Algoritmos Paralelos – Multiplicacion de matrices en cache

Alumno: Carpio Rodriguez David

Hw y Sw :

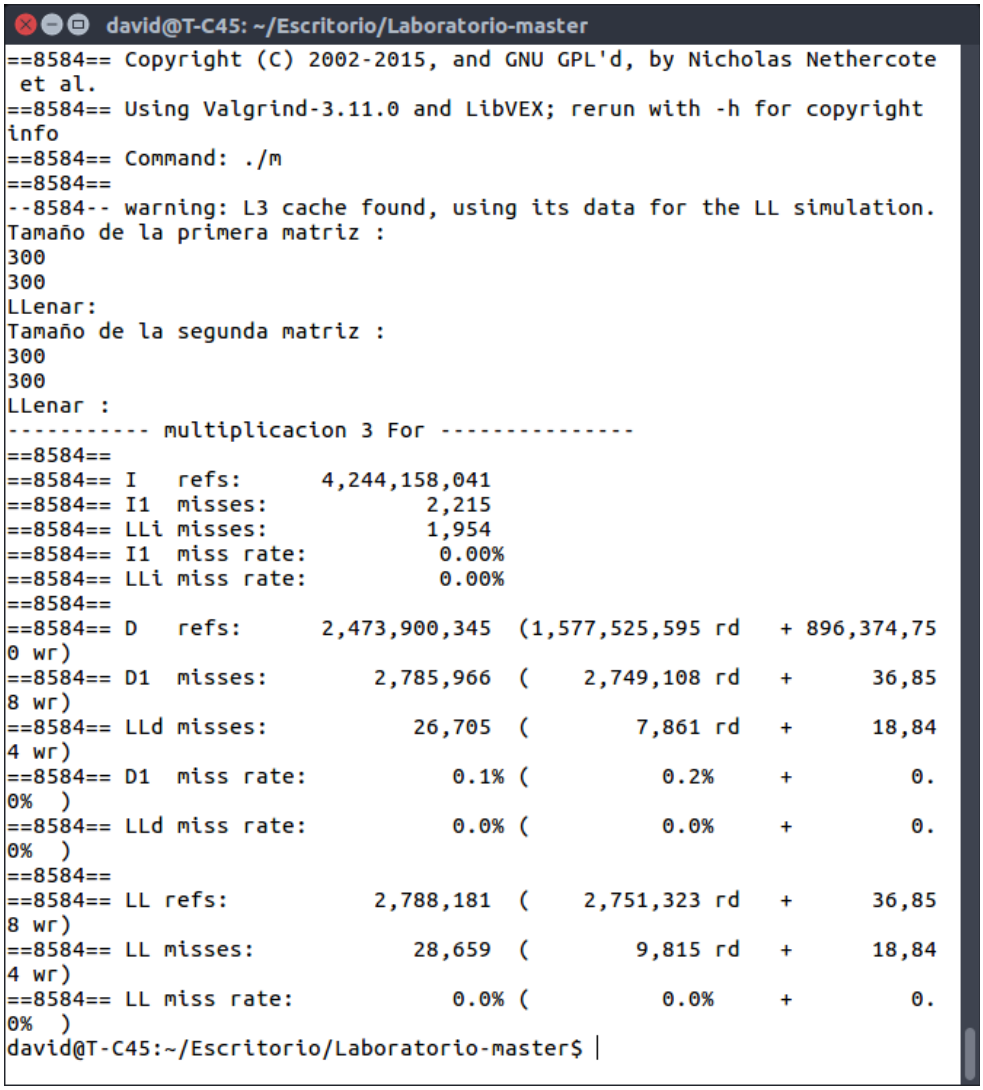
Para las pruebas de multiplicacion de matrices; se tuvo en cuenta un procesador Intel® Core™ i5-3230M CPU @ 2.60GHz × 4 , Memoria Ram de 8Gb DDR3.
Tambien el sistema operativo Ubuntu 16.04 x64

- El codgo utilizado se encuentra en este repositorio:
https://github.com/asiesps/Paralelos

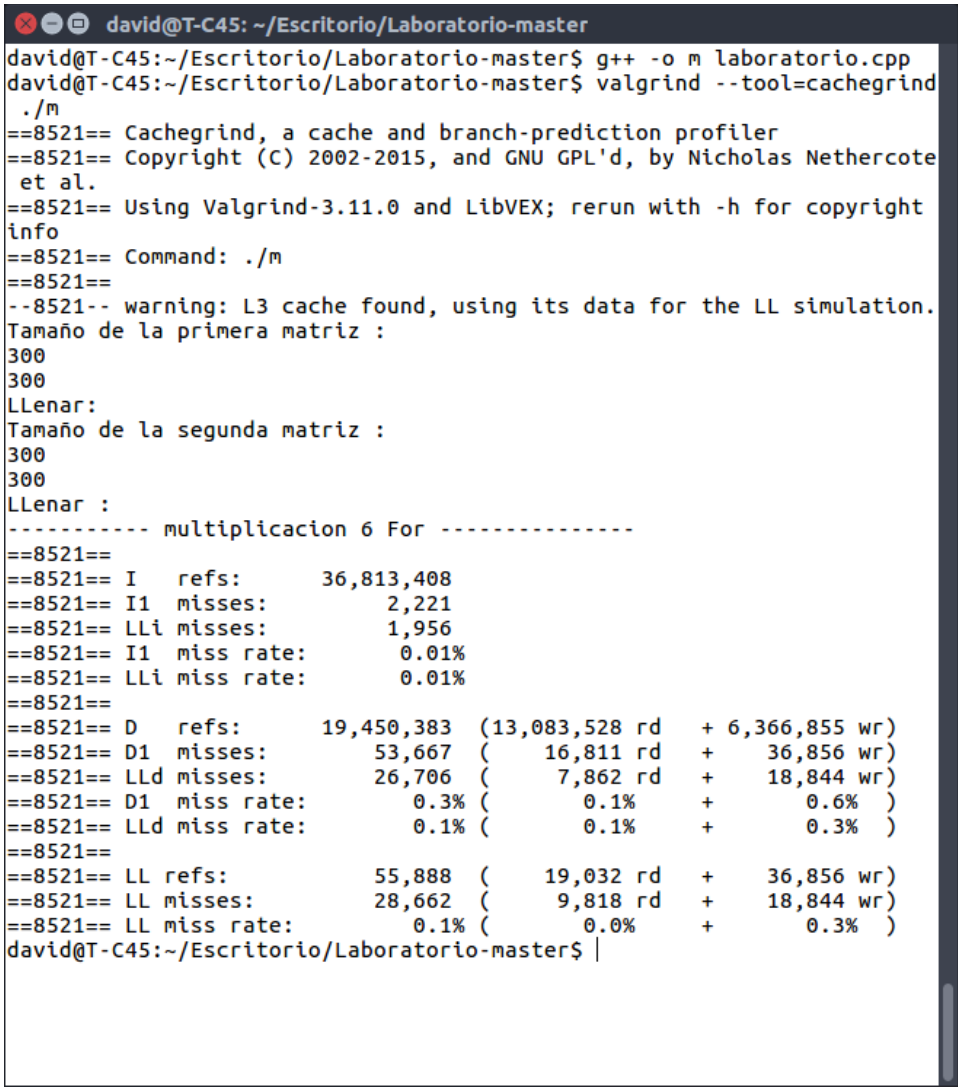
Prueba 1:

Multiplicación con matrices cuadradas de 300 X 300

- Se ejecuto la función : “normal_multi” que tiene 3 bucles anidados para la multiplicación de matrices:



- Se ejecuto la función : “bloq_multi” que tiene 6 bucles anidados para la multiplicación de matrices:



Explicación:

* “miss” : Son las lecturas hechas donde la información no está disponible, una lectura que no encontró la instrucción o dato necesario para continuar la instrucción.

I : Total de accesos a instrucciones realizadas.
I1 : Caché de instrucciones de primer nivel
Lli : Caché de instrucciones de ultimo nivel
D : Total de accesos a datos leidos.
D1 : Caché de datos de primer nivel
LLd : Datos de caché de último nivel

y finalmente un resumen de los accesos de caché de último nivel

Prueba 2:

Multiplicación con matrices cuadradas de 1000 X 1000

- Se ejecuto la función : “normal_multi” que tiene 3 bucles anidados para la multiplicación de matrices:

```
david@T-C45: ~/Escritorio/Laboratorio-master
david@T-C45:~/Escritorio/Laboratorio-master$
david@T-C45:~/Escritorio/Laboratorio-master$ g++ -o m laboratorio.cpp
david@T-C45:~/Escritorio/Laboratorio-master$ valgrind --tool=cachegrind ./m
==3552== Cachegrind, a cache and branch-prediction profiler
==3552== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==3552== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3552== Command: ./m
==3552==
--3552-- warning: L3 cache found, using its data for the LL simulation.
Tamaño de la primera matriz :
1000
1000
LLenar:
Tamaño de la segunda matriz :
1000
1000
LLenar :
---------- multiplicacion 3 For ---------------
==3552== brk segment overflow in thread #1: can't grow to 0x4a3c000
==3552== brk segment overflow in thread #1: can't grow to 0x4a3c000
==3552== brk segment overflow in thread #1: can't grow to 0x4a3c000
==3552== brk segment overflow in thread #1: can't grow to 0x4a3c000

==3552==
==3552== I   refs:      156,318,309,139
==3552== I1  misses:              2,237
==3552== LLi misses:              2,151
==3552== I1  miss rate:            0.00%
==3552== LLi miss rate:            0.00%
==3552==
==3552== D   refs:       91,170,124,448  (58,116,960,056 rd   + 33,053,164,392 wr)
==3552== D1  misses:      1,438,714,612  ( 1,438,334,538 rd   +        380,074 wr)
==3552== LLd misses:         63,270,142  (    62,891,022 rd   +        379,120 wr)
==3552== D1  miss rate:             1.6% (           2.5%     +            0.0%  )
==3552== LLd miss rate:             0.1% (           0.1%     +            0.0%  )
==3552==
==3552== LL refs:         1,438,716,849  ( 1,438,336,775 rd   +        380,074 wr)
==3552== LL misses:          63,272,293  (    62,893,173 rd   +        379,120 wr)
==3552== LL miss rate:              0.0% (           0.0%     +            0.0%  )
david@T-C45:~/Escritorio/Laboratorio-master$
david@T-C45:~/Escritorio/Laboratorio-master$
```

Tomo aproximadamente unos 20 minutos la culminacion de la multiplicacion.

- Se ejecuto la función : “bloq_multi” que tiene 6 bucles anidados para la multiplicación de matrices:

```
david@T-C45: ~/Escritorio/Laboratorio-master
david@T-C45:~/Escritorio/Laboratorio-master$ g++ -o m laboratorio.cpp
david@T-C45:~/Escritorio/Laboratorio-master$ valgrind --tool=cachegrind
./m
==2387== Cachegrind, a cache and branch-prediction profiler
==2387== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote
et al.
==2387== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright i
nfo
==2387== Command: ./m
==2387==
--2387-- warning: L3 cache found, using its data for the LL simulation.
Tamaño de la primera matriz :
1000
1000
LLenar:
Tamaño de la segunda matriz :
1000
1000
LLenar :
---------- multiplicacion 6 For ---------------
==2387== brk segment overflow in thread #1: can't grow to 0x4a3c000
==2387== brk segment overflow in thread #1: can't grow to 0x4a3c000
==2387== brk segment overflow in thread #1: can't grow to 0x4a3c000
==2387== brk segment overflow in thread #1: can't grow to 0x4a3c000
==2387==
==2387== I   refs:      314,768,206
==2387== I1  misses:          2,245
==2387== LLi misses:          2,159
==2387== I1  miss rate:        0.00%
==2387== LLi miss rate:        0.00%
==2387==
==2387== D   refs:      167,210,286  (113,964,489 rd   + 53,245,797 wr)
==2387== D1  misses:        401,887  (     21,815 rd   +    380,072 wr)
==2387== LLd misses:        392,286  (     13,168 rd   +    379,118 wr)
==2387== D1  miss rate:         0.2% (        0.0%     +        0.7%  )
==2387== LLd miss rate:         0.2% (        0.0%     +        0.7%  )
==2387==
==2387== LL refs:           404,132  (     24,060 rd   +    380,072 wr)
==2387== LL misses:         394,445  (     15,327 rd   +    379,118 wr)
==2387== LL miss rate:          0.1% (        0.0%     +        0.7%  )
david@T-C45:~/Escritorio/Laboratorio-master$
```

Conclusión:

La descomposición en bloques reduce la tasa de fallos aumentando la localidad temporal en lugar de operar sobre filas o columnas completas de un array los algoritmos de descomposición en bloques operan sobre submatrices o bloques tomando el objetivo de maximizar los accesos a los datos cargados en la caché antes de ser reemplazados.

La multiplicacion de matrices relativamente pequeñas es de facil realizacion para el procesador, pero cuando hablamos de matrices de gran tamaño, entonces podemos tomar la estrategia de dividir estas en bloques para reutilizar los datos con mayor facilidad y no obtener fallas en la cache que haran que se tomen sucesivamente los mismos datos.

Como podemos apreciar en las pruebas realizadas(1) tenemos mas de 2 millones de accesos contra los algo mas de 50 mil accesos a datos ; de igual manera en la prueba(2) obtenemos mas de mil millones de accesos contra los 400 mil accesos que obtenemos al utilizar mejor la cache de nuestro procesador.