Práctica 1:
Programación con OpenMP. Sesión 1.

(Actualizado del 24/02/2016)

Tiempo de trabajo en casa (previsto): 3:00h
Tiempo de laboratorio: 2:00h
Evaluación de la práctica: 30 m en la siguiente sesión.

1. Introducción

El modelo de programación basado en el paralelismo de datos típico de los procesadores vectoriales y en los array de procesadores (SIMD) tiene su continuación natural en el modelo SPMD (Single Program Multiple Data), desarrollado en los años 80 para los multiprocesadores con memoria compartida, y para el que se construyen los denominados compiladores paralelos o paralelizantes. Estos compiladores eran propios de cada fabricante (IBM, SGI, Gray, etc tenían cada uno el suyo propio). Los científicos e ingenieros dedicados a la resolución de problemas de gran tamaño son los que más se han beneficiado de este tipo de compiladores.

En este modelo de programación el usuario diseña su programa de forma secuencial y es el compilador el que, de forma automática, se encarga de paralelizarlo (fundamentalmente paralelizando bucles) y adecuarlo para su ejecución en varios procesadores. El usuario también tiene la posibilidad de influir directamente sobre el proceso de paralelización, indicando al compilador qué tiene que paralelizar y las características de las variables utilizadas. Todo ello permite el desarrollo de aplicaciones paralelas de manera sencilla y rápida, aunque cuando se buscan incrementos de velocidad importantes se requiere una modificación profunda de los algoritmos, a partir de un estudio detallado de su dependencia de datos, o el reemplazo del algoritmo inicial por otro más adecuado (es decir, más paralelizable).

OpenMP se ha convertido en uno de los estándares actuales en la programación SPMD sobre computadores con memoria compartida como los computadores multinúcleo actuales. El compilador de OpenMP no admite paralelización automática, por lo que el programador es el responsable de la paralelización de los programas (paralelización guiada o explícita), mediante la inclusión de las correspondientes directivas en el código, y de asegurar que los resultados son correctos.

Esta Práctica consta de 3 sesiones de laboratorio, una inicial en la que revisaremos los conceptos visto en clase de forma guiada y práctica, y otras dos sesiones en las que se trabajará la paralelización del problema de estudio elegido por cada grupo. Así, en esta primera sesión se pretende dar a conocer las diferentes utilidades que proporciona OpenMP para el diseño y ejecución de programas paralelos en computadores multinúcleo.

En las sesiones de laboratorio de la asignatura se tratará de afianzar los conocimientos vistos en las clases teóricas y se continuará el desarrollo de las técnicas básicas para la toma de datos y su representación en una tabla o gráfica. Para ello, antes de la sesión de laboratorio se pedirá a los miembros de cada grupo de trabajo que lean la documentación relacionada con el fundamento teórico del trabajo a desarrollar y las herramientas a utilizar. La lectura previa de estos

documentos permitirá utilizar las aplicaciones y el equipamiento del laboratorio, y responder a las preguntas de acerca de su funcionalidad y uso.

En las sesiones de laboratorio es importante establecer una dinámica de trabajo en grupo que permita sacar el máximo provecho de tu estancia en el laboratorio. Para que ello sea posible, se requiere que seas puntal y que prestes atención a las indicaciones del profesor. En esta primera sesión esto es especialmente importante, dado que se presentarán las normas de funcionamiento necesarias para un correcto desarrollo de las sesiones.

Objetivos

El objetivo de esta práctica es aprender a trabajar con OpenMP, estudiando el impacto de las distintas opciones de compilación en la ejecución de los programas. También descubriremos la importancia que tiene el conocimiento profundo de la arquitectura del computador con el que trabajamos en el aprovechamiento de sus recursos y en la aplicación correcta de las opciones de compilación.

Al finalizar esta práctica, el estudiante será capaz de:

- Compilar y ejecutar programas con OpenMP.
- Usar la directiva #pragma omp parallel y los constructores de distribución del trabajo DO/for y sections
- Usar las cláusulas sobre variables private y firstprivate.
- Usar las funciones en tiempo de ejecución: omp_get_thread_num y omp_get_num_threads, omp_set_num_threads, etc.
- Usar las cláusulas de planificación de threads.
- Directivas de acceso a variables compartidas.
- Usar las variables de entorno de OpenMP más útiles.

Recursos

Para el desarrollo de estas sesiones de laboratorio se dispondrá de un computador personal PC con el que se conectará a un servidor (**boe.uv.es**) con Sistema Operativo Linux Centos 6.5.

El servidor **boe** posee dos procesadores multinúcleo Intel Xeon E5-2620V2 a 2,1 GHz con 6 núcleos cada uno (un total de 12 CPU con hyper-threading), con tecnología de 22 nm, 80W de disipación de potencia y una memoria principal DDR3/1600 de 64 GB. Cada núcleo posee tres niveles de Cache: en el nivel L2 hay 0.25 MB por núcleo, y el L3 de cache es unificada de 15MB (2,5 MB por núcleo).

El servidor boe está conectado mediante una red interna GEthernet a 8 nodos de cálculo llamados “compute-0-0” - “compute-0-7”. Cada nodo posee un procesador Intel Xeon E5-2620V2 como los del servidor, pero con una memoria principal de 32GB.

El compilador de C instalado es el GNU gcc versión 4.4.7, que incluye la versión OpenMP 3.0.

Aprendizaje y evaluación del laboratorio.

El aprendizaje y la evaluación de la práctica comienza antes de asistir al laboratorio. Así, el estudiante deberá reunirse con sus compañeros de grupo para preparar cada sesión. El trabajo previo a la práctica consistirá en revisar los conceptos de teoría con los que está relacionada la práctica y comenzar a realizar las actividades propuestas para preguntar las dudas surgidas en la

propia sesión o en tutorías. Todo ello nos preparará para obtener el máximo provecho de la sesión de laboratorio.

Durante el desarrollo de la práctica se evaluará el trabajo desarrollado por el grupo en la sesión presencial en el laboratorio. Habrá diversos apartados, formado por actividades de aprendizaje, que podrán tener una puntuación según su complejidad. A medida que se vaya completando cada actividad propuesta, el estudiante debe llamar al profesor para que compruebe qué ha hecho hasta ese momento. El profesor asignará entonces la puntuación correspondiente a la actividad que se haya completado según su grado de exactitud y las respuestas que el grupo de a las preguntas del profesor.

Al final de algunas sesiones, o inicio de las siguientes, se realizará una evaluación individual relacionada con las actividades y objetivos de ésta.

2. Trabajo previo al Laboratorio – Pre-Lab

Para poder sacar el máximo partido a la sesión de laboratorio es muy importante que leas la práctica y realices parte de las actividades que se proponen, dedicando el tiempo que se indica. Para que este tiempo no supere el tiempo máximo estimado, los miembros del equipo de trabajo deberéis distribuiros las actividades propuestas y realizar posteriormente una puesta en común. Una mala distribución de las mismas, o intentar realizarlo todo un solo miembro, supondrá no poder realizarlas todas o una dedicación de tiempo que estaréis restando de otras actividades.

3. Trabajo en el Laboratorio – In-Lab

En esta sesión se aprenderá a usar los elementos básicos de OpenMP sobre un conjunto de programas ejemplo. Así, en los siguientes puntos se propondrán un conjunto de actividades a realizar seguidas por cuestiones a resolver o acciones que deberás experimentar. La evaluación de esta sesión será individual.

3.1 Inicio: Hola

Vamos a iniciar la práctica con un programa típico (“Hola mundo”) utilizando la directiva *parallel*, que nos creará un conjunto de hebras que se ejecutarán de forma paralela. A continuación, se muestra el código del programa:

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

int main (){
int nthreads, thread;

#pragma omp parallel private(nthreads, thread)
  {
    thread = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    printf("Hola soy la hebra = %d de %d \n", thread, nthreads);
  }
}
```

Como podemos ver, el código comienza con la inclusión de omp.h. La directiva **parallel** afecta al código entre llaves, y le acompaña la directiva **private**, que indica que se harán copias locales de las variables entre paréntesis para cada uno de las hebras.

Para compilar el código utilizaremos la sentencia:

```
> gcc -fopenmp hola.c -o hola
```

Y para ejecutar el programa, simplemente pondremos:

```
> ./hola
```

La salida del programa dependerá del número de hebras que se hayan ejecutado. Por defecto, openMP lanza tantas hebras como núcleos tenga el computador donde se está ejecutando.

A continuación realiza las siguientes pruebas:

1) Cambia el número de hebras que se ejecutan en paralelo mediante la función omp_set_num_threads. Prueba con un número de hebras mayor y menor al de núcleos del computador.
2) ¿Qué pasa al eliminar la cláusula *private*?
3) Crea una variable *int*, asígnale el valor 100, y pásala a la zona paralela como *private*. Comprueba si el valor de la variable es el correcto para todas las hebras y prueba después con la cláusula *firstprivate*.
4) Añade código en la zona paralela para que sólo la hebra “0” imprima por pantalla la variable de tipo *int* anterior.

3.2 Bucles paralelos

Vamos a combinar la directiva **#pragma omp parallel** con la directiva **#pragma omp for** para paralelizar las iteraciones de los bucles. Esta combinación nos permitirá distribuir las iteraciones de un bucle entre las hebras que generemos. A continuación, se muestra el código que utilizaremos como ejemplo:

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

#define N 10
#define nthreads 4

int main (){
  int tid, i;
  omp_set_num_threads(nthreads);
  #pragma omp parallel private(tid)
  {tid = omp_get_thread_num();

     #pragma omp for
     for (i = 0; i<N; i++){
       sleep(i);
       printf("El proceso %d ejecuta la iteracion %d\n",tid,i);
     }
  }
}
```

Utilizando este código como base:

1) Compila este código como en la sección anterior y ejecútalo. Comprueba cuantas iteraciones ejecuta cada hebra, si estas son consecutivas o no y si la distribución de la carga está equilibrada.
2) Prueba los diferentes tipos de planificación que posee la directiva **#pragma omp for schedule([static|dynamic|guided|auto])**, intentando obtener en cada caso la mejor distribución de las iteraciones y anota su comportamiento en cada caso.

3.3 Cálculo de π

Después de practicar las directivas y clausulas encaminadas a la creación de hebras, su reparto y planificación, nos centramos en este punto en el tratamiento de las variables compartidas y los problemas que aparecen cuando accedemos a ellas.

Como caso de ejemplo, utilizaremos el siguiente código que calcula el valor de pi mediante integración:

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


int main(int argc, char *argv[])
{
    int n,i;
    printf("Introduce la precision del calculo (n > 0): ");
    scanf("%d",&n);
    double PI25D = 3.14159265358979323846264 3;
    double h = 1.0 / (double) n;
    double sum = 0.0;

    #pragma omp parallel for shared(sum)
    for (i = 1; i <= n; i++) {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }

    double pi = sum * h;
    printf ("El valor aproximado de PI es: %f, con un error de %f
\n",pi,fabs(pi - PI25D));
    return 0;
}
```

En este programa las hebras creadas se distribuyen las iteraciones del bucle y acumulan los valores calculados en cada iteración en la variable **sum**, que al estar declarada como *shared* está compartida por todas ellas. Cuando mayor es el valor de n, se supone que mayor precisión va a tener el valor de pi calculado.

1) Ejecuta el programa del cálculo de π con precisiones n=10, 100 y 1000. Comprueba los valores obtenidos y justifica su resultado.
2) Para resolver el problema que has visto en el punto anterior, hay que asegurar la atomicidad de los accesos a la variable sum. Para ello, existen 3 maneras de resolverlo utilizando las siguientes directivas y cláusulas:
    a. **#pragma omp critical.**

b. **#pragma omp atomic.**

c. cláusula **reduction**.

Modifica el código del programa para que funcione correctamente utilizando estas soluciones.

3.4 Sections

En este apartado veremos cómo un programa con dos tareas independientes que inicialmente se ejecutan secuencialmente puede ejecutarse de forma más rápida si ambas se ejecutan en paralelo mediante la directiva **#pragma omp sections**.

A continuación, se presenta un programa ejemplo en el que se ejecutan dos tareas de manera secuencial y se mide el tiempo total de la ejecución de ambas:

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

void tarea_uno(){
    sleep(2); /*retardo de 2 segundos*/
}

void tarea_dos(){
    sleep(4);
}

int main (){

   double timeIni, timeFin;

   timeIni = omp_get_wtime();

   printf("Ejecutando tarea 1 \n");
   tarea_uno();

   printf("Ejecutando tarea 2");
   tarea_dos();

   timeFin = omp_get_wtime();

   printf("Tiempo tardado = %f segundos \n", timeFin – timeIni);

}
```

Teniendo como base este código ejemplo:

1) Compílalo y ejecútalo con 2 hebras, comprobando que el tiempo total obtenido es congruente.
2) Utiliza la directiva #pragma omp sections para que se ejecuten las dos tareas de forma paralela. Haz que se muestre qué hebra ejecuta cada *section*.

3) Establece una medida de tiempo en el momento en que las tareas salen del bloque de ejecución paralelo *sections*. Si una tarea tarda más que la otra, las hebras registrarán tiempos con una diferencia aproximada a la diferencia de tiempo de ejecución de las tareas. Comprueba si esta asunción es cierta, y en caso contrario piensa qué hay que hacer para que esto ocurra. Ayuda: piensa en la cláusula **nowait**.
4) Aumenta el número de hebras y observa que ocurre cuando tenemos más hebras que número de **section**.
5) ¿Y si, al contrario, hay más secciones paralelas que hebras? Decrementa el número de hebras (o aumenta el número de secciones) y observa que ocurre en nuestro ejemplo.

3.5 Coste de creación de las hebras y elección de los cores

La creación y destrucción (*fork + join*) de las hebras supone un sobrecoste en la ejecución de un programa paralelo en OpenMP. Por ello, es importante conocer el coste temporal que supone (dependiendo del número de hebras) y si vale la pena la paralelización de una zona de código para un tamaño de problema dado.

Para ilustrar nuestro estudio supongamos el siguiente programa secuencial.

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

int main (){
    unsigned int N;
    int *x, *y;
    printf("Introduce el tamaño del vector: ");
    scanf("%d",&N);
    double timeIni, timeFin;
    x = (int *)malloc(N*sizeof(int));
    y = (int *)malloc(N*sizeof(int));

    timeIni = omp_get_wtime();
    //Secuencial
    for(int i=1; i < N ; i++){
    x[i] = y[i-1] * 2;
    y[i] = y[i] + i;
    }

    timeFin = omp_get_wtime();
    printf("Tiempo tardado secuencial= %f milisegundos\n",
           (timeFin - timeIni)*1000);

    timeIni = omp_get_wtime();

    //Paralelo

    //Escribe aquí el mismo algoritmo de forma paralela

    timeFin = omp_get_wtime();
    printf("Tiempo tardado paralelo= %f milisegundos\n",
           (timeFin - timeIni)*1000);
}
```

Teniendo como base este código ejemplo:

1) Analiza las dependencias del algoritmo secuencial y haz las modificaciones necesarias para que sea paralelizable.
2) Escribe el algoritmo paralelo en el espacio reservado.
3) Ejecuta el código para tamaño igual a 10000 variando el número de hebras (1, 2, 4, 8) utilizando únicamente un solo núcleo. Para ello, fija la variable de entorno para limitar todas las hebras se ejecuten en el Core 0 (GOMP_CPU_AFFINITY=0) y que estas no puedan migrar (OMP_PROC_BIND=True). Infiere con esta información cual sería el coste de la creación de las hebras.
4) Ejecuta varias veces el código para un tamaño igual a 2000, 20000, 200000, 2000000. Observa el tiempo obtenido al ejecutarlo con 2 hebras y 2 cores en tu computador.
5) A partir de los datos del punto anterior, utiliza la cláusula *if* para que se convierta en paralelo a partir de un valor de N que garantice una mayor eficiencia.
6) Utiliza la variable de entorno GOMP_CPU_AFFINITY para ejecutar el código anterior con 2 hebras y utilizando valores de N elevados:
    a. Utilizando 2 Cores de la misma CPU.
    b. Utilizando 2 Cores de CPU distintas

    Comprueba los valores de tiempo obtenidos y justifícalos.

4. Evaluación de la práctica

La evaluación de esta sesión se realizará antes de finalizar la misma, o al inicio de la siguiente, mediante un examen constituido por preguntas de tipo test y de respuesta corta que tendrá una duración máxima de 30 minutos.

OpenMP Reference Sheet for C/C++

Constructs

<parallelize a for loop by breaking apart iterations into chunks>
#pragma omp parallel for [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr), ordered, schedule(type[,chunkSize])]
<A,B,C such that total iterations known at start of loop>
for(A=C;A<B;A++) {
    <your code here>

    <force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
    #pragma omp ordered {
        <your code here>
    }
}

<parallelized sections of code with each section operating in one thread>
#pragma omp parallel sections [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    #pragma omp section {
        <your code here>
    }
    #pragma omp section {
        <your code here>
    }
    ....
}

<grand parallelization region with optional work-sharing constructs defining more specific splitting of work and variables amongst threads. You may use work-sharing constructs without a grand parallelization region, but it will have no effect (sometimes useful if you are making OpenMP'able functions but want to leave the creation of threads to the user of those functions)>
#pragma omp parallel [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(private|shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    <the work-sharing constructs below can appear in any order, are optional, and can be used multiple times. Note that no new threads will be created by the constructs. They reuse the ones created by the above parallel construct.>

    <your code here (will be executed by all threads)>

    <parallelize a for loop by breaking apart iterations into chunks>
    #pragma omp for [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), ordered, schedule(type[,chunkSize]), nowait]
    <A,B,C such that total iterations known at start of loop>
    for(A=C;A<B;A++) {
        <your code here>

        <force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
        #pragma omp ordered {
            <your code here>
        }
    }

    <parallelized sections of code with each section operating in one thread>
    #pragma omp sections [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), nowait] {
        #pragma omp section {
            <your code here>
        }
        #pragma omp section {
            <your code here>
        }
        ....
    }

    <only one thread will execute the following. NOT always by the master thread>
    #pragma omp single {
        <your code here (only executed once)>
    }
}

Directives

shared(vars) *<share the same variables between all the threads>*
private(vars) *<each thread gets a private copy of variables. Note that other than the master thread, which uses the original, these variables are not initialized to anything.>*
firstprivate(vars) *<like private, but the variables do get copies of their master thread values>*
lastprivate(vars) *<copy back the last iteration (in a for loop) or the last section (in a sections) variables to the master thread copy (so it will persist even after the parallelization ends)>*
default(private|shared|none) *<set the default behavior of variables in the parallelization construct. shared is the default setting, so only the private and none setting have effects. none forces the user to specify the behavior of variables. Note that even with shared, the iterator variable in for loops still is private by necessity >*
reduction(op:vars) *<vars are treated as private and the specified operation(op, which can be +, *, -, &, |, &, &&, ||) is performed using the private copies in each thread. The master thread copy (which will persist) is updated with the final value.>*

copyin(vars) *<used to perform the copying of threadprivate vars to the other threads. Similar to firstprivate for private vars.>*
if(expr) *<parallelization will only occur if expr evaluates to true.>*
schedule(type [,chunkSize]) *<thread scheduling model>*

type	chunkSize
static	*number of iterations per thread pre-assigned at beginning of loop (typical default is number of processors)*
dynamic	*number of iterations to allocate to a thread when available (typical default is 1)*
guided	*highly dependent on specific implementation of OpenMP*

nowait *<remove the implicit barrier which forces all threads to finish before continuation in the construct>*

Synchronization/Locking Constructs

<May be used almost anywhere, but will only have effects within parallelization constructs.>

<only the master thread will execute the following. Sometimes useful for special handling of variables which will persist after the parallelization.>
#pragma omp master {
  <your code here (only executed once and by the master thread).
}

<mutex lock the region. name allows the creation of unique mutex locks.>
#pragma omp critical [(name)] {
  <your code here (only one thread allowed in at a time)>
}

<force all threads to complete their operations before continuing>
#pragma omp barrier

<like critical, but only works for simple operations and structures contained in one line of code>
#pragma omp atomic
  <simple code operation, ex. a += 3; Typical supported operations are ++,--,+,,-,/,&,^,<<,>>,| on primitive data types>*

<force a register flush of the variables so all threads see the same memory>
#pragma omp flush[(vars)]

<applies the private clause to the vars of any future parallelize constructs encountered (a convenience routine)>
#pragma omp threadprivate(vars)

Function Based Locking

< nest versions allow recursive locking>

void **omp_init_[nest_]lock**(omp_lock_t*) *<make a generic mutex lock>*
void **omp_destroy_[nest_]lock**(omp_lock_t*) *<destroy a generic mutex lock>*
void **omp_set_[nest_]lock**(omp_lock_t*) *<block until mutex lock obtained>*
void **omp_unset_[nest_]lock**(omp_lock_t*) *<unlock the mutex lock>*
int **omp_test_[nest_]lock**(omp_lock_t*) *<is lock currently locked by somebody>*

Settings and Control

int **omp_get_num_threads**() *<returns the number of threads used for the parallel region in which the function was called>*
int **omp_get_thread_num**() *<get the unique thread number used to handle this iteration/section of a parallel construct. You may break up algorithms into parts based on this number.>*
int **omp_in_parallel**() *<are you in a parallel construct>*
int **omp_get_max_threads**() *<get number of threads OpenMP can make>*
int **omp_get_num_procs**() *<get number of processors on this system>*
int **omp_get_dynamic**() *<is dynamic scheduling allowed>*
int **omp_get_nested**() *<is nested parallelism allowed>*
double **omp_get_wtime**() *<returns time (in seconds) of the system clock>*
double **omp_get_wtick**() *<number of seconds between ticks on the system clock>*
void **omp_set_num_threads**(int) *<set number of threads OpenMP can make>*
void **omp_set_dynamic**(int) *<allow dynamic scheduling (note this does not make dynamic scheduling the default)>*
void **omp_set_nested**(int) *<allow nested parallelism; Parallel constructs within other parallel constructs can make new threads (note this tends to be unimplemented in many OpenMP implementations)>*

<env vars- implementation dependent, but here are some common ones>
OMP_NUM_THREADS "number" *<maximum number of threads to use>*
OMP_SCHEDULE "type,chunkSize" *<default #pragma omp schedule settings>*

Legend

vars is a comma separated list of variables
[optional parameters and directives]
<descriptions, comments, suggestions>
.... above directive can be used multiple times
For mistakes, suggestions, and comments please email e_berta@plutospin.com