ESTUDIANTE: PAVEL ANGEL MENDOZA VILLAFANE

PROBLEMAS

A. Implement in C the simple three-nested-loop version of the matrix product and try to evaluate its performance for a relatively large matrix size.
   Solucion en 2.2.1
B. Implement the blocked version with six nested loops to check whether you can observe a significant gain.
   Solucion en 2.2.1
C. Execute these algorithms step by step to get a good understanding of data movements between the cache and the memory and try to evaluate their respective complexity in term of distant memory access.
   Solucion en 2.2.2
D. Execute these two versions of the code with valgring and kcachegrind to get a precise evaluation of their performance in term of cache misses.
   Solucion en 2.2.2

1. PRIMERA PARTE – ARTICULO 1

(Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?)

1.1 BREVE DESCRIPCION DEL ARTÍCULO

El artículo realiza una comparación entre diferentes algoritmos programados: primero con una rápida implementación y luego con una implementación óptima (agregando el concepto de paralelismo y ventajas del hardware y el compilador como vectorización manual).

1.1.1 Conceptos empleados

- **SIMD**: (Single Instructions Multiple Data). En este tipo de arquitectura paralela, todos los nodos comparten las mismas instrucciones pero trabajan sobre diferentes datos.
- **Ninja Gap**: El rendimiento entre un código simple C/C++ y un código optimizado en procesadores multi-cores.
- **Ninja Code**: Código optimizado para el uso de arquitecturas multi-core.
- **MIC**: Arquitectura Many Integrated Core.

1.1.2 Ideas destacadas

- El autor expone el enorme *ninja gap* en los procesadores modernos (hasta 53X) debido al poco esfuerzo computacional al escribir el código. Muestra también, como el uso de optimizaciones sobre el código puede reducir el gap hasta solo 1.3X.
- Entre los motivos por el mal rendimiento de los programas simples puede ser que los compiladores no pueden automáticamente identificar las regiones paralelas. También, el código no puede ser vectorizado de forma automática por el compilador (lo cual desperdicia las ventajas de las arquitecturas *SIMD*).
- Las pruebas son realizadas sobre procesadores *Intel MIC* (de 2, 4, y 6 núcleos) y hace uso de un conjunto de aplicaciones reales que pueden hacer uso extenso de paralelismo y otras optimizaciones.
- Se hace uso de *OpenMP* para lograr la paralelización. En el caso de la vectorización, se emplea el compilador *Intel Cilk Plus*, la cual permite hacer uso de *#pragma simd* para forzar la vectorización. Emplea además de los pragmas, flags en el compilador como *–no-vec* para desactivar la auto-vectorización y usarlo de forma manual. También encuentra ventaja sobre el uso de los conjuntos de instrucciones *AVX y SSE* (al compilar para el uso de los respectivos conjuntos de instrucciones).
- También emplea los *pragmas*: *unroll* (para desenvolver un bucle), *simd* (para forzar la vectorización), *omp* (para crear bloques paralelos)
- Dentro de los algoritmos se debe buscar la eliminación de operaciones tipo *broadcast* reduciendo la transmisión de datos y cambiar las estructuras de datos por estructuras de arreglos. Por ejemplo: un arreglo de punteros a vectores permite un acceso más rápido a los elementos.

1.1.3 Pruebas realizadas en el artículo

Los algoritmos probados fueron:

- NBody: Permite medir la interacción en un sistema de partículas con N cuerpos.

- BackProjection: Es usado para obtener una proyección de un espacio de dimensión (N-1) a otro de dimensión N.
- 7-Point Stencil: En 3D es usado para computar el grid centrado en un punto dado y determinar sus vecinos 4-connected en los planos X, Y y Z.
- Lattice Boltzmann Method: Es un método de simulación de dinámica de fluidos.
- LIBOR Monte Carlo: El método Monte Carlo es ampliamente usado en simulación para el cálculo de integrales por ejemplo.
- Complex 1D Convolution: Usado para el filtrado de señales 1D como audio.
- BlackScholes.
- TtreeSearch: Usado para crear índices para realizar consultas a una base de datos.
- MergeSort: Es un método de ordenamiento.
- 2D 5X5 Convolution: La convolución 2D es usado para filtrar señales 2D como imágenes por ejemplo.
- Volume Rendering.

1.2 PRUEBAS REALIZADAS BASADAS EN LOS METODOS PROPUESTOS

Debido a que el artículo realiza comparaciones exclusivamente sobre los tiempos de ejecución para medir la aceleración entre una implementación básica y una optimizada, se considerara el mismo criterio en las implementaciones realizadas.

Se implementó 2 métodos descritos en el artículo:

- MergeSort.
- 2D 5x5 Convolution.

En cada caso se realizó una implementación serial y otra paralela para realizar la comparación.

1.2.1 Observaciones de las optimizaciones

- El compilador empleado fue del VS-C++ 2012, el cual tiene por defecto la auto-vectorización habilitada. Para desactivarla se usó: *#pragma loop (no_vec)* antes de cada bucle for.
- La vectorización forzada no está permitida en el VS-C++. *#pragma simd* es solo valido para el compilador CILK Plus de Intel (la cual requiere una licencia de uso).
- Un requisito para la vectorización es que el bucle sea de una sola sentencia.
- El uso del conjunto de instrucciones *AVX y SSE* depende de la arquitectura de trabajo. Las pruebas fueron realizadas sobre un Intel Core i5 M430 cuyo conjunto de instrucciones es SSE. Por lo cual no tiene sentido realizar la compilación simulando AVX.
- El uso del pragma *unroll* solo tiene sentido si el bucle es corto, por lo que solo se aplicó en el caso de la convolución dentro del producto entre la ventana y la sub-imagen.
- Las pruebas fueron realizadas construyendo el reléase en 64bits.

1.2.2 Implementación

El código puede ser revisado en:
https://github.com/pmendozav/ucsp/blob/master/paralelo/paralelo_tareas_cpu/tarea1_ninja_gap/Source.cpp

1.2.3 Resultados obtenidos

Las pruebas fueron realizadas sobre una Core i5 M430 (Core i5 de 1ra generación).

1.2.3.1 Convolución 2D

- La tabla 01 muestra una aceleración de hasta 97.5% haciendo uso de la implementación paralela. Similar resultado se observa en la tabla 02 donde se intentó detener la auto-vectorización por lo que se supone que no funciona de forma adecuada.
- La aceleración de tan solo el doble con 4 hilos puede explicarse con la tabla 03, la cual muestra el tiempo de ejecución empleando de 1 a 4 hilos. Como puede observarse, la aceleración con 3 y 4 hilos es muy similar y no es muy destacable respecto al de 2 hilos. Esto es debido a que el procesador empleado solo posee 2 núcleos físicos y 4 virtuales por lo que era de esperarse la caída en el rendimiento.

N	Tiempo Serial	Tiempo Paralelo	Aceleración (%)
15000	15.15	7.67	97.5
10000	6.83	3.46	97.4
5000	1.71	0.94	81.9

Tabla 1. Prueba con 4 hilos y deteniendo la vectorización por bucles.

N	Tiempo Serial	Tiempo Paralelo	Aceleración (%)
15000	15.07	7.39	100.3
10000	6.66	3.36	98.2
5000	1.71	0.88	94.3

Tabla 2. Prueba con 4 hilos y empleando la vectorización automática de VS-C++ 2012.

Hilos	Tiempo	Mejora (Tn/T1)*100%
1	16.02	1
2	8.06	98.7
3	7.76	106.6
4	7.57	111.6

Tabla 3. Prueba paralela con N=15000 y variando el número de hilos. Considerando la vectorización automática del VS-C++ 2012.

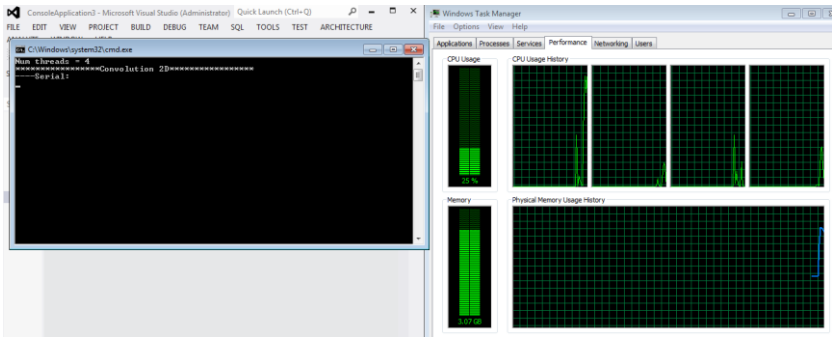

Figura 1. Uso del CPU y RAM de la convolución 2D serial.

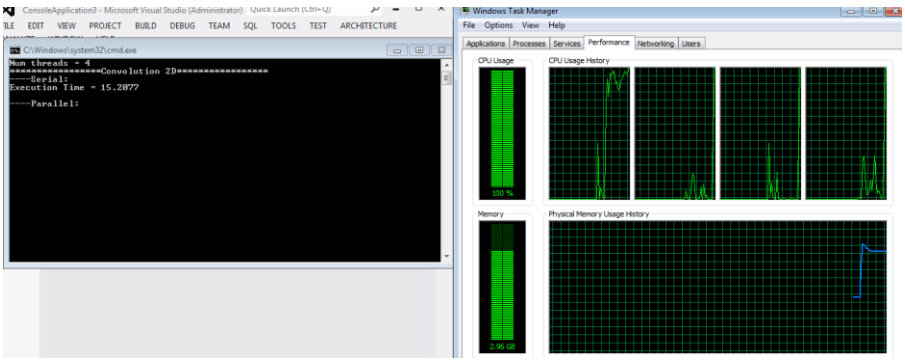

Figura 2. Uso del CPU y RAM de la convolución 2D paralelo.

1.2.3.2 MergeSort

- La tabla 04 muestra una aceleración del 41.7% haciendo uso de la implementación paralela. La tabla 05 muestra resultados similares al caso de la convolución. La tabla 06 muestra el comportamiento del programa basado en el número de hilos (el rendimiento decrece como ya se mencionó debido a que las pruebas se realizaron solo con 2 núcleos físicos).
- La aceleración de tan solo el 41.7% muestra un posible problema en la implementación ya que se esperaba, como en el caso de la convolución, una aceleración de por lo menos el doble. Esto puede deberse ya que, la división del arreglo requiere una copia del mismo en 2 (ya que se usa el tipo std::*vector*) o al uso del pragma *sections*, el cual parece retener el uso de procesador a tan solo en 50% lo cual se observó durante las pruebas al revisar el Administrador de Tareas de Windows.

N	Tiempo Serial	Tiempo Paralelo	Aceleración
8388608	10.08	7.67	31.4
4194304	4.86	3.43	41.7
2097152	2.31	1.65	40.0

Tabla 4. Prueba con 4 hilos y deteniendo la vectorización por bucles.

N	Tiempo Serial	Tiempo Paralelo	Aceleración
8388608	10.15	7.31	38.8
4194304	4.89	3.32	47.3
2097152	2.26	1.70	32.9

Tabla 5. Prueba con 4 hilos y empleando la vectorización automática de VS-C++ 2012.

Hilos	Tiempo	Mejora (T1 - T1/Tn)*100%
1	11.05	1
2	7.31	51.1
3	7.10	55.6
4	7.05	56.7

Tabla 6. Prueba paralela con N=8388608 y variando el número de hilos.

2. SEGUNDA PARTE – ARTICULO 2

(What Every Programmer Should Know About Memory)

2.1 BREVE DESCRIPCION DEL ARTÍCULO

El artículo hace una revisión sobre los niveles de cache y memoria en una arquitectura dada y como un buen uso de estos recursos puede influir positivamente en el rendimiento de la aplicación.

Muestra también algunas herramientas para realizar el *profiling* de una aplicación con el fin de encontrar los cuellos de botella y detectar un mal uso de los recursos del sistema.

2.2 PRUEBAS REALIZADAS BASADAS EN LOS METODOS PROPUESTOS

2.2.1 Programas Implementados

Se realizó varias implementaciones del productos de matrices C = AB.

El código puede ser revisado en:

https://github.com/pmendozav/ucsp/blob/master/paralelo/pruebas_linux/tarea%201%20-%20matrices/functions.h

i. prob1_1_v2: Implementación directa del producto de matrices cuadradas.
ii. prob1_2_v2: Modificación de la multiplicación usando la transpuesta de B.
iii. prob2_1_v2: Modificación propuesta por el autor empleando la multiplicación por bloques con 6 bucles anidados.
iv. prob2_2_v2: Modificación de prob2_1_v2 representando a las matrices como arreglos unidimensionales.

2.2.2 Pruebas de Cache

Las pruebas se realizaron empleando cachegrind ejecutando los algoritmos con matrices de dimensión 1000x1000 sobre un SO Ubuntu virtualizado sobre Windows.

La figura 3 muestra los resultados de la simulación de cache en consola de ejecutar el comando: valgrind --tool=cachegrind ./matrices. Una mejor representación de los resultados puede ser visto en la figura 4 (a, b, c, d). Como puede observarse en la figura 5, el orden de mejora en rendimiento creciente de las implementaciones es (ii), (i), (iii) y (iv). A pesar que (iii) y (iv) representan el mismo algoritmo, la implementación de las matrices en 1D y 2D perjudican el rendimiento considerablemente.

La figura 6 muestra otra prueba de la simulación de cache ejecutando valgrind --tool=cachegrind --L2=8388608,8,64 ./matrices. En la figura 7 puede observarse el rendimiento en el orden (ii), (i), (iii) y (iv) como en la prueba anterior.

Cabe resaltar que durante la ejecución se observó el tiempo de ejecución y reflejaban los resultados mostrados en las figuras 5, 6, 7 y 8. Sin embargo, estas fueron repetidas en VS-C++ 2012 sobre Windows y fue medido el tiempo de ejecución de las mismas funciones con un orden de rendimiento (i), (iii), (ii) y (iv) siendo la implementación (ii) y (iii) muy cercano durante las pruebas. El motivo del cambio sobre

Ubuntu pudo ser provocado a que este fue virtualizado sobre Windows. Otro motivo puede ser el uso del tipo std::vector. El motivo del uso de std::vector es su capacidad para soportar arreglos de gran tamaño.

Las figuras 8 y 9 muestra los resultados por tiempo y el profiling de VS-C++2012 para las implementaciones.

```
--------
==2748==
==2748== I   refs:      290,750,658,074
==2748== I1  misses:              2,879
==2748== LLi misses:              2,807
==2748== I1  miss rate:            0.00%
==2748== LLi miss rate:            0.00%
==2748==
==2748== D   refs:      154,853,223,670  (105,826,189,707 rd   + 49,027,033,963 wr)
==2748== D1  misses:      1,575,386,548  (  1,573,753,492 rd   +      1,633,056 wr)
==2748== LLd misses:        144,039,536  (    143,347,389 rd   +        692,147 wr)
==2748== D1  miss rate:             1.0% (            1.5%     +            0.0%  )
==2748== LLd miss rate:             0.1% (            0.1%     +            0.0%  )
==2748==
==2748== LL refs:         1,575,389,427  (  1,573,756,371 rd   +      1,633,056 wr)
==2748== LL misses:         144,042,343  (    143,350,196 rd   +        692,147 wr)
==2748== LL miss rate:              0.0% (            0.0%     +            0.0%  )
```

Figura 3. Resultados de consola de la prueba de cachegrind

Event Type	Incl.
Instruction Fetch	33 022 020 052
L1 Instr. Fetch Miss	9
LL Instr. Fetch Miss	9
Data Read Access	12 006 006 012
L1 Data Read Miss	1 062 997 042
LL Data Read Miss	63 561 002
Data Write Access	6 005 003 018
L1 Data Write Miss	1 000 000
LL Data Write Miss	62 751
L1 Miss Sum	1 063 997 051
Last-level Miss Sum	63 623 762
Cycle Estimation	50 024 366 762

(a)

Event Type	Incl.
Instruction Fetch	45 036 036 067
L1 Instr. Fetch Miss	13
LL Instr. Fetch Miss	13
Data Read Access	14 011 010 014
L1 Data Read Miss	64 393 215
LL Data Read Miss	63 389 432
Data Write Access	9 006 006 023
L1 Data Write Miss	63 267
LL Data Write Miss	63 266
L1 Miss Sum	64 456 495
Last-level Miss Sum	63 452 711
Cycle Estimation	52 025 872 117

(b)

Event Type	Incl.
Instruction Fetch	42 576 421 179
L1 Instr. Fetch Miss	17
LL Instr. Fetch Miss	17
Data Read Access	14 234 425 262
L1 Data Read Miss	31 375 302
LL Data Read Miss	8 061 700
Data Write Access	1 689 486 644
L1 Data Write Miss	0
LL Data Write Miss	0
L1 Miss Sum	31 375 319
Last-level Miss Sum	8 061 717
Cycle Estimation	43 696 346 069

(c)

Event Type	Incl.
Instruction Fetch	40 347 892 551
L1 Instr. Fetch Miss	13
LL Instr. Fetch Miss	13
Data Read Access	13 640 672 261
L1 Data Read Miss	31 329 754
LL Data Read Miss	7 939 503
Data Write Access	1 183 625 268
L1 Data Write Miss	1
LL Data Write Miss	1
L1 Miss Sum	31 329 768
Last-level Miss Sum	7 939 517
Cycle Estimation	41 455 141 931

(d)

Figura 4. Resultados de ejecutar: valgrind --tool=cachegrind ./matrices

Self	Function
18.37	prob1_2_v2(std::vector…
12.25	prob1_1_v2(std::vector…
3.45	prob2_1_v2(std::vector…
2.41	prob2_2_v2(std::vector…

Figura 5. Rendimiento de cada implementación con parámetros por defecto del cachegrind

Event Type	Incl.
Instruction Fetch	33 022 020 052
L1 Instr. Fetch Miss	9
LL Instr. Fetch Miss	9
Data Read Access	12 006 006 012
L1 Data Read Miss	1 062 997 042
LL Data Read Miss	104 982
Data Write Access	6 005 003 018
L1 Data Write Miss	1 000 000
LL Data Write Miss	50 373
L1 Miss Sum	1 063 997 051
Last-level Miss Sum	155 364
Cycle Estimation	43 677 526 962

(a)

Event Type	Incl.
Instruction Fetch	45 036 036 067
L1 Instr. Fetch Miss	13
LL Instr. Fetch Miss	13
Data Read Access	14 011 010 014
L1 Data Read Miss	64 393 215
LL Data Read Miss	192 568
Data Write Access	9 006 006 023
L1 Data Write Miss	63 267
LL Data Write Miss	15 563
L1 Miss Sum	64 456 495
Last-level Miss Sum	208 144
Cycle Estimation	45 701 415 417

(b)

Event Type	Incl.
Instruction Fetch	42 576 421 179
L1 Instr. Fetch Miss	17
LL Instr. Fetch Miss	17
Data Read Access	14 234 425 262
L1 Data Read Miss	31 375 302
LL Data Read Miss	176 002
Data Write Access	1 689 486 644
L1 Data Write Miss	0
LL Data Write Miss	0
L1 Miss Sum	31 375 319
Last-level Miss Sum	176 019
Cycle Estimation	42 907 776 269

(c)

Event Type	Incl.
Instruction Fetch	40 347 892 551
L1 Instr. Fetch Miss	13
LL Instr. Fetch Miss	13
Data Read Access	13 640 672 261
L1 Data Read Miss	31 329 754
LL Data Read Miss	175 422
Data Write Access	1 183 625 268
L1 Data Write Miss	1
LL Data Write Miss	0
L1 Miss Sum	31 329 768
Last-level Miss Sum	175 435
Cycle Estimation	40 678 733 731

(d)

Figura 6. Resultados de ejecutar: valgrind --tool=cachegrind --L2=8388608,8,64 ./matrices

Self	Function
18.37	prob1_2_v2(std::vector…
12.25	prob1_1_v2(std::vector…
3.45	prob2_1_v2(std::vector…
2.41	prob2_2_v2(std::vector…

Figura 7. Rendimiento de cada implementación con parámetros por defecto del cachegrind

```
prob1_1_v2 time: time: 13.036
-------
prob1_2_v2 time: time: 1.513
-------
prob2_1_v2 time: time: 1.803
-------
prob2_2_v2 time: time: 1.48
-------
```

Figura 8. Resultados de tiempo de la implementaciones sobre Windows con matrices de dimensión 1000x1000.

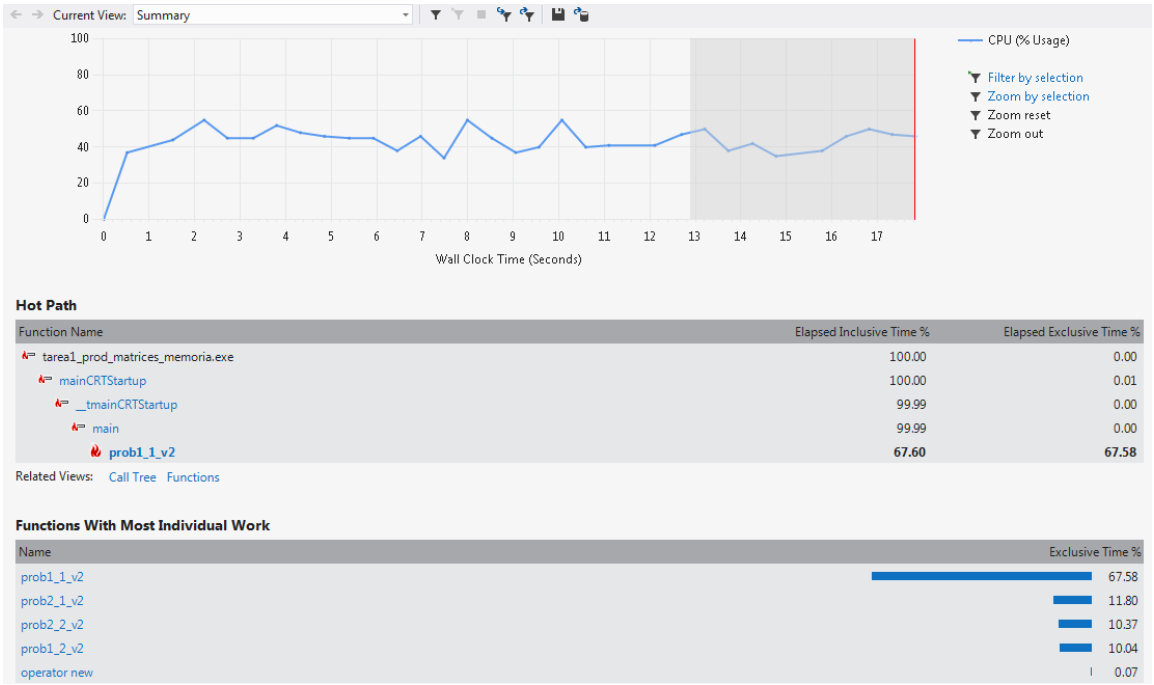

Figura 9. Profiling de VS-C++ 2012, donde puede observarse el uso del CPU por cada función.

CONCLUSIONES

- Se realizó la implementación de algunos métodos de 2 artículos para entender la influencia del uso de la cache sobre los programas y el uso de optimizaciones a nivel del compilador (flags y pragmas), arquitectura empleada y programación (paralelismo) para obtener un código que emplee mejor los recursos del sistema.