

Seminario 0: Atcgrid y TORQUE

• Cluster de prácticas (atcgrid):

1. Componentes:

- Switch: SMC8508T
- Tres nodos de cómputo.
- Cables

2. Placa madre:

- Supermicro X8DTL-i Dual Socket 1366 ATX Server Mainboard Intel 5520 chipset
- Intel 5520 chipset.

3. Chip de procesamiento:

- 6 cores hyperthreading 2.4 GHz
- Intel Xeon E5645 (6 cores / 12 threads, 12M L3 Cache compartida, 2.40 GHz cada core, 5.86 GT/s Intel QPI).
- 3 servidores (atcgrid1, atcgrid2, atcgrid3) = nodos de cómputo

Cada core físico tiene dos cores lógicos
Core físico: cores dentro del memory controller
Core lógico: hebras ejecutadas por cada core físico.

4. Acceso.

Cada usuario tiene un home en el nodo front-end de atcgrid al que accederá:

Para ejecutar comandos con un cliente ssh:

```
ssh username@atcgrid.ugr.es.
```

Para cargar y descargar ficheros con un cliente sftp:

```
sftp username@atcgrid.ugr.es
```

● Sistema de colas TORQUE.

- Gestor de colas y de recursos distribuidos basado en PBS (Portal Batch System)

- Comandos TORQUE a usar:

× pbsnodes: información de nodos PBS.

× qsub: enviar un trabajo a ejecutar.

- Devuelve dos ficheros:

→ STDOUT (salida programa): fichero con extensión que comienza por .o

→ STDERR (Errores): fichero con extensión que comienza por .e.

× qstat: estado de los trabajos.

× qdel: eliminar un trabajo.

- Se ejecutarán en el front-end con conexión ssh.

echo 'hello' qsub -q ac echo './hello' qsub -q ac	Envía a ejecutar el trabajo, formado por el ejecutable "hello", por la cola "ac".
qsub script.sh -q ac	Envía a ejecutar el trabajo, formado por el script "script.sh", por la cola "ac".
echo 'lscpu' qsub -q ac	Envía a ejecutar el comando "lscpu" a través de la cola "ac". Devuelve en el fichero de salida la información de la CPU del nodo al que se ha enviado el trabajo.
qstat	Muestra todos los trabajos que se están ejecutando y los que están encolados en todas las colas.
qstat -n -u username	Muestra todos los trabajos del usuario "username" y los nodos asignados a cada trabajo (-n)
qdel jobid	Elimina el trabajo con identificador "jobid"
pbsnodes [-a]	Lista los atributos de todos los nodos.

● Ejemplo hello OpenMP en atcgrid usando TORQUE

```
#include <stdio.h>
#include <omp.h>

int main(void){
    #pragma omp parallel
    printf (" (%0d: !!Hello world!!) ",
            omp-get-thread-num());

    return(0);
}
```

Cada thread imprime su identificador. El identificador se obtiene con la función OpenMP `omp-get-thread-num()`.

Ejecución de hello en atcgrid sin script:

1. Conectar con ssh y sftp al front-end desde PC local.
 - (1) SSH → Ejecuta en un terminal ssh (introduce pw), se crea el directorio 'hello' con mkdir, y se pasa a dicho directorio con cd.
 - (2) SFTP → Ejecuta en otro terminal sftp (introduce pw) y pasa al directorio 'hello' con cd.
2. Generar el ejecutable en PC local y transferirlo al front-end.
 - (3) Genera ejecutable en PC local → Lista contenido directorio con ls y genera ejecutable con gcc y lo ejecuta.
(gcc -fopenmp -O2 -o HelloOMP HelloOMP.c)
(./HelloOMP)
 - (4) Transferencia de PC local a front-end → Lista con llis el directorio actual en PC local y cambia de directorio con lcd en PC local. Tráfiere con put el ejecutable generado en 3 de PC local a front-end.
Lista con ls el directorio actual en front-end.
- (5) ls.

3. Encolar el ejecutable en la cola ac para su ejecución en los servidores.

(6) Ejecución del ejecutable HelloOMP en los servidores →

Usa `qsub` para enviar a ejecutar "HelloOMP" a los servidores a través de la cola "ac".

Usa `qstat` para ver el estado del trabajo enviado a ejecución

Lista con `ls` el contenido del directorio actual antes y después de la ejecución.

4. Copiar el fichero con resultados desde el front-end al PC local.

(7) Lista el directorio actual en PC local antes de ejecutar `get` en la ventana de `sftp`

(8) Transferencia de resultados de front-end a PC local →
Transfiere con `get` el fichero con la salida de la ejecución de front-end a PC local. → `get STDIN.0195`

(9) Lista el directorio actual en PC local después de ejecutar `get` en la ventana de `sftp`

5. Visualizar el fichero de salida en PC local.

(10) Visualización en PC local del contenido del fichero de salida. → Visualiza en PC local con `cat` el fichero con la salida de la ejecución (STDIN.0195) que se transfirió previamente desde front-end a PC local.
Ejecuta en PC local "HelloOMP" para comparar la salida de PC local y atcgrid.

6. Eliminar el fichero de salida y el de error en front-end.

(11) Borrado de ficheros devueltos por `qsub` en front-

Elimina en front-end con `rm` los dos ficheros generados por el comando `qsub` (STDIN.0195 | STDIN.e195).

Ejecución de hello en atcgrid con script.

1. Transferencia del script desde PC local al front-end usando sftp.
2. Ejecución del script en los servidores del cluster con TORQUE
3. Transferencia del fichero de salida desde front-end a PC local usando sftp.
4. Visualización en PC local del contenido del fichero de salida.

Seminario 1: Directivas

OMP_NUM_THREADS → Hebras que quiero que entren

↳ Si no se indica, nº hebras = nº cores.

Compilación: gcc -fopenmp #ifdef
g++ -fopenmp include <omp.h>

OMP_GET_THREAD_NUM → identificador de la hebra.

OMP_GET_NUM_THREADS → nº hebras.

DIRECTIVAS:

Con barrera implícita: parallel, worksharing, single, do, for.

1. Parallel

- Especifica que cláusulas se harán en paralelo.
- Un thread master crea un conjunto de threads cuando alcanza una directiva parallel.
- Cada thread ejecuta el código incluido en la región que conforma el bloque estructurado.
- No reparte tareas entre threads.

Para distribuir las iteraciones de un bucle entre las threads:
#pragma omp for.

Para distribuir trozos de código independientes entre las threads:
#pragma omp sections.

Para que uno de los threads ejecute un trozo de código secuencial:
#pragma omp single.

2. DO/FOR

- Sincronización: barrera implícita al final
- Características de los bucles:
 - Se tiene que conocer el nº de iteraciones, la variable de iteración debe ser un entero.
 - No puede ser de tipo do-while.
 - Las iteraciones se deben poder paralelizar.
- Cada thread hace una i

3. Sections.

- Sincronización: barrera implícita al final, no al principio.
- Cada hebra ejecuta un sections distinto
- El nº de threads que ejecutan trabajo dentro de un sections coincide con el número de sections. → NO

4. Single

- Ejecución de un trozo secuencial por un thread.
- Barrera implícita al final
- Cualquiera de los threads puede ejecutar el trabajo del bloque estructurado.

5. Barrier → lo usas cuando el valor a imprimir depende de toda las hebras

- Barrera: punto en el código en el que los threads se esperan entre sí.
- Al final del parallel y de las construcciones de trabajo compartido hay una barrera implícita.

6. Critical & Atomic.

- Evita que varios threads accedan a variables compartidas a la vez.
- Sección crítica: código que accede a variables compartidas.
- Atomic más eficiente

7. Master: no tiene barrera.

¿Cuántos threads se usan?

1. Fijado por el usuario modificando la variable de entorno

OMP_NUM_THREADS

2. Fijado por la implementación: nº de cpu.

Seminario 2 : Cláusulas

- Cláusulas: Ajustan el comportamiento de las directivas.
- Directivas con cláusula: Parallel, DoIfor, sections, single.
- Directivas sin cláusula: Master, critical, barrier, atomic.
- Regla general para regiones paralelas:
 1. Las variables declaradas fuera de una región y las dinámicas son compartidas por las threads de la región.
 2. Las variables declaradas dentro son privadas.
- Excepciones:
 1. Índice de bucles for: ámbito predeterminado de privado.
 2. Variables declaradas static.

CLÁUSULAS

1. Private: Cuando entra en el parallel es como si se creara de nuevo, dicha variable contiene basura independiente de su declaración previa.
Debemos inicializar su valor dentro de la sección.
2. Lastprivate: La variable toma el valor que se obtiene tras ejecutar el código secuencial.

Devuelve el valor de lo que haya hecho la última hebra en la última iteración.

Si hay un "sections", el valor que tenga tras la última sección.
3. Firstprivate: Para que el valor no contenga basura al inicializarse, "firstprivate" toma el valor que tenía en la declaración previa al parallel.

4. Default

Con "none" el programador debe especificar de que tipo es cada variable (compartida, privada, ...)

5. Reduction

reduction (operator: list)

Se combinan las soluciones.

Comunicación colectiva todas a uno.

6. Copyprivate: Solo se puede usar con single.

Sobrecarga de
centro de
la región
paralela

Permite que una variable privada de un thread se copie a las variables privadas del mismo nombre del resto de threads (difusión).
Entra una hebra que inicializa a las demás

7. Single: Solo entra una hebra.

Seminario 3 : entorno


Interacción con el entorno:

- + prioridad : 1. Cláusulas : if, schedule, num-threads
- 2. Funciones del entorno de ejecución: omp-get-num-threads
- 3. Variables de entorno: omp-num-threads.
- 4. Variables de control interna: nthreads-var

Rutinas de entorno:

1. omp-get-thread-num → Devuelve al thread su identificador dentro del grupo de thread.
2. omp-get-num-threads → Obtiene el nº de threads que se están usando en una región paralela
3. omp-get-num-procs() → Nº de procesadores disponibles para el programa en el momento de la ejecución. (cores lógicos)
4. omp-in-parallel() → Devuelve "true" si se llama a la rutina dentro de una región paralela activa.
"False" en caso contrario.

Orden de precedencia para fijar el nº de threads

- 
1. El nº que resulte de evaluar la cláusula **(if.)**
 2. El nº que fija la cláusula `num-threads`.
 3. El nº que fija la función `omp-set-num-threads()`.
 4. El contenido de la variable de entorno `OMP_NUM_THREADS`.
 5. Fijado por defecto por la implementación: normalmente el nº de cores de un nodo, aunque puede variar directamente.

CLÁUSULAS.

1. IF:

No hay ejecución paralela si no se cumple la condición

PRECAUCIÓN: Solo en construcciones con `parallel`

2. Schedule: Permite modificar el reparto de las iteraciones de un bucle `for`. Se le asigna el tipo de reparto de forma obligatoria y la granularidad de forma opcional que es el tamaño del bloque.

El primer parámetro es el tipo. Si no se indica va a ser por defecto `static`.

El reparto es estático: se realiza en tiempo de compilación y tiene menor tiempo de carga.

Si hay más bloques que hebras, se le irán asignando los bloques a las hebras según el identificador de hebra.

2.1.- STATIC

Si no se indica el `chunk` se intenta maximizar el tamaño del bloque para que cada hebra ejecute uno.

Reparto estático: se realiza en tiempo de compilación y tiene menor tiempo de carga.

Se irán asignando los bloques a las hebras según el identificador de hebra.

Si hay más bloques que hebras, se vuelve a asignar según el orden.

Si hay menos bloques que hebras, habrá hebras que no ejecuten nada.

2.2.- DYNAMIC

La distribución se hace en tiempo de ejecución, se añade cierta carga.

El número de iteraciones se divide en bloques.

No va por orden, sino la primera hebra que llegue, se le asigna el bloque.

Si no se especifica el chunk el bloque es de una iter.

2.3.- GUIDED

La distribución se realiza en tiempo de ejecución y los bloques se asignan según la hebra que llegue.

⊗ Se distingue de dynamic en que los bloques no son fijos. El tamaño va disminuyendo conforme el reparto de las hebras y vaya quedando menos iteraciones.

El chunk especifica el tamaño mínimo del bloque

El último reparto puede ser menor que el chunk.

STATIC

50 iter
3 hebras
chunk = 15

hebra1	hebra2	hebra3
15	15	15
5		

DYNAMIC

80 iter
3 hebras
chunk = 15

hebra1	hebra2	hebra3
15	15	15
	15	15
		5

GUIDED

chunk tamaño mínimo de reparto = 15
100 iter

hebra1	hebra2	hebra3	hebra4	hebra5
30	20	15	15	15

2.4.- RUNTIME

Se le indica el tipo de reparto mediante la variable "run-sched-var"

Si no hay nada en la variable: static, 1.

schedule(runtime) → busca en la variable de entorno.

VARIABLES de CONTROL:

1. dyn-var: Control de ajuste dinámico del número de hebras.
Puede tomar valores true o false.
@ No de hebras que se crean en la región paralela que se ajustan dinámicamente según el trabajo.
2. nthreads-var: Variable de control que es un número, se modifica mediante variable de entorno, cláusula o función y se puede consultar con funciones de bibliotecas.
3. thread-limit-var: Controla el número máximo de hebras para todo el programa. (solo consulta)
4. nest-var: Si hay dos construcciones paralelas y la primera está activa, crea un grupo de hebras para las hebras de la segunda.

#parallel { // Si está activo, el primer parallel
#parallel { tiene 3 hebras, en el segundo
 por cada hebra se crean mas.
5. run-sched-var: Permite controlar como se reparte las iteraciones del bucle.
(kind, chunk)
 ↳ Granularidad: No iteraciones que forman el tamaño del bloque.
6. def-sched-var: Indica el tipo de reparto que se va a seguir por defecto.

Seminario 4: Optimización de código

• Cuestiones generales:

- Importante analizar los cuellos de botella

Aquel cuello de botella más estrecho, por el cual el resto de elementos del código se van a ver obligados a pasar y por tanto no vamos a conseguir las prestaciones que necesitamos.

- Esperar al final para optimizar, dificulta el proceso
- Mala praxis: eliminar propiedades y funciones del código.
- Se puede optimizar sin tener que acceder al nivel del lenguaje ensamblador.

• Unidades de ejecución:

- División \rightarrow operación costosa.

```
for (i=0; i<100; i++) a[i] = a[i] / y
```



```
temp = 1/y
for (i=0; i<100; i++) a[i] = a[i] * temp
```

- Utilizar desplazamientos:

```
imul    eax, 10
5 * 10 = 50
```



```
lea     ecx, [eax + eax]
lea     eax, [ecx + eax * 8]
```

ecx = 10

10 + 5 * 8 = 50

• Alineamiento de datos

- El acceso a un dato que ocupa dos líneas de caché aumenta tiempo de acceso.
- Hay que alinear nuestro dato
- A más líneas de caché, más tiempo tardamos en tomar el dato.

• Localidad de los accesos

- La forma en la que se declaran los arrays determina la forma en que se almacenan en memoria. Interesa declararlos según la forma en la que se vaya a realizar el acceso.

```
struct {
    int a[500];
    int b[500];
} s;

for (i=0; i<500; i++)
    s.a[i] = 2 * s.a[i];

for (i=0; i<500; i++)
    s.b[i] = 3 * s.b[i];
```

```
struct {
    int a;
    int b;
} s[500];

for (i=0; i<500; i++) {
    s[i].a += 5;
    s[i].b += 3;
}
```

Aquí se guardan los primeros 500 y los segundos 500. Si se hiciese en un bucle, tendríamos que conmutar de una zona a otra → fallo de caché

Organización de memoria.

a	b	a	b	a	b	a	b	...	a	b
1	2	3	4	5					500	

- Intercambiar los bucles para cambiar la forma de acceder a los datos. según los almacena el compilador, y para aprovechar la localidad.

```
for (j=0; j<4000; j++)
    for (i=0; i<4000; i++)
        a[i][j] = 2 * a[i][j];
```

```
for (i=0; i<4000; i++)
    for (j=0; j<4000; j++)
        a[i][j] = 2 * a[i][j];
```

• Desenrollado de bucles.

```
for (i=0; i < ARR; i++) {  
    tmp += a[i] * b[i];  
}
```



```
for (i=0; i < ARR; i+=4) {  
    tmp0 += a[i] * b[i];  
    tmp1 += a[i+1] * b[i+1];  
    tmp2 += a[i+2] * b[i+2];  
    tmp3 += a[i+3] * b[i+3];  
}
```

```
for (i=0; i < 100; i++)  
    if ((i%2 == 0)  
        a[i] = x;  
    else  
        a[i] = y;
```



```
for (i=0; i < 100; i+=2) {  
    a[i] = x;  
    a[i+1] = y;  
}
```

• Reduce el número de saltos

• Código ambiguo.

- Si el compilador no puede resolver los punteros se inhiben ciertas optimizaciones del compilador.
- Si no se utilizan punteros el código es más dependiente de la máquina, y a veces las ventajas de no utilizarlos no compensa.
- Cómo evitar código ambiguo:
 - Utilizar variables locales en lugar de punteros
 - Utilizar variables globales si no se pueden utilizar las locales.
 - Poner las instrucciones de almacenamiento después o bastante antes de las cargas de memoria.

• Saltos.

if (t1 == 0 && t2 == 0 && t3 == 0)

if ((t1 | t2 | t3) == 0) (OR a nivel de bit)

- Cada una de las condiciones separadas por && se evalúa mediante una instrucción de salto distinta.
- Se puede reducir el número de saltos de un programa reorganizando las sentencias switch. (en el caso que una opción se ejecute más que las otras)
- Que la condición que utilices minimice el uso que vas a hacer de la predicción de salto.

switch (i) {

case 16 :
Bloque16
break ;

case 22 :
Bloque22
break ;

case 33 :
Bloque33
break ;

}



if (i == 33)
{ Bloque33 }

else
switch (i) {
case 16 :
Bloque16
break ;

case 22 :
Bloque22
break ;