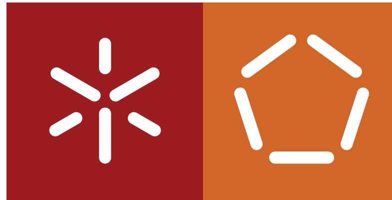
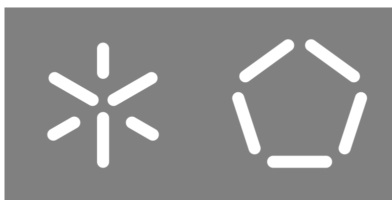


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Jorge Miguel Ferreira de Oliveira

Exploração de paralelismo em aplicações do tipo *stencil*

Janeiro 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Jorge Miguel Ferreira de Oliveira

Exploração de paralelismo em aplicações do tipo *stencil*

Dissertação de Mestrado
Mestrado em Engenharia Informática

Dissertação orientada por
João Luís Ferreira Sobral

Janeiro 2016

AGRADECIMENTOS

Existe um vasto grupo de pessoas que contribuíram para que esta dissertação se tenha tornado uma realidade.

Ao professor João Luís Ferreira Sobral agradeço todo o esforço e dedicação na orientação desta dissertação, obrigado por toda a disponibilidade durante a duração da mesma. Foi um enorme prazer poder aprender e trabalhar consigo durante todo este tempo.

Para os meus pais e irmãos vai o meu maior agradecimento, nada disto seria possível se não fosse o esforço feito por eles para que eu chegasse onde cheguei. Obrigado por todo o apoio e toda a disponibilidade para me ajudarem e suportarem sempre que precisei.

ABSTRACT

Nowadays stencil algorithms are one of the most commonly used in scientific and graphic computing. Many of the finite difference methods with regular structures can be formulated with a stencil code. Since they are heavy computing algorithms and in order to increase their performance a large knowledgement about the best design for them is needed.

In the past the computer industries aimed to increase the clock frequencies, with the problems that those increases brought they started to use new technologies to improve their computational capacities, for example, increasing the amount of cache memory and the increase in the number of cores per processor. It is important to use those resources in a better way in order to algorithms benefit from them.

In this dissertation an heat convection-diffusion simulation algorithm was studied using a Finite Volume Library. Spatial and temporal exploitation techniques were be applied to guarantee that the data cache is accessed in a straight way and is used in his most once loaded. A parallel algorithm to shared memory systems was also developed.

Initially the code was simplified resulting in a decrease in the quantity of executed instructions and reducing 1,2 times the execution time. Afterwards improvement in the spatial data accesses were introduced reducing the cache miss rates even further, resulting in a 3,8 times smaller execution time, in comparison with the previous version.

In a second phase, the algorithm was parallelized so that it uses the different cores available in a shared memory system obtaining an 8 times gain in a system with 16 cores. A version of the parallel code that benefits from temporal locality was also explored.

This dissertation shows the importance of a good data organisation in order to get an efficient parallelism exploration in this kind of stencil algorithms.

RESUMO

Algoritmos *stencil* são um dos tipos de algoritmos mais utilizados na computação científica e na computação gráfica nos dias de hoje. Muitos dos métodos de diferenças finitas com estruturas regulares podem ser formulados com um código *stencil*. São algoritmos bastante pesados em termos computacionais e, por isso mesmo, é cada vez mais necessário entender o melhor e mais eficiente *design* para os mesmos.

No passado a indústria dos processadores apostou no aumento da frequência do relógio, com os problemas inerentes do aumento dessa frequência começaram a apostar em novas tecnologias para melhorar a capacidade computacional dos mesmos, como por exemplo o aumento da quantidade de memória *cache* e o aumento do número de núcleos por processador. É importante utilizar esses recursos do melhor modo para que os algoritmos beneficiem dos mesmos.

Nesta dissertação foi estudado um algoritmo de simulação de convecção-difusão de calor que utiliza uma biblioteca de volumes finitos (*Finite Volume Library*). Foram aplicadas técnicas de exploração da localidade espacial e temporal de dados, garantindo que dados em *cache* são acedidos de modo contíguo e, uma vez carregados, são utilizados na sua totalidade. Foi ainda criada uma versão paralela do algoritmo para sistemas de memória partilhada.

Inicialmente foi simplificado o código resultando numa redução significativa na quantidade de instruções executadas e num ganho de 1,2 vezes no tempo de execução. Posteriormente, foram introduzidas melhorias na localidade espacial dos dados, reduzindo a quantidade de *misses* em *cache*, resultando num ganho de 3,8 vezes no tempo de execução em relação à versão anterior.

Numa segunda fase da dissertação foi paralelizado o código de modo a utilizar vários núcleos disponíveis num sistema de memória partilhada obtendo um ganho de 8 vezes num sistema com 16 núcleos. Foi ainda explorada uma versão do código paralelo que beneficia de localidade temporal.

Esta dissertação mostra a importância de uma boa organização dos dados para que seja possível a exploração eficiente de paralelismo nestes algoritmos do tipo *stencil*.

CONTEÚDO

Contents iii

i MATERIAL INTRODUTÓRIO 0

1 INTRODUÇÃO 1

2 ESTADO DA ARTE 3
- 2.1 Algoritmos do tipo *stencil* 3
- 2.2 Algoritmos sobre matrizes densas 5
  - 2.2.1 Localidade espacial 6
  - 2.2.2 Localidade temporal 7
  - 2.2.3 Frameworks e auto-tuning 9
- 2.3 Algoritmos sobre matrizes esparsas 9
  - 2.3.1 Localidade espacial 13
  - 2.3.2 Localidade temporal 15
  - 2.3.3 Frameworks e auto-tuning 16

3 FRAMEWORK FVLIB 17

ii NÚCLEO DA DISSERTAÇÃO 23

4 OTIMIZAÇÃO DA VERSÃO SEQUENCIAL 24
- 4.1 Simplificação do código 24
- 4.2 Melhoria para a hierarquia de memória 25

5 EXPLORAÇÃO DE PARALELISMO 32
- 5.1 Primeira versão 32
- 5.2 Segunda versão 35

6 CONCLUSÃO E TRABALHO FUTURO 39

iii APENDICES 42

A AMBIENTE EXPERIMENTAL 43

LISTA DE FIGURAS

Figura 1 a) Stencil de 5 pontos b) Stencil de 7 pontos 3
Figura 2 Exemplo do formato CSR 4
Figura 3 Exemplo de uma matrix dividida em 4 blocos e suas fronteiras 6
Figura 4 Exemplo de 3 iterações de um stencil 7
Figura 5 Zonas fantasma ao longo das iterações do stencil 8
Figura 6 Exemplo de uma matriz esparsa A 10
Figura 7 Representação da matriz esparsa A no formato CSR 10
Figura 8 Representação da matriz esparsa A no formato COO 11
Figura 9 Representação da matriz esparsa A no formato DIA 12
Figura 10 Representação da matriz esparsa A no formato ELL 12
Figura 11 Distribuição de formatos de representação entre matrizes estruturadas e não estruturadas. 12
Figura 12 Flooding do algoritmo de Cuthil. 15
Figura 13 Localidade temporal dos dados no formato CSR. 15
Figura 14 Localidade temporal dos dados no formato COO. 16
Figura 15 Exemplo de uma malha. 17
Figura 16 Principais estruturas de dados do caso de estudo. 18
Figura 17 Alguns dos cálculos repetidos no algoritmo original. 20
Figura 18 Matriz original. 21
Figura 19 Comparação das instruções e ciclos de relógio entre a versão original e a versão modificada. 25
Figura 20 Acessos à memória para G e PHI. 26
Figura 21 Comparação dos *cache misses* entre a versão original e as novas versões. 27
Figura 22 Comparação das instruções entre a versão original e as novas versões. 27
Figura 23 Matriz de adjacência depois da renumeração. 28
Figura 24 Relação entre a largura de banda da matriz e a célula inicial do algoritmo. 29
Figura 25 Distribuição da média das larguras de banda. 29
Figura 26 Associação entre célula inicial e largura de banda. 30
Figura 27 Relação entre o erro e a célula inicial do algoritmo. 30
Figura 28 Distribuição do erro. 30

LISTA DE FIGURAS

Figura 29 Comparação dos cache misses entre a versão original e as novas versões. 31
Figura 30 Comparação das instruções entre a versão original e as novas versões. 31
Figura 31 Divisão da matriz para 4 *threads*. 33
Figura 32 Divisão do domínio do problema para *threads*. 34
Figura 33 Comparação entre tempo de execução, resultados por segundo e *speedup* entre versões paralelas. 36
Figura 34 Comparação entre tempo de execução, resultados por segundo e *speedup* entre versões paralelas. 36
Figura 35 Comparação entre tempo de execução das várias versões da tese e o speedup relativamente à versão original do código. 38

LIST OF LISTINGS

2.1 Stencil 7 pontos . . . 4
2.2 Multiplicação de uma matriz no formato CSR por um vetor. . . . 5
2.3 Multiplicação de uma matriz no formato COO por um vetor. . . . 13
2.4 Algoritmo de Cuthil . . . 14

3.1 Algoritmo de convecção-difusão . . . 19
3.2 Algoritmo de partida para a dissertação . . . 21

4.1 Algoritmo sequencial final. . . . 25

5.1 Pseudo-código final da dissertação . . . 37

Parte I

MATERIAL INTRODUTÓRIO

1

INTRODUÇÃO

Ao longo do tempo a simulação computacional ganhou importância na ciência, métodos iterativos para determinar soluções de sistemas de equações lineares, tal como Jacobi ou Gauss-Seidel e soluções de equações diferenciais baseadas em derivadas por diferenças finitas são utilizados, entre outros, em problemas de convecção-difusão, mecânica de gases, fluidos e sólidos. São utilizadas malhas 2D e 3D como estruturas de dados para guardar informações referentes ao sistema, isto é, temperaturas, fluxos, tensões, deformações, velocidades entre outras. Cada vez mais pretende-se lidar com problemas de maior dimensão, que requerem uma maior capacidade computacional. Estes problemas aparecem grande parte das vezes associados a algoritmos do tipo *stencil* e multiplicação matriz-vetor. Também na computação gráfica se podem encontrar muitas vezes este tipo de algoritmos no tratamento de imagens.

Um algoritmo do tipo *stencil* efetua uma travessia de uma matriz ou um vetor que representam malhas (2D ou 3D) e envolvem atualizações sucessivas de valores de uma grelha multi-dimensional com os valores de um conjunto de vizinhos. Por isso mesmo, muitas vezes, estes algoritmos são chamados também de algoritmos de computação sobre os vizinhos mais próximos. Os elementos da malha são, na maior parte das vezes, referenciados como células contendo a informação relevante para os cálculos necessários em cada iteração. São algoritmos intensivos para a memória fazendo com que a execução seja limitada pela sua largura de banda.

Nos últimos anos tem havido limitações por parte da indústria no aumento da frequência do relógio dos novos processadores causadas por limitações de largura de banda da memória e dificuldades na dissipação de calor, problemas estes que, apesar de reduzidos com a introdução de processadores multi-núcleo, continuam a existir.

Assim, os fabricantes de processadores decidiram apostar noutros mecanismos de aumento da capacidade computacional. Melhorias na hierarquia de memória tornaram-se no modo de diminuir a quantidade de ciclos que um determinado processo espera por informação necessária, justificando assim aumentos nos níveis e quantidade de memória *cache* disponíveis. A memória *cache* está distribuída em vários níveis e quanto mais próxima do processador menor vai ser o custo de carregar dados da mesma.

Outra das alternativas encontradas para aumentar a capacidade computacional foi a introdução de um maior número de núcleos por processador. Cada um dos núcleos tem normalmente *cache* privada e partilhada (normalmente a *cache* mais afastada do núcleo é a partilhada enquanto as outras são pri-

vadas ao mesmo). É assim possível correr processos ou fios de execução em paralelo utilizando a capacidade de executar vários fluxos de instruções em simultâneo.

As otimizações dos algoritmos *stencil* existentes focam-se na redução do tráfego de dados entre os diferentes níveis de memória. O grande desafio de escalar este tipo de algoritmos passa por limitações intrínsecas na localidade dos dados, devido à quantidade de cálculos realizados por elemento (O(v) onde v é o número de vizinhos) o que limita a exploração da localidade temporal, implicando um rácio baixo nas operações realizadas por byte. Quando se está a lidar com estruturas de dados irregulares há dificuldade acrescida em explorar a localidade espacial. Estes problemas são agravados quando se divide trabalho por vários núcleos que fazendo com que seja necessária uma atenção especial à concorrência no acesso aos dados. Há por essa razão uma maior sobrecarga de mecanismos de controlo para assegurar a coerência dos dados, de modo que não hajam erros de cálculo que possam influenciar o resultado final.

O objetivo desta dissertação é simplificar um algoritmo de convecção-difusão de calor e aplicar-lhe técnicas de otimização de modo a melhorar a organização de dados e o modo como são acedidos e utilizar técnicas de exploração de paralelismo. Serão identificados cálculos supérfluos e repetidos e possíveis modificações de código para reduzir a quantidade de instruções. Para que se aproveite ao máximo a localidade dos dados será necessário garantir que são organizados por ordem de acesso e para aproveitamentos da localidade temporal será necessário identificar possíveis reutilizações dos mesmos. Para uma exploração eficaz de paralelismo será necessário manter os ganhos na localidade espacial e temporal dos dados quando se divide o trabalho por vários núcleos, para isso será necessária uma atenção especial a partilhas de dados entre vários núcleos e acessos concorrentes aos mesmos.

Como resultado final desta tese espera-se obter melhorias no tempo de execução do caso de estudo, através da modificação da versão sequencial do algoritmo e através da exploração de paralelismo.

A próxima secção apresenta a pesquisa efetuada sobre o estado da arte dos algoritmos *stencil* sobre matrizes densas e esparsas com ênfase em otimizações na localidade espacial e temporal. Serão apresentadas diferentes *frameworks* de *auto-tuning* existentes e os problemas que procuram solucionar.

Na secção seguinte será apresentado um algoritmo utilizado nesta dissertação, que utiliza a biblioteca *Finite Volume Library (FVLib)*. Esta biblioteca suporta estruturas 2D e 3D e não foi preparada para suportar paralelismo, assim sendo será necessária uma análise das estruturas em utilização e suas possíveis modificações.

Nas seguintes secções serão exploradas otimizações da versão sequencial do código, simplificando a sua computação, assim como melhorias para a hierarquia de memória.

Nas secções finais serão apresentadas as versões paralelas do código, uma que beneficia das melhorias efetuadas anteriormente e ainda uma versão que aproveita a localidade temporal dos dados.

2

ESTADO DA ARTE

2.1 ALGORITMOS DO TIPO *stencil*

Um algoritmo do tipo *stencil* percorre um vetor ou uma matriz e atualiza os valores de cada posição com um novo valor influenciado diretamente pelos seus vizinhos, estas matrizes representam malhas (2D ou 3D). É um dos padrões comuns em algoritmos numéricos. Para melhor entendimento do algoritmo consideremos uma matriz M definida em 2D por coordenadas *(x,y)*, uma forma geral de um *stencil* de difusão 2D de 5 pontos, numa iteração n, numa coordenada *(x,y)*, pode ser escrito como:

$$M^{n+1}(x,y) = \frac{1}{5}(M^n(x,y) + M^n(x+1,y) + M^n(x-1,y) + M^n(x,y+1) + M^n(x,y-1)) \quad (1)$$

Nesta situação específica cada valor em cada posição envolvida pesa 20% no valor a guardar na célula em questão, no entanto pode ser dado peso diferente a cada vizinho.

É possível aplicar o mesmo tipo de raciocínio para uma matriz definida em 3D por coordenadas *(x,y,z)* e utilizar o mesmo método para gerar uma relação entre 7 pontos. De notar que nem sempre os vizinhos diretos estão diretamente envolvidos nos cálculos e podem ser considerados dois ou mais vizinhos de cada um dos lados, isto é, uma posição *(x,y)* pode ser diretamente influenciada pelo valor da posição *(x-2,y)* e *(x+2,y)*, etc. A este número de vizinhos envolvidos é chamado tamanho da fronteira.

A Figura 1 exemplifica no espaço as células envolvidas no cálculo de um *stencil* para uma determinada célula central para 5 pontos e para 7 pontos (2D e 3D).

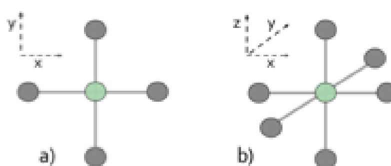


Figura 1.: a) Stencil de 5 pontos b) Stencil de 7 pontos

Um exemplo de um algoritmo *stencil* 3D que analogamente à equação 1, calcula o valor atualizado de uma célula influenciado pelos seus vizinhos diretos, mas num sistema de três dimensões *(x,y,z)*, e, que possa ser associado a b) da figura 1 pode ser traduzido no algoritmo 2.1

```
for (int t = 0; t < T; t++) {
    for (int x = 1 ; x < N-1 ; x++) {
        for(int y = 1 ; y < N-1 ; y++) {
            for(int z = 1 ; z < N-1 ; z++) {
                new[x][y][z] = alpha * old[x][y][z] +
                            (beta/6) * ( old[x][y][z-1] + old[x][y][z+1] +
                                        old[x][y-1][z] + old[x][y+1][z] +
                                        old[x-1][y][z] + old[x+1][y][z] );
            }
        }
    }
}
```

Algoritmo 2.1: Stencil 7 pontos

O algoritmo 2.1 considera uma estrutura 3D no cálculo de *stencils* (sem cálculo de fronteiras) durante T iterações, de lado N, em que não há perda por parte do sistema, logo *alpha+beta=1*, em que a contribuição de cada vizinho numa célula é *1/6*beta* e a antiga informação da célula tem peso *alpha*. Estes pesos poderiam ser assentes numa formulação matemática mais complexa, tornando assim um algoritmo simples num mais complexo. Este tipo de algoritmo é adequado a matrizes densas.

Em muitos casos as matrizes apresentam muitos zeros, sendo necessário recorrer à utilização de matrizes esparsas. Algoritmos de multiplicação matriz-vetor aparecem assim como solução para esses problemas científicos. É comum encontrar algoritmos que multipliquem uma matriz esparsa por um vetor denso. O vetor vai guardando a cada iteração soluções para o sistema, idealmente, cada vez mais próximas da solução final.

As matrizes esparsas têm poucos elementos não nulos por linha sendo necessário encontrar um modo para guardar estes mesmos elementos num formato compactado. O formato compactado de matriz esparsa mais utilizado é conhecido como *Compressed Sparse Row (CSR)*. Um exemplo desta mesma representação pode ser visto na figura 2, neste formato a matriz é guardada em três listas, uma com o índice do primeiro não nulo em cada linha (*row_prt*), outra com o índice da coluna em que cada elemento não nulo se encontra (*col_ind*) e uma terceira com o valor dos elementos não nulos (*values*).

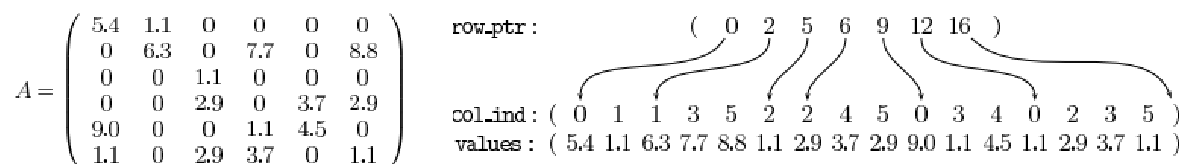


Figura 2.: Exemplo do formato CSR

Formatos como *Compressed Sparse Column (CSC)* e *Coordinate (COO)* são também bastante utilizados. O formato CSC é semelhante ao CSR, a lista com os elementos não nulos mantêm-se, no entanto, ordenada por colunas, a lista dos índices guarda agora o índice dos elementos não nulos em

cada coluna e a última lista o índice do primeiro não nulo de cada coluna. O formato COO guarda três vetores diferentes, um com o índice da linha, outro com a coluna e outro com o valor. Este formato é bastante utilizado quando é necessário utilizar a transposta de uma matriz.

O algoritmo 2.2 é um exemplo simples da multiplicação de uma matriz no formato CSR por um vetor em que os valores da matriz estão guardados na lista *values* e são multiplicados por um vetor *x*. No algoritmo específico, a informação sobre a localização do primeiro elemento não nulo de cada linha encontra-se guardada na lista *row_ptr* enquanto o índice das colunas encontra-se guardado na lista *col_ind*, o resultado da multiplicação é guardado em *y* e *N*N* é o tamanho da matriz esparsa.

```
1 for (int i = 0; i < N; i++)
2   for (int j = row_ptr[i]; j < row_ptr[i+1]; j++)
3       y[i] += values[j] * x[col_ind[j]];
```

Algoritmo 2.2: Multiplicação de uma matriz no formato CSR por um vetor.

2.2 ALGORITMOS SOBRE MATRIZES DENSAS

Tal como referido anteriormente, para matrizes densas, em que o número de valores nulos é reduzido ou até, algumas vezes, inexistente, é utilizado, normalmente, o típico algoritmo *stencil*, percorrendo todos os elementos da matriz e calculando um novo valor influenciado pelos seus vizinhos ao longo de várias iterações.

No entanto devido ao rápido desenvolvimento dos processadores, tornou-se rapidamente necessário ter atenção ao tipo de arquitetura que está a ser utilizada, assim como modos de otimizar a localidade dos dados.

Tendo em consideração a maioria dos problemas *stencil*, em que são utilizadas matrizes densas que não cabem na *cache* do processador, é necessário encontrar modos de melhorar a localidade temporal dos dados garantindo que estes, uma vez carregados para níveis superiores de *cache* sejam utilizados o máximo possível. É necessária uma boa divisão de dados por blocos garantindo que a sua elevada utilização se traduza numa diminuição do tempo que um processo fica em espera de dados para prosseguir a sua execução ao mesmo tempo que se verifica uma maior taxa de acerto na *cache*.

Quando se têm disponíveis vários núcleos por processador é necessário dividir os dados pelos mesmos. Se for considerada, por exemplo, uma matriz regular e um processador com 4 núcleos é possível dividir a matriz em 4 blocos e atribuir cada bloco dessa matriz a um núcleo, tendo em conta que cada um dos núcleos consegue trabalhar independentemente dos outros. A figura 3 apresenta este mesmo exemplo de partição da matriz.

Apesar de se poder dividir a matriz por núcleos, as fronteiras da mesma, tal como assinaladas na figura contêm dependências de dados fora do bloco disponível, assim, perante esta situação será necessária sincronização dos dados nestas zonas fantasma.

Caso o total de bytes por operação necessário pelo algoritmo seja maior que do que o disponibi-

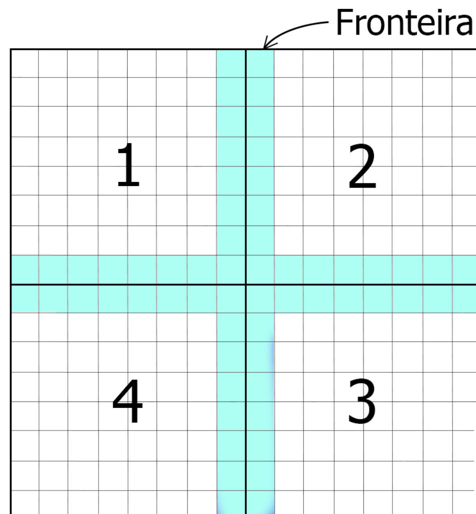


Figura 3.: Exemplo de uma matrix dividida em 4 blocos e suas fronteiras

lizado pela arquitetura, o algoritmo é limitado pela largura de banda da memória, por isso torna-se necessária uma boa utilização da hierarquia de memória.

O algoritmo 2.1 possui um total 16 operações, 6 de adição, 2 de multiplicação, 7 para carregar valores das posições utilizadas pelo *stencil* e 1 operação para guardar o resultado. Existem 8 acessos a memória, que correspondem a um total de 32 bytes de dados no caso de precisão simples e 64 bytes no caso de precisão dupla. Como o algoritmo *stencil* tem uma boa localidade espacial, que será explicada na próxima secção, e tendo em consideração um espaço 3D, em vez de 8 acessos a memória externa apenas serão necessários 2, um para carregar e outro para guardar dados, ou seja, 8 bytes para precisão simples e 16 para dupla. Tendo isto em conta o total de bytes por operação do algoritmo 2.1 é aproximadamente 0,5 para precisão simples e 1 para precisão dupla.

Um processador *core i7* da *Intel* tem um pico de bytes por operação de 0,29 e 0,59 para precisão simples e dupla respetivamente (Nguyen et al. (2010)), caso este algoritmo seja executado nele, ficará limitado pela largura de banda de memória, pois será necessária uma capacidade de 0,5 bytes/op para precisão simples e 1 bytes/op para precisão dupla.

2.2.1 *Localidade espacial*

Se uma aplicação *stencil* tiver uma boa localidade espacial aumenta a sua taxa de acerto na *cache*. Os ganhos na localidade espacial nos algoritmos *stencil* passam pelo acesso a dados contíguos.

A figura 4 apresenta um exemplo do porquê de algoritmos *stencil* terem boa localidade espacial.

Supondo que cada linha carregada para a *cache* contém 4 elementos da matriz, a zona assinalada

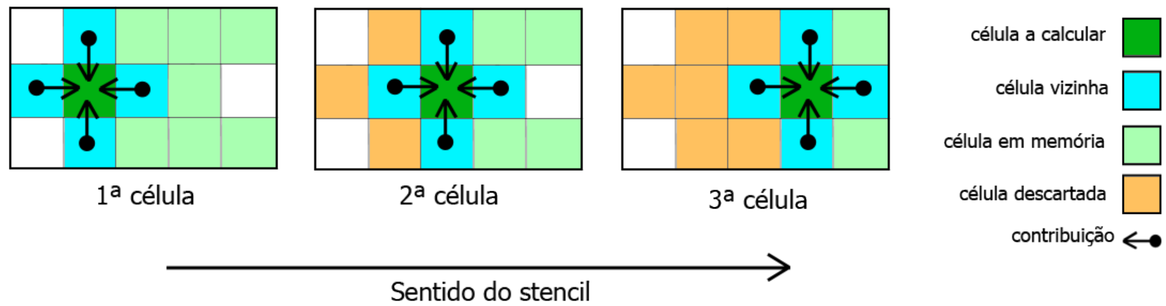


Figura 4.: Exemplo de 3 iterações de um stencil

como célula em memória representa 3 dessas linhas carregadas onde cada célula a calcular sofre contribuições das suas vizinhas.

Com exceção do calculo da terceira célula, em que uma nova linha terá de ser carregada para a *cache*, todos os outros dados necessários encontram-se ao longo de 4 células em *cache*, assim, apenas é necessário carregar novas linhas para a *cache* a cada 4 células.

À medida que as células deixam de ser utilizadas podem ser assinaladas como descartadas, caso não sejam utilizadas para o cálculo de diferentes células, caso contrário há o reaproveitamento temporal dos dados. Utilizar uma partição de blocos por núcleo é considerada uma boa técnica para as aplicações escalarem, no entanto como existem cálculos de células de fronteira, estas envolvem um tratamento especial, por isso mesmo muitos autores optam por carregar um bloco de dados para memória e aplicar o *stencil* a todos os elementos sem dependências de vizinhos enquanto um outro fio de execução auxiliar se encarrega desses casos especiais. Strzodka et al. (2010) opta por calcular o *stencil* utilizando blocos independentes, por exemplo, caso a arquitetura utilizada tenha disponíveis 2 núcleos e caso se considere a matriz da figura 3 e um *stencil* de 5 pontos é possível calcular o resultado para os blocos ímpares em simultâneo e, no final, para os blocos pares.

2.2.2 *Localidade temporal*

O principal problema dos algoritmos *stencil* é o facto dos dados serem utilizados poucas vezes em cada iteração e após isso serem descartados. Tendo em conta que os dados podem vir a ser reutilizados numa futura iteração, são necessárias melhorias na localidade temporal para que esses mesmos dados possam vir a ser utilizados mantendo-se na memória *cache*.

O algoritmo 2.1 possui localidade temporal na medida em que valores em *cache* são utilizados mais do que uma vez quando se itera sobre o eixo z. Se for considerado esse mesmo *stencil* de 7 pontos um valor que se encontre numa posição *(i,j,k)* é utilizado para o cálculo não só da posição *(i,j,k)* como também das posições *(i,j,k-1)* e *(i,j,k+1)*, assim sendo, esse valor será utilizado nos cálculos do *stencil* em 3 células seguidas.

Apesar disto uma utilização do mesmo valor apenas 3 vezes não é suficiente para que o algoritmo deixe de ser limitado pela largura de banda da memória, tal como exemplificado no fim da secção 2.2. Caso se aumente o tamanho da fronteira é possível que um algoritmo deixe de ser limitado pela largura de banda da memória.

Mais uma vez, utilizando o caso do *core i7* da *Intel*, em que o pico de bytes por operação é 0,29 e 0,59 para precisão simples e dupla respetivamente, considerando que estamos a utilizar um algoritmo em 3D, desta vez com 4 vizinhos de cada lado, faz um total de 52 operações, 24 de adição, 2 de multiplicação, 25 de carregamento e 1 para guardar dados. Considerando uma boa divisão por blocos, em que um bloco cabe em memória *cache*, apenas 1 elemento em cada iteração será guardado e 1 carregado de níveis de memória inferiores. Assim sendo este *stencil* iria necessitar de uma capacidade computacional de 0,16 bytes/op para precisão simples e 0,32 para precisão dupla. Neste caso o algoritmo seria limitado pelo processador e não pela memória.

Uma das técnicas de blocos para exploração da localidade temporal consiste em dividir a matriz em blocos que caibam na *cache* e garantir que, assim que é necessário calcular o valor atualizado para a célula abaixo da atual o valor desta ainda esteja em *cache*. Para melhor compreensão, imagine-se o caso em que 2 linhas de uma matriz cabem na *cache*, é possível assim garantir que para uma posição *(i,j)* em 2D, os dados em *(i-1,j)* ainda se encontram em *cache*, aproveitado-se assim a localidade temporal dos mesmos.

Nguyen et al. (2010) opta por explorar localidade temporal entre iterações calculando repetidamente *stencil* num bloco de células utilizando a cada iteração apenas dados que não tenham dependências de outro bloco de células. Um modo fácil de entender isto é imaginar uma janela em que o limite é a fronteira e essa janela encurta ao longo das iterações tendo em consideração uma nova zona fantasma. A figura 5 exemplifica esta situação.

O resultado da zona fantasma da primeira iteração influência o valor da zona fantasma da iteração

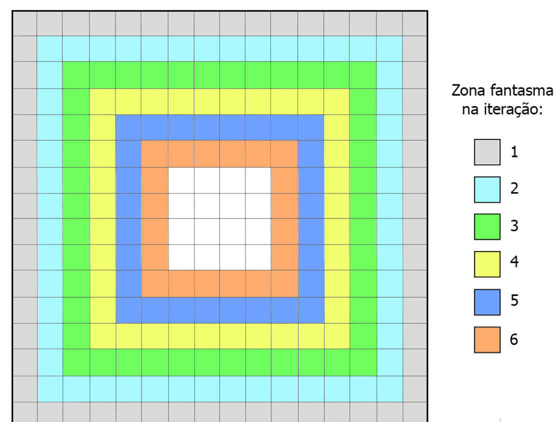


Figura 5.: Zonas fantasma ao longo das iterações do stencil

2 logo a zona fantasma da próxima iteração é alterada e assim sucessivamente. Esta solução tem um limite de iterações e no final desse limite é necessário atualizar as zonas fantasma.

Este tipo de pensamento foi muito importante na medida em que numa das etapas finais desta dissertação foi usado o mesmo tipo de raciocínio para obter ganhos explorando localidade temporal dos dados.

2.2.3 *Frameworks e auto-tuning*

Recentemente surgiram ferramentas que aplicam otimizações a código *stencil* de modo a que seja fácil a sua utilização. Todas as técnicas descritas nas secções anteriores podem ser utilizadas por ferramentas de *auto-tuning*. Datta et al. (2008) utiliza técnicas de *auto-tuning* de alocação de memória, desdobramento de ciclos tirando partido de acessos alinhados a memória, decomposição de domínio com o objetivo de utilizar técnicas de blocos de dados. Lutz et al. (2013) apresenta uma *framework* de *auto-tuning* que permite ao utilizador fornecer parâmetros à aplicação e ajustar a sua execução de forma autónoma dependendo do tipo de capacidade computacional do sistema.

Outro fator muito importante na escalabilidade de aplicações em ambientes multi-núcleo é o bom balanceamento das mesmas, é importante garantir que todos os núcleos tenham trabalho e tentar que estes estejam em espera o mínimo tempo possível. *Auto-tuning* ajuda nesse aspeto pois ferramentas deste tipo se encontram preparadas para gerar código eficiente para diferentes tipos de ambientes de computação.

Existem também bibliotecas que fornecem mecanismos para o programador paralelizar o seu código de uma maneira fácil. A biblioteca *Mint*, implementada por Unat et al. (2011), permite ao programador introduzir *pragmas* no seu código *stencil* semelhantes aos utilizados em *OpenMP* que paralelizam o código não apenas para CPU como também para GPU. Permite ao programador criar secções paralelas, escolhendo dados a ser transferidos entre a memória do hospedeiro (CPU) e o dispositivo (GPU).

Apesar de ser uma opção, *auto-tuning* pode não ser vantajoso, tendo em conta que o programador está limitado às capacidades de uma *framework* específica. A qualidade da *framework* e a sua adaptação a um tipo específico de problema pode ser limitador do sucesso da sua utilização.

2.3 ALGORITMOS SOBRE MATRIZES ESPARSAS

As matrizes esparsas definem-se como matrizes em que uma grande percentagem dos elementos da matriz se tratam de valores nulos. Estas matrizes são irregulares e os acessos aos dados também o podem ser. A largura de banda de uma matriz esparsa pode ser definida como o valor máximo da diferença entre os índices dos elementos da matriz e a sua diagonal.

Existe uma relação entre uma matriz esparsa e um grafo, sendo esta matriz conhecida como matriz de adjacência de um grafo, onde cada valor não nulo representa uma ligação entre dois nodos de um grafo. Por outro lado, uma matriz esparsa pode ser representada como um grafo, se for considerada a matriz como uma matriz de adjacência. Esta relação entre matriz esparsa e grafo revelou-se um aspeto

essencial para o sucesso desta dissertação, tendo em consideração que numa fase inicial o algoritmo do caso de estudo utilizava um grafo de adjacência.

Um algoritmo de multiplicação de uma matriz esparsa por um vetor consiste na operação $y \leftarrow y + Ax$, onde A é uma matriz esparsa e x e y vetores densos. Apesar de ser considerado dos algoritmos mais importantes da computação, é um algoritmo bastante difícil de escalar em multi-núcleos e o principal ganho em termos computacionais passa por escolher o melhor formato para guardar os dados.

Para contrariar o facto das matrizes esparsas conterem muitos zeros é necessária a utilização de um formato de compressão de matrizes, representando apenas os valores não nulos (por exemplo, CSR, apresentado anteriormente). No entanto existem vários formatos que são mais eficientes quando a matriz possui uma dada estrutura.

Os formatos mais conhecidos para representação de matrizes esparsas estão adaptados para 3 tipos de matrizes:

- matrizes diagonais
- matrizes com número de não zeros por linha uniforme
- matrizes com número de não zeros por linha não uniforme

Caso não se considerem modificações da matriz, como por exemplo permuta de linhas, é necessário encontrar um formato adequado para representar a matriz esparsa que melhor se adapte à situação. Para o resto dos formatos que se irão explicar de seguida será utilizada a matriz da figura 6 como exemplo.

Já foi analisado na secção 2.1 o formato CSR, no entanto é necessário acrescentar que este formato

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

Figura 6.: Exemplo de uma matriz esparsa A

é o mais utilizado quando se quer dividir a matriz em blocos de linhas.

A vantagem deste formato é o facto de permitir um diferente número de não zeros por linha sem

$$\begin{aligned} \texttt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \texttt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \texttt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

Figura 7.: Representação da matriz esparsa A no formato CSR

gastar espaço extra na memória para a sua representação. Uma das desvantagens é o facto de ser difícil encontrar um balanceamento de carga, tendo em conta que será necessário percorrer uma vez a matriz para se perceber o melhor modo de distribuir o trabalho por vários núcleos. Outra desvantagem é o facto da zona do vetor resultado não ser só utilizada por 1 núcleo o que envolve comunicação entre processos e necessidade de controlo de concorrência sobre os dados.

Por estas razões o CSR não é o formato ideal quando se quer utilizar processamento paralelo. Apesar dos dados estarem contíguos na memória não significa que estejam contíguos na matriz e por isso mesmo não se pode aproveitar a localidade espacial do vetor x e y. A representação da matriz A no formato CSR pode ser vista na figura 7

O formato COO guarda a matriz em forma de 3 vetores. Um com índice da linha, outro com o índice da coluna e um terceiro com o valor guardado na matriz, tal como exemplificado na figura 8 para a matriz esparsa A. A principal vantagem deste formato é a facilidade em utilizar a matriz transposta para o cálculo da multiplicação matriz-vetor, tendo em conta que basta trocar os índices da linha pelos índices da coluna. Outra das vantagens é os dados poderem ser organizados por linha ou coluna, podendo ser obtidos ganhos na localidade espacial como na localidade temporal, os ganhos serão explicados nas secções seguintes. Este formato ocupa uma maior quantidade de memória do que o formato CSR por guardar 3 valores por cada elemento não nulo na matriz em vez de 1 valor por cada linha da matriz.

$$
\begin{aligned}
\texttt{row} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \\
\texttt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\
\texttt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}
\end{aligned}
$$

Figura 8.: Representação da matriz esparsa A no formato COO

O formato diagonal (DIA) é utilizado quando valores não nulos estão posicionados ao longo de várias diagonais na matriz, este formato é bastante utilizado para guardar matrizes geradas por *stencils* aplicados a matrizes densas.

O formato DIA é formado por dois vetores, um com os valores da matriz e outro com o deslocamento da matriz em relação à diagonal principal. Diagonais acima da diagonal principal têm deslocamentos positivos enquanto diagonais abaixo da principal têm deslocamentos negativos. A representação da matriz A no formato DIA pode ser visualizada na figura 9.

O formato ELLPACK (ELL) é mais generalista do que o formato DIA e é utilizado por algoritmos em arquiteturas com capacidade vetorial. Considere-se que uma matriz de tamanho MxN tem no máximo K elementos por linha, numa representação no formato ELL os elementos não nulos são organizados numa sub matriz MxK alinhados à esquerda com suporte de uma outra matriz MxK com os índices dos mesmos, esse alinhamento dos dados é o porquê deste formato ser utilizado para processamento vetorial.

$$\texttt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \texttt{offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

Figura 9.: Representação da matriz esparsa A no formato DIA

O formato ELL é mais generalista do que o formado DIA porque não há necessidade dos elementos não nulos seguirem algum tipo de padrão (no caso do DIA são necessários valores não nulos em diagonais). A representação da matriz A no formato ELL pode ser visualizada na figura 10.

De um modo geral a figura 11 apresenta a distribuição dos diferentes tipos de representação de

$$\texttt{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \texttt{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

Figura 10.: Representação da matriz esparsa A no formato ELL

matrizes esparsas entre estruturadas e não estruturadas.

Em suma, o formato DIA funciona bem para matrizes com diagonais não nulas, o formato ELL é

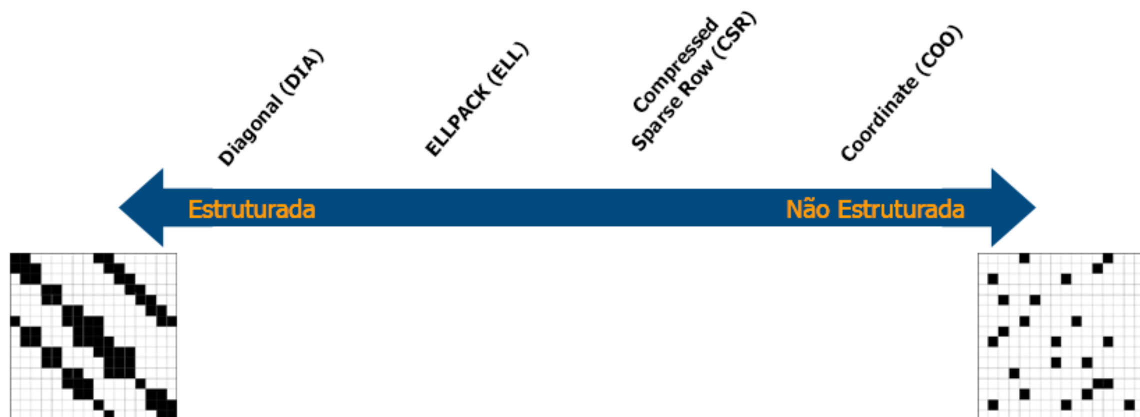


Figura 11.: Distribuição de formatos de representação entre matrizes estruturadas e não estruturadas.

semelhante ao DIA mas um pouco mais flexível e é utilizado para processamento vetorial, o formato CSR suporta matrizes com uma distribuição de elementos por linha regular sendo compacto e o formato COO é o mais genérico e flexível de todos, sendo ideal para utilizar caso seja necessário utilizar a transposta da matriz, sendo apenas necessário permutar o vetor dos índices da coluna com o vetor dos índices da linha.

Existem outros formatos para matrizes esparsas, dois deles são o formato *Compressed Sparse Block* (CSB) introduzido por Buluç et al. (2009) e o formato *Compressed Sparse eXtended* (CSX) introduzido por Kourtis et al. (2011).

O formato CSB possui muitas semelhanças com o formato CSR, na medida em que ambos usam a mesma quantidade de dados para guardar informação dos elementos não nulos, no entanto, o formato CSB procura melhorar o modo como os dados são armazenados. O formato CSB é utilizado para dividir uma matriz esparsa em quadrantes e cada um desses quadrantes voltar a ser repartido sucessivamente, enquanto estas partições ocorrem os mesmos são reorganizados, de modo a serem distribuídos pelos diferentes núcleos de um processador e poderem executar em paralelo sem qualquer tipo de dependências de dados. É um formato não orientado à melhoria da localidade dos dados mas sim à possível utilização em paralelo dos mesmos.

O formato CSX é utilizado para procura de padrões de acesso à memória e sua ordenação. É baseado num formato conhecido por *Compressed Sparse Row Delta Unit* (CSR-DU) que se apoia na existência de áreas densas numa matriz esparsa e usando essa área reduz a quantidade de dados usados na indexação dos valores não nulos da matriz.

2.3.1 *Localidade espacial*

Também nos algoritmos de multiplicação de uma matriz por um vetor é necessária especial atenção à localidade dos dados. Como os dados são organizados em diferentes tipos de formatos é necessário entender as vantagens de cada um.

No caso do formato CSR a exploração da localidade espacial dos dados encontra-se presente no acesso à matriz principal. Como os algoritmos iteram sobre os três vetores que guardam a matriz é garantido que os dados são utilizados quando carregados para a *cache*.

Tendo em consideração o algoritmo 2.2 numa situação em que os dados não cabem todos em *cache* pode-se verificar que os vetores *row_ptr* e y são percorridos em i e os vetores *values* e *col_ind* em j. Já o vetor x é acedido de forma desordenada e, por isso mesmo, será uma causa de desperdício de largura de banda da memória devido à baixa taxa de acerto na *cache*.

Considerando o formato COO e analisando o algoritmo 2.3, em que NZ é o total de elementos não nulos da matriz inicial e utilizando a nomenclatura dos vetores da figura 8, é possível perceber que o aproveitamento da localidade espacial da memória se encontra principalmente no vetor *data* porque este é percorrido em i. Caso os dados tenham sido ordenados por linha é possível aproveitar a localidade espacial também no vetor y, tendo em conta que é acedido ordenadamente, já o vetor x é acedido consoante o índice do elemento guardado na posição i.

```
1 for (int i = 0 ; i < NZ ; i++)
2    y[row[i]] += data[i] * x[indices[i]];
```

Algoritmo 2.3: Multiplicação de uma matriz no formato COO por um vetor.

No caso de ser utilizada a transposta da matriz passa a ser o vetor y acedido de forma desordenada enquanto o vetor x é acedido de forma ordenada.

Tanto o formato DIA como o ELL tem uma boa localidade espacial, no entanto como no segundo é

possível carregar 2 vetores da memória (1 da matriz de valores e outro da matriz de índices) e utilizá-los em processamento vetorial acumulando os resultados no vetor resultado.

No formato DIA há uma noção de diagonal e é possível utilizar técnicas de aproveitamento da localidade explicadas na secção 2.2.1.

É importante uma boa organização de uma matriz esparsa para que a largura de banda da mesma seja o menor possível, isto é, a diferença entre o índice de um elemento e a diagonal da matriz seja minimizada.

Cuthill and McKee (1969) criou um algoritmo que dado um determinado grafo e uma matriz de adjacência gerada por esse grafo aplica uma renumeração aos nodos do grafo. O principal objetivo é minimizar a distância entre os identificadores de cada nodo e dos seus vizinhos diretos aquando da criação da matriz de adjacência respetiva, minimizando a distância entre os valores não nulos à diagonal. A largura de banda é definida pela equação 2, onde Θ_i é a diferença entre os índices do elemento diagonal e o primeiro não zero de uma linha i.

$$max\ \Theta_i \tag{2}$$

O algoritmo baseia-se numa travessia *breadth-first* de um grafo em que a cada iteração são adicionados a uma lista os vizinhos das células atuais ainda não percorridas e em que a cada iteração essas células são renumeradas e ordenadas.

Deste modo é possível comparar este processo a um processo de *flooding* aplicado a uma malha caso se considere cada elemento da mesma um nodo de um grafo tal como representado na figura 12, onde a célula inicial está representada a vermelho.

A cada iteração são adicionados a uma nova lista de nodos os vizinhos de todos os nodos da lista da iteração atual que ainda não tenham sido ordenados, essa nova lista será utilizada na iteração seguinte. Ainda na iteração atual os nodos da lista são ordenados com um critério desejado e renumerados.

Este algoritmo apresenta bons resultados na medida em que consegue reduzir consideravelmente a largura de banda, por isso mesmo será um algoritmo de referência nesta dissertação.

O algoritmo 2.4 apresenta o pseudo-código do algoritmo.

```
1  Graph G = input()
2  while (!cur.empty()) {
3     for (Node n: cur) {
4        for (Node v: G.neighbors(n)) {
5           if (v.level > n.level + 1) {
6              v.level = n.level + 1
7              next.push(v)
8           }
9        }
10       sort(next[index:end])
11       index+=next.size()
12    }
13    for (Node n: next) P[nextId++] = n
```

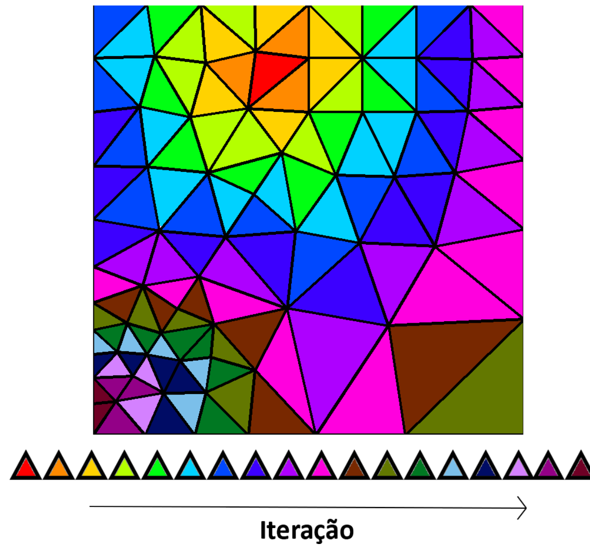


Figura 12.: Flooding do algoritmo de Cuthil.

```
14     cur.clear()
15     swap(cur, next)
16 }
```

Algoritmo 2.4: Algoritmo de Cuthil

2.3.2 *Localidade temporal*

Com a utilização dos formatos compactados de matrizes esparsas existe também aproveitamento, embora de uma forma muito limitada, da localidade temporal dos dados.

Utilizando o formato CSR para a representação de matrizes esparsas é possível aproveitar localidade temporal pois cada valor do vetor com o resultado do algoritmo pode ser utilizado mais do que 1 vez. No algoritmo 2.2, por exemplo, cada *y[i]* é utilizado *row_ptr[i+1]-row_ptr[i]* vezes em cada iteração do ciclo interno. Basicamente, caso uma linha tenha 2 elementos, o valor de *y[i]* será lido e atualizado 2 vezes, também os valor de *row_ptr* são utilizados mais do que uma vez, como é possível de verificar na figura 13. No formato COO apenas é aproveitada localidade temporal dos dados caso

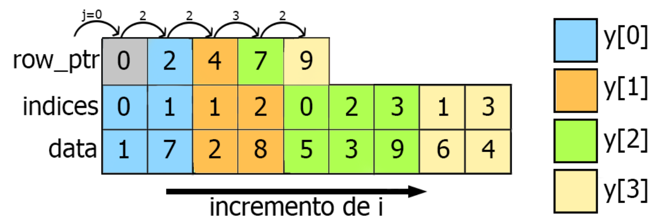


Figura 13.: Localidade temporal dos dados no formato CSR.

os valores da matriz estejam ordenados por linha ou coluna.

Considerando uma matriz ordenada por linha e se uma dessas linhas tiver *nn* valores não nulos, então o valor *y[i]* é utilizado *2*nn* vezes, como ilustrado na figura 14

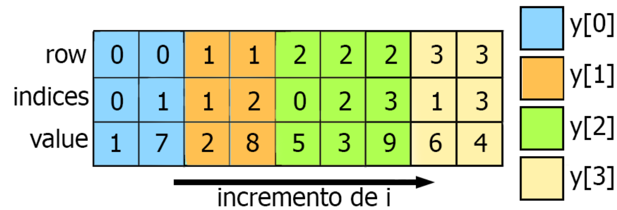


Figura 14.: Localidade temporal dos dados no formato COO.

2.3.3 *Frameworks e auto-tuning*

A mais conhecida e utilizada *framework* de *auto-tuning* de algoritmos de multiplicação de uma matriz esparsa por um vetor foi implementada por Vuduc et al. (2005) e é chamada de OSKI.

Esta *framework* esconde do programador o processo complexo de *tuning* na medida em que automaticamente adapta o código ao tipo de arquitetura a ser utilizado. Oferece algoritmos que tiram partido dos vários níveis de cache. Expõe informação sobre o custo do *tuning*. Permite a opção de *auto-profiling* e permite ao utilizador inspecionar e controlar todo o processo de *tuning*. Esta *framework* está apenas preparada para receber matrizes com representações comuns, por exemplo uma matriz no formato CSR.

A *framework* permite também a utilização de representação de uma matriz por blocos, neste caso uma representação bastante semelhante à CSB já analisada anteriormente.

Tal como Williams et al. (2007) muitos autores utilizam esta *framework* como ponto de partida para testar novas otimizações e migração das técnicas de *tuning* para outros tipos de arquitetura.

Também para algoritmos sobre matrizes esparsas as *frameworks* existentes estão limitadas pela sua pequena flexibilidade sendo necessário o algoritmo estar adaptado ao tipo de problema da *framework*.

3

FRAMEWORK FVLIB

Para caso de estudo nesta dissertação será utilizada a *FVLib*, uma biblioteca em *C++* criada para facilitar a implementação de algoritmos de simulação para computação científica sobre volumes finitos, como por exemplo simulações de rios, ondas, inundações, tsunamis, fogo de florestas e difusão de calor.

Esta biblioteca contém estruturas de malhas 3D e 2D, vários tipos de vetores, células, e diferentes outros tipos de estruturas necessários para a implementação desses mesmos algoritmos. Será utilizado um problema de convecção-difusão formulado por Clain et al. (2012) onde a cada iteração é calculado o fluxo entre células de uma malha, ou seja, a quantidade de calor transferido entre células. Um exemplo de uma malha encontra-se ilustrado na figura 15 (Clain et al. (2012)), onde é possível identificar células (c), bordas (e) e a normal de uma borda (n).

São utilizadas informações sobre tamanho e posicionamento de células, comprimento e velocidade do fluxo em cada borda. A equação 3 representa o objetivo deste algoritmo, o somatório em cada célula a cada iteração.

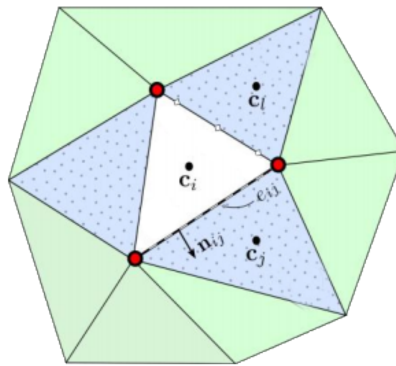


Figura 15.: Exemplo de uma malha.

$$\sum F_{ij} = [V.n_{ij}]^{+}\phi_i + [V.n_{ij}]^{-}\phi_j - k\frac{\phi_j - \phi_i}{|b_i.b_j|} \quad (3)$$

Os resultados desta função (F_{ij}) são guardados num vetor de fluxos, a velocidade do fluxo (V) é diferente para cada borda e mantém-se constante ao longo de toda a execução. O valor n_{ij} é o valor da normal de cada borda. O valor de ϕ é atualizado no inicio de cada iteração e guardado num vetor *PHI*, onde numa data iteração i o valor de *PHI[i]=1* enquanto o resto das posições do vetor se mantém nulas. A constante k é calculada utilizando um parâmetro carregado do ficheiro de entrada. O valor de b é referente ao centro geométrico de cada célula e mantém-se também constante ao longo de toda a execução. O algoritmo em questão utiliza uma malha de duas dimensões, para isso a *FVLib* possui estruturas adequadas, nomeadamente *mesh, edge, cell e point*.

A representação da estrutura inicial da malha está representada na figura 16. Nesta estrutura a malha possui uma lista de células, bordas, vértices, entre outros, cada célula possui três bordas e cada borda aponta para a célula imediatamente à sua esquerda e à sua direita.

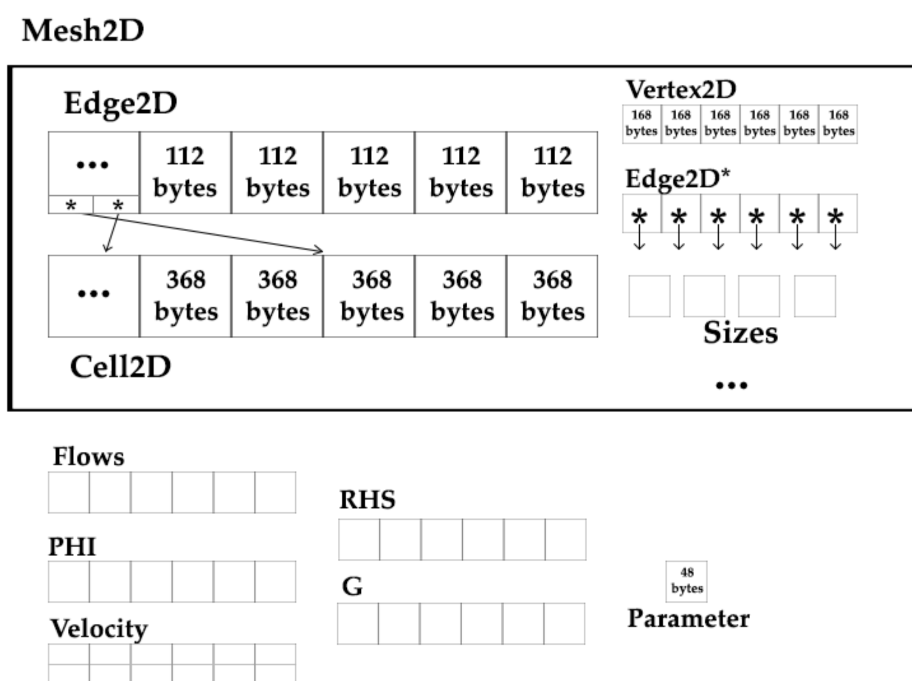


Figura 16.: Principais estruturas de dados do caso de estudo.

O algoritmo consiste em três etapas representadas na equação 4 resultando em três ciclos aninhados no código.

Na equação 3 é criado o vetor F com o fluxo para cada borda, fatores como fronteira e velocidade normal de cada fluxo são considerados nesta etapa.

$$\phi \overset{makeFlux}{\rightarrow} \underset{makeResidual}{F} \overset{update}{\rightarrow} G \overset{matricialmethod}{\rightarrow} \phi \tag{4}$$

No ciclo intermédio é atualizado cada um destes fluxos consoante a área de cada célula e o resíduo para a mesma, estes novos valores são acumulados em G, de notar que estes resíduos são calculados

à priori no inicio da função *main*.

No ciclo mais externo o resultado é colocado numa matriz para posteriormente ser calculado o resultado final e o seu erro.

O algoritmo 3.1 representa o pseudo-código para este problema.

```
1  for N in totalCells do
2     makeFlux(...,F,...){
3        for I in totalEdges do
4           if(!FRONTIER[I])
5              leftPhi = PHI[edge[I]->left_cell_id];
6              rightPhi = PHI[edge[I]->right_cell_id];
7              B=right_cell_centroid-left_cell_centroid;
8              if(edge[I] -> normal_velocity < 0) F[I] = edge[I] -> normal_velocity
                   * rightPhi;
9              else F[I] = NORMAL[I] * leftPhi;
10             done
11             F[I] -= K * (rightPhi-leftPhi) / B;
12             F[I] *=  LENGHT[I];
13          else
14             FRONTIER CALCULATION
15          done
16       done
17
18    };
19     for T in totalEdges do
20       G[edge[T]->left_cell_id]+=F[T];
21       G[edge[T]->right_cell_id]-=F[T];
22    done
23    for J in totalCells do
24       G[J]/=AREA[J];
25       G[J]-=RESIDUAL[J];
26    done
27 done
```

Algoritmo 3.1: Algoritmo de convecção-difusão

No final do algoritmo é calculado o erro do cálculo, quanto mais baixo o valor do erro melhor o resultado.

Inicialmente todos os acessos a estruturas e dados eram efetuados com recurso a apontadores. Esta utilização de apontadores traz dois tipos de problemas, primeiro os acessos não contíguos a dados e depois uma dificuldade de vetorização do código. Estes acessos desalinhados implicam um aumento dos *miss rates* na memória e um aumento da quantidade de instruções resultando num aumento do tempo de execução. A estrutura principal da malha contém também muitos dados supérfluos que aumentam o tamanho dos dados do problema fazendo com que haja uma maior percentagem que não caiba na *cache*

Ao longo da dissertação serão utilizados dois *inputs* de dados, um pequeno para efeitos de validação de resultados e outro para testes mais significativos. O tamanho dos dados para cada estrutura está representado na tabela 1.

Estrutura	Conjunto de dados pequeno	Conjunto de dados grande
# Células / # Bordas / # Vertices	1552/ 2376/ 825	105626/ 158839/ 53214
Tamanho das células (aprox.)	0,55 MB	37 MB
Tamanho das bordas (aprox.)	0,25 MB	17 MB
Tamanho dos vértices (aprox.)	0,13 MB	8,5 MB
Outros vetores (aprox.)	0,13 MB	8,48 MB
Total (aprox.)	1,06 MB	70,1 MB

Tabela 1.: Tamanho aproximado dos dados de teste

Foram efetuadas previamente a esta dissertação modificações ao código com o objetivo de remover alguns desses problemas, esse código será utilizado como ponto de partida.

A primeira modificação teve como objetivo diminuir o tamanho dos dados do problema, sendo depois reduzida a quantidade de cálculos repetidos a cada iteração e melhorado o alinhamento e acesso aos dados. Alguns cálculos referentes às células repetidos passaram a ser pré-calculados e guardados os valores em vetores, também para as bordas passou a ser feito o mesmo tipo de pré-cálculos, tal como representado na figura 17.

$$\sum F_{ij} = [V.n_{ij}]^+ \phi_i + [V.n_{ij}]^- \phi_j - k\frac{\phi_j - \phi_i}{|b_i.b_j|}$$

velocidade normal

Difusão

Figura 17.: Alguns dos cálculos repetidos no algoritmo original.

Foi modificado o código para que o fluxo fosse guardado diretamente no vetor final de cada iteração G, podendo assim ser descartado o vetor de fluxos F (remoção das linhas 20 e 21 do algoritmo 3.1).

A *FVLib* utilizava um grafo de apontadores para fazer a ligação entre diferentes células. Foi convertida essa estrutura com apontadores para uma matriz esparsa. Este tipo de visão foi a principal motivação e o ponto de partida para esta dissertação. Com a ligação entre células guardada num formato de matriz esparsa foi possível aplicar técnicas de matrizes esparsas ao código, que se tornou semelhante a um *stencil*.

Tendo em consideração uma ordenação dos dados por colunas e para facilitar futuro paralelismo, foi utilizado o formato *COO* para guardar as adjacências entre células, foram criados para cada borda dois valores, um com o valor identificador (*id*) da célula à esquerda e outro com a célula à direita de uma borda, estes valores passaram a ser guardados em vetores *LEFT* e *RIGHT*. A matriz de adjacência entre células pode ser representada tal como na figura 18, onde cada ponto negro representa

a existência de uma borda entre uma célula esquerda e direita.

A *FVLib* foi paralelizada com sucesso limitado devido à pequena organização de dados. Para esta tese foi encarado o problema como um problema sobre matrizes esparsas sendo possível explorar técnicas de paralelização conhecidas desse domínio.

De notar que o tamanho dos *inputs* reduziu imenso tendo o *input* pequeno agora aproximadamente *136 KB* e o maior *8,9 MB*. O seu pseudo-código pode ser visto no algoritmo 3.2.

Este foi o ponto de partida da dissertação atual e as comparações serão feitas baseando-se nesta versão do código. De referir que a versão original do código para o conjunto de dados maior tinha um tempo de execução de cerca de 161 minutos, já a versão modificada e utilizada como ponto inicial da dissertação demorava aproximadamente 23 vezes menos, cerca de 7 minutos.

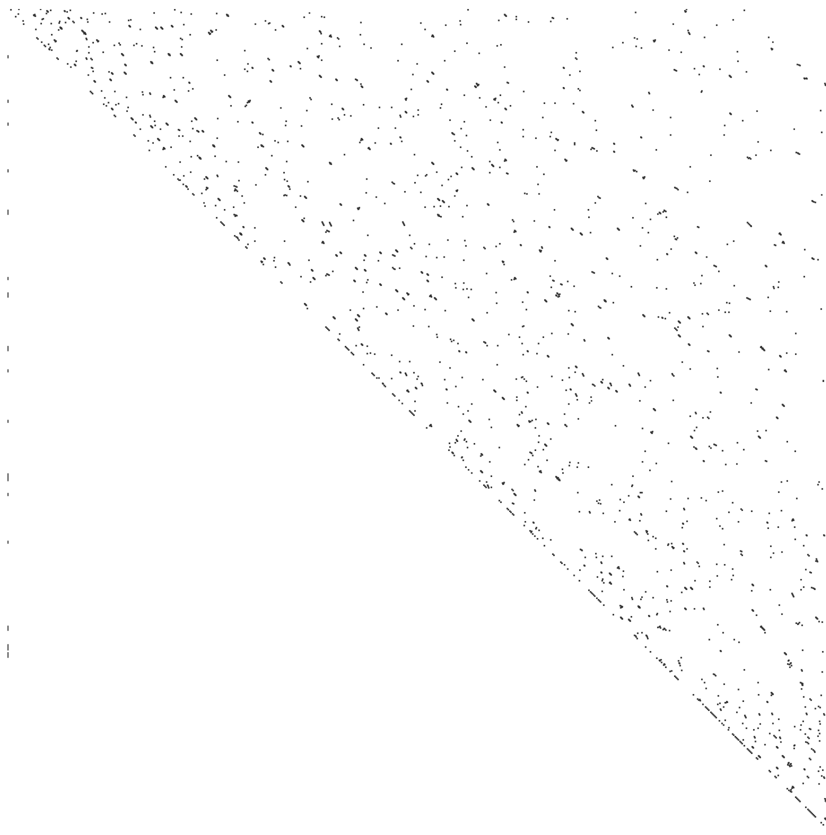


Figura 18.: Matriz original.

```
1 for N in totalCells do
2    makeFlux(...,F,...){
3       for I in totalEdges do
4          if(!FRONTIER[I])
5             leftPhi = PHI[LEFT[I]];
6             rightPhi = PHI[RIGHT[I]];
7             if(NORMAL_VELOCITY[I] < 0) current_edge = NORMAL_VELOCITY[I] *
                 rightPhi;
8             else current_edge = NORMAL_VELOCITY[I] * leftPhi;
```

```
9               done
10              current_edge -= CONSTANT_VALUES[I] * (rightPhi-leftPhi);
11              current_edge *=  LENGHT[I];
12              G[LEFT[I]] += current_edge;
13              G[RIGHT[I]] -= current_edge;
14          else
15              FRONTIER CALCULATION
16          done
17      done
18
19  };
20  for J in totalCells do
21      G[J]/=AREA[J];
22      G[J]-=RESIDUAL[J];
23  done
24 done
```

Algoritmo 3.2: Algoritmo de partida para a dissertação

Parte II

NÚCLEO DA DISSERTAÇÃO

4

OTIMIZAÇÃO DA VERSÃO SEQUENCIAL

Durante a modificação da versão sequencial não foram consideradas, inicialmente, melhorias na hierarquia de memória sendo dada ênfase especial ao ciclo mais interno existente na função *makeFlux* onde são calculados os fluxos nas bordas fronteiriças, onde a maior parte do trabalho está concentrado. Posteriormente foram realizadas melhorias na organização e acesso aos dados de modo a reduzir a quantidade de *cache misses* tornando assim o código amigável para a *cache*.

4.1 SIMPLIFICAÇÃO DO CÓDIGO

Uma boa simplificação do código é extremamente importante para que a quantidade de instruções seja reduzida assim como o acesso aos dados seja feito do melhor modo possível.

Foi analisado o código e foram identificadas possíveis modificações que o tornassem mais simples.

O *if* principal verifica se uma determinada borda era fronteira ou não, ou seja, se tinha o valor de *RIGHT* válido, através de uma análise mais cuidada verificou-se que as bordas fronteiriças se encontram no inicio dos vetores enquanto as outras no final, assim sendo, passou a ser calculada a quantidade de bordas fronteiriças e substituir a condição por dois ciclos, um que tratava das fronteiras e outro que tratava do resto.

O código referente às fronteiras contém outros *if's* condicionais que não foram alterados tendo em conta que a quantidade de fronteiras é pequena e o impacto no tempo de execução também o é.

O código para as não-fronteiras foi analisado com mais cuidado e foram efetuadas duas modificações. Em primeiro lugar era efetuado um produto de uma variável temporária em cada ciclo pelo comprimento de cada borda, constante a cada iteração, este valor passou a ser calculado à priori por outros vetores consoante o tipo de borda (tipo de fronteira e não-fronteira) eliminando assim um produto extra a cada iteração do ciclo interno.

Finalmente existia um último *if* no código que apenas verificava se a velocidade normal do fluxo para cada borda era positiva ou negativa e, atribuindo um valor inicial a cada fluxo por borda, esta instrução condicional foi substituída por uma instrução aritmética.

No final destas alterações foi possível obter um código sequencial mais simples, com uma média de 24 instruções por iteração do ciclo interno enquanto anteriormente essa média era de 38 e o seu pseudo-código pode ser observado em 4.1.

```
for N in totalCells do
   makeFlux(...,F,...){
      for I in 0..totalFrontiers do
         FRONTIER CALCULATION
      done
      for I in totalFrontiers..totalEdges do
         leftPhi = PHI[LEFT[I]];
         rightPhi = PHI[RIGHT[I]];
         (NORMAL_VELOCITY[I] < 0) ? phi_aux = rightPhi : phi_aux = leftPhi;
         current_edge = NORMAL[I] * phi_aux;
         current_edge -= CONSTANT_VALUES[I]*(rightPhi-leftPhi)
         G[LEFT[I]]+=current_edge;
         G[RIGHT[I]]-=current_edge;
      done
   };
   for J in totalCells do
      G[J]/=AREA[J];
      G[J]-=RESIDUAL[J];
   done
done
```

Algoritmo 4.1: Algoritmo sequencial final.

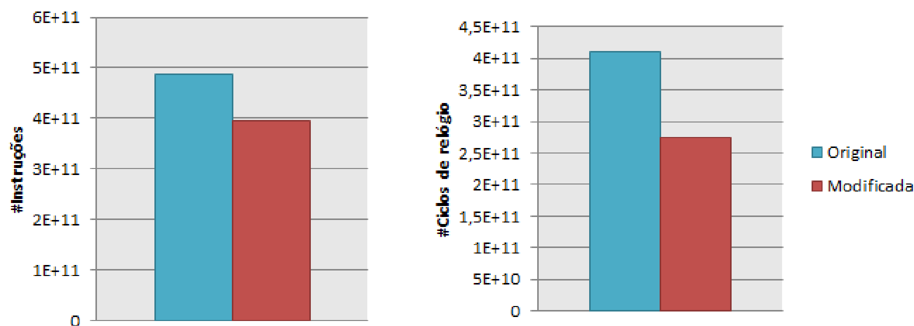


Figura 19.: Comparação das instruções e ciclos de relógio entre a versão original e a versão modificada.

Tal como é possível observar na figura 19 a nova versão reduziu com sucesso a quantidade de instruções e isso foi refletido na redução em 1/3 da quantidade de ciclos de relógio. Depois desta fase foram analisadas possíveis melhorias na hierarquia de memória.

4.2 MELHORIA PARA A HIERARQUIA DE MEMÓRIA

Após uma análise mais detalhada ao código foram identificadas possíveis melhorias no uso da hierarquia de memória com o objetivo de reduzir as taxas de *cache misses*.

Nesta altura o código apresentava quatro *cache misses* problemáticos causados pela mesma razão, o

vetor *G* era modificado consoante o identificador da célula da esquerda e da direita de uma borda a cada iteração assim como o vetor *PHI* era acedido do mesmo modo (linhas 7, 8, 12 e 13 do algoritmo 3.1). Foi analisado o vetor *RIGHT* e verificou-se que estava ordenado por valor crescente, traduzindo-se em acessos alinhados à memória em *G* e *PHI*, minimizando assim os *cache misses* causados pelos acessos a informação motivamos pela célula da direita. No entanto os acessos à informação causados célula da esquerda eram problemáticos, tendo em conta que os acessos a *G* e *PHI* por ela causados eram irregulares. Estes acessos causavam também 2 *cache misses* um em *G* e um em *PHI*. Assim sendo, em teoria, seria possível fundir estes dois vetores num vetor de estruturas em que cada elemento da estrutura teria um valor para *G* e outro para *PHI*. Deste modo quando ocorresse um *cache miss* num dos vetores o valor do outro vetor seria carregado automaticamente na linha para a memória *cache*. Assim, deixou de existir um dos *cache misses* tirando-se partido do alinhamento dos dados na memória, este comportamento encontra-se exemplificado na figura 20.

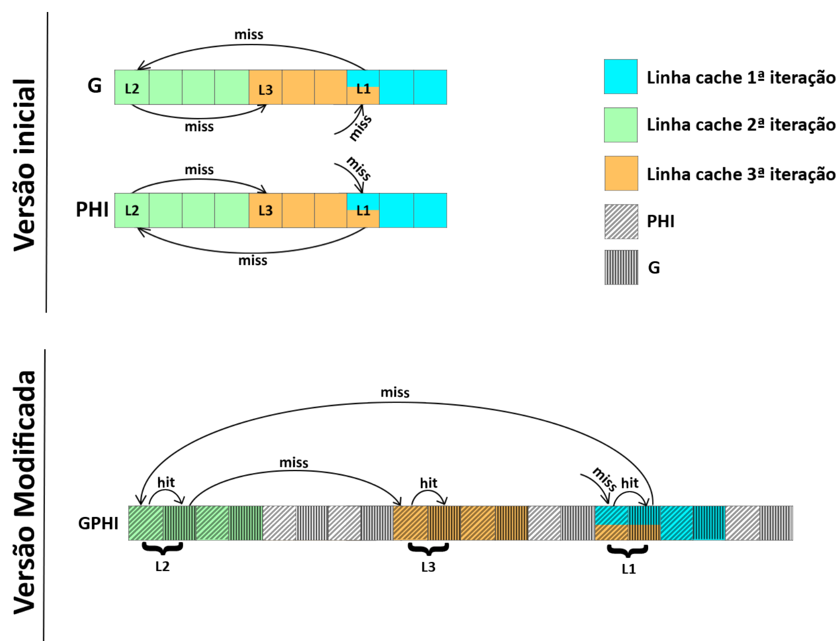


Figura 20.: Acessos à memória para G e PHI.

Depois de modificado o código foram efetuados alguns testes e retiradas as respetivas conclusões.

Como é visível na figura 21, tanto na *cahe L1* como na *cache L2* a quantidade de *misses* reduziu para 3/4 em relação à versão com vetores, no entanto para ter confirmação de que esta melhoria se tinha refletido nos tempos de execução testaram-se o total de ciclos de relógio e o total de instruções, tais resultados podem ser vistos na figura 22.

Apesar de uma melhoria significativa na quantidade de *cache misses* houve um aumento da quantidade de instruções. A cada iteração o vetor *G* é percorrido para os que seus valores sejam inicializados a 0, este processo era vetorizado na versão anterior, já nesta, tal não acontece porque os valores de *G* deixaram de estar contíguos na memória, por isso mesmo, a quantidade instruções aumentou,

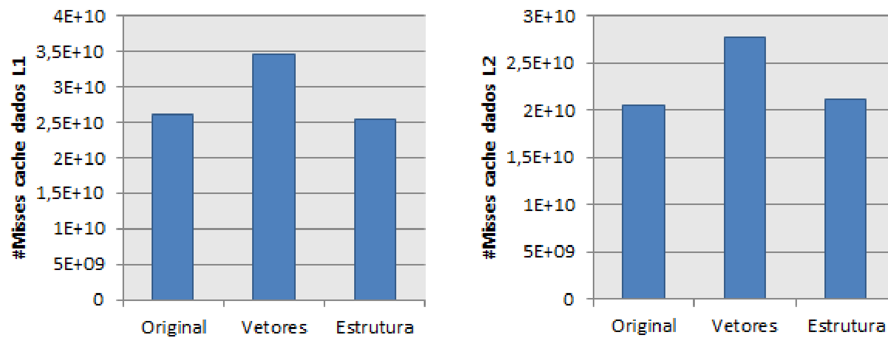


Figura 21.: Comparação dos *cache misses* entre a versão original e as novas versões.

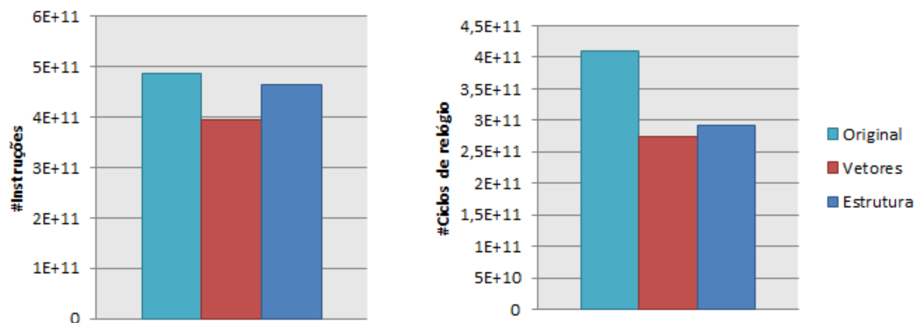


Figura 22.: Comparação das instruções entre a versão original e as novas versões.

refletindo-se diretamente no total de ciclos de relógio. Apesar disto esta versão continuava a ser melhor do que a original. Finalmente, é possível observar que a primeira versão, na qual se reduziu a quantidade de instruções, teve um maior número de *cache misses* do que a versão original, tal deve-se ao facto da redução de instruções ter aumentado a percentagem de instruções que requerem dados de memória.

Neste momento era possível concluir que, apesar da fusão do *G* e do *PHI* não ter bons resultados, a melhor solução passava pela melhoria no acesso a estes dois vetores. Era imperativo que dentro de uma iteração os acessos a *G* fossem o mais próximos possível, ou seja, que a diferença entre o valor de *LEFT* e *RIGHT* fosse minimizada e que o vetor *LEFT* estivesse ordenado tanto quanto possível. Deste modo seria possível obter ganhos na localidade espacial dos dados por estarem ordenados.

A abordagem seguinte consistiu na tentativa de diagonalizar a matriz de adjacência de modo a reduzir os *cache misses*, aproximando os índices da célula da esquerda aos da direita. Surgiu então como possível solução para este problema o algoritmo de Cuthill and McKee (1969), apresentado na secção 2.3.1.

Adaptando o algoritmo ao problema da *FVLib*, recorrendo à versão com vetores, foram renumeradas as células da malha. Foi escolhida uma célula inicial ao acaso e utilizando os apontadores para as 3 células vizinhas existentes na malha, introduzindo uma noção de grafo, ideal para aplicar o algoritmo em questão. Poucas modificações foram efetuadas ao algoritmo original de Cuthill and McKee

(1969), sendo a única relevante a remoção da sub ordenação de cada iteração, sendo apenas numeradas as células pela ordem que foram inseridas na lista.

Optou-se por não fazer esta sub ordenação porque o número de vizinhos é muito reduzido e, por isso mesmo, não seria muito problemática a ordem das células. No final a matriz de adjacência resultante deste processo, para o conjunto de dados pequeno, encontra-se representada na figura 23, onde é possível identificar, para diferentes células iniciais, a diferença entre larguras de banda entre o melhor (célula 385) e pior (célula 1067) casos e a mediana (célula 161).

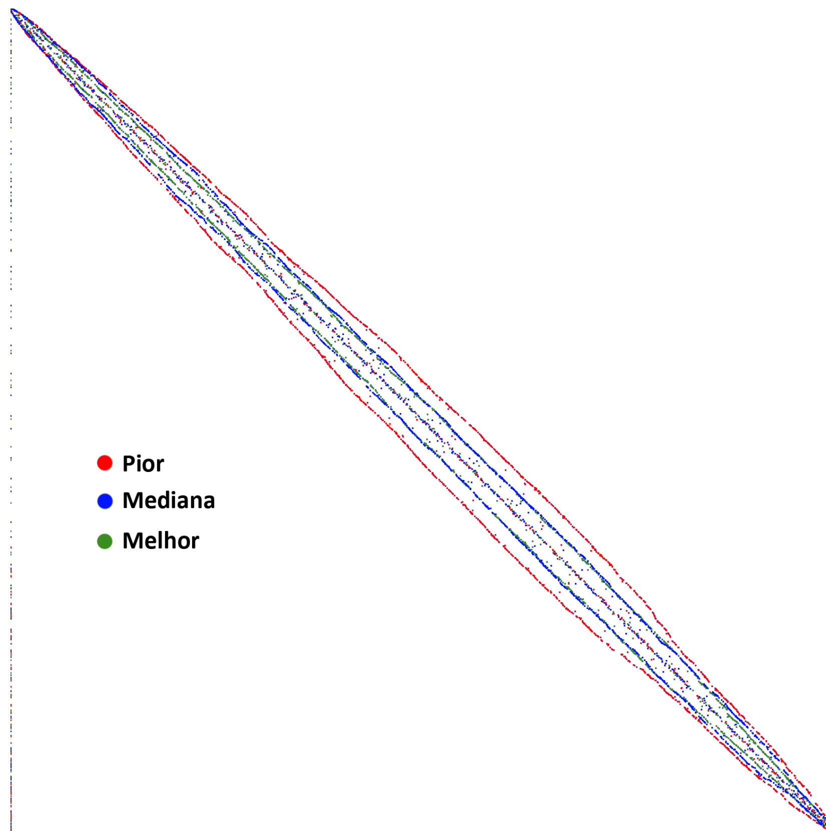


Figura 23.: Matriz de adjacência depois da renumeração.

Foi efetuada uma análise comparativa entre a largura de banda da matriz antiga e da nova, na qual foi considerada a largura de banda como sendo a média das distâncias entre cada não zero de uma linha e a sua diagonal. A relação entre a largura de banda e a célula inicial do algoritmo pode ser observada na figura 24.

É possível concluir que a escolha da célula inicial influência de forma reduzida a largura de banda obtida, assim sendo, a probabilidade de, escolhendo uma célula inicial ao acaso, obter uma largura de banda baixa é alta. A distribuição das larguras de banda da figura 24 pode ser vista na figura 25 e aproxima-se de uma distribuição beta, visível na figura 25.

Na figura 26 aparece associada a cada célula a largura de banda quando utilizada como inicial para o algoritmo de renumeração. Concluiu-se assim que a posição da célula inicial não tem influência

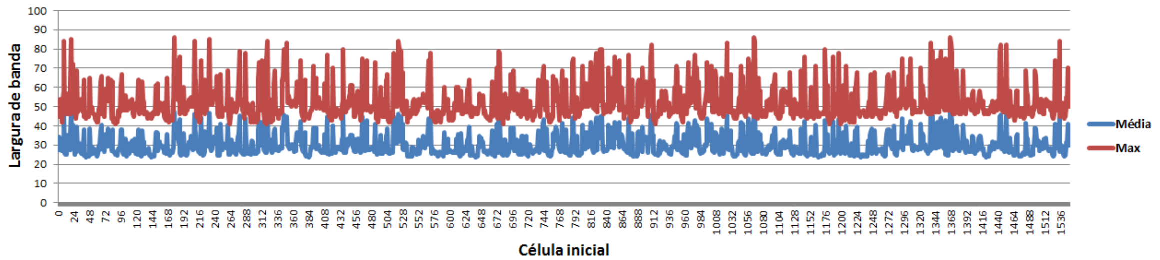


Figura 24.: Relação entre a largura de banda da matriz e a célula inicial do algoritmo.

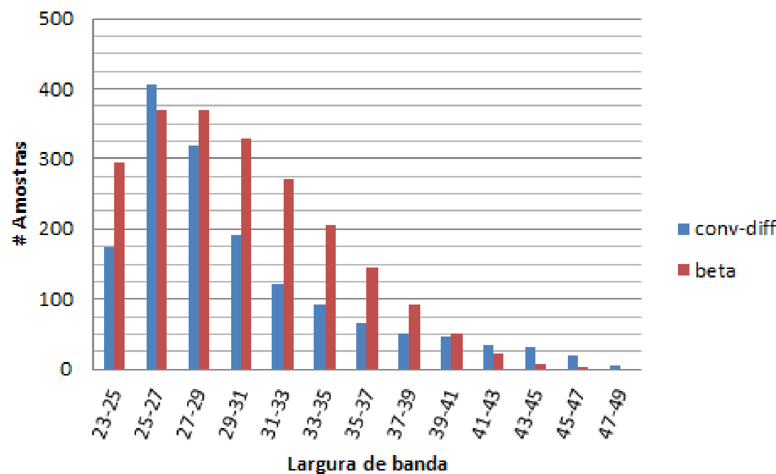


Figura 25.: Distribuição da média das larguras de banda.

direta na largura de banda da matriz.

Estes testes foram efetuados utilizando o conjunto de dados pequeno, no entanto, o comportamento é exatamente o mesmo utilizando um conjunto maior.

Outras modificações tiveram de ser efetuadas ao código para que os resultados se mantivessem válidos. Tendo em consideração que o valor mais baixo do *id* da célula direita diz respeito ao exterior da malha, nomeadamente o '-1' foi necessário ordenar as bordas por ordem crescente de *id* da célula direita. Este problema foi minimizado porque a matriz estava no formato COO. Tendo em conta que a matriz estava agora ordenada por colunas e com o objetivo de melhorar ainda mais os acessos à memória foi efetuada uma sub reordenação consoante o identificador da célula esquerda.

Ambos os algoritmos de ordenação são executados uma vez apenas por execução não tendo grande impacto no tempo de execução da aplicação.

O erro da versão inicial era $5,46 \times 10^{-2}$, depois de renumeradas as células e ordenados os vetores esse erro alterou ligeiramente. A relação entre o erro final e a célula inicial assim como a distribuição desse mesmo erro podem ser vistos na figura 27 e 28 respetivamente.

Depois de validados os resultados, para o maior conjunto de dados, foram efetuados testes comparativos, utilizando como célula inicial no algoritmo de renumeração aquela com largura de banda

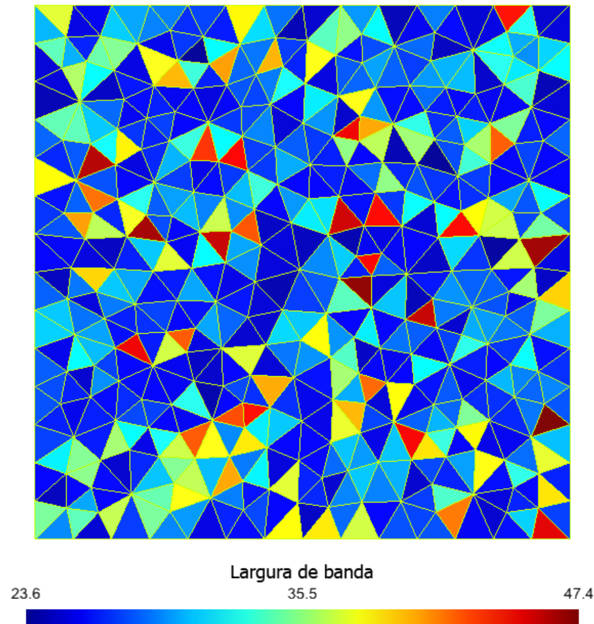


Figura 26.: Associação entre célula inicial e largura de banda.

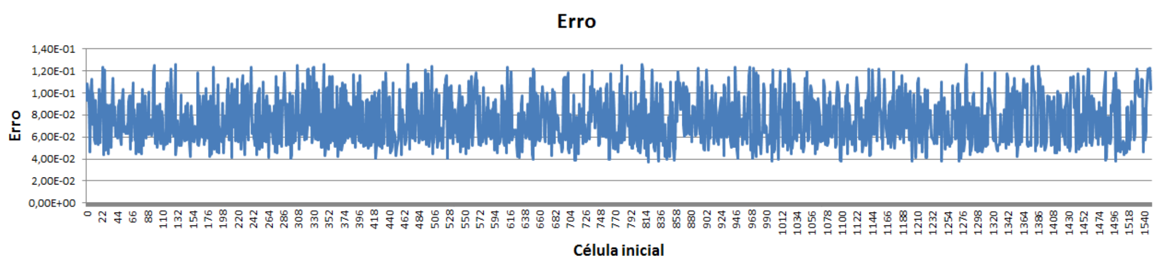


Figura 27.: Relação entre o erro e a célula inicial do algoritmo.

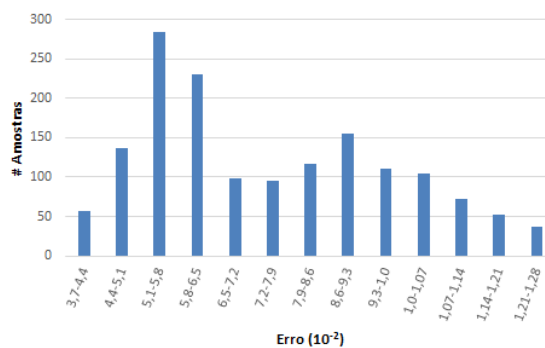


Figura 28.: Distribuição do erro.

mediana (célula 306). Foi tido em consideração a taxa de *cache misses*, total de instruções e o total de ciclos de relógio, as comparações podem ser observadas nas figuras 29 e 30. Deve ter sida em consideração a escolha da mesma célula para cada um dos testes para manter a coerência nos resulta-

dos.

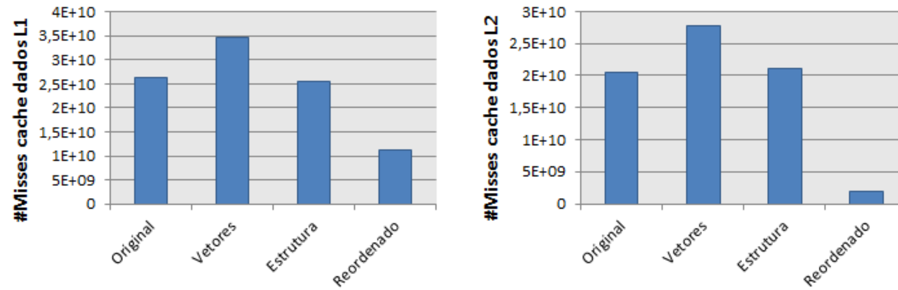


Figura 29.: Comparação dos cache misses entre a versão original e as novas versões.

A nova organização dos dados permitiu reduzir com sucesso os *cache misses* em todos os níveis da memória *cache* estando a maior presente na de nível 2, em que a quantidade de *cache misses* da versão com vetores *G* e *PHI* é aproximadamente 15 vezes maior do que a nova, com renumeração. Já no nível 1 essa redução foi de 3 vezes apenas.

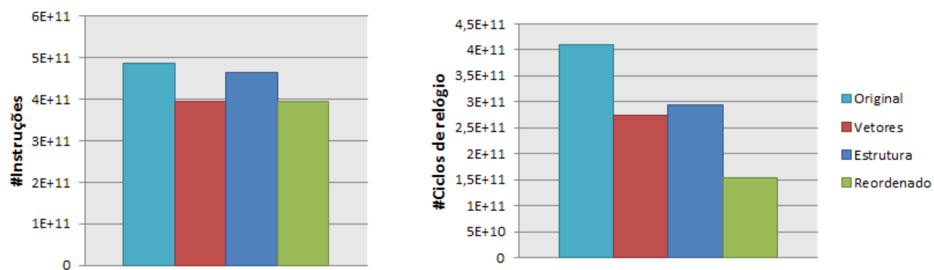


Figura 30.: Comparação das instruções entre a versão original e as novas versões.

Estas melhorias na quantidade de *cache misses* traduziram-se numa diminuição significativa na quantidade de ciclos de relógio, ou seja, numa redução do tempo de execução, visível na figura 30. Também na figura 30 é possível observar que a quantidade de instruções pouco alterou, relativamente à versão com vetores.

Tendo em consideração os resultados positivos avançou-se para a próxima fase, exploração de paralelismo.

5

EXPLORAÇÃO DE PARALELISMO

A *FVLib* recorre à utilização de apontadores para acesso a dados dentro e entre estruturas. Devido a esse tipo de acessos é difícil garantir que dados estão contínuos em memória e é difícil repartir dados por diferentes *threads* disponíveis numa máquina. O código foi paralelizado utilizando *OpenMP*, uma *API* para programação sistemas de memória partilhada.

Para paralelizar o código foram necessárias duas considerações, primeiro a divisão de trabalho e dados pelas diferentes *threads* e depois o acesso concorrente aos mesmos.

Para aplicar técnicas de paralelismo ao algoritmo foi necessário identificar qual o trabalho que podia ser dividido por diferentes *threads*.

5.1 PRIMEIRA VERSÃO

Tendo em conta que o núcleo do algoritmo itera sobre as bordas, isto é, o ciclo mais interno da aplicação percorre todas as bordas existentes no domínio e calcula o fluxo em cada uma, foram identificados quais os dados referentes a cada borda e célula utilizados em cada iteração e quais os dados que poderiam ser utilizados mais do que uma vez, em diferentes iterações. Qualquer tipo de dados que seja modificado em mais do que uma iteração requer um controlo de acesso para garantir que apenas uma *thread* o atualiza ao mesmo tempo. Depois de identificados esses conflitos foi possível dividir o cálculo dos fluxos das bordas por diferentes *threads*.

Os vetores *NORMAL_VELOCITY* , *F*, *RIGHT* e *LEFT* contêm informações sobre as bordas. Estes vetores nunca são modificados, sendo apenas utilizados para leituras e, por isso mesmo, não é necessário controlar acessos concorrentes aos mesmos.

O vetor G contém informação sobre o fluxo a cada iteração em cada célula, este G é atualizado a cada iteração com novos valores e os acessos ao mesmo são dependentes do valor de *LEFT[i]* e *RIGHT[i]* em que i é o identificador de uma determinada borda. Duas iterações diferentes podem aceder e modificar o valor de uma posição específica do vetor G e, por isso mesmo, é necessário aqui uma atenção especial, porque se duas *threads* diferentes atualizarem a mesma posição em simultâneo causam um erro, que torna os resultados inválidos (linhas 12 e 13 do algoritmo 4.1).

Em suma, os vetores referentes às bordas serão repartidos pelas diferentes *threads* enquanto o vetor referente às células será partilhado por várias *threads* levando assim ao conflito de acesso em *G*.

Inicialmente foi utilizada a diretiva *atomic* para garantir que nunca duas *threads* diferentes acediam a uma posição do vetor G em simultâneo.

Este controlo atómico no acesso a G mostrou-se custoso e, por isso mesmo, o algoritmo não escalou como pretendido, assim sendo, foi necessária uma divisão de trabalho e dados pelas *threads* mais cuidada.

Foi dividido o domínio do problema de modo a que nunca duas *threads* diferentes acedessem aos mesmos dados em G. Tirando partido da ordenação por colunas da matriz e do facto de esta, neste momento se encontrar próxima de uma matriz diagonal utilizou-se uma divisão por blocos e uma subdivisão desses mesmos blocos em partições, por colunas, até que fosse possível identificar um modo de calcular fluxos sobre bordas em paralelo garantindo que nunca duas *threads* atualizam a mesma posição do vetor G em simultâneo. Um esboço desta mesma divisão pode ser visto na figura 31.

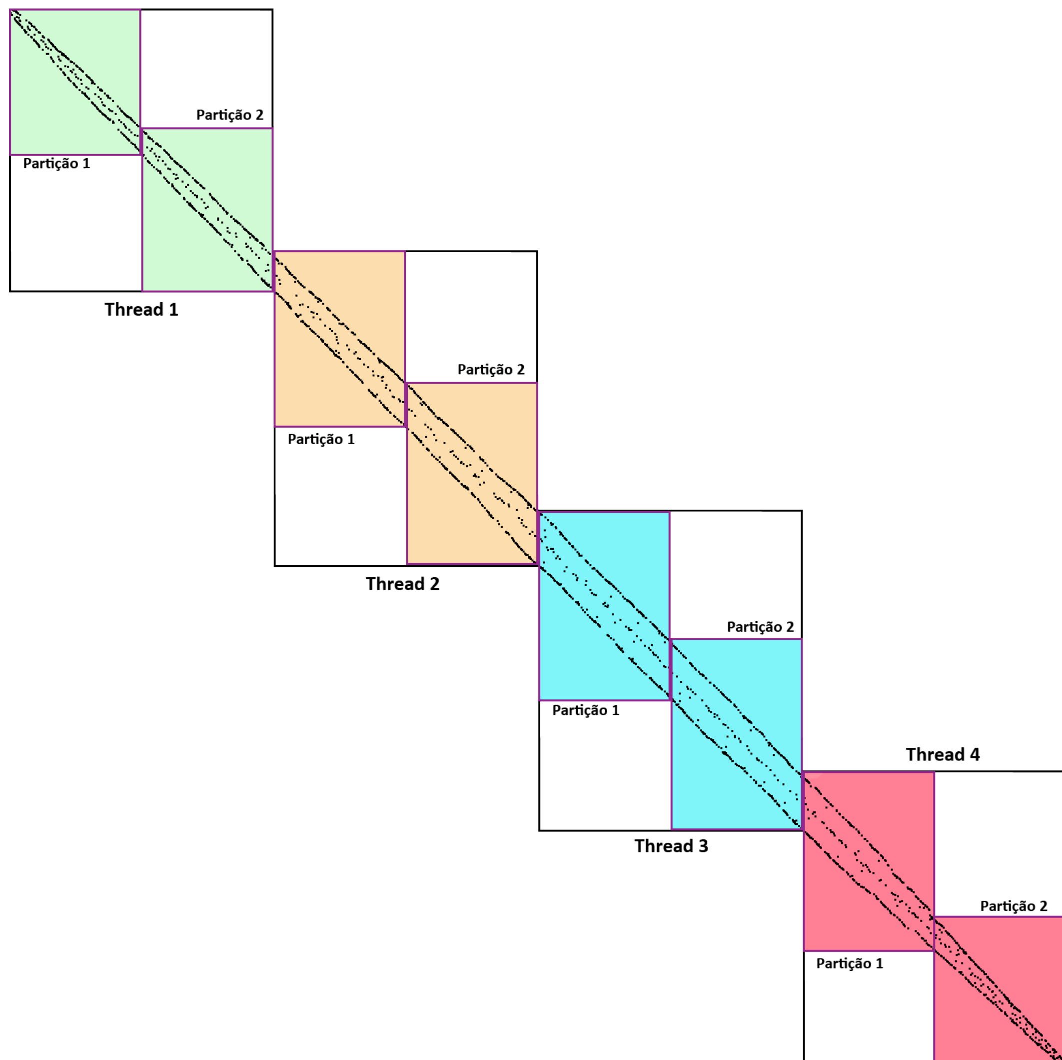


Figura 31.: Divisão da matriz para 4 *threads*.

As partições são executadas por ordem, isto é, as *threads* utilizam e atualizam os dados referentes a uma partição específica e só depois de acabarem o trabalho nessa partição passam à seguinte. Na figura 31, por exemplo, partições ímpares podem ser executadas em simultâneo garantindo que nunca duas *threads* modificam a mesma posição do vetor G.

Este modo de tratar os dados não é muito eficaz quando não há renumeração de células, tendo em conta que encontrar partições válidas, em que não haja concorrência aos dados, não é simples, quando os acessos a G estão ordenados esse problema é minimizado. Porque os dados estão organizados por coluna a divisão é feita por colunas.

Para resolver o problema da partição dos dados de cada *thread* foi criado um algoritmo que, de um modo autónomo, identifica possíveis partições e deteta possíveis conflitos. Caso um conflito seja detetado é incrementado o número de partições e o processo é repetido. Apenas quando não houver conflitos entre a primeira partição de todas as *threads* o algoritmo prossegue. Para o conjunto de dados pequeno são necessárias 3 partições, já para o conjunto de dados grande bastam 2.

A divisão do domínio em partições traduz-se na divisão dos vários vetores por diferentes *threads* e divisão dos dados de cada uma por partições tal como representado na figura 32. O vetor G tem zonas em que é utilizado por diferentes *threads*, essas mesmas zonas estão divididas por partições, assim, nunca duas *threads* atualizam o seu valor em simultâneo.

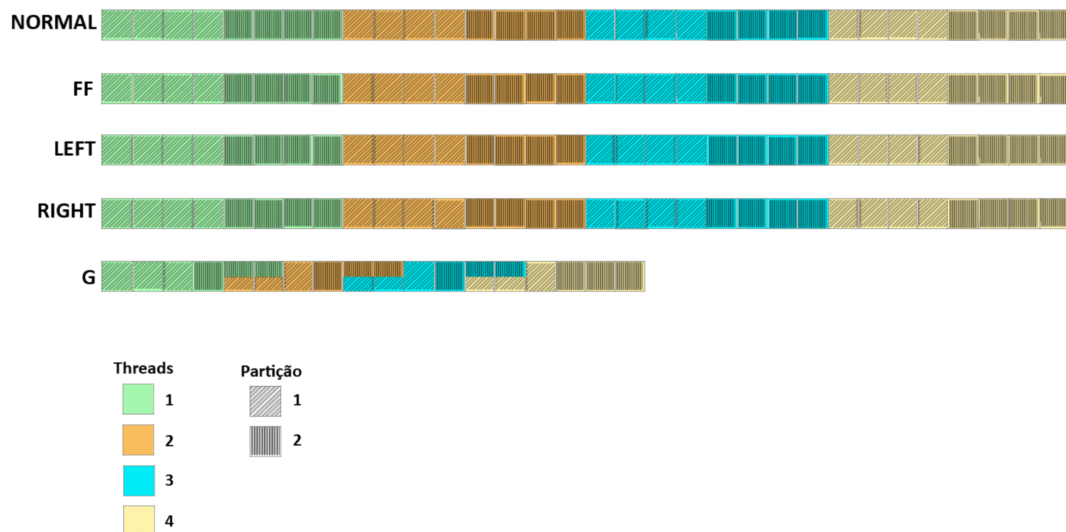


Figura 32.: Divisão do domínio do problema para *threads*.

Esta versão estava limitada principalmente pela reduzida percentagem de código paralelizado. A seguinte versão tem como objetivo resolver esse problema e aproveitar uma possível localidade temporal dos dados.

5.2 SEGUNDA VERSÃO

Foi criada uma versão do código em que ao invés de apenas estar paralelizada a função *makeFlux* se aplicaram outro tipo de primitivas ao código para que desde o início do cálculo de vetores *G* existisse paralelismo. O principal objetivo desta versão foi para além de aumentar a percentagem de paralelismo reduzir a quantidade de trabalho extra para criação e gestão de *threads* por parte do *OpenMP*. Passou a ser necessária uma sincronização extra entre *threads* para garantir que toda a execução do algoritmo se mantivesse válida, nomeadamente entre mudanças de partições e chamadas da função *makeResidual*.

A cada chamada da função *makeResidual* é criado e calculado um novo vetor *G* com os fluxos. Tendo em conta que os vetores com informação sobre as bordas se mantêm inalterados durante toda a execução é possível explorar localidade temporal na memória caso sejam estes utilizados mais do que uma vez por chamada da função.

Foi criada uma versão do código onde é calculado mais do que um vetor *G*. Assim sendo, numa iteração, no ciclo mais interno da função *makeFlux*, são criados vários valores de fluxo, tantos quanto a quantidade de vetores *G* em simultâneo desejados.

Por exemplo, para um caso de três vetores *G* em simultâneo, na primeira chamada da função *makeResidual*, serão criados os vetores *G1*, *G2* e *G3*. Assim, numa única iteração *N* do ciclo mais interno da função *makeFlux* serão utilizados três vezes o valor de *NORMAL_VELOCITY[N]*, *F[N]*, *LEFT[N]* e *RIGHT[N]*. Outro dos objetivos desta versão é reduzir o custo de sincronização entre chamadas da função *makeResidual* tendo em conta que as mesmas seriam reduzidas tanto quanto maior fosse a quantidade de vetores *G* calculados.

Os resultados da figura 33 dizem respeito às diferentes versões paralelas com a criação de um vetor *G* a cada chamada da função *makeResidual*, onde é possível observar a limitação do *self-speedup* da versão particionada, mesmo quando o seu tempo de execução é melhor do que as versões anteriores. Os resultados da figura 34 comparam a melhor versão paralela sem preocupação especial no ganho na localidade temporal dos dados com a versão em que eram criados mais do que um vetor *G* por iteração de *makeResidual* para uma e 16 *threads*.

À medida que foram criadas novas versões também o tempo de execução das mesmas reduziu, sendo a melhor versão paralela aquela em que a maior parte do código era paralelizado, onde apenas um vetor *G* é criado a cada chamada de *makeResidual* e em que as células era reordenadas e repartidas por cada *thread* de modo a não causar acessos simultâneos por duas threads diferentes aos mesmos dados. Nesta versão para além dos tempos de execução terem diminuído consideravelmente refletindo-se pelos resultados por segundo também o *speedup* relativo à sua versão com um *thread* foi consideravelmente maior para um maior número de *threads*.

Comparando a versão de 1 e 16 *threads* com a versão em que se tinha uma especial atenção aos aproveitamentos temporais dos dados na memória existem dois cenários que se podem observar, num primeiro em que apenas é utilizado um *thread*, apesar de, à medida que se aumenta a quantidade de

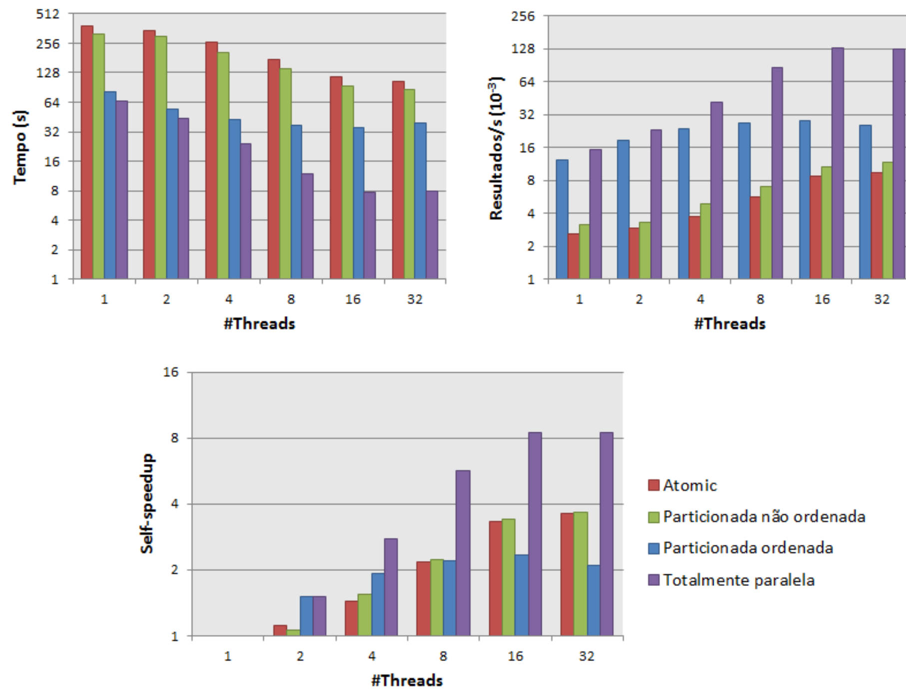


Figura 33.: Comparação entre tempo de execução, resultados por segundo e *speedup* entre versões paralelas.

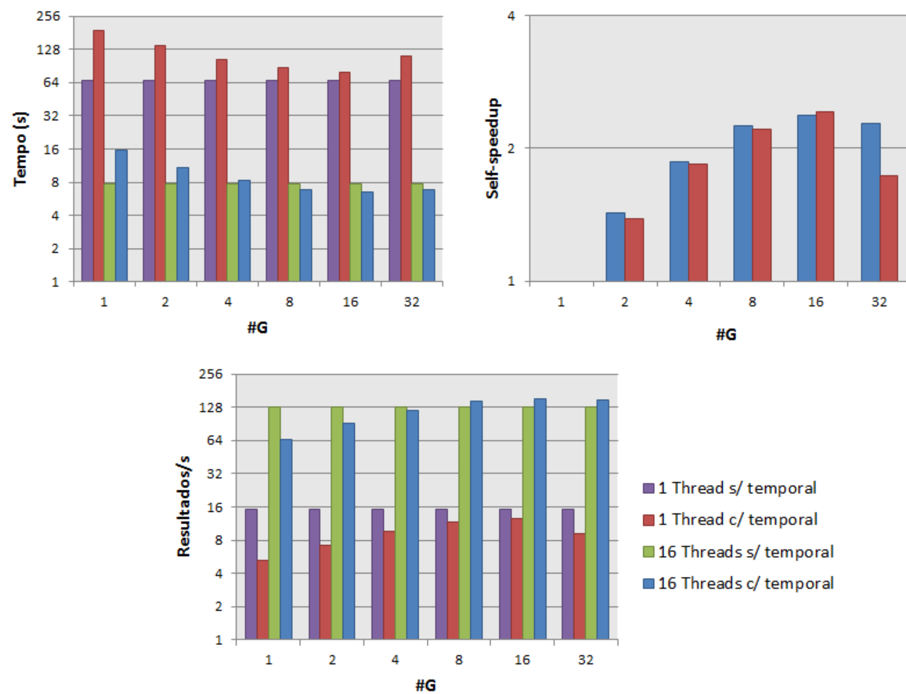


Figura 34.: Comparação entre tempo de execução, resultados por segundo e *speedup* entre versões paralelas.

G's por chamada da *makeResidual* o tempo de execução diminui mas nunca chega a atingir valores inferiores à versão anterior do código para o mesmo número de *threads*, o trabalho extra para calcular estes vetores não compensa o ganho. Com 16 *threads* o resultado é positivo, pois com o cálculo de 8, 16 e 32 vetores *G* por chamada da *makeResidual* o tempo de execução consegue ser inferior à versão em que não se tinha especial atenção aos ganhos temporais.

O algoritmo 5.1 contém pseudo-código final da dissertação já com introdução de paralelismo e aproveitamento da localidade temporal dos dados.

```
1  VECTOR DIVISIONS HAS PARTITION INDEX FOR EACH PARTITION
2  TOTALITERATIONS VALUE REPRESENTES THE AMMOUNT OF G VECTOR CALCULATED PER
       ITERATION
3
4
5  #parallel
6  {
7  for(int IT=0; it < TOTALCELLS; IT++){
8     #barrier
9     makeResidual(...){
10        #master
11        {
12           FRONTIER CALCULATION
13        }
14        #barrier
15        for(int PARTITIONS = 0; PARTITIONS < TOTALPARTITIONS; PARTITIONS++) {
16           int END=THREAD_ID_ACT*TOTALPARTITIONS+1;
17           for(int I = DIVISIONS[END-1]; i < DIVISIONS[END]; I++) {
18              int GPOS=0;
19              for(int NG = 0; NG < TOTALITERATIONS; NG++){
20                 (LEFT[i]==IT) ? leftPhi=1:leftPhi=0;
21                 (RIGHT[i]==IT) ? rightPhi=1:rightPhi=0;
22                 (NORMAL_VELOCITY[I] < 0) ? phi_aux = rightPhi : phi_aux = leftPhi;
23                 current_edge = NORMAL[I] * phi_aux;
24                 current_edge -=CONSTANT_VALUES[I]*(rightPhi-leftPhi);
25                 G[RIGHT[i]+GPOS]-=current_edge;
26                 G[LEFT[i]+GPOS]+=current_edge;
27                 GPOS+=TOTALCELLS;
28                 IT++;
29              }
30              IT-=TOTALITERATIONS;
31           }
32           END++;
33           #barrier
34        }
35     };
36     AUXILIARY CALCULATIONS
```

```
37 }
38 }
```

Algoritmo 5.1: Pseudo-código final da dissertação

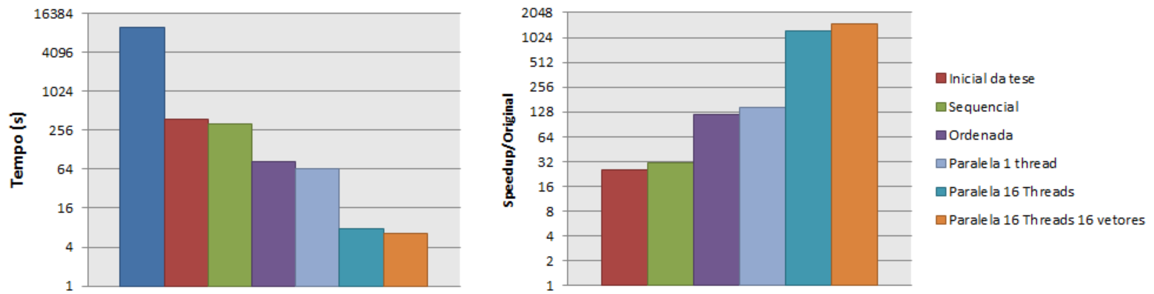


Figura 35.: Comparação entre tempo de execução das várias versões da tese e o speedup relativamente à versão original do código.

Na figura 35 estão presentes as diferentes versões desde a primeira versão fornecida para o caso de estudo, a primeira versão utilizada na tese e as versões modificadas, é também possível observar o *speedup* relativamente à primeira versão fornecida, ainda com apontadores.

6

CONCLUSÃO E TRABALHO FUTURO

O objetivo desta dissertação foi otimizar e paralelizar um algoritmo de convecção-difusão. Para isso foi utilizado como caso de estudo um algoritmo que utilizava uma biblioteca de volumes finitos (*Finite Volume Library*).

Realizou-se uma pesquisa sobre o estado da arte dos algoritmos e otimizações existentes em algoritmos do tipo *stencil*, tendo em conta que o comportamento dos mesmos era semelhante ao comportamento do algoritmo a otimizar. Estudou-se o modo como os algoritmos existentes aproveitavam a localidade temporal e espacial dos dados assim como o modo que eram divididos pelos vários núcleos de uma máquina para uma paralelização eficaz.

Durante o desenvolvimento da dissertação simplificou-se o código e reduziu-se a sua quantidade de instruções. Foi analisado e modificado o modo como o código estava a aceder e a lidar com dados e quais deles eram supérfluos. Por fim criou-se uma versão paralela do algoritmo que beneficiou de todas as modificações efetuadas anteriormente.

Todas as simplificações permitiram obter resultados satisfatórios. A quantidade de *cache misses* reduziu em todas as *caches*. Também a quantidade de instruções reduziu substancialmente e isso refletiu-se nos tempos de execução obtidos.

A reorganização da matriz foi um fator importante que facilitou a paralelização, para além disso as melhorias na versão sequencial refletiram-se na versão paralela na medida em que esta escalou melhor do que versões anteriores.

A renumeração, reodenação e divisão das células e bordas dos diferentes vetores foram os maiores desafios da dissertação, no entanto todo o esforço compensou, tendo sido obtidos bons resultados.

O código paralelo da última versão desta dissertação possui bastante sincronização entre *threads*, esta sincronização pode ser explorada no futuro, de modo a ser reduzida o máximo possível, diminuindo ainda mais o tempo de execução da versão paralela.

Futuras investigações podem ser efetuadas para portar este código para um sistema tanto de memória distribuída como para código híbrido (*GPU* e *CPU*).

BIBLIOGRAFIA

Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA*. ACM, 2009. ISBN 978-1-60558-606-9.

S. Clain, Steven Diot, Raphael Loubère, Gaspar Machado, Rui Ralha, and Rui M.S.Pereira. Very high-order finite volume method for one-dimensional convection diffusion problems. In *Mathematical Models for Engineering Science, Dezembro 2011, Puerto de la Cruz, Tenerife, Espanha*, 2011.

S. Clain, G. J. Machado, and R. M. S. Pereira. A finite volume scheme for the convection-diffusion system using the polynomial reconstruction operator. In *Conferência Nacional sobre computação Simbólica no Ensino e na investigação, 2-3 Abril, Lisboa, Portugal*, 2012.

E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.

Theodoros Gkountouvas, Vasileios Karakasis, Kornilios Kourtis, Georgios I. Goumas, and Nectarios Koziris. Improving the performance of the symmetric sparse matrix-vector multiplication in multi-core. In *IPDPS*, pages 273–283. IEEE Computer Society, 2013. ISBN 978-1-4673-6066-1.

Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2.

Kornilios Kourtis, Vasileios Karakasis, Georgios I. Goumas, and Nectarios Koziris. Csx: an extended compression format for spmv on shared memory systems. In Calin Cascaval and Pen-Chung Yew, editors, *PPOPP*, pages 247–256. ACM, 2011. ISBN 978-1-4503-0119-0.

X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proc. of ICS*, 2013. ISBN 978-1-4503-2130-3.

Thibaut Lutz, Christian Fensch, and Murray Cole. Partans: An autotuning framework for stencil computation on multi-gpu systems. *TACO*, 9(4):59, 2013.

Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, page 11, 2011.

Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *SC*, pages 1–13. IEEE, 2010. ISBN 978-1-4244-7559-9.

Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache oblivious parallelograms in iterative stencil computations. In Taisuke Boku, Hiroshi Nakashima, and Avi Mendelson, editors, *ICS*, pages 49–59. ACM, 2010. ISBN 978-1-4503-0018-6.

Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing CUDA performance in 3d stencil methods with annotated C. In *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011*, pages 214–224, 2011.

Ivan Šimeček. Performance aspects of sparse matrix-vector multiplication. *Acta Polytechnica*, 2009. ISSN 1210-2709.

Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC ’07, pages 38:1–38:12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3.

Parte III

APENDICES

AMBIENTE EXPERIMENTAL

Todos os testes experimentais foram efetuados utilizando o *SeARCH*, um *cluster* existente no departamento de informática da Universidade do Minho. Este *cluster* tem várias máquinas disponíveis das quais foi usada uma máquina com as características da tabela 2.

O sistema operativo utilizado foi o CentOS versão final 6.3 com o compilador *g++ (GCC)* 4.9.0 e, para obter informações dos contadores de *hardware*, foi utilizado o *PAPI* na versão 5.3.2.0.

CPU	
Fabricante	Intel
Microarquitectura	Ivy Bridge
Modelo	Intel Xeon
Referência	E5-2695 v2
Frequência de relógio	2400 MHz
#Núcleos	2 (CPUs) x 12
#Threads/núcleo	2
Tecnologia vetorial	AVX (256 bits)
Memória	
Tamanho	64 GB
Tamanho da L1	12 x 32 KB (D) + 12 x 32 KB (I) (por CPU)
Tamanho da L2	12 x 256 KB (por CPU)
Tamanho da L3	30 MB partilhados

Tabela 2.: Características da máquina utilizada