****

**Project 1**

**Performance Evaluation of a Single Core**

*Relatório*

Computação Paralela

4º Ano do Mestrado Integrado em Engenharia Informática e Computação

**Elementos do Grupo:**

Diogo Vaz Nunes - 201108019 - ei11065@fe.up.pt

Mike Santos Pinto – 201103127 – ei11050@fe.up.pt

7 de Abril de 2015

# **Índice**

[**Descrição do Problema** 3](#_Toc415956671)

[**Algoritmos** 4](#_Toc415956672)

[**Demonstração e Análise de Resultados** 5](#_Toc415956673)

[**Conclusão** 8](#_Toc415956674)

# **Descrição do Problema**

Atualmente a tecnologia evolui a um rácio temporal muito alto, as diferenças com as tecnologias de há duas décadas atrás são muito significativas. Com a abordagem do processamento *multi core* ao invés do *single core* com frequências superiores e a existência de grandes quantidades de memória, é necessário entender quais as vantagens e desvantagens, em termos de processamento, em ter mais ou menos núcleos e a influência do funcionamento do acesso da memória na resolução de certos problemas.

O objetivo deste projeto é estudar, através do cálculo de multiplicação de matrizes, qual o impacto no desempenho do processador tendo em conta a hierarquia da memória no acesso a grande quantidade de dados.

O estudo foi feito com recurso a uma máquina com as seguintes especificações:

* **Processador:** Intel Core2 Quad Q6600 G0 @2.4GHz
* **Memória Ram:** 2x2GB *Kingston* DDR2 800Mhz @667MHz
* **Disco Rígido:** Disco Magnético 1TB 7200RPM - *Toshiba*

# **Algoritmos**

**Tipo de algoritmo**

O algoritmo utilizado no cálculo do produto matricial é um algoritmo *naïve*.

Consiste na multiplicação de cada linha da primeira matriz com uma coluna da segunda matriz obtendo-se um valor da matriz resultado, este processo repete-se até que a matriz resultado esteja totalmente calculada (ver fig.1).

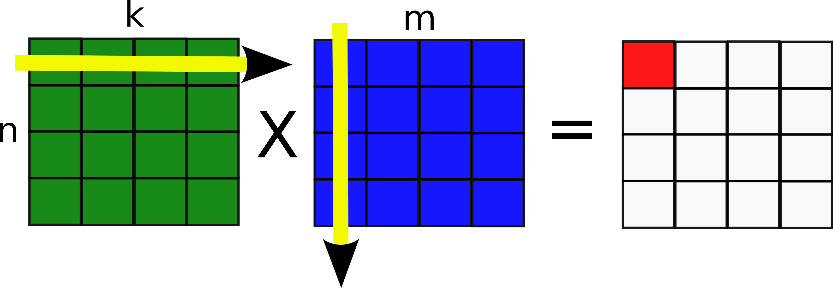


Figura 1

**1ª Versão – algoritmo *naïve* ‘normal’**

O algoritmo é essencialmente igual ao descrito em termos de implementação. Isto dá origem a uma ordem espacial *S(n3)* e uma ordem temporal *O(2n3)* no cálculo de matrizes quadradas.

**2ª Versão – algoritmo *naïve* ‘otimizado’**

Esta versão tem em conta a arquitetura da máquina e como é feito o acesso a memória, especificamente como é feito o acesso à memória cache e o carregamento dos dados entre esta memória e a principal.

O cálculo deixa de ser feito em linha por coluna e é realizado linha por linha, cada elemento da primeira matriz é multiplicado pela linha correspondente da segunda matriz.

Desta forma, o algoritmo possui uma ordem espacial *S(n3)* e uma ordem temporal *O(n3)* no cálculo de matrizes quadradas. Realça-se que a otimização pouco ou nada altera estas variáveis, a otimização feita faz-se notar nos tempos de acesso a memória.

# **Demonstração e Análise de Resultados**

**Testes realizados**

Neste estudo foram realizados vários testes, dividindo-se em 2 categorias.

A primeira refere-se aos testes de execução sequencial, implementados em 2 linguagens diferentes, C++ e Java, com as duas versões do algoritmo. Nesta categoria foram utlizadas matrizes 600x600 até 3000x3000 com incrementos de 400 unidades em ambas as versões e de tamanho e 4000x4000 até 10000x10000 com incrementos de 2000 unidades apenas no otimizado.

A segunda é composta pelos testes de execução paralela, implementados em c++, com as duas versões do algoritmo, utilizando matrizes 600x600 até 3000x3000 com incrementos de 400 unidades de tamanho.

**Execução sequencial**

* **Algoritmo Normal**
* **Algoritmo Otimizado**

**Execução Paralela - C++ com OpenMP**

* **Algoritmo Normal**
* **Algoritmo Otimizado**

**Análise de Resultados**

* **Execução sequencial**

Existe uma disparidade em termos de desempenho óbvia entre as linguagens de C++ e Java. Em comparação de algoritmos, o otimizado claramente é melhor, demonstrando capacidades +4x superiores ao algoritmo normal para o mesmo problema. Ambos os algoritmos seguem uma tendência de perda de desempenho nos intervalos 600 a 3000. No entanto, no algoritmo otimizado, para valores de tamanho entre 4000 e 10000, verifica-se uma tendência de aumento de desempenho com o aumento de tamanho.

* **Execução paralela**

À semelhança com a execução sequencial, o algoritmo otimizado tem um desempenho superior ao algoritmo normal.

No algoritmo normal, utilizar 2 a 4 *threads* tem resultados semelhantes, usar apenas uma *thread* é muito menos eficiente. Já algoritmo otimizado, verifica-se o esperado de quanto mais *threads* (1 a 4 testadas) melhor o desempenho.

A tendência no algoritmo normal é a diminuição do desempenho, quanto maior for o tamanho das matrizes. No algoritmo otimizado, todos têm uma tendência de desempenho decrescente, porém é mais acentuada quanto maior o número de *threads.*

* **Sequencial versus Paralela**

De um modo geral, a versão paralela dos algoritmos é mais eficiente que a sua versão sequencial, considerando que todos os núcleos do processador (4) são utilizados. Com apenas três *threads* os resultados são semelhantes, ganhando o otimizado por uma pequena margem. Com uma ou duas threads os resultados são piores que o algoritmo sequencial, na máquina utilizada.

**Observações**

* Um facto importante de realçar será que os processadores fazem por si só uma otimização/paralelismo próprio, notou-se que mesmo utilizando a versão sequencial, o trabalho do processo era repartido por todos os núcleos do processador.
* Anexado a este documento está o código-fonte do programa utilizado para o estudo, tanto o de C++ como o de Java.
* Anexado a este documento está também uma séries de folhas de cálculo com todas as estatísticas relativas aos gráficos apresentados.

# **Conclusão**