Faculdade de Engenharia da Universidade do Porto

Performance Evaluation of a Single Core

Relatório

Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela, 4º ano

Pedro Dias Faria

ei11167@fe.up.pt

30 de Março de 2016

Descrição do Problema

Com a evolução da tectonologia a uma grande velocidade, as diferenças entre tecnologias de há umas décadas atrás e a atual são muito significativas.
Na existência do processamento *multi core*, com frequências muito superiores em relação ao *single core*, e a existência de grandes quantidades de memória, é preciso identificar as vantagens e desvantagens de ter mais ou menos núcleos e a influência de como é acedida a memória na resolução de certos problemas, em termos de processamento.
O objetivo deste projeto é estudar qual o impacto no desempenho do processador, tendo em conta a hierarquia da memória no acesso a grandes quantidades de dados.
Para isso, iremos tomar o problema do cálculo do produto de duas matrizes.

Especificações da máquina

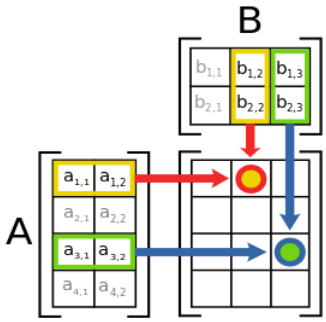
O estudo foi feito com recurso a uma máquina com as seguintes especificações:

- **Processador:** Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz
- **Memórias Ram:** 4GB Samsung DDR3 1600MHz
  2GB Hynix/Hyundai DDR3 1600MHz

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.



Algoritmo *naïve* 'básico' (ijk)

O algoritmo é igual ao descrito em termos de implementação.
Tem como ordem espacial $S(n^3)$ e ordem temporal $O(2n^3)$ no cálculo de matrizes quadradas.

Algoritmo *naïve* 'otimizado' (ikj)

Esta implementação do algoritmo tem em conta a arquitetura da máquina e o acesso a memória, especificamente como é feito o acesso à memória cache e o carregamento dos dados para a memória principal.
O cálculo passa a ser feito linha por linha, ou seja, cada elemento da primeira matriz é multiplicado pela linha correspondente da segunda matriz.
O algoritmo continua com ordem espacial $S(n^3)$ e ordem temporal $O(2n^3)$ no cálculo de matrizes quadradas, pois a otimização nada altera as variáveis. Apenas se nota nos tempos de acesso à memória.

Demonstração e Análise de Resultados

Testes realizados

Foram realizados testes em duas categorias:

- **Execução sequencial:**
  Estes testes foram executados na implementação dos dois algoritmos em duas linguagens diferentes (C++ e Java);
  Foram utilizadas matrizes com tamanhos desde 600x600 até 3000x3000, com incrementos de 400 unidades em ambos os algoritmos, e de 4000x4000 ate 10000x10000 com incrementos de 2000 unidades no otimiziado.
- **Execução paralela:**
  Estes testes foram executados na implementação dos dois algoritmos na linguagem C++, com a utilização da API OpenMP (OMP)
  Foram utilizadas matrizes com tamanhos desde 600x600 até 3000x3000, com incrementos de 400 unidades em ambos os algoritmos, e de 4000x4000 ate 10000x10000 com incrementos de 2000 unidades no otimiziado.
  Para cada uma das iterações, foi também testada a utilização de 1, 2, 3 e 4 threads para o seu processamento.

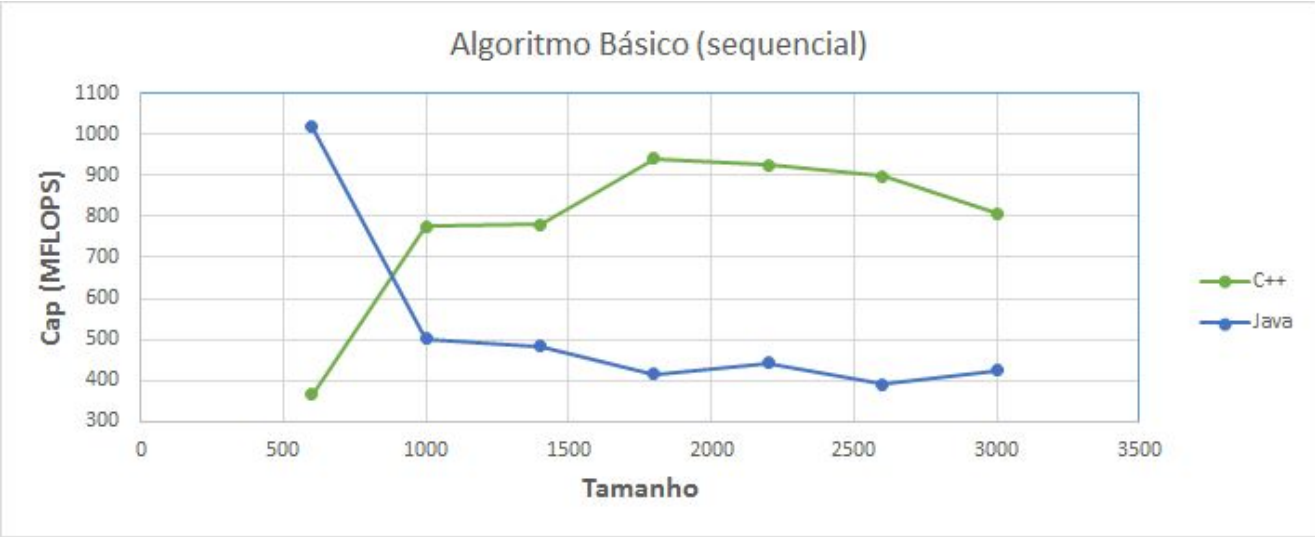
Foi também calculada a capacidade computacional de cada iteração (em MFLOPS) através da fórmula:

$$Cap = 2n^3 / t\,(FLOPS)$$

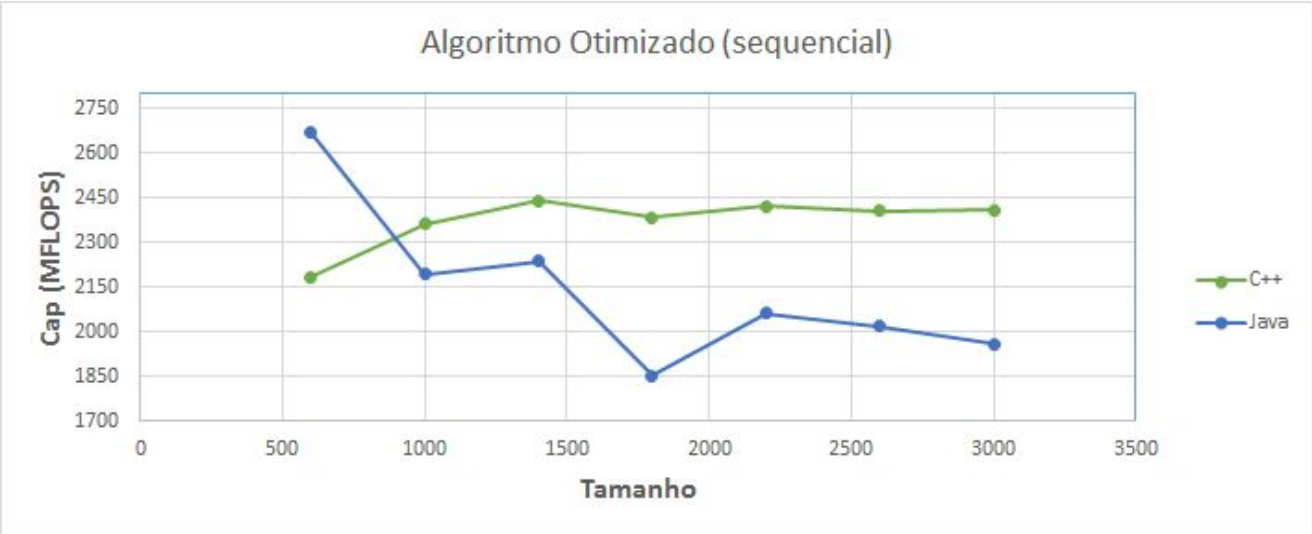
sendo n o tamanho da matriz e t o tempo total do cálculo do produto.

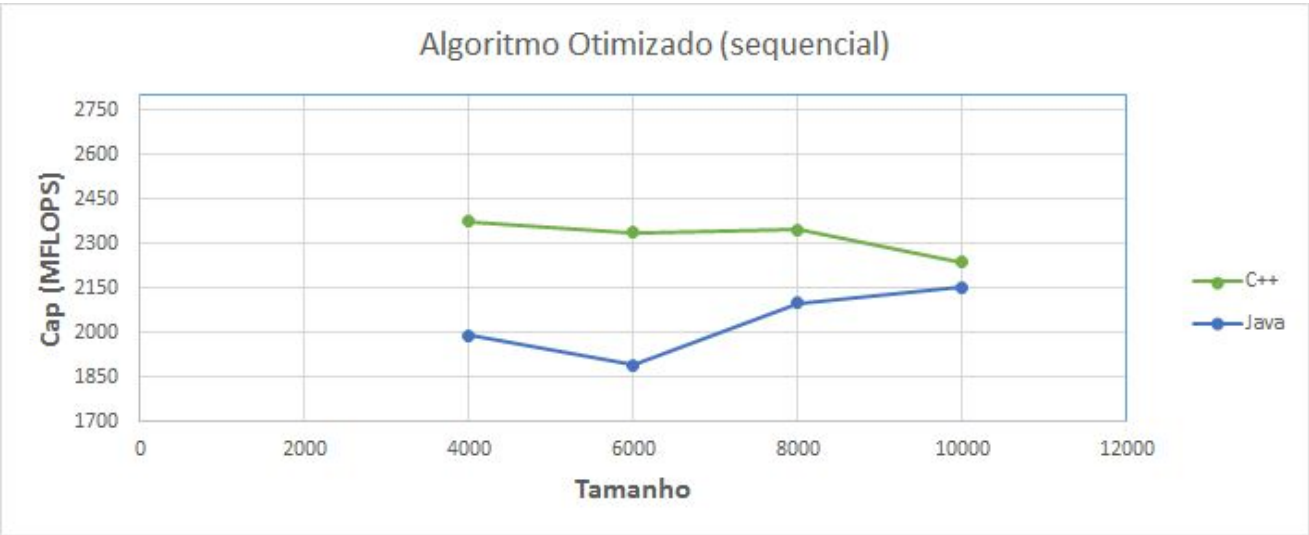
Execução sequencial

- **Algoritmo Básico**



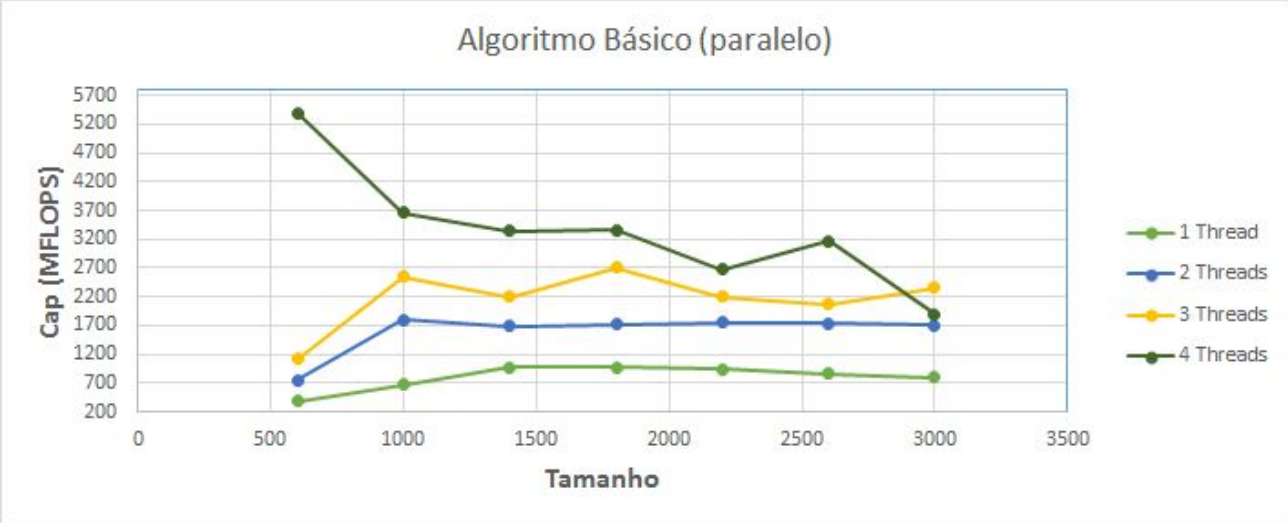
- **Algoritmo Otimizado**



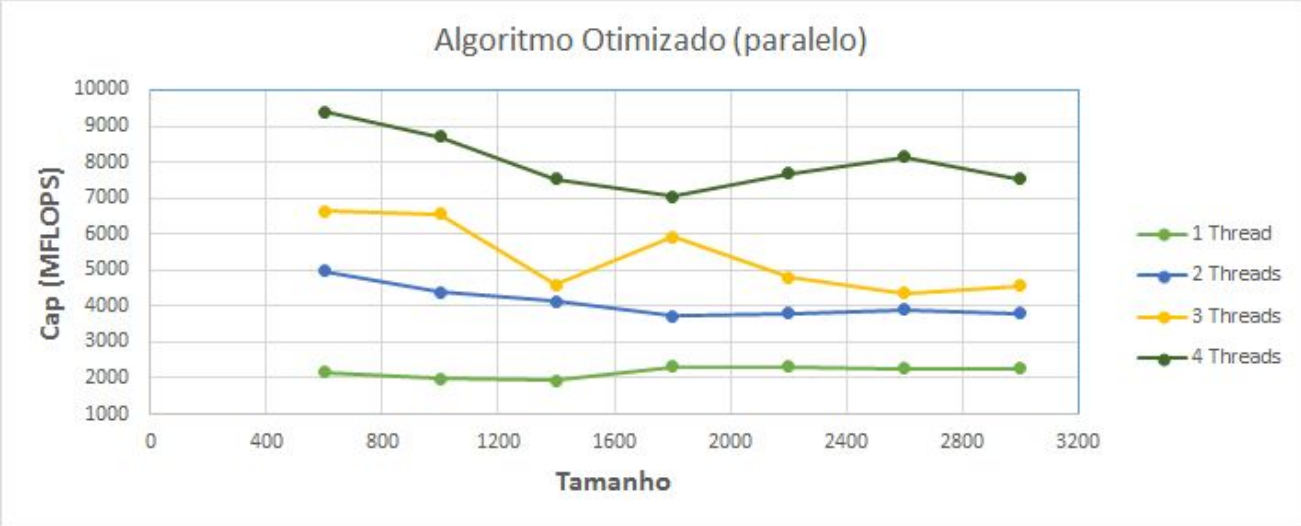


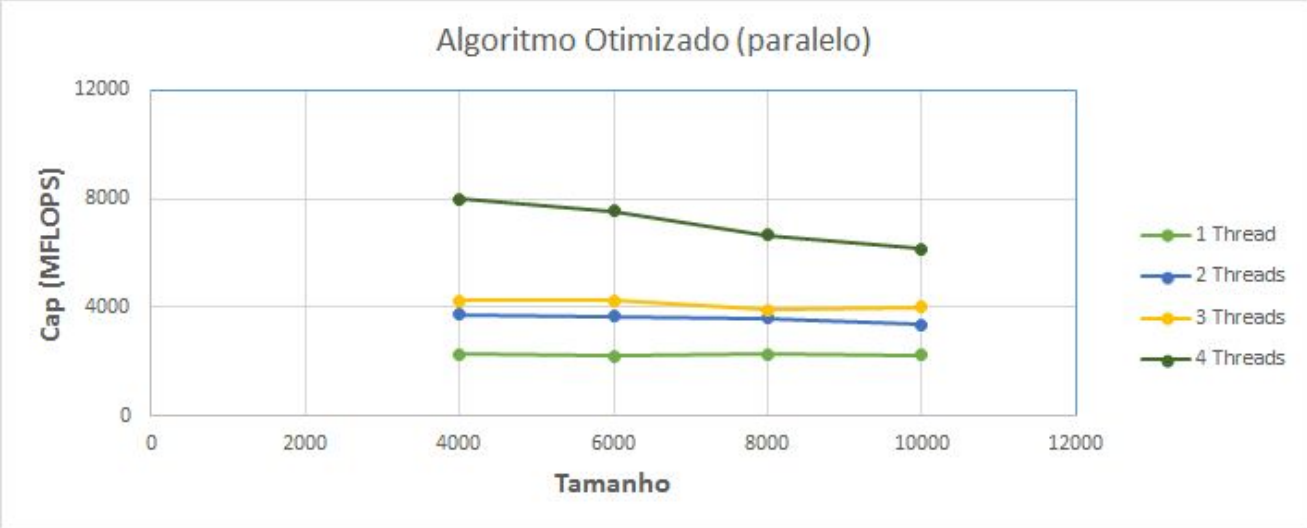
Execução em paralelo (com OpenMP)

- **Algoritmo Básico**



- **Algoritmo Otimizado**





Análise de Resultados

- **Execução sequencial:**

Em termos de desempenho, existe uma grande diferença óbvia entre as linguagens C++ e Java. Comparando os algoritmos, o otimizado demostra capacidades superiores notáveis em relação ao básico, tendo capacidades 3 vezes superiores na linguagem C++ e entre 4 a 5 vezes superiores em Java.
Ambos os algoritmos têm uma tendência de perda de desempenho com o aumento do tamanho da matriz a partir das 1800 unidades, excepto na linguagem Java que mostra um crescimento desde as 6000 unidades no algoritmo otimizado.

- **Execução paralela:**

Assim como a execução sequencial, o algoritmo otimizado demostra um desempenho muito superior ao algoritmo básico.
No algoritmo básico, utilizar 1 a 3 *threads* não existe uma grande discrepância no aumento das capacidades, embora quantas mais usar, maior a eficácia. O desempenho tem tendencia a piorar quanto maior o tamanho das matrizes, no entanto com 4 *threads* é mais acentuado, chegando até a ser pior do que usar 3 *threads* a partir das 3000 unidades.
Já no algoritmo otimizado, verifica-se que quanto mais *threads*, melhor o desempenho. Porém tendem a piorar menos acentuadamente do que o algoritmo anterior, chegando a estabelecer a sua capacidade a partir das 2600 unidades nos casos com 1, 2 e 3 *threads*.

- **Sequencial vs paralelo:**

A versão paralela dos algoritmos é claramente mais eficiente que a sua versão sequencial, sendo a única excepção utilizar apenas 1 *thread*, cujos resultados são bastante semelhantes ao algoritmo sequencial.

Observações

- Na execução para valores superiores a 4000 unidades em Java, é necessária a *flag* "*-Xmx2048m*" para alocação inicial de memória disponível (*heap space*). É uma das desvantagens de por definição, aplicações Java não alocarem memória dinamicamente em *runtime*;
- Anexado a este documento está o código-fonte do programa utilizado para este estudo. Para execução dos testes apenas é necessário correr os *scripts* encontrados em cada uma das pastas da linguagem a testar. Estes *scripts* compilam o programa com otimizações e correm os testes (com os devidos incrementos, algoritmo, número de *threads* em caso de paralelismo e reserva de memória no caso de Java);
- Anexado a este documento estão as folhas de cálculo com todos os dados relativos aos gráficos apresentados.

Conclusão

Analisando os resultados podemos concluir que existem possibilidades de melhoria do tempo de execução de programas utilizando o paralelismo e tendo em conta o funcionamento do acesso à memória cache. Através da modificação destes acessos, os aumentos de performance triplicaram entre as duas versões do algoritmo. Já com a divisão do trabalho a realizar por várias *threads*, houve também ganhos na performance bastante perceptíveis na versão otimizada do algoritmo.
Durante o desenvolvimento do projeto, pude também adquirir uma maior familiaridade sobre os conceitos de paralelismo computacional e em especial, com a API OpenMP.