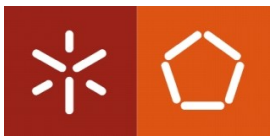
Portfólio

Algoritmos Paralelos

Computação Paralela Distribuída

Mestrado em Engenharia Informática

Universidade do Minho



Duarte Nuno Ferreira Duarte

pg27715

Índice

Introdução global

Este portfólio é relativa às aulas da Unidade Curricular de Algoritmos Paralelos inserida no perfil de Computação Paralela Distribuída do Mestrado em Engenharia Informática na Universidade do Minho. Este portfólio contempla todos os trabalhos realizados ao logo do semestre para esta Unidade Curricular.

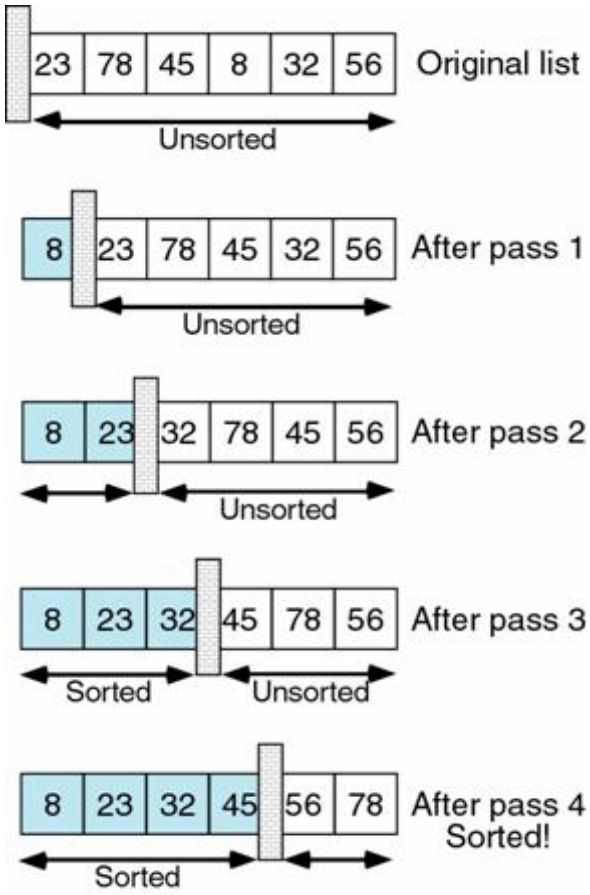
Bubble Sort paralelo (OMP)

Introdução

Com a realização deste trabalho pretende-se elaborar uma versão paralela do algoritmo ***BubbleSort*** que seja eficiente. Dificuldades que possam ser encontradas podem ser no caso de se verificarem com frequência *deadlocks* se não forem tidos em conta alguns problemas simples de sincronização.

Problema

O algoritmo de ordenação ***BubbleSort*** é um algoritmo de ordenação por comparação dos mais simples. Funciona através da comparação de um elemento com o seu seguinte e trocam entre si se o seguinte for menor que o atual. De seguida volta-se a fazer a mesma coisa de forma a que ao fim de n iterações a sequencia esteja



ordenada.

Como é possível observar pela imagem o menor numero de cada iteração da parte não ordenada move-se até à sua posição correcta.

Versão Sequencial

De seguida será possível observar a implementação deste modelo de ordenação em versão sequencial.

```
void Bubble_sort(
    int  A[]  /* in/out */,
    int  n    /* in     */) {
  int list_length, i, temp;
```

Os argumentos passados a esta função são o array A que é o array a ser ordenado e ainda o tamanho da array que é o argumento n.

```
for (list_length = n; list_length >= 2; list_length--)
    for (i = 0; i < list_length-1; i++)
      if (A[i] > A[i+1]) {
        temp = A[i];
        A[i] = A[i+1];
        A[i+1] = temp;
      }
}
```

O primeiro ciclo faz com que a parte do array a ser ordenada vá diminuindo a cada iteração. O segundo vai fazendo trocas sucessivas até que o valor desta iteração chegue à sua posição correcta.

Versão paralela

Agora é possível observar a implementação deste modelo de ordenação em versão paralela sem desenrolar o ciclo.

```
void Bubble_sort_parallel(
                    int  a[]  /* in/out */,
                    int  n    /* in     */) {
    int list_length=n, i, temp;

    int numLock=0;
    int ordered=0;
    int change=0;

    int bloco=n/NUM_THR;

    int offsetValue[NUM_THR];
    omp_lock_t locks[NUM_THR+1];
```

Os argumentos de entrada são o array a ser ordenado e o tamanho desse mesmo array. São alocados dois arrays, um para ficar com os locks relativos aos blocos e um outro para saber qual a posição dos blocos.

```
    for(i=0;i<=NUM_THR;i++)
        omp_init_lock(&(locks[i]));

    for (i=0;i<NUM_THR;i++)
        offsetValue[i]=(i+1)*bloco;
```

Nesta fase são iniciados os locks dos diversos blocos e no array offsetValue é posto o valor em que começa cada bloco.

```
#pragma omp parallel shared(list_length,a,bloco) private(i,numLock,change,temp)
    while (!ordered){
        change = 0;
        numLock = 0;
```

```
        omp_set_lock(&(locks[0]));


        for (i = 0; i < list_length-1; i++){
           if (a[i] > a[i+1]) {
              temp = a[i];
              a[i] = a[i+1];
              a[i+1] = temp;
              change=1;
           }
           if (i==offsetValue[numLock]) {
              omp_set_lock(&(locks[(numLock+1)]));
              omp_unset_lock(&(locks[(numLock)]));
              numLock++;
           }
        }
        if (change==0)
           ordered=1;
        list_length--;
        omp_unset_lock(&(locks[(numLock)]));
     }
  }
```

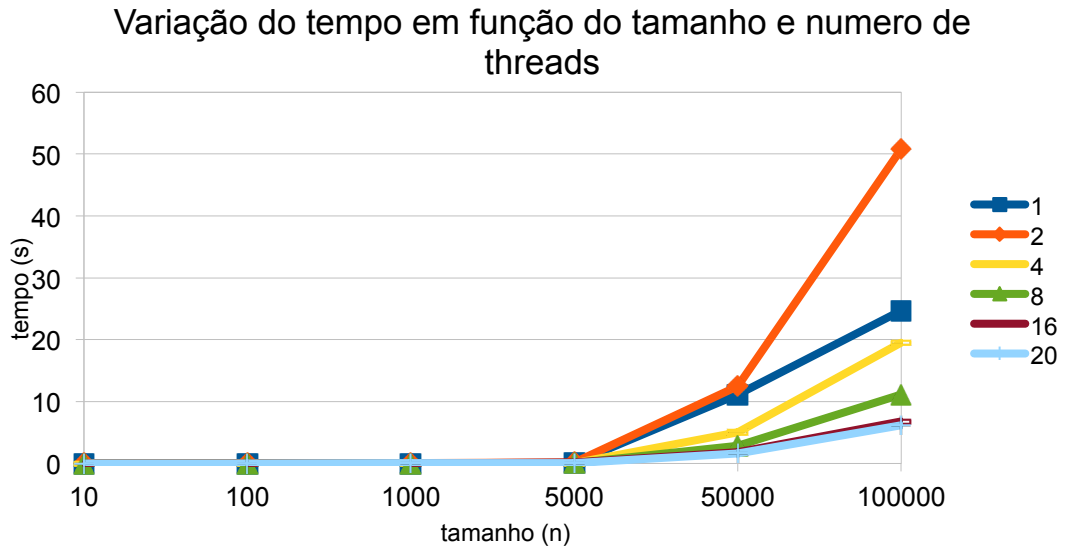
Aqui é que se dá a parte da execução em paralelo. Recorrendo à primitiva do OMP que permite criar um bloco de execução paralela define-se que o tamanho da lista (list_lenght) o array e o tamanho de bloco são variáveis partilhadas pelas diferentes threads. Mas cada uma das threads terá durante a execução a sua variável i, o bloco em que está, o valor da troca e a temporária como variáveis privadas. A primeira thread faz lock ao primeiro bloco o que faz com que todas as outras fiquem à espera que aquele bloco seja libertado e só quando a primeira thread passar para o segundo bloco é que a segunda thread pode ir para o primeiro bloco e assim sucessivamente. No fim todas libertam os locks e avançam para a iteração seguinte.

Resultados

A medição dos resultados foi efectuada usando o omp_get_wtime() e a performance recorreu-se ao OMPP.

tamanho/threads	1	2	4	8	16	20
10	0,001	0,025	0,001	0,002	0,002	0,003
100	0,001	0,023	0,002	0,002	0,002	0,002
1000	0,006	0,03	0,007	0,006	0,01	0,015
5000	0,113	0,202	0,071	0,038	0,128	0,062
50000	11,145	12,51	5,056	2,869	1,872	1,569
100000	24,743	50,817	19,544	11,109	6,811	6,213

Observando a tabela acima que possui o tempo de execução em função do tamanho do array e do numero de threads é possível ver para valores altos a execução com maior numero de threads possui os melhores resultados.



Pelo gráfico é possível constatar uma vez mais que o fio de execução correspondente às 20 threads tem sempre melhores resultados para valores grandes, pois para valores pequenos não vale o custo de criação das threads e dos respectivos arrrays de locks.

```
##BEG overhead analysis report

Total runtime (wallclock) [secs],0.012310
Number of threads,24
Number of parallel regions,1
Parallel coverage [secs],0.011565
Parallel coverage [percent],93.947552

##BEG parallel regions sorted by wallclock time
,Type,Wallclock,(%)
```

```
R00001,PARALLEL,0.011565,93.947552
SUM,0.011565,,93.947552
##END parallel regions sorted by wallclock time

##BEG overheads for parallel region
,Total,Ovhds,(%),Synch,(%),Imbal,(%),Limpar,(%),Mgmt,(%)
R00001,0.277559,0.016415,5.914095,0.000000,0.000000,0.000682,0.245669,0.0000
00,0.000000,0.015733,5.668426
##END overheads for parallel region

##BEG overheads for whole program
,Total,Ovhds,(%),Synch,(%),Imbal,(%),Limpar,(%),Mgmt,(%)
R00001,0.277559,0.016415,5.556147,0.000000,0.000000,0.000682,0.230800,0.0000
00,0.000000,0.015733,5.325347
SUM,0.277559,0.016415,5.556147,0.000000,0.000000,0.000682,0.230800,0.000000
,0.000000,0.015733,5.325347
##END overheads for whole program

##END overhead analysis report
```

Recorrendo a esta parte do relatório do OMPP é possível verificar que aproximadamente 94% do código é executado em paralelo.

```
[=01],R00001,PARALLEL,bubble_parallel.c,183,206,
TID,execT,execC,bodyT/I,bodyT/E,exitBarT,startupT,shutdwnT,taskT
0,0.011570,1,0.010874,0.010874,0.000038,0.000648,0.000007,0.000000
1,0.011570,1,0.010901,0.010901,0.000012,0.000647,0.000008,0.000000
2,0.011570,1,0.010909,0.010909,0.000006,0.000645,0.000008,0.000000
3,0.011570,1,0.010866,0.010866,0.000047,0.000647,0.000007,0.000000
4,0.011570,1,0.010913,0.010913,0.000002,0.000645,0.000009,0.000000
5,0.011570,1,0.010882,0.010882,0.000031,0.000647,0.000008,0.000000
6,0.011570,1,0.010911,0.010911,0.000004,0.000645,0.000007,0.000000
7,0.011570,1,0.010858,0.010858,0.000055,0.000647,0.000007,0.000000
8,0.011570,1,0.010865,0.010865,0.000050,0.000645,0.000008,0.000000
9,0.011570,1,0.010899,0.010899,0.000014,0.000647,0.000007,0.000000
10,0.011570,1,0.010874,0.010874,0.000041,0.000645,0.000008,0.000000
11,0.011570,1,0.010879,0.010879,0.000034,0.000647,0.000007,0.000000
12,0.011570,1,0.010875,0.010875,0.000046,0.000645,0.000004,0.000000
13,0.011570,1,0.010878,0.010878,0.000035,0.000647,0.000007,0.000000
14,0.011570,1,0.010890,0.010890,0.000025,0.000645,0.000007,0.000000
15,0.011570,1,0.010903,0.010903,0.000010,0.000647,0.000007,0.000000
16,0.011570,1,0.010873,0.010873,0.000027,0.000660,0.000007,0.000000
17,0.011570,1,0.010907,0.010907,0.000008,0.000645,0.000008,0.000000
18,0.011570,1,0.010914,0.010914,0.000002,0.000645,0.000007,0.000000
19,0.011570,1,0.010869,0.010869,0.000044,0.000647,0.000007,0.000000
20,0.011570,1,0.010884,0.010884,0.000029,0.000647,0.000008,0.000000
21,0.011570,1,0.010829,0.010829,0.000051,0.000680,0.000008,0.000000
```

```
22,0.011570,1,0.010859,0.010859,0.000054,0.000647,0.000008,0.000000
23,0.011570,1,0.010897,0.010897,0.000016,0.000647,0.000007,0.000000
SUM,0.277679,24,0.261210,0.261210,0.000682,0.015554,0.000179,0.000000
```

Recorrendo a esta parte do relatório do OMPP é possível verificar que todas as threads tiveram aproximadamente a mesma distribuição trabalho.

Conclusão

Com a realização deste trabalho foi possível observar que a paralelização do código faz com que o processamento (neste caso) seja mais eficiente e mais rápido. Mas quando o tamanho de *input* é pequeno o custo da criação das *threads* não compensa.

Foi bom neste trabalho recorrer à ferramenta de análise de performance que é o OMPP que permitiu verificar os balanceamentos entre as threads e a quantidade de código que era executada em paralelo.

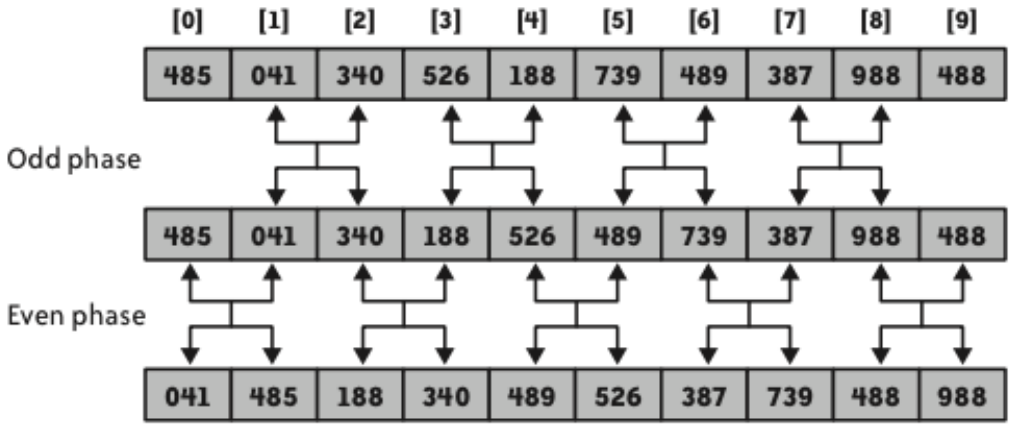
Odd-Even Sort paralelo (OMP)

Introdução

Com a realização deste trabalho pretende-se elaborar uma versão paralela do algoritmo Odd-even que seja eficiente. Dificuldades que possam ser encontradas podem ser no caso de se verificarem com frequência *deadlocks* se não forem tidos em conta alguns problemas simples de sincronização

Problema

O algoritmo de ordenação *Odd-Even* é um algoritmo de ordenação por comparação. Funciona através da comparação de todos os de elementos adjacentes e se um par está na ordem os elementos são trocados. O próximo passo repete isso para os pares. Depois alterna ímpar-par e par-ímpar até esteja ordenado. É uma algoritmo semelhante ao *Bubble-Sort* (implementado anteriormente).



Como é possível observar pela imagem existem duas fazes distintas, a fase par e a fase impar, e assim é possível observar algum potencial de programação paralela.

Versão Sequencial

De seguida será possível observar a implementação deste modelo de ordenação em versão sequencial.

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

O parâmetro NN é o tamanho do *array* que será ordenado e o parâmetro A é o apontador para o *array* a ser ordenado. São declaradas algumas variáveis necessárias ao algoritmo. O *exch* servirá como *flag* para saber se a iteração fez alguma troca. O *start* indica a posição onde o ciclo começará. E por fim, o *temp* é uma variável auxiliar para guardar o valor enquanto se faz a troca entre duas posições.

```
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;
    }
}
```

O algoritmo decorrerá enquanto o houverem trocas ao longo do *array*. Verifa-se o valor da posição adjacente e se esta for menor do que a posição actual troca-se os valores e muda-se o valor da variável *exch* para que seja marcado que houve pelo menos um troca na iteração.

```
if (start == 0) start = 1;
else start = 0;
}
```

```
    }
```

O ciclo *while* acaba a iteração com um um *if*, em que se faz a transição entre a fase par e a fase impar. Se estava em fase para passa para fase impar e vice versa.

Versão paralela

Agora é possível observar a implementação deste modelo de ordenação em versão paralela sem desenrolar o ciclo.

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

As variáveis declaradas com o nome igual à versão sequencial têm a mesma função que na versão sequencial.

```
#pragma omp parallel
{
int temp;
exch = omp_get_num_threads();
```

A directiva *omp parallel* indica que esta região será executada por um grupo de *threads*. É atribuído à variável *exch* o numero de *threads* que executarão aquela região.

```
while (1) {
if(exch <= 0 && start == 0)
        break;
#pragma omp critical
exch--;
#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;
        #pragma omp critical
        exch = omp_get_num_threads();
        //printf("alterei %d\n",exch);
        }
```

```
}
```

Este ciclo decorrerá enquanto houver trocas. O decremento da variável *exch* é feito numa região critica pois apenas uma das *threads* o pode fazer de cada vez. É colocada uma barreira para que as *threads* sincronizem, sem esta barreira o ciclo pode entrar em *deadlock*. De seguida com a primitiva *omp for* dá se entrada num ciclo *for* que será executado por varias *threads*. Se o adjacente tiver um valor inferior à posição actual os valores são trocados. Por fim, numa região critica, é obtido o valor do numero de *threads* para o colocar na variável *exch*.

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

Por ultimo, com a primitiva *omp single*, é efectuada a troca entre fases, fase impar passa a par ou se estiver na fase par passa a impar. É feito apenas por uma das *threads* por isso é usada esta primitiva.

Versão paralela desenrolada

Por fim é possível observar a implementação deste modelo de ordenação em versão paralela com o ciclo *for* desenrolado.

```
void OESort_parallel(int NN, int *A)
{
   int exchO, exchE = 1, turns = 0, i;
```

Nesta versão a variável exch divide-se em duas, a exchO e a exchE que servirá para saber se houve alguma troca na parte par e na parte impar (O – par e E – impar). A variável *turns* conta o numero de iterações efectuadas.

```
while (exchE) {
      exchO = 0;
      exchE = 0;

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

Enquanto houver trocas na parte impar é executado um bloco de código paralelo que verifica se é necessário haver trocas com o adjacente e se for necessário faz essa troca. Se houver troca nesta parte a variável *exchO* é colocada com o valor a 1.

```
if(exchO || !turns){
            #pragma omp for
            for (i = 1; i < NN-1; i+=2) {
               if (A[i] > A[i+1]) {
                  temp = A[i];
                  A[i] = A[i+1];
                  A[i+1] = temp;
                  exchE = 1;
               }
            }
         }
      }
      turns=1;
   }
```

```
}
```

Se houver trocas no primeiro *for* ou não forem feitas travessias então é efectuado o segundo *for* e troca, na fase par, se for necessário.

Resultados

A medição dos resultados foi efectuada usando o omp_get_wtime(). Os tempos são apresentados em segundos. No eixo do X será colocado o tamanho do input (o tamanho do *array* a ser ordenado) e no eixo do Y será o tempo correspondente ao processamento

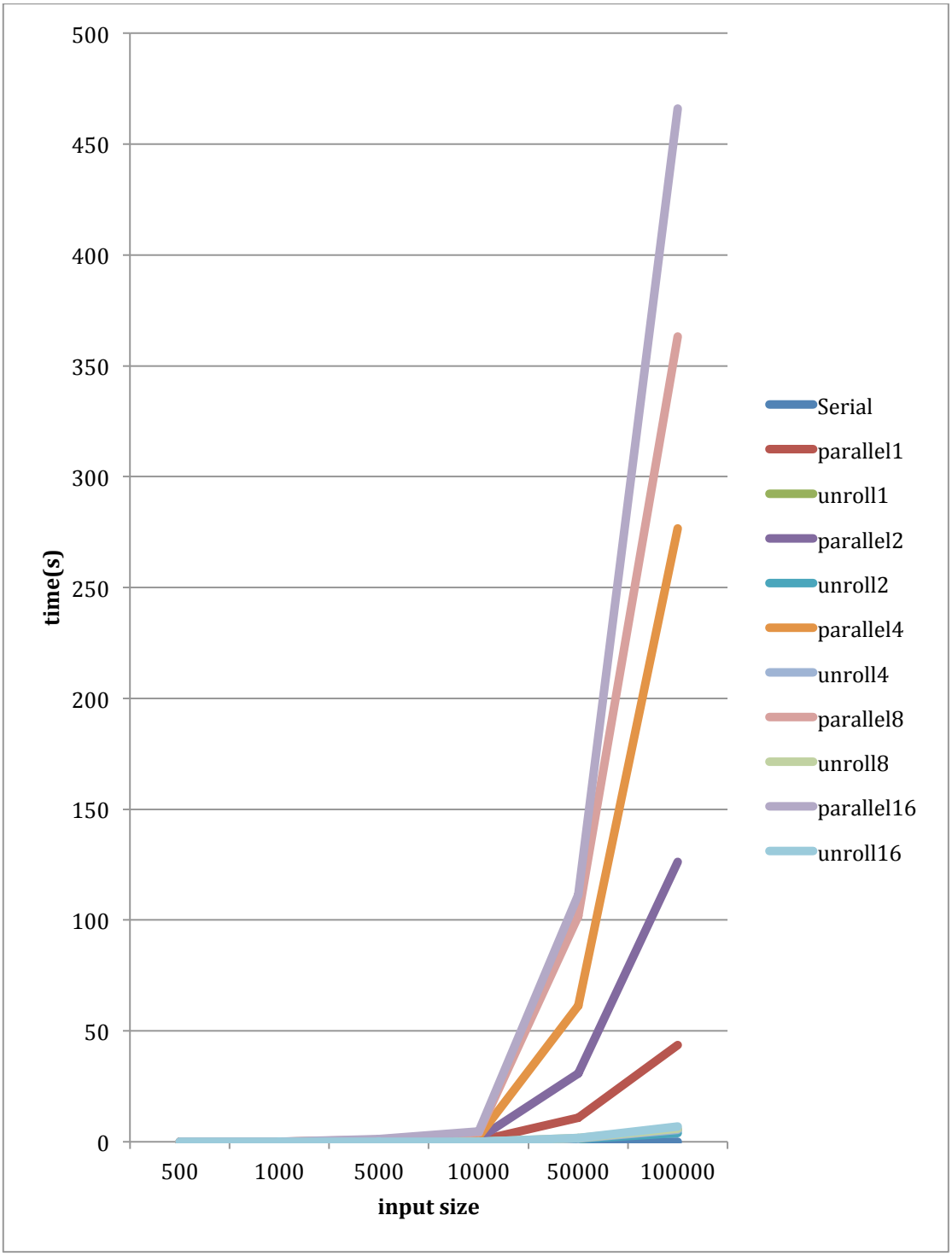


Tabela de resultados

Size	Serial	#THREADS										MIN	EP
		1		2		4		8		16			
		parallel	unroll	parallel	unroll	parallel	unroll	parallel	unroll	parallel	unroll		
500	0.000273736	0,00260192	0,000483744	0,00458469	0,000792137	0,00893269	0,00126382	0,017163	0,00236027	0,0263329	0,00354059	0,000483744	103360
1000	0.00107505	0,00933208	0,0014197	0,0198028	0,00182915	0,0277099	0,00255424	0,0506317	0,00355097	0,0663387	0,00422397	0,0014197	704374,
5000	0.0133838	0,11103	0,018464	0,336508	0,01523	0,651244	0,0235778	1,04034	0,0310301	1,22043	0,0313669	0,01523	328299,
10000	0.0722914	0,440484	0,0540772	1,35014	0,0447605	2,47953	0,0705723	4,0898	0,0894816	4,72742	0,0954972	0,0447605	223411,
50000	1.23835	10,9233	1,2887	30,7555	1,2285	61,494	1,43769	101,914	1,64495	111,462	1,72077	1,2285	40700,
100000	6.37177	43,7072	5,31863	126,299	4,01054	276,762	5,33758	363,28	6,01113	466,008	6,82565	4,01054	24934,2

Se observarmos o gráfico e a tabela correspondente podemos rapidamente perceber que a versão *unroll* é melhor para qualquer tamanho. Sendo que nem sempre é melhor ter maior numero de *threads*. Para inputs pequenos o custo da criação de *threads* não compensa muito. Olhando para a coluna EPS (elementos por segundo) é possível observar que para valores de *input* maiores é mais eficiente.

Conclusão

Com a realização deste trabalho foi possível observar que a paralelização d
faz com que o processamento (neste caso) seja mais eficiente e mais rápi
quando o tamanho de *input* é pequeno o custo da criação das *thre*
compensa.

Agora uma opinião pessoal, no inicio pensei que os resultados obtidos
paralelização fossem melhores do que os que no fim se obtiveram.

Bubble Sort (Pthreads)

Introdução

Com a realização deste trabalho pretende-se elaborar uma versão paralela do algoritmo Odd-even que utilize pthreads. Dificuldades que possam ser encontradas podem ser no caso que já foi verificado na versão deste algoritmo em OMP que se trata de se verificarem com frequência *deadlocks* se não forem tidos em conta alguns problemas simples de sincronização. Nesta versão não será apresentado o problema pois trata-se do mesmo apresentado anteriormente.

Versão paralela (Pthreads)

Será possível observar a implementação deste modelo de ordenação em versão paralela recorrendo a pthreads.

```
int offsetValue[NUM_THR];
pthread_t thread[NUM_THR];
int iThread[NUM_THR];
int numLockThread[NUM_THR];
int changeThread[NUM_THR];
int tempThread[NUM_THR];
int N;
pthread_mutex_t locks[NUM_THR+1];
```

Estas variáveis são todas globais para que as diferentes threads tenham acesso a elas.

```
for(i=0; i<NUM_THR;i++)
   pthread_mutex_init(&locks[i], NULL);

bloco=n/NUM_THR;
N=n;

for (i=0;i<NUM_THR;i++)
   offsetValue[i]=(i+1)*bloco;

for(i = 0; i < NUM_THR; i++){
   pthread_create(&thread[i], NULL, &Bubble_sort, a);
}
for(i = 0; i < NUM_THR; i++){
   pthread_join(thread[i], NULL);
}
```

Na main são todas inicializadas, os locks são criados, o offsetValue preenchido com o tamanho correspondente de blocos. E de seguida são criadas as pthreads com recorrendo ao pthread_create. Depois a main espera pelo fim de todas as threads.

```
void Bubble_sort_parallel(int  a[]) {
   int list_length=N , i, temp;

   int numLock=0;
   int ordered=0;
   int change=0;

while (!ordered){
      change = 0;
      numLock = 0;
      pthread_mutex_lock(&(locks[0]));

      for (i = 0; i < list_length-1; i++){
         if (a[i] > a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
            change=1;
         }
         if (i==offsetValue[numLock]) {
            pthread_mutex_lock(&(locks[(numLock+1)]));
            pthread_mutex_unlock(&(locks[(numLock)]));
            numLock++;
         }
      }
      if (change==0)
         ordered=1;
      list_length--;
      pthread_mutex_unlock(&(locks[(numLock)]));
   }
}
```

Este algoritmo em pthreads é em tudo semelhante ao da versão em OMP. Enquanto não está ordenado a primeira thread a conseguir fazer o lock ao primeiro bloco executa nesse bloco e depois de libertar e passar para o próximo bloco é que uma outra thread pode entrar e executar nesse bloco. Só assim é possível garantir que não há corridas concorrentes dentro do mesmo bloco. Quando o array estiver ordenado todos os locks são libertados.

Resultados

A medição dos resultados foi efectuada usando o omp_get_wtime(). Os tempos são apresentados em segundos.

2p	4p	8p	16p	20p	tamanho/threads
0,05	0,001	0,002	0,002	0,002	10
0,04	0,002	0,002	0,003	0,002	100
0,035	0,007	0,01	0,015	0,002	1000
0,276	0,071	0,056	0,128	0,005	5000
13,25	4,97	2,61	1,431	1,2	50000
30,21	17,29	9,013	6,811	5,01	100000

Observando a tabela é possível ver que para maiores tamanhos o numero de threads faz toda a diferença. Para tamanhos pequenos não compensa criar pthreads pois o tempo perdido a criar essas threads não compensa no tempo final.

Conclusão

Com a realização deste trabalho foi possível utilizar algo de "novo" que é o caso de pthread. Quando o tamanho de *input* é pequeno o custo da criação das *threads* não compensa.

Numa opinião mais pessoal, pensei que os resultados obtidos com a paralelização em pthreads fossem bastante melhores do que os obtidos com OMP.

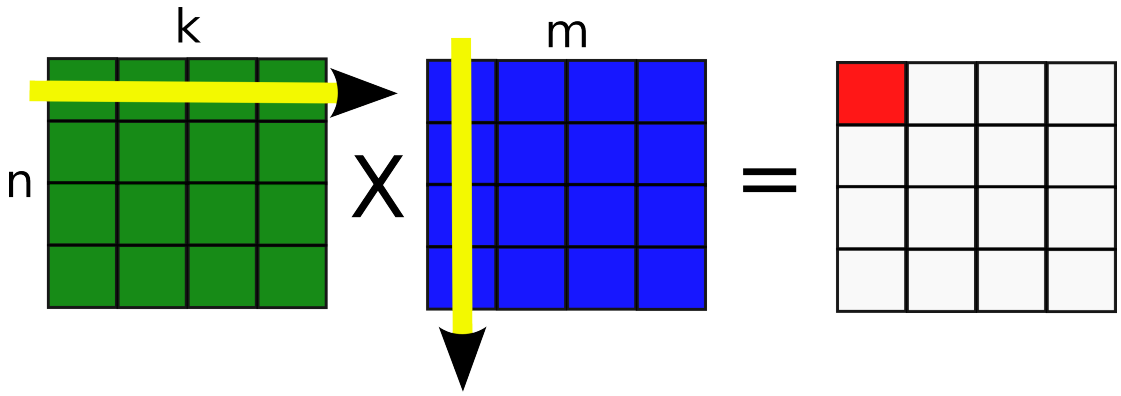
Matrix multiply (OMP)

Introdução

Com a realização deste trabalho pretende-se elaborar uma versão paralela do algoritmo de multiplicação de matrizes que seja eficiente recorrendo a OMP. Dificuldades que possam ser encontradas devem-se a que a multiplicação de matrizes é na maior parte dos casos “inimiga” da cache.

Problema

A multiplicação de matrizes trata-se de um problema que apesar de simples requer bastante trabalho computacional e é utilizado em inúmeros casos em diferentes contextos.



O resultado da primeira célula (1,1) é a multiplicação do valor da primeira linha com a primeira coluna das duas matrizes a serem multiplicadas.

Versão paralela (OMP)

De seguida é apresentada a versão OMP da multiplicação de matrizes.

```
void Omp_mat_vect(double A[], double x[], double y[],
     int m, int n, int thread_count) {
  int i, j;
  double start, finish, elapsed, temp;

  GET_TIME(start);
#  pragma omp parallel for num_threads(thread_count)  \
     default(none) private(i, j, temp)  shared(A, x, y, m, n)
  for (i = 0; i < m; i++) {
     y[i] = 0.0;
     for (j = 0; j < n; j++) {
        temp = A[i*n+j]*x[j];
        y[i] += temp;
     }
  }

  GET_TIME(finish);
  elapsed = finish - start;
  printf("Elapsed time = %e seconds\n", elapsed);

}
```

Para isso é adicionada uma primitiva omp parallel para criar uma região paralela em que as variáveis i, j e temp são privadas na execução de cada thread e a matrix A e as variáveis x, y, m e n são partilhadas por todas as threads.

Resultados

São apresentados os resultados para a versão série e OMP com diferentes optimizações. De seguida são apresentados primeiro os resultados sequenciais.

original values	8M x 8		8k x 8k		8 x 8M	
Threads	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0,3220	1,0000	0,2640	1,0000	0,3330	1,0000
2	0,2190	0,7352	0,1890	0,6984	0,3000	0,5550
4	0,1410	0,5709	0,1190	0,5546	0,3030	0,2748

Estes são os resultados relativos à execução com a versão OMP com –O0.

"-O0"						
best values	8M x 8		8k x 8k		8 x 8M	
OMPThreads	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0,3848	1,0000	0,3501	1,0000	0,3531	1,0000
2	0,1930	0,9971	0,1753	0,9983	0,2445	0,7219
4	0,1580	0,6089	0,1710	0,5117	0,3211	0,2749

Estes são os resultados relativos à execução com a versão OMP com –O3.

"-O3"						
best values	8M x 8		8k x 8k		8 x 8M	
OMPThreads	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0,1044	1,0000	0,0897	1,0000	0,0822	1,0000
2	0,0829	0,6299	0,0896	0,5004	0,1001	0,4107
4	0,0826	0,3161	0,0897	0,2500	0,0782	0,2630

Conclusão

Como era esperado a versão em OMP é mais rápida para valores grandes do que a versão série. Também é verdade que eficiência se vai perdendo não sendo esta parte o resultado esperado. Era de esperar uma maior eficiência aliada aos melhores resultados.

Matrix multiply (Pthreads)

Introdução

Com a realização deste trabalho pretende-se elaborar uma versão paralela do algoritmo de multiplicação de matrizes que seja eficiente recorrendo a Pthreads. Dificuldades que possam ser encontradas devem-se a que a multiplicação de matrizes é na maior parte dos casos “inimiga” da cache.

Versão paralela (Pthreads)

É agora apresentada a versão paralela da multiplicação de matrizes recorrendo a pthreads.

```
int area=m/thread_count;

  for(i = 0; i < thread_count; i++){
    startPos=i*area;
    finishPos=(i+1)*area-1;
    argums = (int*)malloc(sizeof(int)*2);
    argums[0] = startPos;
    argums[1] = finishPos;
    pthread_create(&thread[i], NULL, &Pthreads_mat_vect, argums);
  }
  for(i = 0; i < thread_count; i++)
    pthread_join(thread[i], NULL);
```

É guardada na variável área o tamanho de cada bloco, que se trata do tamanho total a dividir pelo numero de threads.
São passados dois argumentos a cada pthread o local onde começa e onde acaba.
Depois a main espera pelo fim de cada uma das threads.

```
void Pthreads_mat_vect(int *argums) {
  int j=0;
  int i=argums[0];
  int fin=argums[1];
  double temp;

  for(i; i<=fin;i++){
    y[i] = 0.0;
    for (j = 0; j < n; j++) {
      temp = A[(i*n)+j]*x[j];
      y[i] += temp;
    }
  }
}
```

Cada pthread calcula a multiplicação do seu bloco.

Resultados

São apresentados os resultados para a versão em Pthreads com diferentes optimizações. De seguida são apresentados primeiro os resultados para –O0.

best values	8M x 8		8k x 8k		8 x 8M	
PTHREADS	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0,4022	1,0000	0,3734	1,0000	0,3130	1,0000
2	0,2022	0,9948	0,1888	0,9887	0,3731	0,4194
4	0,1003	1,0028	0,0990	0,9431	0,3780	0,2070

Estes são os resultados de pthreads com optimização –O3.

best values	8M x 8		8k x 8k		8 x 8M	
PTHREADS	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0,0856	1,0000	0,0968	1,0000	0,1174	1,0000
2	0,0572	0,7478	0,0493	0,9817	0,0594	0,9880
4	0,0328	0,6533	0,0290	0,8349	0,0376	0,7811

Comparação com a versão OMP

OMP

cache

Em OMP observando a aba seguinte referente à cache, verifica-se que para qualquer numero de threads se observe mais falhas na cache na versão de 8x8M que é a mais prejudicial para a cache.

tempos

O mais rápido em OMP é a versão de 8Mx8 e com 4 threads. Compreende-se que seja o 8Mx8 porque está é a forma "mais amiga" da cache e assim tira-se partido do facto de os valores a ser acedigos de seguida estejam na cache nas iterações seguintes.

Pthreads

cache

Em Pthreads observando a aba seguinte referente à cache, verifica-se que para qualquer numero de threads se observe mais falhas na cache na versão de 8x8M que é a mais prejudicial para a cache.

tempos

O mais rápido em Pthreads é a versão de 8Mx8 e com 4 threads. Compreende-se que seja o 8Mx8 porque está é a forma "mais amiga" da cache e assim tira-se partido do facto de os valores a ser acedigos de seguida estejam na cache nas iterações seguintes.

Global

cache

Ambas as versões apresentam ao nivel das falhas à cache as mesmas combinações, 8x8M no que se refere ao tamanho.

tempos

Ambas as versões apresentam ao nivel dos tempos as mesmas combinações, 8Mx8 no que se refere ao tamanho e ambos com 4 threads.

Conclusão

Como era de esperar a multiplicação de matrizes é mais eficiente nos formatos que são “amigos” da cache. Os melhores tempo são obtidos com mais threads, apesar de que era esperado que eficiência fosse melhor do que realmente foi. Uma vez mais foi bom voltar a entrar em contacto com pthreas.

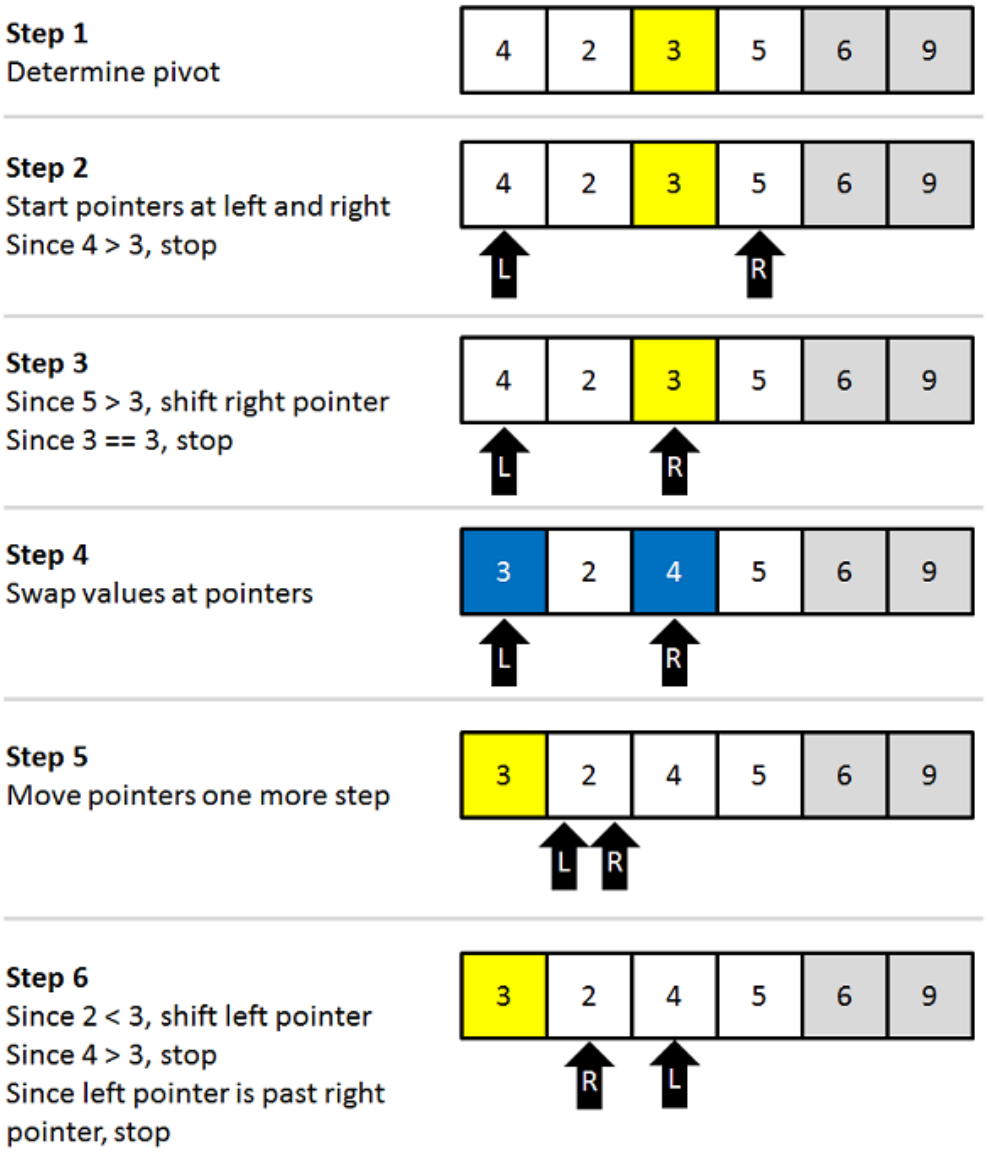
Quicksort (OMP)

Introdução

Com a realização deste trabalho pretende-se elaborar uma versão paralela do algoritmo de ordenação QuickSort que seja eficiente recorrendo a OMP. Dificuldades que possam ser encontradas devem-se ao facto de como este algoritmo recorre a chamadas recursivas pode haver conflitos entre as diferentes threads.

Problema

O algoritmo ***Quicksort*** é um algoritmo de ordenação muito rápido e eficiente. Trata-se de um algoritmo por comparação não estável. O ***Quicksort*** adota a estratégia de "divisão e conquista". Primeiro dá-se a escolha do pivô, um elemento da lista de escolhido de forma aleatória. Depois arranja-se a lista de forma a que os elementos à sua esquerda sejam todos menores do que ele e à direita todos maiores e assim obtém-se duas sublistas não ordenadas. Este é o processo da divisão. De seguida, recursivamente, ordena-se a sublista da esquerda e a sublista da direita. Este algoritmo é finito pois listas com um elemento estão ordenadas.



Versão sequencial

De seguida é possível observar o algoritmo sequencial de ordenação do ***quickSort***.

```
void quicksort(int lo, int hi)
{
   int i=lo,j=hi,h;
   int x=array[(lo+hi)/2];

   //partition
   do
   {
      while(array[i]<x) i++;
      while(array[j]>x) j--;
      if(i<=j)
      {
         h=array[i]; array[i]=array[j]; array[j]=h;
         i++; j--;
      }
   } while(i<=j);

   //recursion
   if(lo<j) quicksort(lo,j);
   if(i<hi) quicksort(i,hi);
}
```

A variável x fica com o valor do pivô, e de seguida verifica-se os valores que são maiores ou menores do que o x e se necessário troca-los. No fim desta verificação chamada de partição, avança-se para a parte recursiva que se trata de chamar a função sobre as duas sublistas.

Versão paralela (OMP)

A versão paralela do algoritmo recorrendo a primitivas OMP.

```
int main() {

   initialize();

   double time = omp_get_wtime();
   #pragma omp parallel
   #pragma omp master
   quicksort(0,SIZE-1);
   printf("Time=%f\n", omp_get_wtime()-time);

   validate();
}
```

Cria-se uma região de trabalho paralelo e depois apenas o master chama a função quicksort.

```
void quicksort(int lo, int hi)
{
   int i=lo,j=hi,h;
   int x=array[(lo+hi)/2];

   //partition
   do
   {
      while(array[i]<x) i++;
      while(array[j]>x) j--;
      if(i<=j)
      {
         h=array[i]; array[i]=array[j]; array[j]=h;
         i++; j--;
      }
   } while(i<=j);

   //recursion
   #pragma omp task
   if(lo<j) quicksort(lo,j);
   #pragma omp task
   if(i<hi) quicksort(i,hi);
}
```

A partição é feita por apenas uma thread mas depois a chamada recursiva tratam-se de tarefas. Ou seja cada thread pode executar uma tarefa.

Conclusão

Em relação ao código sequencial pouco muda, apenas são adicionadas algumas primitivas OMP. O algoritmo sequencial já é muito eficiente portanto uma versão paralelizada teria como era esperado uma melhor performance e uma redução substancial no tempo de execução.

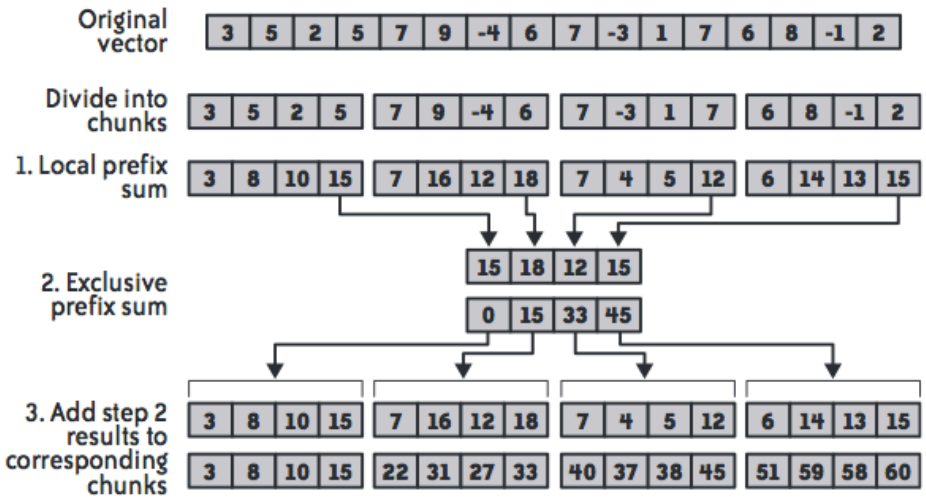
Prefix Scan (Pthreads)

Introdução

Com a realização deste trabalho pretende-se elaborar uma versão paralela do algoritmo que soma um array, PrefixScan que seja eficiente recorrendo a OMP. As dificuldades que podem advir serão relacionadas com o trabalho ser partilhado entre as threads e a main e assim terão de ser controladas com semáforos.

Problema

O problema trata-se de obter a soma de um array sendo o mais amigável para a cache possível para isso a soma é feita no próprio array. Inicialmente divide-se o array em tantos pedaços quanto o numero de threads. De seguinda é somando de forma sequencial cada um desses blocos. Sendo que assim na ultima posição do bloco está a soma total desse bloco. Depois é feito um array com a soma com shift à direita das somas dos blocos e enviada a posição correspondente desse array de somas novamente para os blocos iniciais e estes com esse novo valor somam os locais e assim no fim dessa iteração tem-se na ultima posição do array a soma total.



Versão paralela (Pthreads)

De seguida é apresentada a versão paralela do algoritmo de somar um array recorrendo a pthreads.

```
int N=NUM;  // number of elements in array X
int X[NUM];//={3,5,2,5,7,9,-4,6,7,-3,1,7,6,8,-1,2};
int inTotals[NUM_THREADS];  // global storage for partial results
int outTotals[NUM_THREADS];
sem_t semaforo[NUM_THREADS], semaforo2[NUM_THREADS];
```

Aqui são inicializadas as variáveis globais que servirão para a resolução do problema. O N é o numero de elementos do array, o X é o array dos elementos, o inTotals e o outTotals são para as somas parciais e os dois arrais de semáforos são usados para controlar o fluxo entre a main e as threads.

```
void *Summation (void *pArg)
{
 int tNum = *((int *) pArg);
 int size = N/NUM_THREADS;
 int start=tNum*size, end=(tNum+1)*size;
 int j=0;


 for(j=start+1;j<end;j++){
  X[j]+=X[j-1];
 }

 inTotals[tNum]=X[end-1];

 sem_post(&semaforo2[tNum]);
 sem_wait(&semaforo[tNum]);

 for (j=start; j<end; j++){
  X[j]+=outTotals[tNum];
 }

 return 0;
}
```

Neste algoritmo são calculadas as somas de cada bocado do bloco. Depois para no semáforo para que a main saiba que pode prosseguir quando estiverem de todas as threads e depois de a main liberta os semáforos pode prosseguir com as somas globais.

```
InitializeArray(X,&N);  // get values into A array; not shown
 for(j=0;j<NUM_THREADS;j++){
  sem_init(&semaforo[j], 0, 0);
```

```
  sem_init(&semaforo2[j], 0, 0);
}

for (j = 0; j < NUM_THREADS; j++) {
  int *threadNum = (int*)malloc(sizeof (int));
  *threadNum = j;
  pthread_create(&tHandles[j], NULL, Summation, (void *)threadNum);
}

for(j=0;j<NUM_THREADS;j++){
  sem_wait(&semaforo2[j]);
}

outTotals[0]=0;
for(j=1; j<NUM_THREADS;j++){
  outTotals[j]=inTotals[j-1]+outTotals[j-1];
}

for (j=0;j<NUM_THREADS;j++){ //threads podem voltar a trabalhar
  sem_post(&semaforo[j]);
}

for (j = 0; j < NUM_THREADS; j++) {
  pthread_join(tHandles[j], NULL);
}

printf("The sum of array elements is %d\n", X[N-1]);
for(j=0;j<NUM_THREADS;j++){
  sem_destroy(&semaforo[j]);
  sem_destroy(&semaforo2[j]);
}
```

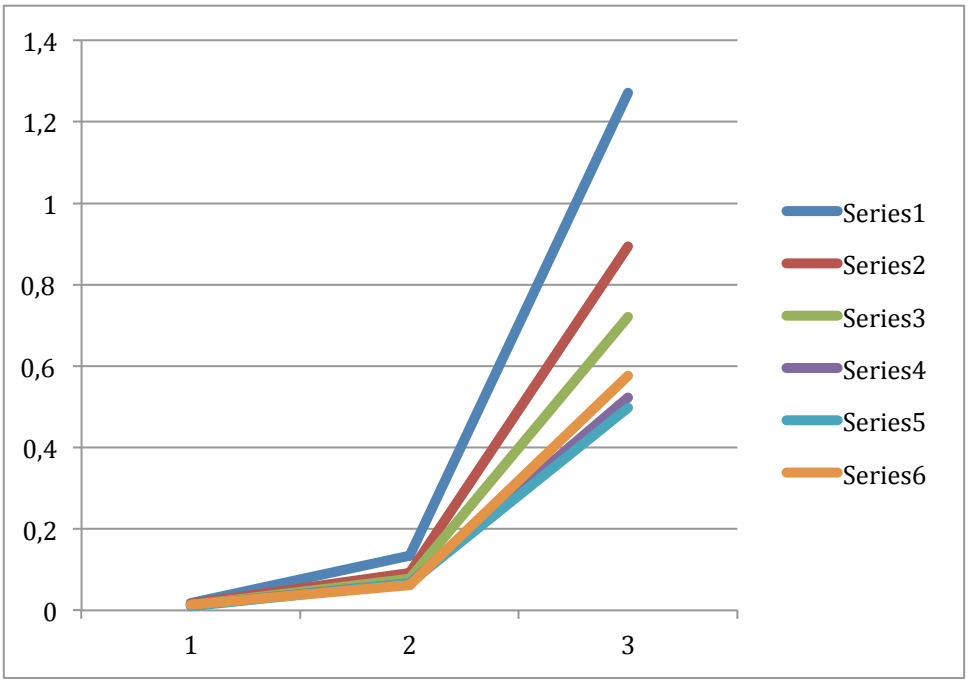
A main inicialmente bloqueia todos os semáforos e depois distribui o trabalho pelas diferentes threads. De seguida espera que estas libertem os seus semáforos e quando estas o fizerem ela faz o calculo do inTotals para o outTotals e indica que elas podem prosseguir libertando os seus semáforos. No fim espera pela conclusão das threads para apresentar o resultado.

Resultados

De seguida é possível ver uma tabela com os resultados obtidos.

size/nPthreads	1	2	4	8	12	16
1000000	0,018	0,016	0,013	0,013	0,01	0,014
10000000	0,134	0,092	0,078	0,067	0,067	0,062
100000000	1,272	0,894	0,721	0,522	0,498	0,577

Como é possível observar em dois dos três tamanhos o melhor é com 12 pthreads apenas num é com 16 pthreads. Os inputs são grandes no entanto os tempos são pequenos pois a carga de trabalho encontra-se igualmente distribuída.



No gráfico é possível observar que a série 5 (correspondente a 12 threads é quase sempre o melhor (a série 1 corresponde a 1 thread, a série 2 a 2 threads, a 3 a 4 threads e assim respectivamente).

Conclusão

Foi um trabalho diferente dos que tinham sido apresentados até agora porque o algoritmo teve uma diferente interpretação. É uma forma diferente de pensar, pensar em não "gastar" mais espaço do que o que temos à partida. O facto de não termos arrays auxiliares para fazer as somas faz com que a nossa forma de pensar tenha de mudar um pouco e foi o que aconteceu. Foi útil na medida em que nos mostrou uma diferente forma de fazer as coisas.

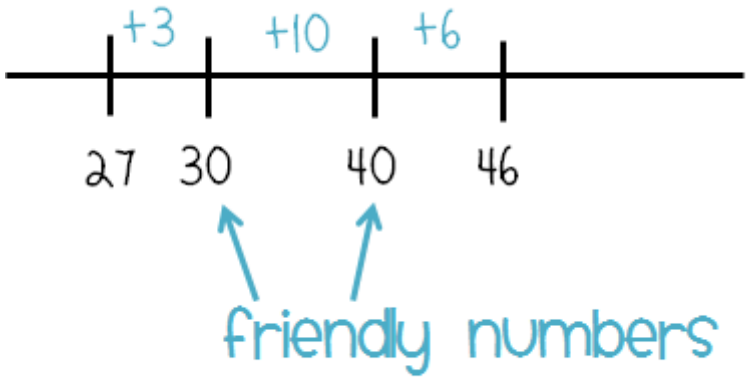
Friendly Numbers (MPI)

Introdução

Com a realização deste trabalho pretende-se elaborar uma versão paralela do algoritmo que encontra todos os números divisíveis entre si até um determinado valor recorrendo a MPI.

Problema

Este problema está assente em que números amigáveis são dois ou mais números naturais com uma divisão comum, o rácio entre a soma dos divisores de um número e o número em si. Dois números com a mesma divisão formam um par amigável.



Versão paralela (MPI)

É apresentada agora a versão paralela do algoritmo de FriendlyNumbers recorrendo a MPI e mantendo OMP (única diferença para ser apenas com MPI seria comentar as linhas referentes a OMP).

```
comm_sz=-1;
myrank=-1;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

int space = last/comm_sz;
FriendlyNumbers(myrank*space+start,(myrank+1)*space);
```

Aqui são criadas as variáveis que correspondem à criação de um ambiente MPI. Depois é calculado o espaço de trabalho de cada processo. E é executado o algoritmo de FriendlyNumbers (que se trata do mesmo algoritmo da versão sequencial).

```
if(myrank==0){ //0 den; 1 num
    for(i=1;i<comm_sz;i++){
      MPI_Recv(&den[i*space], space, MPI_INT, i, 0,
      MPI_COMM_WORLD, NULL);
      MPI_Recv(&num[i*space], space, MPI_INT, i, 1,
      MPI_COMM_WORLD, NULL);
#pragma omp for schedule (static, 8)
    for (i = 0; i < last; i++) {
      for (j = i+1; j< last; j++) {
        if ((num[i] == num[j]) && (den[i] == den[j]))
          printf ("%d and %d are FRIENDLY \n", start+i, start+j);
      }
    }
```

De seguida se for o processo 0 recebe dos outros processos os espaços deles e começa a imprimir no ecran os resultados.

```
}else{
    MPI_Send(&den[0], space, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Send(&num[0], space, MPI_INT, 0, 1, MPI_COMM_WORLD);
}

MPI_Finalize();
```

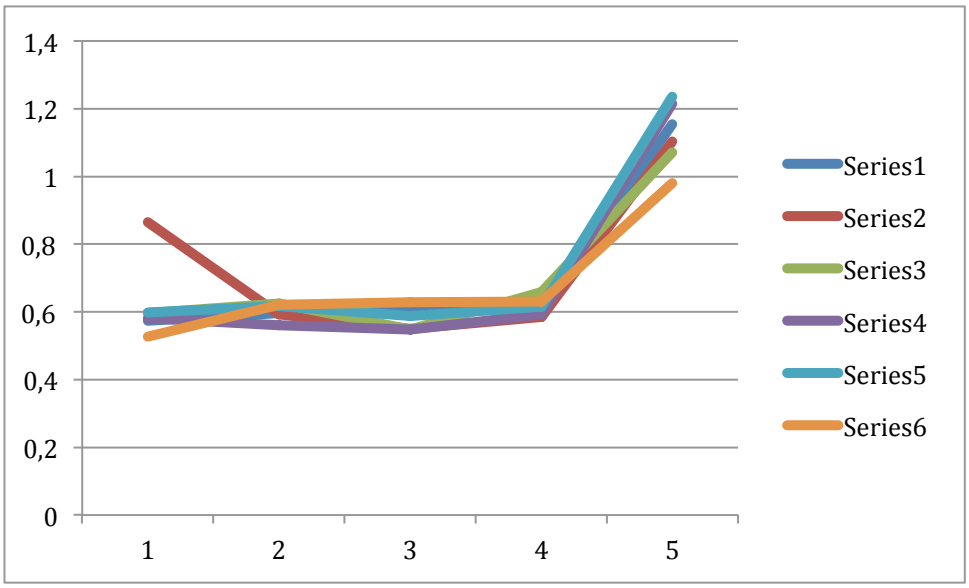
Se forem os outros processos apenas têm de enviar os seus espaços, denominadores e numeradores.

Resultados

De seguida são apresentados os resultados para a execução do algoritmo de Friendly Numbers recorrendo a MPI.

length/nProcs	1	2	4	8	12	16
200	0,574	0,865	0,596	0,581	0,599	0,527
400	0,597	0,593	0,625	0,56	0,618	0,621
1000	0,605	0,55	0,548	0,548	0,588	0,628
5000	0,606	0,585	0,659	0,592	0,616	0,63
100000	1,154	1,103	1,072	1,215	1,235	0,981

Como é possível observar para maiores tamanhos a utilização de mais processos compensa, para tamanhos pequenos o custo da criação de processos não compensa no peso final.



Como é possível observar pelo gráfico para o nível 5 (input de 100000) a série6 é a melhor (corresponde a 16 processos). Os níveis são o tamanho de input por ordem sequencial e as séries o numero de processos também sequenciais.

Análise com MPIP

Recorrendo ao MPIP para obter considerações em relação à distribuição da carga e da performance durante a execução foi possível obter os seguintes resultados.

Task	AppTime	MPITime	MPI%
0	0.000299	0.000267	89.30
1	0.000114	9.2e-05	80.70
2	0.000106	8.5e-05	80.19
3	0.000112	8.8e-05	78.57
4	0.000107	8.4e-05	78.50
5	0.000115	8.7e-05	75.65
6	0.000128	9.9e-05	77.34
7	0.000114	8.5e-05	74.56
8	0.000111	8.5e-05	76.58
9	0.000144	0.000107	74.31
*	0.00135	0.00108	79.93

É possível ver pela tabela acima que houve uma boa distribuição ao nível de MPI visto que todos os processos demoram aproximadamente o mesmo tempo e têm a mesma percentagem em MPI.

Call	Site	Time	App%	MPI%	COV
Send	4	0.783	58.00	72.57	0.09
Recv	2	0.244	18.07	22.61	0.00
Send	3	0.029	2.15	2.69	0.34
Recv	1	0.023	1.70	2.13	0.00

É possível ver pela tabela acima que 58% do tempo de agregação da aplicação está a ser nos envios e 18 nas recepções.

Conclusão

Mais uma vez é possível constatar que a utilização do MPI só compensa quando se passa a usar inputs grandes pois a criação de processos e a comunicação entre eles têm sempre um custo bastante alto associado. Para minimizar esses custos é necessário que o input requeira pela parte desses processos no mínimo o mesmo tempo de trabalho que o tempo que demoraria em sequencial.

Friendly Numbers (Map Reduce)

Introdução

Com a realização deste trabalho pretende-se elaborar uma versão paralela do algoritmo que encontra todos os números divisíveis entre si até um determinado valor recorrendo a uma biblioteca Map-Reduce.

Versão paralela (MR)

É apresentada agora a versão paralela do algoritmo de FriendlyNumbers recorrendo a MPI mas utilizando uma biblioteca externa de Map Reduce.

```
using namespace MAPREDUCE_NS;
int size, pid, nprocs, start, end;
```

Aqui é declarado o namespace do MapReduce e são declaradas as variáveis globais.

```
void FriendlyNumbers(int itask, KeyValue *kv, void* ptr)
{
   int end=start+(offset*(pid+1)-1);
   int start+=(offset*(pid));

   int i, j, factor, ii, sum, done, n;

   for (i = start; i <= end; i++) {
      ii = i - start;
      sum = 1 + i;
      done = i;
      factor = 2;
      while (factor < done) {
         if ((i % factor) == 0) {
            sum += (factor + (i/factor));
            if ((done = i/factor) == factor) sum -= factor;
         }
         factor++;
      }
      num = sum; den = i;
      n = gcd(num, den);
      num /= n;
      den /= n;
      MapEntry me;
      me.key = num;
      me.value = den;
      int toSave = i;
      kv->add((char*)&me,sizeof(MapEntry),(char*)&toSave,sizeof(int))
   }
}
```

Aqui é possível observar a função que fará o papel de Map, fazer a sua parte das contas.

```
void results(char *key, int keybytes, char *multivalue, int nvalues, int *valuebytes,
KeyValue *kv, void *ptr){
   int i=0;
   if(nvalues){
```

```
      for (; i < nvalues; i++) {
         printf("%d ",multivalue[i]);
      }
      printf("\n");
   }
}
```

Aqui a função que tem o papel de Reduce, converte os valores no multivalue em output.

```
int main(int argc, char **argv)
{
   start = atoi(argv[1]);
   end = atoi(argv[2]);

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&pid);
   double time=MPI_Wtime();

   size = end-start+1;
   int block = size / nprocs;

   MapReduce *mr = new MapReduce(MPI_COMM_WORLD);
   mr->verbosity=1;
   mr->timer=1;
   MPI_Barrier(MPI_COMM_WORLD);

   int nTasks = mr->map(nprocs,&FriendlyNumbers);
   mr->collate(NULL);
   mr->reduce(results,NULL);
   MPI_Barrier(MPI_COMM_WORLD);

   if(pid==0)
      printf("Time:%lf seconds\n",time=MPI_Wtime()-time);

   delete mr;
   MPI_Finalize();
}
```

Aqui é possível ver a inicialização do ambiente e da distribuição do trabalho pelos processos usando a primitiva mr->map. Depois usa-se o mr->reduce para obter os valores.

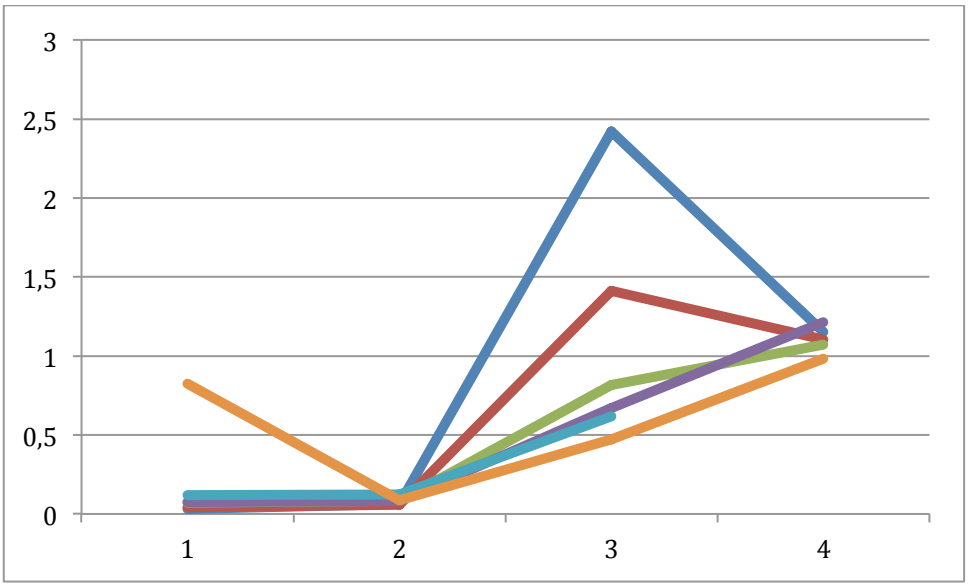
Resultados

De seguida são apresentados os resultados para a execução do algoritmo de Friendly Numbers recorrendo à biblioteca MapReduce.

length/nProcs	2	4	8	10	12	16
1000	0,0316	0,0406	0,0718	0,0752	0,1178	0,824
10000	0,0628	0,0588	0,0814	0,0812	0,1252	0,0867
100000	2,423	1,4125	0,8153	0,6724	0,6181	0,4713

100000	1,154	1,103	1,072	1,215		0,981

Como é possível observar para maiores tamanhos a utilização de mais processos compensa, para tamanhos pequenos o custo da criação de processos não compensa no peso final tal como é possível ver na versão do MPI. A ultima linha corresponde à execução do MPI para ser mais facilmente comparável.



Constata-se agora no gráfico que na maior parte das vezes a mais rápida é a linha alaranjada correspondente a 16 processos. A exceção verifica-se para o 1 que corresponde ao imput de 1000, o mais pequeno.

Análise com MPIP

Recorrendo ao MPIP para obter considerações em relação à distribuição da carga e da performance durante a execução foi possível obter os seguintes resultados.

```
Task    AppTime    MPITime     MPI%
   0       0.587      0.551    93.87
   1      0.0951   0.000218     0.23
   2       0.149   0.000192     0.13
   3         0.2   0.000192     0.10
   4       0.228   0.000188     0.08
   5       0.221     0.0775    35.03
   6       0.226     0.0592    26.21
   7       0.396   0.000187     0.05
   8       0.396      0.137    34.59
   9       0.489   0.000198     0.04
  10       0.491      0.153    31.27
  11       0.489       0.21    43.02
  12       0.524   0.000198     0.04
  13       0.587   0.000188     0.03
  14       0.589      0.244    41.38
  15       0.589      0.102    17.33
   *        6.25       1.54    24.56
---------------------------------------
```

É possível ver pela tabela acima que houve uma boa distribuição menos uniforme ao nível de MPI do que na versão sem o MapReduce e o tempo que demorou também.

```
Call            Site       Time    App%    MPI%     COV
Send               3        984   15.74   64.09    1.30
Recv               2        551    8.81   35.86    0.00
Recv               1      0.451    0.01    0.03    0.00
Send               4      0.364    0.01    0.02    0.11
```

É possível ver pela tabela acima que houve grandes melhorias ao nível de tempo perdido na comunicação em relação à versão sem MapReduce.

Conclusão

Este trabalho foi o mais “fora do comum” deste semestre pois pôs-me em frente a um biblioteca map-reduce que eu nunca tinha usado. Foi bom na medida em que nos obriga a utilizar software desenvolvido por outros pois será o que faremos no mercado de trabalho e por outro lado faz-nos também utilizar outra forma de abordar os problemas que até agora se baseavam no conceito de threads e processos.

Conclusão global

Foi uma Unidade Curricular muito prática o que contribuiu muito para a nossa aprendizagem ao longo do semestre. Foi ótimo poder aprofundar conhecimentos a experimentar e ver as coisas correrem mal. Porque houveram alturas que não correram tão bem mas com dedicação e tempo foi possível corrigir esses problemas e chegar ao pretendido.

O facto de ser bastante prática faz com tenhamos tempo para poder errar e aprender com os nossos erros de forma assistida pois se houvessem problemas maiores tínhamos a quem recorrer.