# Programación de SM de Memoria Compartida

Noelia Díaz Gracia UO188772
Lucas Díaz Sanzo UO189670
Aida Fernández Méndez UO16559

Arquitectura y Tecnología de Computadores –Escuela de Ingeniería Informática - Universidad de Oviedo Campus de Viesques E-33271 Gijón

**Resumen.** Este documento realiza una revisión sobre los diferentes métodos de programación paralela, estudiando para cada uno de ellos su funcionamiento y rendimiento respecto a los otros métodos existentes. De esta manera, se trata de ofrecer una visión global de los aspectos más característicos de cada una de estas técnicas y ofrecer una perspectiva de las ventajas y puntos débiles de cada una de ellas respecto a las demás.
Para ello, en primer lugar se hace un breve resumen de los aspectos más importantes a tener en cuenta a la hora de utilizar técnicas de programación paralela, atendiendo especialmente a los requisitos que existen para el acceso a regiones de memoria compartida y a las dificultades que puede encontrar un programador de cara a de hacer uso de la misma.
Tras ello, el texto analiza tres de los métodos más comunes de programación paralela: el uso de compiladores paralelos; la utilización de librerías que incluyan acceso a hilos y señales de control para el acceso a memoria; y la programación utilizando lenguajes paralelos, empleando como ejemplo el lenguaje UPC (Unified Paralell C).

**Palabras clave:** Paralelismo, Memoria compartida, Concurrencia, Compiladores, Open MP, Posix Threads, Threads, Sincronizacion, Mutex, Semaforos, Variables condicionales, UPC, UPC barrier.

## 1 Introducción

El concepto de memoria compartida puede referirse a conceptos relacionados tanto con el Hardware como con el Software.
Los sistemas multiprocesadores pueden clasificarse según diferentes criterios:

- Modelo de comunicación entre procesos: variables compartidas o paso de mensajes.
- Arquitectura de memoria: Memoria compartida o memoria distribuida.
- Combinación de ambos criterios:
    - Multiprocesadores de memoria compartida centralizada: variables compartidas + memoria compartida.

- Multiprocesadores de memoria compartida distribuida: variables compartidas + memoria distribuida.
- Multi-computadores, clusters de PC´s; Paso de mensajes + memoria distribuida.

No viene al caso entrar en una descripción de todas las diferentes opciones y sus arquitecturas, ya que no está dentro de los límites del trabajo. Así que por tanto será preciso centrar el estudio en las técnicas de programación que hacen uso de memoria compartida y los fundamentos que subyacen en el uso de esta técnica. El objetivo de este estudio será tratar de descubrir cómo resolver problemas de paralelismo implícito.
Para llevar a cabo una programación usando técnicas de memoria compartida será preciso tener en cuenta que:

- Cualquier procesador debe poder acceder a cualquier módulo de memoria.
- El acceso a un módulo de memoria por pate de un procesador no debe de privar a otro procesador del acceso a memoria.
- Hay que tratar de minimizar los tiempos de acceso a memoria para penalizar lo menos posible el rendimiento global de sistema.
- Mantener la coherencia de la memoria es imprescindible y la sincronización supone el mayor reto con el que será preciso lidiar.

Para asegurar el cumplimiento de estas condiciones a la hora de hacer uso de esta técnica se han desarrollado lenguajes de programación específicos que permiten el uso de Memoria Compartida. La mayor parte de estos no son más que lenguajes secuenciales ampliados mediante un conjunto de llamadas al sistema especiales que producen primitivas de bajo nivel para el paso de mensajes, sincronización de procesos, exclusión mutua, etc.,...
El uso de estos lenguajes implica un problema de portabilidad entre diferentes arquitecturas. Para tratar de resolverlo se han venido desarrollando librerías (PVM, MPI) que implementan el paso de mensajes y el uso de memoria Compartida (Open MP) incorporando primitivas independientes de la arquitectura del computador.

El objetivo del presente estudio es mostrar una visión global del problema al que un programador se enfrenta a la hora de hacer uso de memoria compartida. Podría decirse que lo fundamental es visualizar el programa como una serie de procesos accediendo a una zona central de memoria compartida, la cual puede ser accedida por diferentes procesos de modo simultánea. Para evitar comportamientos inesperados se han desarrollado mecanismos de exclusión mutua.

Para lograr una programación con memoria compartida, respetando los principios básicos de la existencia de un espacio de direcciones único que puede ser accedido por cualquier proceso se plantean diferentes alternativas:

1. Desarrollar un nuevo lenguaje de programación
2. Modificar un lenguaje secuencial existente.
3. Usar subrutinas y librerías de los lenguajes secuenciales ya conocidos.
4. Programación secuencial paralelizando la compilación, obteniendo así un código ejecutable paralelo.
5. Procesos Unix (ICP)

6. Hilos (Pthreads, Java).

Los primeros lenguajes de programación paralela surgieron a mediados de los 70 de los cuales podrían destacarse: Concurrent Pascal, Ada, Modula P, C*, Concurrent C, Frotran D.

En el desarrollo del trabajo se explicarán con mayor detalle los aspectos más relevantes relacionados con la programación paralela. Como aproximación podría decirse que el primer objetivo es el de crear el paralelismo, es decir procesos concurrentes recurriendo a primitivas como fork() y join(), o haciendo uso de procesos maestros y esclavos o bien de hilos. Tras crear e paralelismo es preciso asignar a estos recursos de memoria compartida haciendo uso de primitivas de exclusión mutua y sincronización. Para hacer uso de esos mecanismos y primitivas será precisa la definición de secciones críticas en el código dónde se haga uso de herramientas de bloqueo (lockf) y además habrá que programar sincronización de barreras (procesos esperando unos por otros para continuar la ejecución).

## 2 Compiladores

Los compiladores paralelos basan su funcionamiento en el uso de directivas, funciones específicas y variables de entorno.
Al utilizar un compilador paralelo, no es necesario que el programador cree a mano diferentes hilos de ejecución para paralelizar una aplicación. En su lugar, se utilizan directivas para marcas aquellas regiones de código que se desea ejecutar el paralelo, y los diferentes flujos de ejecución se crearán en tiempo de compilación.
Aunque obtiene un rendimiento peor que otros métodos, como pueden ser el uso de librerías o lenguajes paralelos, su sencillez de uso los hace adecuados en aquellas aplicaciones en las que resulta más interesante minimizar el tiempo de desarrollo que obtener menores tiempos de ejecución.
Habitualmente su funcionamiento se basa en el análisis, sentencia a sentencia, del código. Se buscan las dependencias entre las diferentes partes del código y se divide la tarea original entre los diferentes nodos disponibles. Un nodo puede ser desde un procesador hasta un equipo, si estamos trabajando en un entorno distribuido.
El ejecutable se genera intercalando en el código original aquellas instrucciones necesarias para ejecutar un hilo paralelo en cada uno de estos nodos. Se conoce como grado de granularidad al número de instrucciones que deben ejecutarse como un todo porque no pueden asignadas por separado a los diferentes nodos.
Habitualmente las optimizaciones de paralelización se realizan en los bucles del código, asignando iteraciones diferentes a cada núcleo. Ésta es actualmente la mejora más importante que pueden ofrecer los compiladores paralelos.

## 2.1 Rendimiento

Los compiladores paralelos minimizan la inversión en tiempo y esfuerzo que debe realizar un desarrollador , obteniendo resultados adecuados cuando se ejecutan tareas repetitivas, como pueden ser bucles de operaciones. Sin embargo, no resultan eficientes para programas que no disponen de estas estructuras.
Además, cabe destacar que el rendimiento del la compilación paralela se puede ver afectada por la distribución de los datos tratados. El rendimiento obtenido para conjuntos de datos densos[1] es considerablemente mayor que aquel alcanzado al utilizar conjuntos de datos poco densos.
Para ilustrar las situaciones anteriores se plantea el siguiente ejemplo. Se trata de un código que, utilizando las directivas de compilación de OpenMP, realiza multiplicaciones de matrices de dos dimensiones:

```
#pragma omp parallel for private(sum, rowend, rowbeg,
nz)
//multiplicación de matrices
for (i = 0; i < num_rows; i++)
{
       sum = 0.0;
       rowend = Arow[i+1];
       rowbeg = Arow[i];
       for (nz=rowbeg; nz<rowend; ++nz)
       {
              sum += Aval[nz] * x[Acol[nz]];
       }
       y[i] = sum;
}
```

**Ejemplo 2.1 Compilación paralela de una multiplicación de matrices**

El procesador sobre el que se ha realizado la prueba es un Intel Xeon E5450 (3.0 GHz) y los resultados reflejan un importante aumento del rendimiento al incrementar el número de hilos, llegando a conseguir una ganancia del 45% al usar dos hilos en lugar de uno (compilación tradicional).

[1] Las matrices densas no tienen gran cantidad de elementos nulos, como pasa en Linpack.

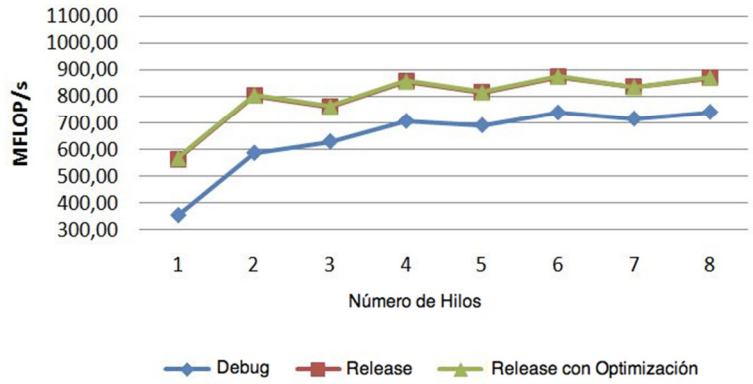



**Figura 2.1 Rendimiento de la aplicación generada (MFLOPS) según el número de hilos**

Sin embargo, se puede observar que existe un límite a partir del cual el número de hilos no representa un aumento del rendimiento. En el ejemplo que nos ocupa, esto se debe al ya citado problema del conjunto de datos dispersos. Al ser así es necesario realizar frecuentes accesos a memoria principal, lo que supone que el rendimiento del procesador se ve limitado.

### 2.2 Ventajas

Una de las principales ventajas de este tipo de compiladores es la de ser más sencillos de usar que los lenguajes y bibliotecas para memoria compartida porque no es necesario proteger explícitamente la integridad de los datos ni realizar la comunicación entre diferentes regiones de memoria. Todas estas operaciones se realizan durante la compilación y son transparentes al programador.
Por otra parte, poseen la ventaja añadida de mejorar la portabilidad del código desarrollado. Un compilador no paralelo ignoraría las directivas de paralelización, que le resultarían desconocidas, pero podría tratar el resto del código de manera correcta.

## 2.3 OpenMP

OpenMP es un compilador que cuenta con implementaciones para conocidos lenguajes, como C, C++ y Fortran. Un aspecto que lo hace interesante es que sea aplicable también a arquitecturas distribuidas, siempre que ambas utilicen OpenMP o MPI. Además, cuenta con funciones que permiten conocer el estado del programa, los hilos creados, etc.
Las directivas soportadas por el compilador permiten la creación del número de hilos deseados por el programador, la paralelización de funciones concretas, el uso de

variables compartidas o la definición de regiones de exclusión. Dichas directivas siguen el siguiente formato:

```
# pragma omp <directiva> [cláusula [ , ...] ...]
```

## 3 Hilos y memoria compartida: Sincronización

El empleo de hilos permite que un proceso pueda ejecutarse de forma concurrente. Se dice que existe concurrencia cuando un proceso tiene al menos dos hilos están en progreso la vez y que el paralelismo se da cuando dos hilos están ejecutándose simultáneamente.
El estándar más utilizado para el uso de hilos es el IEEE 1003.1.c POSIX THREADS desarrollado por Solaris. Esta librería de hilos viene integrada en la mayoría de las API´s de desarrollo y proporciona una interfaz para:

- Gestión de hilos
- Sincronización

Un hilo es un camino de ejecución independiente dentro de un programa y el uso de múltiples hilos permite a una aplicación solapar diferentes operaciones y también le permite servir peticiones de múltiples usuarios.
Los hilos de un mismo proceso comparten todos sus recursos (descriptores de fichero, memoria,…) y por ello los datos son compartidos entre todos los hilos de control existentes en el proceso. A continuación se muestra el esquema de un proceso Unix que emplea múltiples hilos:

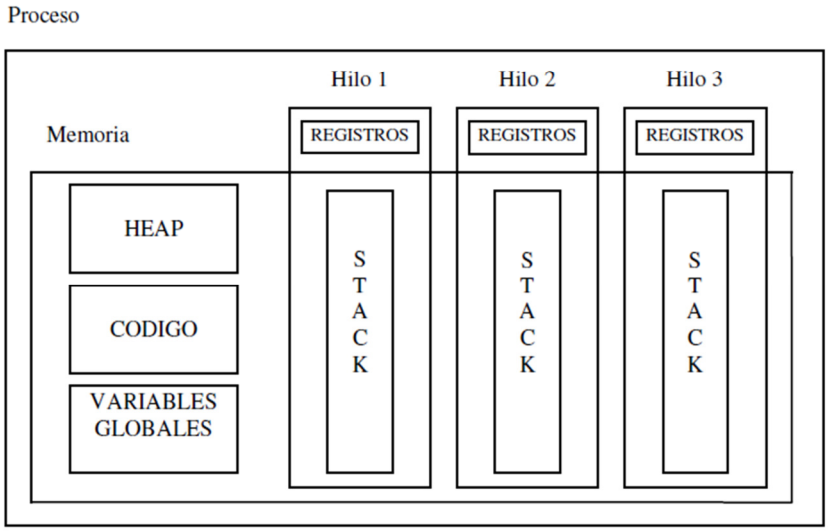



**Figura 3. Proceso Unix con 3 hilos.**

La utilización de hilos aporta múltiples beneficios, mejores programas y mejor rendimiento tanto en sistemas multiprocesador como monoprocesador. Pueden

destacarse los siguientes beneficios: Mejorar la respuesta de las aplicaciones, Utilización eficiente de los sistemas multiprocesador, mejora en la estructuración, mejor utilización de los recursos del sistema y sobre todo, mejora del rendimiento que se logra gracias al solapamiento de cálculos de entrada y salida, creación y destrucción de procesos que efectúan tareas concretas.
A continuación se nombran algunas de las aplicaciones prácticas de los hilos:

- Multiplexación de la entrada/salida: Se consigue que un programa grande procese datos asociados a múltiples descriptores de ficheros, creando un hilo por cada descriptor que se queda bloqueado a la espera de datos sin bloquear el resto de entradas. Las principales ventajas son: Incremento de la modularidad e incremento del rendimiento.
- Procesamiento asíncrono: Los sucesos asíncronos no tiene por qué estar relacionados en el tiempo. Los programas que proporcionan servicios a otros programas a través de mecanismos de intercomunicación entre procesos (IPC) pueden usar hilos para responder simultáneamente.
- Relización de un Handler de señal: se trata de una función que realiza su trabajo dentro de sí misma o invocando a una o dos pequeñas funciones más. Se consigue haciendo uso de un hilo que espere de forma síncrona por la llegada de señales asíncronas.

Hay cuatro modelos de programas en los que la programación mediante hilos se ajusta bastante bien:

- Modelo jefe-Subordinado: Uno de los hilos funciona como jefe que asigna tareas a los subordinados, y cada uno de estos realiza tareas diferentes. Una variación de este modelo es el modelo de cola de trabajos, dónde el jefe coloca trabajos en la cola y los empleados los toman de la misma.
- Modelo de trabajo en equipo: Varios hilos trabajan en una misma tarea juntos. La tarea es dividida en partes y cada hilo ejecuta una.
- Modelo de tubería: La tarea se divide verticalmente en etapas dónde cada una debe estar en secuencia y el trabajo realizado en cada etapa es prerrequisito para el trabajo de la siguiente etapa

Una vez hecha la división de una aplicación en múltiples hilos de control y su planificación es preciso realizar la sincronización, aspecto fundamental para lograr un buen funcionamiento de la aplicación.
Para hacer un uso correcto de estos es preciso definir **secciones críticas** en el código dónde otro hilo no podrá hacer nada que afecte a esa sección mientras uno está ejecutándola. El código de una sección crítica viene definido por: *Sección de entrada, Sección crítica, sección de salida y sección restante*.
Cualquier solución al problema de la secciones críticas debe satisfacer las propiedades de **exclusión mutua** (un solo hilo puede estar en la sección crítica de un recurso en un instante), **progreso** (si un hilo no está en la sección crítica y uno quiere entrar, podrá hacerlo) y **espera limitada** (ningún hilo puede ser pospuesto ilimitadamente). Esto implica que todos los datos compartidos deben ser protegidos mediante **mecanismos de sincronización**.

- *La granularidad del bloqueo*: las librerías de hilos proporcionan un conjunto de mecanismos para sincronización que incluyen semáforos, cerrojos (locks)

y variables condicionales. Al aplicar alguno de estos mecanismos hay que delimitar el bloqueo determinando la cantidad de datos a proteger.

- *Variables de sincronización*: La sincronización se consigue a través de un conjunto de funciones de sincronización que manejan variables especiales de la memoria de usuario, las cuales tienen funciones de inicialización y destrucción.
    - ***Mutex***: el tipo de sincronización más simple y más comúnmente utilizado. Es el mecanismo más eficiente para realizar la exclusión mutua y se utiliza para serializar el acceso a recursos compartidos:
        - Pthread_mutex_lock y pthread_mutex_unlock
    - ***Variables Condicionales***: es el mecanismo de sincronización más eficiente, sobre todo cuando se trata de trabajo bajo situaciones complejas. Se recurre a su uso siempre que se pueda expresar una condición en un programa. Las variables condicionales crean un entorno seguro para comprobar la veracidad de la condición, bloquear el hilo cuando sea falsa y despertarse cuando sea cierta.
        - Pthread_cond_signal
        - Pthread_cond_broadcast
        - Pthread_cond_timedwait
    - ***Semáforos***: Utilizan más memoria que las variables condicionales, pero su utilización es más sencilla. Su funcionamiento básico puede asemejarse al de contadores que se incrementan o decrementan y las funciones básicas de manejo son:
        - Sem_pot: incrementa
        - Sem_wait: decrementa

La sincronización puede acarrear ciertos problemas cuando no se utilizan variables de sincronización correctamente, llamados Interbloqueos. El interbloqueo se presenta si se presentan las siguientes condiciones:

1. Exclusión mutua
2. Retención y espera
3. No apropiación
4. Espera circular

# 4 UPC (Unified Parallel C)

UPC es un lenguaje de programación explícitamente paralelo que facilita la especificación del paralelismo de los programas y el control de la distribución y acceso de los datos. Este lenguaje es una extensión de ISO C y en consecuencia, cualquier programa en C es también un programa en UPC, sin embargo, éste podría comportarse de forma diferente si se ejecuta en un entorno paralelo.

El número de hilos, o grado de paralelismo se fija o bien en el compilador o en una configuración inicial del programa y no se puede cambiar en mitad de la

ejecución. Cada uno de los hilos creados ejecuta el mismo programa, aunque éstos pueden tomar caminos de ejecución diferentes.

```
Upcc -o holamundo -T=4 holamundo.upc
```

**Ejemplo 4.1 Compilación del programa holamundo.upc con 4 hilos**

En el ejemplo anterior se está compilando, haciendo uso del compilador upcc, el programa holamundo.upc con 4 hilos.

### 4.1 Control del flujo de ejecución

El principal mecanismo para controlar el camino de ejecución de cada hilo se basa en utilizar dos variables globales predefinidas: MYTHREAD y THREADS.

THREADS indica el número de hilos totales que están ejecutando el programa, un valor común para todos los hilos. Continuando con el ejemplo 1, en ese caso THREADS tomaría el valor 4. El valor de MYTHREAD es un índice que indica el hilo que está ejecutándose en este momento, comenzando en 0 y llegando a THREADS-1. En el ejemplo 1, los 4 hilos creados recibirían los índices 0, 1, 2 y 3.

```
for (i = 0; i < N; i++){
       if (MYTHREAD == i%THREADS){
              //operar
       }
}
```

**Ejemplo 4.2 Control básico del camino de ejecución en un bucle**

En el ejemplo 2, para cada una de las iteraciones que deba hacer el programa, las operaciones sólo se realizarán cuando el hilo que se está ejecutando cumpla que MYTHREAD == i%THREADS, es decir, que el hilo 0 ejecutará las iteraciones 0, THREADS+1, 2*THREADS+1, etc… Si se tuvieran 2 hilos, el hilo 0 ejecutaría las iteraciones 0, 2 y 4 y el hilo 1 las iteraciones 1, 3 y 5. Esta mejora reduce a la mitad las operaciones que debe efectuar cada hilo, sin embargo, todos ellos deben realizar todas las evaluaciones del bucle for. Esto se puede evitar introduciendo una mejora respecto a la versión del ejemplo 2:

```
for (i = MYTHREAD; i < N; i+=THREADS){
       //operar
}
```

**Ejemplo 4.3 Control optimizado del camino de ejecución en un bucle**

En el ejemplo 3, cada hilo evalúa la condición del bucle for solo cuando va a realizar las operaciones (mas la vez que no se cumple y sale del bucle)

## 4.2 Modelo de memoria

En el apartado anterior se ha visto cómo controlar lo que ejecuta cada hilo, pero ¿cómo accede cada hilo a la zona de memoria común? UPC divide el espacio de direcciones en dos tipos: espacio privado y espacio compartido. El diagrama siguiente muestra esta distribución:

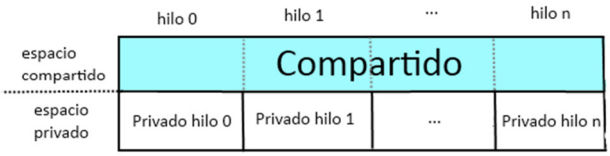



**Figura 4.1 Organización de la memoria en UPC**

En lo que respecta a la programación propiamente dicha, se utilizan dos tipos de punteros a datos: puntero-a-compartida (Del inglés: pointer-to-shared) y puntero privado. Ambos tipos de punteros pueden ser ubicados y liberados de forma dinámica.

Un puntero privado puede acceder a la zona privada del hilo y también a su zona compartida. Su declaración es similar a ISO C.

Los punteros-a-compartida pueden acceder al espacio compartido de cualquier hilo. Para declarar una variable de este tipo se utilizan los modificadores *static shared.* Cuando se utiliza una variable compartida no hay garantías del orden de ejecución de los hilos, por lo que una opción común es utilizar un array en el que en cada posición hay una copia de la variable propiamente dicha, de modo que cada hilo pueda modificarla sin afectar a los demás.

## 4.3 Sincronización y consistencia de la memoria

El método más simple de sincronización con el que cuenta UPC es lo que se denomina upc_barrier. Esta sentencia hace que todos los hilos esperen a que el resto lleguen a ese punto.

```
// bloque de sentencias 1

upc_barrier;

//bloque de sentencias 2
```

**Ejemplo 4.4 Sincronización mediante barreras UPC**

En el ejemplo 4, ningún hilo comenzará la ejecución del bloque de sentencias 2 hasta que todos hayan ejecutado el bloque 1.

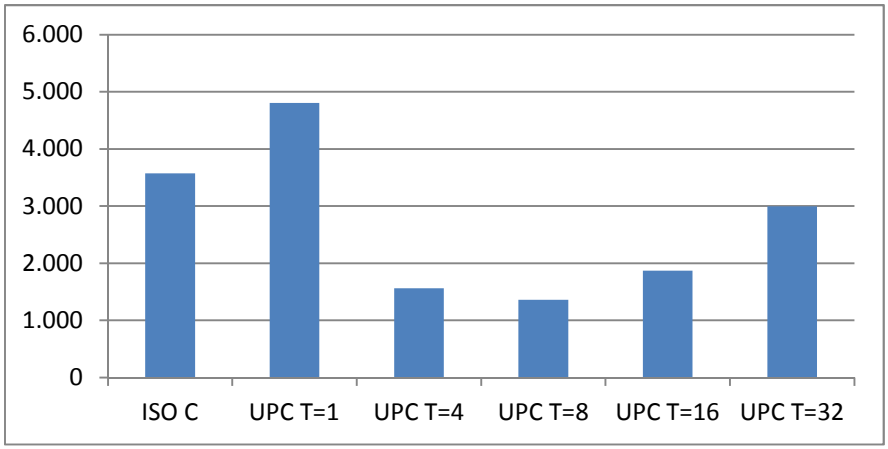
Aunque existen más mecanismos de sincronización, como cerrojos y gran parte de los mecanismos clásicos, estos se dejan fuera de los límites de este documento.

### 4.4 Ventajas de la utilización de UPC

Para comprobar que realmente la ejecución es más rápida utilizando UPC que C estándar, se probará un pequeño programa que multiplica una matriz por un vector (una modificación del utilizado en prácticas de Arquitectura y Tecnología de Computadores en la Universidad de Oviedo) y se medirá el tiempo de ejecución de 1000 matrices de 200x200, utilizando ISO C, y UPC utilizando 1, 4, 8 y 16 hilos. En los Anexos I y II se puede ver el código utilizado. Estos tiempos son resultado de la ejecución en una máquina con un procesador de 4 núcleos con tecnología hyperthreading.

| Lenguaje | Tiempo de ejecución(s) |
|---|---|
| **ISO C** | 3.570 |
| **UPC T=1** | 4.805 |
| **UPC T=4** | 1.560 |
| **UPC T=8** | 1.358 |
| **UPC T=16** | 1.872 |
| **UPC T=32** | 2.995 |

**Tabla 4.1 Resultados de la ejecución**




**Grafica 4.1 Resultados de la ejecución**

A pesar de tener un rendimiento menor si se ejecuta con un único hilo, a medida que el número de hilos aumenta, lo hace también el rendimiento hasta alcanzar una cota máxima en el número de hilos que se pueden ejecutar de forma simultánea. Si a partir de esta cota se sigue aumentando el número de hilos, el rendimiento vuelve a decrecer, pudiendo llegar a empeorar el rendimiento original.

## 5 Conclusiones

El empleo de técnicas de programación que hacen uso de memoria compartida supone una inevitable complicación en el proceso de programación, depuración y detección de errores. A pesar de ello las ventajas son más que notorias, y permite maximizar la concurrencia y el paralelismo, y por tanto el rendimiento de las máquinas, tanto en arquitecturas multiprocesador como en monoprocesador. En las arquitecturas multiprocesador, obviamente, se maximizan los recursos de computación y en las monoprocesador ayuda a evitar que determinadas operaciones acaparen el uso de CPU durante más tiempo del debido.
El empleo de estas técnicas de programación ha ido aumentando a lo largo de los años llegando a convertirse en una norma día de hoy. La programación orientada a memoria compartida, como cualquier otra técnica dentro del campo de la informática ha sufrido un proceso evolutivo. Dentro de este proceso evolutivo pueden reseñarse técnicas como la paralelización de código fuente en el proceso de compilación (compiladores paralelos), el uso de librerías estándar incluidas en la API´s de las diferentes arquitecturas (POSIX pthreads, JAVA Threads) o el desarrollo de nuevos lenguajes de programación orientados a la producción de código en paralelo. En el presente estudio se han analizado las diferentes técnicas, y como, fruto del proceso evolutivo, estas técnicas han conseguido lograr la optimización de los recursos existentes.

## 6 Bibliografía

1. Cory Quammen, “Introducción a la Programación en ambientes con Memoria Compartida y Memoria-Distribuída”, http://bc.inter.edu/facultad/jyeckle/cursos/parallel%20computing/papers/2-%20paradigma%20threads%20y%20mpi.pdf

2. Johm Mellor Crummey, “Programming Shared-memory Platforms with Pthreads”, Enero 2011, http://www.clear.rice.edu/comp422/lecture-notes/comp422-2011-Lecture6-Pthreads.pdf

3. Manish Parashar, “Shared Memory Programming: Threads”, 1998, http://www.ece.rutgers.edu/~parashar/Classes/98-99/ece566/slides/lecture_pthreads.PDF

4. Domingo Gimenez Cánovas, “Programación paralela: Lenguajes y Modelos”, 2003, http://www.slidefinder.net/p/programación_paralela_tema_lenguajes_modelos/15238786

5. S. Dasgupta, “Computer Architecture: A Modern Synthesis” (Tomo 2), John Wiley & Sons, 1989.

6. J.L. Hennessy y D. A. Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kauffman Publishers, 2002.

7. Apuntes de Programación paralela de la Universidad Politécnica de Valencia (Temas 15 y 16), http://cgi.di.uoa.gr/~halatsis/Advanced_Comp_Arch/UPoValencia/t15.pdf y http://cgi.di.uoa.gr/~halatsis/Advanced_Comp_Arch/UPoValencia/t16.pdf

8. Ruud Van der Pass, “Basic concepts in Paralellization”, 2010, http://www.compunity.org/training/tutorials/2%20Basic_Concepts_Parallelization.pdf

9. Ruud Van der Pass, “An overview to OpenMP”, 2010.

10. B. Chapman, G. Jost, R. van der Pas, “Using OpenMP”, October 2007.

11. D. Giménez Cánovas y Javier Cuenca, “Programación en Memoria Compartida”, 2005, http://dis.um.es/~domingo/doctorado/0506/ComMatParUPCT/Sesion7ProMemCom.pdf

12. Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers Barry Wilkinson and Michael Allen  Prentice Hall, 1999

13. T. El-Gazawi, W. Carlson, T. Sterling, K. Yelick, “UPC Distributed Shared Memory Programming”, Wiley-Intersign, 2005

14. T. El-Gazawi, “Unified Paralell C – UPC Tutorial”, 2009, The George Washington University.

# 7 Apéndices

## 7.1 Apéndice I

Código de mxvint.upc

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <upc.h>

#define N   1000
#define M   1000

static shared int X[N], A[M*N], Y[M];
void inicializa(double X[N],double A[M*N],int fil,int col,int Y[M]);
double dtime();
```

```
static shared double start, end;
static shared int num;
static shared int got_end;
main()
{
    FILE *fiy;
    int i,j, fil, col;
    fil = 200;
    col = 200;
    got_end = 0;
    if (MYTHREAD == 0) {
                    printf("\n Inicializando la matriz....");
                    inicializa(X,A,fil,col,Y);
                    /* Toma del tiempo inicial justo antes de empezar el cálculo */
                    printf("\n Iniciando el cálculo....");
          }
          upc_barrier;
          start = dtime();
          upc_barrier;

          /* Multiplicacion*/
          for (num=MYTHREAD; num<1000; num+=THREADS)
                    for (i=0; i<fil; i++)
                              for (j=0; j<col; j++)
                                        Y[i]= Y[i] + A[i*col+j]*X[j];
          upc_barrier;
          if (got_end == 0){
                    end = dtime();
                    got_end = 1;
          }
          if (MYTHREAD == 0){
                    printf("End: %lf\n", end);
                    printf("\n\n Tiempo total de cálculo: %.2lf  sec \n", (end-start));
          }
}
void inicializa(int X[N],int A[M*N],int fil,int col,int Y[M])
{
    int i,j,num;
    double pp;
    for (i=0;i< fil;Y[i++]=0);
    srand(24);
    for (i=0; i<fil; i++)
    {
        for (j=0; j<col; j++)
        {
             pp=rand();
             num=((rand()*10) > 5)?-1:1;
             A[i*col+j]= (int)pp/(rand()*num+1);
        }
    }
    for (i=0; i<col; i++)
    {
         pp=rand();
         num=((rand()*10) > 5)?-1:1;
         X[i]= (int)pp/(rand()*num);
    }
}
/*****************************************************/
/*  UNIX dtime(). This is the preferred UNIX timer.  */
/*  Provided by: Markku Kolkka, mk59200@cc.tut.fi    */
/*  HP-UX Addition by: Bo Thide', bt@irfu.se         */
/*****************************************************/
#include <sys/time.h>
#include <sys/resource.h>

struct rusage rusage;

double dtime()
{
   double q;
   getrusage(RUSAGE_SELF,&rusage);
   q = (double)(rusage.ru_utime.tv_sec);
   q = q + (double)(rusage.ru_utime.tv_usec) * 1.0e-06;
 return q;
}
```