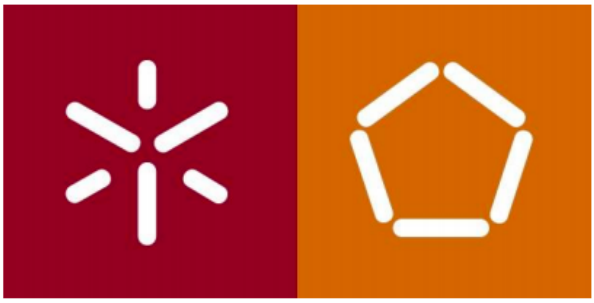

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Algoritmos Paralelos

Trabalho Prático 3

Eduardo Lourenço da Conceição (A83870)
Rui Nuno Borges Cruz Oliveira (A83610)

17/06/2021
Braga

Índice

1 Introdução

Para o trabalho prático final da cadeira de Algoritmos Paralelos foi-nos proposta a implementação de dois algoritmos, à nossa escolha de entre uma gama de algoritmos de diferentes áreas, utilizando técnicas de paralelismo aprendidas no perfil, e otimizar este algoritmo de modo a obter a melhor *performance* possível.

2 Algoritmos Escolhidos

Dos vários algoritmos que tínhamos à escolha, foi-nos proposta a ideia de considerar um que fosse "bom" e um que fosse "mau". Mas como definimos o que é um algoritmo mau e um bom, no contexto de algoritmos paralelos?

Para nós avaliarmos a qualidade de um algoritmo paralelo e como é que ele poderá escalar, temos de analisar o padrão de acesso aos dados do domínio que o mesmo tem. Um algoritmo que maximize a reutilização de dados da forma mais eficiente possível terá o melhor comportamento num contexto paralelo quando utilizamos um modelo de memória multi-nível. Esta forma de analisar algoritmos surge devido ao facto que a grande parte das arquiteturas de sistemas de computação modernos utilizam uma hierarquia *multi-level* de memória, com os vários níveis apresentando latências distintas e não negligenciáveis. Assim, uma boa forma de avaliar um algoritmo é pela quantidade de dados do domínio que utiliza por ponto do domínio, bem como pela quantidade de operações que se realizam sobre ditos elementos. Como tal, podemos avaliar a qualidade de um algoritmo neste aspeto olhando para a seguinte imagem, disponibilizada nos *slides* teóricos da cadeira:

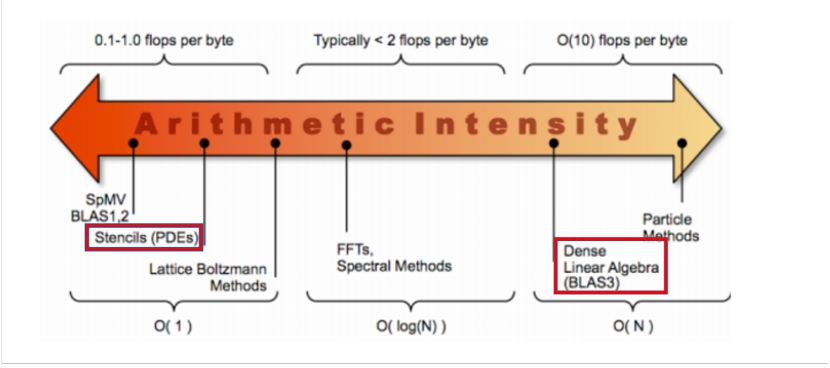

Figure 1: Eixo uni-dimensional da Intensidade Aritmética

As duas famílias de algoritmos destacadas são as que decidimos explorar, sendo que algoritmos à esquerda não irão escalar tão bem pois são computacionalmente pouco intensos, e os à direita apresentam complexidade computacional maior e deverão, quando bem paralelizados, escalar melhor.

Os *stencils* são algoritmos com um comportamento que escala de forma relativamente má, uma vez que a quantidade de dados que utilizamos por ponto do domínio é fixa, sendo, no nosso caso, 5 elementos, o que torna um algoritmo que é visto classicamente como $O(N)$ num algoritmo $O(1)$ neste ponto de vista novo e, como tal, não é esperado que escale muito bem.

Por outro lado, a família de algoritmos *BLAS* (*Basic Linear Algebra Subprograms* é consideravelmente melhor. Em particular, os algoritmos *GeMM* (*General Matrix Multiply*) são excelentes em termos de utilização de dados, uma vez que, para uma multiplicação $C = A \times B$, para cada elemento da matriz C, vamos utilizar uma coluna da matriz B e uma linha da A, sendo que o algoritmo terá de complexidade $O(N^{\frac{3}{2}})$. Um algoritmo desta classe será mais complexo de paralelizar devidamente, mas também apresentará melhor comportamento quando paralelizado (pois a sua versão sequencial já é, por si, boa).

Deste modo, ficamos com dois algoritmos nos lados opostos do espectro, como pedido.

3 Implementação

3.1 Domínio dos Dados

Para ambos os problemas o domínio dos dados é constituído por matriz de *doubles*, sob a forma de *arrays* estáticos bidimensionais (*double A[][]*). Estas estruturas estão forçosamente alinhadas em memória em blocos de 32 *bytes*, de modo a melhorar a estruturação das linhas de *cache* e, como tal, a *performance* do programa. A utilização de estruturas regulares como estas (em oposição a estruturas baseadas em apontadores) é importante pois terá um impacto positivo na *performance* no sentido que o acesso às mesmas tem o potencial de aproveitar ao máximo localidade espacial (e, por consequência dos algoritmos, temporal).

3.2 *GeMM*

Para implementar um algoritmo de multiplicação de matrizes, utilizamos a variante *axpy*, com o ciclo por ordem *ikj*. Esta é a variante que tem o melhor comportamento em termos de *performance* mesmo sem paralelismo, pelo que se torna naquele que escalará de pior forma. No entanto, é um algoritmo bastante bem comportado em termos de utilização de memória.

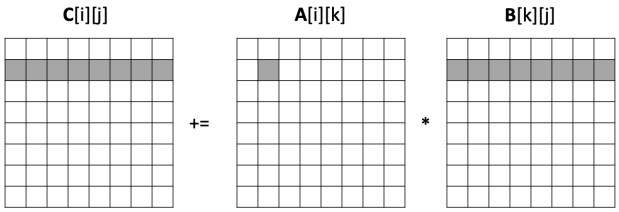

Figure 2: Visualização dos elementos utilizados para as iterações do ciclo j

Aqui, poderemos guardar o valor de $A(i,k)$ em registo, uma vez que o mesmo valor vai ser reutilizado para todos os cálculos do ciclo j. Por outro lado, iremos incorrer em $\frac{N}{cache\_line\_size/sizeof(double)}$ *misses* para percorrer as matrizes C e B. Como podemos guardar os valores de A em registo, poderemos fazer *tilling* em registo com a matriz A, guardando alguns valores consecutivos em registo, possivelmente A(i:i+1,j:j+1), de modo a criar uma *tile* quadrada. No entanto, esta abordagem não provou ter efeitos positivos na *performance*, pelo que decidimos não avançar com ela.

No entanto, de modo a podermos melhorar a utilização de memória, implementamos *cache tiling*, utilizando tamanho de bloco igual a 8, uma vez que é o número de *doubles* que uma linha de *cache* de 64 *bytes* transporta. Deste modo, esperamos poder ter reutilização ideal de cada linha de *cache*, guardando toda a informação de um bloco na *cache L1*, sendo este bloco quadrado, que será o tipo de dimensionamento ideal. Cada bloco corresponderá a uma sub-matriz, pelo que utilizar *tiling* corresponde na prática a fazer várias iterações do mesmo algoritmo de multiplicação de matrizes sobre várias matrizes mais pequenas, que podem ser guardadas, no nosso caso, na *cache L1*. a paralelização é feita com *OpenMP* (tanto para este algoritmo como para o *stencil*), e é feita ao nível dos blocos, paralelizando o ciclo mais externo (ii). O aspeto dos ciclos *for* pode ser observado no seguinte excerto de código:

```
for(ii = 0; ii < N; ii += BLOCK_SIZE)
    for(kk = 0; kk < N; kk += BLOCK_SIZE)
        for(jj = 0; jj < N; jj += BLOCK_SIZE)
            for(i = ii; i < ii + BLOCK_SIZE; i++)
                for(k = kk; k < kk + BLOCK_SIZE; k++)
                    for(j = jj; j < jj + BLOCK_SIZE; j++)
                        ...
```

O ciclo mais interior, o j, uma vez que tem *stride* unitário e padrões de acesso às matrizes C e B regulares pode ser vetorizado. Compilar o programa com *-march=native* garante isso no nodo do *cluster* que utilizamos.

O escalonamento que provou ter melhores resultados para este caso de estudo foi escalonamento dinâmico, o que nós achamos estranho, pois a carga de trabalho por *thread* está equilibrada, e esperar-se-ia que escalonamento estático tivesse melhores efeitos, mas isto não aconteceu, o que não conseguimos explicar.

3.3 *Stencil* de 5 Pontos

O *stencil* de 5 pontos é um padrão relativamente simples de implementar, sendo que o algoritmo apenas envolve percorrer uma matriz bidimensional e atualizar cada ponto do domínio utilizando os valores dos seus vizinhos da seguinte forma:

$$A(i,j) = 0.2 * (A(i,j) + A(i-1,j) + A(i-1,j) + A(i,j-1) + A(i,j+1))$$

Em primeira vista, podemos observar que atualizações de elementos consecutivos da matriz iriam levar a *data races* e geraria informação inconsistente, se não houver alguma mecanismo de controlo de concorrência. Uma forma de ultrapassar este problema será varrer a matriz utilizando uma ordenação diferente, nomeadamente a **ordenação *red-black*** que utilizamos no segundo trabalho da cadeira para implementar o algoritmo de *Gauss-Seidel* e de *S.O.R.*. Esta ordenação obriga a dois varrimentos diferentes, percorrendo as casas vermelhas e depois as outras da matriz, como podemos observar na seguinte figura:

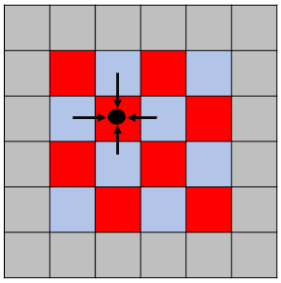

Figure 3: Exemplificação do varrimento *red-black* (não aplicamos os cálculos às bordas, a cinzento)

Como podemos ver, o ponto destacado a preto na figura 3 utiliza valores que serão atualizados apenas no próximo varrimento, pelo que podemos fazer as atualizações em paralelo em cada varrimento sem problemas. No entanto, esta ordenação tem dois problemas: em primeiro lugar, a reutilização de memória é muito mais fraca, pois não utilizamos para escrita metade dos elementos de uma linha de *cache* (o que nos leva a utilizar para este propósito apenas 4 dos 8 *doubles* que uma linha de *cache* traz) e vamos ter de carregar a mesma linha de *cache* duas vezes para atualização (uma em cada um dos varrimentos) e, em segundo lugar, como obriga à utilização de um *stride* não unitário no ciclo mais interior, impede vetorização. No entanto, por outro lado, permite-nos paralelizar o algoritmo sem aplicar nenhuma forma de controlo de concorrência e sem ter de utilizar duas matrizes, como a versão original *naive* utiliza.

Tal como aconteceu com o *GeMM*, para melhorar a *performance* do algoritmo, experimentamos implementar um mecanismo de *tiling* com blocos quadrados de tamanho 8×8. Estes blocos não são perfeitos, pois os elementos nos limites do bloco terão de ir buscar à memória central elementos de blocos adjacentes para fazerem os cálculos, mas provou ter efeitos benéficos na *performance* do programa. A paralelização é feita

ao nível dos blocos, o que permitiria utilizar apenas um varrimento da matriz ao invés de dois da ordenação *red-black*, mas são os acessos a outros blocos por parte dos elementos no limite que impossibilita isto.

Ao olhar para a seguinte imagem:

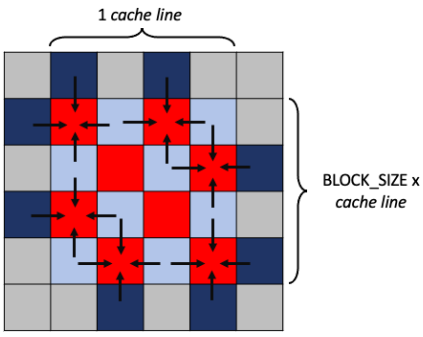

Figure 4: Dependências nos limites

Podemos observar que as dependências nos limites vão adicionar $\frac{BLOCK\_SIZE}{2}$ *cache misses* fora do bloco para as colunas e 1 *cache miss* para as linhas, incorrendo num total de:

$$\frac{N^2}{BLOCK\_SIZE^2} * (2 * BLOCK\_SIZE + 2)$$

cache misses por varrimento, assumindo uma matriz quadrada $N \times N$ e blocos quadrados de tamanho $BLOCK\_SIZE \times BLOCK\_SIZE$, duplicando esse número tendo em conta os dois varrimentos por iteração do algoritmo.

O algoritmo não envolve apenas um varrimento total da matriz, envolve N varrimentos consecutivos, com $N = 1000$ no nosso caso de estudo. O ciclo de aplicação dos varrimentos não pode ser paralelizado, como é óbvio.

O escalonamento utilizado aqui foi estático, que, ao contrário do *GeMM*, provou ser benéfico, e faz sentido visto que a carga de cada *thread* é equilibrada.

4 Análise de *Performance*

4.1 Condições e Metodologia de Teste

Os testes foram executados no *cluster SeARCH*[2], no nodo r641, como pedido. Sendo que o nodo tem 16 *cores* físicos, requisitamos, sempre que possível, 32 processos, de modo a utilizar o máximo de *cores* virtuais. Por consequência, utilizamos também sempre 32 *threads* e forçamos o impedimento de migração de *threads* entre CPUs (com *OMP_PROC_BIND=true*). Por algum motivo que não conseguimos explicar, ainda existiu algum nível de migração entre CPUs, como iremos mostrar mais à frente.

Os resultados que vamos apresentar serão uma mediana e uma média de 5 medições. As medições foram feitas sempre que possível de noite e em alturas que houvesse uma menor carga de trabalho sobre o *cluster*, de modo a termos o mínimo de interferência possível, visto que em alturas de elevado tráfego no *cluster* os resultados apresentavam valores muito mais erráticos. Mais ainda, a *cache* é sempre limpa antes de fazermos os cálculos importantes, utilizando uma pequena função fornecida no código de uma das fichas de Arquiteturas Avançadas, pelo professor André Pereira.

Ambos programas foram compilados com as *flags* "*-march=native -g -O3 -fopenmp*".

4.2 Tempos de Execução

4.2.1 *GeMM*

Os tempos de execução que nós obtivemos para o *GeMM* foram os seguintes:

	Média	Mediana
T_{exec} (s)	1.260089	1.257453

Table 1: Tempo de Execução do *GeMM*

Como podemos ver, os resultados estão dentro do esperado, sendo que o intervalo aceitável encontrava-se entre 1 a 2 segundos. Este resultado seria de prever quando nós aplicamos as várias otimizações, e tendo em conta que os recursos disponíveis foram bem utilizados, como iremos ver na secção seguinte. Com tal, não acreditamos que este resultado seja anormal e está dentro do previsto.

4.2.2 *Stencil* de 5 Pontos

Os resultados que obtivemos do *stencil* foram os seguintes:

	Média	Mediana
T_{exec} (s)	0.620949	0.621759

Table 2: Tempo de Execução do *Stencil*

Estes resultados foram bastante melhores do que nós esperávamos. Sendo que o esperado seria estes valores estarem entre 2 e 4 segundos, estão muito melhores do que o previsto. Acreditamos que a utilização de *tiling* para mitigar a fraca utilização da *cache* por parte do varrimento *red-black* terá sido o fator mais importante nisto. No entanto, não será o único, como veremos na secção a seguir.

4.3 *Profiling*

Para além do tempo de execução, um *profiling* mais detalhado, com a ajuda do *perf stat* ajudará na compreensão do comportamento dos algoritmos. Os valores apresentados aqui serão relativos à execução que produziu o resultado da mediana.

Olhando primeiro para a multiplicação de matrizes:

T_{exec}	1.257453 s
CPUs Utilizados	27.373
Migrações de CPU	2
Instruções	52525642881 *instructions*
Ciclos	118987777999 *clock-cycles*
IPC	0.44 *insts/cycle*

Table 3: Perfil de Execução do *GeMM*

Aqui podemos ver alguns aspetos positivos do algoritmo, e alguns negativos. Um aspeto que justifica bastante o tempo de execução razoavelmente bom é o número de CPUs utilizados. Sendo cada nodo do ramo r641 uma máquina de 16 *cores* físicos e 32 virtuais, vemos que estamos a utilizar uma grande parte desses *cores* virtuais ao atribuir 32 *threads* com afinidade ao CPU, sendo que utilizamos aproximadamente 85.5% dos recursos computacionais disponíveis, o que é bastante bom, e é o fator mais importante a retirar destas métricas, uma vez que, como vimos em Arquiteturas Avançadas, o algoritmo é *compute bound*, de acordo com o a avaliação do seu *Roofline Graph*[1]. Por outro lado, o IPC é consideravelmente baixo, e isso pode acontecer devido à latência de acesso à memória, e ao *stalling* que surge daqui, se bem que esta métrica não era suportada pelo *perf* no nodo r641, pelo que não a conseguimos avaliar.

Avaliando agora o *stencil*:

T_{exec}	0.621759 s
CPUs Utilizados	27.924
Migrações de CPU	0
Instruções	75367873792 *instructions*
Ciclos	58067122106 *clock-cycles*
IPC	1.30 *insts/cycle*

Table 4: Perfil de Execução do *Stencil*

Podemos ver que o comportamento é muito semelhante ao *GeMM* em termos de utilização do CPU, mas o IPC deste é muito diferente, sendo que a execução foi superescalar, tendo um $IPC > 1$. Este algoritmo difere do *GeMM* no sentido que é mais intenso computacionalmente, uma vez que são feitas mais operações por elemento do domínio (se bem que a diferença não é muito grande, mas não pode ser ignorada). Mais ainda, vemos que as migrações entre CPUs neste caso são nulas, o que também terá algum impacto positivo na *performance*, pois não temos de nos preocupar com o tempo de mudança de contexto. Isto não é muito surpreendente quando vemos que a execução é feita com escalonamento estático, o que em princípio deveria minimizar este tipo de trocas de contexto.

Em geral, os dois algoritmos apresentaram comportamentos que esperávamos, em grande parte, com o aspeto mais fraco sendo o IPC no *GeMM*, que é o ponto mais inesperado dos dois, para nós.

5 Conclusão

Em conclusão, acreditamos ter feito um bom trabalho. Conseguimos implementar *kernels* bons para ambos os algoritmos, e conseguimos fazer uma boa análise teórica e prática de ambos, tendo a segunda várias métricas de perfil como base.

Os pontos mais fracos do nosso trabalho serão a simplicidade e fraca otimização do *kernel* do *GeMM*, que apresentava vários aspetos que podiam ter sido melhor explorados, e o facto que existem mais métricas interessantes que podíamos ter medido, nomeadamente as *misses* nos vários níveis de *cache*, que são aspetos importantes na avaliação do desempeho dos algoritmos e que não tocamos.

No entanto, estamos confiantes que produzimos um bom trabalho final da cadeira de Algoritmos Paralelos.

6 Referências

[1] David A. Patterson Samuel W. Williams Andrew Waterman. “Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures”. In: (2008).

[2] *SeARCH Cluster*. URL: `http://search6.di.uminho.pt/wordpress/`.