Relatório de Experiência Laboratorial

Universidade do Porto

Faculdade de Engenharia

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Unidade Curricular de Computação Paralela

João Manuel Ferreira Trindade – 201109221 – ei11118@fe.up.pt

Sérgio Manuel Soares Esteves – 201100784 – ei11162@fe.up.pt

7 de Abril de 2015

Conteúdo

Introdução

Neste projeto estudamos a influencia que vários aspectos da programação têm sobre o desempenho computacional. Em primeiro foi analisado o impacto da gestão de memória cache, e como é possivel tirar partido das estratégias de gestão de cache já implementadas pelas linguagens com vista a minimizar os tempos de execução de operações computacionais complexas. De seguida estudamos as performances de várias linguagens de programação na solução de um mesmo problema. E por ultimo analizamos os "ganhos computacionais" da introdução de tecnicas de paralelização.

Para examinar o efeito utilizou-se problema da Multiplicação de matrizes.

Objectivos

Este projecto teve como objectivos principais:

- Compreensão das vantagens da computação paralela
- Compreensão dos diferentes tipos de gestão de memória (*row-major order* e *column-major order*)
- Analise do impacto da gestão de memória cache
- Aplicação da *Lei de Amdahl* num caso prático para o cálculo de SpeedUp
- Aplicação de mecanismos do OpenMP para a paralelização de "ciclos-for"

Descrição do Problema

Com vista a analisar o impacto da gestão de memórias e da paralelização na performance foi utilizado o problema de multiplicação de matrizes pois tem um custo computacional suficientemente elevado e é de fácil paralelização.
A multiplicação de matrizes é um problema matemático que consiste na multiplicação de uma matriz com **C1** colunas e **L1** linhas por uma outra matriz com **C2** colunas e **L2** linhas, originando uma matriz resultado com **L1** linhas e **C2** colunas. Assim, podemos desde já identificar uma restrição desta operação sobre matrizes: apenas é possível multiplicar a matriz A pela matriz B se o **número de colunas da matriz A for igual ao número de linhas da matriz B**.

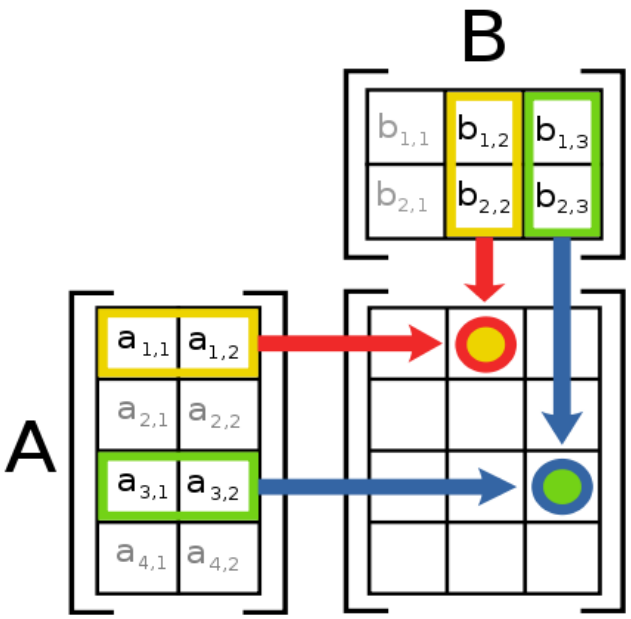
Explicação dos Algoritmos

Para multiplicar duas matrizes é necessário dividir o problema num sub problema já conhecido. Assim, sabendo que a multiplicação de matrizes é uma generalização do problema matemático do produto escalar de vetores, aplicamos a resolução deste problema sobre cada linha da matriz A e sobre cada coluna da matriz B para obter uma célula da matriz resultado.

O produto escalar de dois vetores consiste na multiplicação de cada valor do primeiro vetor com o respetivo valor na mesma posição do segundo vetor e por fim fazemos a soma de cada uma das multiplicações parciais obtendo assim um único valor.

$$\begin{bmatrix} A_x & A_y & A_z \end{bmatrix} \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} = A_x B_x + A_y B_y + A_z B_z = \vec{A} \cdot \vec{B}$$

Aplicando esta resolução para cada linha da matriz A com a respetiva coluna da matriz B obtemos a matriz resultado com os valores pretendidos.



Algoritmo Base

No algoritmo convencional, necessitamos de percorrer a matriz A linha a linha, no pseudo-código representado pela variável I, percorrendo os seus valores coluna a coluna, representado pela variável K, multiplicando este valor(I,K) pelo valor correspondente na matriz B que se encontra na mesma linha que as colunas da matriz A, ou seja, a mesma variável K, percorrendo depois as colunas, representado pela variável J.

No final de cada iteração em K, ou seja, quando cada linha e cada coluna das matrizes A e B respectivamente forem percorridas somos capazes de chegar ao resultado de uma célula da matriz resultado que será posicionado utilizando as variáveis I (Linha de A) e J (Coluna d B).

Código em Java:

```
for(int i = 0 ; i < lin ; i++) {
   for(int j = 0 ; j < col ; j++) {
      temp = 0;
      for (int k = 0 ; k < lin; k++) {
         temp += matrix_A[i][k] * matrix_B[k][j];
      }
      matrix_C[i][j] = temp;
   }
}
```

Algoritmo Alternativo

Analisamos agora o segundo algoritmo implementado, ao contrário do que acontece no algoritmo base, neste, a cada iteração não é calculado o valor de um elemento de C, mas sim os valores parciais de vários elementos de C.

A cada iteração, não obtemos desde logo um valor que seria o resultado de uma célula em C, obtemoss porém resultados parciais de uma linha de C, e conseguimos em comparação com o algoritmo anterior, percorrer apenas uma vez a matriz A.

Código em Java:

```
for(int i = 0 ; i < lin ; i++) {
   for(int k = 0 ; k < col ; k++) {
      for (int j = 0 ; j < lin; j++) {
         matrix_C[i][j] = matrix_C[i][j] + matrix_A[i][k] * matrix_B[k][j];
      }
   }
}
```

Metodologia

Ambos os algoritmos foram desenvolvidos nas linguagens: C, Java, Pascal e Fortran (95). Algumas notas importantes sobre a compilação das diferentes versões:

- A versão na linguagem C foi compilada na versão mais recente do MSVC, para plataformas de 64 bits para ser possível alocar toda a memória necessária para matrizes de grandes dimensões.
- A versão na linguagem Fortran foi compilada utilizando o Intel Parallel Studio XE Professional Edition e também a versão gratuita do compilador desenvolvido pela SilverFrost (FTN95 – Plato3)

Todas as versões foram executadas para matrizes de teste de acordo com a especificação. Com excepção dos testes em Fortran do compilador SilverFrost, todos os testes foram executados num computador com as seguintes especificações:
Intel 4790 3.6 GHz 8MB Cache & 16GB RAM DDR3 1600MHz

Resultados

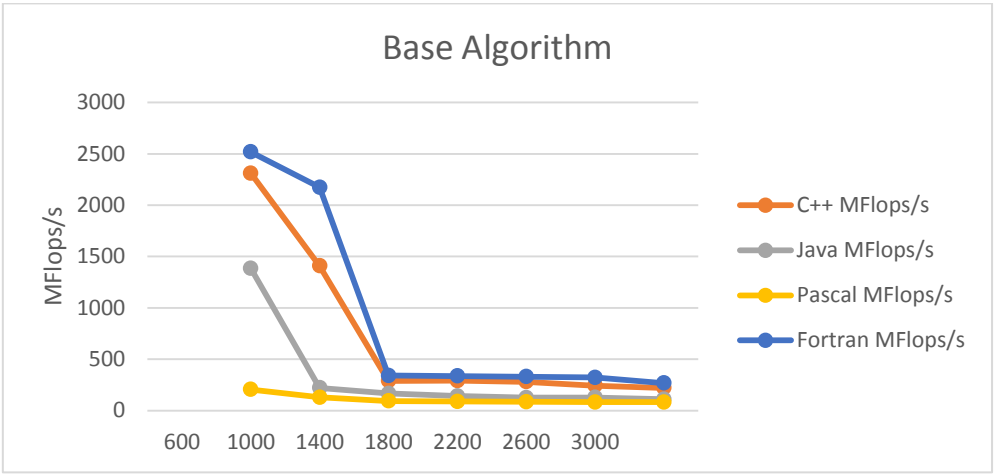


Gráfico 1. Capacidade das 4 linguagens para o Algoritmo Base para diferentes tamanhos de matrizes

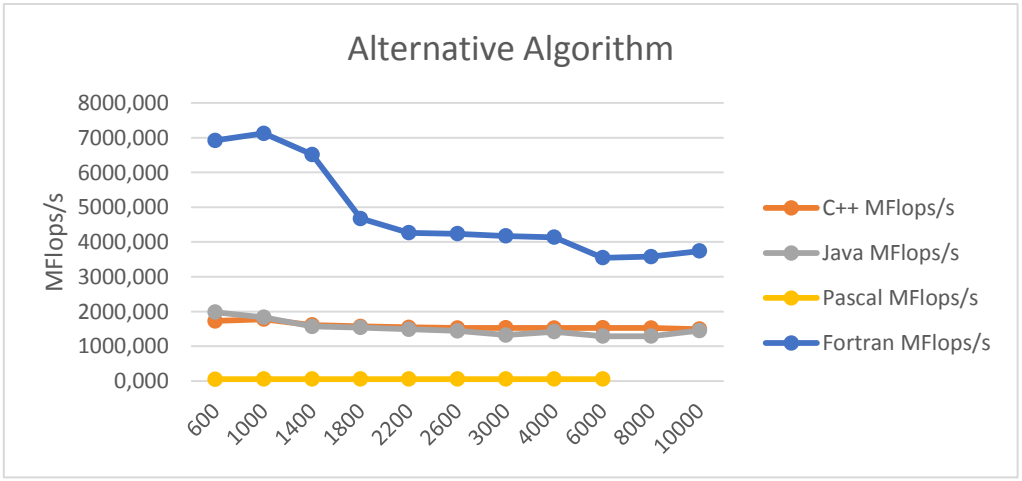


Gráfico 2. Capacidade das 4 linguagens para o Algoritmo Alternativo para diferentes tamanhos de matrizes

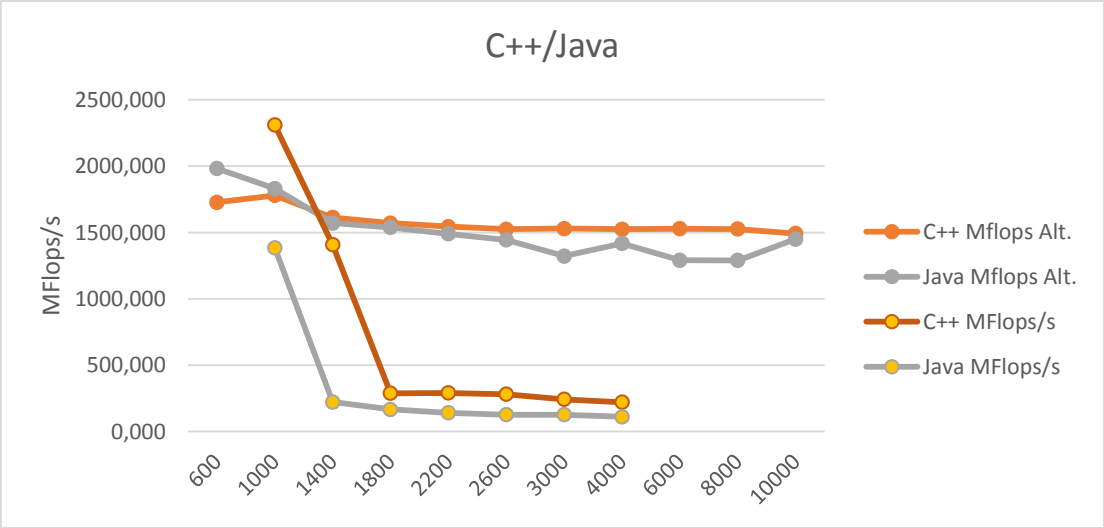


Gráfico 3. C & Java: Algoritmo Base vs. Algoritmo Alternativo

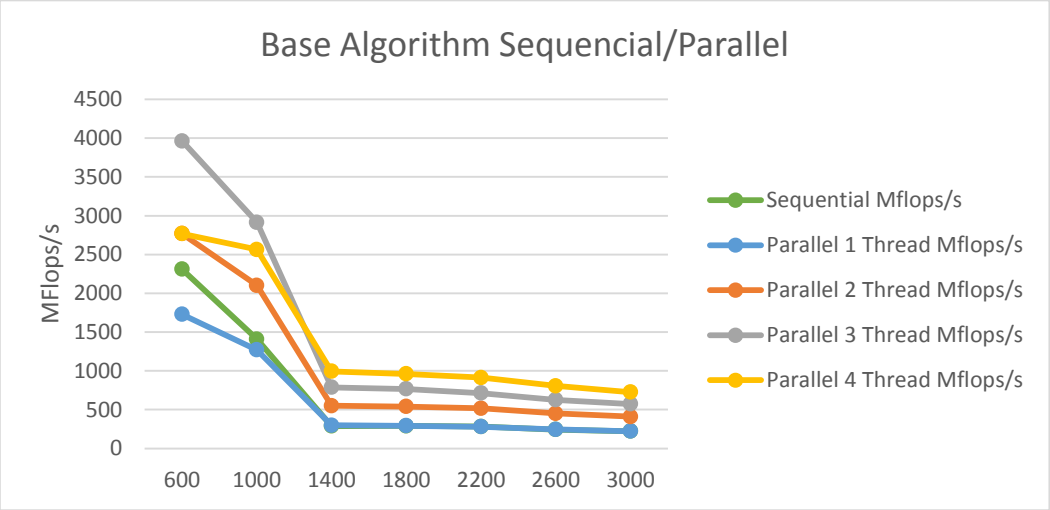


Gráfico 4. Algoritmo Base. Capacidade em Paralelo vs. Sequencial

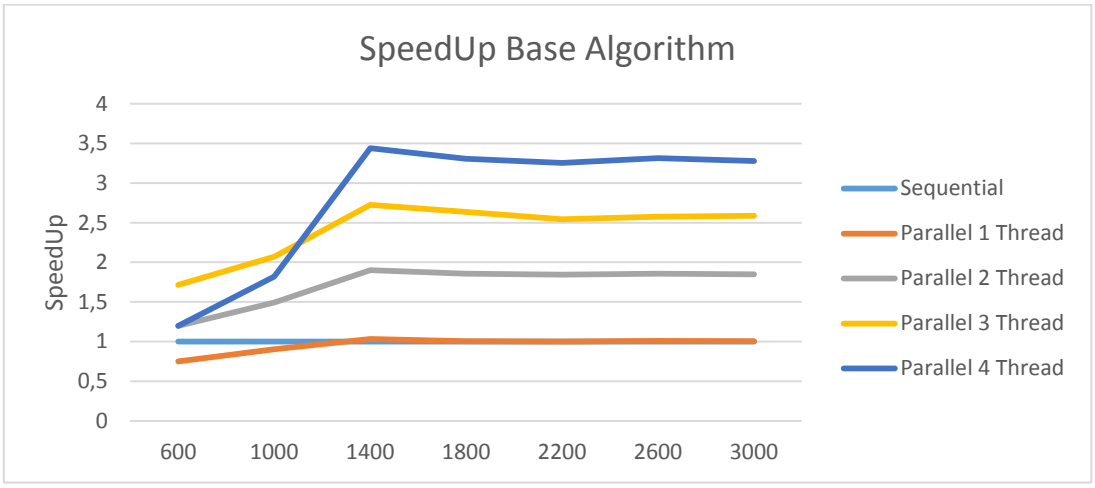


Gráfico 5. SpeedUp Algoritmo Base

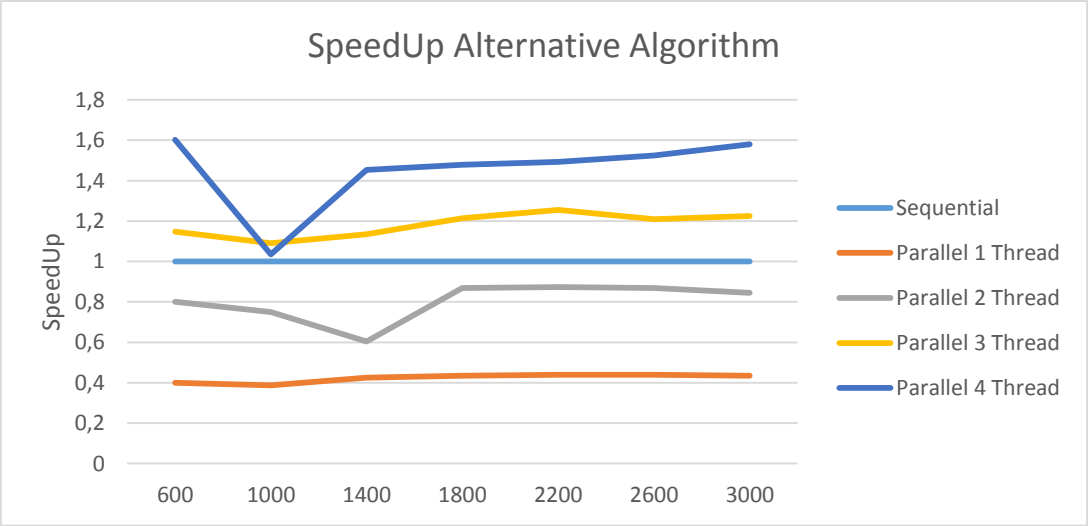


Gráfico 6. SpeedUp Algoritmo Alternativo

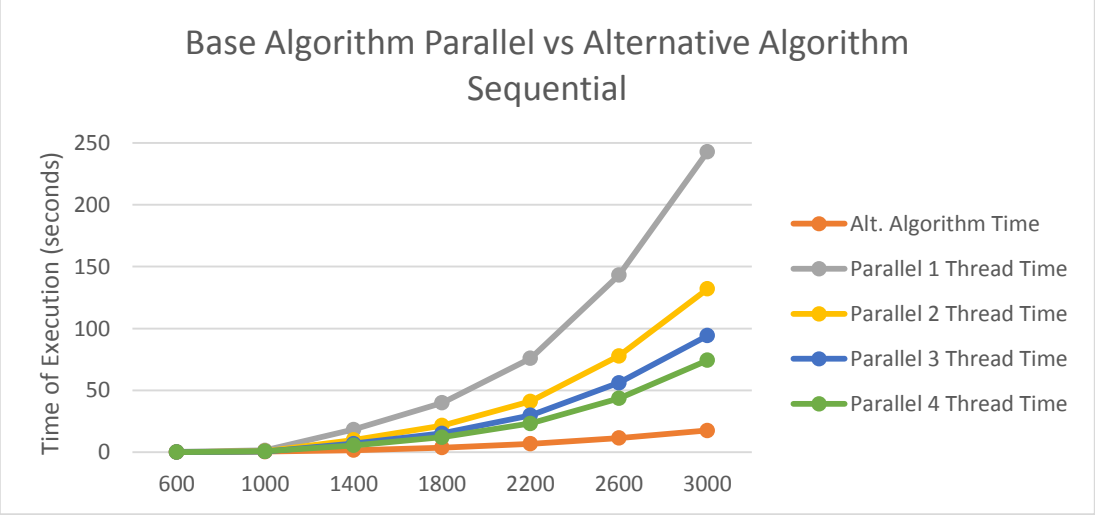


Gráfico 7. Algoritmo Base em Paralelo vs. Algoritmo Alternativo Sequencial

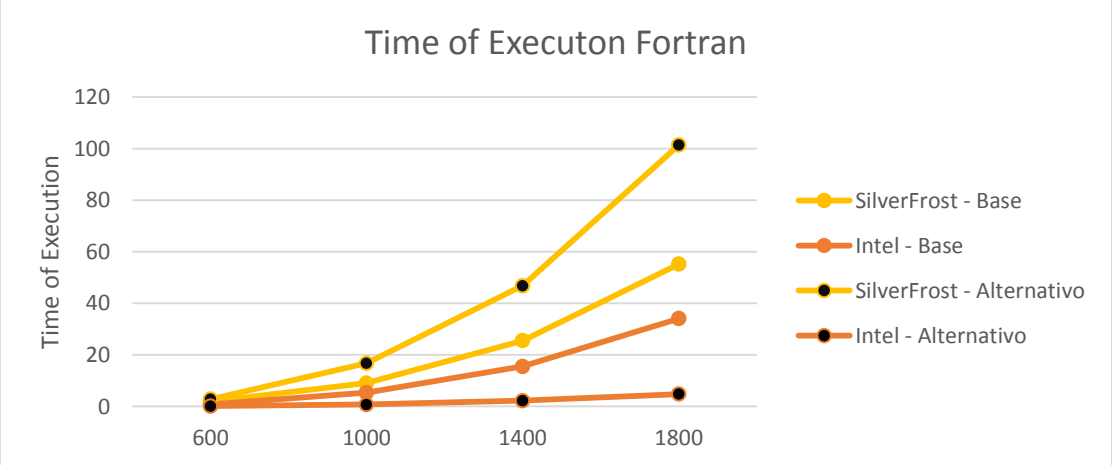


Gráfico 8. Caso Fortran – Diferentes comportamentos em diferentes compiladores

Análises

Analisando os Gráficos 1 e 2, concluímos que em todas as linguagens a versão alternativa do algoritmo (linha vezes linha) superou muito pronunciadamente a versão base do algoritmo. Na linguagem C, por exemplo passamos de uma capacidade de 221,6 MFLOPS/s na versão base para 1492,1 MFLOPS/S, um aumento de 675%, como é possível analisar no Gráfico 3.

Um resultado que não foi ao encontro das expectativas foi na linguagem Fortran, o grupo esperava que, dada a gestão de memória "column-major" que esta linguagem tradicionalmente implementa, seria de esperar um melhor desempenho no algoritmo base, contrariando as outras linguagens, todas elas "row-major".

Dos Gráficos 4 e 5, podemos concluir os grandes ganhos tirados da implementação de mecanismos de paralelização no algoritmo base, o speedup para 2/3/4 threads no caso maior foi respectivamente de 1.84, 2.58 e 3.27 o que demostra os notáveis ganhos destas técnicas de paralelização.
Já quanto á paralelização do algoritmo alternativo, houve também melhorias, mas estas não foram tão pronunciadas, como se mostra no Gráfico 6, os speedups para 2/3/4 threads foram, no caso maior, respectivamente de 0.84, 1.22, e 1.57, o que mostra que, apenas com dois cores, neste algoritmo temos uma performance inferior ao que teríamos sem paralelização, isto deve-se possivelmente a *overheads* introduzidos pelos mecanismos de paralelização.

No Gráfico 7 podemos observar um facto notável, nesta comparação entre o algoritmo base paralelizado, e o algoritmo alternativo na sua forma sequencial observa-se que, o algoritmo alternativo, mesmo na sua forma sequencial consegue ser bastante superior ao base paralelizado, mesmo quando executado em 4 threads. De facto, no caso maior, mesmo quando executado em 4 threads, o algoritmo base paralelizado foi cerca de 4 vezes mais lento que o algoritmo alternativo executado sequencialmente.

O Gráfico 8 representa a investigação que fizemos aos resultados inesperados que obtivemos nos Gráficos 1 e 2. Vemos aqui que no compilador Intel a versão alternativa é mais rápida que a versão base (tal como acontece nas restantes linguagens, e aconteceu nos gráficos 1 e 2), mas que no compilador da SilverFrost, o algoritmo base é mais rápido a executar que o algoritmo alternativo.

Conclusões

As conclusões retiradas deste projecto foram:

- Diferentes linguagens têm diferentes capacidades – overheads das próprias linguagens
- Certas linguagens estão mais habilitadas para determinadas operações, no caso tratado o Fortran foi claramente mais rápido.
- Utilizando o OpenMP conseguimos muito facilmente SpeedUps de 3.27
- É preferível ter um bom algoritmo á partida, do que paralelizar um algoritmo ineficiente.
- Extra. Caso Fortran: diferentes compiladores aplicam diferentes estratégias de gestão de memória. O Compilador da Intel aplica uma estratégica "row-major" ao contrario do que tradicionalmente acontece nesta linguagem

Bibliografia

- Apontamentos das Aulas
- “Using OpenMP”, Barbara Chapman
- https://software.intel.com/en-us/forums/topic/271228