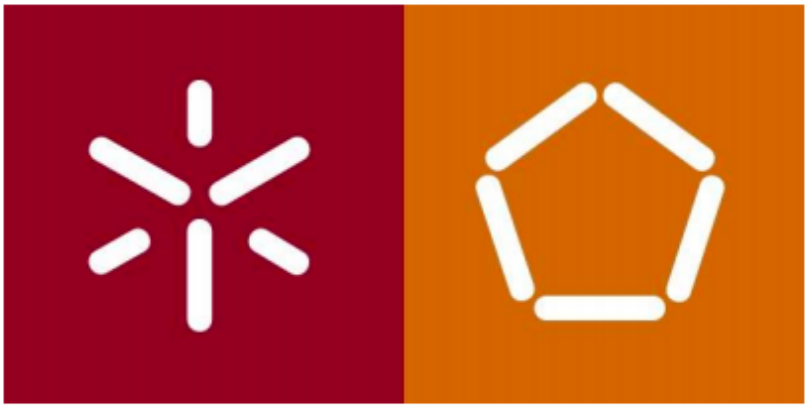

Universidade do Minho

Mestrado Integrado em Engenharia Informática

7/04/2020

Algoritmos Paralelos
Trabalho Prático 2

António Gonçalves (A85516)
Gonçalo Esteves (A85731)

Contents

1 Introdução

Neste relatório iremos expor e explicar as nossas resoluções e estratégias no que toca à resolução do segundo trabalho proposto para a cadeira de Algoritmos Paralelos.

O segundo trabalho consiste em criar versões sequenciais e paralelas de um problema que utiliza o método de *Poisson*. Foram então fornecidas 4 versões deste, com diferentes otimizações, por forma a facilitar uma futura implementação paralela. Uma utilizando iterações de *Jacobi*, ou usando iterações *Gauss-Seidel*, outra que para além de *Gauss-Seidel* utiliza *red and black ordering*, e finalmente uma que utiliza sobre-relaxações sucessivas e *red and black ordering*.

De seguida iremos analisar o problema mais detalhadamente. O objetivo do programa será, numa placa quadrada de metal com tamanho N com os limites a temperaturas diferentes, encontrar uma situação de equilíbrio. Assim, os limites inferiores, e laterais encontram-se inicialmente a 100ºC, enquanto que o limite superior está a 0ºC, e o restantes pontos a 50ºC. Esta placa de metal será representada por uma matriz N por N, onde cada valor representa o valor de temperatura no dado instante.

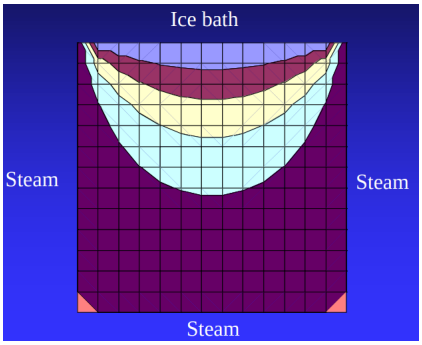

Figure 1: Exemplo de uma solução estável.

De forma a ter a certeza que a solução encontrada é a ótima, em cada iteração iremos encontrar a diferença máxima entre os valores antigos e o novos de temperatura na placa. Caso este seja este seja inferior ao valor de tolerância a solução encontrada é considerada como estável, uma vez que a variação entre os valores deixa de ser suficientemente significativa para continuar o algoritmo.

Ao longo das iterações, cada valor do interior da placa irá variar consoante os valores dos pontos que o rodeiam, ou seja, para um ponto (i,j), teremos que ter em conta os valores para os pontos $(i-1,j)$, $(i,j-1)$, $(i,j+1)$ e $(i+1,j)$. O novo valor é então calculado da seguinte forma:

$$w(i,j) = (u(i-1,j) + u(i,j-1) + u(i,j+1) + u(i+1,j))/4$$

onde w(i, j) será o novo valor para a posição (i, j), e u(i, j) o valor antigo.

2 Abordagens ao Problema

Como ja foi referido, existem quatro versões deste algoritmo: A primeira utilizando iterações de *Jacobi*, outra usando iterações de *Gaus-Seidel*, outra que para além das iterações *Gaus-Seidel* utiliza *red and black ordering* e finalmente uma que utiliza *red and black ordering* e o método de sobre-relaxações sucessivas. De seguida iremos explicar as diferenças entre cada uma desta abordagens, bem como as nossas estratégias sobre como realizar as implementações paralelas das mesmas.

2.1 Método com Iterações de Jacobi

Neste algoritmo são criadas duas matrizes: a matriz U, onde iremos guardar os valores da iteração anterior, e a matriz W, onde iremos guardar os valores novos calculados na iteração atual. Assim, para cada par (i, j) iremos usar os valores vizinhos da mesma posição na matriz U para calcular W[i][j]. Deste modo conseguimos garantir que todos os novos valores calculados têm por base os valores iniciais da iteração do ciclo, não utilizando valores já alterados na execução dessa iteração, permitindo a obtenção de valores mais aproximados aos reais.

Após percorrer a matriz inteira, será necessário calcular a diferença máxima entre os valores antigos (da matriz U) e os novos calculados (da matriz W), sendo este utilizado no seu valor absoluto. Para tal, e de forma a tornar o código mais eficaz, este calculo é efetuado logo depois do calculo de cada valor de W, evitando assim percorrer a matriz novamente no final para encontrar esse mesmo máximo. Passado algumas iterações este valor será suficientemente pequeno para que se possa afirmar que a solução encontrada já esteja estável.

Não foi desenvolvida uma versão paralela deste algoritmo uma vez que, para além de ser complicado garantir a sua correta execução, também se tornaria extremamente ineficiente, devido à grande quantidade de zonas críticas e operações atómicas que iriam ser executadas. Isto acontece devido à dependência de dados presente no problema, que faz com que, para alterar o valor da matriz numa posição seja necessário consultar os valores nas posições adjacentes.

2.2 Método com Iterações de Gauss-Seidel

Este método é bastante parecido ao anterior. Contudo, em vez de inicializar a matriz U ao mesmo tempo que a W, é criada uma cópia em cada iteração do algoritmo, que é apenas utilizada no final, de forma a que se possa, calcular a diferença máxima. Assim sendo, os novos valores de W vão ser sempre calculados com os valores guardados em W, o que faz com que sejam utilizados valores já atualizados durante a execução do algoritmo na mesma iteração.

De forma a facilitar o cálculo da diferença, em vez de criar uma copia de W em cada iteração do ciclo *while*, é possível criar uma cópia do valor de W[i][j] em cada iteração dos ciclos i e j, de forma a calcular a diferença no momento, em vez de o fazer no final de todas as iterações (i, j), fazendo assim com que não seja necessário voltar a percorrer a matriz no fim.

Tal como no algoritmo anterior, não foi possível desenvolver uma versão paralela eficiente do algoritmo.

2.3 Método com Iterações de Gauss-Seidel e *Red and Black ordering*

A única diferença entre este e o anterior é a utilização de *Red and Black ordering*. Com isto iremos dividir os cálculos das posições alternadas de forma a chegar aos novos valores em dois ciclos diferentes, dividindo assim o ciclo em dois mais pequenos.

Red and Black ordering consiste mais especificamente em garantir que não existem dependências entre os valores que estão a ser calculados. Assim, e tal como podemos ver na imagem a baixo, caso estejamos

a calcular um valor assinalado como *Black* na placa, teremos a certeza de que apenas estamos a aceder a valores marcados com *Red*. Desta forma, conseguimos impedir o *data-race* que aconteceria numa execução paralela caso os cálculos fossem pela ordem normal dos pontos, uma vez que, ao distribuir estes por *threads* iríamos permitir a ocorrência de leituras e escritas em simultâneo, para um mesmo valor.

Utilizando esta técnica fazemos com que o primeiro conjunto de pontos a ser calculado (*red* ou *black*) utilize os valores iniciais da iteração, enquanto que o segundo conjunto já irá utilizar valores atualizados.

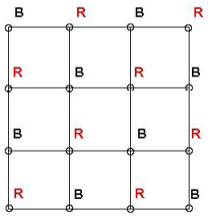

Figure 2: Divisão dos valores utilizando *red and black ordering*

Quanto à paralelização, criamos um vetor com tamanho igual ao número de *threads*, e tornamos o interior do ciclo *while* numa região paralela. Em seguida, fizemos com que os dois ciclos *for* fossem executados de forma paralela com o auxílio de todas as *threads*, sendo que cada uma iria utilizar o vetor criado para registar qual foi a diferença máxima que encontrou. Por fim, uma só *thread* consulta o vetor para verificar qual foi a diferença máxima registada, e passa-se para a próxima iteração.

2.4 Método com sobre-relaxações sucessivas (SOR) e *Red and Black ordering*

Finalmente temos a implementação que utiliza o método de sobre-relaxações sucessivas. Para este utilizamos um parametro de relaxação p, que é calculado da seguinte forma:

$$p = 2/(1 + sin(\pi/(N-1)))$$

Assim, o calculo do novo valor será:

$$W[i][j] = (1-p) * W[i][j] + p * (W[i-1][j] + W[i][j-1] + W[i][j+1] + W[i+1][j])/4$$

Desta forma o algoritmo consegue aproximar-se mais do valor onde a placa está estável, o que irá diminuir bastante o número de iterações necessárias.

A paralelização desta implementação é igual à realizada na versão *Gauss-Seidel*, uma vez que entre as duas apenas muda a forma como é calculado o novo valor na matriz.

3 Análise dos Resultados

De seguida iremos analisar os resultados obtidos com a execução dos algoritmos anteriormente citados (exceto a implementação com iterações de Jacobi). Para isto utilizamos um dos nodos 652 do *cluster SeARCH* da Universidade do Minho, requisitando sempre todos os seus 20 *cores*. Os valores de tempo foram calculados tendo como base a média de 5 medições diferentes

3.1 Resultados com Algoritmos Sequenciais

Numa primeira fase, executamos as versões sequenciais do código para diferentes tamanhos de matriz por forma a analisar o desempenho dos algoritmos e os comportamentos demonstrados pelos mesmos. São em seguida apresentados os resultados obtidos.

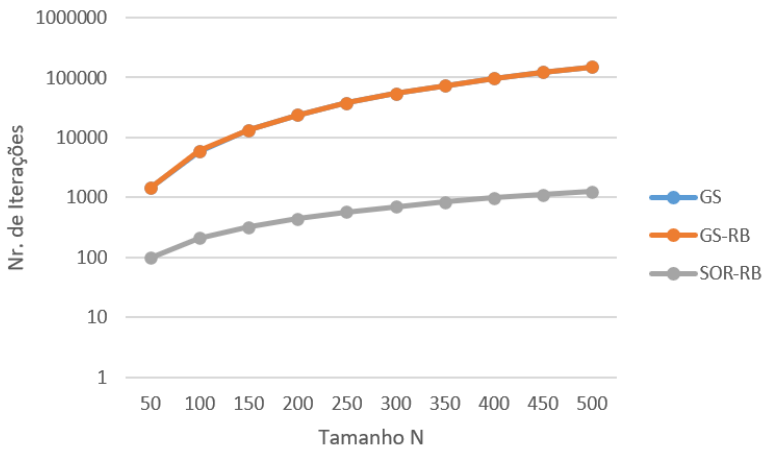

Figure 3: Número de Iterações obtido para cada algoritmo.

	50	100	150	200	250	300	350	400	450	500
GS	1446	5926	13441	23992	37577	54198	73855	96546	122273	151035
GS-RB	1458	5950	13478	24041	37639	54272	73941	96645	122385	151159
SOR-RB	98	210	325	446	576	705	845	983	1115	1258

Table 1: Número de Iterações obtido para cada algoritmo.

Como podemos observar, o número de iterações é substancialmente inferior, para um mesmo tamanho, utilizando sobre-relaxação (como seria espectável). Quando comparando as implementações com *Gauss-Seidel*, verificamos que a que utiliza *Red and Black ordering* necessita de mais iterações. Isto pode ser explicado com base na forma com os valores da matriz são calculados em cada uma. Recorrendo a *Red and Black ordering*, o primeiro conjunto de pontos (*red* ou *black*) é calculado com os valores iniciais da matriz na iteração, enquanto que o segundo já é calculado com os valores finais. Assim sendo, as aproximações dos primeiros valores não serão tão boas, o que levará à necessidade de mais iterações. Com a outra implementação, sempre que se calcula um ponto da matriz todos os anteriores já foram calculados. Assim, à medida que se vai percorrendo a matriz, as aproximações vão sendo cada vez melhores, o que leva a que sejam necessárias menos iterações.

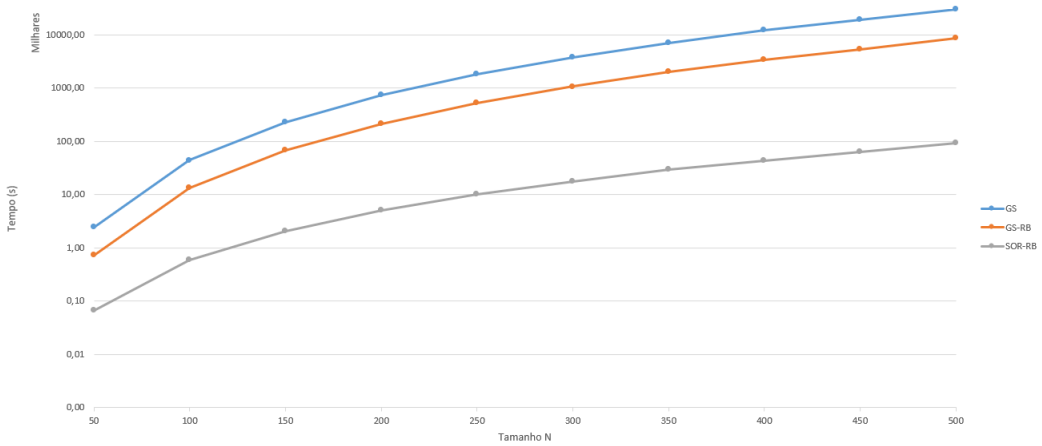

Figure 4: Resultados obtidos para a implementação sequencial, variando o tamanho da placa.

	50	100	150	200	250	300	350	400	450	500
GS	2 439,52	44 168,38	229 839,66	739 239,80	1 824 887,63	3 810 445,96	7 099 560,41	12 154 982,50	19 524 308,75	29 820 875,56
GS-RB	742,84	13 201,39	67 839,41	211 471,30	522 829,02	1 067 661,17	2 013 318,26	3 373 678,29	5 400 700,76	8 741 956,88
SOR-RB	65,95	598,06	2 083,28	5 042,95	10 301,51	17 797,95	29 222,69	44 062,73	63 196,85	94 592,31

Table 2: Resultados obtidos, em ms, para as diferentes implementações do algoritmo sequencial, variando o tamanho da placa N.

Analisando os tempos de execução, verificamos que para todos os algoritmos o aumento do tamanho da matriz leva a um aumento exponencial do tempo de execução. Isto advém do facto de que aumentar o tamanho da matriz implica não só mais pontos para processar mas também a necessidade de um maior número de iterações até encontrar o resultado final.

Comparando as diferentes implementações verificamos que a mais rápida é a que utiliza sobre-relaxações sucessivas e *Red and Black ordering*, tal como seria de esperar, uma vez que esta utiliza o parâmetro de relaxação para realizar várias aproximações, o que leva a que sejam necessárias menos iterações. Por outro lado, verificamos que entre as duas implementações que utilizam *Gauss-Seidel*, a que recorre a *Red and Black ordering* é mais rápida, apesar de necessitar de mais iterações. Não conseguimos justificar este comportamento uma vez que seria esperado o contrário, não só devido ao maior número de iterações necessárias mas também devido ao facto de que a travessia da matriz é efetuada duas vez com *Red and Black ordering*, o que deveria levar a tempos de execução maiores.

3.2 Resultados com Algoritmos Paralelos

Em seguida procedemos à obtenção de resultados utilizando as versões paralelas criadas. Calcularam-se os *speedups* e interpretaram-se os resultados, utilizando novamente matrizes de diferentes tamanhos bem como diferentes quantidades de *threads*. Os resultados são apresentados em seguida, bem como uma análise dos mesmos tendo em conta a variação do número de *threads* e do tamanho da placa.

3.2.1 Variação do número de *threads*

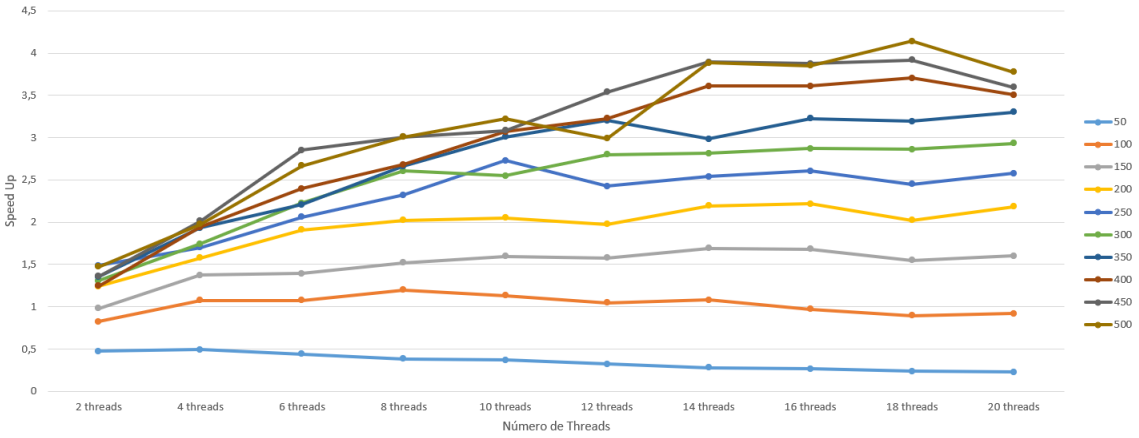

Figure 5: *Speedup* obtido para a implementação *Gauss-Seidel*, com *red and black ordering*, tendo em conta o número de *threads*.

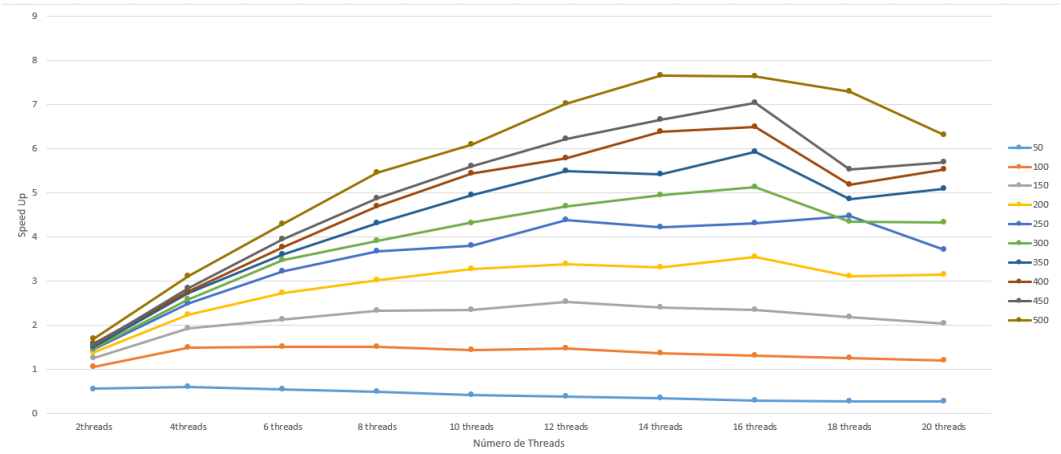

Figure 6: *Speedup* obtido para a implementação SOR, com *red and black ordering*, tendo em conta o número de *threads*.

Em ambos os casos, o uso de *threads* para o menor tamanho da placa testado dá prejuízo, ou seja, não compensa executar os algoritmos de forma paralela para tamanhos iguais ou inferiores a 50, muito provavelmente devido ao facto de que o custo da criação das *threads* é superior aos ganhos obtidos. Para além disto, o aumento do número de *threads* faz com que o tempo de execução aumente ainda mais, já que vão haver custos de criação de *threads* cada vez maiores sem que o trabalho efetuado por cada *thread* seja suficiente para compensar o custo de criação.

No caso da implementação com *Gauss-Seidel*, a utilização da versão paralela continua a não trazer grandes vantagens para um tamanho de placa de 100, e, neste caso, a partir de 8 *threads*, o aumento da quantidade das mesmas prejudica cada vez mais a execução do algoritmo. Para os restantes tamanhos, já se começa a justificar o uso da versão paralela do algoritmo, sendo que de um modo geral o uso de entre 14 a 20 *threads* permite atingir o *speedup* mais alto, como seria expectável tendo em conta a quantidade de *cores* (20) da máquina.

Relativamente à implementação com SOR, o uso de 4 *threads* ou mais para tamanhos de 100 e 150 permite obter sempre mais ou menos o mesmo *speedup*. Para tamanhos superiores, o aumento do número de *threads* leva a um aumento gradual do *speedup* obtido, sendo que o valor de pico é obtido, de uma forma geral, com 16 *threads*, quantidade a partir da qual o *speedup* volta a diminuir de forma gradual.

Em jeito de remate final, é seguro dizer que os *speedups* teóricos esperados seriam aproximadamente iguais ao número de *threads* utilizadas, no entanto, tal não se verifica na maior parte dos casos. A principal justificação para tal deverá ser o custo de criação de *threads*, já que estas são criadas a cada iteração do ciclo *while*. Isto justifica não só o porque dos *speedups* não serem tão elevados quanto era suposto, mas também o porquê dos *speedups* na implementação com *Gauss-Seidel* serem significativamente inferiores aos obtidos com a implementação SOR, sendo que a única alteração ao nível do código entre as duas é a expressão utilizada para calcular o valor nos pontos. Na primeira implementação são executadas muitas mais iterações, o que leva a um aumento significativo da quantidade de vezes que *threads* são criadas, diminuindo os ganhos obtidos com a paralelização. Na segunda implementação, como não são necessárias tantas iterações também não são criadas tantas vezes *threads*, o que faz com que o seu custo de criação influencie menos o tempo de execução, permitindo que os *speedups* sejam também superiores.

3.2.2 Variação do tamanho da placa

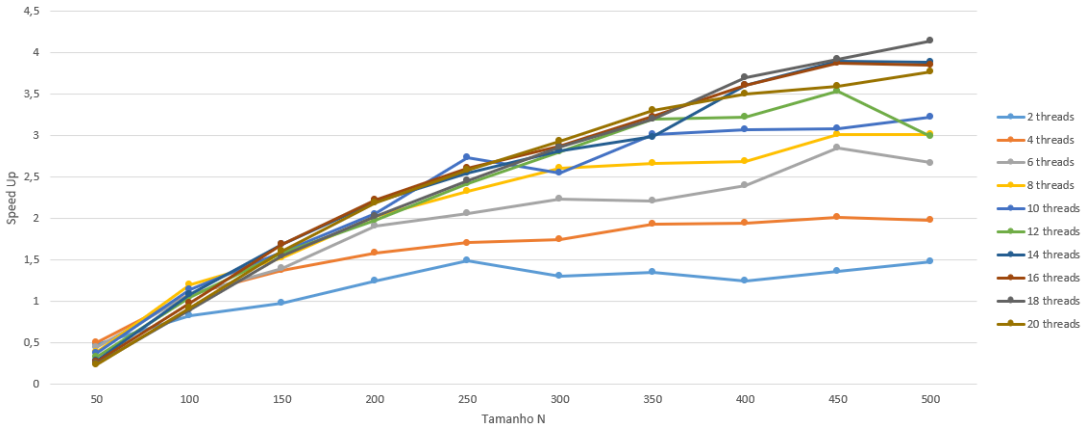

Figure 7: *Speedup* obtido para a implementação *Gaus-Seidel*, com *red and black ordering*, tendo em conta o tamanho da placa.

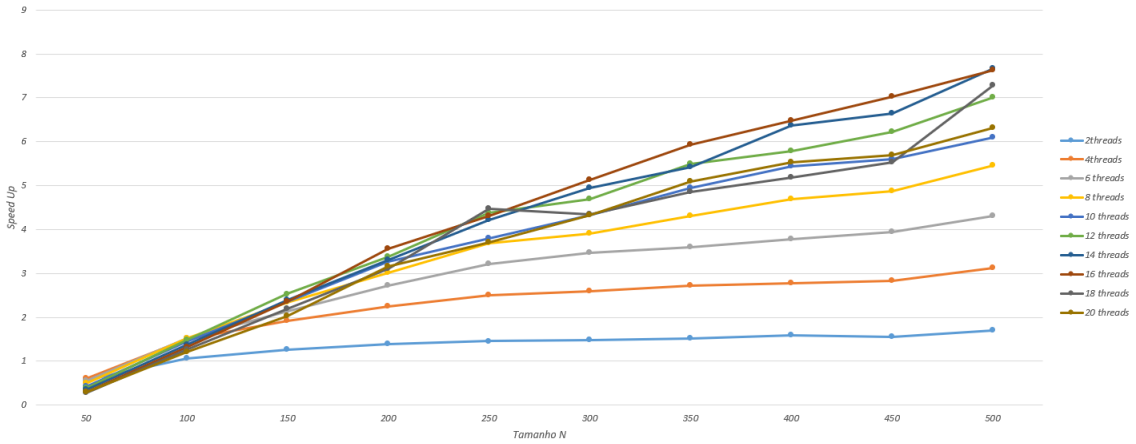

Figure 8: *Speedup* obtido para a implementação SOR, com *red and black ordering*, tendo em conta o tamanho da placa.

Para ambas as implementações, e para qualquer quantidade de *threads*, é fácil constatar que, de um modo geral, o aumento do tamanho da placa leva à obtenção de *speedups* maiores. Este facto é justificável tendo em conta que o aumento do tamanho da placa leva a um aumento do trabalho efetuado por cada *thread*, tornando assim a paralelização mais eficiente, já que o custo da criação das *threads* estará mais mascarado nos ganhos obtidos com a distribuição da carga de trabalho pelas diferentes *threads*.

4 Conclusão

Após o desenvolvimento deste trabalho sentimos que ficamos a perceber melhor não só como funcionam certas técnicas de "desconstrução" de algoritmos, como o *red and black ordering*, que nos permite eliminar dependências de dados em algoritmos paralelos, mas também como podemos manipular os próprios algoritmos, de forma a permitir que estes possam ser executados de forma concorrente sem por em causa a sua correta execução.

5 Anexos

5.1 Tabelas com resultados

5.1.1 Implementação GS-RB Paralela

	50	100	150	200	250	300	350	400	450	500
2 Threads	1570,3280	16001,9141	69201,6031	170519,8521	351820,5617	817328,7309	1487736,5555	2703722,3539	3975745,1365	5935193,1133
4 Threads	1504,8472	12259,9116	49454,2481	134208,0798	307001,0015	612868,1338	1042632,6967	1737558,4039	2682207,4191	4423414,1813
6 Threads	1688,0527	12256,5238	48764,9040	110681,7328	253854,7123	479132,2354	912196,7284	1407924,4113	1893626,4739	3275397,6167
8 Threads	1929,2052	11023,0225	44587,9942	104549,0037	225151,8655	409793,4112	756231,1730	1258026,2634	1795284,9426	2904793,1280
10 Threads	1996,3986	11640,7906	42544,3223	103063,0515	191557,3351	418579,7827	668831,2750	1098733,2659	1751954,8644	2712553,1968
12 Threads	2291,5724	12612,0885	43022,0651	106961,1289	215712,1751	380975,5271	628535,1821	1046320,1979	1525591,3080	2924301,5770
14 Threads	2675,2157	12209,8841	40137,5754	96398,8474	205694,2021	379415,1144	674857,3007	934673,8649	1387492,1225	2250099,7545
16 Threads	2782,8777	13578,6470	40296,8476	95289,5812	200528,2954	371944,4333	623759,3713	934673,2607	1394205,3558	2269097,9397
18 Threads	3106,3334	14775,3945	43801,5077	104523,3228	213263,2468	373224,8220	630345,4688	911079,0955	1378883,4708	2111293,0867
20 Threads	3229,8116	14352,4884	42381,3259	96867,7187	202740,7280	364216,0936	609747,8287	962794,9485	1502826,0747	2316011,4031

Table 3: Tempos obtidos para a implementação GS-RB, variando o tamanho da placa N e o número de Threads

	50	100	150	200	250	300	350	400	450	500
2 Threads	**0,4730**	0,8250	0,9803	1,2402	1,4861	1,3063	1,3533	1,2478	1,3584	1,4729
4 Threads	0,4936	1,0768	1,3718	1,5757	1,7030	1,7421	1,9310	1,9416	2,0135	1,9763
6 Threads	0,4401	1,0771	1,3912	1,9106	2,0596	2,2283	2,2071	2,3962	2,8520	2,6690
8 Threads	0,3851	**1,1976**	1,5215	2,0227	2,3221	2,6054	2,6623	2,6817	3,0083	3,0095
10 Threads	0,3721	1,1341	1,5946	2,0519	**2,7294**	2,5507	3,0102	3,0705	3,0827	3,2228
12 Threads	0,3242	1,0467	1,5769	1,9771	2,4237	2,8024	3,2032	3,2243	3,5401	2,9894
14 Threads	0,2777	1,0812	**1,6902**	**2,1937**	2,5418	2,8140	2,9833	3,6095	3,8924	3,8851
16 Threads	0,2669	0,9722	1,6835	2,2192	2,6073	2,8705	3,2277	3,6095	3,8737	3,8526
18 Threads	0,2391	0,8935	1,5488	2,0232	2,4516	2,8606	3,1940	**3,7029**	**3,9167**	**4,1406**
20 Threads	0,2300	0,9198	1,6007	2,1831	2,5788	**2,9314**	**3,3019**	3,5040	3,5937	3,7746

Table 4: Speedups obtidos para a implementação GS-RB, variando o tamanho da placa N e o número de Threads.

5.1.2 Implementação SOR-RB Paralela

	50	100	150	200	250	300	350	400	450	500
2 Threads	117,4216	563,2680	1655,4742	3630,5078	7087,4764	12013,5365	19341,3795	27733,8028	40888,4880	55690,3622
4 Threads	108,0188	398,6899	1082,0637	2244,7009	4121,7536	6870,3349	10717,7012	15873,7489	22262,6883	30338,1754
6 Threads	118,3934	395,0790	973,2833	1847,6255	3203,5582	5130,4092	8119,7910	11676,1564	16032,3625	21994,0137
8 Threads	132,6620	395,1693	891,8657	1668,9228	2798,8627	4546,3296	6792,7919	9391,5479	12968,9059	17322,1505
10 Threads	155,3796	414,7018	883,6244	1538,9278	2712,4070	4120,2645	5917,8317	8107,8869	11283,6478	15514,7458
12 Threads	168,3466	405,2728	823,3946	1491,5354	2347,1276	3787,3039	5317,3670	7622,1584	10158,9303	13489,0502
14 Threads	191,2081	438,3001	869,2816	1525,8377	2440,3089	3604,2428	5388,4415	6915,5321	9509,9606	12348,9604
16 Threads	219,8325	456,4274	884,4365	1418,2892	2386,8015	3469,1149	4925,1153	6793,2878	8994,2645	12394,8013
18 Threads	235,7572	474,6073	952,1561	1624,5798	2303,3752	4089,6251	6026,5871	8506,6536	11436,5047	12994,1200
20 Threads	229,2868	495,0476	1023,9519	1599,4106	2773,3996	4116,0653	5742,9726	7960,7080	11103,7995	14977,4356

Table 5: Tempos obtidos para a implementação SOR-RB, variando o tamanho da placa N e o número de Threads.

	50	100	150	200	250	300	350	400	450	500
2 Threads	0,5616	1,0618	1,2584	1,3890	1,4535	1,4815	1,5109	1,5888	1,5456	1,6985
4 Threads	**0,6105**	1,5001	1,9253	2,2466	2,4993	2,5906	2,7266	2,7758	2,8387	3,1179
6 Threads	0,5570	1,5138	2,1405	2,7294	3,2156	3,4691	3,5989	3,7737	3,9418	4,3008
8 Threads	0,4971	1,5134	2,3359	3,0217	3,6806	3,9148	4,3020	4,6917	4,8730	5,4608
10 Threads	0,4244	1,4422	2,3577	3,2769	3,7979	4,3196	4,9381	5,4346	5,6007	6,0969
12 Threads	0,3917	**1,4757**	**2,5301**	**3,3810**	4,3890	4,6994	5,4957	5,7809	6,2208	7,0125
14 Threads	0,3449	1,3645	2,3966	3,3050	4,2214	4,9381	5,4232	6,3716	6,6453	**7,6599**
16 Threads	0,3000	1,3103	2,3555	3,5557	4,3160	**5,1304**	**5,9334**	**6,4862**	**7,0263**	7,6316
18 Threads	0,2797	1,2601	2,1880	3,1042	4,4724	4,3520	4,8490	5,1798	5,5259	7,2796
20 Threads	0,2876	1,2081	2,0346	3,1530	**3,7144**	4,3240	5,0884	5,5350	5,6915	6,3157

Table 6: Speedups obtidos para a implementação SOR-RB, variando o tamanho da placa N e o número de Threads.

5.2 Ficheiro poissonMain.c

```
#include "../include/poisson.h"
#include "../include/auxiliar.h"

int main(int argc, char* argv[]){

    if(argc != 3){
        printf("<app> <implementation> <size>\n");
        return 0;
    }

    double** U;
    double** W;
    int N = atoi(argv[2]);
    int iter;
    double time;

    switch (argv[1][0]){
        case 'j':
            for(int i = 0; i < 5; i++){
                clearCache();

                initMatrixesJac(N, &U, &W);

                startTimer();
                iter = poissonJac(N, &U, &W);
                time = stopTimer();

                printf("%d, %2f\n", iter, time);
            }
            //printMatrix(N, &U);
            break;

        case 'g':
            for(int i = 0; i < 5; i++){
                clearCache();

                initMatrixes(N, &W);

                startTimer();
```

```
                iter = poissonGS(N, &W);
41              time = stopTimer();

43              printf("%d, %2f\n", iter, time);
            }
45          //printMatrix(N, &W);
            break;
47
        case 's':
49          switch (argv[1][1]){
                case 'G':
51                  for(int i = 0; i < 5; i++){
                        clearCache();
53
                        initMatrixes(N, &W);
55
                        startTimer();
57                      iter = poissonGSRBSeq(N, &W);
                        time = stopTimer();
59
                        printf("%d, %2f\n", iter, time);
61                  }

63                  //printMatrix(N, &W);
                    break;
65
                case 's':
67                  for(int i = 0; i < 5; i++){
                        clearCache();
69
                        initMatrixes(N, &W);
71
                        startTimer();
73                      iter = poissonSORRBSeq(N, &W);
                        time = stopTimer();
75
                        printf("%d, %2f\n", iter, time);
77                  }

79                  //printMatrix(N, &W);
                    break;
81
                default:
83                  break;
            }
85          break;

87      case 'p':
            switch (argv[1][1]){
89              case 'G':
                    for(int i = 0; i < 5; i++){
91                      clearCache();

93                      initMatrixes(N, &W);

95                      startTimer();
                        iter = poissonGSRBPar(N, &W);
97                      time = stopTimer();

99                      printf("%d, %2f\n", iter, time);
                    }
```

```
101                         //printMatrix(N, &W);
103                         break;
105
                    case 's':
107                         for(int i = 0; i < 5; i++){
                                clearCache();
109
                                initMatrixes(N, &W);
111
                                startTimer();
113                             iter = poissonSORRBPar(N, &W);
                                time = stopTimer();
115
                                printf("%d, %2f\n", iter, time);
117                         }
119                         //printMatrix(N, &W);
                            break;
121
                    default:
123                         break;
                }
125             break;

127         default:
                break;
129     }

131
    return 1;
133 }
```

5.3 Ficheiro poissonJac.c

```
1  #include "../include/poisson.h"
   #include "../include/auxiliar.h"
3
   int poissonJac(int N, double*** U, double*** W){
5      int i, j;
       double TOL = 1.0/((N-1)*(N-1));
7      int ITER = 0;
       double DIFF = TOL + 1;
9
       while (DIFF > TOL){
11         DIFF = 0;

13         for(i = 1; i < N-1; i++)
               for(j = 1; j < N-1; j++){
15                 (*W)[i][j] = ((*U)[i-1][j] + (*U)[i][j-1] + (*U)[i][j+1] + (*U)[i+1][j]) /
   4;

17                 if(fabs(((*W)[i][j] - (*U)[i][j])) > DIFF)
                       DIFF = fabs(((*W)[i][j] - (*U)[i][j]));
19             }
```

```
21          for(i = 0; i < N; i++)
                for(j = 0; j < N; j++)
23                  (*U)[i][j] = (*W)[i][j];

25          ITER++;
        }
27
        return ITER;
29  }
```

5.4 Ficheiro poissonGS.c

```
1   #include "../include/poisson.h"
    #include "../include/auxiliar.h"
3
    int poissonGS(int N, double*** W){
5       int i, j;
        double n;
7       double TOL = 1.0/((N-1)*(N-1));
        int ITER = 0;
9       double DIFF = TOL + 1;

11      while (DIFF > TOL){
            DIFF = 0;
13
            for(i = 1; i < N-1; i++)
15              for(j = 1; j < N-1; j++){
                    n = (*W)[i][j];
17                  (*W)[i][j] = ((*W)[i-1][j] + (*W)[i][j-1] + (*W)[i][j+1] + (*W)[i+1][j]) /
        4;

19                  if(fabs(((*W)[i][j] - n)) > DIFF)
                        DIFF = fabs(((*W)[i][j] - n));
21              }

23          ITER++;
        }
25
        return ITER;
27  }
```

5.5 Ficheiro poissonGSRB.c

```
1   #include "../include/poisson.h"
    #include "../include/auxiliar.h"
3
    int poissonGSRBSeq(int N, double*** W){
5       int i, j;
        double n;
7       double TOL = 1.0/((N-1)*(N-1));
        int ITER = 0;
9       double DIFF = TOL + 1;
```

```
11      while (DIFF > TOL){
            DIFF = 0;
13
            for(i = 1; i < N-1; i++){
15              for(j = 1 + (i%2); j < N-1; j+=2){
                    n = (*W)[i][j];
17                  (*W)[i][j] = ((*W)[i-1][j] + (*W)[i][j-1] + (*W)[i][j+1] + (*W)[i+1][j]) /
    4;
19                  if(fabs(((*W)[i][j] - n)) > DIFF)
                        DIFF = fabs(((*W)[i][j] - n));
21              }
            }
23
            for(i = 1; i < N-1; i++){
25              for(j = 1+ ((i+1)%2); j < N-1; j+=2){
                    n = (*W)[i][j];
27                  (*W)[i][j] = ((*W)[i-1][j] + (*W)[i][j-1] + (*W)[i][j+1] + (*W)[i+1][j]) /
    4;
29                  if(fabs(((*W)[i][j] - n)) > DIFF)
                        DIFF = fabs(((*W)[i][j] - n));
31              }
            }
33
            ITER++;
35      }

37      return ITER;
}
39
int poissonGSRBPar(int N, double*** W){
41      int i, j;
        double n;
43      double TOL = 1.0/((N-1)*(N-1));
        int ITER = 0;
45      double DIFF = TOL + 1;

47      int threads, id;
        double* DIFFAux;
49
        #pragma omp parallel
51      {
            #pragma omp single
53          {
                threads = omp_get_num_threads();
55              DIFFAux = malloc(sizeof(double) * threads);
            }
57      }

59      while (DIFF > TOL){

61          DIFF = 0;

63          #pragma omp parallel
            {
65              id = omp_get_thread_num();
                DIFFAux[id] = 0;
67
                #pragma omp for
69              for(i = 1; i < N-1; i++){
```

```
                    for(j = 1+(i%2); j < N-1; j+=2){
                        n = (*W)[i][j];
                        (*W)[i][j] = ((*W)[i-1][j] + (*W)[i][j-1] + (*W)[i][j+1] + (*W)[i+1][j])
    / 4;

                        if(fabs(((*W)[i][j] - n)) > DIFFAux[id])
                            DIFFAux[id] = fabs(((*W)[i][j] - n));
                    }
                }

                #pragma omp for
                for(i = 1; i < N-1; i++){
                    for(j = 1+ ((i+1)%2); j < N-1; j+=2){
                        n = (*W)[i][j];
                        (*W)[i][j] = ((*W)[i-1][j] + (*W)[i][j-1] + (*W)[i][j+1] + (*W)[i+1][j])
    / 4;

                        if(fabs(((*W)[i][j] - n)) > DIFFAux[id])
                            DIFFAux[id] = fabs(((*W)[i][j] - n));
                    }
                }
            }

            for(i = 0; i < threads; i++){
                if(DIFFAux[i] > DIFF)
                    DIFF = DIFFAux[i];
            }

            ITER++;
        }

        return ITER;
}
```

5.6 Ficheiro poissonSORRB.c

```
#include "../include/poisson.h"
#include "../include/auxiliar.h"

#ifndef M_PI
    #define M_PI 3.14159265358979323846
#endif

int poissonSORRBSeq(int N, double*** W){
    int i, j;
    double p, n;
    double TOL = 1.0/((N-1)*(N-1));
    int ITER = 0;
    double DIFF = TOL + 1;

    p = 2 /(1 + sin(M_PI/(N-1)));

    while (DIFF > TOL){
        DIFF = 0;

        for(i = 1; i < N-1; i++){
            for(j = 1+ ((i)%2); j < N-1; j+=2){
```

```
22                  n = (*W)[i][j];
                    (*W)[i][j] = (1-p) * (*W)[i][j] + p * ((*W)[i-1][j] + (*W)[i][j-1] + (*W)[i
    ][j+1] + (*W)[i+1][j]) / 4;
24
                    if( fabs(((*W)[i][j] - n)) > DIFF )
26                      DIFF = fabs(((*W)[i][j] - n));
                }
28          }
            for(i = 1; i < N-1; i++){
30              for(j = 1+ ((i+1)%2); j < N-1; j+=2){
                    n = (*W)[i][j];
32                  (*W)[i][j] = (1-p) * (*W)[i][j] + p * ((*W)[i-1][j] + (*W)[i][j-1] + (*W)[i
    ][j+1] + (*W)[i+1][j]) / 4;
34                  if( fabs(((*W)[i][j] - n)) > DIFF )
                        DIFF = fabs(((*W)[i][j] - n));
36              }
            }
38
            ITER++;
40      }

42      return ITER;
}
44
int poissonSORRBPar(int N, double*** W){
46      int i, j;
        double p, n;
48      double TOL = 1.0/((N-1)*(N-1));
        int ITER = 0;
50      double DIFF = TOL + 1;

52      p = 2 /(1 + sin(M_PI/(N-1)));

54      int threads, id;
        double* DIFFAux;
56
        #pragma omp parallel
58      {
            #pragma omp single
60          {
                threads = omp_get_num_threads();
62              DIFFAux = malloc(sizeof(double) * threads);
            }
64      }

66      while (DIFF > TOL){
            DIFF = 0;
68
            #pragma omp parallel
70          {
                id = omp_get_thread_num();
72              DIFFAux[id] = 0;

74              #pragma omp for
                for(i = 1; i < N-1; i++){
76                  for(j = 1+(i%2); j < N-1; j+=2){
                        n = (*W)[i][j];
78                      (*W)[i][j] = (1-p) * (*W)[i][j] + p * ((*W)[i-1][j] + (*W)[i][j-1] + (*W
    )[i][j+1] + (*W)[i+1][j]) / 4;
```

```
80                      if(fabs(((*W)[i][j] - n)) > DIFFAux[id])
                            DIFFAux[id] = fabs(((*W)[i][j] - n));
82                  }
                }
84
                #pragma omp for
86              for(i = 1; i < N-1; i++){
                    for(j = 1+((i+1)%2); j < N-1; j+=2){
88                      n = (*W)[i][j];
                        (*W)[i][j] = (1-p) * (*W)[i][j] + p * ((*W)[i-1][j] + (*W)[i][j-1] + (*W
    )[i][j+1] + (*W)[i+1][j]) / 4;
90
                        if(fabs(((*W)[i][j] - n)) > DIFFAux[id])
92                          DIFFAux[id] = fabs(((*W)[i][j] - n));
                    }
94              }
            }
96
            for(i = 0; i < threads; i++){
98              if(DIFFAux[i] > DIFF)
                    DIFF = DIFFAux[i];
100         }

102         ITER++;
        }
104
        return ITER;
106 }
```