Arquitectura de Computadores. Curso 2015/2016. Benchmark de Problemas

1. La simulación de una configuración de diseño de una turbina en un procesador a 2 GHz tarda 100 segundos.
(a) Indique cuántas configuraciones puede simular en 150.000 segundos si dispone de un supercomputador con 1000 procesadores iguales al indicado, teniendo en cuenta que el tiempo de overhead correspondiente a la creación/finalización de los procesos, la comunicación de datos y parámetros a los procesadores, etc. es igual a 50p segundos, siendo p el número de procesadores que se utiliza.

Tiempo de overhead para 1000 procesadores: T_o=50*1000=50.000 segundos
Tiempo en el que trabajan los 1000 procesadores en paralelo (tiempo total menos el tiempo de overhead): 150.000-50.000=100.000 segundos.

En esos 100.000 segundos, un procesador puede simular:
(100.000 s)/(100(s/config.)= 1000 conf.

Como hay 1.000 procesadores, en total se simularán
(1000 conf./proces.)*(1000 proces.)=**1.000.000 configuraciones**

(b) Teniendo en cuenta que, lógicamente, el tiempo de overhead es 0 segundos si se utiliza un único procesador para simular las configuraciones ¿Qué ganancia de velocidad se obtiene con los 1000 procesadores en la simulación de las configuraciones del apartado anterior?.

Un procesador tardará en simular todas las configuraciones:
1.000.000 config.*100 (s/config.)=100.000.000 s. = $T_{secuencial}$

La ganancia de velocidad será el cociente de $T_{secuencial}$ y el tiempo del programa paralelo (incluyendo el overhead, lógicamente):

S= 100.000.000/150.000 = 1000*2/3 = 666.67

2. Un multiprocesador NUMA con cuatro nodos (N0,N1,N2,N3) ejecuta el código siguiente, para n=8 utilizando dos hebras (nthread=2) cuyos índices son ithread=0 e ithread=1, y se ejecutan, respectivamente, en el nodo N0 y en el nodo N1:

```
for (i=ithread; i<n; i=i+nthread) sump=sump+x[i];
fetch&add(sum,sump);
```

El array compartido x[i] (i=0,..,n-1) está constituido por números de 64 bits y se encuentra almacenado en posiciones de memoria consecutivas a partir de una posición de la memoria principal local del nodo N2 del multiprocesador, que coincide con el comienzo de un marco de línea de caché. A continuación del array se encuentra almacenada la variable compartida sum, también de 64 bits. La variable sump es local y puede considerar que se almacena en un registro del procesador que ejecuta la hebra.

Teniendo en cuenta que las líneas de caché son de 32 Bytes:

(a) ¿Cuántos marcos de línea de cache ocupan en la memoria principal local de N2 los datos compartidos del programa (el array x[] y la variable sum)?. Denótelos como L0, L1,....

(b) Indique los estados por los que pasan las líneas correspondientes en las caches de los nodos N0 y N1 al ejecutar el programa teniendo en cuenta que implementan un protocolo MSI para mantener la coherencia de cache. Si hay varios alternativas posibles en el orden en que las líneas pasan a caché, considere solo una de las alternativas posibles.

NOTAS:
- *Las caches están inicialmente vacías*
- *Suponga que el mapeo de memoria cache no asigna la misma línea de cache a marcos de línea distintos, de forma que no hay que reemplazar un bloque para introducir otro distinto en la cache.*

(a) Como cada dato tiene 64 bits = 8 Bytes, y el marco de línea (= tamaño que la línea) tiene 32 bytes, en cada marco (= en cada línea) caben 32 Bytes/(8 Bytes/dato)=4 datos

Por lo tanto hacen falta 2 marcos de línea (= líneas) para el array x[] y la variable compartida estará en el siguiente marco de línea. En total, **el programa utiliza tres marcos de bloque (= líneas) L0, L1, L2**

(b) Las hebras thread=0 y thread=1 accederán en paralelo a L0 y L1 para leer los datos del array x[] que necesitan. Como se accede al array **solo para lectura**, esos marcos de bloque estarán en las caches del nodo N0 y en la cache del nodo N1 en el estado **S**

Estado (L0,C0)=I → Estado (L0,C0)=S
Estado (L1,C0)=I → Estado (L1,C0)=S
Estado (L0,C1)=I → Estado (L0,C1)=S
Estado (L1,C1)=I → Estado (L1,C1)=S

En cuanto a L2, como las hebras acceden a través de una instrucción fetch_and_add() que implementa una sección crítica, accederá primero N0 y después N1 o N1 y después N0, y el nodo que acceda realizará la lectura de sum y luego la escritura (sin que se intercalen accesos del otro nodo). Suponiendo que acceden en el orden N0, N1 tendremos:

Estado(L2,C0)=I; Estado(L2,C1)=I
Estado(L2,C0)=S; Estado(L2,C1)=I (R(L2,N0))
Estado(L2,C0)=M; Estado(L2,C1)=I (W(L2,N0))
Estado(L2,C0)=S; Estado(L2,C1)=S (R(L2,N1)
(se actualiza L2 desde C0 antes de cargar L2 en C1)
Estado(L2,C0)=I; Estado(L2,C1)=M (W(L2,N1)

También estaría bien si se considera que, en realidad, el protocolo de coherencia se podría implementar de forma que la instrucción fetch_and_add se considera solo como una escritura (dado que la lectura-modificación-escritura que implementa se realiza de forma atómica). Si se considera esto, tendríamos:

Estado(L2,C0)=I; Estado(L2,C1)=I
Estado(L2,C0)=M; Estado(L2,C1)=I (R(L2,N0)-W(L2,N0))
Estado(L2,C0)=I; Estado(L2,C1)=M (R(L2,N1)-W(L2,N1))
(se actualiza L2 desde C0 antes de cargar L2 en C1)

NOTACIÓN:

Estado(Li,Cj): Estado de la línea Li en la cache Cj
R(Li,Nj): Lectura de la línea Li generada por el nodo Nj
W(Li,Nj): Escritura en la línea Li generada por el nodo Nj