Universidade do Porto

Faculdade de Engenharia

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

4º ano 2º semestre

Relatório

Projeto 1

Performance evaluation of a single core

13/03/2017

Grupo:

- **Hugo Drumond - nº 201102900 - hugo.drumond@fe.up.pt**
- **Pedro Moura - nº 201306843 - up201306843@fe.up.pt**

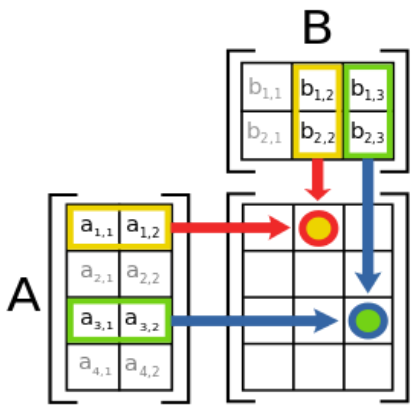
1 - Problem description and algorithms explanation

No âmbito da unidade curricular de computação Paralela, foi proposto efectuar um estudo sobre o desempenho do processador tendo em conta o impacto da hierarquia de memória durante os acessos a um grande volume de dados.

Para tal foi sugerido o produto de matrizes segundo dois algoritmos. Bem como a amostragem de resultados e comparação destes, acrescentando explicações e conclusões críticas dos mesmos.

1.1 - Algoritmo Line By Column

Este algoritmo baseia-se na transformação imediata para código da definição do produto entre matrizes.



$(AB)_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j}$, *m* representa o número de colunas de *A* e linhas de *B*. Que terá de ser igual para que o produto entre estas matrizes seja válido. A matrix resultado (*AB*) terá a dimensão “linhas de *A*” vezes “colunas de *B*”.

A codificação será:

```
for (size_t row_a = 0; row_a < matrix_a.rows_length; row_a++) {
  for (size_t column_b = 0; column_b < matrix_b.columns_length; column_b++) {
    double temp_sum = 0;
    for (size_t k = 0; k < matrix_a.columns_length; k++) {
      temp_sum += matrix_a(row_a, k) * matrix_b(k, column_b);
    }
    (*matrix_result)(row_a, column_b) = temp_sum;
  }
}
```

Ao fazer o produto desta maneira é possível saber o resultado final de $(AB)_{i,j}$ após sair do loop mais interior. No entanto, há pouco aproveitamento da informação que está em cache de cada matriz. Ou seja tira pouco proveito do modo de funcionamento de carregamento de posições de memória para cache, que funciona de acordo com as observações de *Temporal and Spatial Locality*. Isto advém do facto de estarmos a dar saltos nas linhas da matriz *B* não aproveitando as posições anteriores do array unidimensional que já estão cached. Aumentando assim o número de cache missings e por conseguinte o tempo de execução.

Este algoritmo tem uma complexidade temporal $O(m^3)$. E um número de instruções de vírgula flutuante $2m^3$ = (para cada linha de A) * (para cada coluna de B) * (para cada elemento) * (1 soma + 1 multiplicação).

1.2 - Algoritmo Line By Line

Neste algoritmo o cálculo é feito multiplicando um elemento de uma linha da Matrix *A* pela linha da Matrix *B* respetiva incrementando o valor da Matrix *Resultado* por passos.
Por exemplo:

$[[a_{11},a_{12}] , [a_{21},a_{22}]] \times [[b_{11},b_{12}], [b_{21},b_{22}]]$

Matrix resultado:

$[[\mathbf{a_{11}b_{11}}, \mathbf{a_{11}b_{12}}] , [0, 0]] \rightarrow [[a_{11}b_{11}+\mathbf{a_{12}b_{21}}, a_{11}b_{12}+\mathbf{a_{12}b_{22}}], [0, 0]] \rightarrow$

$\rightarrow [[...], [\mathbf{a_{21}b_{11}}, \mathbf{a_{21}b_{12}}]] \rightarrow [[...], [a_{21}b_{11}+\mathbf{a_{22}b_{21}}, a_{21}b_{12}+\mathbf{a_{22}b_{22}}]]$

A codificação será:

```
for (size_t row_a = 0; row_a < matrix_a.rows_length; row_a++) {
  for (size_t k = 0; k < matrix_a.columns_length; k++) {
    for (size_t column_b = 0; column_b < matrix_b.columns_length;
         column_b++) {
      (*matrix_result)(row_a, column_b) +=
          matrix_a(row_a, k) * matrix_b(k, column_b);
    }
  }
}
```

Este algoritmo é uma adaptação do *Line by Column* que aproveita a linha da Matrix *B* que já está em cache de modo a melhorar o desempenho. No entanto, o cálculo de cada posição $(AB)_{i,j}$ é feito de forma incremental.

Tem a mesma complexidade temporal e número de instruções de vírgula flutuante que o *Line by Column***.**

1.3 - Codificação dos Algoritmos

Foi criada uma classe *Matrix*. Esta contém um array unidimensional e o número de rows e colunas.

O acesso às posições do array unidimensional no C++ é feito através da chamada ao operador parenthesis que foi *overloaded*, *inlined* e que não é sujeito a verificação de dimensões. Os algoritmos pertencem à classe mas são *static*.

Não utilizou-se quaisquer function calls que pudessem afetar o desempenho nos loops. A codificação em todas as linguagens foi feita da maneira mais próxima possível. Inclusivé a activação das otimizações de cada compilador.

2 - Performance metrics and evaluation methodology

No caso do C++, para ser possível uma análise detalhada de cada um dos algoritmos e as suas versões paralelas, foram feitas várias medições usando a biblioteca **PAPI** (Performance Application Programming Interface). A ferramenta de análise de performance **perf** (Performance Counters for Linux) foi utilizada para fazer a comparação entre as linguagens C++, Java, C#, Python e Golang (por falta de tempo as implementações em Python e Golang não foram analisadas). Todos os testes foram corridos no *linux*. Usou-se o gcc 6, jdk 1.8, .NET Core SDK 1.0.1, python3 e golang1.8.

O mecanismo de medição foi automatizado: usando bash scripts bench.sh, run.sh, benchPerf.sh e benchPapi.sh para o cpp; cmake e makefile com targets com e sem PAPI e com modo debug e release; script Python3 converter.py de conversão de outputs dos programas para csv para importação no excel; formulas excel para cálculo automático após importação dos csv.

Recolheu-se bastantes métricas tanto no PAPI como no perf no entanto as mais importantes são: o tempo de execução, L1 e L2 data cache misses.

Também calculou-se certos valores interessantes para a análise: MFLOPS, L1 data cache misses / MFLOP, L2 data cache misses / MFLOP, L1 data cache misses / Instructions, L2 data cache misses / Instructions, comparação de tempo por rácio lbc / lbcx e lbl / lblx (speedup), comparação de tempo entre linguagens. Cache misses / MFLOP = Cache Misses / $2m^3$

As restantes métricas e cálculos encontram-se nos ficheiros excel fornecidos. Nestes separou-se as medições, selecção e agregação de medições, cálculos e gráficos por sheets.

Foram feitas 6 medições por cada execução e computador. Nos cálculos é utilizada a medição que demorou menos tempo. Tal foi feito de modo a minimizar o impacto de outros processos que estivessem a correr em paralelo.

As características do computador que produziu o documento dados-pedro-new-openmp-implementation.xlsx são as seguintes: Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz, 4 cores, 2 threads por core, CPU max 3400 MHz, CPU min 800 MHz, L1d cache 32K, L1i cache 32K, L2 cache 256k, L3 cache 6144K, L1d e L1i e L2 por core, L3 shared. A análise e os gráficos apresentados foram baseados nestes testes.

Os testes presentes nos ficheiros excel que contêm o nome hugo foram realizados com o hardware seguinte: CPU Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz, 2 cores, 2 threads por core. Esta informação só é usada para analisar o comportamento com threads > 2.

3 - Results and analysis

3.1 - Tempos de Execução

Nos documentos fornecidos nos anexos e nos ficheiros excel é possível verificar que a multiplicação de matrizes Line By Line é sempre mais rápida que a do Line By Column seja qual for a linguagem onde foi feita a implementação. Na versão Line By Column a diferença entre o tempo de execução do C++ e as outras linguagens é mais reduzida que no caso Line By Line. Das linguagens comparadas o C++ é claramente superior aumentando a sua vantagem com o aumento da dimensão da Matriz. Como regra geral temos que o tempo de execução do C++ < Java < C#. O Java embora seja corrido numa máquina virtual tem uma performance muito elevada. Os tempos deste encontram-se muito mais perto do C++ do que o C#. Na versão Line by Column aquando o incremento do número de threads não existe uma diferença tão acentuada como no Line By Line, pode-se dizer então, que este último é mais escalável.

3.2 - Análise de cache

No *Line By Line* há uma diminuição enorme no número de cache missings. O que leva a um tempo de execução muito inferior. Chegou-se também à conclusão que uma boa solução sequencial é em muitos casos melhor que uma má versão em paralelo. Por exemplo, com uma matriz de dimensão 3000, o lbl (Line By Line sem openmp) quase que consegue ser mais rápido que o lbc3t (Line By Column com 3 threads). Pela análise dos gráficos conclui-se que o número de misses L1 / instruções e misses L2 / instruções das versões Line By Column (lbc e lbcx) é constante, respectivamente 16% e 1.80%. Constata-se que o número de misses L1 do Line By Line é muito inferior e que para dimensões pequenas praticamente não existem data cache misses no L2. Deste modo evitando ter de ir ao L3 e possivelmente à memória principal.

3.3 - Análise de *Floating Point Operations*

Há uma diferença bastante acentuada entre o número de MFLOPS do lbl e lbc. Tal deve-se ao lbl não perder tanto tempo a carregar dados para cache. Permitindo assim efectuar mais cálculos por unidade de tempo. Verificou-se que ao aumentar o número de threads no lbl há sempre um aumento no número de MFLOPS. O mesmo só não acontece no lbc para 4 threads com dimensão 600.

3.4 - Análise de pragma, schedule e threads

O pragma omp parallel for foi colocado no ciclo for mais exterior de modo a evitar o overhead da criação de threads. Foi feita uma implementação com pragmas no interior para teste, presente nos ficheiros excel old-openmp, onde é claro este overhead. Foi usada uma distribuição de trabalho de modo estático uma vez que a quantidade de trabalho é igual

para todas as threads. Além do mais foi usado o schedule(static) default que devide a gama por chunks loop_count/number_of_threads. Por exemplo com um for com 12 iterações, temos: Thread 0: 0,1,2,3; Thread 1: 4,5,6,7; Thread 2: 8,9,10,11 . No nosso entender e pelos testes que realizámos este é o melhor chunk size. Todos os outros que testámos têm menor performance. Uma possível explicação é o facto de poder estar a acontecer false sharing. Por exemplo, com chunks com tamanho 1, a thread 0 iria estar a atualizar a iteração 0 e a thread 1 a iteração 1 resultando em updates isolados e muito próximos ativando o mecanismo de controlo de inconsistência de cache que é indesejado. Também testámos o uso de collapse(2) uma vez que os loops são independentes um do outro no entanto é óbvio neste caso o problema de false sharing.

O speedup das versões com 4 threads é melhor que o das 3 threads, mas não é muito díspar. Pode-se dizer então que foi encontrado o melhoramento ideal para este algoritmo nesta máquina, 4 threads. Tal não acontece na máquina de 2 cores uma vez que as threads "competem" pelo uso da cache L1 e L2 de cada core quando threads > 2.

4 - Conclusions

As arquiteturas de CPUs atuais contêm mecanismos que permitem fazer prefetch em bloco de posições de memória principal para cache próxima do CPU (L1->L2->L3). Esta técnica tenta eliminar espaços mortos no CPU ao alimentá-lo com posições de memória próximas às que estão a ser utilizadas num dado momento. Tal evita o bottleneck de von Neumann. Cabe ao programador o exercício de construir algoritmos que usem ao máximo a informação que já está em cache, mas que evitem a alteração de variáveis independentes por threads que partilhem a mesma cache line (false sharing).

O algoritmo Line By Line é superior ao Line By Column uma vez que usa a cache de uma maneira mais oportunística. Evitando assim perda de tempo a carregar posições de memória CPU <- L1 per core <- L2 per core <- L3 shared <- Main Memory. É esta a razão pelos MFLOPS do Line By Line serem muito superiores aos do Line By Column.

Tanto o Java como o C# são linguagens que não são desejáveis quando se trata de problemas de natureza matemática, física, etc que têm de ser escaláveis. Tal deve-se ao overhead de correr numa Máquina Virtual.

Existem soluções mais sofisticadas que aumentam ainda mais o desempenho da multiplicação de matrizes. Por exemplo algoritmos que usem loop blocking/tilling e Advanced Vector Extensions.

5 - Referências:

- Material fornecido nas aulas de Computação Paralela pelo Professor
- An Introduction to Parallel Programming by Peter Pacheco
- http://stackoverflow.com/questions/15829223/loop-tiling-blocking-for-large-dense-matrix-multiplication

6 - Anexos:

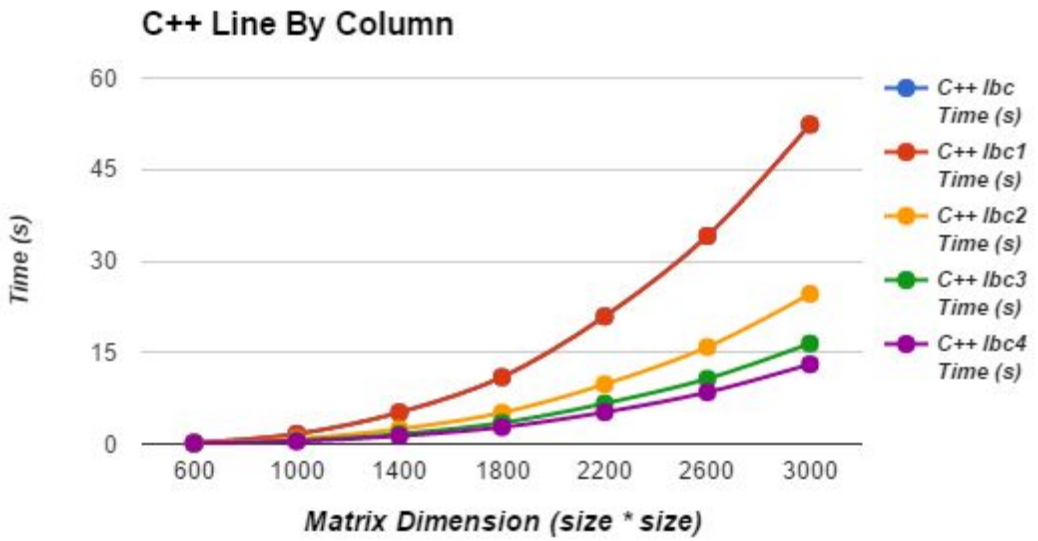
6.1 - Código openmp:

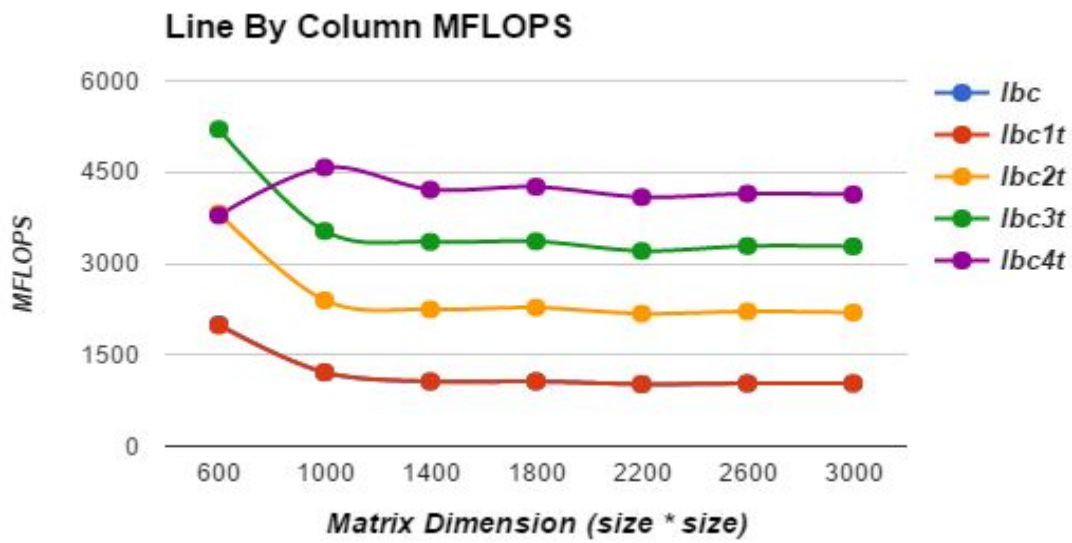
```
#pragma omp parallel for num_threads(number_of_threads) schedule(static)
  for (size_t row_a = 0; row_a < matrix_a.rows_length; row_a++) {
    for (size_t column_b = 0; column_b < matrix_b.columns_length; column_b++) {
      double temp_sum = 0.0;
      for (size_t k = 0; k < matrix_a.columns_length; k++) {
        temp_sum += matrix_a(row_a, k) * matrix_b(k, column_b);
      }
      (*matrix_result)(row_a, column_b) = temp_sum;
    }
  }
```

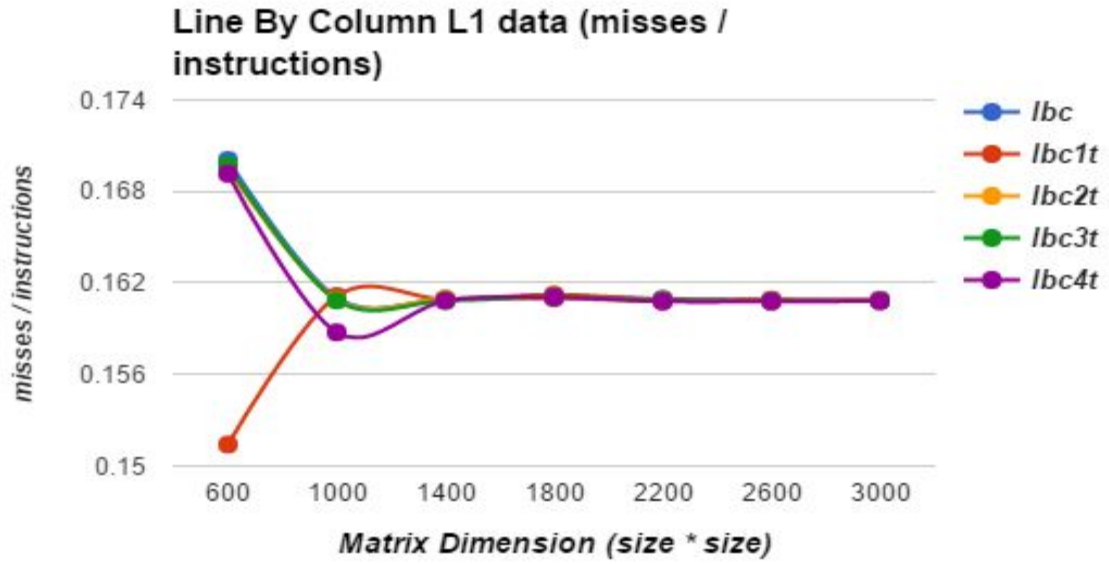
```
#pragma omp parallel for num_threads(number_of_threads) schedule(static)
  for (size_t row_a = 0; row_a < matrix_a.rows_length; row_a++) {
    for (size_t k = 0; k < matrix_a.columns_length; k++) {
      double row_a_column_k = matrix_a(row_a, k);
      for (size_t column_b = 0; column_b < matrix_b.columns_length;
           column_b++) {
        (*matrix_result)(row_a, column_b) += row_a_column_k * matrix_b(k, column_b);
      }
    }
  }
```

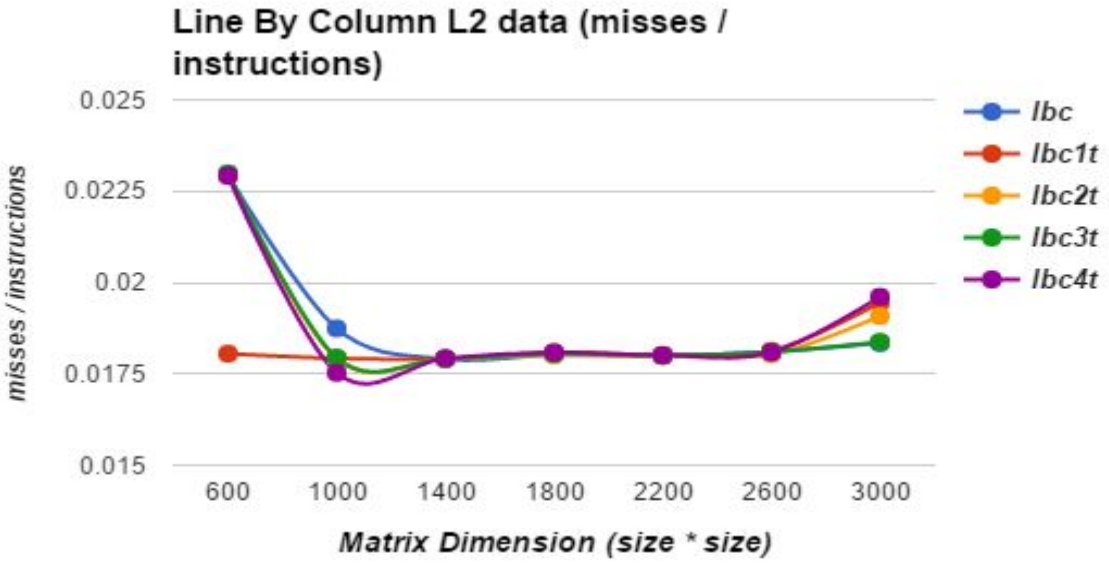
6.2 - Gráficos PAPI:

6.2.1 - Line By column C++

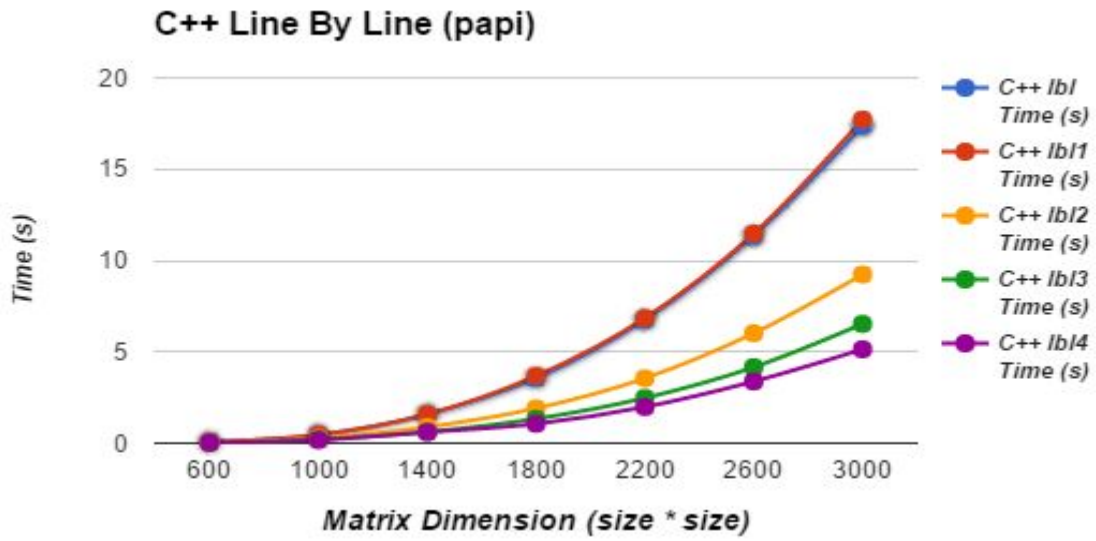


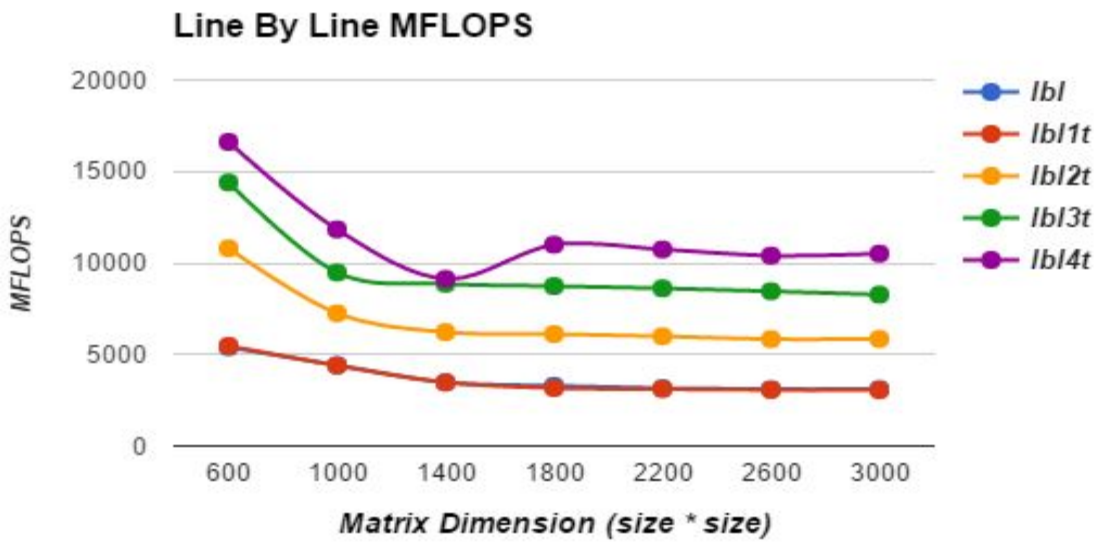


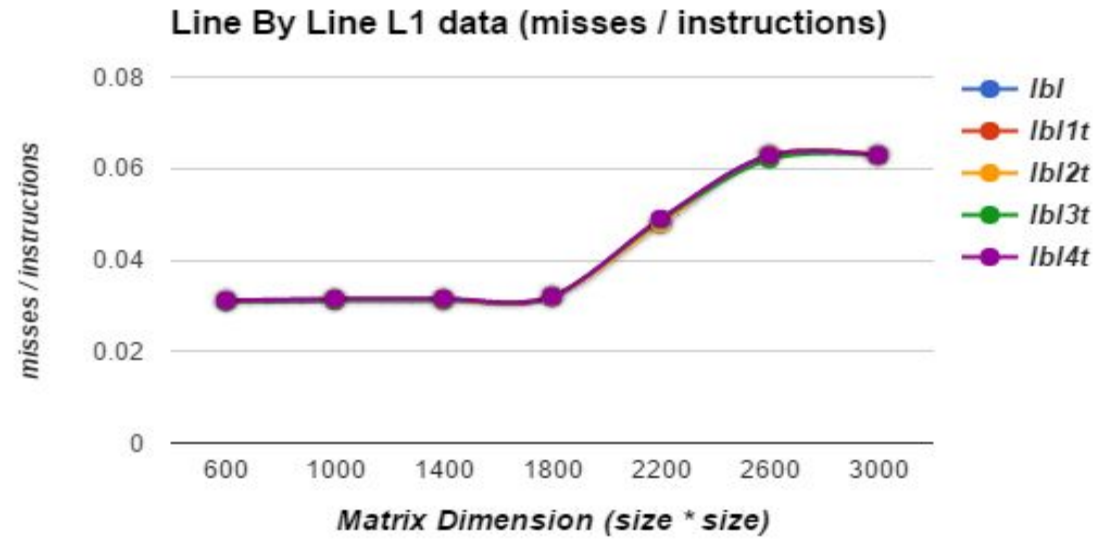

Line By Column L1 data (misses / instructions)
0.174
0.168
0.162
0.156
0.15
misses / instructions
600
1000
1400
1800
2200
2600
3000
Matrix Dimension (size * size)
lbc
lbc1t
lbc2t
lbc3t
lbc4t


Line By Column L2 data (misses / instructions)
0.025
0.0225
0.02
0.0175
0.015
misses / instructions
600
1000
1400
1800
2200
2600
3000
Matrix Dimension (size * size)
lbc
lbc1t
lbc2t
lbc3t
lbc4t

6.2.2 - Line By Line C++



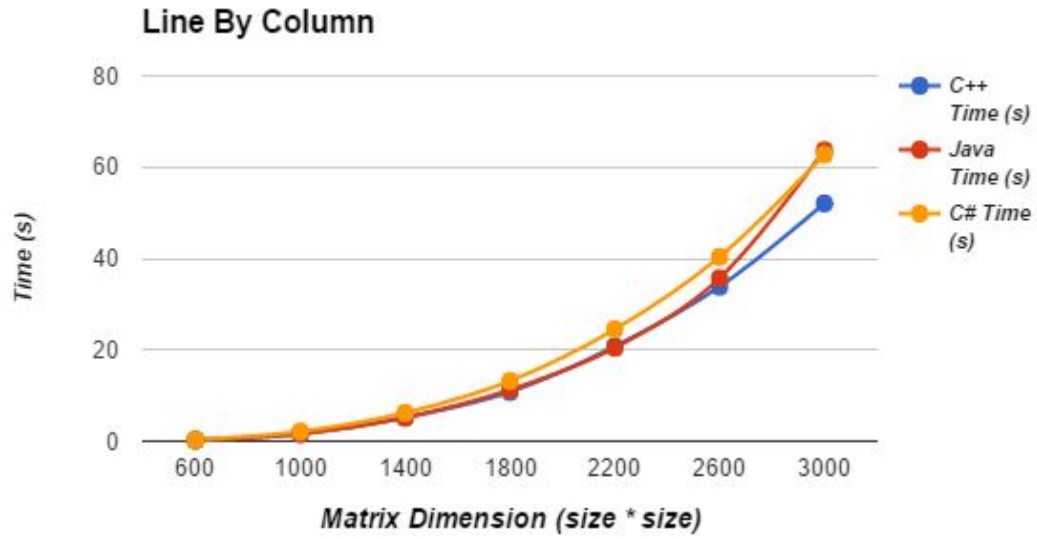


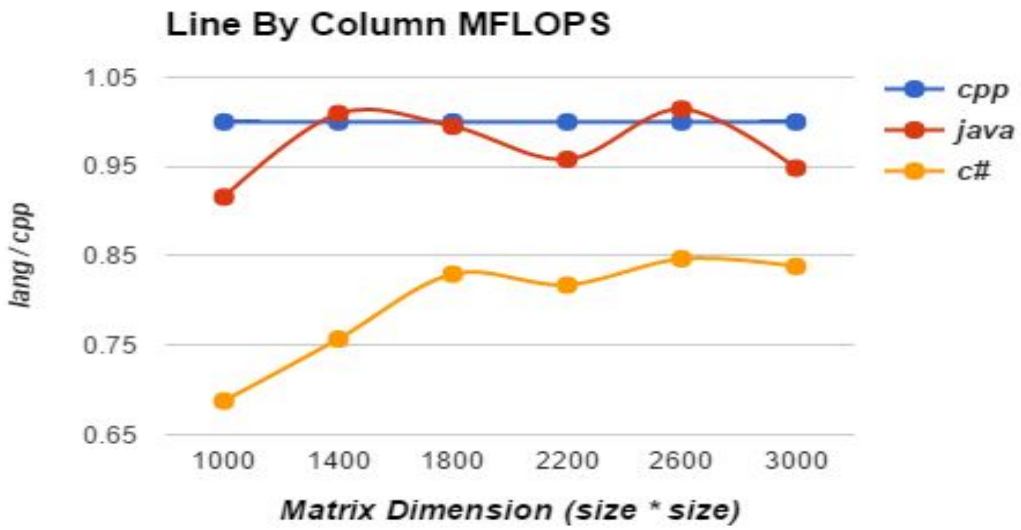

Line By Line L1 data (misses / instructions)
misses / instructions
0.08
0.06
0.04
0.02
0
600
1000
1400
1800
2200
2600
3000
Matrix Dimension (size * size)
lbl
lbl1t
lbl2t
lbl3t
lbl4t

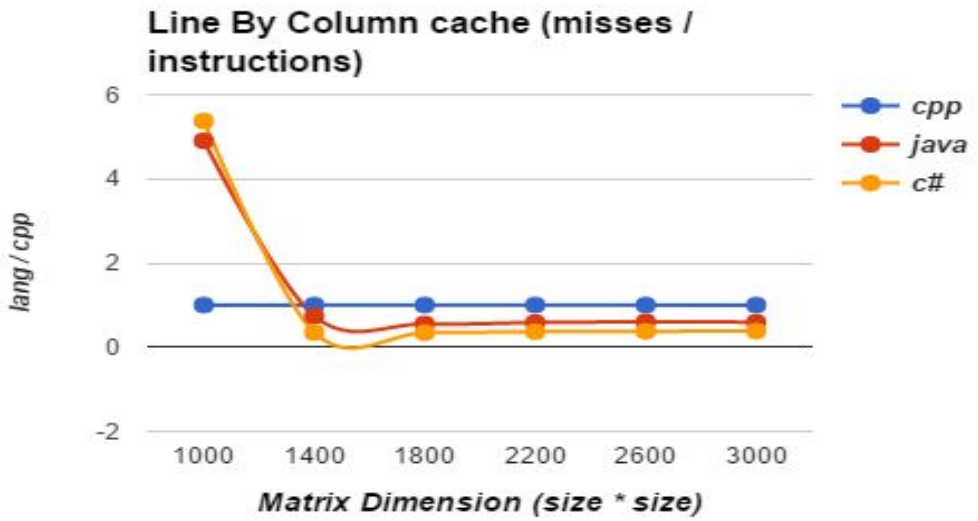

Line By Line L2 data (misses / instructions)
misses / instructions
0.03
0.0225
0.015
0.0075
0
600
1000
1400
1800
2200
2600
3000
Matrix Dimension (size * size)
lbl
lbl1t
lbl2t
lbl3t
lbl4t

6.3 - Graficos perf

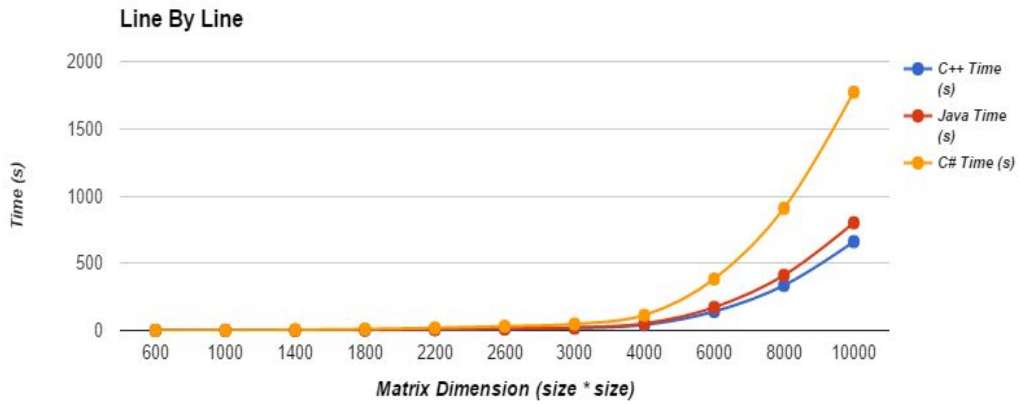
6.3.1 - Line By Column

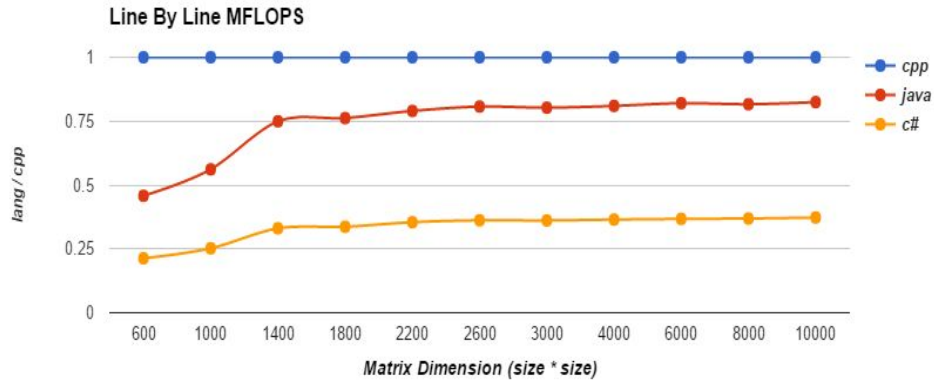


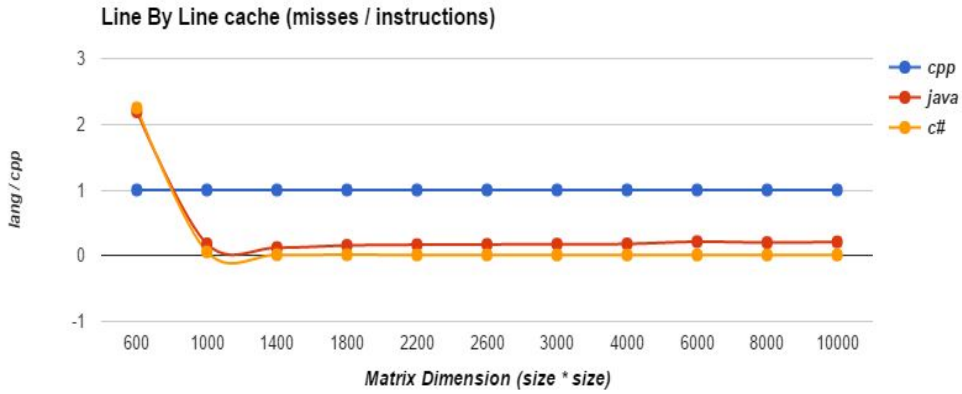




6.3.2 - Line By Line






Line By Line cache (misses / instructions)
3
2
1
0
-1
lang / cpp
600
1000
1400
1800
2200
2600
3000
4000
6000
8000
10000
Matrix Dimension (size * size)
cpp
java
c#