Práctica 1: Memoria compartida

Sección 2: Paralelización

En esta sección el alumno realizará varias implementaciones paralelas de la versión secuencial de multiplicación de matrices. El objetivo es explorar distintas opciones de paralelización, para poder elegir la que mejor se adapte al problema dado.

Por último se realizará un análisis de rendimiento de las paralelizaciones realizadas. Para ello se realizarán varias tomas de tiempos de cada ejecución, y se compararán contra la versión secuencial.

Primera parte: Codificación de las versiones paralelas

Se van a implementar tres esquemas de reparto de trabajo:

Esquema “A lo loco”:

En este esquema se propone generar el máximo número de threads posible (hay una limitación por el sistema operativo) para realizar la multiplicación. Para eso, tomando como base la función “multiplicar matrices”, desde ahí se ejecutará la función “multiplicaVector” en un thread aparte, y se generarán “FilasxColumnas” threads. (Ej: Una matriz 100x100 generará 10000 threads de multiplica vector),

- Opciones:

Para facilitar el desarrollo, se pueden colocar los datos de las matrices de entrada y de la resultado en variables globales , de forma que cada thread solo recibe el número de la fila y la columna que debe de multiplicar, y dónde debe escribirlo.

Otra opción sería crear una estructura “parámetros” que almacene los punteros a los vectores que deben de multiplicarse. Se creará una estructura por cada uno de los threads, y se pasará por parámetros en pthread_create.

Para devolver los datos resultado se puede usar la función “pthread_exit”, o bien escribirlos directamente en la matriz resultado.

Esquema “División estática”:

El esquema anterior presenta varios problemas. Por un lado se gasta mucho tiempo creando threads cuando crece el tamaño de las matrices, y por otro se necesitan estructuras extra para los parámetros de configuración de cada thread.

Una solución más adecuada para los sistemas multiprocesador es el esquema de división de trabajo estático:

- Sólo se crea un thread por cada uno de los procesadores disponibles.
- El trabajo de entrada se divide de forma equilibrada entre cada uno de los threads.
- Para cada thread, se crea un "paquete de configuración", que usará para saber qué trozo de las matrices debe multiplicar.

Por ejemplo, si se multiplicaran dos matrices de 1000x1000, se realizarían 1000000 de multiplicación de vectores (uno por cada fila/columna). Si se usaran cuatro procesadores, cada hilo de ejecución multiplicaría únicamente 250000 vectores, siendo todos iguales en tamaño.

Para eso, se propone paralelizar la función "multiplicar matrices" en vez de "multiplicar vectores". Cada thread recibirá una estructura de tipo "paquete de trabajo" que contendrá los siguientes datos:

- Datos Matriz1
  - Vectores
  - Fila Inicial
  - Fila Final
- Datos Matriz2
  - Vectores
  - Columna Inicial
  - Columna Final
- Matriz resultado

Antes de crear los threads, se creará un array de "paquetes de trabajo" que contendrá los datos para cada hilo. Se puede crear una función "Crear paquetes de trabajo" que realice el reparto de trabajo y rellene las estructuras para cada thread.

Por ejemplo, si se crearan 4 threads para multiplicar dos matrices de 1000x1000, la función "Crear paquetes de trabajo" crearía un array de 4 paquetes. La configuración del tercer thread sería la siguiente:

- Datos Matriz1
  - Fila Inicial = 500
  - Fila Final = 749
- Datos Matriz2
  - Columna Inicial = 0
  - Columna Final = 999

Esquema “Balanceo de carga”

Aunque el esquema anterior soluciona muchos de los problemas, podría ocurrir que alguno de los threads realizara más trabajo que el resto. Esto es debido a que no se conoce el “peso” de los paquetes de trabajo de antemano. En caso de que un thread quedara desbalanceado, retrasaría la ejecución global del programa.

Para solucionarlo, se propone un esquema en el que se crean más paquetes de trabajo que threads, y cuando un thread acaba pasa a buscar un nuevo trabajo. Por ejemplo, si se disponen de 3 threads, se pueden generar 100 trabajos similares a los anteriores, almacenados en una lista global. Cuando el thread acabe, accederá a esa lista para sacar un nuevo paquete de trabajo. Si la lista está vacía, el thread terminará su ejecución.

El problema que aparece es el acceso a la lista de trabajos, ya que se necesita actualizar el número de trabajos disponible en paralelo. Para solucionarlo, se usará un cerrojo que protegerá la lista de trabajos, serializando los accesos.

El alumno deberá encontrar un “tamaño de trabajo” que se adapte bien al número de threads usado, de forma que los accesos a la lista de trabajos no penalicen el tiempo de ejecución. El esquema de ejecución quedará de la siguiente manera:

- Inicio (main)
- Crear lista de trabajos (lista enlazada o array)
    - Generar “M” trabajos, siendo “M” mayor que el número de threads
- Por cada thread:
    - Mientras (lista de trabajos llena)
        - Sacar un trabajo de la lista
        - Ejecutar
        - Guardar resultado

Segunda parte: Evaluación

La evaluación de esta parte se realizará con tomas de tiempo reales, calculando el speedup de la nueva implementación respecto de la versión secuencial. El alumno deberá generar una gráfica de escalabilidad que refleje las mejoras introducidas y que demuestre que el uso de un número mayor de procesadores permite una reducción de tiempos. Para ello, se realizarán diversas tomas de tiempos de los programas implementados. El alumno deberá realizar las siguientes tomas de datos:

- Para cada uno de los tres esquemas implementados, compilar sin usar los flags de prifiling (dejar “-g”, quitar “-pg” wn los makefiles) , y hacer una ejecución con las matrices de “100x100”, “1000x1000” y “1000x1000”:
    - Crear un ejecutable que use 1, 2, 4 y 8 procesadores.
    - Para cada uno de los ejecutables generados recoger:
        - Tiempo total de ejecución.
        - Tiempos de ejecución sin tener en cuenta la carga de datos (lecturas/escrituras en fichero)

Una vez recogidos los tiempos de ejecución, calcular el speedup real respecto de la versión secuencial. Para eso se usará la siguiente fórmula:

Speedup= (Tiempo Secuencial)/(Tiempo Paralelizado)

En total se generarán 6 gráficas de escalabilidad, que se agruparán en dos gráficas a la hora de realizar la memoria:

- Una gráfica de escalabilidad “total” de los tres esquemas:
    - Mostrará el Speedup real usando los tiempos totales de ejecución (incluyen carga/guardado de datos) de las versiones paralelas respecto de la secuencial, usando 1, 2, 4 y 8 procesadores
- Una segunda gráfica igual a la anterior, pero usando los tiempos de ejecución sin tener en cuenta la carga de datos.

Una vez se tengan las gráficas, el alumno decidirá cuál de los tres esquemas implementados se adapta mejor al ordenador usado (cual tiene mejor Speedup y qué número de procesadores es el más adecuado). Reflejará sus conclusiones en un párrafo al final de la memoria, en el que explique los problemas de escalabilidad y razone sus conclusiones.

APÉNDICE: Toma de tiempos

El alumno implementará un sistema software que permita tomar tiempos dentro de su programa. Para eso, podrá utilizar las librerías que desee, a continuación se detalla el uso de las librerías proporcionadas por el profesor.

- Gprof
  El mismo programa GProf proporciona una estimación de tiempos de ejecución de cada una de las partes del programa. Esta estimación es útil cuando se dispone de un programa secuencial (uniprocesador), pero los datos no son fiables en entornos multiprocesador. Por eso se aconseja

usarlo solo para la toma de tiempos de la versión original de la implementación.

- Cabecera “debug_time.h”
  Junto con esta memoria se proporciona una cabecera con diversas macros que permiten realizar una toma de tiempo de un trozo de programa. A continuación se muestra un ejemplo de uso:

```
“archivo.cpp”

#include “debug_time.h”
….
Int main(int argc, char** argv)
{
        DEBUG_TIME_INIT;   //Inicializa las variables de toma de tiempos
        ….
        DEBUG_TIME_START; // Inicia la toma de tiempos
                //Código que se quiere medir
        DEBUG_TIME_END;
        DEBUG_PRINT_FINALTIME(“Tiempo Total: ”);
        return 0;
}
```

El código anterior tomará tiempos en una sección de llamadas del main. Si se quieren realizar dos tomas de tiempos (una global, y otra de una función en concreto), se pueden anidar llamadas a “DEBUG_TIME_INIT” en distintas subsecciones del código. Un ejemplo con dos tomas de tiempos anidadas sería la siguiente:

```
“archivo.cpp”

#include “debug_time.h”
```

```
....
Int main(int argc, char** argv)
{
        DEBUG_TIME_INIT;   //Inicializa las variables de toma de
tiempos
        ....
        DEBUG_TIME_START; // Inicia la toma de tiempos
                //Código que se quiere medir
                {       //Nueva sección de código, importante usar
llaves
                        DEBUG_TIME_INIT;
                        DEBUG_TIME_START;
                                Func2();
                        DEBUG_TIME_END;
                        DEBUG_PRINT_FINALTIME("Tiempo     func2():
");
                }
        DEBUG_TIME_END;
        DEBUG_PRINT_FINALTIME("Tiempo Total: ");
        return 0;
}
```

De esta manera, se tendrán los tiempos de ejecución de todo el programa separados de los tiempos de ejecución de una función en concreto.