‘Odd-Even Sort’ Parelizado

CPD – Algoritmos Paralalos

Hélder Gonçalves – PG28505

ABSTRACT

Computação paralela nem sempre é viável para todos os algoritmos, requeresse sempre uma análise prévia sobre o algoritmo original para o explorar da melhor forma e obter os melhores resultados.

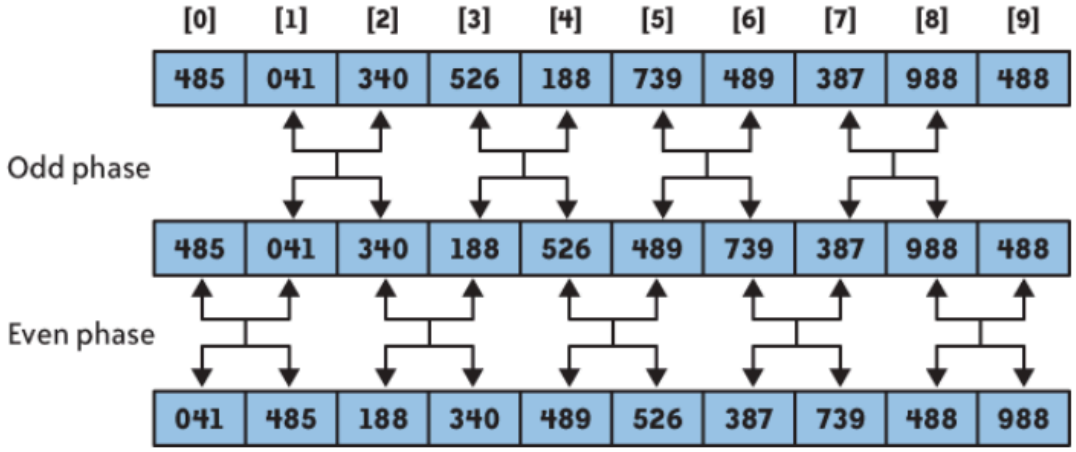
Índice

2. Introdução

Como caso de estudo deste trabalho temos o *'Odd-Even Sort'*. Assim como no trabalho anterior, este é um algoritmo de ordenação relativamente simples. É um algoritmo de ordenação por comparação baseado no bubble sort com o qual compartilha muitas características.

3. Problema

Ele funciona através da comparação de todos os pares indexados (ímpar, par) de elementos adjacentes na lista e, se um par está na ordem errada (o primeiro é maior do que o segundo), os elementos são trocados. O próximo passo repete isso para os pares indexados (par, ímpar) de elementos adjacentes. Em seguida, ele alterna entre etapas de (ímpar, par) e (par, ímpar) até que a lista é ordenada. Pode ser pensado como a utilização de processadores paralelos, cada qual usando um *'BubbleSort'*, mas a partir de diferentes pontos na lista (todos os índices ímpares para a primeira etapa). Este algoritmo de ordenação é apenas ligeiramente mais difícil do que o *'Bubble Sort'* para implementar.



4. Código

a. Versão Sequencial

Antes de começar a tentar paralelizar qualquer algoritmos é importante estudar o seu comportamento base na sua versão mais simples, a sequencial. Como tal, este não foge a esta regra de ouro.

```
void OESort(int NN, int *A)
{
    int exch = 1, start = 0, i;
    int temp;

    while (exch || start) {
        exch = 0;
        for (i = start; i < NN-1; i+=2) {
            if (A[i] > A[i+1]) {
                temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                exch = 1;
            }
        }
        if (start == 0) start = 1;
        else start = 0;
    }
}
```

Como se viu na aula, temos a variável *exch* para nos informar se existiu alguma alteração enquanto o vector de inteiros A é percorrido, tomando este o valor de 1 no caso de

ser efectuada pelo menos uma alteração. Depois também existe a variável start que nos indica qual é a fase actual do problema (par ou ímpar), e por este motivo a fase troca no final de cada iteração. A última parte essencial do problema é o ciclo *for* que percorre todo o vector e no caso do índice *i+1* ser inferior ao indice *i* é efectuada uma troca.

Agora que vimos em detalhe o algoritmo é possivel identificar as zonas possivemente paralelizáveis. A zona que chama a atenção é quando o algoritmo está a percorrer o vector fornecido.

Nos próximos capítulos vamos sugerir algumas possíveis implementações.

b. Parelelização Versão 1 (aula)

Esta é a versão mais simples de todas as versões apresentadas neste trabalho.

```
void OESort_Parallel_V0(int NN, int *A)
{
    int exch = 1, start = 0, i;
    int temp;

    while (exch || start) {
        exch = 0;

        #pragma omp parallel for private(temp)
        for (i = start; i < NN-1; i+=2) {
            if (A[i] > A[i+1]) {
                temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                exch = 1;
            }
        }
        if (start == 0) start = 1;
        else start = 0;
    }
}
```

Para ter obtermos a primeira versão paralela basta adicionar um único *#pragma* no *for* que percorre o vector para ter uma versão perfeitamente funcional. Só temos de ter cuidado que a variável *temp* tem de ser privada a cada uma das threads existentes (esta variável é privada em todas as versões apresentadas, assim como a *exch* e *start* são partilhadas).

Mas se tiver atenção, nota-se que a zona paralela está sempre a abrir e a fichar para executar cada uma das fazes. Este overhead vai causar perda de efeciencia no nosso algoritmo, para evitar isto vamos abrir uma única vez a zona paralela. Para isso ser possivel, o *#pragma* da abertura do ambiente tem de estar antes do ciclo *while*, obtendo então a nossa primeira versão aceitável.

```
void OESort_Parallel_V1(int NN, int *A)
{
    int exch = 1, start = 0, i;

    #pragma omp parallel
    {
        int temp;
        while (exch || start) {

            #pragma omp barrier
            exch = 0;
            #pragma omp barrier

                #pragma omp for
                for (i = start; i < NN-1; i+=2) {
                    if (A[i] > A[i+1]) {
                        temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                        exch = 1;
                    }
                }
```

```
        #pragma omp single
        if (start == 0) start = 1;
        else start = 0;

        }
    }//Close parallel
}
```

Para chegar a esta implementação tivemos que ter em atenção a vários pormenores das directivas do openMP. Sabemos que as directivas *single* e *for* tem barreiras implicitas. Este pormenor dá-nos algumas ajudas aquando a implementação e quais serão as lacunas a culminar para que o algoritmo funcione perfeitamente.

Ao passar o *#pragma omp parallel* para antes do *while*, a nossa primeira reacção foi colocar um directiva *for* para percorrer o vector paralelamente e depois para cuidar da troca da fase a executar usamos a directiva *single*.

À partida ista parecia que ia funcionar, mas infelizmente não. Qual é o prolema!? O problema é que a program entrava em deadlock porque a primeira thread a entrar no ciclo ia alterar logo o valor da variável exch para 0 fazendo com que as threads seguintes nunca chagassem a entrar no ciclo, daí a utilização da primeira directica *'omp barrier'*. A segunda é necessária porque nada garante que quando uma thread esteja a fazer uma alteração dentro do ciclo for, todas as outras *threads* já tenham colocado *exch* a 0, podendo desta forma existir inconsistências, levando assim à colocação de uma segunda barreira.

c. Paralelização Versão 2

Esta é uma segunda versão que implementei a partir da anterior. Pessoalmente não gosto nada desta versão, mas a foi a primeira que implementei com este novo intuito proposto.

```
void OESort_Parallel_V2(int NN, int *A)
{
    int exch, start = 0, i;

    #pragma omp parallel
    {
        exch = omp_get_num_threads();
        int temp;
        while (1) {
            if(exch<=0 && start==0) break;

            #pragma omp barrier
            #pragma omp critical
            exch--;

            #pragma omp for
            for (i = start; i < NN-1; i+=2) {
                if (A[i] > A[i+1]) {
                    temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                    #pragma omp critical
                    exch = omp_get_num_threads();
                }
            }

            #pragma omp single
            if (start == 0) start = 1;
            else start = 0;
        }
    }
}
```

Tivemos um pensamento um pouco diferente da versão anterior. Agora temos o *while* em ciclo infinito, podendo ser quebrado por um *if*. Isto dá-nos um pouco de mais liberdade como vamos ver em próximos algoritmos. O objectivo é colocar em exch o número de *threads* que estão a resolver o problema sempre que é efectuada uma alteração no vector e depois na iteração seguinte cada uma das *threads* decrementar a *exch* até este chegar a 0. Mas se este já estiver a 0 já no principio do ciclo quer dizer que na iteração auterior não foi

efectidada nenhuma alteração no mesmo, isto é, o vector já está ordenado e podemos fazer um *break* para sair do ciclo.

Para estas contas baterem certo é necessário um critical na operação de decremento para evitar as conhecidas *race conditions*. Mas ainda existe outro problema, se não tivermos as barreiras da versão 1 vamos enfrentar um problema de inconsistência de dados, como já foi explicado, isto fará com que o exch possa atingir valores negativos devido a não ter existido consistencia nos dados e por exemplo, no final do o exch é diferente do número de threads existentes ou de 0. Por este motivo colocamos o sinal de '<=' na condição de paragem. Agora para evitar que o problema entre em deadlock temos 1 alternativa, que é colocar um barreira antes ou depois da operação de decremento. Isto faz com que se uma thread passar na condição de paragem, todas as outras vão ter o mesmo resultado.

d. Paralelização Versão 3

Esta versão é na minha opinião a que iria obter os melhores resultados (antes de efectuar testes), porque consegues ser superior relativamente ao melhor uso das primitivas disponíveis. Digo isto porque devido à alteração da ordem dos conceitos anteriores consegue aproveitar as barreiras implicitas das directivas usadas nas zonas principais do algoritmo (*for* e *single*), e ainda mantém o facto do ambiente paralelo ser aberto uma única vez.

```
void OESort_Parallel_V3(int NN, int *A)
{
    int exch, start = 1, i, numThreads;

    #pragma omp parallel
    {
        exch = numThreads = omp_get_num_threads();
        int temp;
        while (1) {

            #pragma omp for
            for (i = start; i < NN-1; i+=2) {
                if (A[i] > A[i+1]) {
                    temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;

                    exch = numThreads;
                }
            }//Implicit Barrier

            if(exch==0 && start==0) break;

            #pragma omp critical
            exch--;

            #pragma omp single //Implicit Barrier
            if (start == 0) start = 1;
            else start = 0;

        }
    }
}
```

Segue exatamente o pensamento da versão 2 mas aproveita as barreiras implicitas evitando desta forma que haja inconsistência no valor do de *exch* que vai ser sempre 0 ou o número de threads.

e. Paralelização Versão 4

Esta é uma quarta versão implementada tem um objectivo diferente dos apresentados até agora. Este faz o unrolling das duas fazes em um só, reduzindo assim para metade o número de inicializações do ambiente paralelo, e ainda aproveita de outras propriedade próprias desta técnica (não vão ser aqui abordadas).

```
void OESort_Parallel_V4(int NN, int *A)
{
    int exch0, exch1=1, trips=0, i;
```

```
    while (exch1) {
        exch0 = exch1 = 0;

        #pragma omp parallel
        {
            int temp;
            #pragma omp for
            for (i=0; i<N-1; i+=2) {
                if (A[i] > A[i+1]) {
                    temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                    exch0 = 1;
                }
            }

            if(exch0 || !trips){
                #pragma omp for
                for (i=1; i<N-1; i+=2) {
                    if (A[i] > A[i+1]) {
                        temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                        exch1 = 1;
                    }
                }
            }
        }
        trips=1;
    }
}
```

Agora são usadas duas *flags* para o caso de ocorrer uma mudança (exch0 e exch1), que vão ser reinicializadas sempre que se entra no ciclo. Faz a primeira fase e no caso de não se efctuar nenhuma mudança o algoritmo dá-se como terminado. Caso contrário ele efectua as duas fases. A primeira execução é uma excepção e tem obrigatoriamente de executar as duas fases, daí advém o uso da variável *trips*.

Esta técnica vai mudar a granularidade e ainda vai cortar em computação extra em mais código de controlo.

f. Paralelização Versão 5

Esta é uma quinta versão implementada e última apresentada. Tem como objectivo diminuir o *overhead* da versão anterior.

```
void OESort_Parallel_V5(int NN, int *A)
{
    int exch0=0, exch1=1, trips=0, i;

    #pragma omp parallel
    {
        while (exch1) {

            #pragma omp barrier
            #pragma omp single
            exch0 = exch1 = 0;

            int temp;

            #pragma omp for
            for (i=0; i<N-1; i+=2) {
                if (A[i] > A[i+1]) {
                    temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                    exch0 = 1;
                }
            }

            if(exch0 || !trips){
                #pragma omp for
                for (i=1; i<N-1; i+=2) {
                    if (A[i] > A[i+1]) {
                        temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                        exch1 = 1;
                    }
                }
            }
```

```
            trips=1;
        }
    }
}
```

Neste algoritmo encontramos os mesmos problemas que em versões anteriores, daí o recurso às mesmas técnicas de resolução. Neste caso só tivemos de adicionar uma barreira antes de depois de reinicializar as variáveis de controlo. O problema é a inconsistência dos dados, podendo não serem detectadas alterações feitas no vector se não tivermos a segunda barreira, implicita pelo *single*. A primeira barreira é para permitir que todas as *threads* entrem no ciclo e evitar uma situação de *deadlock*.

5. Testes

a. Máquina de testes

Foi usado um único nó para a realização de todo os testes necessários neste problema. O nó usado foi o 'compute-652-2' que tem as seguintes caracteristicas gerais:

```
-bash-4.1$ papi_avail | grep CPU
Model string and code    : Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz (62)
CPU Revision             : 4.000000
CPUID Info               : Family: 6  Model: 62  Stepping: 4
CPU Max Megahertz        : 2501
CPU Min Megahertz        : 1200
CPUs per Node            : 20
Total CPUs               : 40
```

Como podemos ver o nó tem 40 CPU's dos quais só foram usados 12 de cada vez no máximo.

a. Tempos

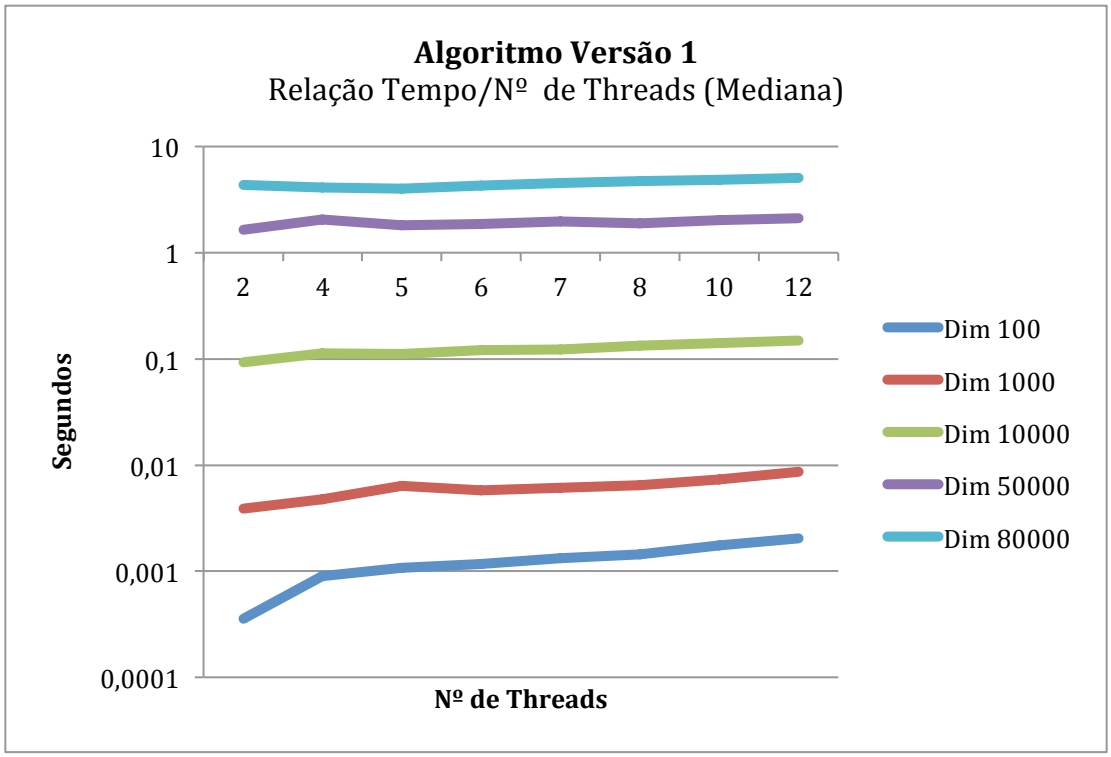
Todos os tempos apresentados neste relatório e no anexo enviado estão na unidadade do Sistema Internacional, em segundos. Foram testados os 5 algoritmos apresentados anteriormente para as dimensões do vector (100, 1000, 10000, 50000, 80000), e ainda para um pequeno número de *threads* para perceber como é que o algoritmo realmente se comporta em maiores números de *threads*. Sendo assim foram feitos testes com 2, 4, 5, 6, 7, 8, 10 e 12 *threads*.

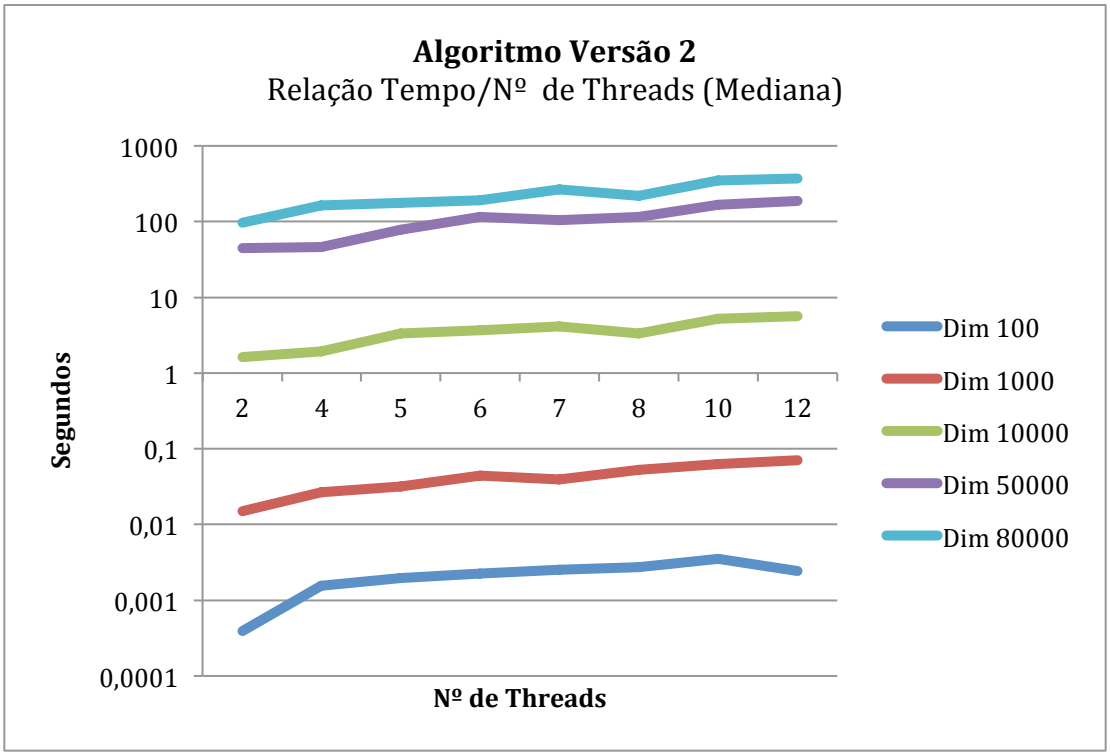
Para ajuda na realização de todos estes testes foi utilizado o seguinte script:

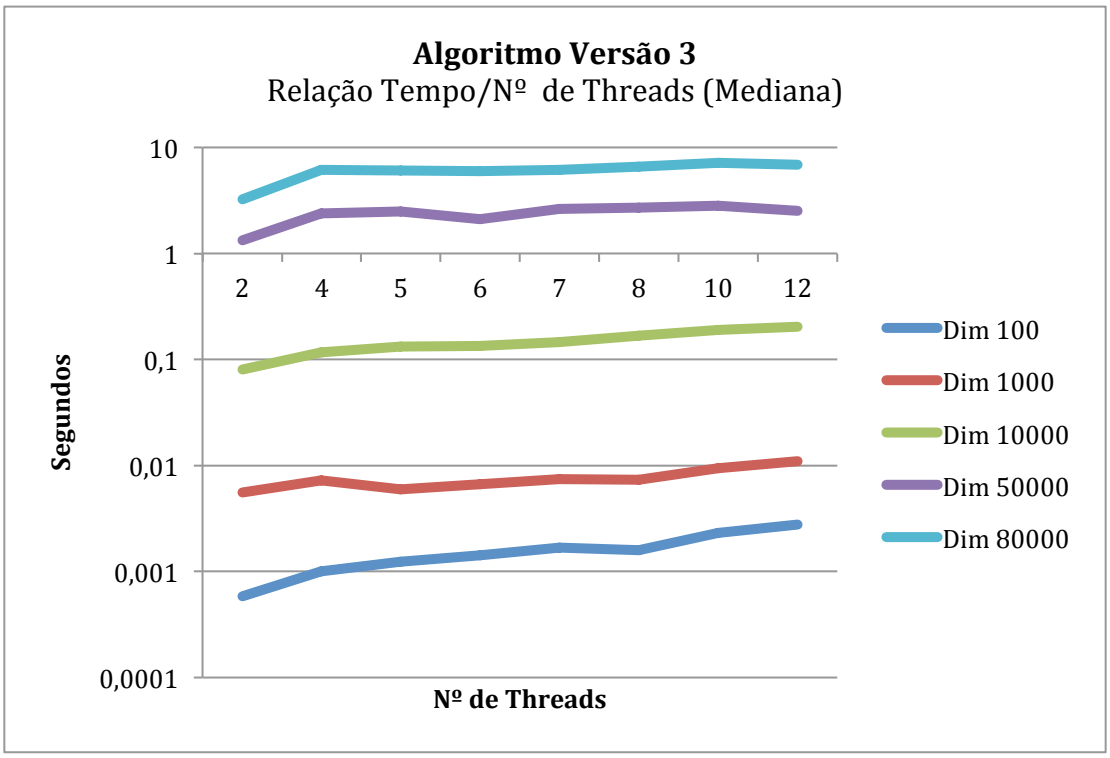
```
 1
 2 ##Nos Usados
 3 cat $PBS_NODEFILE
 4
 5 ##Versões Paralelas
 6 for version in 1 2 3 4 5
 7 do
 8         echo "Algorithm Version: $version "
 9
10         for size in 100 1000 10000 50000 80000
11         do
12                 echo "+++++++++++++++++ Dimensão $size ++++++++++++++++++"
13
14                 for nthreads in 2 4 5 6 7 8 10 12
15                 do
16                         echo "Num threads: $nthreads "
17                         export OMP_NUM_THREADS=$nthreads
18                         ./run $version $size
19                         ./run $version $size
20                         ./run $version $size
21                 done
22         done
23 done
24
25 ##Versão Single
26 echo "+++++++++++++++ SINGLES +++++++++++++++"
27 for size in 100 1000 10000 50000 80000
28 do
29         echo "+++++++++++++++++ Dimensão $size ++++++++++++++++++"
30         ./run 0 $size
31         ./run 0 $size
32         ./run 0 $size
33 done
```

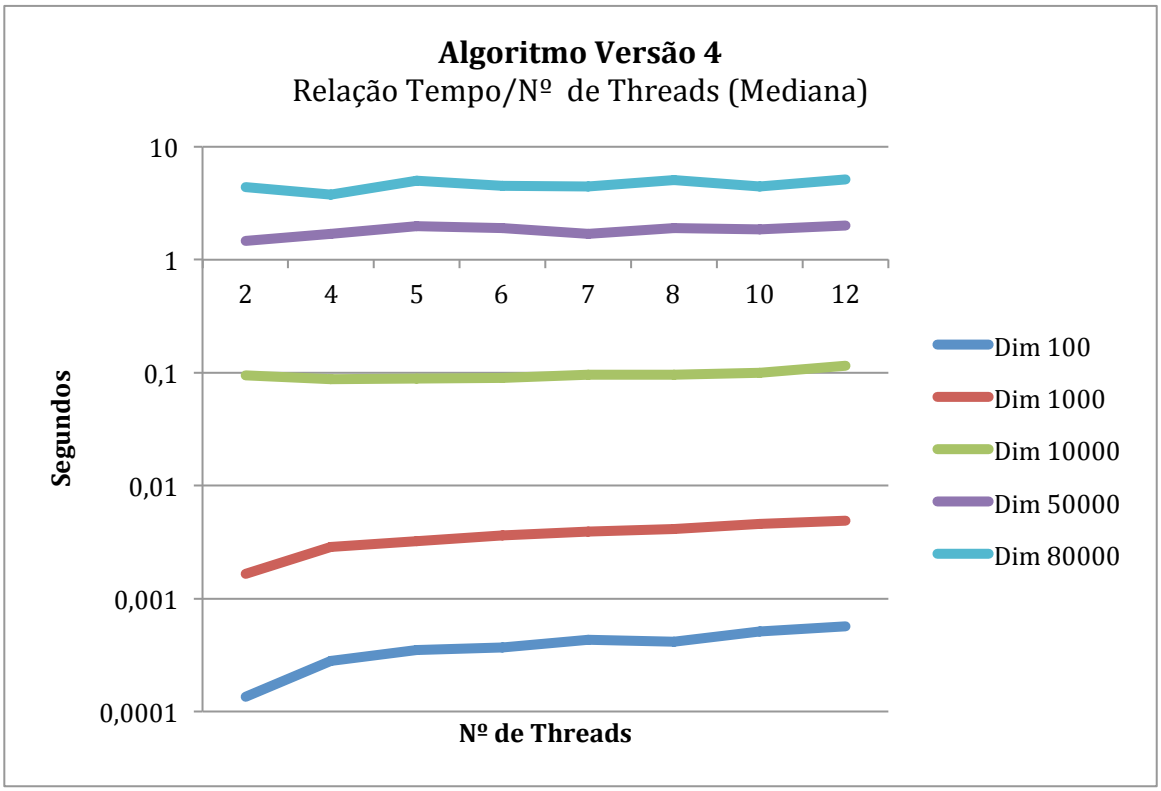
Todos os dados estão em suporte no ficheiro excel enviado como anexo.

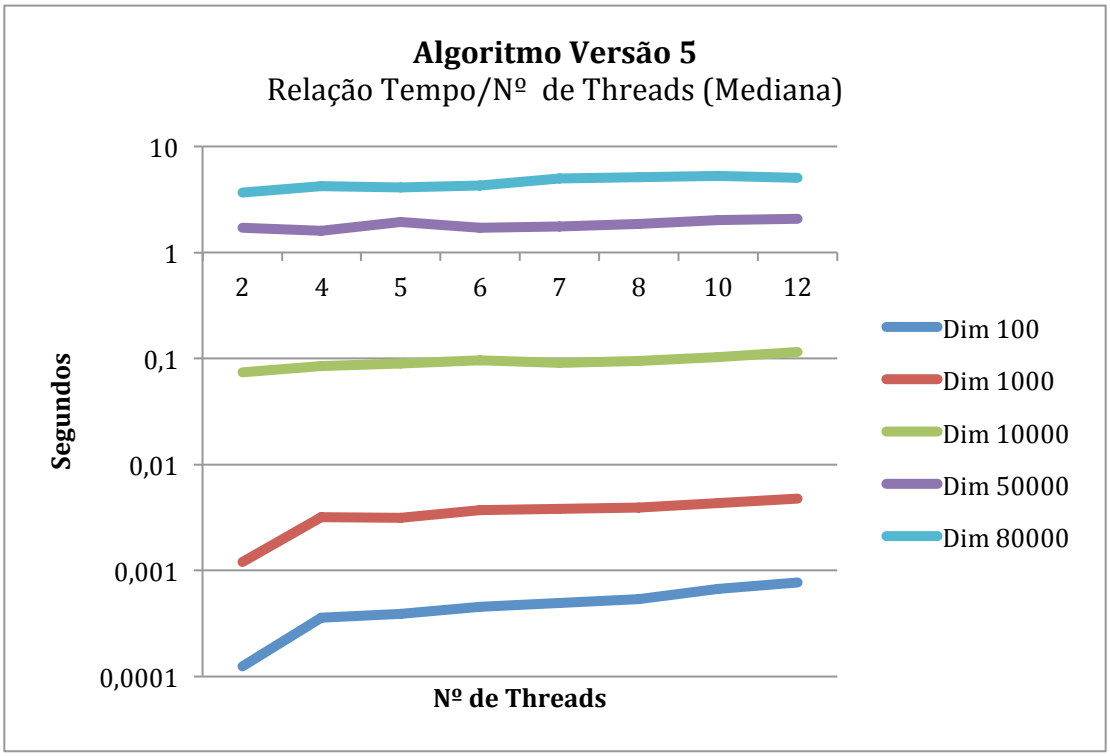
A primeira análise e a que saí mais à vista quando se analisa os gráficos obtido é que em todos os algoritmos, obtemos piores resutados com o aumento do número de *threads*.



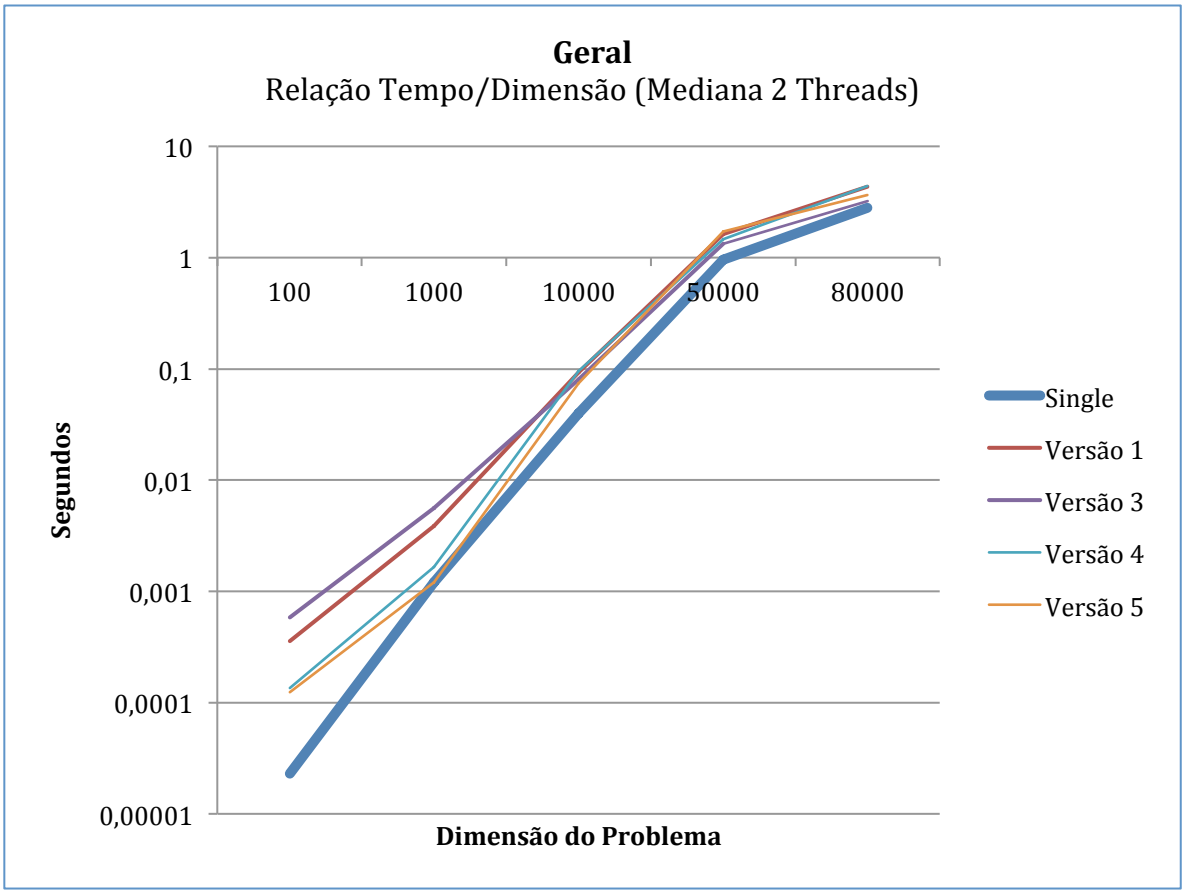



Algoritmo Versão 3
Relação Tempo/Nº de Threads (Mediana)
10
1
0,1
0,01
0,001
0,0001
Segundos
2
4
5
6
7
8
10
12
Nº de Threads
Dim 100
Dim 1000
Dim 10000
Dim 50000
Dim 80000


Algoritmo Versão 4
Relação Tempo/Nº de Threads (Mediana)
10
1
0,1
0,01
0,001
0,0001
Segundos
2
4
5
6
7
8
10
12
Nº de Threads
Dim 100
Dim 1000
Dim 10000
Dim 50000
Dim 80000



Agora vamos comparar para a mesma dimensão o diferente desempenho das mais diversas versões. Como pudemos reparar nos modelos anteriores a versão 2 teve resultaldos substancialmente piores que as restantes versões, por esse motivo essa versão não aparece no seguinte gráfico.



Conclusão

Foi um trabalho interessante que permitiu que fossem abordados temas que apesar de já falados não estavam consolidados e este trabalho veio colmatar essa lacuna. Apesar de nunca ser fácil fazer este tipo de observações, pois é preciso que o programador conheça muito bem o algoritmo que tem em mãos e a partir daí fazer o que mais lhe convém para o seu problema.

Infelizmente não se conseguiu obter nenhum algoritmo que consegui-se obter melhores resultados que a própria versão sequêncial, pois depois de todo o trabalho que uma pessoa teve não existe aquele prémio pessoal de conseguir algo melhor do que foi dado inicialmente.

A versão so fazer unroll provou ser bastante eficaz em relação às restantes mas não o suficiente. Acredito que a versão 5 ainda pode ser melhorada de forma a obter um desempenho superior à versão serial.