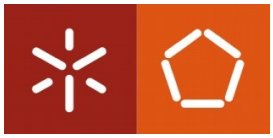
Odd-Even Sort paralelo

Algoritmos Paralelos

Computação Paralela Distribuída

Mestrado em Engenharia Informática

Universidade do Minho



Duarte Nuno Ferreira Duarte

pg27715

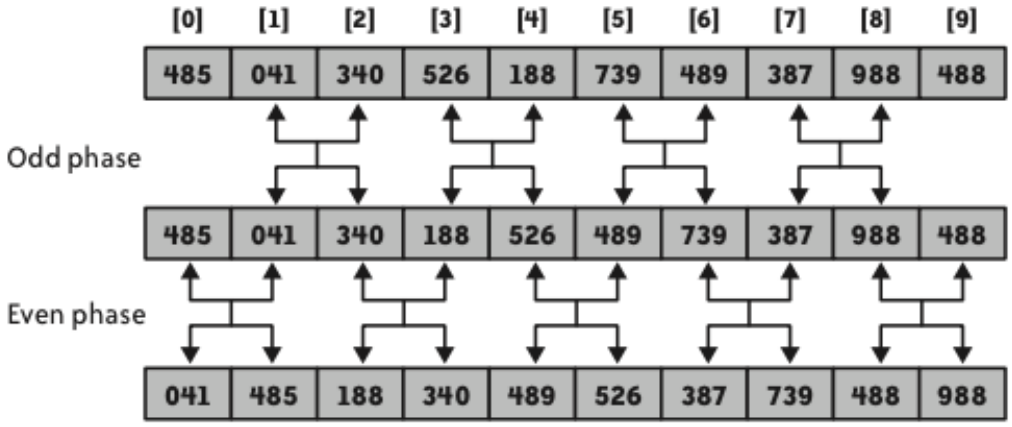
Índice

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

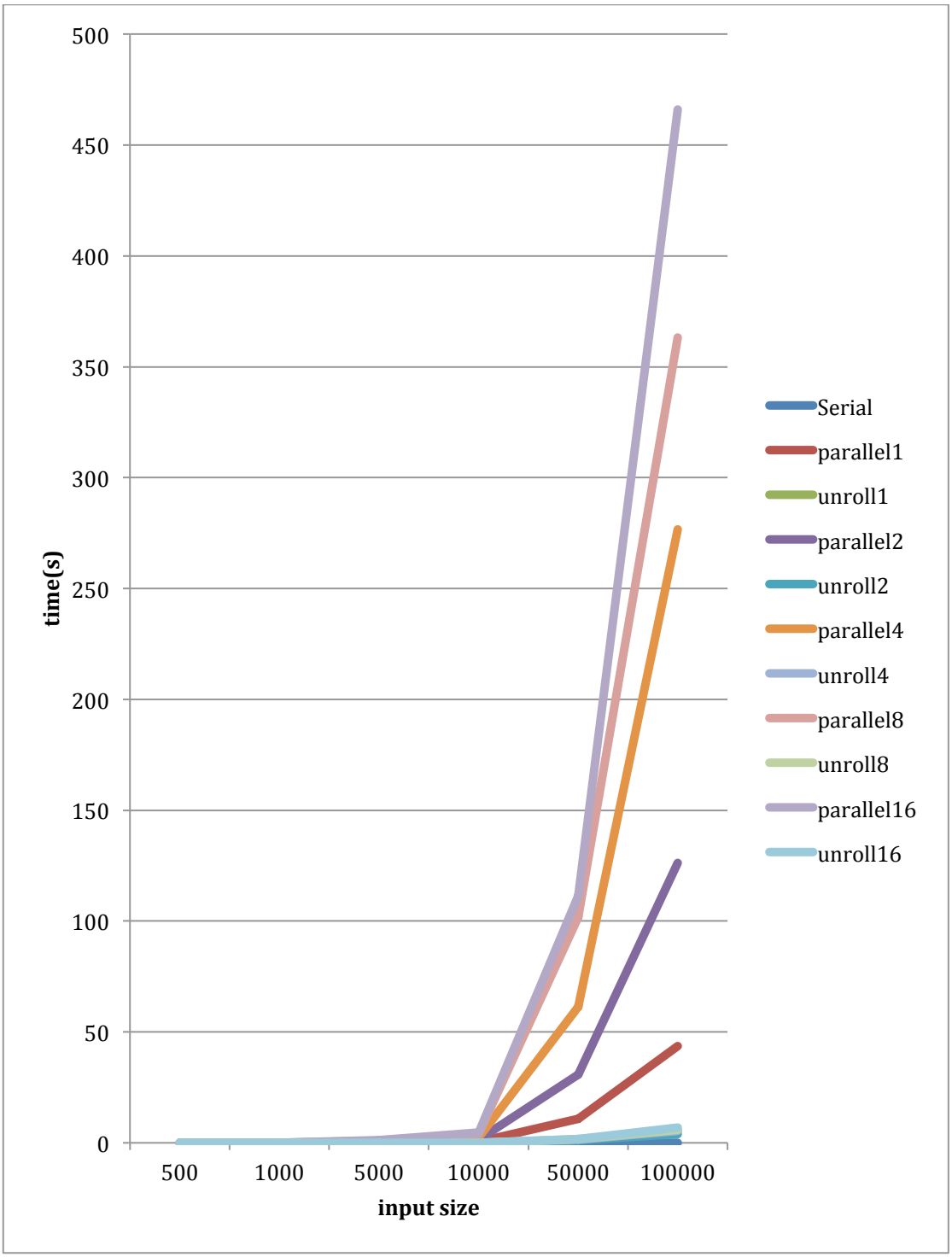


Tabela de resultados

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