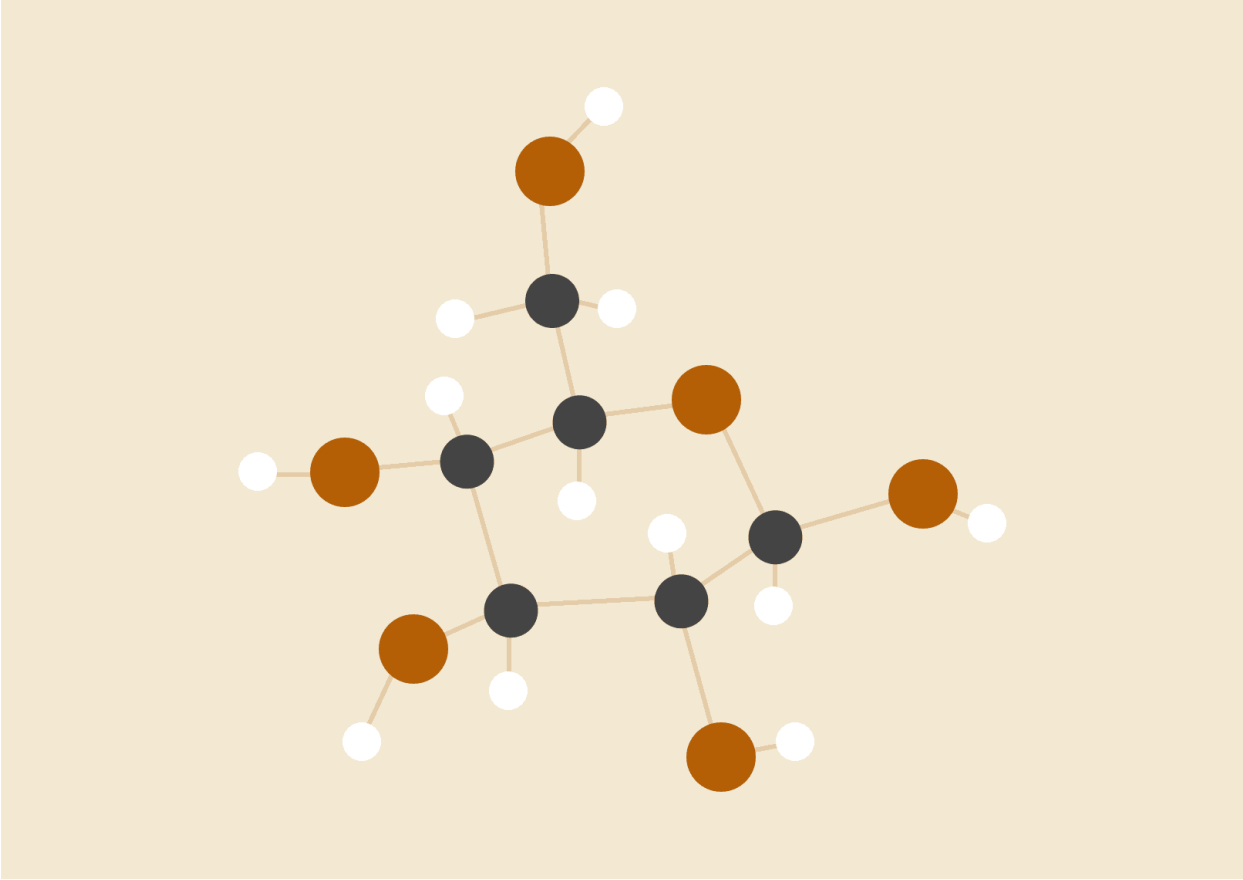
Entrega 2

Programación en memoria compartida



Grupo 13

Del Grosso, Augusto

Sánchez Magariños, Juan Pablo

07.05.2021

UNLP

INTRODUCCIÓN

Este trabajo es la continuación de la Entrega 1, en donde ideamos distintas técnicas para realizar un calculo de matrices en el lenguaje C.

En la Entrega 1 realizamos 3 algoritmos diferentes para resolver el problema: Uno que tilizaba la multiplicacion convencional, que llamaremos **`msec`**, uno utilizando la multiplicación por bloques, que llamaremos **`bsec`**, y otro que realizaba los calculos por medio de sucesivos llamados a una función. Este último lo descartamos por los resultados obtenidos en dicha entrega.

Es esta entrega lo que realizaremos es la implementación en paralelo de dichos algoritmos, utilizando las librerias Pthreads y OpenMP. Luego se realizá un análisis de rendimiento para estudiar cómo mejorar esta técnica a nuestros algoritmos.

Un informe en PDF que describa brevemente las s**oluciones planteadas**, **análisis de resultados** y **conclusiones**. Debe incluir el **detalle del trabajo experimental** (características del hardware y del software usados, pruebas realizadas, etc), además de las **tablas** (y posibles gráficos, en caso de que corresponda) con los tiempos de ejecución, Speedup y Eficiencia. El **análisis de rendimiento** debe hacerse tanto en forma individual a cada solución paralela como en forma comparativa.

TECNOLOGÍAS UTILIZADAS

```
# Para compilar el código
      - gcc version 10.2.0 (Ubuntu 10.2.0-13ubuntu1)
      - NPTL 2.32
# Para editar y compartir el código
      - Visual Studio Code 1.55.2
      - git 2.27.0
# Equipo Hogareño:
      - OS: Ubuntu 20.10 groovy
      - Kernel: x86_64 Linux 5.8.0-50-generic
      - Shell: bash 5.0.17
      - CPU: Intel Core i3-3110M @ 4x 2,4GHz [52.0°C]
      - GPU: Mesa DRI Intel(R) HD Graphics 4000 (IVB GT2)
```

```
      -  RAM: 2928MiB / 3813MiB
# Cluster:
      -  Ocho (8) nodos de procesamiento, cada uno con:
            ■  8GB de memoria RAM
            ■  Dos procesadores Intel Xeon E5405 a 2.0GHz de 4 cores
      -  1Gbit Ethernet
```

DESARROLLO

Algoritmos secuenciales

Corregimos algunos errores mínimos que habían quedado en los algoritmos y modificamos algunas variables y procedimientos para acoplarnos a la modificación de la consigna, específicamente la adición de una nueva matríz.

Algoritmos Pthread

Para paralelizar nuestros algoritmos, el primer obstáculo que nos encontramos fue cómo dividir la carga de trabajo entre los hilos.

En **`mpar`**, paralelismos **`msec`** y no fué un gran desafío, pudimos ver inmediatamente que lo más natural sería dividir las matrices *por filas.* Es decir, ya que siempre que trabajamos con uno de estos arreglos, hacemos primero un for por filas, podríamos dividir este recorrido en partes iguales para cada core. Además gracias a las propiedades matemáticas de la multiplicación de matrices, para realizar el cálculo de elemento $(R_1*A)_{i\ j}$ se necesita toda la fila **`i`** de la matriz R_1 y toda la columna **`j`** de la matriz A.

Es por eso que es importante que la matriz R_1 tenga toda la fila cargada al momento de calcular ese elemento. Dividiendo la carga de trabajo por filas y haciendo que cada hilo calule primero las filas que le corresponde de R_1 y *luego* haga las mismas filas de la matriz producto **R_1A,** nos aseguramos de que tiene ya cargada toda la fila que necesita para ese producto.

Si hubiese que calcular también las matrices que aparecen a la derecha de los productos (las que se recorren por columnas) sería un problema ya que no se dispondría de la columna en su totalidad, y habría que pensar otro tipo de sincronización. Pero esto no sucede ya que son datos que se cargan al inicio, están completamente disponibles por todos los hilos.

Específicamente en este caso, teniendo **`N`** filas, simplemente se divide por la cantidad de

hilos, y se le asigna a cada uno recorre las filas **`i`** desde `id*N/T` hasta `(id+1)*N/T`.

A tener en cuenta, que el cálculo de los promedios. En el algoritmo secuencial se acumulaban los valores en una variable que se actualizaba en cada iteración de los for. Siendo ahora esta una variable compartida, deberían guardarse con un mutex y sería muy costoso tomarlo en todas las iteraciones, por lo que es mejor calcularlo de manera local al hilo y actualizar la variable compartida una sola vez cada uno. Como son dos promedios a calcular de dos matrices distintas, utilizamos dos *mutex* para maximizar la concurrencia.

También es importante para el cálculo de la matríz C que estén totalmente calculados los promedios, por eso además es necesaria una barrera para sincronizar el comienzo de dicho cálculo.

En **`bpar`** lo que cambia es que en lugar de distribuir filas, se distribuyen *tiras*, es decir un conjunto de filas del ancho del bloque seleccionado. Cabe aclarar que en el trabajo anterior determinamos empíricamente que el tamaño de bloque óptimo para ésta arquitectura es de 64x64. Por esa razón utilizamos esta medida para todos nuestros cálculos.

Algoritmos OpemMp

mopenmp

Decidimos tomar el algoritmo **`msec`** y gracias a las prestaciones que ofrece OpenMP, simplemente agregando algunas cláusulas entre líneas se puede paralelizar.

Previo a cada for agregamos la directiva `#pragma omp parallel for private`, definiendo las variables locales que corresponden en cada caso con la cláusula private. La librería se encarga de distribuir el trabajo equitativamente por defecto static.

Luego para el caso de la iteración en donde se calcula el promedio agregando la cláusula `reduction (+:avgR1)`, se consigue que se totalice el valor de la variable compartida con los valores de una copia local de esa variable.

Se utiliza `#pragma omp single` para dividir el total por **`n*n`** con lo cual solo uno de los hilos lo realiza, ahorrando quizás algunos cálculos al resto.

Al haber una barrera implícita en cada una de esas directivas, no fue necesario explicitar antes de calcular la matriz resultado **`C`**.

openmp

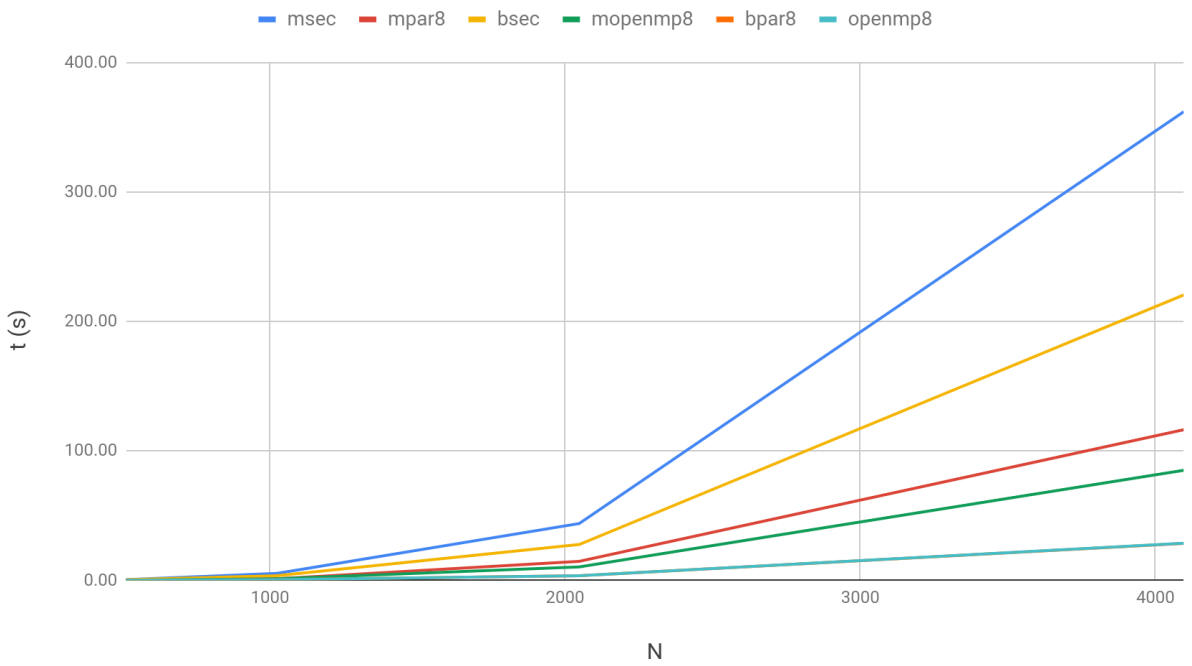
Basándonos en la idea de Pthread para el algoritmo por bloques, decidimos seguir utilizando la función `calculate()` para paralelizar. En ella implementamos la directiva `#pragma omp parallel private(i,j,k,ii,... )` para indicar la porción de código a ejecutar entre los hilos y las variables privadas para los mismos.

De esta manera todo el procesamiento de la función es paralelo, por lo que al llegar a la sección de 'informar' el promedio que calcula cada hilo, se deble programar explicitamente el acceso a una sección crítica. Para ello utilizamos la cláusula **`#pragma omp critical`**. Seguido a esto encontraremos la última cláusula utilizada en nuestro código, **`#pragma omp barrier`** que cumple el mismo rol que en código de Phread, esperar a que todos los hilos generen la información necesaria para calcular C.

Validación de los resultados

Para realizar la verificación simplemente se reinicializan las matrices calculadas, excepto C que se calculará sobre otra variable C_CHECK, y se realizan el procesamiento secuencialmente, habiendo detenido la medida del tiempo anteriormente. Luego se comparan las dos matrices resultado, elemento a elemento.

Resultados



TIEMPO						
Algoritmo		Threads	N			
			512	1024	2048	4096
Multiplicación convencional	secuencial		0.45	5.32	43.83	361.80
	Pthreads	1	0.51	5.29	42.43	353.91
		2	0.26	3.01	26.08	211.48
		4	0.13	2.41	20.64	171.10
		8	0.07	1.19	14.70	116.30
	OpenMP	1	0.46	5.13	42.61	352.00
		2	0.23	3.38	25.66	209.71
		4	0.12	2.38	20.63	166.46
		8	0.06	1.19	10.32	84.94
Bloques x 64	secuencial		0.46	3.53	27.65	220.42
	Pthreads	1	0.47	3.56	27.90	222.32
		2	0.24	1.78	13.97	111.93
		4	0.12	0.90	7.05	56.06
		8	0.06	0.46	3.59	28.53
	OpenMP	1	0.46	3.56	27.90	222.45
		2	0.23	1.79	13.96	111.27
		4	0.12	0.91	7.05	56.15
		8	0.06	0.46	3.58	28.59

SPEEDUP						
Algoritmo		Threads	N			
			512	1024	2048	4096
multip convencional	Pthreads	1	0.89	1.01	1.03	1.02
		2	1.76	1.77	1.68	1.71
		4	3.46	2.21	2.12	2.11
		8	6.77	4.48	2.98	3.11
	OpenMP	1	0.98	1.04	1.03	1.03
		2	1.95	1.57	1.71	1.73
		4	3.86	2.24	2.12	2.17
		8	7.46	4.46	4.25	4.26
Bloques x 64	Pthreads	1	0.99	0.99	0.99	0.99
		2	1.95	1.98	1.98	1.97
		4	3.90	3.91	3.92	3.93
		8	7.53	7.64	7.70	7.73
	OpenMP	1	0.99	0.99	0.99	0.99
		2	1.97	1.98	1.98	1.98
		4	3.90	3.90	3.92	3.93
		8	7.65	7.66	7.72	7.71

EFICIENCIA						
Algoritmo		Threads	N			
			512	1024	2048	4096
Multiplicación convencional	Pthreads	1	0.89	1.01	1.03	1.02
		2	0.88	0.88	0.84	0.86
		4	0.87	0.55	0.53	0.53
		8	0.85	0.56	0.37	0.39
	OpenMP	1	0.98	1.04	1.03	1.03
		2	0.98	0.79	0.85	0.86
		4	0.97	0.56	0.53	0.54
		8	0.93	0.56	0.53	0.53
Bloques x 64	Pthreads	1	0.99	0.99	0.99	0.99
		2	0.98	0.99	0.99	0.98
		4	0.98	0.98	0.98	0.98
		8	0.94	0.96	0.96	0.97
	OpenMP	1	0.99	0.99	0.99	0.99
		2	0.99	0.99	0.99	0.99
		4	0.98	0.97	0.98	0.98
		8	0.96	0.96	0.96	0.96

CONCLUSIÓN

Podemos decir que en el algoritmo de multiplicación por bloque encontramos unos altos índices de mejora con respecto al secuencial al incorporar más hilos. Se pudo optimizar gran parte del código, y maximizamos la concurrencia generando la mínima cantidad de barreras. Esto sumado a las ventajas antes mencionadas del aprovechamiento del principio de localidad que otorga el procesamiento por bloques se vio reflejado en los valores de Seedup casi perfectos.

Por otro lado en el algoritmo de multiplicación convencional no le saca provechoal mencionado principio, y por lo tanto se podía esperar que no presente gran diferencia al paralelizar. También cabe agregar que en mopenmp se utilizan más barreras para realizar el programa, lo que no permite maximizar concurenciay se ve afectada la eficiencia del programa.