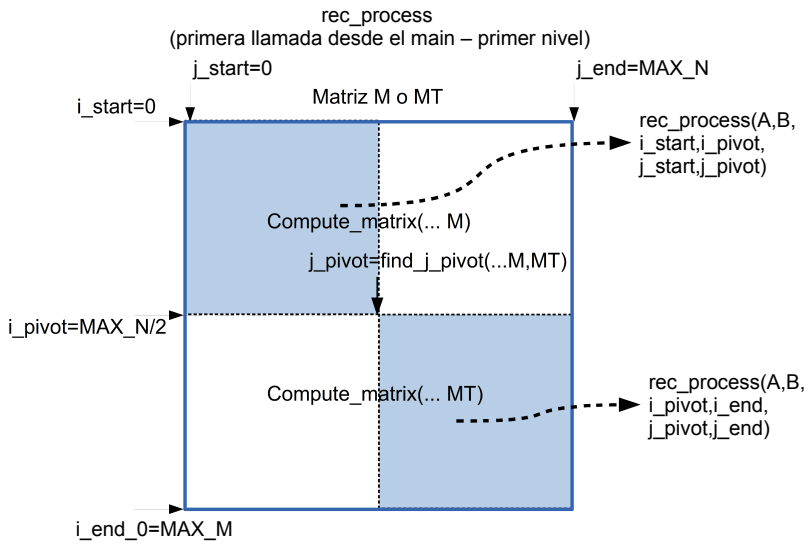
(a) (1.0 punto) Sabemos que el coste de SOLO comunicación de la estrategia de la izquierda, siguiendo el modelo de compartición de datos visto en clase, es $T_{comm} = (P-1)*(t_s+1*t_w)$. Determina el coste de comunicación para la estrategia de la derecha de la figura, en función de un número P de procesadores, siguiendo el modelo de compartición de datos visto en clase.

(b) (1.0 punto) Siguiendo la estrategia de la izquierda de la figura para la reducción, explica, sin implementarlo, una estrategia de comunicación entre procesadores para realizar un `MPI_Gather` en una red de interconexión linear con 8 procesadores, en el que el proceso 0 hace de destino. Determina el coste de SOLO comunicación para el algoritmo que planteas para el caso específico de 8 procesadores.

4. (3 puntos) Queremos paralelizar el siguiente código:

```
char A[MAX_M], B[MAX_N];
solution_t  global_stack_sol;
void main() {
    ...
    rec_process(A,B,0,MAX_M,0,MAX_N);
    ...
}
void rec_process(char *A,char *B,int i_start, int i_end, int j_start, int j_end)
{
   int M[MAX_M][MAX_N], MT[MAX_M][MAX_N];
   int i_pivot,j_pivot;
   solucion_t sol;

   if (caso_base(i_start,i_end,j_start,j_end)) {
      sol = prepare_leaf_sol(A, B, i_start, i_end, j_start, j_end);
      push_solution(&global_stack_sol, sol);
   } else {
      i_pivot = (i_start+i_end)/2;

      // calcula la mitad superior de la matriz M
      // en funcion de A, B y los limites
      compute_matrix(A,i_start,i_pivot,B,j_start,j_end,M)

      // calcula la mitad inferior de la matriz MT
      // en funcion de A,B y los limites
      compute_matrix(A,i_pivot,i_end,B,j_start,j_end,MT)

      // obtiene j_pivot y sol en funcion de los valores calculados
      // de M y MT en compute_matrix(... M) y  compute_matrix(.... MT)
      <j_pivot,sol>=find_j_pivot(i_start,i_pivot,i_end,M,MT)

      // check_solution solo consulta i_pivot, j_pivot y sol
      if (!check_solution(i_pivot,j_pivot,sol)) {

          rec_process(A,B,i_start,i_pivot,j_start,j_pivot);
          rec_process(A,B,i_pivot,i_end,j_pivot,j_end);
      }
      else {
          // Dos soluciones
          sol = prepare_sol(A,i_pivot)
          push_solution(&global_stack_sol, sol)
          sol = prepare_sol(A,i_pivot+1)
          push_solution(&global_stack_sol, sol)

          rec_process(A,B,i_start,i_pivot-1,,j_start,j_pivot);
          rec_process(A,B,i_pivot+1,i_end,  ,j_pivot,j_end);
      }
   }
}
```

Para entenderlo mejor, en la siguiente figura os mostramos esquemáticamente lo más representativo de la ejecución de la primera llamada recursiva a la función `rec_process` desde el programa principal:



La figura muestra la matriz `M` o `MT` dependiendo de la llamada a `compute_matrix(... M o MT)`. Si miramos la figura y el código, observamos que la función `rec_process` llama dos veces a `compute_matrix`: `compute_matrix(... M)` y `compute_matrix(... MT)`. Ambos cálculos (llamadas a `compute_matrix`) son independientes pero se necesitan para calcular `j_pivot` con la función `find_j_pivot(...M,MT)`. Una vez obtenido `j_pivot`, la función `check_solution`, que aparece en el segundo `if` del código, consulta los valores de los parámetros para determinar si se añade o no una nueva solución. En cualquier caso se harán dos llamadas recursivas a la función `rec_process`.

Responde a las siguientes cuestiones, sabiendo que ninguna de las funciones realiza operaciones de lecturas y/o escrituras sobre variables globales que no se pasen como parámetro y que ninguna función modifica los vectores A y B. También sabemos que la función `push_solution` es un código secuencial que forma parte de una librería que no soporta paralelismo: es decir, no dispone de ningún control en su código para evitar `Data Race`. Sin embargo, no importa en que orden se realizan los `push_solution` en la ejecución. Finalmente, se sabe que la función `compute_matrix` es un código secuencial que no se ha paralelizado.

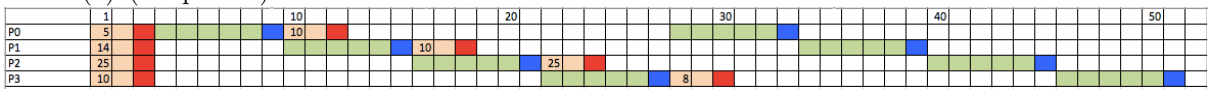
(a) (0.5 puntos) **Indica cual es el speedup ideal (S_∞) que se obtendría paralelizando únicamente el cálculo realizado internamente por `compute_matrix`**. Para ello suponed que, para la entrada que se quiere ejecutar el programa, esta función ocupa el 90% del tiempo de ejecución secuencial del programa.

(b) (2.5 puntos) Escribe una versión paralela del código que tenga en cuenta los costes de creación de tareas y de sincronización entre tareas. Recordad todos los supuestos del enunciado para garantizar el correcto funcionamiento de vuestro código paralelo.

Solution

1. (3.0 puntos)

   (a) (0.5 puntos)



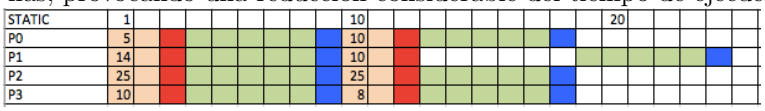
   siendo naranja: hash_function; rojo: omp_set_lock; verde: insert_elem; y azul: omp_unset_lock.

   (b) (1.0 punto)

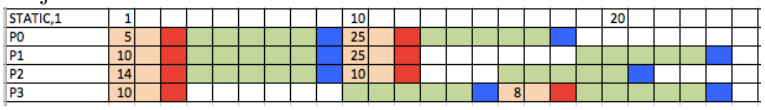
```
typedef struct {
    omp_lock_t lock[SIZE_TABLE];
    element *  table[SIZE_TABLE];
} HashTable;
...
#pragma omp parallel for
for (i=0; i<SIZE_TABLE; i++) omp_init_lock(&table.lock[i]);
#pragma omp parallel for private(index) schedule(static)
    for (i = 0; i < num_elem; i++) {
        index = hash_function(elems[i], SIZE_TABLE);
        omp_set_lock (&table.lock[index]);
        insert_elem (table.table, elems[i], index);
        omp_unset_lock (&table.lock[index]);
    }
#pragma omp parallel for
for (i=0; i<SIZE_TABLE; i++) omp_destroy_lock(&table.lock[i]);
...
```

   (c) (0.75 puntos)

   `(static)`: con la nueva implementación se permiten inserciones en distintas entradas de la tabla de has, provocando una reducción considerable del tiempo de ejecución:



   `(static,1)`: Los threads ejecutarán las inserciones en un orden distinto, pero sin afectar al tiempo de ejecución total:



   (d) (0.75 puntos)

```
typedef struct {
    int data;
    omp_lock_t lock;
    element * next;
} element;

typedef struct {
    element *  table[SIZE_TABLE];
} HashTable;
```

2. (2.0 puntos)

(a)
```
    // No necesitamos locks
    element *  table[SIZE_TABLE];
...
#pragma omp parallel private(index)
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    int index_start = id*SIZE_TABLE/nt;
    int index_end   = (id==nt-1)?SIZE_TABLE:(id+1)*(SIZE_TABLE/nt);

    // No se necesitan locks
    for (i = 0; i < num_elem; i++) {
        index = hash_function(elems[i], SIZE_TABLE);
        if (index>=index_start && index<index_end)
             insert_elem (table, elems[i], index);
    }

}
...
```

(b)
```
typedef struct {
    omp_lock_t lock[SIZE_TABLE];
    element *  table[SIZE_TABLE];
} HashTable;
...
#pragma omp parallel for
for (i=0; i<SIZE_TABLE; i++) omp_init_lock(&table.lock[i]);

#pragma omp parallel private(index)
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();

    // cyclic geometric decomposition. Comienza en id y va saltando
    // de num_threads en num_threads
    for (i = id; i < num_elem; i+=nt) {
        index = hash_function(elems[i], SIZE_TABLE);
        omp_set_lock (&table.lock[index]);
        insert_elem (table.table, elems[i], index);
        omp_unset_lock (&table.lock[index]);
    }
}
#pragma omp parallel for
for (i=0; i<SIZE_TABLE; i++) omp_destroy_lock(&table.lock[i]);
...
```

3. (2.0 puntos)

(a) (1.0 punto)
Suponiendo P potencia de 2:
$T_{comm} = \log P \times (t_s + 1 * t_w)$

(b) (1.0 punto)
Siguiendo la estrategia de la izquierda el coste el `MPI_Gather` será el de enviar los trozos de m elementos al procesador 0, que los irá colocando consecutivamente en un vector: $T_{comm} = (P-1) * (t_s + m * t_w)$. En el caso del ejemplo m es 1.

4. (3 puntos)

   (a) (0.5 puntos)
   $S_\infty = \frac{1}{(1-0.9)} = 10\times$

   (b) (2.5 puntos)

```
char A[MAX_M], B[MAX_N];
solution_t  global_stack_sol;

#define MAX_DEPTH 3

void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    rec_process(A,B,0,MAX_M,0,MAX_N,0);
    ...
}
void rec_process(char *A,char *B,int i_start, int i_end, int j_start,
                  int j_end, int depth)
{
   int M[MAX_M][MAX_N], MT[MAX_M][MAX_N];
   int i_pivot,j_pivot;
   solucion_t sol;

   // Si depth es superior al nivel del arbol donde dejamos crear tasks,
   // no crearemos mas tasks recursivas ni tasks para compute_matrix.
   // Sin embargo, tenemos que asegurar los criticals en los push_solutions

   if (caso_base(i_start,i_end,j_start,j_end)) {
        sol = prepare_leaf_sol(A, B, i_start, i_end, j_start, j_end);
        #pragma omp critical
        push_solution(&global_stack_sol, sol);
   } else {
     if (depth>=MAX_DEPTH)
     {
        i_pivot = (i_start+i_end)/2;
        compute_matrix(A,i_start,i_pivot,B,j_start,j_end,M)
        compute_matrix(A,i_pivot,i_end,B,j_start,j_end,MT)
        <j_pivot,sol>=find_j_pivot(i_start,i_pivot,i_end,M,MT)

        if (!check_solution(i_pivot,j_pivot,sol)) {
          rec_process(A,B,i_start,i_pivot,j_start,j_pivot,depth);
          rec_process(A,B,i_pivot,i_end,j_pivot,j_end,depth);
        }
        else {
          sol = prepare_sol(A,i_pivot)
          #pragma omp critical
          push_solution(&global_stack_sol, sol)
          sol = prepare_sol(A,i_pivot+1)
          #pragma omp critical
          push_solution(&global_stack_sol, sol)

          rec_process(A,B,i_start,i_pivot-1,,j_start,j_pivot,depth);
          rec_process(A,B,i_pivot+1,i_end,  ,j_pivot,j_end,depth);
        }
```

```
      }
       else {
         i_pivot = (i_start+i_end)/2;
         #pragma omp task
         compute_matrix(A,i_start,i_pivot,B,j_start,j_end,M)
         #pragma omp task
         compute_matrix(A,i_pivot,i_end,B,j_start,j_end,MT)
         #pragma omp taskwait
         <j_pivot,sol>=find_j_pivot(i_start,i_pivot,i_end,M,MT)

         if (!check_solution(i_pivot,j_pivot,sol)) {
           #pragma omp task
           rec_process(A,B,i_start,i_pivot,j_start,j_pivot,depth+1);
           #pragma omp task
           rec_process(A,B,i_pivot,i_end,j_pivot,j_end,depth+1);
         }
         else {
           sol = prepare_sol(A,i_pivot)
           #pragma omp critical
           push_solution(&global_stack_sol, sol)
           sol = prepare_sol(A,i_pivot+1)
           #pragma omp critical
           push_solution(&global_stack_sol, sol)
           #pragma omp task
           rec_process(A,B,i_start,i_pivot-1,,j_start,j_pivot,depth+1);
           #pragma omp task
           rec_process(A,B,i_pivot+1,i_end,  ,j_pivot,j_end,depth+1);
         }

       }
   }
}
```

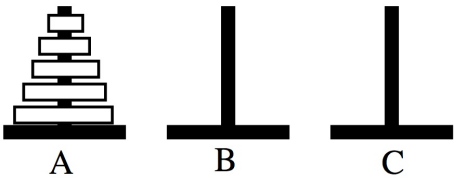
Parallelism - 2^{nd} In-term Exam - 2013/14 -Q1

December 16th, 2013

1. (3 points)

   The Tower of Hanoi is a puzzle that consists of three pegs and a set of disks D. Each disk has a different diameter and a hole in the middle so that the disk can fit onto any of the pegs. The initial puzzle setup has two of the pegs empty and all the disks on the third peg (source) in monotonically decreasing order of diameter from bottom to top, which forms a structure that is reminiscent of a tower. The goal of the puzzle is to move the tower from the source peg to a specified peg (destination) using the other peg to temporarily hold disks. The two rules of the Tower of Hanoi puzzle are that only one disk at a time can be moved from the top of a stack of disks on a peg to some other peg, and disks with larger diameter cannot be placed on top of smaller diameter disks. The problem can be solved recursively with the following algorithm:

```
Move N disks from Start to Finish
if N > 0 then
  Move N-1 disks from Start to Extra
  Move one disk from Start to Finish
  Move N-1 disks from Extra to Finish
```



   We can visualize the solution moves of the Tower of Hanoi organized as a tree. It will be a perfect binary tree where each node is a move of one disk from a source peg to a destination peg. The left child of a non-leaf node is the tree solving the first recursive call, while the right child is the tree solving the second recursive call.

   Though the disks can only be moved one at a time, we want to obtain the sequence of movements in parallel. We want to keep the list of movements stored in an array which can later be used to print the sequence of movements. Luckily, with the Tower of Hanoi, one can compute where each and every executed output statement will fall within the entire output results. This is handled with idx and offset in the code below, though we do not really need to understand the details for the purpose of parallelizing the code. What is important is that we can do the computations simultaneously since we know that the current node, and each of its child subtrees access different memory positions.

```
// define numnodes as 2^D
char plan[numnodes][2]; /* Define space to keep the 2^D - 1 moves
                           starting from position plan[1][*] */

void tower_Hanoi(char src, char dest, char temp, int idx, int offset)
{
  if (offset > 0) {
    tower_Hanoi(src, temp, dest, idx-offset, offset/2);
    /* Store that at this stage src has to be moved into dest. */
    plan[idx][0] = src; plan[idx][1] = dest;
    tower_Hanoi(temp, dest, src, idx+offset, offset/2);
  }
  else {
    /* Store that at this stage src has to be moved into dest. */
    plan[idx][0] = src; plan[idx][1] = dest;
  }
}
```

```
main (...)
{
  // Initializations:
  // Set idx to 2^(D-1)
  // Set offset to 2^(D-2)

  /* We have to move tower A with n disks to tower
     C using tower B as an auxiliary tower. */
  ...
  tower_Hanoi('A', 'C', 'B', idx, offset);
  ...

}
```

We ask you to write an efficient parallel code using OpenMP controlling that the number of pending tasks to be executed is not higher than `MAX_TASKS` multiplied by the number of available threads.

(a) Write the representative excerpt of parallel code for the main.

(b) Write the parallel code for subroutine `tower_Hanoi`.

2. (3 points) We have two serial versions of a code that computes the product of two triangular matrices L and U. L is lower triangular, which means all elements above the main diagonal are zero. U is upper triangular, which means all elements below the main diagonal are zero.

```
//  Version 1
for ( j=0; j<N; j++ )
 for ( i=0; i<N; i++ )
  for ( k=0; k<MIN(i,j); k++ )
   {
    C[i][j] += L[i][k] * U[k][j];
   }
```

```
//  Version 2 (Assuming N is even)
for ( j=0; j<N/2; j++ )
 for ( i=0; i<N; i++ )
  for ( k=0; k<MIN(i,j); k++ )
   {
    C[i][j] += L[i][k] * U[k][j];
    C[i][N-j-1] += L[i][k] * U[k][N-j-1];
   }
```

We ask you to write an efficient parallel code using OpenMP that takes into account the load balance.

(a) Write the representative excerpt of parallel code for version 1. Justify what you do to obtain the best possible performance.

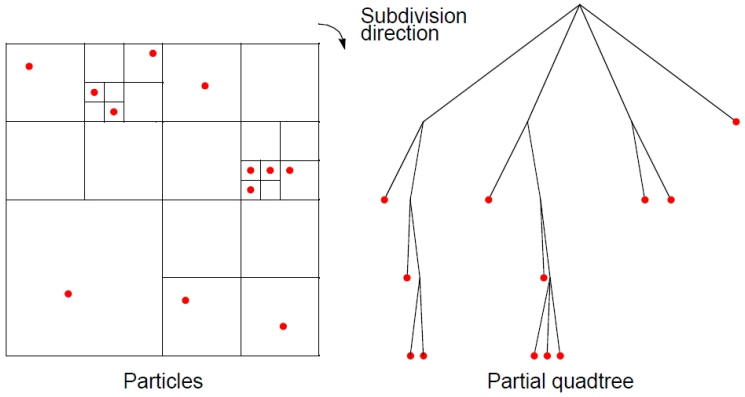
(b) Write the representative excerpt of parallel code for version 2. Justify what you do to obtain the best possible performance.

3. (3 points) 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 one or more instances of `doComputation`?

(b) Assuming that the execution of `doComputation` takes 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 equivalent parallel code that makes use of OpenMP tasks, in which the task creation is limited at the same level of recursion.

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

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

4. (1 point) Write a pseudo-code that implements the MPI_BCAST primitive using MPI primitives other than the broadcast.

Solution

1. (3 points)

   (a) Representative excerpt of parallel code for the main:

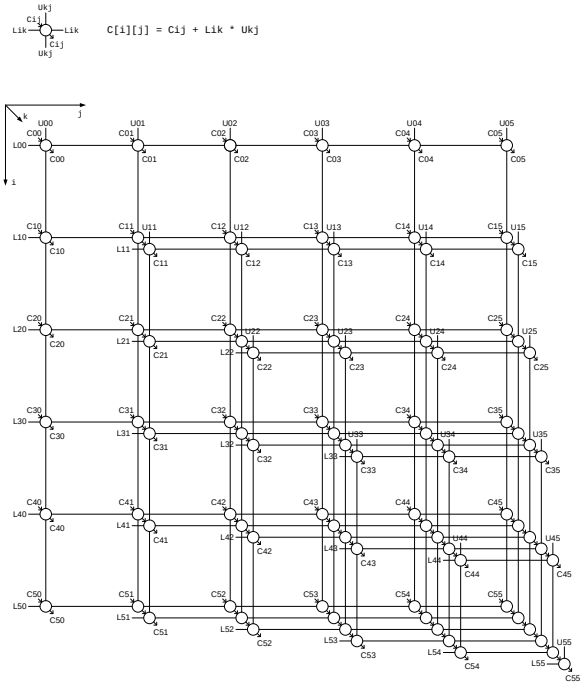
```
// define MAX_TASKS with the desired value
// define numnodes as 2^D

char plan[numnodes][2]; /* Define space to keep the 2^D - 1 moves
                           starting from position plan[1][*] */
int pending=1;
int num_max_tasks;

main (...)
{
  // Initializations:
  // Set idx to 2^(D-1)
  // Set offset to 2^(D-2)

  /* We have to move tower A with n disks to tower
     C using tower B as an auxiliary tower. */
  ...
  #pragma omp parallel
  {
    num_max_tasks = MAX_TASKS * omp_get_num_threads() - 2;
    #pragma omp single
    tower_Hanoi('A', 'C', 'B', idx, offset);
  }
  ...
}
```

   (b) Parallel code for subroutine `tower_Hanoi`:

```
void tower_Hanoi(char src, char dest, char temp, int idx, int offset)
{
  #pragma omp atomic
  pending--;

  if ( offset > 0 ) {
    if ( pending <= num_max_tasks ) {
      #pragma omp atomic
      pending += 2;
      #pragma omp task
      tower_Hanoi(src, temp, dest, idx-offset, offset/2);
      /* Store that at this stage src has to be moved into dest. */
      plan[idx][0] = src; plan[idx][1] = dest;
      #pragma omp task
      tower_Hanoi(temp, dest, src, idx+offset, offset/2);
    }
    else {
      tower_Hanoi(src, temp, dest, idx-offset, offset/2);
      /* Store that at this stage src has to be moved into dest. */
      plan[idx][0] = src; plan[idx][1] = dest;
      tower_Hanoi(temp, dest, src, idx+offset, offset/2);
```

```
      }
    }
    else {
      /* Store that at this stage src has to be moved into dest. */
      plan[idx][0] = src; plan[idx][1] = dest;
    }
  }
```

2. (3 points) Let us consider the data dependence graph shown in the figure below:



We can observe that the amount of work necessary to compute an element of matrix C increases as we increase i and/or j. With the goal of obtaining good load balancing, for both versions we have considered as correct any solution that includes a `parallel for` before the outer loop, defining a *chunk size equal to one* for any scheduling policy.

3. (3 points)

   (a) During the tree traversal the code tries to use 4 threads each time the current subtree is not a leaf and the recursion level, indicated in variable d, is less than 3. Thus, it tries to use the following number of threads for each level: 4 when d is 0; 3×4 when d is 1; and 2×4 when d is 2.

   If nested parallelism is disabled or not supported, then the new team that is created by a thread encountering a parallel construct inside a parallel region will consist only of the encountering thread. Thus, during the tree traversal only four threads will be created and will execute routine `doComputation`.

   However, if nested parallelism is supported and enabled, then the new team can consist of more than one thread. When a thread encounters a parallel construct, the thread creates a team composed of itself and some additional (possibly zero) number of threads. The encountering thread becomes the master of the new team.

   We assume we can create as many threads as we want in as many nested levels as we want. Thus, during tree traversal 19 threads will be created and 9 of them will execute routine `doComputation`.

   We have also considered as correct answers with a new set of 4 threads for each level of nested parallelism, i.e. considering that the new threads are different from the one that encountered

the parallel construct. In that case the answer would be 24 threads created during the tree traversal with 9 of them executing routine `doComputation`.

(b) The parallel time will be that of executing the critical path, which corresponds to the subtree that ends with three leaves. The execution of that subtree will imply: on one hand the creation of 3 parallel regions, taking 4 time units each; and on the other hand, a single thread will have to execute the three leaves, i.e. execute routine `doComputation` three times.

$T_p = 3 \times 4 + 3 \times 25 = 12 + 75 = 87$ time units

(c) A new equivalent parallel code that makes use of OpenMP tasks, in which the task creation is limited at the same level of recursion:

```
void traverse(TreeNode* subTree, int d) {
  int i, j;
  if(subTree) {
     if(!subTree->isLeaf) {
        if(d<3)
           for(i=0; i<2; i++)
              for(j=0; j<2; j++)
                 #pragma omp task
                 traverse(subTree->quadrant[i][j], d+1);
           #pragma omp taskwait  // This taskwait is not necessary to have a
                                 // correct code but is added to have an
                                 // equivalent parallel code
        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() {
      ...
      #pragma omp parallel
      #pragma omp single
      traverse(root, 0);
      ...
}
```

(d) The number of threads created will depend on the value set through the environment variable OMP_NUM_THREADS, or via a call to omp_set_num_threads() if it was included in the code. The number of tasks created during the traversal of the tree shown above is 24.

(e) The parallel time will be that of executing the critical path, which corresponds to the subtree that ends with three leaves. The execution of that subtree will imply: on one hand the creation of the tasks for each of the three levels of recursion; and on the other hand, a single thread will have to execute the three leaves, i.e. execute routine `doComputation` three times. At each level of recursion, a thread will be creating the tasks sequentially. Since we do not know which of the four subtrees is the one shown in the figure we will consider the worse case. The creation of the tasks will be bounded by the creation of the last one at each level, i.e. 4×2 time units for each of the three levels of recursion. Thus,

$T_p \leq 3 \times (4 \times 2) + 3 \times 25 = 24 + 75 = 99$ time units

4. (1 point)

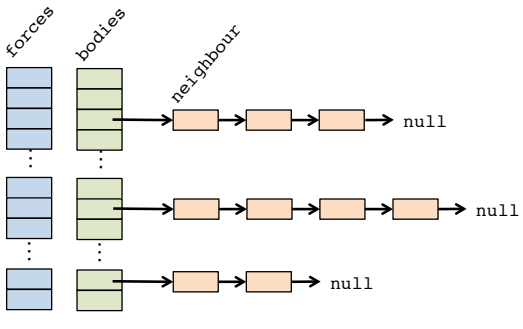
   Since we are not required to implement an efficient parallel code it suffices to provide a simple serial implementation:

```
#include "mpi.h"
int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, int root,
                      MPI_Comm comm )
{
 ...
 int nprocs, myid;
 int tag = 0;
 MPI_Status status;

 MPI_Comm_size(comm, &nprocs);
 MPI_Comm_rank(comm, &myid);

 if ( myid == root ) {
   for (int i=0; i < nprocs; i++)
    {
     if ( i != root )
     {
       MPI_Send ( buffer, count, datatype, i, tag, comm);
     }
    }
 }
 else {
    MPI_Recv ( buffer, count, datatype, root, tag, comm, &status);
    /* Could check status here. */
 }
}
```

Segundo Control de PAR – Curso 2013/14-Q1
16 de Diciembre de 2013

Dada la siguiente definición de estructura de datos para un problema *N-body* que almacena los cuerpos (vector `bodies` de tamaño `NUM_BODIES`) que forman un sistema y, para cada uno de ellos, la lista de cuerpos vecinos con los que interacciona. Para cada cuerpo se almacena un puntero a sus datos (`data`) y un puntero al primer elemento de la lista de vecinos (`first`), cada elemento de tipo `neighbour`. Para cada vecino en la lista, se almacena el identificador (`body_id`) que permite acceder al vector `bodies` y un puntero al siguiente vecino en la lista (número de vecinos en cada lista variable, último vecino apuntando a `null`). Además se dispone de un vector `forces`, del mismo tamaño que `bodies`) que almacena la fuerza que se ejerce sobre dicho cuerpo por parte de sus vecinos.

```
#define NUM_BODIES 1048576
typedef struct {
    int body_id;
    neighbour * next;
} neighbour;

typedef struct {
    theBody * data;
    neighbour * first;
} bodies[NUM_BODIES];

double forces[NUM_BODIES];
```



Suponiendo que se dispone de la función:

```
double compute_force (theBody * b, neighbour * n);
```

que calcula la fuerza que ejercen los cuerpos de una lista apuntados por `n` sobre el cuerpo apuntado por `b`, y la paralelización mostrada a continuación:

```
#define tmax 10000
for (timestep = 0; timestep < tmax; timestep++) {
    #pragma omp parallel for schedule(static)
    for (int body = 0; body < NUM_BODIES; body++) {
        forces[body] += compute_force (bodies[body].data, bodies[body].first);
    }
}
```

Pregunta 1 (0.5 puntos) Indicar la causa principal que puede provocar que el tiempo de ejecución paralela T_p con p procesadores este muy alejado del ideal T_1/p.

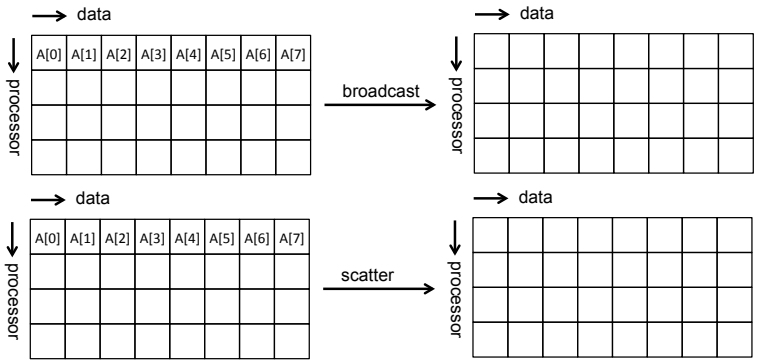
Pregunta 2 (1 punto) Si se cambia el reparto de iteraciones a threads por `schedule(dynamic, 1)`, indicar las dos causas principales que provocarán un tiempo de ejecución T_p con p procesadores muy alejado del ideal T_1/p.

Pregunta 3 (2 puntos) Siguiendo con el problema anterior, y con el objetivo de mejorar el tiempo de ejecución paralela se propone utilizar una *data decomposition* que se determina en tiempo de ejecución: en una primera fase se ejecuta UNA iteración de *Inspección* en la que se determina la asignación de cuerpos a *threads* y una segunda fase de *Ejecución* en la que se ejecutan el resto de iteraciones (`tmax - 1`) siguiendo (recordando) la *data decomposition* determinada en la fase de *Inspección*, tal como se muestra en el siguiente código incompleto:

```
#define tmax 10000
... // definir estructura de datos

// FASE DE INSPECCION
#pragma omp parallel for schedule(dynamic, 1)
for (int body = 0; body < NUM_BODIES; body++) {
    ... = omp_get_thread_num();
    forces[body] += compute_force (bodies[body].data, bodies[body].first);
}
// FASE DE EJECUCION
for (timestep = 1; timestep < tmax; timestep++) {
    #pragma omp parallel
    for (int body = 0; body < NUM_BODIES; body++) {
        if (...)  forces[body] += compute_force (bodies[body].data, bodies[body].first);
    }
}
```

Se pide: completar el código anterior, definiendo una nueva estructura de datos que permita implementar la propuesta, evitando la aparición de *false sharing* en el acceso a la misma.

Pregunta 4 (1 punto) Completar los dos siguientes diagramas para expresar la diferencia entre las comunicaciones colectivas `MPI_broadcast` y `MPI_scatter`, suponendo que en ámbos casos el procesador 0 es el que realiza el envío del vector `A`.



Pregunta 5 (2.5 puntos) Un estudiante ha venido a consultas con la siguiente solución para el problema 3 del tema *Task decomposition* en la colección de problemas. El código en C equivalente, el cuál realiza la reordenación aleatoria de las columnas de la matriz `iparent`, como parte de un algoritmo genético, es:

```
#define NPOP 1 << 10     // numero de cromosomas
#define LCHROME 1 << 24  // longitud de un cromosoma
int iparent[LCHROME][NPOP], temp1;
double fitness[NPOP], temp2;
int j, jother; // columnas a intercambiar
omp_lock_t lock[NPOP];

// retorna valor aleatorio: 0 <= valor < max, distinto de num
int random (int num, int max);

void main() {
    // inicializacion del vector de locks con omp_init_lock (no escribir)
    #pragma omp parallel ... // completar clausulas de datos, si necesario
    #pragma omp single
    for (j=0; j<NPOP; j++) {
        jother = random(j, NPOP);

        // Intercambio de cromosomas
        #pragma omp task ... // completar clausulas de datos, si necesario
        {
            omp_set_lock(lock[j]);
```

```
            for (long i = 0; i < NCOLS; i++) {
                temp1 = matriz[i][jother];
                matriz[i][jother] = matriz[i][j];
                matriz[i][j] = temp1;
            }
            temp2 = fitness[jother];
            finess[jother] = fitness[j];
            fitness[j] = temp2;
            omp_unset_lock(lock[j]);
        }
    }
    // destruccion del vector de locks omp_destroy_lock (no escribir)
}
```

La respuesta del profesor fué: "Deberias pensar si los *pragmas* `parallel` y `task` requiren alguna cláusula de datos y completar la sincronización para que la ejecución paralela no provoque condiciones de carrera (*race conditions*) ni *deadlock*." **Se pide:** realiza los cambios que el estudiante debería de haber realizado después de ir a consultas.

Pregunta 6 (3 puntos) Se quiere paralelizar una función que cuenta el número total de ficheros que hay en una estructura de directorios. Para ello se dispone del tipo de datos `File` y funciones que nos lo permiten manipular (ver cabecera en el código adjunto para un par de ellas):

```
int isFile (File f); // devuelve 1 si f es fichero, 0 si es un directorio
File fileOrDir (File f); // devuelve el siguiente fichero/directorio al mismo nivel

int countFiles(File f) {
    if (isFile(f)) return 1;

    // Contar hijos recorriendo recursivamente subdirectorios
    int count = 0;
    File next = f;
    for (int i = 0; i < numChildren(f); i++) {
        count += countFiles( next );
        next = fileOrDir(next);
    }
    return count;
}

int totalFiles;
void main() {
    File root = ... // inicializacion de root
    totalFiles = countFiles(root);
    ...
}
```

Se pide: modificar el código, si fuera necesario, e insertar los pragmas *OpenMP* necesarios para realizar en paralelo dicho conteo, limitando la creación de tareas hasta un nivel determinado (`MAX_LEVEL`) de recursividad.

Solution

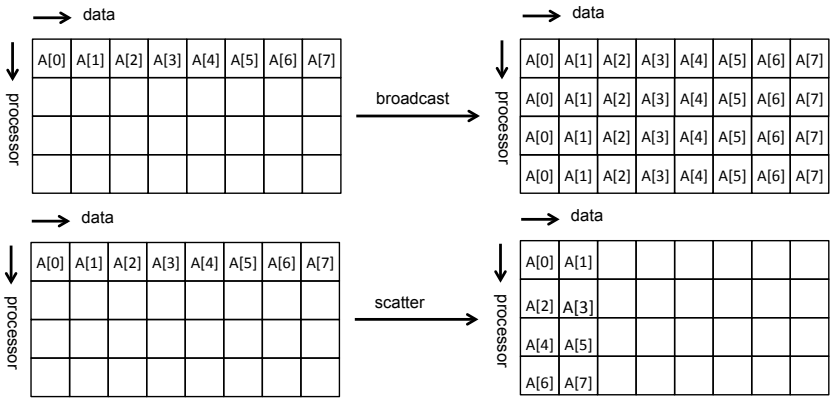
Pregunta 1 (0.5 puntos)
Dado que cada cuerpo del vector `bodies` puede tener un número de vecinos distinto, una planificación `static` puede provocar un problema de desbalance de carga (*load balance*).

Pregunta 2 (1 punto)
Al cambiar la planificación a `(dynamic,1)` el problema de *load balance* puede desaparecer, pero por contra nos aparece una sobrecarga (*overhead*) de planificación de iteraciones a *threads* que puede ser excesiva. Con un tamaño de `chunk` adecuado podríamos reducir drástitamente dicho *overhead* y además eliminar el segundo problema que esta relacionado con el *false sharing*, dado que *threads* consecutivos pueden escribir sobre elementos consecutivos del vector `forces`. Finalmente, un tercer motivo podría ser la poca localidad espacial que ofrece la planificación dinámica, ya que una misma iteración no tiene por que ser ejecutada por el mismo *thread* en distintas iteraciones del bucle `timestep`.

Pregunta 3 (2 puntos)

```
who[NUM_BODIES][CACHE_LINE_SIZE/4] // vector para recordar quien actualizo  cada cuerpo
#pragma omp parallel for schedule(dynamic, 1)
for (int body = 0; body < NUM_BODIES; body++) {
    who[body][0] = omp_get_thread_num();
    forces[body] += compute_force (bodies[body].data, bodies[body].first);
}
for (timestep = 1; timestep < tmax; timestep++) {
    #pragma omp parallel
    for (int body = 0; body < NUM_BODIES; body++) {
        if (who[body][0] == omp_get_thread_num())
            forces[body] += compute_force (bodies[body].data, bodies[body].first);
    }
}
```

Pregunta 4 (1 punto)



Pregunta 5 (2.5 puntos)
Es necesario garantizar que ámbas columnas `j` y `jother` quedan protegidas sin provocar deadlock. Para ello los adquiriremos en orden. Adem—'as, la tarea necesita una copia `firstprivate` de las variables `j` y `jother`, así como una copia privada de las variables temporales `temp1` y `temp2`.

```
#pragma omp task private(temp1, temp2) firstprivate(j, jother)
{
    if ( j < jother ) {
        omp_set_lock(lock[j]); omp_set_lock(lock[jother]);
    } else {
        omp_set_lock(lock[jother]); omp_set_lock(lock[j]);
    }
    ...
    omp_unset_lock(lock[j]); omp_unset_lock(lock[jother]);
}
```

Pregunta 6 (3 puntos)
En el programa principal necesitamos crear los *threads* que participarán en la ejecución paralela y que uno de ellos empiece con la generación de tareas mientras el resto se quedan esperando en la barrera para la ejecución de tareas. La función `countFiles` recibirá un nuevo argumento para controlar el nivel de recursividad, inicialmente a 0.

```
#pragma omp parallel
#pragma imp single
    totalFiles = countFiles(root, 0);
```

En función puede ser como muestra a continuación, creando una nueva tarea para recorrer un subdirectorio, protegiendo la actualización de la variable compartida `count` con `atomic` (no `critical` ya que éste provocaría la ejecución sequencial de todas las tareas) para evitar posibles condiciones de carrera. También es necesario esperar a que todas las tareas de un mismo nivel finalicen antes de retornar con el valor total de `count`.

```
int countFiles(File f, int level) {
    if (isFile(f)) return 1;
    int count = 0;
    File next = firstChild(f);
    if (level < MAX_LEVEL) {
        for (int i = 0; i < numChildren(f); i++) {
            #pragma omp task shared(count)
            #pragma omp atomic
                count += countFiles( next, level + 1 );
            next = fileOrDir(next);
        }
        #pragma omp taskwait
    } else {
        for (int i = 0; i < numChildren(f); i++) {
            count += countFiles( next, level );
            next = fileOrDir(next);
        }
    }
    return count;
}
```

Segundo Control de PAR – Curso 2013/14-Q2
4 de Junio de 2014

Pregunta 1 (3 puntos) Nos han dado la version paralela de un código que cuenta el número de veces que aparece una determinada clave `key` en los ficheros que forman parte de una estructura de directorios. Para ello se dispone del tipo de datos `File` y de las funciones siguientes:

- `int isFile (File f)`: función que devuelve 1 si `f` es fichero, 0 si es un directorio.
- `File firstChild (File f)`: función que devuelve el primer fichero/directorio que es hijo de `f`. Devuelve `Null` si no existe.
- `File nextBrother (File f)`: función que nos devuelve el siguiente fichero/directorio al mismo nivel que `f`. Devuelve `Null` si no existe.
- `int count (File f, int key)`: función que nos devuelve el número de veces que aparece la clave `key` en el fichero `f`.

El código paralelo inicial se muestra a continuación:

```
#define MAX_LEVEL 5
omp_lock_t vlock[MAX_LEVEL];

int totalKeys = 0;

int countKeyFiles (File f, int key, int level) {
    if (isFile(f)) return(count (f, key));

    // Contar hijos recorriendo recursivamente subdirectorios
    int count = 0;
    File next = firstChild (f);
    while (next != NULL) {
        #pragma omp task shared(next) private(count) firstprivate(level)
            {
            omp_set_lock(&vlock[level]);
            count += countKeyFiles (next, key, level+1);
            omp_unset_lock(&vlock[level]);
            }
        next = nextBrother (next);
    }
    #pragma omp taskwait
    return(count);
}

void main() {
    ...
    File root = ... // inicializacion de root
    ...
    totalKeys = countKeyFiles (root, key, 0);
    ...
}
```

Se pide:

a) Completar el código del programa principal `main` para que el programa se ejecute en paralelo siguiendo el esquema propuesto.

b) Indicar si las cláusulas de datos de la directiva `task` son los adecuados para la ejecución correcta de la aplicación; realizad los cambios que sean oportunos para conseguirlo (si fuera necesario).

c) Modificar el código para tener en cuenta los *overheads* de la creación de tareas en la exploración recursiva de la estructura de datos, utilizando el mismo argumento `level` para dicho fin y la constante `MAX_LEVEL`.

Después de conseguir que el programa funcionase, nos dimos cuenta de que presentaba un problema de rendimiento, siendo incapaz de reducir el tiempo de ejecución más allá de 1 procesador. **Se pide:**

d) Modificar el código para que se permita mejorar el paralelismo que se pueda extraer en la paralelización del código anterior.

Pregunta 2 (2 puntos) Escribir una versión equivalente (en cuanto a número de *chunks* e iteraciones en cada *chunk*, no en cuanto a su asignación a procesadores ni en cuanto a su orden de ejecución) en la que SÓLO se utilicen las construcciones de OpenMP `parallel`, `single` y `task`, para cada uno de los siguientes bucles en OpenMP:

- Bucle 1:

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

- Bucle 2:

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

Nota: suponed que `p` divide de forma entera a `n`

Pregunta 3 (1 punto) Dado el siguiente código indicad si puede darse algún tipo de problema en su ejecución con 2 procesadores. En caso afirmativo, describid el potencial problema y alguna manera de solucionarlo.

```
MPI_Comm_rank (comm, &rank);
if (rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT, 1, ...);
   MPI_Recv (recvbuf, count, MPI_INT, 1, ...);
} else if (rank == 1) {
   MPI_Send (sendbuf, count, MPI_INT, 0, ...);
   MPI_Recv (recvbuf, count, MPI_INT, 0, ...);
}
```

Nota: los argumentos de las llamadas `MPI_Send` y `MPI_Recv` indican la dirección del *buffer* de envio/recepción, el número de elementos a enviar/recibir y su tipo de datos y a que proceso se envia o de que proceso se recibe, respectivamente.

Pregunta 4 (4 puntos) A continuación se presenta un código que realiza la transposición a bloques de una matriz. Los bloques son de tamaño $BS \times BS$ y la matriz de tamaño $N \times N$, donde N es múltiplo de BS.

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

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

El código de la transposición a bloques es:

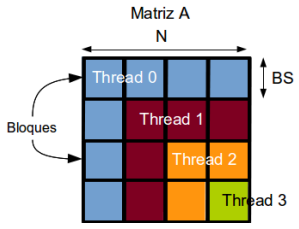
```
void tranpose(int A[N][N]) {
    int Ablock_ij[BS][BS], Ablock_ji[BS][BS];
    int i, j;

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

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

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

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

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

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

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

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



Solution

Pregunta 1 (3 puntos)

a)
```
void main() {
    File root = ... // inicializacion de root
    for (int i=0; i< MAX_LEVEL; i++) omp_init_lock(&vlock[i]);
    #pragma omp parallel
    #pragma omp single
    totalKeys = countKeyFiles (root, key, 0);
    for (int i=0; i< MAX_LEVEL; i++) omp_destroy_lock(&vlock[i]);
}
```

b) No lo son. Las directivas que se deberían especificar para el correcto funcionamiento son: `shared(count)`. Para el resto de variables (`level, next, key`) no hace falta especificar nada, el compilador las considerará firstprivate por defecto que es el comportamiento deseado.

c)
```
int countKeyFiles (File f, int key, int level) {
    if (isFile(f)) return(count(f, key));

    // Contar hijos recorriendo recursivamente subdirectorios
    int count = 0;
    File next = firstChild (f);
    if (level < MAX_LEVEL) {
        while (next != NULL) {
            #pragma omp task shared(count)
                {
                omp_set_lock(&vlock[level]);
                count += countKeyFiles (next, key, level+1);
                omp_unset_lock(&vlock[level]);
                }
            next = nextBrother(next);
            }
        #pragma omp taskwait
        }
        else {
            while (next != NULL) {
                count += countKeyFiles (next, key, level);
                next = nextBrother(next);
            }
        }
    }
    return(count);
}
```

d) Substituimos la exclusión mútua implementada con *locks*, que impide que haya paralelismo entre las multiples tareas de un mismo nivel, por un simple `atomic`:
```
    while (next != NULL) {
        #pragma omp task shared(count)
            {
            #pragma omp atomic
            count += countKeyFiles (next, key, level+1);
            }
        next = nextBrother(next);
    }
    #pragma omp taskwait
```

Pregunta 2 (2 puntos) Nota: suponemos que `p` divide de forma entera a `n`

- Bucle 1:

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

Solución:

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

- Bucle 2:

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

Solución:

```
#pragma omp parallel num_threads(p) private(i)
{
    int myid = omp_get_thread_num();
    for (int ii = myid*p; ii < n; ii += (p*p))
        for (i = ii; i < ii + p; i++)
            b[i] = foo(a[i]);
}
```

Pregunta 3 (1 punto)

Este código tiene potencialmente un problema de concurrencia: `deadlock`. Las llamadas `MPI_Send` se bloquean hasta que la contraparte ejecuta la operación correspondiente. Esto implica que tal y como está el código original, un proceso esperará al otro que haga el `MPI_Recv`, quedándose ambos bloqueados.

Una posible solución consistiría en reordenar las llamadas para que un proceso haga primero el `MPI_Send` mientras que el otro hace el `MPI_Recv`, y viceversa. Esto asegura que la contraparte ejecuta la operación correspondiente sin que se bloqueen. La solución es la siguiente:

```
MPI_Comm_rank (comm, &rank);
if (rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
```

Aunque no se explicaron con suficiente detalle en clase, otra posible solución se basaría en usar llamadas MPI no bloqueantes, como por ejemplo:

```
MPI_Comm_rank (comm, &rank);
if (rank == 0) {
   MPI_Isend(sendbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &request);
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
   MPI_Wait (&request, &status);
} else if (rank == 1) {
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
```

Pregunta 4 (4 puntos)

a) Tal como esta escrita la solución parece más natural aplicar una *input data decomposition*, en la que el thread lee su bloque y escribe en el de otro thread, el cálculo de los índices `i` y `j` sería:

```
    ...
    int myid= omp_get_thread_num();
    int i = (myid/(N/BS))*BS;
    int j = (myid%(N/BS))*BS;
    ...
```

Sin embargo seía posible también calcular dichos índices suponiendo una *output data decomposition*.

Se precisa una sincronización entre la lectura que realiza cada thread de un bloque `(i,j)` (con `read_block`) de la matriz `A` y la escritura del bloque `(i,j)` (con `write_block_back`), una vez transpuesto, que realiza otro thread sobre la misma matriz `A`. Notad que para los bloques de la diagonal no es necesario realizar la sincronización ya que es un mismo thread el que realiza la lectura y la escritura sobre el bloque `(i,i)`, pero para simplificar la sincronización y el código, se ha propuesto una solución utilizando `barrier`.

b)

```
void tranpose(int A[N][N]) {

 #pragma omp parallel num_threads(N/BS)
 {
    int Ablock_ij[BS][BS];
    int Ablock_ji[BS][BS];
    int i, j;
    int myid= omp_get_thread_num();
    i = myid * BS;
    read_block(A,i,i,Ablock_ij);
    transpose_block(Ablock_ij);
    write_block_back(Ablock_ij,A,i,i);
    for (j=i+BS; j<N; j+=BS){
        read_block(A,i,j,Ablock_ij);
        read_block(A,j,i,Ablock_ji);
        transpose_block(Ablock_ij);
        transpose_block(Ablock_ji);
        write_block_back(Ablock_ij,A,j,i);
        write_block_back(Ablock_ji,A,i,j);
    }
  }
}
```

Part III

Final Exams

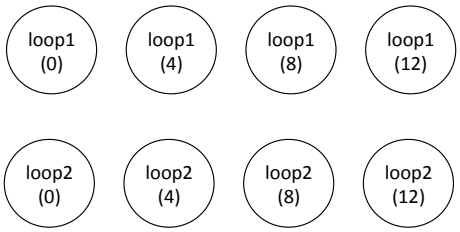
Examen Final de PAR – Curso 2012/13-Q2
7 de Junio de 2013

Problema 1 (4 puntos, repartidos 1 punto / 3 puntos). Dados los dos bucles siguientes:

```
#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++)              // Suponemos que N es multiplo
        for (int j = 0; j < N; j++)              // de BS
            b[i][j] = foo(a[i][j]);
    tareador_end_task();
}
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 para
                                                     // asegurar acceso dentro del
                                                     // rango de la matriz 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();
}
tareador_OFF();
}
```

Se pide:

1. Completad el grafo de tareas que se generaría con *Tareador*, sabiendo que las funciones `foo` y `goo` sólo operan con los datos que se les pasa y no actualizan ninguna variable global.



2. Escribid las expresiones que determinan el tiempo de ejecución T_p para 4 procesadores (T_4) (desglosado en tiempo de cálculo $T_{4(calc)}$ y tiempo de comunicación $T_{4(comm)}$) para cada una de las dos asignaciones de tareas a procesadores indicadas en la tabla siguiente.

   Para ello suponed: 1) arquitectura de memoria distribuida con 4 procesadores; 2) distibución inicial de las matrices `a`, `b` y `c` por bloques de filas (N/BS filas consecutivas por procesador, recorridas en el código anterior por la variable `i`); 3) después de la ejecución del segundo bucle NO es necesario dejar los datos en su distribución inicial; 4) modelo de compartición de datos basado en 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 siendo m el tamaño del mensaje; y 5) el tiempo de ejecución de una iteración del cuerpo del bucle más interno es t_c. Las expresiones sólo deberán quedar en función de t_s, t_w y t_c.

Tarea	Asignación 1	Asignación 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

Problema 2 (4 puntos: 2/1/1). Dado el siguiente código secuencial en C que encuentra en un vector `DB` la primera posición en la que aparece una determinada clave `key`:

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

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

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

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

Se pide:

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

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

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

Problema 3 (2 puntos: 0.5/0.5/1). Dado el siguiente código secuencial que calcula el histograma de los valores (en el rango `0..RANGE_OF_VALUES-1`) de los elementos de la variable `input`:

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

Se propone la siguiente solución paralela para memoria compartida:

```
#define CACHE_LINE_SIZE 128

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

element histogram[RANGE_OF_VALUES];

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

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

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

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

Se pide:

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

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

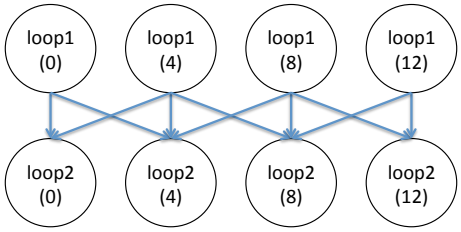
3. Rellenad la tabla siguiente con las transacciones y/o eventos del protocolo de coherencia, si los hay, que se producen entre los nodos NUMA durante la ejecución del bucle paralelo que sean provocados por el acceso a la variable `histogram`. Para ello considerad las siguientes condiciones: 1) `RANGE_OF_VALUES=64`;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

Tiempo	Iteración bucle i	Transacción y/o evento (indicar tipo y nodos origen y destino)
0	0	
1	2	
2	1	
3	3	

Solution

Problema 1 (4 puntos, repartidos 1 punto / 3 puntos).

1. The resulting task graph is:



2. Expressions are:

Tarea	Asignación 1	Asignación 2
T_4: cálculo	$2 \times 64 \times t_c$	$(64 + 256) \times t_c$
T_4: comunicación	$2 \times (t_s + 16 \times t_w)$	$3 \times (t_s + 64 \times t_w)$

Problema 2 (4 puntos: 2/1/1)

1. ```
   #pragma omp parallel
       {
       unsigned long i, num_elems, lower;
       int myid = omp_get_thread_num();
       int howmany = omp_get_num_threads();
       lower = myid * (DBsize / howmany) +
               (myid < (DBsize % howmany) ? myid : DBsize % howmany);
       num_elems = (DBsize / howmany) + (myid < (DBsize % howmany));
       for (i = lower; (i < (lower + num_elems)) && (i < position); i++) {
           #pragma omp critical
           if ((DB[i] == key) && (i<position)) position = i;
       }
   ```

2. ```
    #pragma omp parallel
        {
        unsigned long i, num_elems, lower;
        int myid = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        lower = myid * (DBsize / howmany) +
                (myid < (DBsize % howmany) ? myid : DBsize % howmany);
        num_elems = (DBsize / howmany) + (myid < (DBsize % howmany));
        for (i = lower; (i < (lower + num_elems)) && (i < position); i++) {
            if (DB[i] == key) {
                #pragma omp critical
                if (i<position) position = i;
                }
            #pragma omp flush
        }
   ```

   El `pragma omp flush` es necesario para garantizar la consistencia de la variable `position`.

3. Descomposición CYCLIC, ya que de esta manera todos los threads avanzan desde el principio del vector `DB`, llegando a la primera posición que contiene `key` antes.. La implementación solo cambiará en el cálculo de los limites y recorrido del bucle:

```
#pragma omp parallel
    {
    unsigned long i;
    int myid = omp_get_thread_num();
    int nt= omp_get_num_threads();
    for (i = myid; (i < DBsize) && (i < position); i+=nt) {
        if (DB[i] == key) {
            #pragma omp critical
            if (i<position) position = i;
        }
        #pragma omp flush
    }
```

Problema 3 (2 puntos: 0.5/0.5/1)

1. `char tmp[CACHE_LINE_SIZE-sizeof(int)-sizeof(omp_lock_t)];`

   Esta modificación ayuda a eliminar el `false sharing` accediendo a histogram

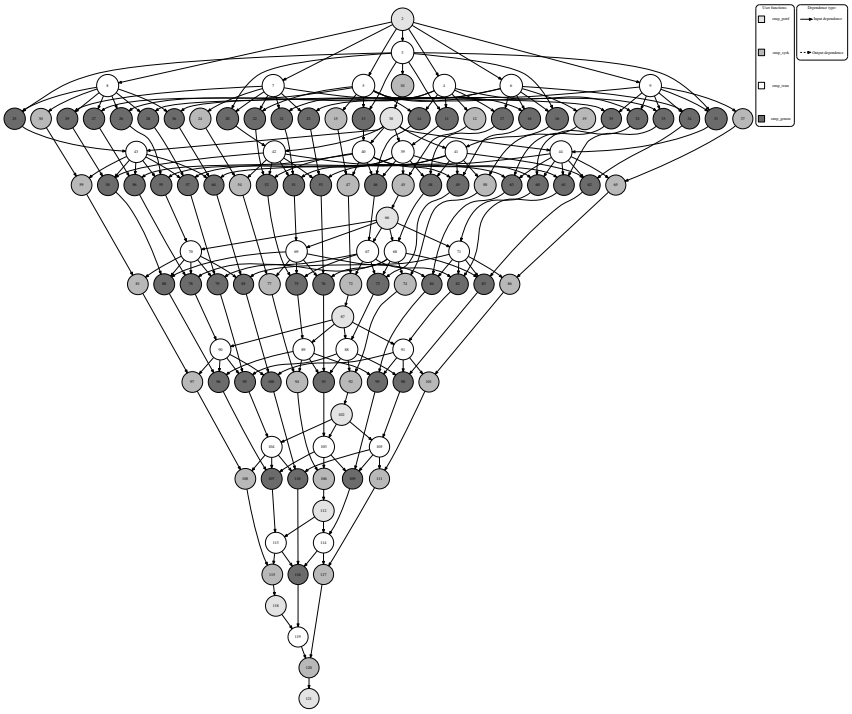
2. Directorio de la memoria del Numa Node 0 debido al acceso de inicialización del histogram por parte del thread master

3. Transactions and/or events are:

Tiempo	Iteración bucle i	Transacción y/o evento (indicar tipo y nodos origen y destino)
0	0	No hay eventos/transacciones entre NUMA nodes porque los datos están en mismo Numa Node 0
1	2	RdXReq del Numa Node 1 al Numa Node 0 (home y Owner). Numa Node 0 devuelve el Dato (suponemos que optimiza el protocolo para devolver Dato. Sino, entonces se necesitaria hacer: Numa Node 0 envia Owner Numa Node 1. Numa Node 1 envia intervencion Numa Node 0 (Owner). Numa Node 0 envia Dato a Numa Node 1 y WB_D es local, no se envia. Numa Node 1 se ha convertido en el Owner
2	1	RdXReq del Numa Node 0 al Numa Node 0 (home) , no hay envio de ningun evento. Numa Node 0 no hace falta que envia Owner al Numa Node 0. Numa Node 0 envia intervencion Numa Node 1 (Owner). Numa Node 1 envia Dato a Numa Node 0 y WB_D al Numa Node 0. Numa Node 0 se ha convertido en el Owner.
3	3	RdXReq del Numa Node 1 al Numa Node 0 (home y Owner). Numa Node 0 devuelve el Dato (suponemos que optimiza el protocolo para devolver Dato. Sino, entonces se necesitaria hacer: lo mismo que para el primer caso.

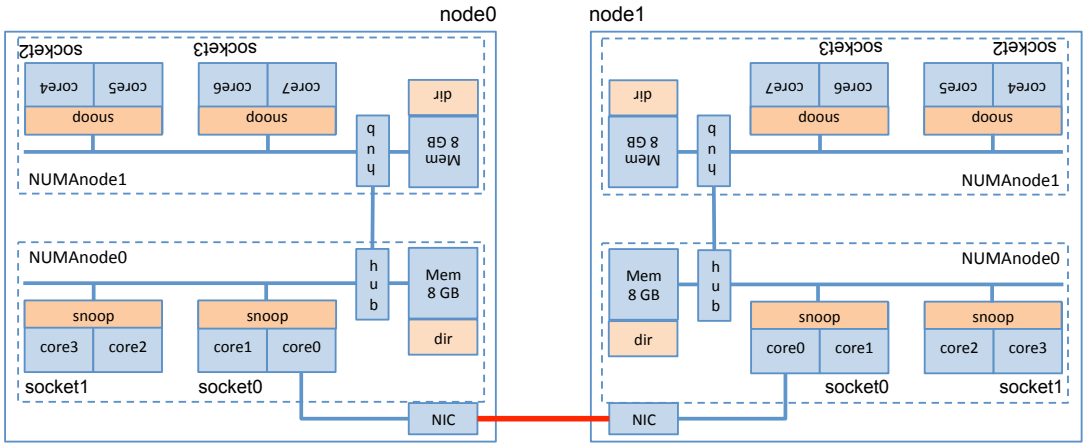
Examen final de PAR – Curso 2013/14-Q1
8 de Enero de 2014

Problema 1 (1 punto): Dado el siguiente grafo de tareas obtenido a partir de la instrumentación *Tareador* de un código secuencial:



Sabiendo que la ejecución de cada nodo del grafo tarda 1 unidad de tiempo **se pide calcular** el T_1, T_∞ y paralelismo de este grafo.

Problema 2 (1 punto): Dada la arquitectura de sistema multiprocesador mostrada en la figura siguiente, ofreciendo memoria compartida coherente dentro de un nodo y memoria distribuida entre nodos:



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.
nodeZ: Nodo con 2 NUMAnodes. Cada nodo incluye un NIC (Network Interface Card, Ethernet).

Se pide indicar si cada una de las siguientes afirmaciones es cierta o falsa (cada cuestión bien contestada suma 0.2 puntos y mal contestada resta 0.07 puntos):

a) Dentro de un *NUMAnode*, el protocolo de coherencia garantiza que para una misma dirección física de memoria no puedan existir cópias con valores distintos en las memorias cache de cores distintos.

b) Dentro de un *NUMAnode*, el false sharing se produce cuando existen dos o más copias modificadas de una misma línea de memoria con valores distintos en las caches de los cores.

c) Una dirección física de memoria sólo puede estar asignada a (almacenada en) un *NUMAnode*, aunque múltiples cópias de la misma puedan estar almacenadas temporalmente en la memoria cache de varios *NUMAnode* dentro de un *node*.

d) La estructura de directorio que hay en un *NUMAnode* proporciona información que permite localizar donde se encuentra la copia del dato que se puede acceder con el menor tiempo de acceso.

e) Los controladores de red (NIC) en cada *node* ofrecen la posibilidad, a modelos de programación tipo *MPI*, de implementar distintas estrategias de intercambio de datos, punto a punto y colectivas, entre las tareas de una aplicación paralela.

Problema 3 (3 puntos): Dado el siguiente fragmento de programa:

```
#define MAX_ITER 10000
#define NUM_ELEM 8192

// vector data alineado al inicio de linea de cache
double data[NUM_ELEM];
double compute( int elem, ... );

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

siendo `compute` una función con las dos características siguientes: 1) no provoca dependencias entre iteraciones del bucle `elem` y 2) su tiempo de ejecución varia según el valor de `elem`. Con el objetivo de paliar el desbalanceo de carga (*load balance*) que se puede provocar en la ejecución paralela del bucle `elem` se define un vector de listas que almacene para cada thread la lista de iteraciones que debe de ejecutar:

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

iteration * decomposition[NUM_THREADS];

// Inserta el elem en la cabecera de la lista
void insert(int elem, iteration * list);
// Inicializa el vector de listas con cada elemento apuntando a NULL
void initialize(int num_list, iteration * list);
```

La estrategia de task decomposition que se quiere aplicar sigue un esquema *Inspector/Ejecutor*: en una primera fase se ejecuta UNA iteración de *Inspección* en la que se determina la asignación de iteraciones a *threads* y una segunda fase de *Ejecución* en la que se ejecutan el resto de iteraciones (`MAX_ITER - 1`) siguiendo la descomposición determinada en la fase de *Inspección*, tal como se muestra en el siguiente código incompleto:

```
initialize(NUM_THREADS, decomposition);

// FASE DE INSPECCION
#pragma omp parallel for schedule( /* apartado a) */ )
for (int elem = 0; elem < NUM_ELEM; elem++) {
    /* apartado b) */
    data[elem] += compute (elem, ... );
}

// FASE DE EJECUCION
for (timestep = 1; timestep < MAX_ITER; timestep++) {
    /* apartado c) */
}
```

Se pide:

a) Determina la estrategia de reparto de iteraciones para el bucle de inspección que garantice un reparto razonable de carga (*load balance*) y evite la aparición de *false sharing* en el acceso a la estructura `data` (sin modificar la definición de la estructura `data`). Nota: suponed que la constante `CACHE_LINE_SIZE` define el tamaño de una línea de cache en bytes y recordad que el operador `sizeof(data_type)` devuelve el numero de bytes que ocupa un deteminado `data_type`.

b) Completar el código de la fase *Inspector* utilizando la estructura `decomposition` y funciones asociadas para realizar su cometido.

c) Completar el código de la fase *Ejecutor* de manera que, utilizando la información almacenada en la estructura de datos `decomposition`, siga la task decomposition determinada en la fase *Inspector*.

d) Reemplazar el código del *Inspector* con un código equivalente que pueda paralelizarse con `#pragma omp task` usando una estrategia recursiva *divide-and-conquer Tree*, programando el caso base de la recursividad para garantizar las mismas características especificadas en el apartado a).

Problema 4 (3 puntos): La función `matmul` mostrada a continuación realiza la paralelización con OpenMP de la multiplicación de matrices A y B, por submatrices:

```
#define N  128
#define BS 32

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

void matmult(float A[N][N], float B[N][N], float C[N][N]) {
#pragma omp parallel num_threads(N/BS)
#pragma omp for collapse(2) /*... Apartado c) ...*/
  for (int i=0;i<N;i+=BS)
    for (int j=0 ;j<N; j+=BS)
      for (int k=0 ;k<N; k+=BS)
         matmul_submatriz(N, BS, &A[i][k], &B[k][j], &C[i][j]);
}
```

siendo la función `matmul_submatriz` la encargada de multiplicar una submatriz de BSxBS elementos de A por otra de B y actualiza el resultado en la submatriz de BSxBS elementos de C. Esta realización se basa en que la multiplicación de $C = A \times B$ puede verse como el cálculo de sus submatrices $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$ tal como se describe a continuación:

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

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Se pide: (ver nota al pie de página[2])

a) Dibujar las matrices A, B y C, divididas en sub-bloques (submatrices), donde se indique qué *thread*/s accede/n (lee/n o escribe/n) a cada bloque. También deberéis indicar el tipo de estrategia *Data Decomposition* seguido en A, B y C, debajo del dibujo de cada matriz.

b) Escribir un código alternativo al propuesto que, sin utilizar `#pragma omp for` ni `#pragma omp parallel for` aplique la misma estrategia de *Data Decomponsition*.

c) Suponiendo ahora que se añade `schedule(static,1)` al `#pragma omp for` del código original, repetir los dos apartados anteriores para la nueva propuesta de paralelización.

Problema 5 (2 puntos): Disponemos de una rutina que implementa en paralelo el algoritmo de Floyd para calcular el camino más corto entre todos los pares de nodos de un grafo almacenado como una matriz de adyacencia (matriz `a` de `NxN` elementos, siendo `N` el número de nodos del grafo). La paralelización se ha realizado usando MPI y la matriz `a` se ha distribuido usando una descomposición por bloques de filas.

```
// P: numero de procesadores
// N: dimension de la matriz
void compute_shortest_paths (int id, int **a) {
  // id: identificador del proceso MPI que realiza la llamada (rank)
  // matriz de adyacencia a de NxN elementos

  int owner; // proceso propietario de la fila de la que hay que hacer broadcast
  int * tmp = malloc (N * sizeof(int)); // fila de la que se hace broadcast

  for (int k = 0; k < N; k++) {
    owner_prepares_row (a, k, tmp, &owner);
A:  MPI_Bcast (tmp, N, MPI_INT, owner, MPI_COMM_WORLD);
    for (int i = 0; i < BLOCK_SIZE(id); i++)
      for (int j = 0; j < N; j++)
B:      a[i][j] = MIN(a[i][j], a[i][k] + tmp[j]);
  }
  free (tmp);
}
```

La función `compute_shortest_paths` es ejecutada por cada uno de los procesos MPI. En cada iteración del bucle externo cada proceso requiere la fila k-ésima, la cual no está necesáriamente almacenada en la memoria de ese proceso pues la matriz `a` está distribuida. Si la rutina auxiliar `owner_prepares_row` es ejecutada por el proceso propietario de la fila k, entonces éste copiará dicha fila k en el buffer apuntado por `tmp`. En cualquier caso, esta rutina retorna en `owner` el identificador del proceso propietario de fila k. La fila copiada en el buffer es enviada a continuación a todos los demás procesos mediante la colectiva de broadcast `MPI_Bcast`. La función `BLOCK_SIZE` retorna el número de filas asignadas al proceso `id`.

Sabiendo que: 1) N es múltiplo de P; 2) únicamente las sentencias etiquetadas como A: y B: contribuyen al tiempo de ejecución; 3) el tiempo necesario para enviar un mensaje de m elementos se puede modelar como $t_s + m \times t_w$, siendo m el número de elementos a enviar; 4) la sentencia etiquetada como B: se ejecuta en tiempo t_c; y 5) no se solapan las comunicaciones y los cálculos.

Se pide: obtener la expresión que modela el tiempo de ejecución del código anterior (expresado en función de N, P, t_c, t_s y t_w) suponiendo que el coste del broadcast es lineal respecto al número de procesadores.

[2]Sacado del manual de OpenMP: "If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space."

Solution

Problema 1 (1 punto)
$T_1 = 120$, que corresponde al número de nodos.
$T_\infty = 22$, que corresponde al camino crítico.
Paralelismo de este grafo = $T_1/T_\infty = 120/22$

Problema 2 (1 punto)

a) **(Cierta)**

b) **(Falsa)**

c) **(Cierta)**

d) **(Falsa)**

e) **(Cierta)**

Problema 3 (3 puntos)

a) Dado que el tiempo de ejecución del bucle varia según el valor de `elem`, el reparto de iteraciones adecuado es un `dynamic`. Para evitar *false sharing* en el acceso a `data` sin utilizar *padding* la mejor solución es utilizar un tamaño de *chunk* que asegure que los elementos en una misma línea de cache son escritos por un mismo procesador. Así la planificación sera `schedule(dynamic, CACHE_LINE_SIZE / sizeof(double))`.

b) Cada thread deberá insertar en su lista las iteraciones que la planificación dinámica le asigna:

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

c) Cada *thread* deberá realizar el recorrido de su lista con el objetivo de obtener las iteraciones que ha ejecutado en la fase de inspección:

```
// FASE DE EJECUCION
#pragma omp parallel private(timestep)
for (timestep = 1; timestep < MAX_ITER; timestep++) {
    iteration * iter = decomposition[omp_get_thread_num()];
    for ( ; iter != NULL; iter = iter.next) {
        data[iter.elem] += compute (iter.elem, ... );
    }
}
```

d)
```
void inspect_rec(int iter, int elements) {
    // base case when number of elements fits in a cache line
    if (elements <= CACHE_LINE_SIZE / sizeof(double)) {
        for (int elem = iter; iter < elements; elem++) {
            insert(elem, decomposition[omp_get_thread_num()]);
            data[elem] += compute (elem, ... );
        }
        return;
    }
    // divide and conquer
    int elemdiv2 = elements / 2;
    #pragma omp task   // first private by default
```

```
    inspect_rec(iter, elemdiv2);
    #pragma omp task
    inspect_rec(iter+elemdiv2, elements-elemdiv2);
}

// FASE DE INSPECCION
#pragma omp parallel
#pragma omp single
inspect_rec (0, NUM_ELEM);
```

Problema 4 (3 puntos):

a) Matrices A y C: Block Data Decomposition por filas. Para B no hay "Data decomposition". Todos los threads leen de todas las submatrices de B y por consiguiente, si fuera necesario, se replicaría toda la matriz B (por ejemplo, en una máquina con memoria distribuida).

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

Table 1: Matrices A y C: Block Data Decomposition por filas

b)
```
#define N  128
#define BS 32

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

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

c) Matrices B y C: Block Data Decomposition por columnas. Para A no hay "Data decomposition". Todos los threads leen de todas las submatrices de A y por consiguiente, si fuera necesario, se replicaría toda la matriz A (por ejemplo, en una máquina con memoria distribuida).

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

Table 2: Matrices B y C: Block Data Decomposition por columnas

Una solución equivalente:

```
#define N  128
#define BS 32

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

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

Problema 5 (2 puntos):

Consideramos `BLOCK_SIZE(id)` $\approx \frac{N}{P}$

Y si P es grande: $P - 1 \approx P$

Entonces

$$T_P \approx \underbrace{\frac{N^3}{P} \times t_c}_{T_{Comp}} + \underbrace{N \times (P-1) \times (t_s + N \times t_w)}_{T_{Comm}} \approx \underbrace{\frac{N^3}{P} \times t_c}_{T_{Comp}} + \underbrace{N \times P \times (t_s + N \times t_w)}_{T_{Comm}}$$

Examen final de PAR – Curso 2013/14-Q2
17 de Junio de 2014

Dado el siguiente programa OpenMP que debiera retornar el número de veces que aparece un determinado `elemento` en un vector de listas utilizando una estrategia de paralelización *master–worker*, en la que uno de los *threads* (denominado *master*) recorre el vector de listas y envia trabajo al resto de threads (denominados *workers*) que se encargan de realizar la búsqueda en cada una de las listas que le indica el *master*:

```
#define NUM_ELEMS 10000

typedef struct {
    int elem;
    list * next;
} list; // the basic component of a list

list * data[NUM_ELEMS]; // vector of lists, with varying number of elements
int element; // value for which the program counts number of times it appears in data
int count = 0;

// variable to control the interaction between master and workers
int thread_busy[MAX_THREADS]; // all elements initialized to -1

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

void main() {
    init_data(data); // initialization of data structure
    readln("Enter element to find: %d\n", &element); // read element to be found
    #pragma omp parallel
        {
        int me = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        howmany = howmany - 1; // the master does not participate in the search
        int count = 0, i = 0;
        if (me == howmany) {   // last thread is the master
            for (int entry = 0; entry < NUM_ELEMS; entry++) {
                while (thread_busy[i]>=0) i = (i + 1) % howmany; // next thread available
                #pragma omp critical
                thread_busy[i] = entry;
            }
        } else {    // I am a worker
            while(true) {
                while (thread_busy[me] == -1);
                int tmp = thread_busy[me];
                count  += list_search(data[tmp], element);
                #pragma omp critical
                thread_busy[me] = -1;
                }
        }
}
if (count == 0) printf("Element %d not found in data structure\n", element);
else printf("%d instances of element %d found in data structure\n", count, element);
}
```

Nota: los problemas 1 y 2 se basan en el código anterior pero se pueden realizar de forma independiente.

Problema 1 (4 puntos): Después de compilar el programa anterior y ejecutarlo con más de 1 procesador, observamos que el programa "no funciona", en concreto, que no finaliza su ejecución. Se pide:

a) Identificar con que variable y en que línea(s) de código se está provocando un problema potencial de *memory consistency*. Realizar los cambios pertinentes para que dicho problema desaparezca.

b) Identificar con que variable y en que línea(s) de código se esta provocando un problema de *false sharing*. Realizar los cambios pertinentes para que dicho problema desaparezca.

c) Suponiendo arreglados los dos problemas anteriores, vemos que el `parallel` no finaliza. Realizar los cambios pertinentes para que la región paralela finalice.

d) Finalmente nos damos cuenta de que el programa no imprime un valor correcto para la variable `count`. Realizar los cambios pertinentes para que el cálculo sea correcto.

Problema 2 (1 punto): Escribir una nueva versión en la que no se utilice el paradigma *master–worker*, es decir, en la que todos los *threads* se dediquen a realizar búsquedas en distintas listas y no se provoquen esperas innecesarias por parte de ningun thread. La estrategia se basará en una *iterative task decomposition*. Podeis utilizar para ello cualquier `pragma` de OpenMP para realizar esta nueva versión y suponer que la función `list_search` está implementada.

Problema 3 (3 puntos):
Para los siguientes fragmentos de códigos `OpenMP` extraídos de aplicaciones distintas:

```
Codigo 1
#pragma omp parallel for num_threads(p)
for (int i=0; i<N; i++)
   for (int j=0; j<N; j++)
      A[i][j] = B[j][i]


Codigo 2
#pragma omp parallel num_threads(p)
{
  int myid = omp_get_thread_num();
  for (int i=myid*N/p; i<(myid+1)*N/p; i++)
    for (int j=0; j<N; j++)
         A[i][j] = B[rand()%N][rand()%N];
         // rand() devuelve un numero aleatorio positivo
}

Codigo 3
#pragma omp parallel num_threads(nt)
{
  int myid    = omp_get_thread_num();
  int sqrt_nt = sqrt(nt);
  int i_start = (sqrt_nt-1-(myid/sqrt_nt))*(N/sqrt_nt);
  int j_start = (myid%sqrt_nt)*(N/sqrt_nt);
  for (int i=i_start; i<i_start+N/sqrt_nt; i++)
    for (int j=j_start; j<j_start+N/sqrt_nt; j++)
        A[i][j]= B[i][j];
}

Codigo 4
#pragma omp parallel num_threads(p)
{
  int myid = omp_get_thread_num();
  for (int i=myid; i<N; i+=p)
    for (int j=0; j<N; j++)
      A[i][j] = B[j][i]
}
```

Suponed que queremos ejecutar dichos códigos en una arquitectura de memoria compartida NUMA, en la que un procesador experimenta una diferencia importante entre el tiempo de acceso a su memoria local y el tiempo de acceso a la memoria de un procesador remoto. También suponed que cada `thread` se ejecutará en un `NUMA node` diferente y que hay tantos `NUMA nodes` como `threads`.
Se pide: proponer la *data decomposition* para las matrices `A` y `B` que seria más adecuada para cada uno de los cuatros códigos anteriores, indicando claramente los datos de `A` y de `B` que le deberían ser locales a cada procesador o si sería necesaria su replicación, con el objetivo de minimizar el número de accesos remotos que se realizan.

Problema 4 (2 puntos):
Tenemos una función que implementa el *core* de una iteración de un algoritmo de tipo *Gauss-Seidel* (similar al código que has usado en una de las prácticas de laboratorio). La función es ejectuada simultáneamente por `P` procesadores, cada uno de ellos con su propio identificador `myid`, y asume que la matriz `u`, originalmente de `N` por `N` elementos, ha sido distribuida usando una descomposición a bloques por filas. Por tanto, cada proceso trabaja can una submatriz, que en la función se visualizan como `nrows` por `ncols` elementos.

```
#define B ...  /* Block size: Number of columms in the block. */

double gauss_seidel_mpi (double *u, unsigned int nrows, unsigned int ncols,
                         unsigned int myid, unsigned int P) {
    double unew, diff, sum=0.0;
    int nby = ncols/B;

    for (int jj=0; jj<nby; jj++) {
        // All (except first) should receive last row in block before start
        if (myid != 0) MPI_Recv(&u[1+jj*B], B, MPI_DOUBLE, myid-1, 0, ...);

        for (int i=1; i<=nrows-2; i++)
            for (int j=1+jj*B; j<=min((jj+1)*B, ncols-2); j++) {
                unew= 0.25 * (
                      u[     i*ncols    + (j-1) ]+  // left
                      u[     i*ncols    + (j+1) ]+  // right
                      u[ (i-1)*ncols    + j     ]+  // top
                      u[ (i+1)*ncols    + j     ]); // bottom
                diff = unew - u[i*ncols+ j];
                sum += diff * diff;
                u[i*ncols+j]=unew;
            }
        // All (except last) should send last row in block after calculation)
        if (myid != P-1)
            MPI_send(&u[(nrows-2)*ncols + 1+jj*B], B, MPI_DOUBLE, myid+1, 0, ...);
    }
    return sum;
}

int main()
{
  ...
  rows = N/P;
  while (residual > THRESHOLD) {
    ...
    residual = gauss_seidel_mpi(u, rows, N, myid, P);
    ...
  }
  ...
}
```

Nota: los argumentos de las llamadas `MPI_Send` y `MPI_Recv` indican la dirección del *buffer* de envio/recepción, el número de elementos a enviar/recibir y su tipo de datos, y a qué proceso se envia o de qué proceso se recibe, respectivamente.
Se pide: que escribáis un modelo de rendimiento que incluya el tiempo de cálculo y el tiempo de comunicaciones para la ejecucion de la función `gaus_seidel_mpi` en todos los procesos. Considerad que la ejecución de una iteración del bucle más interno supone un tiempo t_c. Como es habitual, considerad que las comunicaciones se expresan como $t_s + m \times t_w$, siendo m el tamaño del mensaje en elementos, t_s el tiempo de start-up, y t_w el tiempo por elemento. Puedes suponer que el tamaño de bloque `B` usado en el código divide de forma exacta al número de columnas `ncols`, y que `P` divide de forma exacta a `N`.

Solution

Problema 1 (4 puntos):

a) Se provoca un problema potencial de *memory consistency* en los accesos a la variable `thread_busy` que se realizan en los dos bucles de espera, tanto en el *master*:

```
while (thread_busy[i]>=0) i = (i + 1) % howmany;
```

como en los *workers*:

```
while (thread_busy[me] == -1);
```

Las actualizaciones se realizan dentro de `critical` que garantizan la consistencia, pero para las lecturas se debe de incorporar `flush`, tal como se muestra a continuación para uno de los dos bucles:

```
while (thread_busy[me] == -1)
   #pragma omp flush
   ;
```

b) De nuevo los accesos a la variable `thread_busy` provoca false sharing dado que distintos procesadores leen/escriben posiciones consecutivas dentro de una misma línea de memoria *cache*. Para evitarlo se puede utilizar *padding* en la declaración del vector de manera que cada thread acceda a una línea distinta:

```
int thread_busy[MAX_THREADS][CACHE_LINE_SIZE / sizeof(int)];
```

y posteriormente realizar los accesos accediendo a una columna concreta, por ejemplo:

```
thread_busy[me][0] = -1;
```

c) El problema esta en el bucle `while(true)` que tenemos en los *workers*. Una posibilidad es que el *master* indique a los *workers* cuando ha finalizado el reparto de trabajo. Otra podría ser que los propios *workers* sepan cuando no hay mas trabajo a recibir (variable compartida incrementada de forma atómica) y salgan del bucle `while`. Por ejemplo:

```
while(done < NUM_ELEMS) {
    ...
    #pragma omp atomic
    done++;
    while ((thread_busy[me] == -1) && (done < NUM_ELEMS))
        #pragma omp flush
        ;
}
```

Observad que es necesario realizar la comprobación en los dos bucles `while`.

d) El problema está en que la variable `count` está declarada como local dentro del `parallel`. Tiene declararse global e indicar que está afectada por una operación de reducción:

```
int count = 0;
#pragma omp parallel reduction(+:count)
    {
    ...
    count  += list_search(data[tmp], element);
    ...
    }
```

Problema 2 (1 punto):

```
#pragma imp parallel for schedule(dynamic, 1) reduction(+: count)
    for (entry = 0; entry < NUM_ELEMS; entry++)
        count  += list_search(data[entry], element);
```

realizando la reducción sobre la variable `count` y repartiendo las iteraciones de forma dinámica con un tamaño de *chunk* de 1 para hacerlo equivalente a la solución inicial basada en *master–worker*.

Problema 3 (3 puntos):

- Código 1: Geometric Block Data Decomposition por filas de A, y por columnas de B.
- Código 2: Geometric Block Data Decomposition por filas de A. B replicada.
- Código 3: Geometric Block Data Decomposition en las dos dimensiones tanto de A como de B. El bloque que se le asigna a cada thread es una submatriz de $(N/sqrt\_nt) \times (N/sqrt\_nt)$ elementos. El mapeo de estos bloques a threads es empezando por la submatriz inferior izquierda, de izquierda a derecha, y de abajo a arriba. A continuación se muestra un ejemplo de mapeo de submatrices para el caso de 9 threads. Cada número indica el thread asignado a una submatriz de la matriz A y B.

  ```
  6 7 8
  3 4 5
  0 1 2
  ```

- Código 4: Geometric Cyclic Data Decomposition por filas de A, y por columnas de B.

Problema 4 (2 puntos):
Supondremos que la dimensión del problema es grande y $\frac{N}{P} - 2 \approx \frac{N}{P}$. El modelo de tiempo de ejecución con P procesadores es:

$$T_P \approx \underbrace{(\frac{N}{B} + P - 1) \times \frac{N}{P} \times B \times t_c}_{T_{Comp}} + \underbrace{(\frac{N}{B} + P - 2) \times (t_s + B \times t_w)}_{T_{Comm}}$$