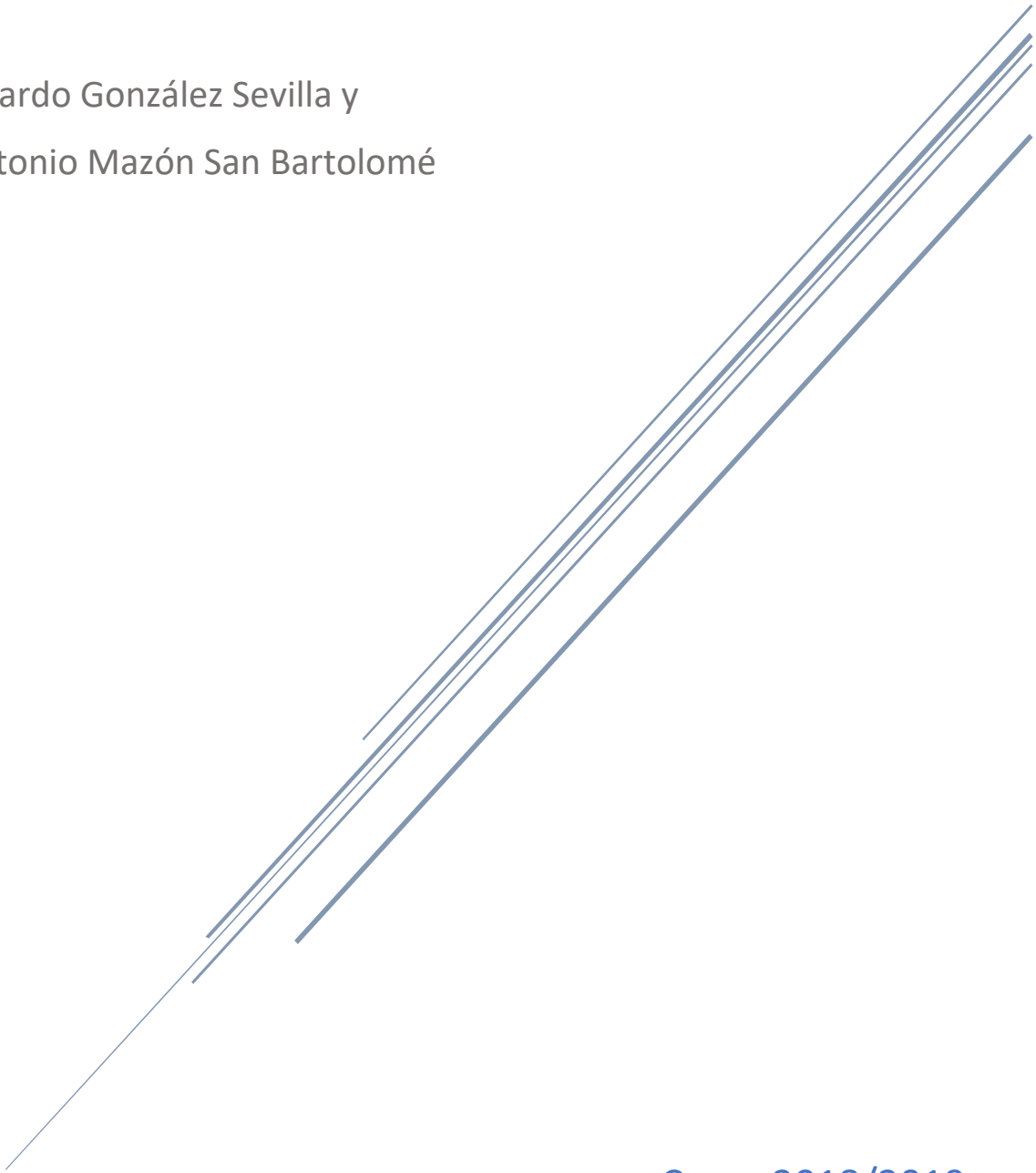
UNIVERSIDAD DE MURCIA

MEMORIA PRÁCTICA 3

Arquitectura y Organización de Computadores

Por Eduardo González Sevilla y

José Antonio Mazón San Bartolomé



Curso 2018/2019

Grupo 2.1

ÍNDICE

BLOQUE I

1.

El comportamiento de esta versión *multicore* es el siguiente: la matriz de temperaturas de *n* filas se divide en bloques de *fpb* filas y los procesos se reparten el trabajo actualizando cada uno un cierto número de bloques (*bpp*).

Así, el número de bloques que actualiza cada proceso se calcula a partir del número de filas de la matriz, el número de filas que componen un bloque y el número de procesos que se utilizarán siguiendo la siguiente fórmula:

$$bpp = \frac{n}{fpb * number\_of\_processes}$$

Se garantiza que esta ejecución obtiene los mismos resultados que la ejecución del programa no paralizado gracias al uso de cerrojos y barreras de la siguiente forma: antes de entrar al bucle principal del procedimiento `solve()` los procesos tienen que obtener su *pid*, para lo que se utiliza un cerrojo dado que se modificará una variable global, y esperar a que al resto de procesos se le haya asignado su *pid*, para lo que se utiliza una barrera.

Una vez todos los procesos tienen su *pid*, realizan los cálculos de temperaturas para sus respectivos bloques y actualizan la diferencia global si la suya local es superior (para esta actualización también se hace uso de cerrojos, pues se actualiza la diferencia global, una sección crítica).

En cuanto los procesos hayan realizado esto último (para lo que se utiliza otra barrera), el proceso 0 comprueba, mientras el resto espera en una barrera, si la diferencia global es mayor que la tolerancia, en cuyo caso se termina la ejecución y de lo contrario seguiría el bucle.

2.

Como hemos explicado anteriormente, en esta implementación se divide la matriz de temperaturas en bloques, y cada proceso actualizará un determinado número de bloques en lugar de hacer los cálculos de toda la matriz. Cada bloque está formado por un cierto número de filas de la matriz de temperaturas. A este número de filas que componen cada bloque se le denomina *fpb* (filas por bloque).

Dada la implementación, cuanto mayor sea el *fpb*, más aprovecharemos la localidad espacial, lo cual podría llevarnos a concluir que a mayor *fpb* mejor. Sin embargo, un *fpb* muy alto aumentará el número de colisiones en la caché,

repercutiendo negativamente en el tiempo de ejecución y, por ende, en el rendimiento. Así, debemos optar por una solución de compromiso, buscando el término medio para el valor de *fpb*.

AssociateAddrNode se utiliza para asignar el nodo home de la caché a un determinado rango de memoria pasado por parámetro. En esta implementación se utiliza para designar el nodo home de las matrices *A* y *B* tratando de aprovechar la localidad espacial.

3.

Con esta configuración, contando con que no modificamos los bordes, tenemos que:

$$bpp = 2; fpb = 1; n = 4;$$

Así, para cada hilo el primer *for* ejecutará 2 iteraciones, el segundo 1 iteración y el tercero 4.

Dado que estudiaremos solo fallos de caché en la iteración k-ésima con k > 2, las primeras 3 iteraciones del último *for* no serán tenidas en cuenta:

En cada núcleo se hacen 7 accesos de memoria por cada iteración del tercer *for* (el único en que se accede a memoria). Dado que se utilizan bloques de 8 bytes (del tamaño de un *double*) y se leen y escriben *doubles*, no se aprovecha en absoluto la localidad espacial (dado que solo se traerá a caché el bloque que ha producido el fallo), con lo que los primeros 6 accesos resultan en fallos en frío. Sin embargo, el séptimo resulta en acierto, pues al ser la caché totalmente asociativa, el bloque que contiene *A[i + row_offset][j]* está en caché.

De esta forma, tenemos 6 fallos en frío por cada iteración del tercer *for*.

Además, el algoritmo hace que los hilos se repartan las filas de forma que la fila 0 la realiza el hilo 0, la fila 1 el hilo 1, la fila 2 el hilo 0, la fila 3 el hilo 1 y así sucesivamente. Así, cada vez que un hilo accede a una fila del otro hilo para actualizar los valores de las casillas de la fila de la que se encarga, se producirán 8 fallos de coherencia.

Por tanto, tenemos que en la iteración k-ésima se producen 6 fallos en frío y 8 fallos de coherencia por cada núcleo, resultando en **14 fallos de caché L2 por núcleo**.

BLOQUE II

4.

Al utilizar únicamente un hilo, el valor óptimo para el *fpb* sería el máximo posible, es decir, el número de filas de la matriz, ya que el valor óptimo para el *fpb* viene dado por el número de filas dividido entre el número de procesos. **Con lo que el valor óptimo para el *fpb* es 64.**

- 1 *fpb* -> 119.968.324,5 ciclos
- 8 *fpb* -> 119.829.526,5 ciclos
- 64 *fpb* -> 119.802.019 ciclos

Se observa cómo al incrementar el *fpb* hasta el máximo, se reduce el tiempo simulado de ejecución.

5.

Tiempo de ejecución simulado: 92.500.000 ciclos

$$Speedup = \left(\frac{119.802.019}{92.500.000} - 1\right) \cdot 100$$

$$Speedup = 29{,}5156\ \%$$

A continuación, se muestran los datos de las estadísticas obtenidas con la ejecución del programa con 1 hilo y 64 *fpb* (la mejor configuración del ejercicio anterior) y con 2 hilos y 1 *fpb*:

1 hilo y 64 *fpb*

Número total de fallos de L2: 911.999

Fallos de lectura:

- Fallos en frío: 1.175
- Fallos de conflicto: 92.449
- Fallos de capacidad: 406.353
- Fallos de coherencia: 0
- Fallos de actualización: 0

Fallos de escritura:

- Fallos en frío: 1.088
- Fallos de conflicto: 0
- Fallos de capacidad: 410.176
- Fallos de coherencia: 0
- Fallos de actualización: 379

2 hilos y 1 *fpb*

Número total de fallos: 432.663

Fallos de lectura:

- Fallos en frío: 1.685
- Fallos de conflicto: 1.885
- Fallos de capacidad: 218.099
- Fallos de coherencia: 1.960
- Fallos de actualización: 0

Fallos de escritura:

- Fallos en frío: 544
- Fallos de conflicto: 0
- Fallos de capacidad: 205.088
- Fallos de coherencia: 0
- Fallos de actualización: 2.645

Los resultados evidencian que se producen menos de la mitad de fallos de L2 usando 2 hilos y un *fpb* de 1. Aunque tener dos hilos provoca que tengamos fallos de coherencia en las lecturas (algo que no ocurre con un hilo), los fallos de capacidad tanto en lecturas como escritura se reducen a casi la mitad, y teniendo en cuenta que estos últimos son mucho más abundantes que los anteriores, el balance general es muy positivo.

6.

El valor óptimo para el *fpb* sería 32, ya que el valor óptimo para el *fpb* viene dado por el número de filas dividido entre el número de procesos: 64/2 = 32.

Tiempo de ejecución simulado: 52.841.546 ciclos

$$Speedup = \left(\frac{92.500.000}{52.841.546} - 1\right) \cdot 100$$

$$Speedup = 102{,}2671\ \%$$

A continuación, se muestran los datos de las estadísticas obtenidas con la ejecución del programa con 2 hilos y 1 fpb (la mejor configuración del ejercicio anterior) y con 2 hilos y 32 *fpb*:

2 hilos y 1 *fpb*

Número total de fallos: 432.663

Fallos de lectura:

- Fallos en frío: 1.685
- Fallos de conflicto: 1.885
- Fallos de capacidad: 218.099
- Fallos de coherencia: 1.960
- Fallos de actualización: 0

Fallos de escritura:

- Fallos en frío: 544
- Fallos de conflicto: 0
- Fallos de capacidad: 205.088
- Fallos de coherencia: 0
- Fallos de actualización: 2.645

2 hilos y 32 *fpb*

Número total de fallos: 122.178

Fallos de lectura:

- Fallos en frío: 623
- Fallos de conflicto: 55.789
- Fallos de capacidad: 22
- Fallos de coherencia: 1.894
- Fallos de actualización: 0

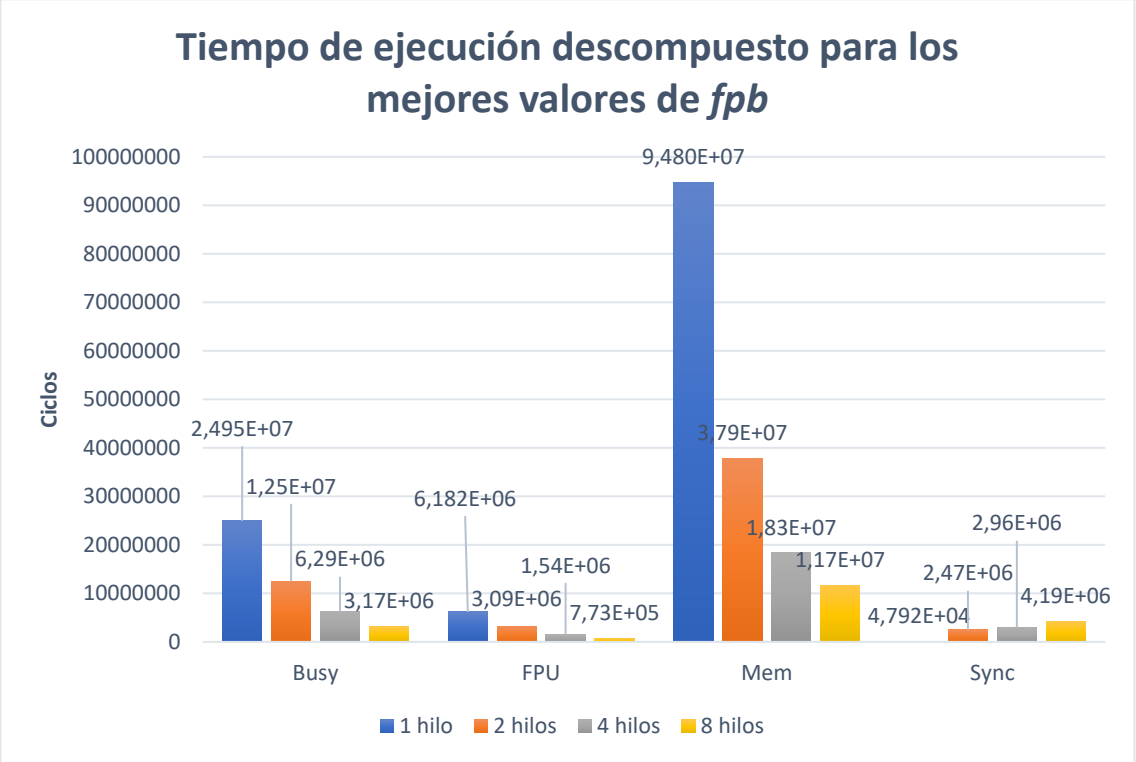
Fallos de escritura:

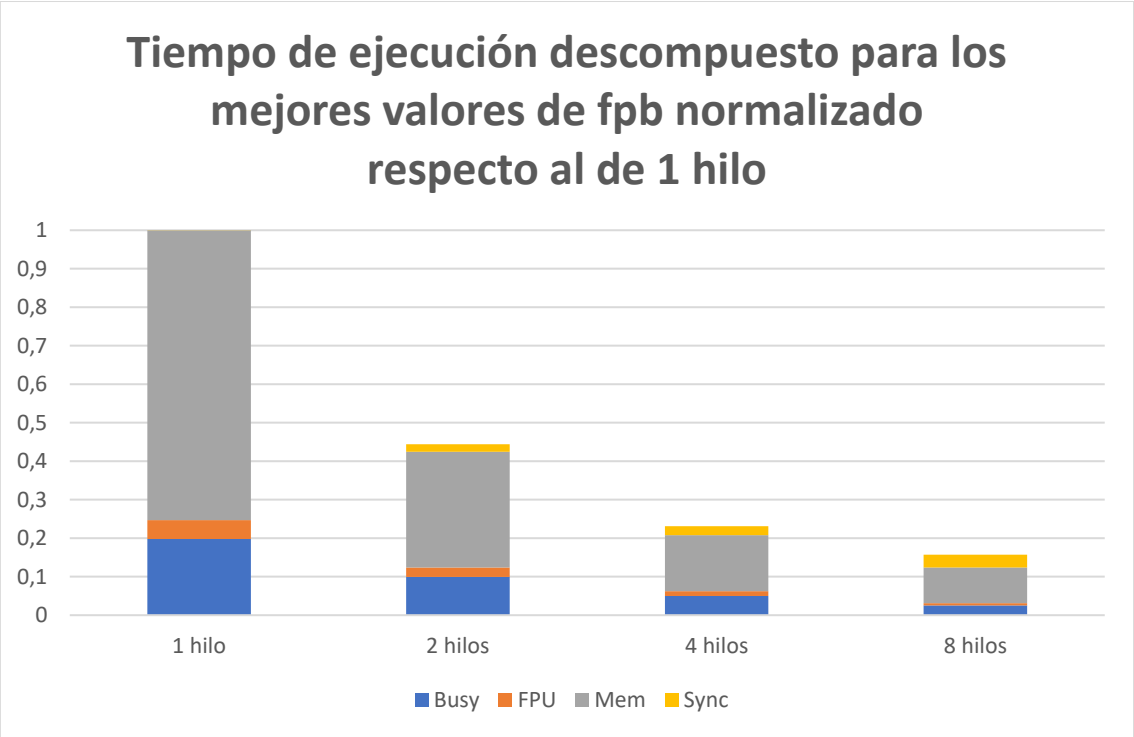
- Fallos en frío: 544
- Fallos de conflicto: 57.121

- Fallos de capacidad: 177
- Fallos de coherencia: 0
- Fallos de actualización: 5.251

Los resultados evidencian que, con un *fpb* de 32 se producen poco más de un cuarto de los fallos de L2 de los que ocurrían con un *fpb* de 1. Observamos que los fallos de capacidad se reducen muchísimo (más de un 800% en las lecturas) y los fallos de coherencia se ven ligeramente reducidos. Pese a que aumentan los fallos de conflicto en un 2.859,62% en las lecturas, estos son lo suficientemente comedidos como para que el balance general siga siendo muy positivo.

7.





En efecto, podemos afirmar que la aplicación escala adecuadamente, pues el tiempo empleado en la sincronización aumenta poco a poco al añadir más núcleos, al mismo tiempo que se reduce drásticamente el tiempo empleado en el resto de operaciones.

Por su parte, consideramos que el principal cuello de botella es la memoria.

8.

Analizando el código, concluimos que todos *locks* son necesarios, pues hay que acceder a las secciones críticas en exclusión mutua.

Sin embargo, las barreras 1 puede ser desactivada sin que cambie el resultado de la ejecución.

Así, hemos desactivado únicamente la primera barrera, ya que no es necesario que todos los procesos hayan obtenido su *pid* para que continúe la ejecución del programa.

Como muestra de que se obtienen los mismos resultados se adjunta la salida con las configuraciones de los ejercicios 7 y 8 para 1 hilo y 64 *fpb*:

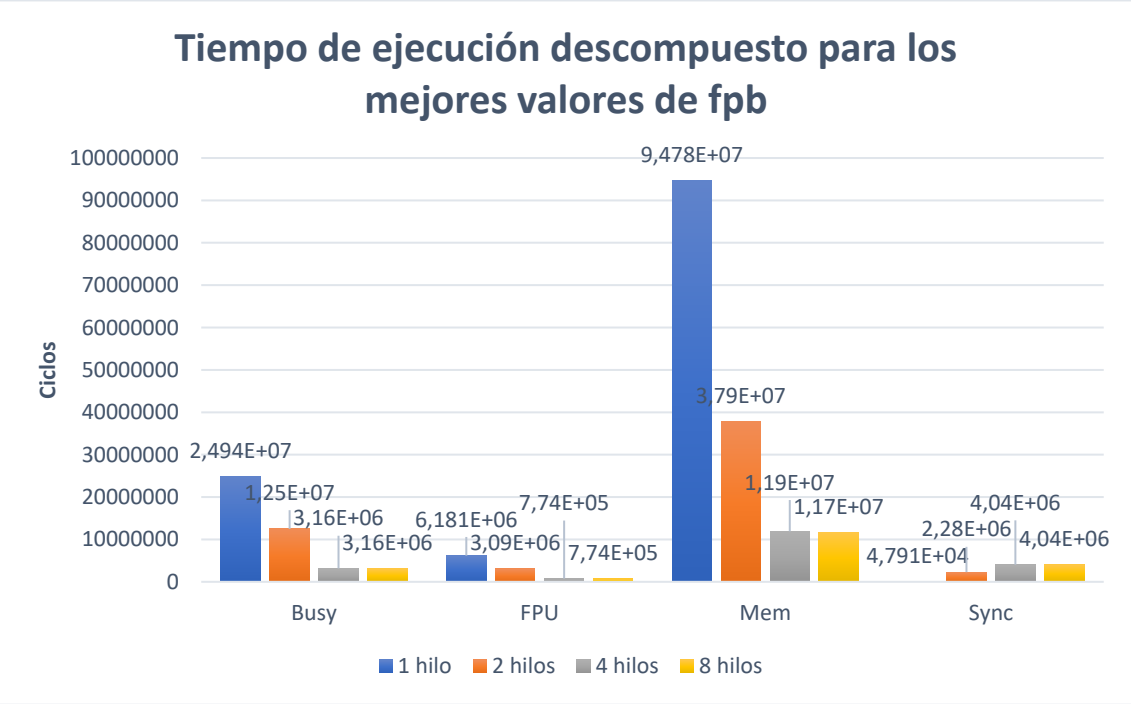
Salida 7

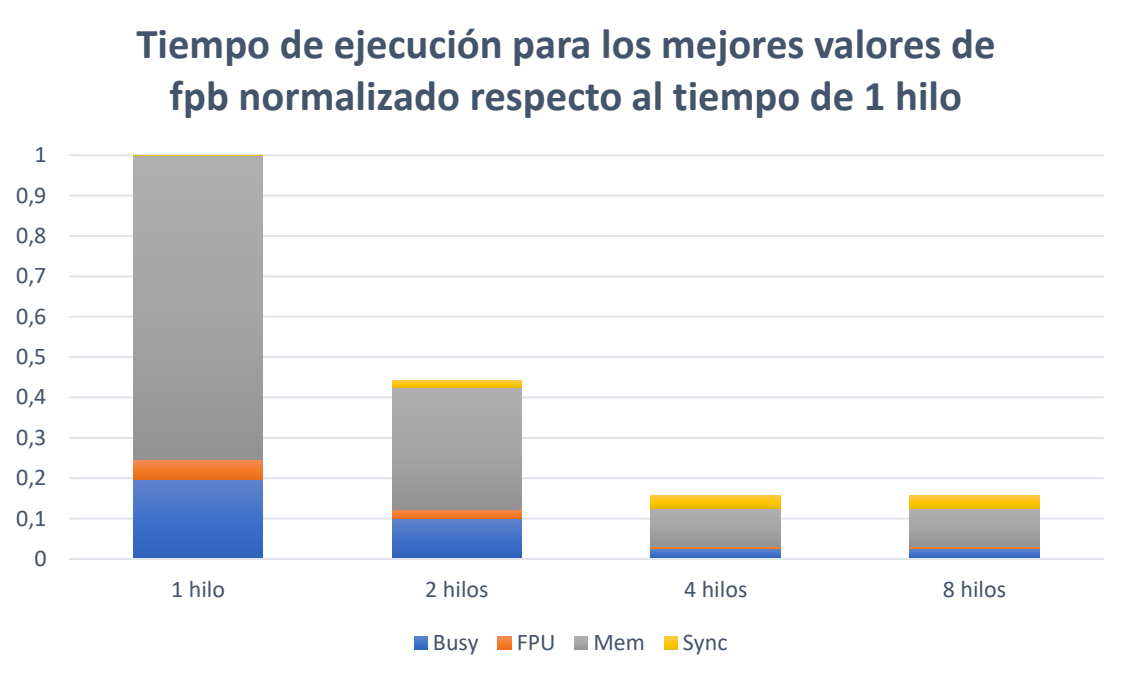
```
N = 64, Número de hilos = 1, Tolerancia = 0.100000, FPB = 64
Flags de usuario: 1 1 1 1 1 1 1 1
Número de iteraciones: 378
```

Salida 8

```
N = 64, Número de hilos = 1, Tolerancia = 0.100000, FPB = 64
Flags de usuario: 1 0 1 1 1 1 1 1
Número de iteraciones: 378
```

Sin más, mostramos en dos gráficas los resultados obtenidos:





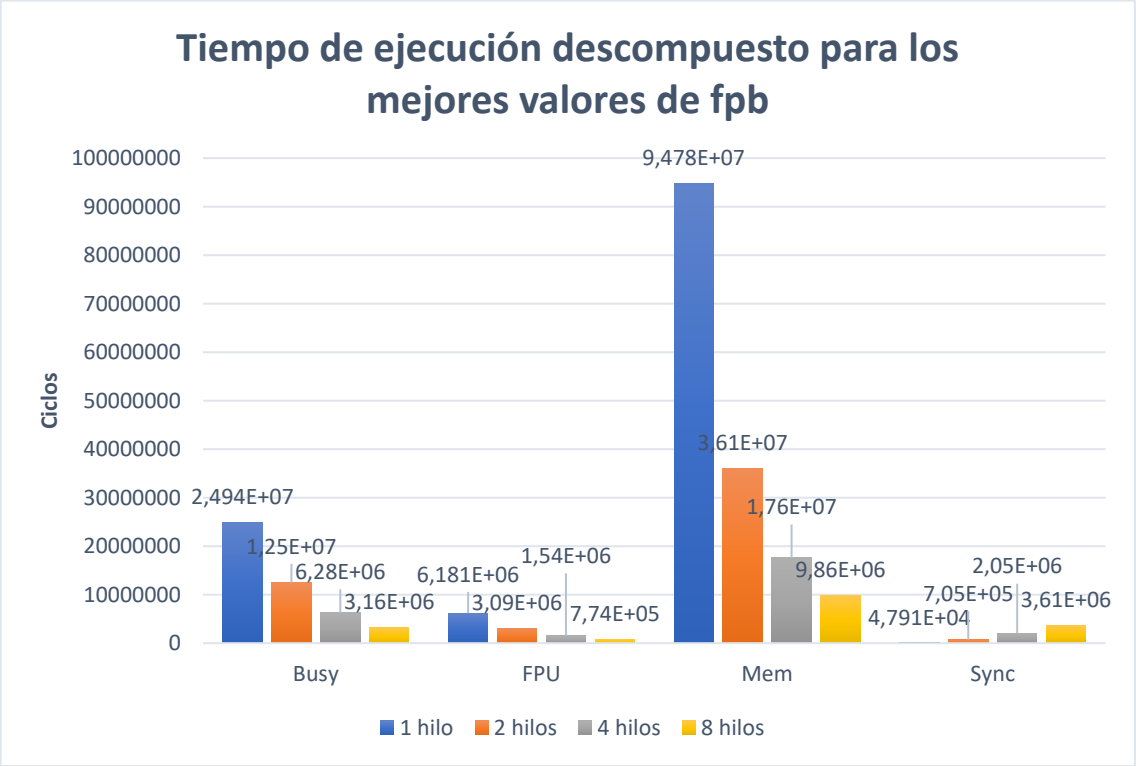
De nuevo, concluimos que la aplicación es escalable, dado que el tiempo empleado en sincronización se incrementa poco a poco.

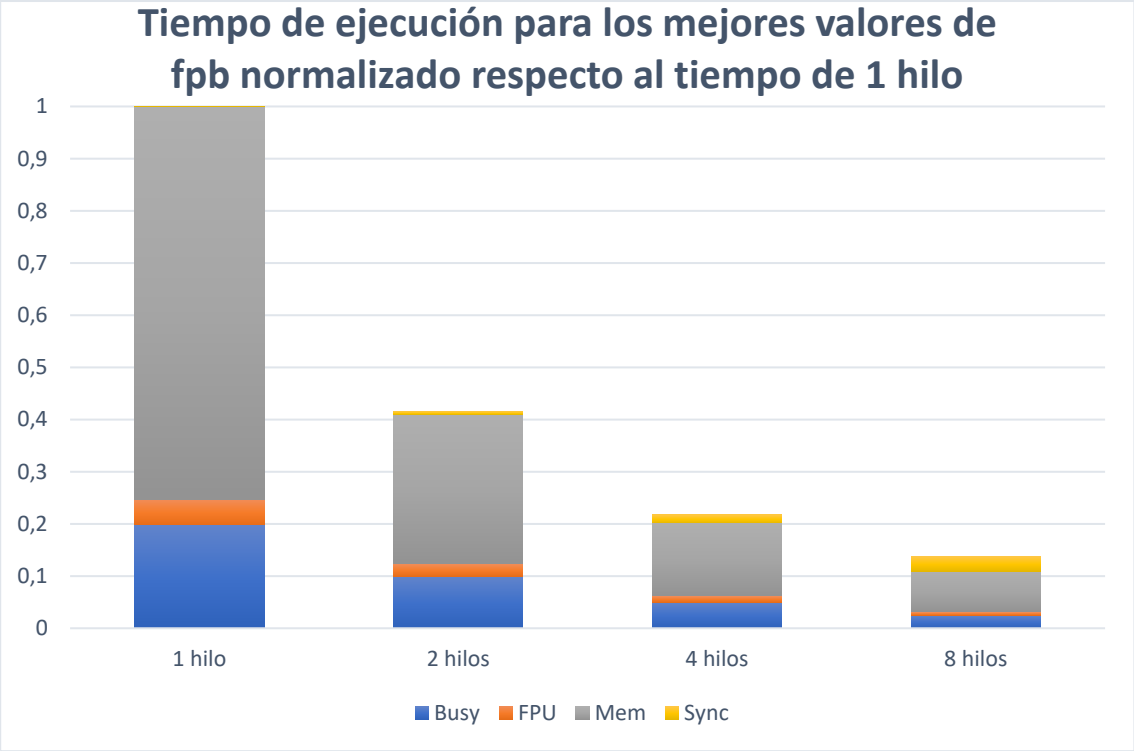
Asimismo, el principal cuello de botella es, otra vez, la memoria.

9.

Al desactivar el *flag* para el `AssociateAddrNode` se aplicará una política de *first-touch*, es decir, que la política de memoria se hará efectiva cuando se produzca el alojamiento de bloques.

Así, las siguientes gráficas ofrecen los resultados de esta configuración:





A partir de estas gráficas extraemos la conclusión de que al desactivar dicho *flag* la aplicación sigue siendo escalable, pues el tiempo de sincronización aumenta poco a poco con el número de núcleos. Asimismo, dado que se usa política *firt-touch* el tiempo de memoria se reduce, obteniendo resultados mejores que los de los ejercicios anteriores cuando utilizamos 2 y 8 hilos, aunque obteniendo peores datos con 4 hilos.

BLOQUE III

10.

Puesto que el mejor resultado para el superescalar lo obteníamos con una configuración con un *predictor 2bitagree de 64 contadores*, que arrojaba un tiempo de ejecución de 13.635.393 ciclos, y dado que el mejor tiempo de ejecución para el multihilo es de 16.635.589 ciclos, proporcionado por la ejecución con *8 hilos y 8 fpb* del ejercicio anterior, **concluimos que el superescalar, para esta aplicación en concreto, ofrece un mejor resultado, pues su tiempo de ejecución es menor**, lo cual no quita que el multihilo se pueda comportar mejor en otras aplicaciones.