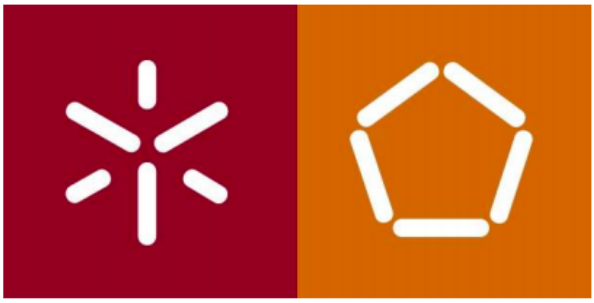

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Algoritmos Paralelos

Solução Numérica para a Equação de Poisson

Trabalho Prático 2

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

07/05/2021
Braga

Índice

1 Introdução

Para o segundo trabalho prático da cadeira de Algoritmos Paralelos, do perfil de Computação Paralela e Distribuída, do Mestrado Integrado em Engenharia Informática, foi-nos proposta a implementação de uma resolução numérica para a Equação de Poisson em C, utilizando os métodos de Gauss Seidel e de Sobre-relaxação Sucessiva. Mais ainda, estudaremos as possíveis otimizações em termos de paralelização, utilizando *OpenMP* e analisaremos os ganhos que obtivemos.

2 Equação de Poisson

Comecemos por introduzir o problema em mão. O objetivo deste trabalho é implementar métodos para resolver equações diferenciais parciais, ou PDEs (*partial differential equations*), utilizando um **Método de Diferenças Finitas**. Este método involve converter a PDE numa equação matricial. Em particular, viramos a nossa atenção para o problema da **Distribuição de Calor no Estado de Equilíbrio**, em que o nosso objetivo é determinar a temperatura de um ponto numa placa de largura L de um material condutor de calor no seu estado de equilíbrio, sabendo as coordenadas x e y do ponto na placa.

Este sistema pode ser representado com a *Equação de Poisson*:

$$\frac{\partial^2 u}{\partial x}(x,y) + \frac{\partial^2 u}{\partial y}(x,y) = g(x,y)$$

Para podermos resolver esta equação numericamente, discretizamos o espaço correspondente à placa condutora num conjunto finito de pontos, e computamos $u(x,y)$, ou $u(i,j)$ para cada um desses pontos.

Olhando para a seguinte imagem, podemos ver uma discretização do espaço sob a forma de uma malha (*mesh*) de pontos 6x6.

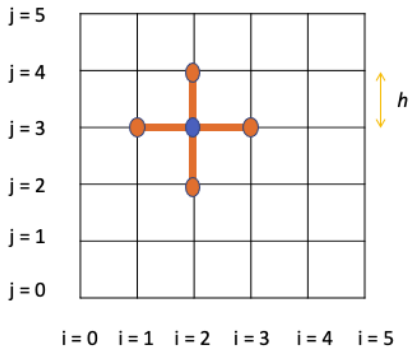

Figure 1: Discretização do Espaço

Na figura 1, fizemos questão de frizar um ponto aleatório da malha, o ponto de coordenadas (2,3), a azul, de modo a podermos exemplificar como calcular um dos pontos. Para este caso, para obter $u(2,3)$, teríamos de analisar os pontos vizinhos, a laranja, pelo que podemos ver que o algoritmo que utilizaremos é um *stencil* de cinco pontos. Ficaríamos então com:

$$u(2,3) = \frac{u(2,2)+u(1,3)+u(3,3)+u(2,4)}{4}$$

Podemos generalizar esta igualdade de modo a definir a regra para o cálculo de cada um dos pontos:

$$u(i,j) = \frac{u(i,j-1)+u(i-1,j)+u(i+1,j)+u(i,j+1)}{4}$$

Obviamente, não aplicaremos esta fórmula às bordas da malha, pois essas são-nos conhecidas. Assim, ficamos com um **sistema de equações lineares**, que poderemos facilmente resolver com um de vários métodos que veremos mais à frente. Como a matriz que representa o sistema é uma **matriz esparça**, não precisamos de guardar a matriz em memória, apenas teremos de guardar a malha (que, em termos de C, corresponderá a uma matriz de *doubles*, sob a forma de um duplo apontador).

Um aspeto que é importante mencionar é que o tempo não é um fator no sistema. Isto deve-se ao facto que o nós apenas analisamos o sistema uma vez que ele chega ao seu estado de equilíbrio, pelo que, para todos os efeitos, estamos a calcular a temperatura para um sistema que cujas características não variam com o tempo.

2.1 Métodos Iterativos de Resolução de Sistemas de Equações Lineares

Agora que reduzimos o problema a um sistema de equações lineares, precisamos de procurar um método para o resolver.

Existem vários tipos de métodos de resolução de sistemas de equações, dos quais podemos destacar os métodos diretos, como o Método de Eliminação de Gauss, e os **métodos iterativos**, os quais vamos utilizar neste projeto e explicar com um pouco mais detalhe.

Nos métodos iterativos, de um modo geral, quando queremos resolver um sistema do tipo:

$$Ax = b$$

Começamos por escolher uma aproximação inicial, $x^{(0)}$, e atualizamos este valor inicial sucessivamente até este se encontrar dentro de um domínio aceitável, pelo que aí o método para.

Em geral, os métodos passam por um processo de escrever o sistema sob uma forma equivalente

$$x = Ex + f$$

sendo que, a partir de $x^{(0)}$, geramos uma sequência de aproximações $\{x^{(k)}\}$, iterando:

$$x^{(k+1)} = Ex^{(k)} + f$$

Sendo que E e f são matrizes obtidas a partir de A e b, de uma forma diferente consoante o algoritmo.

Um fator interessante de notar é que a matriz A nunca é alterada ao longo das iterações. Isto implica que a matriz A pode ser tratada como uma caixa negra, nem sendo necessário guardar a mesma em memória, especialmente quando falamos de matrizes esparças, como é o caso deste problema.

Em termos de convergência dos métodos iterativos, os métodos convergem se o valor absoluto dos valores próprios de E forem menores do que 1. Como a matriz E é obtida de formas diferentes para métodos diferentes, a mesma matriz A pode convergir num método e divergir no outro. No entanto, a convergência do método não é dependente de $x^{(0)}$, ou seja, da aproximação inicial.

2.1.1 Método de Jacobi

Em primeiro lugar faremos uma pequena análise do método de Jacobi que, apesar de não ser o foco da análise deste trabalho, é o ponto de partida para a explicação dos métodos que o são.

A fórmula geral que temos de aplicar para atualizar os pontos da *mesh* em cada iteração no método de Jacobi é a seguinte:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=0}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

De uma forma mais simples, em cada iteração utilizamos os valores da última iteração para obter o valor atualizado de $x_i^{(k+1)}$, que, no contexto do nosso problema, corresponde a $u(i,j)$.

Um fator interessante sobre a convergência deste algoritmo é o facto que a matriz A ser diagonal dominante é suficiente para que o algoritmo convirja.

2.1.2 Método de Gauss-Seidel

O método de Gauss-Seidel é muito semelhante ao método de Jacobi.

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=0}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

A grande diferença entre os dois métodos é que este segundo utiliza na iteração $k+1$ elementos já calculados na iteração $k+1$, em vez de apenas utilizar elementos da iteração k, como acontece no primeiro caso. Isto tem várias vantagens, sendo que a maior é que a convergência do método é consideravelmente mais rápida na maioria dos casos, quando comparada com a do método de Jacobi. Outra vantagem que isto tem é que, enquanto que o método de Jacobi, dum ponto de vista computacional, obriga-nos a guardar em memória a matriz $x^{(k+1)}$ e a matriz $x^{(k)}$, no método de Gauss-Seidel apenas precisamos de guardar uma matriz, que será mais ou menos uma combinação das duas.

2.1.3 Método de *Successive Over Relaxation*

Continuando a analisar o método de Gauss-Seidel, nós podemos escrevê-lo da seguinte forma:

$$x_i^{(k+1)} = x_i^{(k)} + \frac{1}{a_{ii}}\left(b_i - \sum_{j=0}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

$$\Leftrightarrow x_i^{(k+1)} = x_i^{(k)} + \delta_i^{(k)}$$

A este $\delta_i^{(k)}$ que surge nesta equivalência chamamos termo de correção, pois corresponde à diferença entre duas iterações, que levará eventualmente à aproximação final da solução.

O método S.O.R. parte do mesmo princípio que o método de Gauss-Seidel, mas, a este termo de correção, adiciona um parâmetro de relaxação, dito ω.

$$x_i^{(k+1)} = x_i^{(k)} + \omega \times \delta_i^{(k)}$$

Pelo que ficamos com:

$$x_i^{(k+1)} = (1-\omega) \times x_i^{(k)} + \omega \times \frac{1}{a_{ii}}\left(b_i - \sum_{j=0}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

A teoria por trás disto dita que, com este parâmetro de relaxação, o processo de convergência é mais rápido que no método de Jacobi e de Gauss-Seidel, pois, em cada "salto" que é dado, "vamos um pouco mais longe" e aproximamo-nos mais rápido da solução.

Para que este método funcione, no entando, o ω deverá ser:

$$1 < \omega < 2$$

Se isto não se verificar, a convergência ou será muito mais lenta ou até poderá não convergir de todo. O valor ótimo de ω deverá ser próximo de 1.6, mas o valor ideal, para um dado $N - 1 = \frac{L}{h}$, será:

$$\omega = \frac{2}{1+sin(\frac{\pi}{N-1})}$$

2.2 Ordenação Lexicográfica vs. *Red-Black*

Como descrevemos anteriormente, o algoritmo requer um varrimento de uma matriz que representa a *mesh* da placa condutora cujo estado de equilíbrio queremos simular. Para este efeito, o algoritmo não requer que façamos o varrimento por uma ordem específica. Por esta razão, utilizamos dois tipos de varrimento: por ordem lexicográfica e por ordem *Red-Black*.

A **ordenação lexicográfica** é a ordem de varrimento mais óbvia para nós, em que a matriz é varrida da esquerda para a direita e de cima para baixo. Para uma implementação sequencial de um algoritmo, este varrimento é plenamente aceitável. No entanto, como vamos ver mais à frente, apresenta problemas nas implementações paralelas. Esta ordenação também é melhor em termos de utilização ao máximo de uma

linha de *cache* em comparação à ordem *red-black*, uma vez que faz acessos contíguos a posições de memória contíguas, aproveitando o princípio de localidade espacial.

A **ordenação *Red-Black***, por outro lado, obriga a que hajam dois varrimentos à matriz. Olhando para a matriz como um tabuleiro de xadrez, com algumas casas pretas e algumas casas brancas (ou vermelhas), no primeiro varrimento apenas processamos as casas pretas e no segundo varrimento só processamos as casas brancas. Em termos de acessos à memória, este varrimento é consideravelmente pior, uma vez que , para cada linha de *cache* que vamos buscar, que irá ter oito *doubles* (assumindo que uma linha de *cache* tem 64 *bytes*), apenas quatro serão usados. No entanto, este varrimento é consideravelmente mais amigável para paralelização, como vamos ver mais à frente.

Em termos de convergência do algoritmo quando olhamos para os dois varrimentos diferentes, em princípio não deverá haver uma diferença significativa, pelo que ambos terão convergência semelhante.

2.3 Erro de Truncatura

A Equação de Poisson é uma PDE de segunda ordem, pelo que o será definida pelas seguintes fórmulas:

$$\frac{\partial f}{\partial x}(x,y) = \frac{f(x+\frac{h}{2},y)-f(x+\frac{h}{2},y)}{h} + O(h^2)$$

$$\frac{\partial f}{\partial y}(x,y) = \frac{f(x,y+\frac{h}{2})-f(x,y+\frac{h}{2})}{h} + O(h^2)$$

Como podemos observar nestas duas fórmulas, em ambas é adicionado um termo de correção $O(h^2)$, que corresponde ao **erro de truncatura**. Em contraste às fórmulas de primeira ordem, em que o termo de correção é $O(h)$, as fórmulas de segunda ordem demonstram um erro de truncatura muito mais baixo, uma vez que $h < 1$, sempre. Em prática, isto vai implicar que a tolerância que deveremos dar para o término do algoritmo é menor, o que se traduzirá numa solução mais próxima da real.

A tolerância que deveremos escolher para cada um dos casos será aproximadamente:

$$TOLERANCE = h^2 \Leftrightarrow TOLERANCE = (\frac{L}{N-1})^2$$

Assumindo, como faremos daqui a diante, que $L = 1$, ficamos com:

$$TOLERANCE = \frac{1}{(N-1)^2}$$

A diferença entre N e $N - 1$ é minúscula para a maioria dos casos de estudo, especialmente quando olhamos para o seu inverso, pelo que optamos por simplificar os cálculos para:

$$TOLERANCE = \frac{1}{N^2}$$

2.4 Complexidade

A complexidade dos três métodos é relativamente simples de analisar. Assumindo que trabalhamos com uma placa quadrada de largura *L*, com uma *mesh* com um dado espaçamento h entre cada ponto em qualquer um dos eixos, como podemos ver na figura 1, e que o algoritmo terá de percorrer a matriz *iter* vezes, ficamos com complexidade limitada por:

$$O((\frac{L}{h} - 2)^2 * iter)$$

Na prática, quando falamos de $(\frac{L}{h} - 2)^2$ estamo-nos a referir ao tamanho da matriz que representa a malha de pontos, que em cada iteração do algoritmo é percorrida no seu todo exceto nas bordas, uma vez que os pontos nas bordas não precisam de ser atualizados.

Em princípio, de partida não sabemos o valor de *iter*, mas, teoricamente é calculácel visto que o algoritmo é plenamente determinístico, pelo que, para uma mesma aproximação inicial, teremos sempre o mesmo número de iterações.

2.5 Caso de Paragem

Para aplicar um caso de paragem para o algoritmo atingir a terminação, para os três casos, utilizamos o valor absoluto da máxima diferença entre as matrizes $x^{(k)}$ e $x^{(k-1)}$, chamadas w e u respetivamente, e comparamos este valor à tolerância que escolhemos, ficando:

$$Max(|U - W|) < TOLERANCE$$

Este cálculo trará alguns problemas inicialmente devido à complexidade que adiciona ao algoritmo, mas, como veremos mais à frente, conseguimos eventualmente tratar deles.

2.6 Escalabilidade

Em termos de escalabilidade, estas implementações de uma solução para a equação de Poisson são excelentes. Os algoritmos são leves, de um ponto de vista computacional, uma vez que apenas englobam aritmética simples, e de um ponto de vista de memória, também são bons, uma vez que, como referimos antes, não precisamos de guardar a matriz com as equações lineares que queremos resolver, tendo apenas de guardar a malha de *input* em memória. Assim, estas implementações de resoluções para sistemas de equções lineares são ideias para lidar com sistemas com muitas variáveis, conseguindo resolver sistemas com centenas ou até milhares de variáveis em tempo útil, especialmente o método S.O.R., que converge mais rápido.

Apesar disto, o tempo de execução ainda é considerável, uma vez que o número de iterações tem tendência a ser elevado, em particular para o método de Gauss-Seidel, daí o tamanho das matrizes *input* que utilizamos, como exporemos mais à frente.

3 Implementação

O ponto fulcral deste trabalho, tal como aconteceu com o primeiro trabalho prático, é criar uma implementação em C ou C++ do código em MATLAB dado pelo professor Rui Ralha. A utilização de uma destas duas linguagens justifica-se devido à excelente *performance* que elas apresentam quando bem exploradas, que é um ponto importante de tirar partido quando falamos de problemas nesta área.

Em primeiro lugar, críamos uma API para a utilização de cada um dos algoritmos, sendo ela a seguinte:

- *void poisson_jac(double** u, size_t N, double TOLERANCE, unsigned long* iter)* : implementação sequencial do método de Jacobi para a resolução numérica da equação de Poisson;
- *void poisson_gs(double** u, size_t N, double TOLERANCE, unsigned long* iter)* : implementação do método de Gauss-Seidel para a resolução numérica da equação de Poisson;
- *void poisson_gs_rb(double** u, size_t N, double TOLERANCE, unsigned long* iter)* : implementação paralela do método de Gauss-Seidel com ordenação *Red-Black* para a resolução numérica da equação do Poisson;
- *void poisson_sor(double** u, size_t N, double TOLERANCE, unsigned long* iter, double omega)* : implementação sequencial do método de S.O.R. para a resolução numérica da equação de Poisson;
- *void poisson_sor_rb(double** u, size_t N, double TOLERANCE, unsigned long* iter, double omega)* : implementação paralela do método de S.O.R. com ordenação *Red-Black* para a resolução numérica da equação do Poisson;

Na implementação em MATLAB, estas funções devolvem todas a matriz u modificada e o número de iterações $iter$, no entanto na versão em C são passados apontadores para ambos. Para o caso da implementação do algoritmo S.O.R., também é passado o valor do parâmetro de relaxação, o ω ($omega$). Isto serve apenas para o caso de quermos utilizar valores que não o ótimo para ω, em que podemos explicitamente passar o valor do mesmo. Caso queirámos utilizar o valor ótimo, devemos passar um valor negativo para esse parâmetro. Como exploramos nas aulas o efeito que escolher um valor que não o ótimo produz sobre a convergência do algoritmo, achamos que não foi necessário fazer experiências com outros valores, sendo que o resultado seria que a convergência seria mais lenta, o número de iterações aumentava e, como consequência, o tempo de execução seria maior.

Em geral, a primeira implementação dos vários algoritmos foi uma tradução um para um do código MATLAB para C. No entanto, isto apresentava alguns problemas em termos de utilização de recursos de memória:

- Para todos os casos, utilizamos duas matrizes: u e w. A matriz w serve como auxílio aos cálculos, e guarda a versão mais recente dos valores de u, sendo que esta guarda os valores da última iteração. Isto implicaria ter o dobro da memória a ser utilizada para a mesma matriz *input*, o que é altamente ineficiente.

- Mais ainda, a utilização do máximo das diferenças da matriz u e w como termo de comparação para o caso de paragem involve percorrer as matrizes u e w uma vez mais para cada iteração. Isto alberga uma quantidade de *loads* de memória descomunal que aumenta significativamente o *overhead* de acessos à memória do programa, apesar dos acessos serem contíguos.

Ambos os problemas que mencionamos podem ser resolvidos, e falaremos da solução que encontramos na secção seguinte.

4 Otimizações

Nesta secção iremos falar das várias otimizações que aplicamos à implementação dos algoritmos, nomeadamente no que toca ao cálculo do parâmetro de paragem e da paralelização.

4.1 Cálculo de *Max(|U - W|)*

Como mencionamos anteriormente, este cálculo que é feito no fim de cada iteração para verificarmos o caso de paragem obriga, em primeira análise, a guardarmos as duas matrizes em memória, tanto a u como a w. Isto é completamente verdade para a implementação do método de Jacobi, mas não para os outros dois.

Tanto o método de Gauss-Seidel como o método S.O.R. não necessitam explicitamente de uma matriz com os valores da última iteração. Uma vez que, para uma iteração $k+1$ utilizamos tanto os valores já calculados em $k+1$ como os de k, podemos utilizar apenas uma matriz para este efeito, que será a matriz de *input* u. Assim, enquanto que na primeira versão teremos de guardar $2 * N^2$ *doubles* em memória, que se traduz em $2 * N^2 * 8$ *bytes*, agora apenas precisamos de guardar $N^2 * 8$ *bytes*, o que significa que reduzimos o *overhead* de memória para metade.

Mas, se não guardamos os valores da última iteração, como podemos calcular $Max(|U - W|)$? Bem, na versão anterior, nós eramos obrigados a fazer isto no final de cada iteração, mas não é uma obrigação real. Podemos calcular o máximo à medida que atualizamos a matriz u.

Originalmente, tínhamos o seguinte código para a atualização de u, para o caso da implementação do método de Gauss-Seidel com ordenação lexicográfica:

```
for(unsigned i = 1; i < N-1; i++)
{
    for(unsigned j = 1; j < N-1; j++)
    {
        // update U(i,j)
        U[i][j] = ( U[i-1][j] + U[i][j-1]
                  + U[i][j+1] + U[i+1][j] ) / 4.0;
    }
}
```

No entanto, se nós guardarmos o valor de $U[i][j]$ antes de ser atualizado, atualizarmos normalmente e depois calcularmos a diferença máxima parcial, podemos

ir calculando *Max(|U - W|)* à medida que atualizamos a matriz u, o que nos retira o *overhead* que a operação $u - w$ implicaria em C, ficando assim o código:

```
for(unsigned i = 1; i < N-1; i++)
{
    for(unsigned j = 1; j < N-1; j++)
    {
        // store the old value of U(i,j)
        double tmp = U[i][j];
        // update U(i,j)
        U[i][j]  = ( U[i-1][j] + U[i][j-1]
                   + U[i][j+1] + U[i+1][j] ) / 4.0;
        // update maximum difference
        diff = diff > fabs(U[i][j] - tmp) ?
               diff : fabs(U[i][j] - tmp);
    }
}
```

Em suma, com esta pequena otimização, podemos diminuir tanto o *overhead* computacional como o de memória, tornando o algoritmo sequencial consideravelmente mais eficiente. Isto terá, no entanto, algumas consequências na versão paralelizada que requererão alguma atenção especial.

4.2 Paralelização com *OpenMP*

Em termos de paralelização do algoritmo, a primeira coisa que deverá ser feita é uma análise de que partes do algoritmo têm critérios para ser paralelizadas. Como o foco do trabalho é nos métodos de Gauss-Seidel e do S.O.R., deixaremos de parte o método de Jacobi para esta análise.

Uma iteração k é completamente dependente da iteração $k-1$, pelo que não existe uma forma direta e prática de dividir o trabalho de cada iteração por *threads*, de modo a ter uma *pipeline*. Assim, o único local que nos resta para paralelizar é no varrimento e atualização da matriz u. Para este efeito, poderíamos dividir o trabalho por colunas ou por linhas. Para que haja um melhor aproveitamento da memória, decidimos dividir o trabalho por linhas, ou seja, cada processo/*thread* fica com um conjunto de linhas para processar. Como o número de linhas e de colunas é conhecido logo de início e a carga de trabalho é mais ou menos a mesma para cada linha, o escalonamento do trabalho pode ser estático.

Ficamos agora com a escolha de que ferramenta utilizaremos para paralelizar, pelo que optamos pelo *OpenMP*, devido ao facto que estamos bastante familiarizados com a sua API e demonstra ganhos significativos quando bem empregado. Decidimos não escolher MPI pois achamos que o *overhead* comunicacional que a comunicação entre processos adiciona seria desnecessário e iria prejudicar a *performance* do programa, especialmente quando temos em conta que o tamanho das matrizes que vamos utilizar é relativamente pequeno.

Assim sendo, temos agora um problema quando olhamos para o algoritmo mais clássico de varrimento das matrizes, que é a ordem lexicográfica. Como este algoritmo

trata-se de um *stencil* de 5 pontos, os pontos que pertencem às linhas que não a que uma certa *thread* trata poderão estar a ser acedidos por outras *threads*, o que leva a *data races* e ao mau funcionamento do algoritmo. É exatamente este problema que a ordenação *Red-Black* vem resolver. Na ordenação *Red-Black*, nunca vamos ter *threads* a escrever em valores de u que estejam a ser lidos, como mencionamos anteriormente, e então resolvemos o problema das *data races*.

Assim, concluímos que os algoritmos que vamos paralelizar serão o Gauss-Seidel e o S.O.R. com ordenação *Red-Black*.

Para este efeito, deveremos paralelizar o ciclo *for* que percorre as linhas, utilizando a cláusula *#pragma omp parallel for*. Contudo, isto não é suficiente, pois devemos ainda tratar da variável *diff* que utilizamos para determinar a diferença máxima.

A variável *diff*, como está de momento, é perigosa pois é partilhada por todos os processos. Isto leva a que todas as *threads* tentem escrever sem controlo no *diff*, e pode levar a problemas. A solução é ter um *diff* parcial, ao qual chamamos *diff_tmp*, que calcula $Max(|U_i - W_i|)$, sendo i o índice da linha e, depois de percorrermos as linhas, atribuímos a *diff* o valor do *diff* partical e fazemos uma redução para o valor máximo de *diff*.

Mas, se nós fazemos a redução do *diff* no fim, não nos chegava utilizar o *diff*? Porquê utilizar o auxiliar *diff_tmp*? Para comparar a diferença, assumindo que é um valor absoluto e não há, por isso, valores negativos, teremos de ter o *diff* inicializado a 0, o menor valor possível que este pode tomar, mas também queremos que ele seja privado a cada uma das *threads*, para que não hajam escritas concorrentes. Como tal, ele tem de ser *firstprivate*. O problema é que uma variável não pode ser utilizada como parâmetro para uma cláusula *firstprivate* e ainda para uma cláusula *reduction*, e portanto temos de separar as duas, ficando:

firstprivate(diff_tmp) reduction(max:diff)

Em termos de escalonamento, como dissemos anteriormente, o trabalho é basicamente igual para cada uma das *threads*, e portanto podemos ficar com escalonamento estático (*schedule(static)*). Como este é o escalonamento por omissão, não foi necessário escrevê-lo.

5 Testes

Nesta secção iremos expôr a metodologia que utilizamos para realizar os testes, bem como as suas condições e resultados.

5.1 Condições de Teste

As medições foram feitas utilizando o *cluster SeARCH6*[3] da Universidade do Minho, nodo r662. Este nodo contém dois *Intel® Xeon® Processor E5-2695 v2*[1], para um total de 24 *cores*. De modo a testar com vários nodos, testamos com 1 nodo, 2 nodos e 4 nodos, sempre pedindo 8 processos por nodo.

Em termos de *datasets* utilizados, experimentamos para malhas com N compreendido entre 50 e 500. Estamos cientes que estes são tamanhos relativamente pequenos,

mas para obtermos os tempos com mais do que uma repetição tivemos dificuldades uma vez que os métodos Gauss-Seidel (com e sem *Red-Black Ordering*) começavam a ser bastante lentos a partir desse ponto e ultrapassavam o limite de tempo dado pelo *cluster*.

Para cada tamanho, repetimos as medições 5 vezes, menos que as 10 que fizemos para o trabalho anterior, mas isto deve-se mais uma vez ao facto que o tempo de execução para o método Gauss-Seidel era demasiado elevado para tamanhos maiores que 500. Em termos do *multithreading*, experimentamos um número de *threads* igual ao número das potências de 2 compreendidas entre 1 e 32.

Os valores mais interessantes a ser medidos neste trabalho, de modo a comparar a *performance* dos vários algoritmos são o tempo de execução e o número de iterações.

Para medir o tempo de execução do programa em C utilizamos uma pequena biblioteca desenvolvida a partir de uma base dada pelo professor André Pereira em Arquiteturas Avançadas para medir o tempo de execução, e em MATLAB foi utilizada a função *timeit()*. É importante referir que esta comparação não pôde ser feita no *cluster*, uma vez que não podemos executar programas de MATLAB no mesmo, pelo que os testes foram feitos com a máquina da equipa, um *MacBook Air, modelo early 2015*[2]. Medições dos tempos de execução do algoritmo em C também foram tiradas na máquina da equipa de modo a podermos fazer uma comparação mais justa.

5.2 Número de Iterações

Olhando primeiro para o número de iterações, podemos comparar para diferentes valores de N o número de iterações:

	100	200	300	400	500
Gauss-Seidel Lexicográfico	5946	24032	54259	96627	151136
Gauss-Seidel Red-Black	5970	24081	54333	96726	151260
S.O.R. Red-Black	210	447	705	984	1258

Optamos por representar estes valores numa tabela e não num gráfico pois as escalas são muito diferentes, pois as iterações do S.O.R. não ultrapassam as mil e trezentas, enquanto que as dos outros dois chegam às várias dezenas de milhar, ultrapassando até as cem mil.

Como podemos observar, o número de iterações que a implementação com o método de S.O.R. executa é muito, muito menor que o do método de Gauss-Seidel, o que comprova a teroria que o primeiro converge muito mais rapidamente que o segundo.

Também é interessante observar o facto que, apesar de as ordenações lexicográfica e *Red-Black* terem o mesmo ritmo de convergência, a ordenação *Red-Black* produziu para todos os casos um número ligeiramente mais elevado de iterações, aproximadamente mais cem, uma diferença pouco significativa.

5.3 Tempo de Execução

Com a introdução de paralelismo, faz sentido olharmos para o *speedup* que obtivemos quando contrastamos com a versão sequencial.

Olhando primeiro para o *speedup* que esperamos ter, uma vez que o trabalho é feito completamente em paralelo, sem uma porção sequecial significativa, de acordo com a **Lei de Amdhal**, esperamos que o *speedup* seja linear e proporcional ao número de fios de execução utilizados.

Para podermos fazer uma melhor comparação da *performance* de cada uma das variantes, utilizamos o maior tamanho com que testamos, que foi 500.

Assim, podemos observar o *speedup* para 1 nodo que obtivemos no seguinte gráfico:

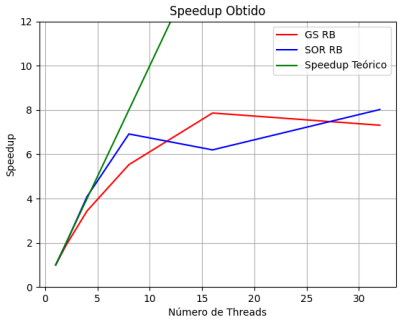

Figure 2: *Speedup* obtido

Como podemos ver na figura 2, o *speedup* obtido em ambos os casos é aceitável, mas encontra-se sempre abaixo do oito. Acreditamos que isto aconteça devido ao facto que a ordenação *Red-Black* tem padrões de acesso à memória relativamente fracos, uma vez que obriga a dois varrimentos à matriz que não utilizam de uma forma ótima a *cache*, devido às razões que mencionamos anteriormente. Para além disto, também acreditamos que seja porque o tamanho das matrizes *input* (as malhas) são relativamente pequenos, pelo que o trabalho de que cada *thread* fica encarregue é computacionalmente leve (que também é assumidamente uma característica do algoritmo).

Será interessante notar que existe um vale em termos de *speedup* para o S.O.R para 16 *threads*, o que é estranho. Em primeira análise, esperar-se-ia que fosse um pequeno desvio nos testes, mas estes foram executados várias vezes, pelo que este resultado é mais estranho por causa disso.

Olhando agora para as execuções utilizando mais do que um nodo, para o caso do método de S.O.R.:

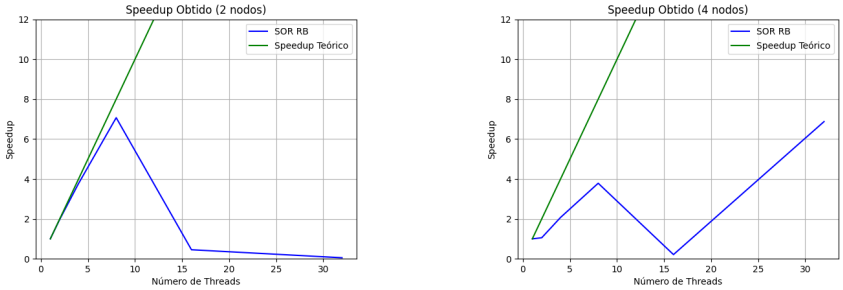

Figure 3: *Speedup* obtido utilizando 2 e 4 nodos

Aqui vemos que os *speedups* obtidos são consideravelmente piores do que no caso anterior com apenas um nodo. Acreditamos que isto tenha a ver com dois fatores: em primeiro, o tráfego no *cluster* é relativamente elevado, e era quando executamos os testes, pelo que seria algo previsível termos resultados um pouco abaixo do esperado, e em segundo, e possivelmente mais importante, a utilização de mais nodos implica utilizar mais *cores*, e a transferência da execução das *threads* entre *cores* pode justificar um pouco os resultados questionáveis. Mais uma vez, podemos ver que existe um vale de *performance*, em particular para o caso dos 4 nodos, para as 16 *threads*.

5.3.1 Implementação em C vs. MATLAB

Escusado será dizer que a implementação em C é consideravelmente mais rápida do que a mesma em MATLAB. Sendo C uma linguagem de relativamente baixo nível, pelo menos quando comparado com MATLAB, e com o algoritmo aperfeiçoado em alguns pontos de modo a fugir a algumas operações que adicionam complexidade, seria de esperar que realmente houvesse algum *speedup* em relação à versão em MATLAB, que se verificou. Utilizando um dos algoritmos que desenvolvemos (escolhemos a implementação do S.O.R. por ser o mais rápido em ambas as versões) comparamos o tempo de execução dos algoritmos em MATLAB e em C:

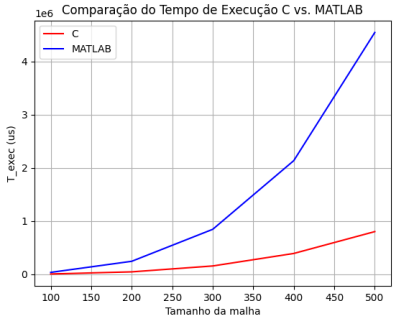

Figure 4: Comparação do Tempo de Execução nas duas linguagens

Como podemos ver, enquanto que a diferença não é muito notória para grãos mais grossos da malha, quanto mais fino maior é a diferença entre as duas implementações, sendo que o C é efetivamente melhor que o MATLAB sempre.

6 Conclusão

Em geral acreditamos ter conseguido atingir os objetivos principais do trabalho. Desenvolvemos uma implementação em C dos vários algoritmos pedidos, aplicamos algumas otimizações, nomeadamente a paralelização, e tivemos a capacidade de analisar os dados obtidos com os vários testes que fizemos no *cluster*. Mais ainda, pudemos fazer uma pequena comparação com a versão inicial em MATLAB, vendo os aspetos que conseguimos melhorar em relação a esta versão.

7 Referências

[1] *Intel Xeon Processor E5-2695 v2*. URL: https://ark.intel.com/content/www/us/en/ark/products/75281/intel-xeon-processor-e5-2695-v2-30m-cache-2-40-ghz.html.

[2] *MacBook Air, Early 2015 model*. URL: https://support.apple.com/kb/SP714?locale=en_US.

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