# PRÁCTICAS
DE
# INGENIERÍA
DE LOS
# COMPUTADORES

## PRÁCTICA 3

## PARALELIZACIÓN EN HILOS
## Uso de OpenMP para la paralelización en Threads

Javier Bellver García (jbg39@alu.ua.es)
Alejandro Reyes Albillar (ara65@alu.ua.es)
Cesar Enrique Pozo Vásquez (cepv1@alu.ua.es)
Curso 2015 / 2016
Grado en Ingeniería Informática

Universitat d'Alacant
Universidad de Alicante

**ÍNDICE**

## 1. Introducción

En esta práctica, se nos ha planteado la mejora del código de la práctica anterior haciendo uso de los mecanismos de paralelización en hilos que nos proporciona la herramienta “OpenMP”, librería de la cual hablaremos más en detalle en una de las secciones de esta memoria.

En esta memoria, explicaremos brevemente qué es y cómo funciona OpenMP, así como los desafíos que nos ha supuesto la paralelización de un problema secuencial. También compararemos los resultados del rendimiento entre la versión sin paralelizar de la práctica anterior y la versión paralelizada que será entregada junto a esta memoria para obtener la ganancia que se consigue con la paralelización.

Pese a que ya lo explicamos en la práctica anterior, el problema que vamos a optimizar es el problema del cálculo de la covarianza de matrices. Un problema que consta de muchas operaciones matriciales que suponen un gran número de bucles secuenciales, por lo que nos pareció una buena elección como algoritmo a tratar.

## 2. Estado del arte

Durante la práctica anterior expusimos nuestro problema, el cálculo de covarianza de matrices, y presentamos nuestra implementación secuencial en C++ de este problema en concreto.

Cuando comenzamos esta práctica, nuestra implementación se hallaba en el mismo estado en el que acabamos la anterior. Por tanto mediante el uso de la paralelización de OpenMP pretendemos mejorar los tiempos de ejecución que tenemos con la implementación secuencial.

Al igual que en la práctica anterior, vamos a utilizar una misma muestra de datos para todas las mediciones que vamos a tratar. Esta muestra de datos, formada por matrices a tratar, tiene 10 matrices de 1 millón de elementos por columna, agrupandolas en 7 columnas, es decir 10 matrices de 1000000 x 7 elementos que representan los datos estadísticos de la medidas del cuerpo humano.

Este es, por tanto, el estado inicial con el que nos hemos encontrado a la hora de realizar esta práctica.

## 3. Introducción a OpenMP

A continuación realizaremos una breve introducción explicativa de la API de OpenMP. Hemos utilizado esta biblioteca para realizar la paralelización de nuestro problema, ya que permite paralelizar código en C++ y Fortran usando simples directivas de precompilador. Esto nos proporciona una API sencilla a la vez que potente.

Los objetivos de OpenMP son:

1- La estandarización de diferentes arquitecturas y plataformas de memoria compartida.

2- Establecer un conjunto simple de directivas que proporcionan un paralelismo significativo con instrucciones sencillas.

3- Permitir una paralelización cada vez mayor con directivas simples.

4- Portabilidad de la API e implementación de esta en diferentes compiladores de C++ y Fortran.

OpenMP está diseñado para procesadores con múltiples núcleos y memoria compartida, pudiendo utilizar tanto la arquitectura de tipo UMA o NUMA.

OpenMP cumple los objetivos de paralelización exclusivamente mediante el uso de Threads o hilos. Como ya sabemos, un Thread es la unidad más pequeña ejecutable en un Sistema Operativo. Además, los threads sólo existen dentro del contexto de un proceso, ya que fuera de este proceso estos hilos dejan de existir. Normalmente, el número máximo de threads viene dado por número de núcleos del procesador, pero su uso está gestionado por una aplicación, por lo que pueden existir más hilos ejecutando diferentes partes de dicha aplicación.

La API de OpenMP es un modelo de programación explícita, que no automática, que ofrece al desarrollador total control sobre la paralelización que va a usar. La paralelización puede ser tan simple como poner pragmas en los sitios adecuados o tan complejo como implementar subrutinas para incluir ciertos niveles de paralelismo en el programa.

OpenMP utiliza el modo de Fork-Join para la ejecución donde todos los programas empiezan en un mismo thread llamado el “Master-Thread”. Dicho thread se ejecuta secuencialmente hasta que encuentra la siguiente región paralela.

Cuando llega a una región paralela, se hace un “Fork” y entonces se crean un conjunto de hilos separados que se ejecutarán según las directivas que hayamos especificado. Cuando la ejecución de los hilos se completa, se realiza un “Join” y los hilos se cierran y sincronizan. El número de hilos y de regiones de paralelismo es arbitrario.

El API nos proporciona funciones para la creación, destrucción y manejo dinámico de hilos durante el tiempo de ejecución. Esta funcionalidad estará determinada por la implementación del programa en concreto.

Con respecto al manejo de memoria, OpenMP establece una caché dentro de los hilos que no tienen la obligación de mantener las variables actualizadas entre ellos, por tanto esta labor recae sobre el programador.

El API de OpenMP tiene tres componentes:

1- Directivas de precompilador
2- Librería de subrutinas en tiempo de ejecución
3- Variables de entorno

Las diferentes implementaciones de OpenMP pueden restringir o no implementar ciertos elementos en particular de la API.

Ahora introduciremos algunas de las instrucciones de OpenMP que nos parezcan importantes y/o que hayamos utilizado para la realización de esta práctica en particular:

1- #pragma omp parallel {}: Con esta directiva creamos un nuevo hilo que ejecutará aquel código que esté dentro de este bloque.

2- #pragma omp parallel for: Este pragma se utiliza justo antes de un bucle para indicar que crearemos una nueva región paralela donde cada iteración del bucle será ejecutado por un hilo en particular.

3- #pragma omp barrier: Con esta directiva obligamos al programa a que espere hasta que todos los threads hayan llegado hasta donde se encuentra esta instrucción.

Con esto terminamos nuestra breve introducción teórica a OpenMP y entramos en el apartado que corresponde a la implementación de nuestra práctica.

También cabe destacar que al hacer una directiva podemos indicar:

- #shared( var1,var2) las variables de memoria compartida
- #private(var1) las que no necesitan como copia local de la variable y son temporales
- #reduction( op: var1) indica una variable temporal el cual se actualiza con una operación de suma, resta, multiplicacion o division.

## 4. Tipos de paralelización implementados

En este apartado analizaremos los diferentes tipos de paralelización que hemos utilizado en nuestro programa. Mostraremos estos tipos de paralelización junto a diferentes trozos de nuestro código que nos servirán para explicar el tipo de paralelización en concreto y nuestra decisión de utilizarla para la resolución de nuestro problema.

Primero un breve apunte sobre la compilación del proyecto. Como ya hemos apuntado antes, nuestro proyecto esta hecho en C++, por lo que utilizaremos el compilador de gcc proporcionado por los laboratorios de la EPS. Es primordial utilizar la opción “-fopenmp” para que nuestro programa funcione utilizando los pragmas de OpenMP.

Antes de comenzar citando los diferentes tipos de paralelización y su código asociado, es importante indicar que modo de paralelización hemos usado. Existen tres modos de paralelización. El primero es SPMD, “Single Program Multiple Data”, en este modo tenemos un único programa que realiza la paralelización en múltiples datos, normalmente mediante el uso de funciones dentro de un mismo programa. El segundo modo es MPMD, “Multiple Program Multiple Data”, en este modo tenemos múltiples programas que realizan la paralelización sobre múltiples datos. El último es el híbrido, donde se utilizan ambos métodos.

Nuestro programa está basado en el método SPMD, tenemos un programa escrito en C++ con diferentes funciones, las cuales usan de OpenMP para su paralelización. Ahora explicaremos cada uno de los tipos de paralelización que hemos utilizado en cada una de las funciones de nuestro programa.
La primera función que hemos paralelizado en nuestro código se trata de la función “mean”.

```
double mean(vector<double> &data) {
    double mean = 0.0;
    unsigned i=0;
    int chunk=1000;

    #pragma omp parallel for \
    shared(data,chunk) private(i)  \
    schedule(static,chunk) \
    reduction(+:mean)

    for(i=0; (i < data.size());i++)
      mean =mean+ data[i];
    }

    mean = mean/data.size();
    return mean;
```

Aquí podemos ver que nuestro método tiene un sólo bucle que hemos paralelizado mediante la directiva “omp parallel for” que crea una nueva sección paralela en la que a cada hilo realiza una de las iteraciones.

Aparte, con la directiva shared, estamos indicando variables que deben compartirse entre todos los hilos que vayan a ejecutarse. Con private, indicamos que la variable i debe ser privada con respecto a cada hilo. Con schedule dividimos las diferentes iteraciones entre los hilos. En este caso estamos realizando un schedule static, en este caso en trozos estáticos de 1000 iteraciones por thread.

El siguiente método que hemos paralelizado, es el método “traspuesta” que calcula y devuelve una matriz traspuesta.

```
void traspuesta(vector<vector<double> > & m,vector<vector<double> > & t){
    int colLim=m[0].size();
    int filLim=m.size();
    int i=0;
    #pragma omp parallel for
    for( i=0;i<colLim; i++){
      vector <double> relleno;
      int j=0;

      for( j=0;j<filLim;j++){
  relleno.push_back( m[j][i]);
      }
      t.push_back(relleno);
    }
```

En este caso hemos recurrido a un omp parallel for simple en el bucle externo, ya que debido a la naturaleza de la operación push_back en el vector dinámico, la paralelización del bucle interno resultaba en una pérdida de tiempo debido a deadlocks.

Acabaremos con la última paralelización que hemos realizado, con esta estructura en el main, hemos conseguido realizar una paralelización de datos importante haciendo que cada hilo procese 10 matrices, consiguiendo una paralelización de datos óptima:

```
int main(int argc,const char* argv[])
{
  if(argc == 2)
  {
    clock_t start;
    readMatrices(argv[1]);
    start = clock();
    cout << "Tiempo de comienzo de la aplicacion: " <<CPU_time(start) <<endl;
    start = clock();

    int chunk=10;

    unsigned int i=0;
    vector< vector <double> > resuelta;
    #pragma omp parallel for \
    shared(matrices,resuelta,chunk) private(i)\
    schedule(dynamic,chunk)


    for(i=0;i<matrices.size();i++)
    {
      compute_covariance_matrix(matrices[i],resuelta);
      resuelta.clear();
    }
    cout << "Tiempo de finalización de la aplicación: " << (double)CPU_time(start) <<endl;
  }else{
    cout << "Número de parámetros incorrecto" <<endl;
    cout << "Formato $./main nombre_de_fichero" <<endl;
    return -1;
  }
  return 0;
}
```

## 5. Mediciones

En este apartado realizaremos mediciones en nuestros equipos para comparar el rendimiento obtenido en estos y compararlo con la ejecución secuencial de nuestro programa.

El método de medición es el siguiente. Se obtendrán cinco tiempos de cada equipo. Por razones técnicas descartamos el primer tiempo (que tardará más probablemente ya que el mismo computador no habrá optimizado la ejecución del programa). Nos quedaremos con las 4 mediciones restantes y realizaremos la media de estas.

HARDWARE ORDENADOR 1:
Procesador: Intel Core 2 Duo
Memoria RAM: 6 Gbyte
Disco duro: 1 Terabyte Sata
Sistema operativo: Linux - Ubuntu

cesar, mira la carpeta capturas para que veas como lo he hecho yo

Medición secuencial:

| Ejecución 1 | Ejecución 2 | Ejecución 3 | Ejecución 4 | Ejecución 5 | Media |
|---|---|---|---|---|---|
| 58.71 | 57.46 | 57.56 | 59.19 | 57.74 | 58,13 |

Medición paralela:

| Ejecución 1 | Ejecución 2 | Ejecución 3 | Ejecución 4 | Ejecución 5 | Media |
|---|---|---|---|---|---|
| 62.93 | 65.52 | 64.83 | 63.07 | 64.65 | 64,32 |

No ofrece rendimiento esperado ya que el ordenador posee una arquitectura que no aprovecha bien openMP. Debido probablemente a que no sea capaz de lanzar los hilos suficientes o a un bajo rendimiento a la hora de manejar la sincronización y los deadlocks.

HARDWARE ORDENADOR 2:
Procesador: Intel Core i7 - 4710HQ CPU
MHz CPU:  2494 MHz
Memoria física total:  7,92 GB
Memoria virtual total:  14,3 GB
Disco:  750 GB

Medición secuencial:

| Ejecucion1 | Ejecucion2 | Ejecucion3 | Ejecucion4 | Ejecucion5 | Media |
|---|---|---|---|---|---|
| 29.04 | 27.05 | 30.02 | 29.79 | 28.49 | 28,8375 |

Medición paralela:

| Ejecucion1 | Ejecucion2 | Ejecucion3 | Ejecucion4 | Ejecucion5 | Media |
|---|---|---|---|---|---|
| 30.39 | 28.63 | 27.63 | 29.79 | 28.96 | 28,7525 |

HARDWARE ORDENADOR 3:
Procesador: Intel Core i3  3110M CPU
MHz CPU:  2400 MHz
Memoria física total:  7,88 GB
Memoria virtual total:  10,0 GB
Disco:  1,00 T

Medición secuencial:

| Ejecucion1 | Ejecucion2 | Ejecucion3 | Ejecucion4 | Ejecucion5 | Media |
|---|---|---|---|---|---|
| 45.38 | 49.3 | 44.79 | 44.99 | 42.95 | 45.482 |

Medición paralela:

| Ejecucion1 | Ejecucion2 | Ejecucion3 | Ejecucion4 | Ejecucion5 | Media |
|---|---|---|---|---|---|
| 44.37 | 47.99 | 41.54 | 41.48 | 41.14 | 43.304 |

## 6. Conclusiones

Acabaremos la memoria con unas breves conclusiones sobre nuestro trabajo y el uso de OpenMP para la resolución del problema, el cálculo de la covarianza de un gran conjunto de matrices.

Como hemos indicado tanto en este práctica como en la anterior, teníamos la presunción de que teóricamente nuestra implementación secuencial del cálculo de la covarianza era realmente paralelizable. Tras utilizar OpenMP para paralelizar nuestra implementación y comparando los tiempos de nuestra implementación secuencial y nuestra implementación paralelizada, podemos asegurar que definitivamente la paralelización ha generado una implementación que gana en rendimiento pero no tanto como esperábamos.

Esto se debe a que en muchos casos la paralelización de ciertas funciones del problema nos ha dado peores tiempos, debido a problemas de sincronización y deadlocks causados por ciertos métodos que tratan con vectores dinámicos.

Con respecto al uso de OpenMP, hemos encontrado una API mucho más fácil de lo que nos esperábamos a la hora de implementar paralelismo basado en Threads. Habiendo dicho esto, es cierto que nuestro programa ha requerido un trabajo complejo con los recursos que nos ofrece la API para evitar ciertos problemas como deadlocks o corrupción de memoria.

Aquí acaban nuestras conclusiones con respecto a la memoria en particular y con esto acaba nuestro trabajo. Ahora incluimos la bibliografía de los recursos que hemos utilizado para ayudarnos con la implementación de nuestro sistema y con la parte teórica de este mismo trabajo.

## 7. Bibliografía

-Tutorial OpenMP (Con parte teórica)

https://computing.llnl.gov/tutorials/openMP/

-Tutorial práctico OpenMP (Con ejemplos)

http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

-Tutorial pequeño en Castellano

http://lsi.ugr.es/jmantas/OpenMP.pdf

-Multiplicacion matrices Universidad de Murcia
http://www.ditec.um.es/~javiercm/curso_psba/sesion_03_openmp/PSBA_OpenMP.pdf

-Universidad de Valencia - memoria compartida matriciales
https://riunet.upv.es/bitstream/handle/10251/43569/tfm.pdf?sequence=1

-Universidad murcia
http://dis.um.es/~domingo/apuntes/AlgProPar/0809/matrices.pdf