Universidad Nacional de La Plata
Facultad de Informática
Sistemas Paralelos

Informe Trabajo Práctico 1 | Optimización de Algoritmos Secuenciales

Por Franco Kumichel y Norberto Ariel Chaar

Introducción

Se propone realizar un informe que describa brevemente las soluciones planteadas, análisis de resultados y conclusiones. El informe incluye el detalle del trabajo experimental (características del hardware y del software usados, pruebas realizadas, etc), con los tiempos de ejecución.

Características del hardware y software usados:

- Instancia EC2 en AWS con 4 vCPU - Instance type: “m5a.xlarge”; Platform: Ubuntu; AMI: ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20240301
- Clúster provisto por la Cátedra de Sistemas Paralelos de la Facultad de Informática UNLP

Resolución

1. Resuelva el ejercicio 6 de la Práctica Nº 1 usando dos equipos diferentes: (1) cluster remoto y (2) equipo hogareño al cual tenga acceso (puede ser una PC de escritorio o una notebook):

6. Dada la ecuación cuadrática: $x^2 - 4.0000000\ x + 3.9999999 = 0$, sus raíces son $r_1 = 2.000316228$ y $r_2 = 1.999683772$ (empleando 10 dígitos para la parte decimal).

   a. El algoritmo *quadratic1.c* computa las raíces de esta ecuación empleando los tipos de datos *float* y *double*. Compile y ejecute el código. ¿Qué diferencia nota en el resultado?
   b. El algoritmo *quadratic2.c* computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución?
   c. El algoritmo *quadratic3.c* computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución? ¿Qué diferencias puede observar en el código con respecto a *quadratic2.c?*

   Nota: agregue el flag *-lm* al momento de compilar. Pruebe con el nivel de optimización que mejor resultado le haya dado en el ejercicio anterior.

a) Los resultados obtenidos tanto para el cluster como el equipo hogareño fue el siguiente:

Soluciones Float: 2.00000 2.00000
Soluciones Double: 2.00032 1.99968

Lo cual demuestra la diferencia de precisión entre los tipos de datos float (32 bits) y los tipos de datos double (64 bits) en C. Hay que tener en cuenta la importancia de la precisión en los datos, ya que si tomaramos tipos de datos float para este caso por ejemplo, llegaríamos a que la ecuación posee una sola raíz y sería erróneo.

Cabe resaltar que para este ejercicio no hemos activado ningún flag de optimización a la hora de realizar las compilaciones, ya que para este caso el resultado no se verá afectado si se utiliza un optimizador u otro, o si directamente no se utiliza.

Optimizadores

Antes de proseguir con las demás soluciones a los problemas planteados, realizaremos diferentes pruebas sobre los archivos quadatric2.c y quadatric3.c de modo de decidir si es factible utilizar optimizadores, y en caso de ser así, cuál utilizar.

Primero daremos una breve descripción del flag -O. Este flag controla el nivel de optimización de todo el código. Al cambiar este valor, la compilación de código tomará algo más de tiempo, y utilizará mucha más memoria, especialmente al incrementar el nivel de optimización. Algunos de los diferentes niveles de optimización son los siguientes:

- -O0: Es el predeterminado si no se especifica ningún nivel -O. Lo que significa que el código no recibirá ningún tipo de optimización.

- -O1: Es el nivel de optimización más básico, se encarga principalmente de generar un código rápido y pequeño en tamaño sin generar demasiadas demoras en el tiempo de compilación.
- -O2: Posee todas las características brindadas por -O1 y además, intenta aumentar el rendimiento del código sin comprometer el tamaño y el tiempo de compilación empleado
- -O3: Posee todas las características brindadas por -O2 y además activa optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. No garantizando una mejora de rendimiento, pudiendo producir el efecto contrario al ralentizar un sistema por el uso de binarios de gran tamaño generados. Una de las mejoras es la vectorización dentro de los bucles utilizados.

Dada una breve descripción, lo que se procede a realizar son pruebas en la ejecución de los algoritmos en quadatric2.c y quadatric3.c activando los diferentes optimizadores y comparar los diferentes tiempos de ejecución, obteniéndose los siguientes resultados:

Tipo solución / Optimizador utilizado	Sin optimizador	-O1	-O2	-O3
Float	71.873961	2.661472	2.433929	2.434929
Double	68.709121	1.964033	1.618787	1.591017

Tiempos de quadatric2 con TIMES = 100 para diferentes niveles de optimización en equipo hogareño

Tipo solución / Optimizador utilizado	Sin optimizador	-O1	-O2	-O3
Float	46.255611	6.148494	5.976894	5.988643
Double	45.132759	6.710286	6.577648	6.659155

Tiempos de quadatric2 con TIMES = 100 para diferentes niveles de optimización en clúster remoto

Tipo solución / Optimizador utilizado	Sin optimizador	-O1	-O2	-O3
Float	48.235812	1.585675	1.352355	1.349416
Double	68.692554	2.002110	1.593738	1.581110

Tiempos de quadatric3 con TIMES = 100 para diferentes niveles de optimización en equipo hogareño

Tipo solución / Optimizador utilizado	Sin optimizador	-O1	-O2	-O3
Float	69.136530	3.482941	3.398769	3.399773
Double	51.625041	6.716951	6.592315	6.640922

Tiempos de quadatric3 con TIMES = 100 para diferentes niveles de optimización en clúster remoto

En las pruebas realizadas puede observarse que el tiempo de ejecución se reduce considerablemente cuando se activan las opciones de optimización. Además, los tiempos de ejecución más reducidos se obtienen con el nivel de optimización -O3, por lo que se ha decidido de acá en adelante utilizar este nivel de optimización

b) Ejecutamos el algoritmo quadatric2.c en el equipo hogareño y clúster remoto, obteniendo los siguientes resultados:

Tipo de datos / TIMES	10	100	1000
Float	0.242445	2.424053	24.324350
Double	0.161422	1.594025	15.870091

Tiempos de quadatric2 variando TIMES en equipo hogareño

Tipo de datos / TIMES	10	100	1000
Float	0.598414	5.989152	59.780431
Double	0.660737	6.646245	65.786873

Tiempos de quadatric2 variando TIMES en cluster remoto

Se puede notar que en el equipo hogareño la solución con tipo de dato double se calcula un 15% más rápido en promedio que el tipo de dato float, sin embargo en el cluster remoto la solución con tipo de dato float se calcula un 11% más rápido en promedio que la solución con el tipo de dato double.

c) Ejecutamos el algoritmo quadatric3.c en el equipo hogareño y clúster remoto, obteniendo los siguientes resultados:

Tipo de datos / TIMES	10	100	1000
Float	0.134844	1.349738	13.499587
Double	0.157179	1.578098	16.016261

Tiempos de quadatric3 variando TIMES en equipo hogareño

Tipo de datos / TIMES	10	100	1000
Float	0.340704	3.396154	34.302477
Double	0.666113	6.579724	66.595339

Tiempos de quadatric3 variando TIMES en cluster remoto

Puede observarse que el tiempo requerido para la solución usando datos de tipo double es mayor que el tiempo requerido para la solución usando datos de tipo float, ya que el double utiliza más memoria y requiere más tiempo de procesamiento para realizar cálculos de precisión doble. Además se aprovechan las funciones powf (para calcular la potencia) y sqrtf (para

calcular la raíz cuadrada). Estas funciones provistas por la librería “math.h”, son específicas para el tipo de dato float, las resuelve de manera más eficiente, y como consecuencia mejorando los tiempos de ejecución. Otro aspecto que mejora los tiempos de ejecución es que se declaran variables globales de tipo float, así teniendo que evitar un casteo de tipo de datos.

2. Desarrolle un algoritmo en el lenguaje C que compute la siguiente ecuación:

$$R = \frac{(MaxA \times MaxB - MinA \times MinB)}{PromA \times PromB} \times [A \times B] + [C \times D]$$

− Donde A, B, C, D y R son matrices cuadradas de NxN con elementos de tipo double.
− MaxA, MinA y PromA son los valores máximo, mínimo y promedio de la matriz A, respectivamente.
− MaxB, MinB y PromB son los valores máximo, mínimo y promedio de la matriz B, respectivamente.
Mida el tiempo de ejecución del algoritmo en el clúster remoto. Las pruebas deben considerar la variación del tamaño del problema (N={512, 1024, 2048, 4096}). Por último, recuerde aplicar las técnicas de programación y optimización vistas en clase.
Aclaración: el acceso remoto puede sufrir caídas por alteraciones en la provisión de energía eléctrica. Ante estos casos, si bien se hará lo posible por resolverlas rápido, se recomienda no dejar las pruebas para último momento.

Se confeccionó el algoritmo *matrices.c*, para el cual se tuvieron en cuenta los siguientes aspectos para la realización y optimización del mismo:

- Se decidió definir a las matrices como **‘arreglos dinámicos como vector de elementos’** principalmente para poder beneficiarse de las características que esta ofrece, como lo son el permitir recorrer la matriz por filas o columnas o el aprovechar que todos sus datos sean almacenados contiguos en memoria mediante diferentes algoritmos de optimización.

- Se decidió tener en cuenta el orden en el que se almacenaban los datos de cada una de las matrices. Esto se debe a que en el producto matricial, para cada valor de la matriz resultante, el recorrido realizado para el primer factor es realizado mediante filas mientras que el segundo factor se recorre por columnas. Por lo que las matrices A y C (al ser el primer factor de sus respectivas multiplicaciones) se ordenan por filas, y las matrices B y D se recorren por columnas. De esta forma aprovechamos el principio de localidad, es decir que cuando se accede a la memoria se traen los datos contiguos que efectivamente voy a utilizar para el siguiente elemento minimizando la cantidad de accesos a memoria y causando menos “fallos de páginas”.

- Se usan variables locales para ir almacenando valores, lo que permite ir acumulando los resultados de las multiplicaciones en una variable y finalmente cuando tengo el resultado final, volcarlo en la matriz resultante. Esto minimiza los accesos a la matriz, y ayuda a reducir el tiempo de ejecución.
- Para aprovechar mucho más la localidad de los datos y evitar reiterados accesos a memoria, se utiliza la técnica de **multiplicación de matrices por bloques** en donde se centra en dividir la matriz en submatrices, permitiendo trabajar con estas como si fueran independientes y maximizando de esta forma el uso de cada uno de los datos de la matriz mientras este se encuentre en caché.
- Para el cálculo de los máximos, mínimos y sumas se ha decidido realizarlo en diferentes loops, uno para los correspondientes a la matriz A y otro para los correspondientes a la matriz B. Esto es porque se puede acceder a los valores de cada matriz por separado y aprovechar mejor la caché debido a la localidad de datos.

Teniendo en cuenta lo explicado anteriormente, una cuestión que queda a tener en cuenta es que tamaño de bloque es el más óptimo. Para esto se ha decidido encontrar el mismo de una manera práctica, la cual consiste en ejecutar el algoritmo diseñado para la resolución del problema planteado variando el tamaño de la matriz (N) y del bloque (BS). Los resultados obtenidos se muestran en la siguiente sección.

Resultados obtenidos

BS/N	512	1024	2048	4096
4	0.713213	5.848824	46.627113	372.031391
8	0.585870	4.755708	37.947054	301.599300
16	0.508869	4.098599	32.663988	260.268187
32	0.458822	3.668576	29.124579	231.320126
64	0.436654	3.474978	27.613517	220.325417
128	0.429472	3.411914	27.059460	216.003174
256	0.443445	3.521324	27.922734	411.290836
512	0.430306	3.431716	43.493389	374.487860

Tiempos obtenidos en clúster remoto variando N y BS compilando con -O3 (N: Tamaño de las matrices, BS: Tamaño de las submatrices)

Al realizar las pruebas sobre matrices.c y variar el tamaño de la matriz N = 512, 1024, 2048, 4096 y además tomando tamaños de bloques BS = 4, 8, 16, 32, 64, 128, 256, 512 se puede

observar para el cluster que los menores tiempos de ejecución se dan cuando el tamaño de bloque es de 128, por lo que se puede concluir que este es el tamaño de bloque óptimo.