Variables private y shared

En el ejemplo del producto punto:

```
double ProductoPunto(double* a, double* b, int size)
{
    double c = 0;
    #pragma omp parallel for reduction(+:c)
    for (int i = 0; i < size; ++i)
    {
        c += a[i]*b[i];
    }
    return c;
}
```

OpenMP considera **a** y **b** como variables compartidas (**shared**), es decir, todos los threads las pueden accesar y modificar.

Para las variables del ciclo **i** y de la reducción **c**, OpenMP hace un copias de las variables al stack de cada thread, con lo cual se convierten a privadas (**private**). Al terminar se sumarán las **c** de cada thread en la variable **c** que ahora se considerará shared.

Además, todas las variables declaradas dentro del bloque parallel serán privadas.

También es posible decidir de forma explícita que variables queremos que tengan un comportamiento private o shared.

El siguiente ejemplo muestra como:

```
int i[10];
int j = 5;
float* a;
float* b;
#pragma omp parallel private(i) firstprivate(j) shared(a,b)
{
    int k;
    // ... codigo ...
}
```

En este caso, tanto i, j y k serán privadas y a y b serán compartidas.

firstprivate indica que el valor de j se inicializará con el valor que tenía antes del bloque parallel.

Private vs firstprivate

¿Cuál será el resultado en el siguiente ejemplo?

```
int main()
{
    int a = 7;

    #pragma omp parallel private(a)
    {
        printf("private a = %i\n", a);
    }

    #pragma omp parallel firstprivate(a)
    {
        printf("firstprivate a = %i\n", a);
    }

    return 0;
}
```

```
g++ -o private -fopenmp private.cpp
./private
```

Schedule

Por default, al paralelizar un ciclo el número de iteraciones se divide por igual entre cada thread. Sin embargo, esto no es eficiente si las iteraciones tardan tiempos diferentes en ejecutarse.

Es posible hacer entonces reajustar la distribución de iteraciones de forma dinámica, esto se hace con schedule.

```
#pragma omp parallel for schedule(type[, chunk_size])
for (int i = 0; i < size; ++i)
{
    c[i] = a[i] + b[i];
}
```

Los tipos de schedule son:

static	Las iteraciones se dividen en chunks de un tamaño definido.
dynamic	Cada thread ejecuta un chunk de iteraciones y al terminar su parte busca otro chunk para procesar.
guided	Cada thread ejecuta un chunk de iteraciones y al terminar su parte busca otro chunk para procesar. El tamaño del chunk varía, siendo grande al iniciar y éste se va reduciendo hasta llegar a *chunk_size*.
auto	La decisión del tipo de schedule es tomada por el compilador/sistema operativo.
runtime	El schedule y el tamaño del chunk se pueden cambiar con la función **omp_set_schedule** o la variable de ambiente **OMP_SCHEDULE**.

Paralelizar secciones de código

Se puede hacer que varias secciones de código se ejecuten de forma simultánea

```
int threads = 2;
#pragma omp parallel sections num_threads(threads)
{
    #pragma omp section
    {
        // ... algo de codigo
    }
    #pragma omp section
    {
        // ... otro codigo
    }
}
```

Con num_threads podemos especificar cuantos threads/cores se utilizarán. En este caso uno por cada sección.

Secciones críticas

```
file1 = fopen("variables.mat", "rb");
file2 = fopen("result.mat", "wb");

#pragma omp parallel for
for (int i = 0; i < 10000000; ++i)
{
    ...
    #pragma omp critical (read_from_file)
    {
        fread(file1, “%f”, x);
    }
    ...
    #pragma omp critical (write_to_file)
    {
        fwrite(file2, “%i\n”, i);
    }
    ...
}

fclose(file2);
fclose(file1);
```

La ejecucción dentro de un área crítica se restringe a un solo thread a la vez.

Operaciones atómicas

Permiten que una variable sea actualizada por un thread a la vez.

```
int counter[10] = 0;

#pragma omp parallel for
for (int i = 0; i < 1000000; ++i)
{
    ...
    #pragma omp atomic update // [read | write | update | capture]
    counter[k]++;
    ...
}
```

En este ejemplo, al utilizar **atomic** se pueden actualizar diferentes elementos de counter en paralelo. Las operaciones atómicas son menos costosas que poner una sección crítica. Pero sólo funcionan para operaciones sencillas. Por ejemplo:

update: ++x, x++, --x, x--, x += y, x -= y, x *= y, x /= y, x |= y, x &= y, x ^= y, x = y + z, x = y - z, x = y*z, x = y/z, x = y | z, x = y & z, x = y ^ z

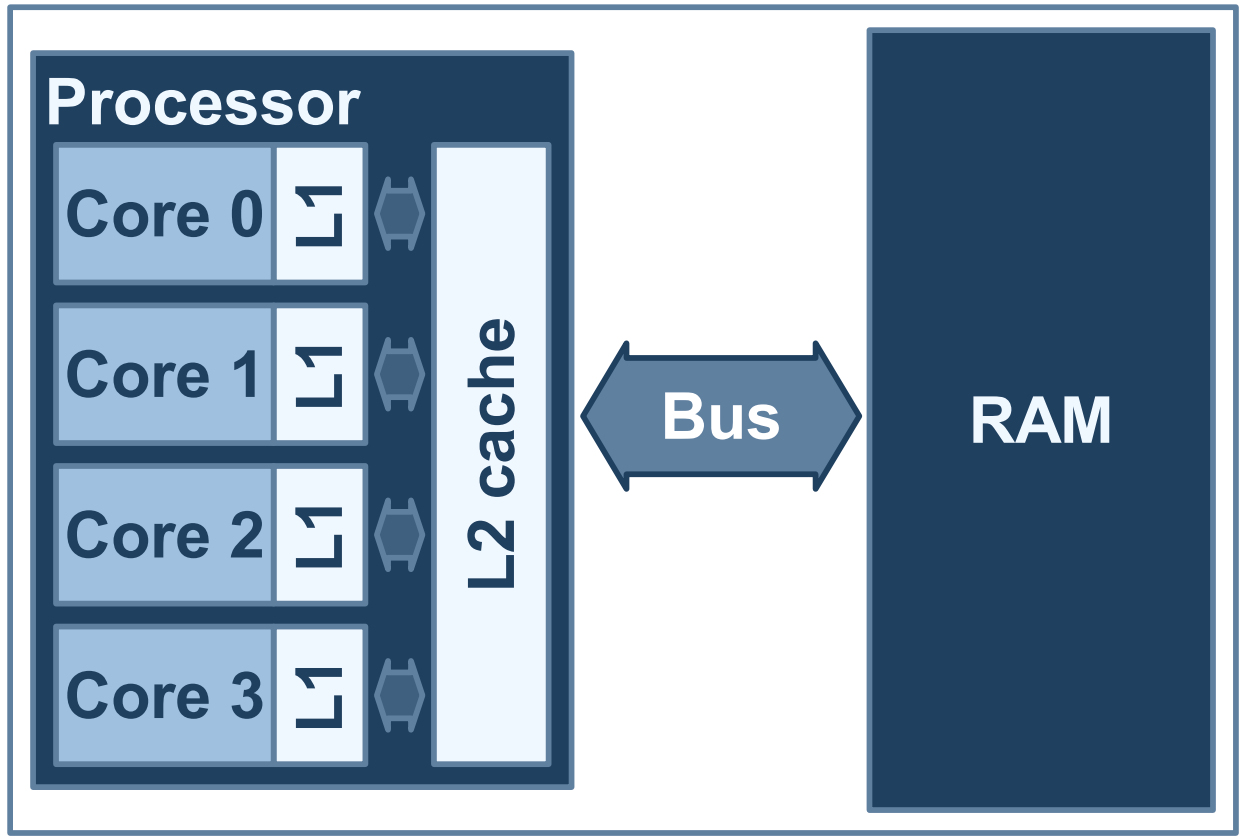
read: x = y

write: x = y

Cómputo en paralelo con memoria compartida

Hace 15+ años

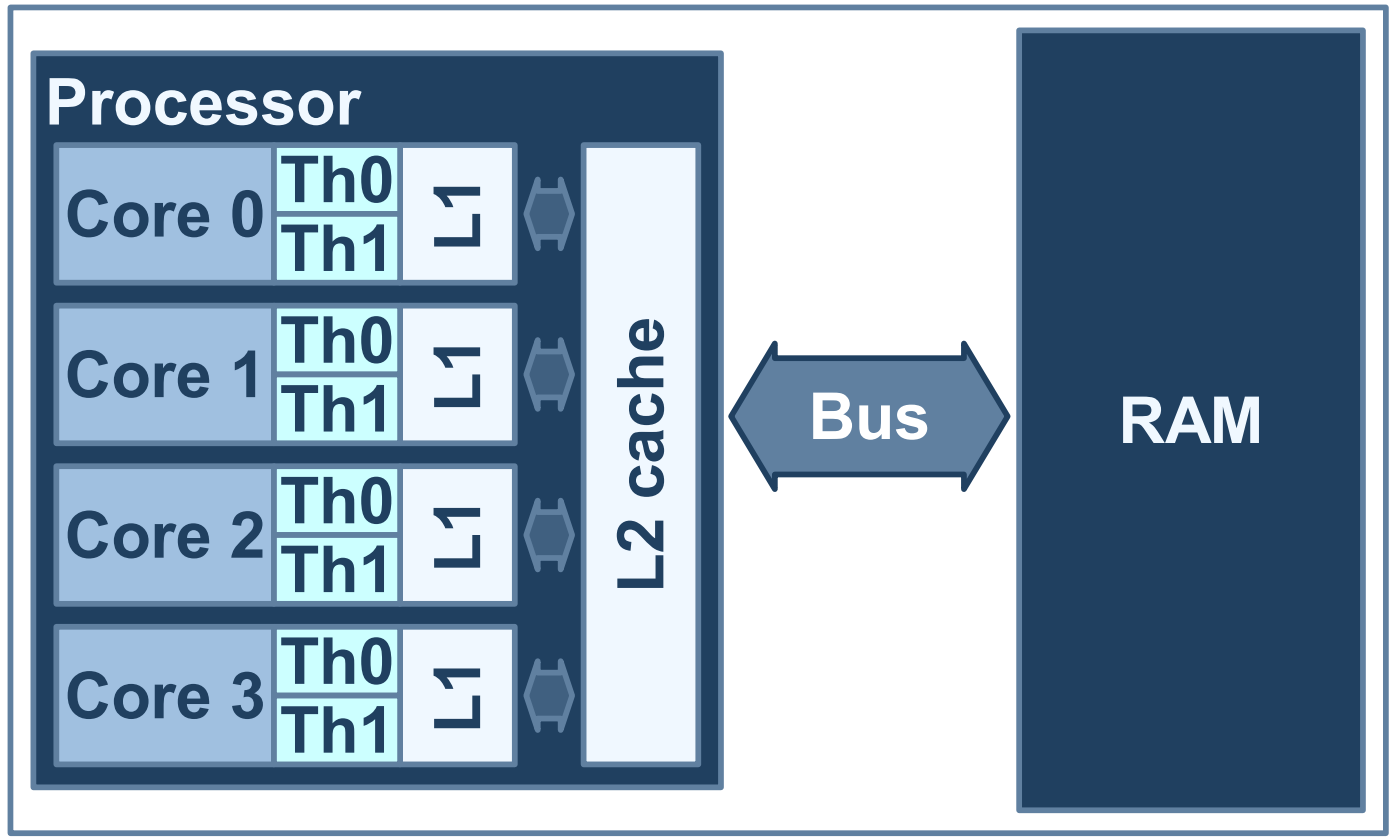
Los procesadores multi-core comienzan a dominar el mercado. En un solo procesador hay varias unidades de procesamiento o *cores*.



Esta configuración se llama de “memoria compartida”. El principal cuello de botella es que ¡los procesadores trabajan en **paralelo** pero el acceso a la RAM sigue siendo **serial**!

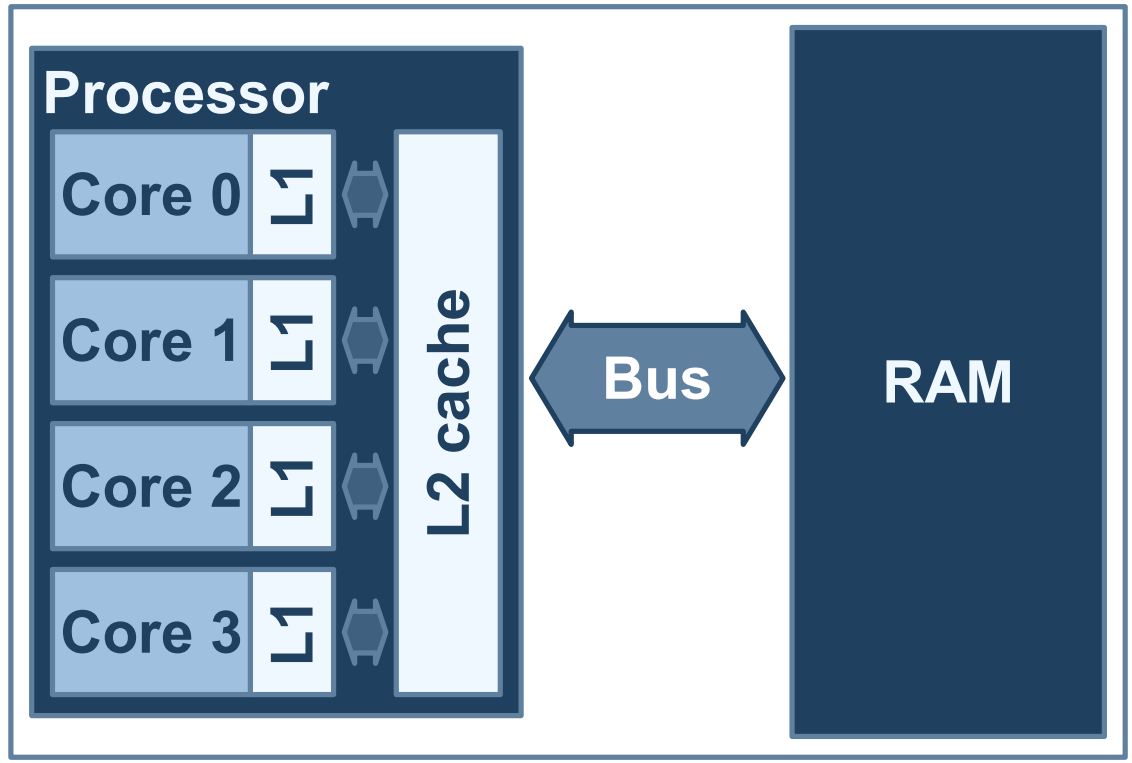
Intel Hyper-threading

Intel desarrollo una tecnología que permite “duplicar” la cantidad de unidades de procesamiento.



Los hyper-threads tienen la funcionalidad de poder controlar hilos de ejecución (duplicando los registros del CPU y el decodificador de instrucciones), pero para hacer la mayor parte de las operaciones requieren de compartir de forma alternada los circuitos del core. Además comparten el cache. Para procesos que requieren poca memoria el hyper-threading puede ser de gran ayuda, pero para procesos que requieren mucha memoria puede haber conflicto porque se van a estar peleando por los caches.

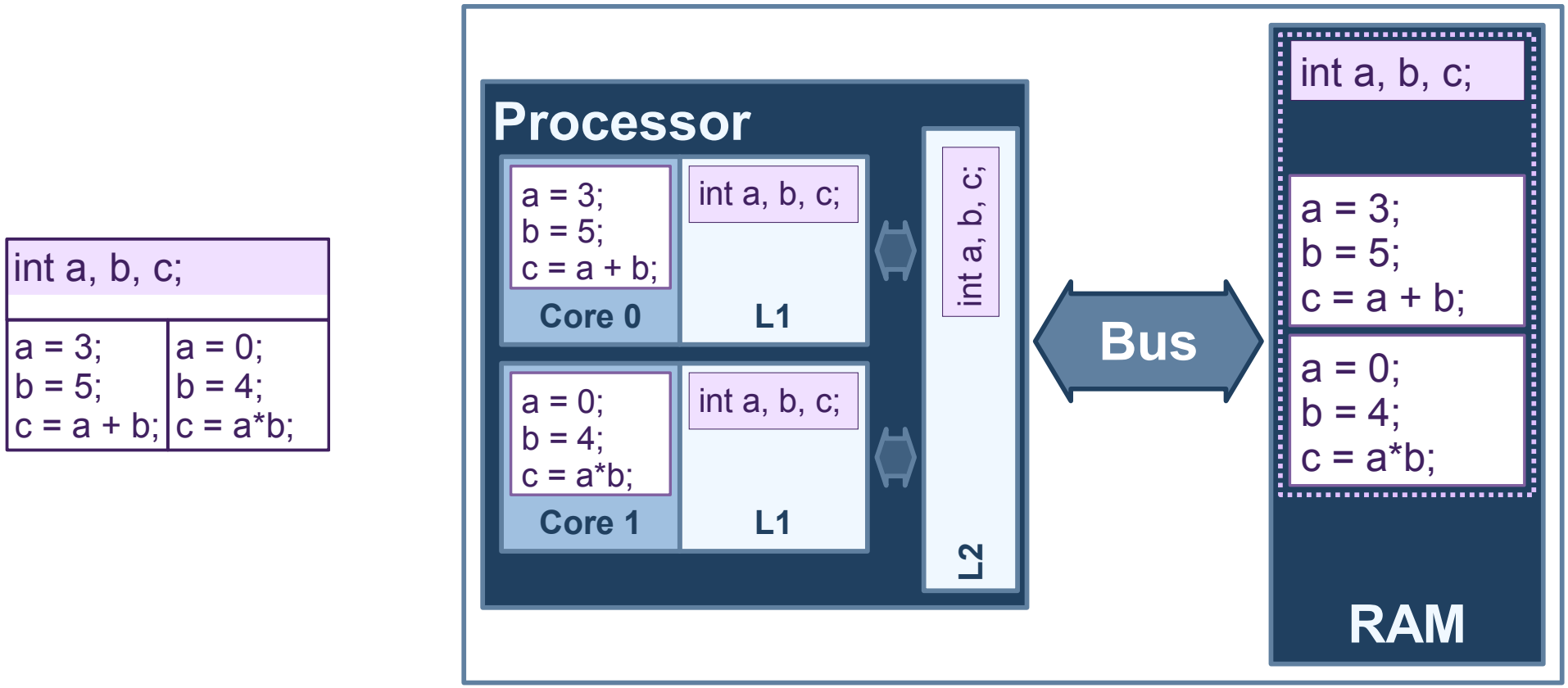
Procesadores multi-core



- Cada thread se ejecuta en un core. Tenemos un paralelismo real.
- El principal cuello de botella es en el acceso a la RAM.
- Sólo existe un canal de comunicación (bus de datos/direcciones) con la RAM.
- Si dos cores tratan de leer de la RAM uno tendrá que esperar a que el otro termine. Para reducir ésta latencia se le ha agregado otro nivel de cache, llamado L2. A veces también hay un L3.
- La pelea por el acceso a la RAM será más y mas notoria cuando el sistema tenga más cores.

El cache con multi-core

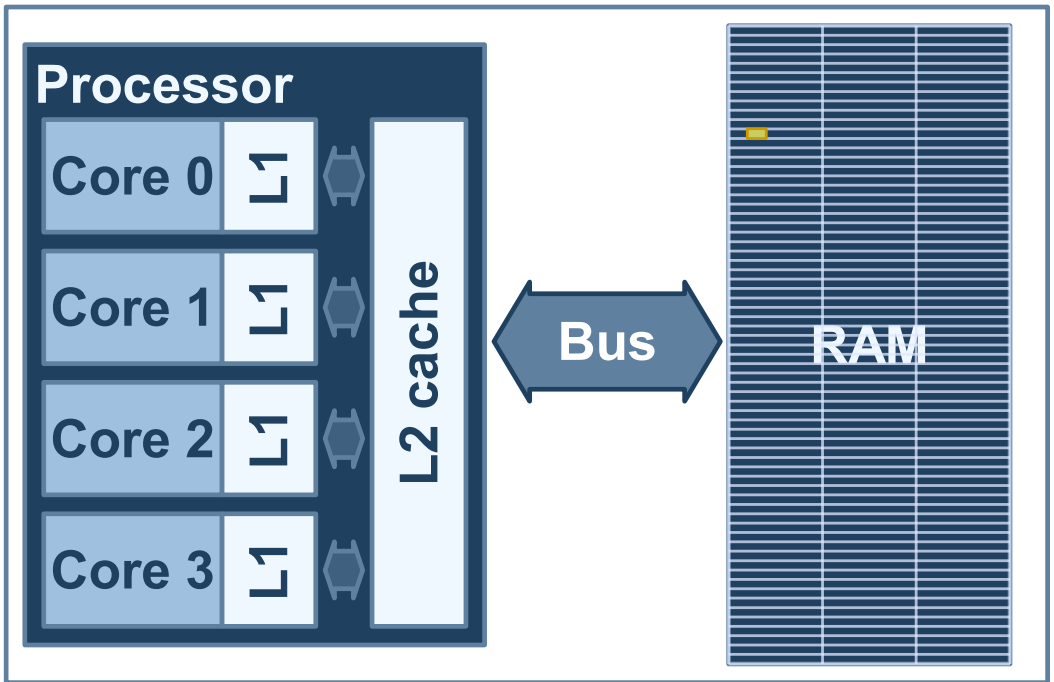
El problema de mantener el datos en cache aumenta con del esquema de procesamiento en paralelo con memoria compartida.



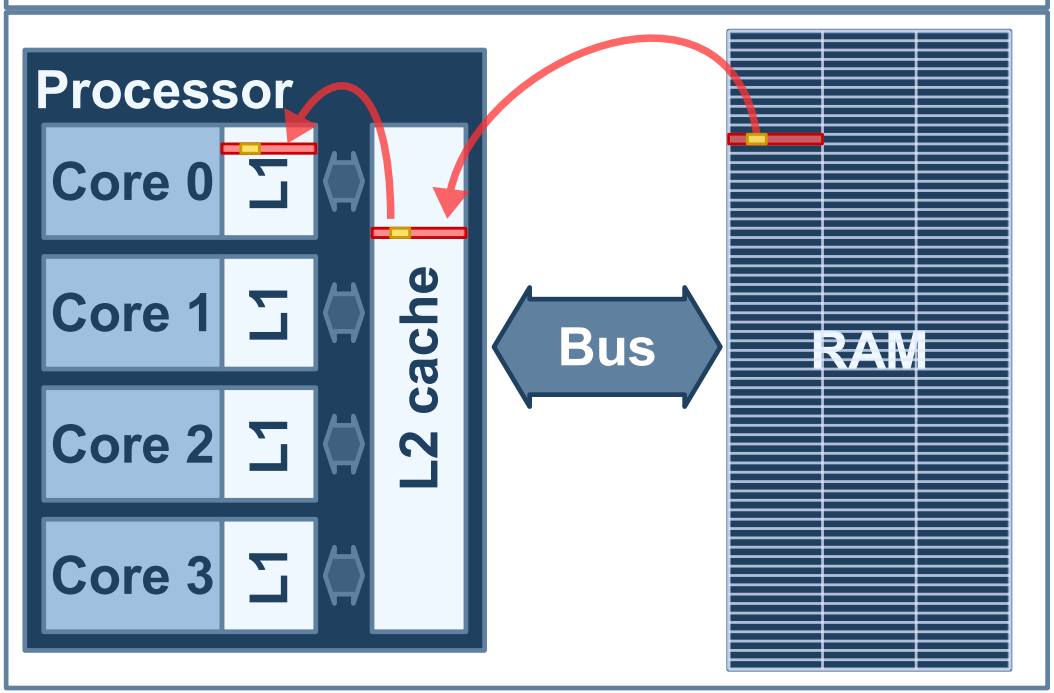
Si un core trata de actualizar memoria de la cual existe copia en el cache de otro core, entonces se crea un **cache-fault** y es necesario que uno de los cores accese a L2 o la RAM para sincronizarse.

Mantener coherencia en la información cuando varios cores accesan la misma memoria RAM es complejo. Los detalles se pueden consultar en [Drep07].

Acceso al cache con múltiples cores



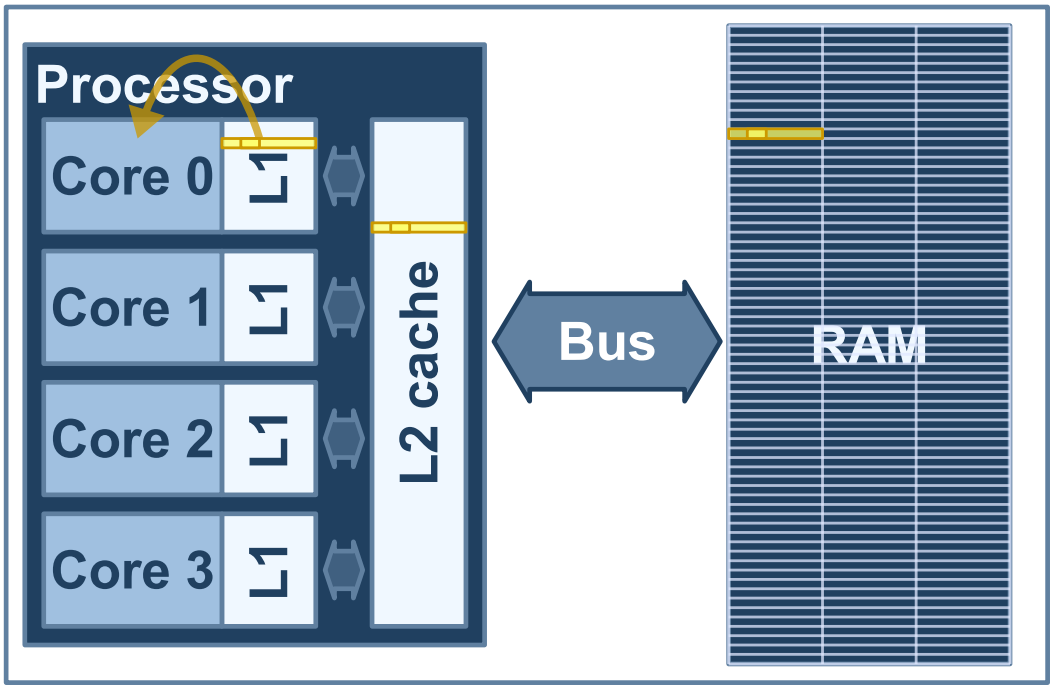
El core 0 solicita trabajar con d[1].



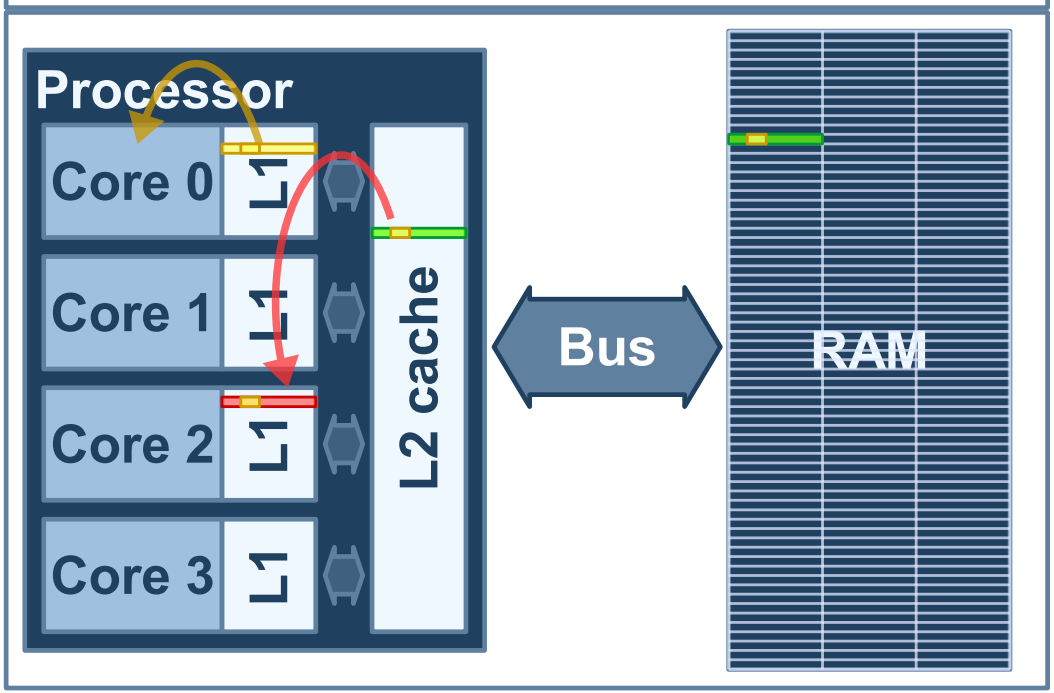
d[1] no existe en el cache L1 del core 0, se produce un **cache-miss**.

d[1] se tiene que copiar de la RAM al cache L2 y al cache L1 del core 0.

Acceso al cache con múltiples cores

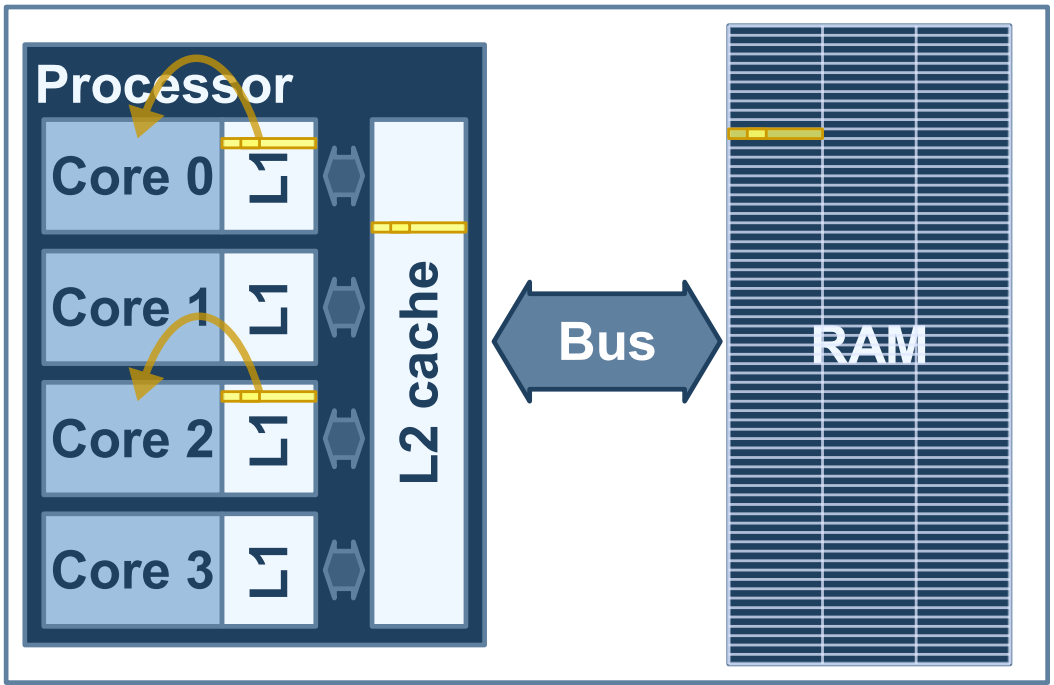


El core 0 puede trabajar con d[1].

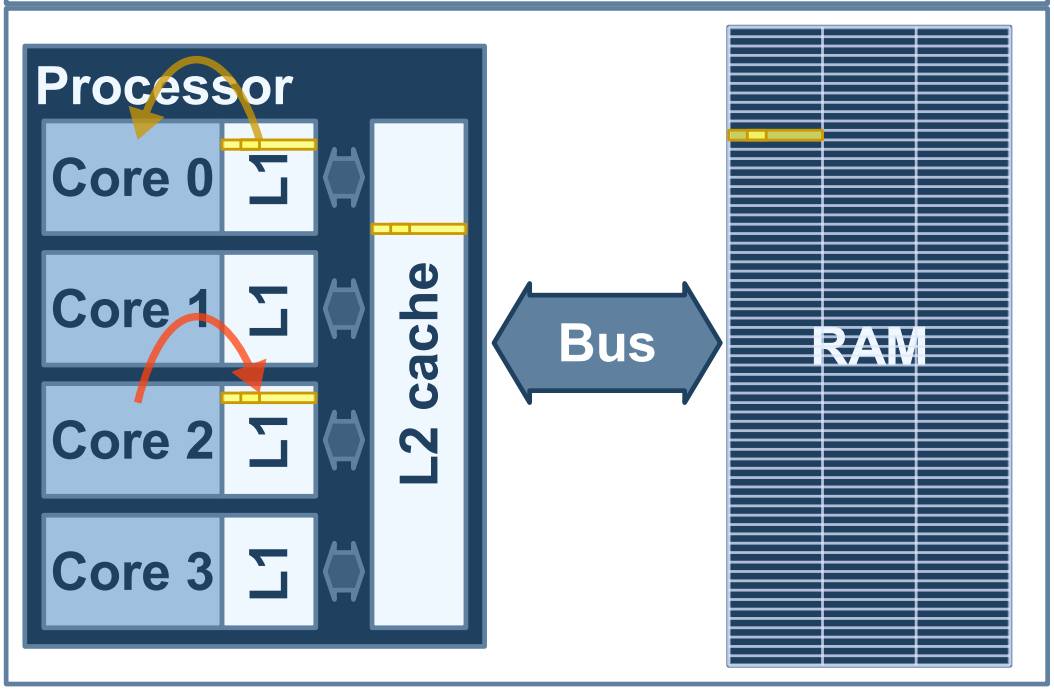


El core 2 pide trabajar con d[1], éste existe en L2 se produce un **cache-hit**, se copia al cacle L1 del core 2.

Acceso al cache con múltiples cores

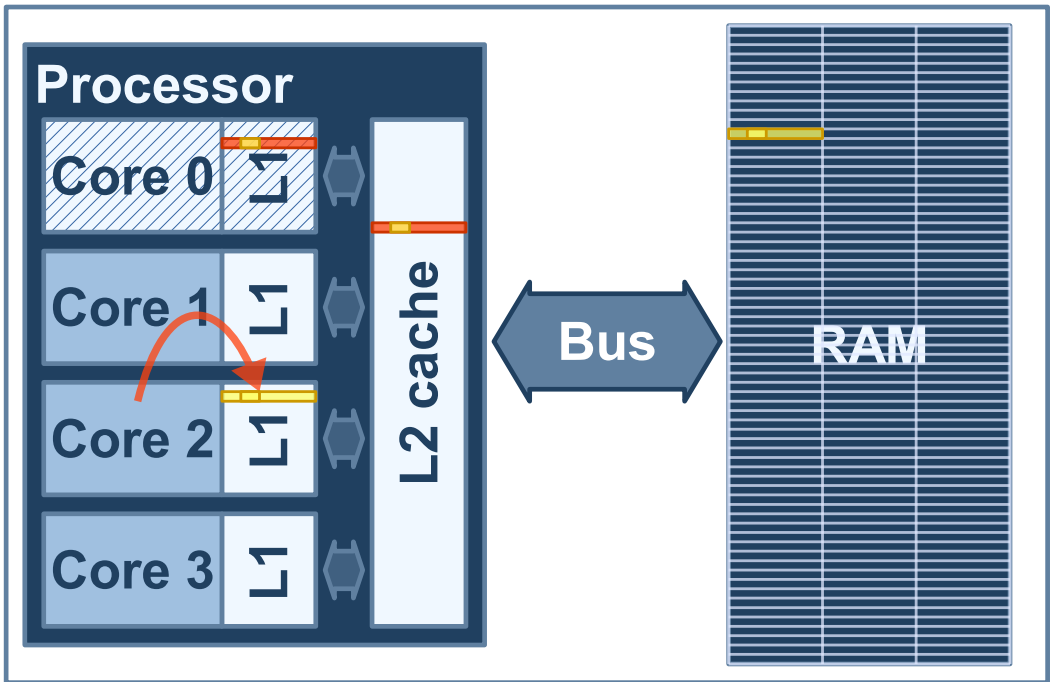


El core 2 también puede trabajar con d[1].

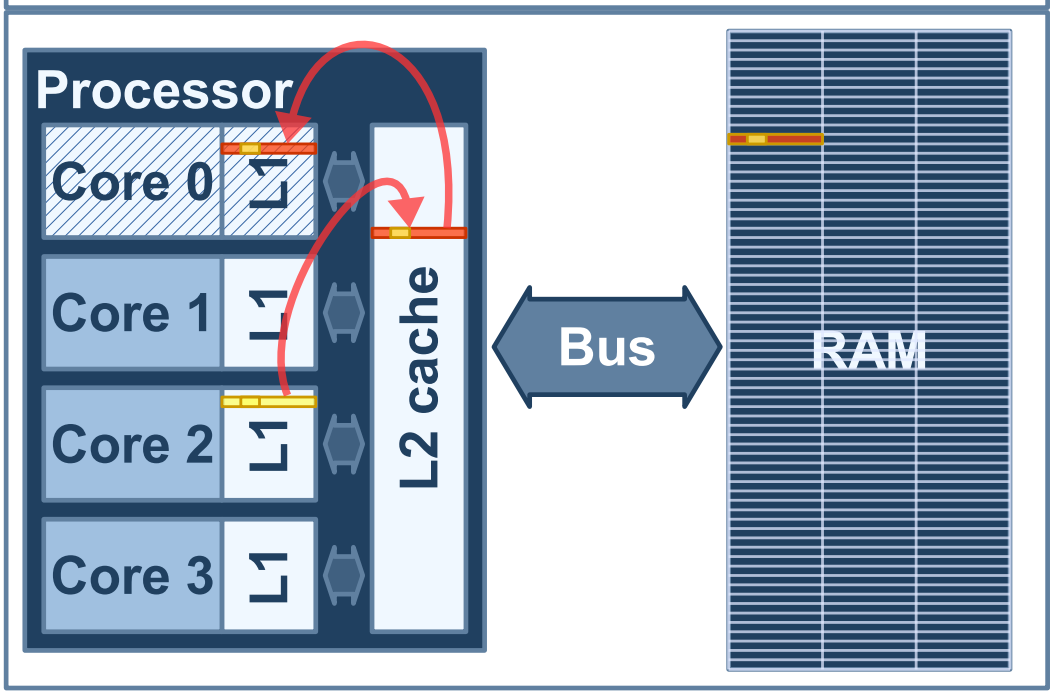


El core 2 modifica d[1] mientras el core 0 trabaja con el d[1] del su cache L1.

Acceso al cache con múltiples cores

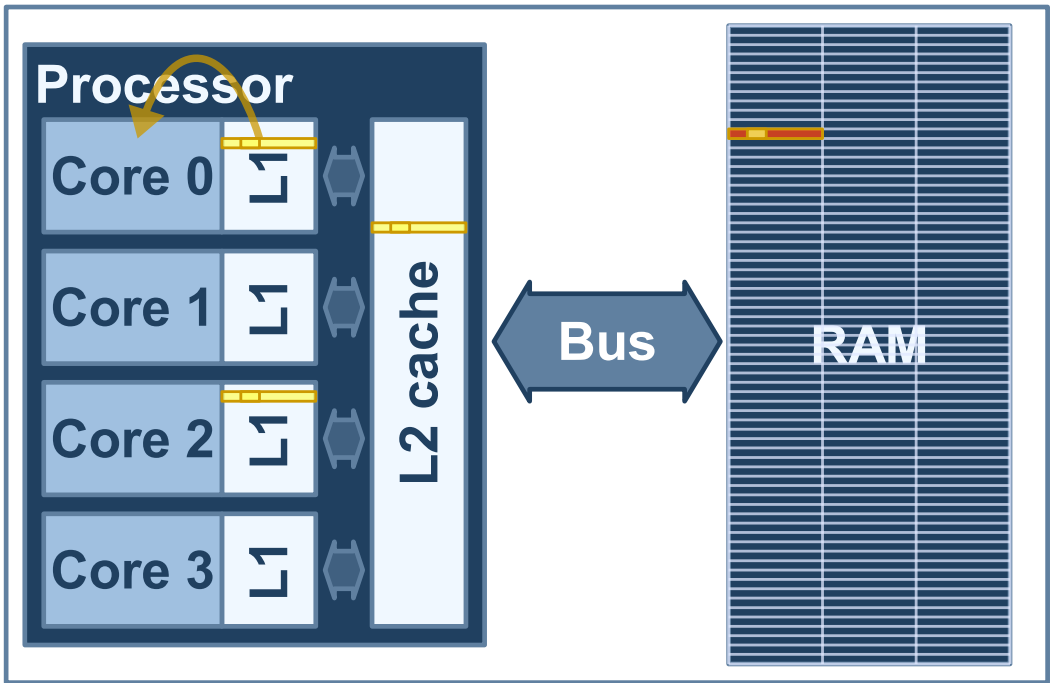


El valor de d[1] en el cache L1 del core 2 es diferente al valor de d[1] en el cache L2 y al valor de d[1] en el cache L1 del core 0. Se produce un **cache-fault**, **el core 0 se detiene**.

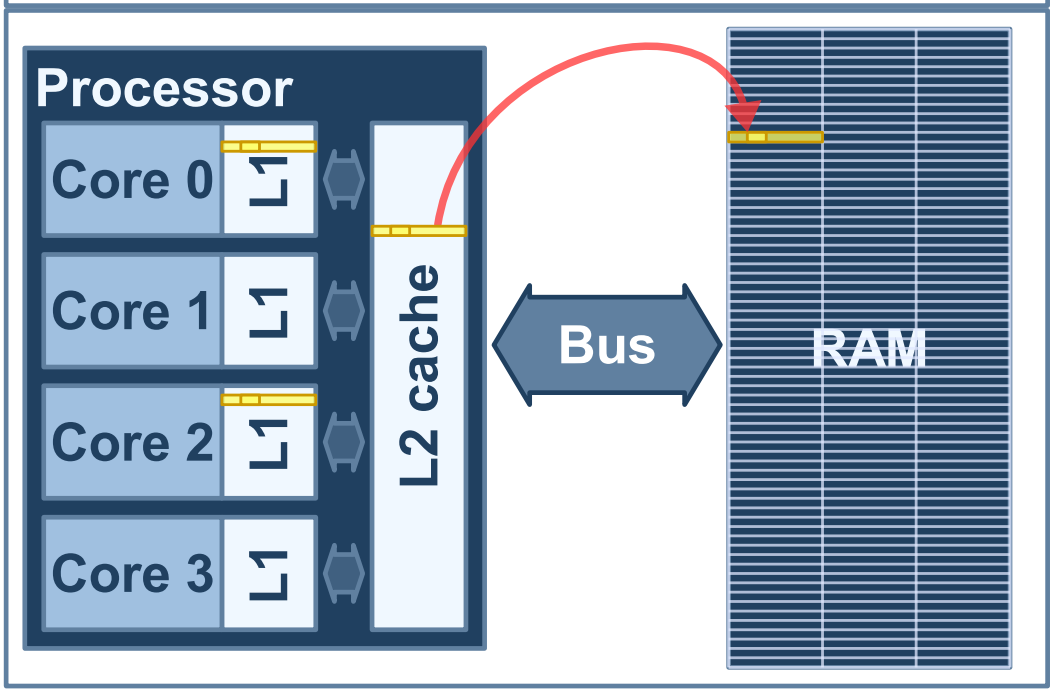


Los caches se tienen que sincronizar. d[1] del cache L1 del core 2 se copia al cache L2 y al cache L1 del core 0.

Acceso al cache con múltiples cores



El core 0 está listo para trabajar con d[1]. **El core 0 se desbloquea**.



Posteriormente, cuando L2 se llene, el valor de d[1] se actualiza a la RAM.

False sharing

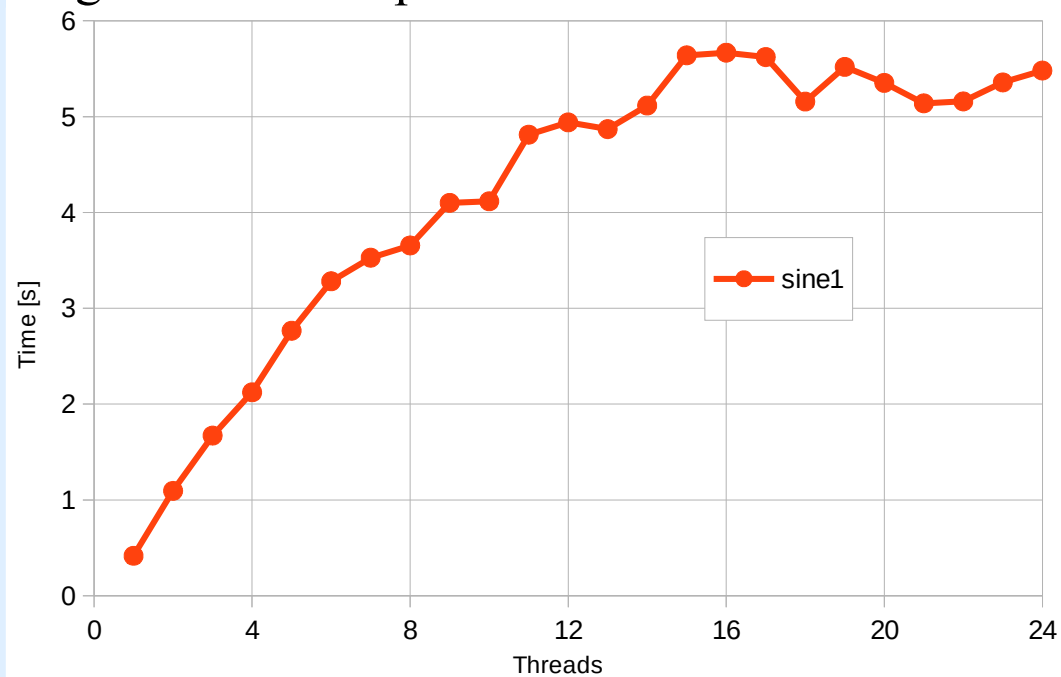
¿Por qué este código es mucho más lento en paralelo que en serie?

```
// sine1.cpp
int n = omp_get_max_threads();
double* s = new double[n];
double* r = new double[n];

for (int t = 0; t < n; ++t)
    r[t] = rand()*3.141592654/RAND_MAX;

#pragma omp parallel
{
    int t = omp_get_thread_num();
    double x = r[t];

    s[t] = x;
    double term = x;
    for (int i = 0, j = 2; i < STEPS; ++i, j += 2)
    {
        term *= -x*x/j/(j + 1);
        s[t] += term;
    }
}

for (int t = 0; t < n; ++t)
    printf("sin(%g) = %g\n", r[t], s[t]);

delete [] r;
delete [] s;
```

Este código calcula el seno de n valores elegidos al azar, utilizando la fórmula

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1}.$$

La gráfica de tiempo vs. número de threads es:

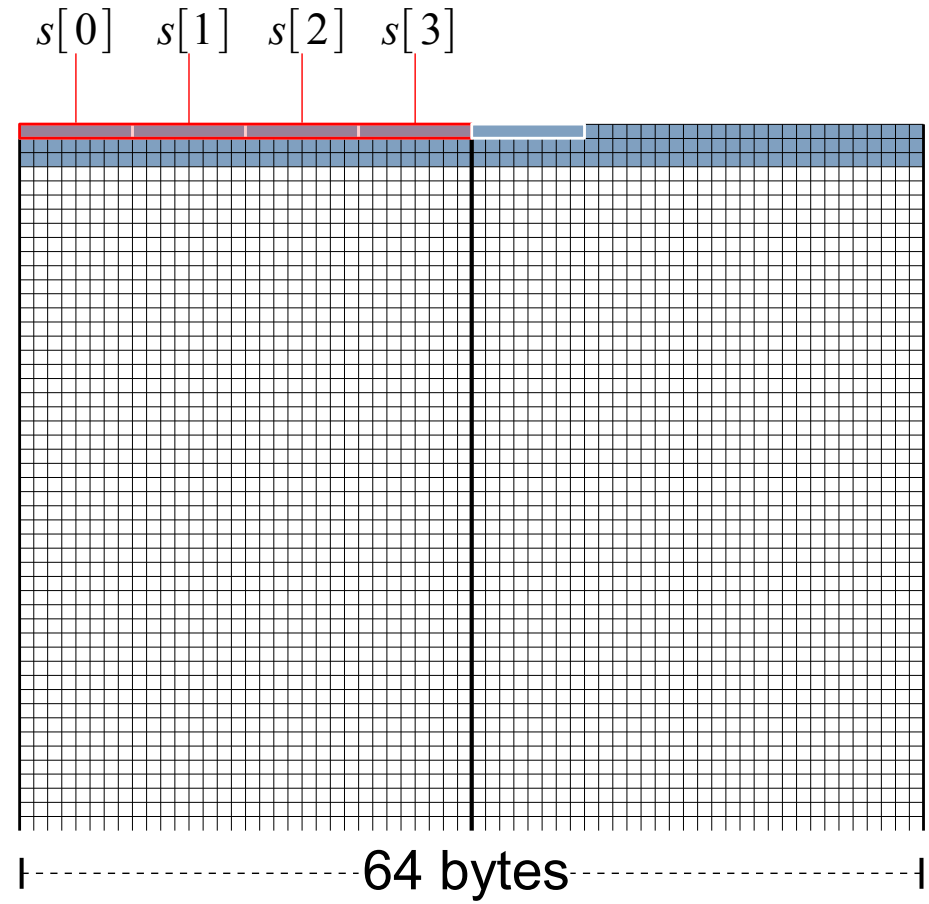


El resultado es correcto no hay dependencia en las variables.

El problema es una línea del código.

```
// sine1.cpp
int n = omp_get_max_threads();
double* s = new double[n];
double* r = new double[n];

for (int t = 0; t < n; ++t)
    r[t] = rand()*3.141592654/RAND_MAX;

#pragma omp parallel
{
    int t = omp_get_thread_num();
    double x = r[t];

    s[t] = x;
    double term = x;
    for (int i = 0, j = 2; i < STEPS; ++i, j += 2)
    {
        term *= -x*x/j/(j + 1);
        s[t] += term;
    }
}

for (int t = 0; t < n; ++t)
    printf("sin(%g) = %g\n", r[t], s[t]);

delete [] r;
delete [] s;
```

El problema es que s[0], s[1], ... comparten una línea de cache, al accesarse en paralelo se generan **cache-faults**.



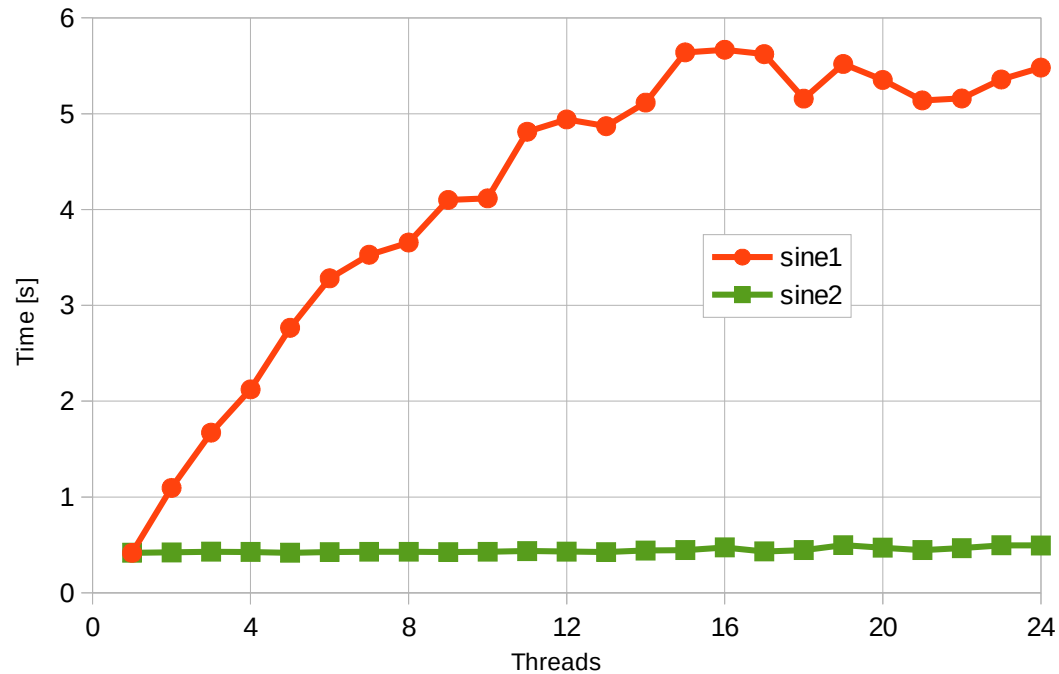
Los **cache-faults** hacen que los caches de los cores tengan que sincronizarse y demorar el procesamiento. A este fenómeno se le llama **false-sharing**.

El resultado es que el código en paralelo es mucho más lento que el código en serie.

Con un pequeño cambio los tiempos mejoran significativamente. Dando el resultado esperado, que el tiempo de ejecución fuese siempre el mismo.

```
// sine2.cpp
int n = omp_get_max_threads();
double* s = new double[n];
double* r = new double[n];

for (int t = 0; t < n; ++t)
    r[t] = rand()*3.141592654/RAND_MAX;

#pragma omp parallel
{
    int t = omp_get_thread_num();
    double x = r[t];

    double sum = x;
    double term = x;
    for (int i = 0, j = 2; i < STEPS; ++i, j += 2)
    {
        term *= -x*x/j/(j + 1);
        sum += term;
    }
    s[t] = sum;
}

for (int t = 0; t < n; ++t)
    printf("sin(%g) = %g\n", r[t], s[t]);

delete [] r;
delete [] s;
```

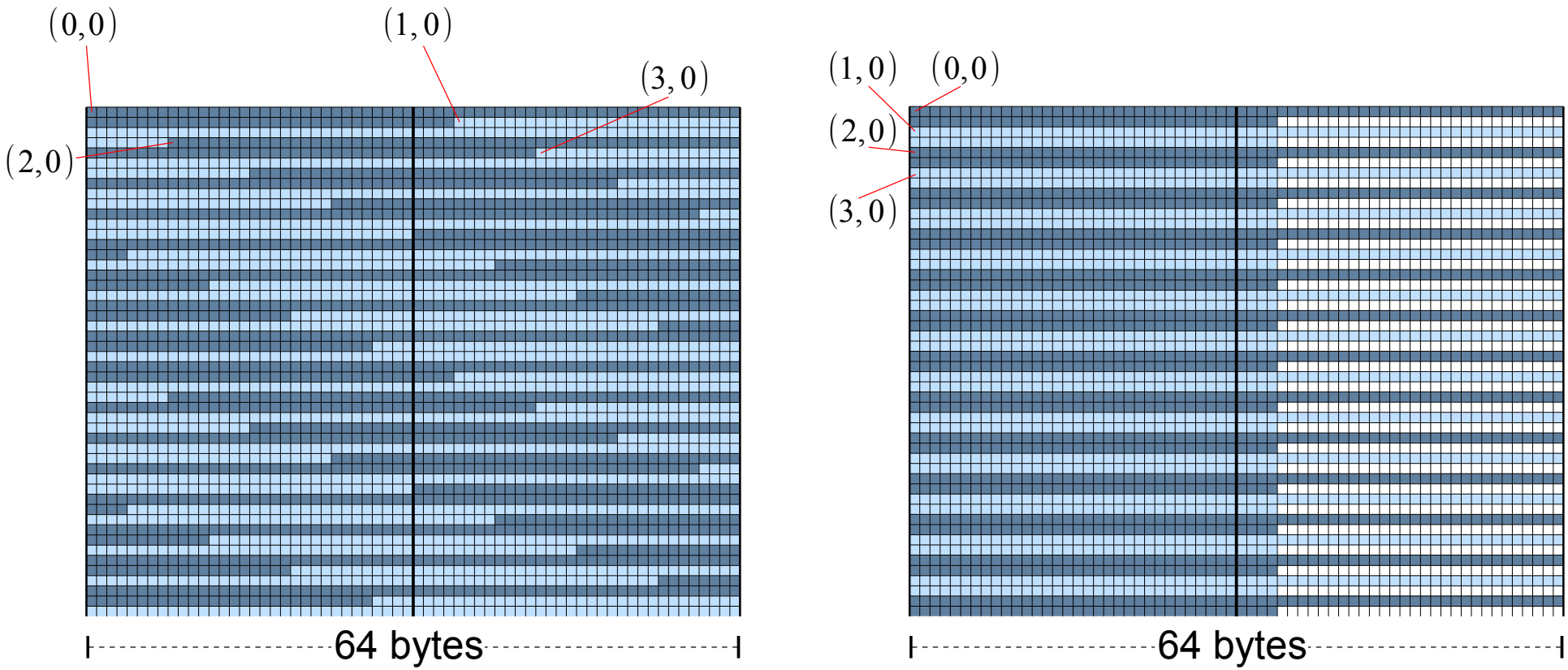
La gráfica de tiempo vs. número de threads es:



Estrategia de programación para aprovechar el cache en paralelo

Las líneas de cache hacen referencia a bloques en la memoria, las direcciones de inicio están en múltiplos del tamaño de la memoria cache.

Las líneas de caché de 32 bytes podrán ser mapeadas a direcciones en memoria 0x00000000, 0x00000020, 0x00000040, 0x00000060, etc. Podemos optimizar alineando el inicio de cada renglón de una matríz.



Así, si el cache de un core accesa un renglón, éste no interferirá con otro core que accese otro renglón.

La función para reservar memoria alineada:

Sistema operativo	Include	Reservar	Liberar
Windows	malloc.h	_aligned_malloc	_aligned_free
OS X	malloc/malloc.h	posix_memalign	free
Linux	malloc.h	posix_memalign	free
FreeBSD	malloc_np.h	posix_memalign	free

El operador **new** de C++ no tiene por default soporte para reservar memoria alineada. Una opción es redefinir los operadores **new**/**delete**.

```
#define ALLOCATION_ALIGN 16

void* operator new (size_t size) throw()
{
	void* memory;
	return posix_memalign(&memory, ALLOCATION_ALIGN, size) ? (void*)0 : memory;
}

void* operator new [] (size_t size) throw()
{
	void* memory;
	return posix_memalign(&memory, ALLOCATION_ALIGN, size) ? (void*)0 : memory;
}

void operator delete (void* object) throw()
{
	free(object);
}

void operator delete [] (void* object) throw()
{
	free(object);
}
```

Acerca de rand()

Qué sucede si ejecutamos...

```
// rand.cpp
int n = omp_get_max_threads();
double* r = new double[n];

#pragma omp parallel for
for (int t = 0; t < n; ++t)
    r[t] = rand()*3.141592654/RAND_MAX;

printf("Parallel rand()\n");
for (int t = 0; t < n; ++t)
    printf("r(%i) = %g\n", t, r[t]);

for (int t = 0; t < n; ++t)
    r[t] = rand()*3.141592654/RAND_MAX;

printf("Serial rand()\n");
for (int t = 0; t < n; ++t)
    printf("r(%i) = %g\n", t, r[t]);

delete [] r;
```

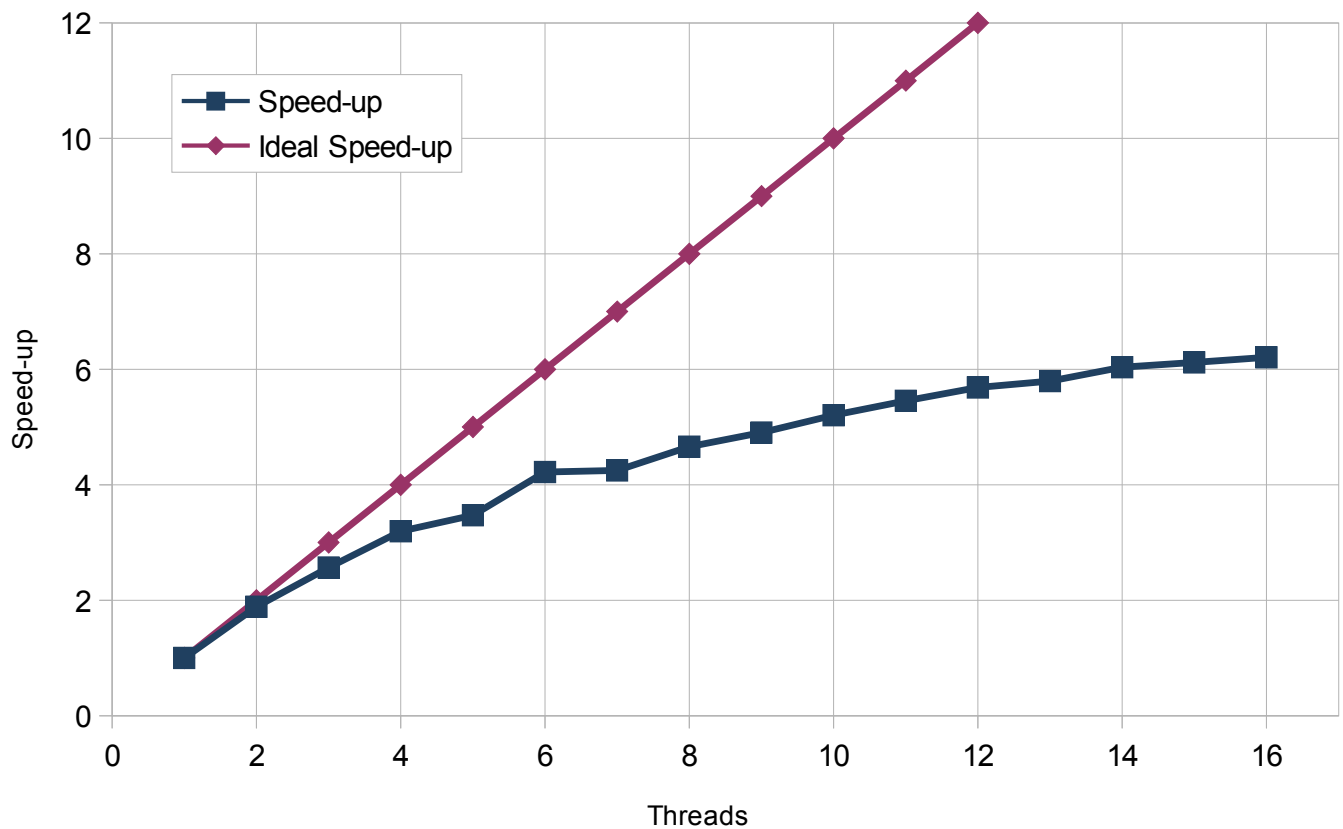
La función rand() no es **thread-safe**. Es decir que no está diseñada para ejecutarse en paralelo. Hay muchas funciones de muchas librerías que no son **thread-safe**. Tienen que revisar todas las funciones que quieran ejecutar en paralelo.

Speed-up

Sea t_n el *tiempo* que tarda en resolverse un problema con n threads/cores.

El speed-up de un programa en paralelo es cuántas veces más rápido puede ser el programa al aumentar el número de threads/cores,

$$\text{speed-up} = \frac{t_1}{t_n}.$$



Ley de Amdahl

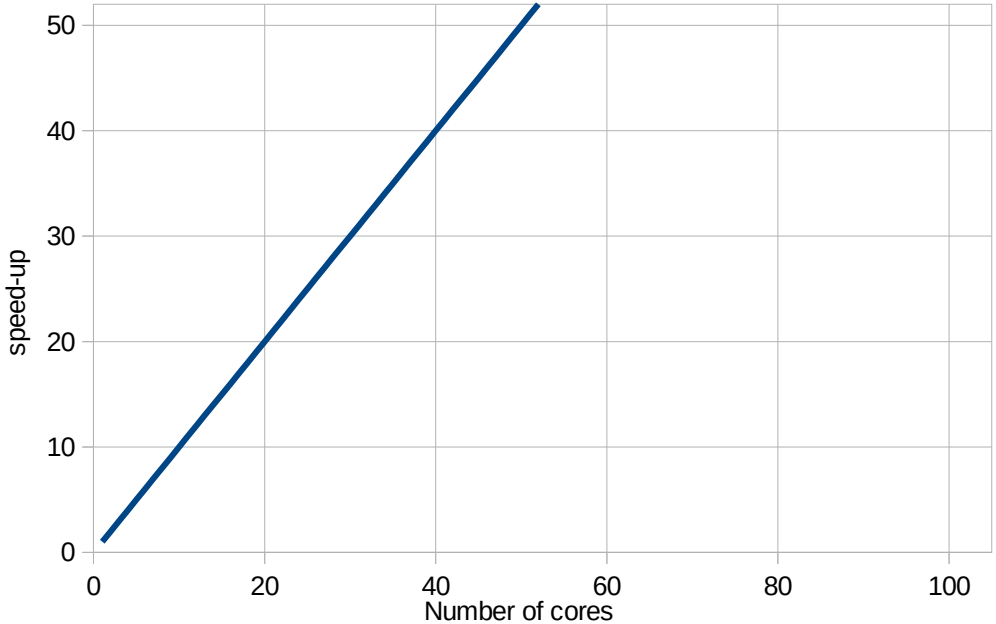
La ley de Amdahl [Amda67] dice que hay un límite en el speed-up para un código que ha sido paralelizado. Este límite tiene que ver con la proporción entre la cantidad de código que has sido paralelizado con la cantidad de código que sigue en serie.

Sean r_s y r_p las proporciones de código en serie y paralelo, de tal forma que $r_s + r_p = 1$. El speed-up (normalizado, $t_1 = 1$) para n threads/cores es

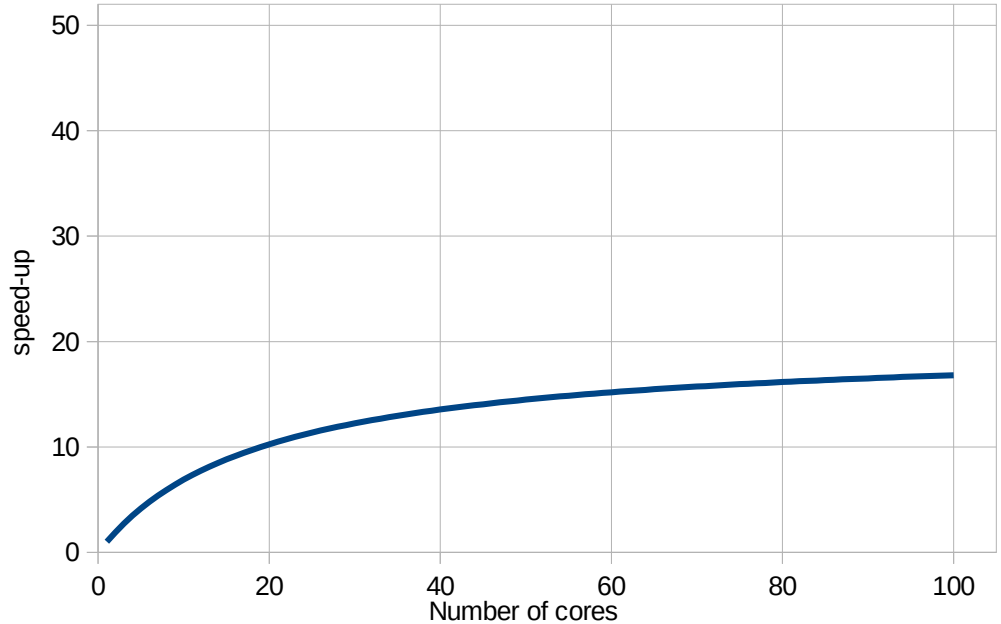
$$\text{speed-up} = \frac{1}{r_s + \frac{r_p}{n}}.$$

Si se incrementa el número de cores, podemos encontrar un límite para el speed-up,

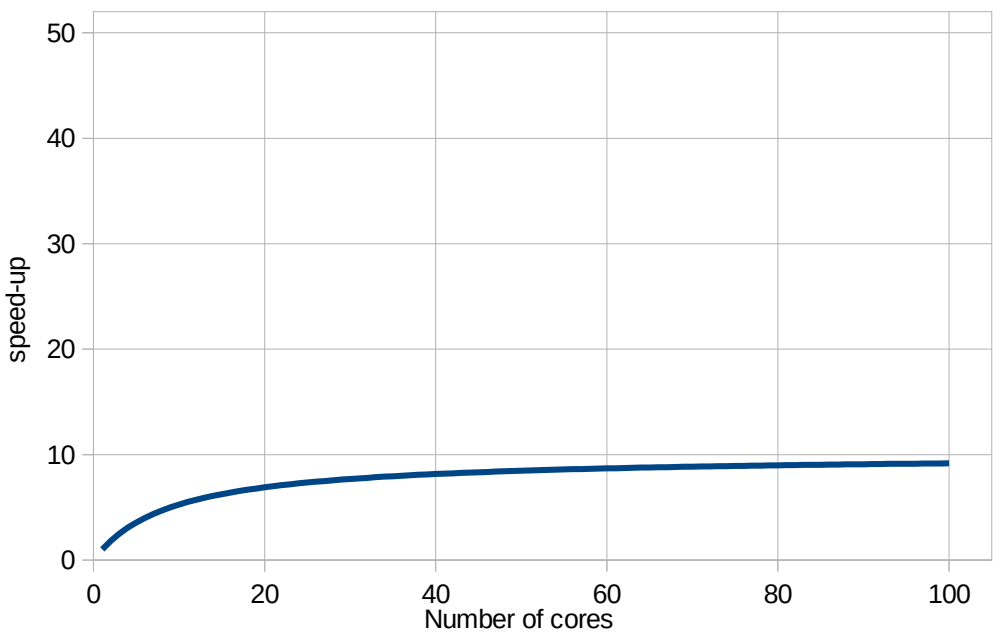
$$\text{speed-up}_{\max} = \lim_{n \to \infty} \frac{1}{r_s + \frac{r_p}{n}} = \frac{1}{r_s}.$$



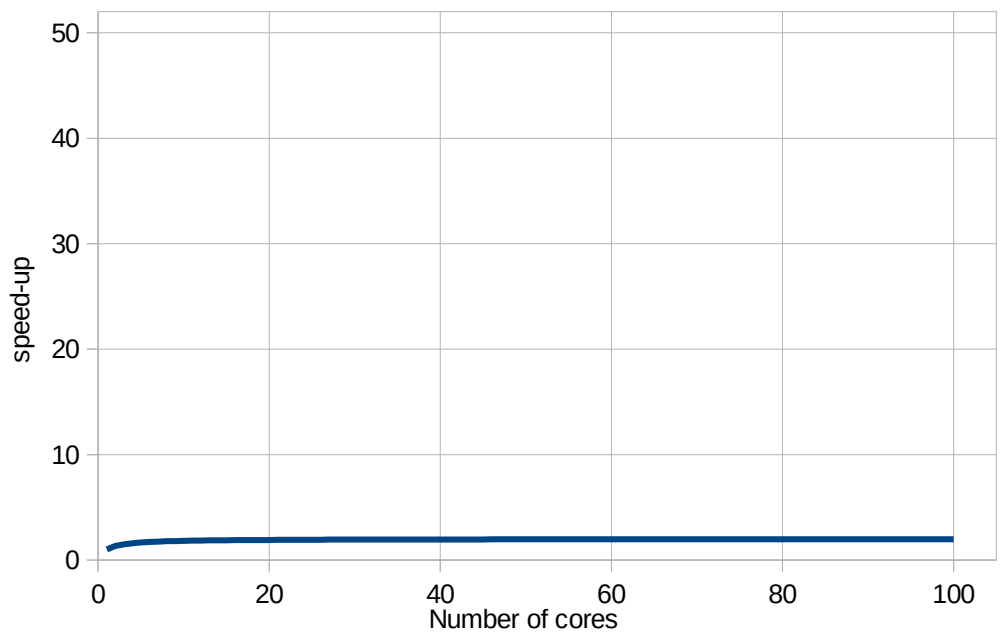
$r_s = 0,\ r_p = 1$



$r_s = 0.05,\ r_p = 0.95,\ \text{speed-up}_{max} = 20$



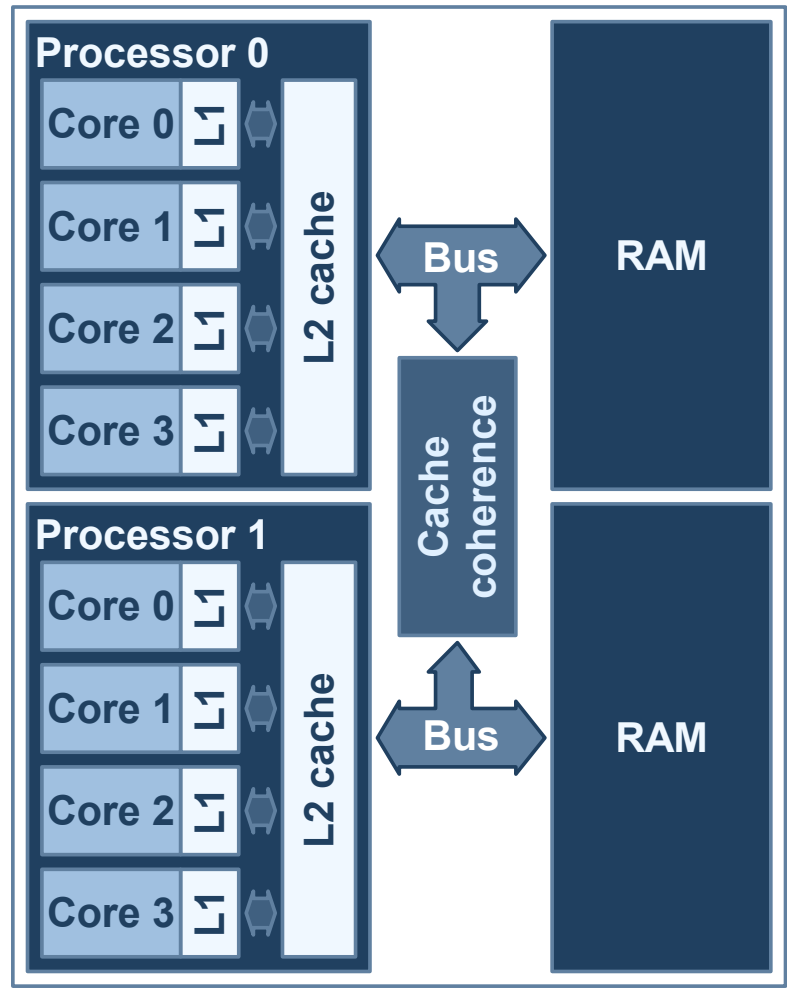
$r_s = 0.1,\ r_p = 0.9,\ \text{speed-up}_{max} = 10$



$r_s = 0.5,\ r_p = 0.5,\ \text{speed-up}_{max} = 2$

Hace 10+ años

Aparecen sistemas con varios procesadores multi-core. Están diseñados para que cada procesador accese a su propio banco de memoria de forma rápida (NUMA, non-uniform memory access).



Accesar al banco de memoria del otro procesador es más lento. Ahora se requieren de circuítos que mantengan la coherencia del cache entre los cores de todos los procesadores (ccNUMA).

Cuál es la lección

Es importante conocer que existen niveles de memoria (RAM, cache L2, cache L1, registros) para diseñar código que sea eficiente:

El acceso a la memoria es el mayor cuello de botella.

Lo más es esencial para el cómputo de alto rendimiento, particularmente al paralelizar:

Optimizar primero la estructura de los datos luego el código.

No todos los algoritmos se pueden paralelizar bien, algunos requieren rediseñarse por completo, pero algo que es fundamental es que:

No se puede tener código en paralelo eficiente sin antes haber optimizado el código en serie.

¿Preguntas?

miguelvargas@cimat.mx

Referencias

[Amda67] G. M. Amdahl. Validiy of the single processor approach to achieving large scale computing capabilities. AFIPS spring joint computer conference. 1967.

[Chap08] B. Chapman, G. Jost, R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, 2008.

[Drep07] U. Drepper. What Every Programmer Should Know About Memory. Red Hat, Inc. 2007.