

Práctica 4 - OpenMP

Arquitectura de computadores

Dpto. Informática - UJA

diciembre 2017

Índice

- 1 Computadores paralelos
 - Sistemas multiprocesador
 - APIs de programación para sistemas multiprocesador
- 2 Introducción a OpenMP
 - Primeros pasos
 - Directivas y cláusulas
 - Funciones y variables de entorno
- 3 OpenMP en la práctica
 - Entorno de trabajo
 - Código fuente
- 4 Ejercicios con OpenMP
 - El "hola mundo" de OpenMP
 - Paralelización de un bucle for
 - Ejecución paralela con reducción de resultados
 - Cómo medir el tiempo de ejecución
 - Ejemplos adicionales y bibliografía

Índice

- 1 Computadores paralelos
 - Sistemas multiprocesador
 - APIs de programación para sistemas multiprocesador
- 2 Introducción a OpenMP
 - Primeros pasos
 - Directivas y cláusulas
 - Funciones y variables de entorno
- 3 OpenMP en la práctica
 - Entorno de trabajo
 - Código fuente
- 4 Ejercicios con OpenMP
 - El "hola mundo" de OpenMP
 - Paralelización de un bucle for
 - Ejecución paralela con reducción de resultados
 - Cómo medir el tiempo de ejecución
 - Ejemplos adicionales y bibliografía

Introducción

- Los sistemas multiprocesador son computadores paralelos de memoria compartida
- Hay un único banco de memoria en el que se almacenan programas y datos
- El sistema cuenta con varios procesadores lógicos independientes que pueden estar alojados en circuitos integrados físicamente separados o bien compartiendo un mismo *die*
- Cada procesador lógico tendrá su propio hilo de ejecución independiente, lo que permite acceder a paralelismo de alto nivel controlado por el sistema y las aplicaciones

Configuraciones multiprocesador

- Los ordenadores de escritorio y portátiles incorporan desde hace años microprocesadores con múltiples núcleos (2, 4, 8 o incluso más)
- Las tabletas y teléfonos móviles de última generación también cuentan con una configuración multiprocesador, si bien con un diseño más eficiente de uso de energía
- En configuraciones de servidor es habitual que el sistema disponga de múltiples zócalos, cada uno alojando un microprocesador con múltiples núcleos
- Algunos núcleos de procesamiento son capaces de ejecutar dos hilos en paralelo (tecnología *Hyper-threading* de Intel y equivalentes)

Múltiples alternativas - (*fork/join*, *threads*)

- A la hora de aprovechar el paralelismo de un sistema multiprocesador existen distintas alternativas:
 - ✓ Una aplicación puede asumir que se ejecuta en un sistema aislado sin multiprocesamiento, dejando que el sistema operativo gestione el reparto de hilos de ejecución entre los procesos que haya en marcha
 - ✓ Un programa puede lanzar procesos secundarios según el patrón *fork/join*, ejecutando en paralelo partes del código
 - ✓ El programa puede crear explícitamente los hilos de ejecución que precise y asignar tareas de procesamiento en paralelo (*threads*, *threads* de Java, etc.)
- La creación y control explícito de hilos de ejecución por parte de un programa es una tarea delicada, al implicar potenciales problemas de sincronización entre hilos y acceso compartido a datos

Múltiples alternativas - (OpenMP)

- OpenMP es una de las vías que nos permite aprovechar el paralelismo en sistemas de memoria compartida
- Está basado en la creación de múltiples hilos de ejecución (*threads*) no en la generación de procesos secundarios (*fork/join*)
- Frente al enfoque imperativo de APIs como *pthread*s o los servicios de *threads* de Java, OpenMP propone un enfoque declarativo
- OpenMP es un estándar respaldado por las más importantes empresas del sector, siendo una solución multiplataforma y multilenguaje

Índice

- 1 Computadores paralelos
 - Sistemas multiprocesador
 - APIs de programación para sistemas multiprocesador
- 2 Introducción a OpenMP
 - Primeros pasos
 - Directivas y cláusulas
 - Funciones y variables de entorno
- 3 OpenMP en la práctica
 - Entorno de trabajo
 - Código fuente
- 4 Ejercicios con OpenMP
 - El "hola mundo" de OpenMP
 - Paralelización de un bucle for
 - Ejecución paralela con reducción de resultados
 - Cómo medir el tiempo de ejecución
 - Ejemplos adicionales y bibliografía

¿Qué es OpenMP?

- OpenMP es una API accesible desde C/C++ y Fortran y disponible para distintas variantes de Unix, GNU/Linux, OS X y Windows
- El programador usa esta API mediante directivas del compilador, en el caso de C/C++ la directiva `pragma`
- Al compilar un programa que usa OpenMP es necesario indicar al compilador que incluya la biblioteca de servicio correspondiente
- Aquí usaremos OpenMP en GNU/Linux con el compilador de C/C++ de la familia de compiladores `gcc` (*GNU Compiler Collection*)

Sintaxis de las directivas OpenMP

Las directivas OpenMP para C/C++ se ajustan a la siguiente sintaxis

```
#pragma omp directiva [par1 [par2 ...]]
```

Listado 1: Sintaxis directivas OpenMP

La *directiva* que sigue a `omp` afecta exclusivamente a la siguiente sentencia del código del programa, por lo que si queremos aplicarla a un bloque es necesario delimitar este entre llaves

```
#pragma omp parallel
{
    // Bloque de sentencias a ejecutar en paralelo
}
```

Listado 2: Directiva aplicada a un bloque de código

Sintaxis de las directivas OpenMP

La directiva puede ir seguida de una o más cláusulas, cuyo fin es establecer parámetros de configuración adicionales

```
#pragma omp parallel shared(variable1) private(variable2)
{
    // Sentencias
}
```

Listado 3: Cláusulas en una directiva

En este ejemplo se usan dos cláusulas para indicar qué variables serán compartidas por los hilos de ejecución y cuáles contarán con una copia privada para cada hilo

En <http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf> se ofrece una referencia completa de directivas, cláusulas, funciones y variables de entorno

Directivas OpenMP - Secciones paralelas/secuenciales

- `parallel`: Inicia un bloque que será ejecutado en paralelo por múltiples hilos
- `for`: Se asocia a bucles de tipo `for` a fin de ejecutar sus iteraciones en paralelo en lugar de secuencialmente
- `sections`: Identifica secciones del código a asignar a cada hilo de ejecución
- `single`: Establece que un bloque de código ha de ser ejecutado exclusivamente por un hilo (no necesariamente el hilo maestro, creado al inicio del programa)
- `ordered`: Se asocia a bucles de tipo `for` que se encuentran en un bloque `parallel` con el objetivo de que se ejecuten secuencialmente

Directivas OpenMP - Sincronización/Control de acceso

- `atomic`: Se asocia a una expresión que accede a memoria a fin de garantizar que su ejecución sea atómica, evitando que varios hilos puedan escribir en ella simultáneamente
- `barrier`: Establece una barrera en la que se esperará a la finalización de todos los hilos de ejecución antes de continuar
- `critical`: Restringe la ejecución de un bloque de código a un solo hilo en cada momento, sin preferencia por uno en particular
- `master`: Restringe la ejecución de un bloque de código exclusivamente al hilo maestro, el que pone en marcha la aplicación
- `taskwait`: Sincroniza una tarea con la finalización de otra

Cláusulas OpenMP

- `shared`: Establece las variables que serán compartidas por los distintos hilos al ejecutar un bloque paralelo
- `private`: Establece las variables de las que se creará una copia independiente para cada hilo al ejecutar un bloque paralelo
- `reduction`: Indica que se aplicará una operación de reducción sobre las variables privadas de cada hilo al final de la ejecución
- `firstprivate`: Similar a `private`, en cuanto fija que cada hilo contará con una copia de las variables indicadas, pero además inicializa cada copia con el valor que tenga originalmente cada variable antes de entrar en bloque de ejecución paralela
- `lastprivate`: Declara que cada hilo de un bloque paralelo contará con una copia de una variable y, además, que la versión de esta externa al bloque tomará el valor que tenga el último hilo de ejecución
- `num_threads`: Indica el número de hilos de ejecución que se usará para procesar un bloque

Funciones de OpenMP

- `omp_get_num_procs()`: Devuelve el número de procesadores disponibles en el actual entorno de ejecución
- `omp_get_num_threads()`: Devuelve el número de hilos de ejecución en el bloque paralelo actual
- `omp_set_num_threads()`: Establece el número de hilos para bloques posteriores en los que no se indique explícitamente este parámetro
- `omp_get_thread_num()`: Devuelve el identificador del hilo que está ejecutando la sentencia actual
- `omp_in_parallel()`: Devuelve `true` si la sentencia desde la que se invoca a la función está ejecutándose en un bloque paralelo

Asimismo hay disponibles múltiples funciones para tareas de sincronización, estableciendo y comprobando bloqueos.

Variables de entorno

Mediante las variables de entorno de OpenMP se facilita la configuración de determinados parámetros sin necesidad de modificar el código del programa. Las de uso más habitual son las siguientes:

- `OMP_NUM_THREADS`: Establece el número de hilos que se asignarán a un bloque paralelo en el que no se especifique explícitamente este parámetro, al igual que la función `omp_set_num_threads()`
- `OMP_NESTED`: Puede tomar los valores `TRUE` y `FALSE`, indicando si se permite o no el anidamiento de bloques paralelos. Se puede modificar con la función `omp_set_nested()`
- `OMP_DYNAMIC`: Puede tomar los valores `TRUE` y `FALSE`, indicando si se permite el ajuste dinámico del número de hilos dentro de un bloque paralelo

Índice

- 1 Computadores paralelos
 - Sistemas multiprocesador
 - APIs de programación para sistemas multiprocesador
- 2 Introducción a OpenMP
 - Primeros pasos
 - Directivas y cláusulas
 - Funciones y variables de entorno
- 3 OpenMP en la práctica
 - Entorno de trabajo
 - Código fuente
- 4 Ejercicios con OpenMP
 - El "hola mundo" de OpenMP
 - Paralelización de un bucle for
 - Ejecución paralela con reducción de resultados
 - Cómo medir el tiempo de ejecución
 - Ejemplos adicionales y bibliografía

Sistema y compilador

- OpenMP es un estándar y su uso está contemplado por distintos compiladores en múltiples sistemas operativos, incluyendo Windows, OS X y GNU/Linux
- Los compiladores de Intel y Microsoft (Visual C++ 2015) para Windows soportan OpenMP, al igual que la familia de compiladores `gcc` para varios sistemas
- Aquí usaremos el compilador de C/C++ de la familia `gcc`. Si trabajamos en OS X o GNU/Linux probablemente ya lo tengamos instalado

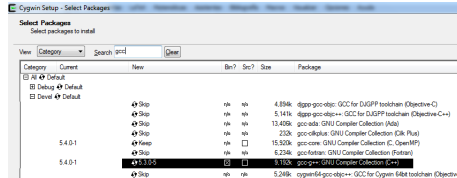
Podemos recurrir al gestor de paquetes correspondiente para instalarlo, en caso necesario

- En Windows es recomendable instalar Cygwin a fin de contar con el mismo entorno de trabajo que tendríamos en GNU/Linux. También podemos usar la consola **Bash en Ubuntu** de Windows 10.

Instalación de gcc en Windows

Si queremos trabajar en Windows pero usando gcc como haríamos en GNU/Linux, podemos configurar nuestro equipo como se detalla a continuación:

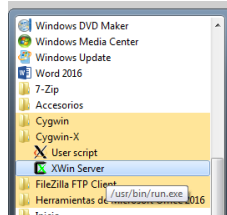
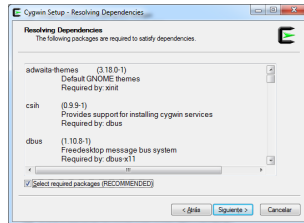
- 1 Descargamos el instalador de Cygwin de <https://cygwin.com/install.html> y lo ejecutamos (escogemos la versión de 32/64 bits según la edición de Windows que usemos)
- 2 El instalador nos permitirá seleccionar los paquetes que queremos descargar e instalar en el sistema. Nos aseguramos de seleccionar el compilador gcc-g++, como se aprecia en la figura superior derecha



- 3 Si queremos contar con un entorno gráfico, deberemos seleccionar también el paquete xinit. Asimismo podemos seleccionar gedit para contar con un editor básico

Instalación de gcc en Windows

- 4 Hacemos clic en **Siguiente** para avanzar y confirmamos (figura superior derecha) que queremos instalar todos los paquetes requeridos por los que hemos elegido previamente
- 5 Cuando la instalación haya terminado abrimos el menú del botón **Inicio** y seleccionamos la opción **XWin Server** de la carpeta **Cygwin-X**, como se aprecia en la figura inferior derecha
- 6 En la Barra de tareas de Windows aparecerá un icono desde el que podemos abrir una ventana de terminal y otros accesorios, incluyendo gedit



Bash en Ubuntu en Windows 10

Si usamos Windows 10 como sistema operativo, podemos activar la consola de Bash Ubuntu y trabajar como lo haríamos en GNU/Linux. Para ello:

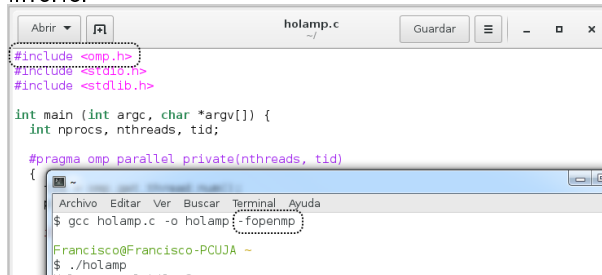
- 1 Ir a **Configuración**, abrir la página **Para programadores** y activar la opción **Modo programador**
- 2 Abrir el **Panel de control**, seleccionar la opción **Activar o desactivar las características de Windows** y activar el elemento **Subsistema de Windows para Linux**
- 3 Abrir el menú **Inicio**, escribir `bash` y elegir la opción **Bash en Ubuntu en Windows**

La consola¹ que se abre es análoga al terminal de Ubuntu y trabajaríamos de igual manera.

¹La ruta al sistema de archivos de Bash Ubuntu es `%localappdata%\lxss`.

Opciones de compilación

Para compilar un programa en el que va a utilizarse OpenMP, asumiendo que el compilador es `gcc`, debemos agregar la opción `-fopenmp`, tal y como se muestra en la consola de la imagen inferior



The image shows a code editor window titled 'holamp.c' with the following code:

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

int main (int argc, char *argv[]) {
    int nprocs, nthreads, tid;

    #pragma omp parallel private(nthreads, tid)
    {
```

Below the code editor is a terminal window showing the compilation command:

```
$ gcc holamp.c -o holamp -fopenmp
```

The terminal output shows the user's prompt and the command to run the program:

```
Francisco@Francisco-PCUJA ~
$ ./holamp
```

La citada opción conlleva la inclusión automática de `-pthread`, por lo que es imprescindible tener también instaladas las librerías *pthread*s

Archivo de cabecera

Los prototipos de funciones, definiciones de constantes y otros elementos relativos a OpenMP se encuentran alojados en el archivo de cabecera `omp.h`, de ahí que esta deba incluirse en toda aplicación que vaya a usar los servicios de OpenMP

```
#include <omp.h>
// Otros archivos de cabecera

int main(int argc, char *argv[]) {
    // Programa OpenMP
}
```

Listado 4: Inclusión de archivo de cabecera

Directivas OpenMP

En el código del programa se introducirán las directivas de OpenMP recurriendo a la sintaxis anteriormente descrita. Siempre ha de tenerse en cuenta que la directiva afectará exclusivamente a la siguiente sentencia del programa, por lo que si se quiere configurar la ejecución de un bloque es preciso delimitar este entre llaves

```
#include <omp.h>

int main(int argc, char *argv[]) {
    int nprocs, nthreads, id;
    #pragma omp parallel
    {
        // Bloque de sentencias a ejecutar en paralelo
    }
}
```

Listado 5: Directivas OpenMP y bloques de código

Compilación condicional

Con el objetivo de obtener programas que puedan funcionar tanto con OpenMP como sin esta infraestructura, es recomendable utilizar compilación condicional tal y como se muestra en el siguiente ejemplo:

```
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif
int main(int argc, char *argv[]) {
    int idthread;
    // La directiva se ignorará si no se compiló con -fopenmp
    #pragma omp parallel private(idthread)
    {
        // Tomará el valor 0 si no se ha compilado con OpenMP
        idthread = omp_get_thread_num();
        ...
    }
}
```

Listado 6: Compilación condicional

Índice

- 1 Computadores paralelos
 - Sistemas multiprocesador
 - APIs de programación para sistemas multiprocesador
- 2 Introducción a OpenMP
 - Primeros pasos
 - Directivas y cláusulas
 - Funciones y variables de entorno
- 3 OpenMP en la práctica
 - Entorno de trabajo
 - Código fuente
- 4 Ejercicios con OpenMP
 - El "hola mundo" de OpenMP
 - Paralelización de un bucle for
 - Ejecución paralela con reducción de resultados
 - Cómo medir el tiempo de ejecución
 - Ejemplos adicionales y bibliografía

El "hola mundo" de OpenMP

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

int main (int argc, char *argv[]) {
    int nprocs, nthreads, tid;

    // El siguiente bloque se ejecutará en paralelo
    // En él, cada hilo contará con su copia de nthreads y tid
    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num(); // Obtenemos el nº de thread
        printf("Hola, soy el hilo %d\n", tid);

        if (tid == 0) { // Si es el hilo maestro
            nprocs = omp_get_num_procs();
            nthreads = omp_get_num_threads();
            printf("Hay un total de %d procesadores y %d hilos\n",
                nprocs, nthreads);
        }
    }
}
```

El "hola mundo" de OpenMP

Las sentencias que hay dentro del bloque son ejecutadas en paralelo por tantos hilos como disponga el sistema

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda

Francisco@Francisco-PCUJA ~
$ gcc holamp.c -o holamp -fopenmp

Francisco@Francisco-PCUJA ~
$ ./holamp
Hola, soy el hilo 2
Hola, soy el hilo 3
Hola, soy el hilo 1
Hola, soy el hilo 5
Hola, soy el hilo 7
Hola, soy el hilo 4
Hola, soy el hilo 6
Hola, soy el hilo 0
Hay un total de 4 procesadores y 8 hilos ejecutándose

Francisco@Francisco-PCUJA ~
$
```

Usando la cláusula `num_threads(n)` se puede establecer explícitamente el número de hilos que se desea utilizar

Paralelización de un bucle for

```
#define TAM_VECTOR 25

int main(int argc, char *argv[]) {
    int V1[TAM_VECTOR], V2[TAM_VECTOR], Prod[TAM_VECTOR], tid;

    for(int i=0; i < TAM_VECTOR; i++) {
        V1[i] = i << 2;    // Introducimos valores iniciales
        V2[i] = i << 4;    // en cada uno de los vectores
    }

    // Producto de un vector por otro ejecutado en paralelo
    #pragma omp parallel for shared(V1,V2,Prod) private(i,tid)
    for(int i = 0; i < TAM_VECTOR; i++) {
        tid = omp_get_thread_num();
        Prod[i] = V1[i] * V2[i];
        printf("Ciclo %d ejecutado por el hilo %d\n",i, tid);
    }

    for(int i = 0; i < TAM_VECTOR; i++)
        printf("%d X %d = %d\n", V1[i], V2[i], Prod[i]);
}
```

Paralelización de un bucle for

Cada iteración del bucle se asigna a un hilo de ejecución, dividiéndose el número total de ciclos entre el número de *threads* disponibles

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
Iteración 15 ejecutada por el hilo 4
Iteración 21 ejecutada por el hilo 6
Iteración 9 ejecutada por el hilo 2
Iteración 12 ejecutada por el hilo 3
Iteración 18 ejecutada por el hilo 5
Iteración 2 ejecutada por el hilo 0
Iteración 24 ejecutada por el hilo 7
Iteración 3 ejecutada por el hilo 0
0 X 0 = 0
4 X 16 = 64
8 X 32 = 256
12 X 48 = 576
16 X 64 = 1024
20 X 80 = 1600
24 X 96 = 2304
28 X 112 = 3136
32 X 128 = 4096
36 X 144 = 5184
```

Los vectores son compartidos por todos los hilos, pero la variable que identifica el índice del elemento es privada a fin de evitar colisiones entre hilos

Ejecución paralela con reducción de resultados

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

#define TAM_VECTOR 10

int main(int argc, char *argv[]) {
    int V1[TAM_VECTOR], Suma = 0;

    for(int i=0; i < TAM_VECTOR; i++)
        V1[i] = i + 1; // Introducimos valores 1 a N

    #pragma omp parallel for reduction(+:Suma)
    for(int i = 0; i < TAM_VECTOR; i++) {
        int Prod = V1[i] * V1[i]; // Cálculo en paralelo
        printf("Hilo %d Prod = %d\n", omp_get_thread_num(), Prod);
        Suma += Prod; // Reducción de resultados a variable común
    }

    printf("Suma = %d\n", Suma);
}
```

Con la cláusula `reduction` se indica la operación de reducción a efectuar, en este caso la suma, y las variables implicadas, en este caso `Suma`

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
$ gcc reduction.c -o reduction -fopenmp

Francisco@Francisco-PCUJA ~
$ ./reduction
Hilo 1 Prod = 9
Hilo 2 Prod = 25
Hilo 4 Prod = 49
Hilo 6 Prod = 81
Hilo 3 Prod = 36
Hilo 5 Prod = 64
Hilo 7 Prod = 100
Hilo 0 Prod = 1
Hilo 1 Prod = 16
Hilo 0 Prod = 4
Suma = 385

Francisco@Francisco-PCUJA ~
$
```

El resultado es la acumulación de los resultados en la variable indicada sin problemas de sincronización por acceso simultáneo de varios hilos a la misma

Cómo medir el tiempo de ejecución

```
#include <time.h> // Además de otros include

typedef struct timespec timespec;

long diff(timespec inicio, timespec fin) {
    return ((fin.tv_sec - inicio.tv_sec) * 1e9 +
            (fin.tv_nsec - inicio.tv_nsec)) / 1000;
}

int main(int argc, char *argv[]) {
    timespec inicio, fin;

    clock_gettime(CLOCK_REALTIME, &inicio);
    #pragma omp parallel
    {
        // Código paralelizado
    }
    clock_gettime(CLOCK_REALTIME, &fin);

    printf("Tiempo con OpenMP: %10d microsegundos\n", diff(inicio, fin));

    clock_gettime(CLOCK_REALTIME, &inicio);
    // Mismo código anterior sin paralelizar
    clock_gettime(CLOCK_REALTIME, &fin);

    printf("Tiempo sin OpenMP: %10d microsegundos\n", diff(inicio, fin));
}
```

La función `diff()` se encarga de calcular el tiempo transcurrido a partir de la información devuelta por `clock_gettime()`. El valor está expresado en microsegundos

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
$

Francisco@Francisco-PCUJA ~
$ ./ganancia
Tiempo con OpenMP:      665038 microsegundos
Tiempo sin OpenMP:     1380078 microsegundos

Francisco@Francisco-PCUJA ~
$
```

Es recomendable ejecutar varias veces, por ejemplo 5 o 10, y calcular la media de los tiempos para obtener una medida más fiable. Con estos datos podríamos calcular la ganancia real obtenida al usar OpenMP

- Colección de ejemplos facilitada por OpenMP.org:
http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf
- Extensa colección de ejercicios con soluciones de la Universidad de Berkeley: <https://people.eecs.berkeley.edu/~knight/bootcamp2013/omp-exercises.pdf>
- Libro libre de la Universitat Oberta de Catalunya con sección dedicada a OpenMP y varios ejemplos:
<https://openlibra.com/es/book/download/introduccion-a-las-arquitecturas-paralelas>