Performance evaluation of a single core Performance evaluation of a multi-core implementation

(1.º Trabalho Laboratorial)

Computação Paralela e Distribuída

LEIC - 02/2024

Turma 13 - Grupo 6

Bruno Leal - up2020080047@edu.fe.up.pt

João Fernandes - up202108044@edu.fe.up.pt

Rui Silveira - up202108878@edu.fe.up.pt

22 de fevereiro de 2024

Índice

1. Introdução/Descrição do Problema

Este relatório contém todo o estudo da variação da performance de processador single-core e multi-core na memória hierárquica quando abordando grandes quantidades de dados. Para esse efeito, estudamos várias abordagens de multiplicações de 2 matrizes quadradas, não tendo em vista a matriz resultante, mas toda a performance do processador(es) nestes processos.

2. Explicação dos algoritmos

Tendo em conta que estamos a usar a multiplicação de matrizes para medir e analisar a manipulação de memória, vamos ter em consideração três algoritmos de forma a, sobretudo, verificar como elas tiram vantagem da alocação de memória, sendo eles:

- **Simple Matrix Multiplication**
- **Line Matrix Multiplication**
- **Block Matrix Multiplication**

Importante ter em conta que os dois primeiros algoritmos foram implementados tanto em C/C++ (o **simple matrix multiplication** já foi fornecido em um ficheiro base), como em Java, linguagem escolhida devido à similaridade em termos de sintaxe, já que esta última deriva de C++, para além de ser também uma linguagem compilada. Já o último, só foi implementado em C/C++. A escolha de C/C++ também se deve à PAPI (Performance API) que permite obter determinados valores importantes para métricas de desempenho e a OpenMP, uma API que facilita uma computação paralela entre núcleos do processador, entre outros.

Simple Matrix Multiplication

Este algoritmo multiplica duas matrizes em uma abordagem “tradicional” (Ex: multiplicando uma linha da primeira matriz pela respetiva coluna da segunda matriz). De complexidade $O(n^3)$. O código em C/C++ encontra-se abaixo:

```
C/C++
      for(i=0; i<m_ar; i++)
          {   for( j=0; j<m_br; j++)
              {   temp = 0;
                  for( k=0; k<m_ar; k++)
                  {
                      temp += pha[i*m_ar+k] * phb[k*m_br+j];
                  }
                  phc[i*m_ar+j]=temp;
              }
          }
```

Line Matrix Multiplication

Este algoritmo multiplica duas matrizes, multiplicando um elemento da primeira matriz pela linha correspondente da segunda matriz. De complexidade $O(n^3)$. O código em C/C++ encontra-se abaixo:

```
C/C++
      for (i = 0; i < m_ar; i++) {
              for (j = 0; j < m_br; j++) {
                  for (k = 0; k < m_ar; k++) {
                      phc[i*m_ar+k] += pha[i*m_ar+k]*phb[j*m_ar+k];
                  }
              }
          }
```

Block Matrix Multiplication

Este último algoritmo, é uma versão superior da **Line Matrix Multiplication**, em que pretende multiplicar esta matriz, dividindo-a em matrizes menores (blocos) que são calculados separadamente e juntos no final. Novamente, com complexidade $O(n^3)$. O código em C/C++ encontra-se abaixo:

```
C/C++
      for (i = 0; i < m_ar; i += bkSize) {
              for (j = 0; j < m_br; j += bkSize) {
                  for (k = 0; k < m_ar; k += bkSize) {
                      for (i_blk = i; i_blk < i + bkSize; i_blk++) {
                          for (j_blk = j; j_blk < j+ bkSize; j_blk++) {
                              for (k_blk = k; k_blk < k+bkSize; k_blk++) {
                                  phc[i_blk*m_ar+k_blk] += pha[i_blk * m_ar +
j_blk] * phb[j_blk * m_br + k_blk];
                              }
                          }
                      }
                  }
              }
          }
```

Line Matrix Multiplication - Análise Multi-Core

Estes dois próximos algoritmos são versões paralelas da segunda implementação. Ambas possuem a mesma complexidade, variando no paralelismo entre threads no

processador. Iremos identificar ambas implementações para facilitar a distinção. A primeira versão distingue-se da segunda no sentido em que esta pretende paralelizar o loop de “i” ao longo das threads disponíveis, sendo a “j” e “k” sequenciais por cada thread, ao invés da segunda versão que pretende paralelizar o loop de “k”, sendo “i” e “j” sendo executados de maneira sequencial.

Versão 1

```C/C++
    #pragma omp parallel for
        for (i = 0; i < m_ar; i++) {
            for (j = 0; j < m_br; j++) {
                for (k = 0; k < m_ar; k++) {
                    phc[i*m_ar+k] += pha[i*m_ar+k]*phb[j*m_ar+k];
                }
            }
        }
```

Versão 2

```C/C++
    #pragma omp parallel private (i,j)
        for (i = 0; i < m_ar; i++) {
            for (j = 0; j < m_br; j++) {
                #pragma omp for
                for (k = 0; k < m_ar; k++) {
                    phc[i*m_ar+k] += pha[i*m_ar+k]*phb[j*m_ar+k];
                }
            }
        }
```

3. Métricas de desempenho

Tendo em vista a medição de desempenho, serão usadas várias métricas, entre as quais:

Obtidos por Medição Direta

- **L1 e L2 Data Cache Misses:** A quantidade de vezes que durante o correr do programa, ocorreram tentativas falhadas de aceder a posição de memória de determinado valor. Importante ter em conta que, por L1 ser menor que L2, os data cache misses tendem a ser menores, portanto, sendo a informação em L1 acessada de maneira mais rápida.Estes valores são possíveis de obter de uma maneira direta e não derivada, graças à PAPI.

- **Tempo em segundos:** Tempo necessário para calcular toda a matriz.
- **Valores da máquina de testes:** Estes testes foram realizados em uma máquina com um i7-9700K Octa-core com 4.7GHz, com 16KB de L1 cache e 256KB de L2 cache. Importante também realçar o uso da flag -O2, recomendada pela Unidade Curricular para uma melhor performance dos algoritmos.

Obtidos por Medição Indireta

- **FLOPS (FLoating point Operations Per Second):** Este valor é possível de obter através da seguinte fórmula:

$$FLOPS = 2 * \frac{matrix\ size^{3}}{CPU\ Time}$$

- **SpeedUp:** Este valor será relevante para o multiCore, em que se traduz para o ganho obtido pelo programa em paralelo. Este valor será calculado através da fórmula (Nota: S -> A percentagem do programa que pode ser realizado de forma paralela) :

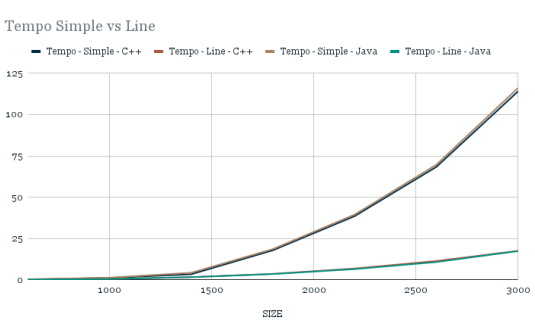
$$SpeedUp = \frac{1}{\frac{1-S}{NºCores} + S} \text{ ou } \frac{TSequencial}{TParalelo}$$

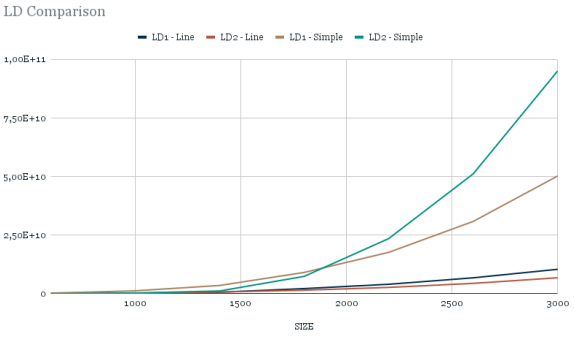
- **Eficiência:** Este valor será relevante para o multiCore, que se traduz para o ganho por Core do paralelismo. Este valor será calculado através da fórmula:

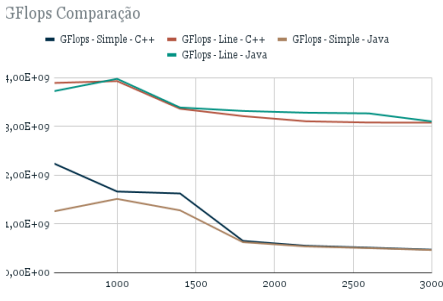
$$Eficiência = \frac{SpeedUp}{Nº\ Cores}$$

4. Resultados e análise

4.1 Simple vs Line Matrix Multiplication



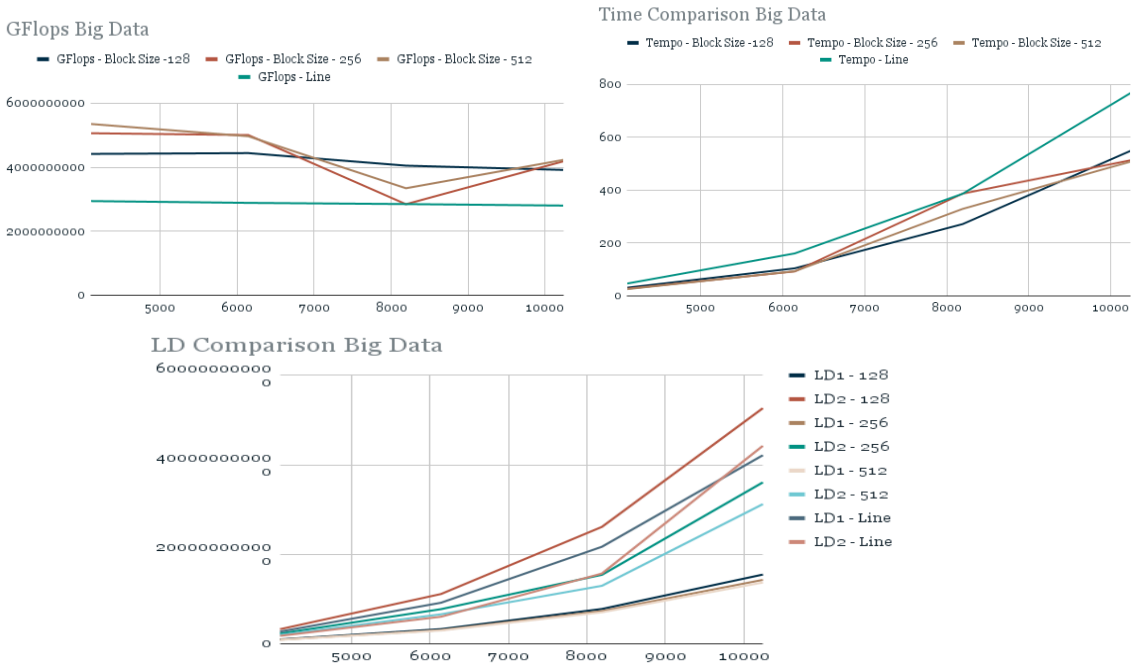




Aqui temos os três gráficos comparativos de tempo, LD1 e LD2 e GFlops entre os algoritmos Simple e Line Matrix Multiplication, e, dados estes gráficos, é possível constatar alguns factos:

- O Simple demora sempre mais do que o Line, quer seja ou não significativamente, e isto deve-se ao número reduzido de Data Cache Misses quer nível 1, quer nível 2, devido ao acumulador que este segundo apresenta. O gráfico do Simple Matrix revela ser exponencial, isto é, conforme o tamanho da matrix, o tempo evolui de maneira exponencial.
- A utilização da linguagem Java como segunda linguagem revelou-se interessante, no sentido de demonstrar que, apesar de se observar tempos inferiores, não era significativo, o que demonstra que ambas as linguagens tem uma performance semelhante.
- Como se esperava, os GFlops são inversamente proporcionais ao tempo, obtendo um número maior de GFlops no Line Matrix Multiplication, algoritmo mais eficiente do que o Simple.

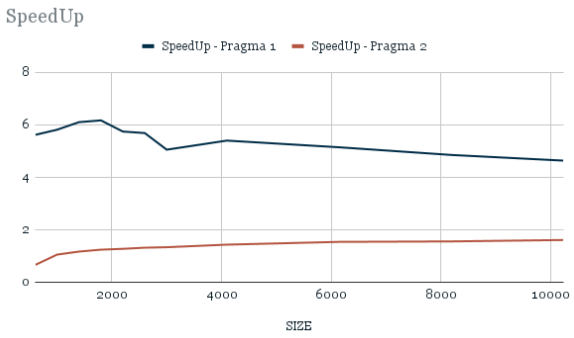
4.2 Line vs Block Matrix Multiplication

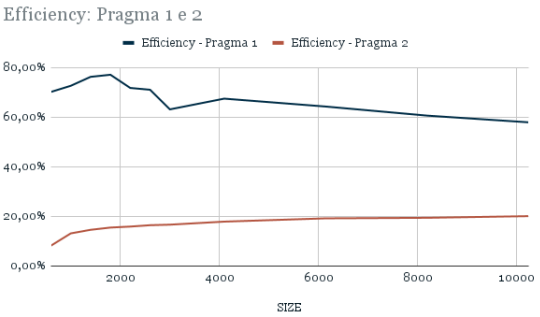


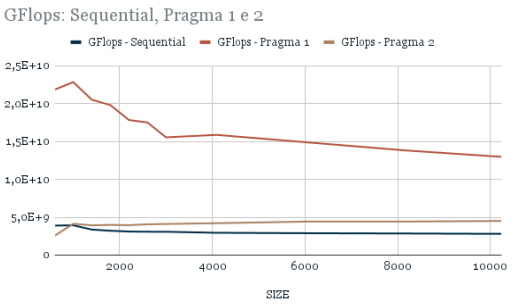
Aqui temos as comparações entre o algoritmo Line e o Block, em termos de GFlops, tempo e L1 e L2 cache misses. Existem algumas constatações que podem ser retiradas destes gráficos:

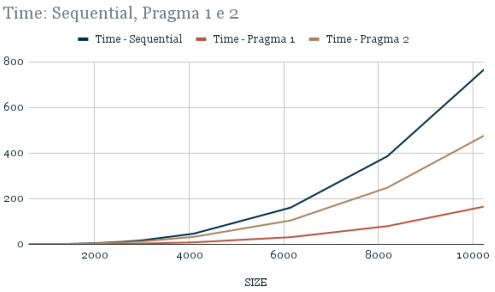
- Em geral, a versão que divide a matriz em blocos é mais eficiente que a versão Line, e conforme o aumento do BlockSize, há uma diminuição do tempo necessário para computar a multiplicação. Exceto no valor de 8192 que o mesmo não se verificou. Isto traduz-se também nos GFlops.
- Em termos de Data Cache misses, a L1 em Line é geralmente superior, porém, em termos de L2, varia: quando o BlockSize é 128 o L2 é superior, porém a nível de 256 e 518, é inferior. Isto pode ser consequência do blockSize ser relativamente pequeno, e necessitar de mais pesquisas em L2, por esta ser uma cache maior do que L1, para acessar determinados lugares da memória.

4.3 Sequential vs Parallel - Line Multiplication Matrix









Neste tópico abordaremos a diferença entre sequencial e paralelo, bem como as diferenças entre os dois pragmas, dos quais será possível retirar algumas constatações:

- Ambos os pragmas são, como esperado, mais rápidos que o algoritmo base, isto é, o paralelismo torna os algoritmos mais eficientes. Porém, é notório que a versão 1, que efetua a paralelização já na primeira iteração, é mais eficiente que a versão 2, que só efetua na última iteração. Isto é notório nos gráficos de SpeedUp (relação entre tempo sequencial e o tempo da versão com paralelismo) e dos GFlops, em que mostra que estes algoritmos permitem um número maior de Float Operations por segundo.
- A eficiência do primeiro algoritmo é bem superior ao segundo, o que revela que: uma maior paralelização, consegue tornar melhor partido do multi-threading, isto é, consegue usar as cores do processador de uma maneira mais eficiente.

5. Conclusão

Com este trabalho foi-nos possível aprofundar os nossos conhecimentos sobre o quão importante é o gerenciamento da memória para obter mais eficiência, quer em programas sequenciais e paralelos, e também sobre a importância da paralelização que pode tirar muito partido sobre a sequencialização, se bem utilizada.Podemos concluir também que o tempo neste tipo de algoritmos cresce de maneira exponencial.

Conseguimos notar também a importância de Level 1 e Level 2 Cache para estes programas, como eles são estruturados e como eles afetam a performance.

6. Anexos

6.1. Simple Matrix Multiplication

6.1.1. C++ version - time tests and their average (s)

Simple - C++	Time	LD1	LD2
1	0,192	244737133	39334666
2	0,191	244786303	40662153
3	0,193	244775992	41920572
4	0,194	244745655	39600318
5	0,196	244749999	38669926
1	1,374	1235508371	309243310
2	1,201	1228522821	284010616
3	1,075	1228084105	326599783
4	1,112	1223721395	268408492
5	1,406	1229482674	266934771
1	3,325	3523104679	1048821523
2	3,238	3433864589	943530365
3	3,376	3529707931	1164270861
4	3,775	3523253659	1758418791
5	3,541	3529874859	1340403333
1	17,879	9084516187	8065684508
2	18,065	9089556691	6652631488
3	17,666	9076250665	7399884362
1	38,541	17662381677	22767519385
2	38,062	17630565927	23597309763
3	39,593	17661780771	23534352529
1	68,468	30881225329	51470993242
2	68,338	30874597894	51192881083
3	67,661	30881396420	51061849800
1	114,044	50303678304	94216352420
2	114,044	50294027098	95739365610
3	114,14	50294417818	95181037856

SIZE - Simple	Tempo	LD1	LD2
600	0,193	244749999	39600318
1000	1,201	1228522821	284010616
1400	3,376	3523253659	1164270861
1800	17,879	9084516187	7399884362
2200	38,541	17661780771	23534352529
2600	68,338	30881225329	51192881083
3000	114,044	50294417818	95181037856

6.1.2. Java version - time tests and their average (s)

Line - C++	Time	LD1	LD2
1	0,11	27163214	58392327
2	0,111	27118255	57584233
3	0,112	27123528	57384957
1	0,512	126394392	266632338
2	0,509	126237718	266368978
3	0,501	126234886	263730978
1	1,65	586540302	729784599
2	1,632	631581887	726150558
3	1,629	631616533	725889151
1	3,732	1876070050	1488424940
2	3,626	2188620603	1469261366
3	3,632	2188550985	1488602904
1	7,545	4029753644	2684137650
2	6,831	4030880836	2713877365
3	6,857	4031716896	2694293863
1	11,328	6793527799	4419143074
2	11,412	6793416932	4447020416
3	12,067	6788116660	4411285887
1	17,545	10430742467	6804067467
2	17,496	10427208839	6672648534
3	17,637	10426776645	6869553534

Line - C++ - Big Times	Time	LD1	LD2
1	47,185	27245160732	17363226825
2	46,594	27244116099	17677551977
3	46,74	27250887450	17865875562
1	160,829	91838606080	61971925408
2	162,437	91669432805	60456220490
3	160,148	91723783449	60328679282
1	388,075	216886147095	152706204262
2	386,239	216916037179	157010500462
3	385,099	216944175688	167016687833
1	763,458	421852351905	438168279372
2	769,616	421552370028	442580884285
3	767,067	421519899357	449155835477

SIZE - Line	Tempo	LD1	LD2
600	0,111	27123528	57584233
1000	0,509	126237718	266368978
1400	1,632	631581887	726150558
1800	3,632	2188550985	1488424940
2200	6,857	4030880836	2694293863
2600	11,412	6793416932	4419143074
3000	17,545	10427208839	6804067467
4096	46,74	27245160732	17677551977
6144	160,829	91723783449	60456220490
8192	386,239	216916037179	157010500462
10240	767,067	421552370028	442580884285

Simple - Java	Time	Line - Java	Time
1	0,34	1	0,115
2	0,348	2	0,118
3	0,343	3	0,116
1	1,284	1	0,499
2	1,321	2	0,503
3	1,355	3	0,503
1	4,669	1	1,604
2	4,166	2	1,62
3	4,29	3	1,712
1	18,296	1	3,512
2	19,252	2	3,516
3	18,626	3	3,605
1	39,488	1	6,49
2	39,195	2	6,486
3	40,323	3	6,51
1	70,731	1	10,726
2	69,463	2	10,842
3	69,666	3	10,759
1	116,09	1	17,408
2	117,47	2	17,593
3	114,501	3	16,65

SIZE	Tempo - Simple - Java	Tempo - Line - Java
600	0,343	0,116
1000	1,321	0,503
1400	4,29	1,62
1800	18,626	3,516
2200	39,488	6,49
2600	69,666	10,759
3000	116,09	17,408

6.3. Block Matrix Multiplication

6.3.1. C/C++ version - time tests and their average (s)

Block - C++	Attempt	Time	LD1	LD2
128	1	31,917	9715810397	32490946034
128	2	31,003	9718932627	32585113656
128	3	31,138	9713974352	32615862186
256	1	27,095	9079216809	22925919257
256	2	27,164	9079250859	22895613936
256	3	27,363	9079335401	22937649178
512	1	25,46	8762105123	19646357260
512	2	25,694	8761270508	19759705899
512	3	26,176	8761953595	19592579128
128	1	105,186	32819640273	111411531507
128	2	104,487	32811334391	111235535986
128	3	104,315	32823190262	109620851863
256	1	92,714	30642539591	77413892167
256	2	94,205	30647613263	77376246241
256	3	92,072	30644072170	77398617727
512	1	93,42	29610204047	65895217730
512	2	93,688	29614195727	65873939527
512	3	92,727	29609232133	67370559264
128	1	265,162	77800057280	262347962048
128	2	275,202	77799600117	261196069694
128	3	271,68	77767378192	261336746676
256	1	384,138	73114253525	156021934103
256	2	386,836	73101754909	154134253014
256	3	387,847	73079920245	152984402210
512	1	328,36	70397161050	129610280000
512	2	329,593	70409038479	129595329749
512	3	329,093	70411888192	129225771986
128	1	548,47	154727399609	519333823477
128	2	548,656	154722434797	526805351450
128	3	550,117	154722172544	526702833381
256	1	509,779	142850480997	368337520719
256	2	515,107	142855439957	360874193001
256	3	513,26	142852598316	359989194712
512	1	504,971	136999987378	312539595719
512	2	507,932	136983089828	305850120112
512	3	511,434	136983161878	312156257425

SIZE - Block Size	Tempo - 128	LD1 - 128	LD2 - 128	Tempo - 256	LD1 - 256	LD2 - 256	Tempo - 512	LD1 - 512	LD2 - 512
4096	31,138	9715810397	32585113656	27,164	9079250859	22925919257	25,694	8761953595	19646357260
6144	104,487	32819640273	111235535986	92,714	30644072170	77398617727	93,42	29610204047	65895217730
8192	271,68	77799600117	261336746676	386,836	73101754909	154134253014	329,093	70409038479	129595329749
10240	548,656	154722434797	526702833381	513,26	142852598316	360874193001	507,932	136983161878	312156257425

6.4. Line Matrix Multiplication with Paralellism

6.4.1 Pragma 1 - C/C++ version - time tests and their average (s)

Line Pragma 1 - C++	Time	LD1	LD2
1	0,020164	3408099	7270994
2	0,019768	3397886	7179588
3	0,017562	3401911	7163991
1	0,087571	15818448	33050990
2	0,088549	15801117	32547151
3	0,086314	15802400	33140722
1	0,261182	73493479	91619079
2	0,268577	78973930	90990009
3	0,267494	78994152	91062655
1	0,574616	235627953	186685890
2	0,588791	273323380	185706654
3	0,778653	273233446	178303972
1	1,316303	505382342	331006120
2	1,169705	505290528	333869784
3	1,194064	505224734	335617441
1	2,304931	851281858	540211879
2	1,939304	851148605	548747286
3	2,007462	851258213	544978098
1	3,472988	1306470733	847706872
2	3,133638	1306369192	840042142
3	3,629227	1307570959	829492651
1	7,4417	3424421571	2233064979
2	8,686506	3419966736	2186074357
3	8,659028	3422017004	2223026546
1	29,067009	11518032152	8210484780
2	32,989925	11504129364	8145216179
3	31,270924	11509900027	8112037636
1	79,632385	27143896613	22499724450
2	82,299575	27159022323	
3	77,923367	27168605755	22569737263
1	169,11005	52753350069	58712887455
2	165,518	52770016215	58342519216
3	164,230822	52752889654	59098132836

SIZE - Line Pragma 1	Tempo	LD1	LD2
600	0,019768	3401911	7179588
1000	0,087571	15802400	33050990
1400	0,267494	78973930	91062655
1800	0,588791	273233446	185706654
2200	1,194064	505290528	333869784
2600	2,007462	851258213	544978098
3000	3,472988	1306470733	840042142
4096	8,659028	3422017004	2223026546
6144	31,270924	11509900027	8145216179
8192	79,632385	27159022323	22534730857
10240	165,518	52753350069	58712887455

6.4.2 Pragma 2 - C/C++ version - time tests and their average (s)

Line Pragma 2 - C++	Time	LD1	LD2
1	0,169051	8056179	32408174
2	0,175938	8038096	32442248
3	0,167825	8042573	32432240
1	0,491601	28862691	110827606
2	0,48376	28791199	110957015
3	0,469173	28794456	111273897
1	1,351534	68973842	229264061
2	1,397882	68540662	228001345
3	1,749183	68767186	219391602
1	2,905444	133456579	365959828
2	2,937361	133314167	365759075
3	2,928494	133194437	366758249
1	5,434626	231917811	584160265
2	5,391281	231901594	580815604
3	5,381328	231270510	581016514
1	8,640893	364676925	867354824
2	8,698954	365105134	868190022
3	8,670612	364610281	863901667
1	13,156243	541625655	1217589286
2	13,162159	541883028	1209240759
3	13,115617	541359849	1223105613
1	33,111449	1231595904	2449041387
2	32,651424	1228943732	2479024288
3	31,981477	1228485031	2409902814
1	105,368767	4219134622	8166239529
2	104,624002	4194158617	8040058139
3	104,604575	4237372095	8063291790
1	247,48143	9789052582	16085926990
2	250,981091	9772365377	16296191450
3	248,403117	9986540336	16519186755
1	476,293677	29068491959	31780314599
2	476,867254	28887127056	32503496757
3	478,290613	27993390784	32486965777

SIZE - Line Pragma 2	Tempo	LD1	LD2
600	0,169051	8042573	32432240
1000	0,48376	28794456	110957015
1400	1,397882	68767186	228001345
1800	2,928494	133314167	365959828
2200	5,391281	231901594	581016514
2600	8,670612	364676925	867354824
3000	13,156243	541625655	1217589286
4096	32,651424	1228943732	2449041387
6144	104,624002	4219134622	8063291790
8192	248,403117	9789052582	16296191450
10240	476,867254	28887127056	32486965777