PRAM

Algoritmos Paralelos - MEI

Hélder José Alves Gonçalves – PG28505

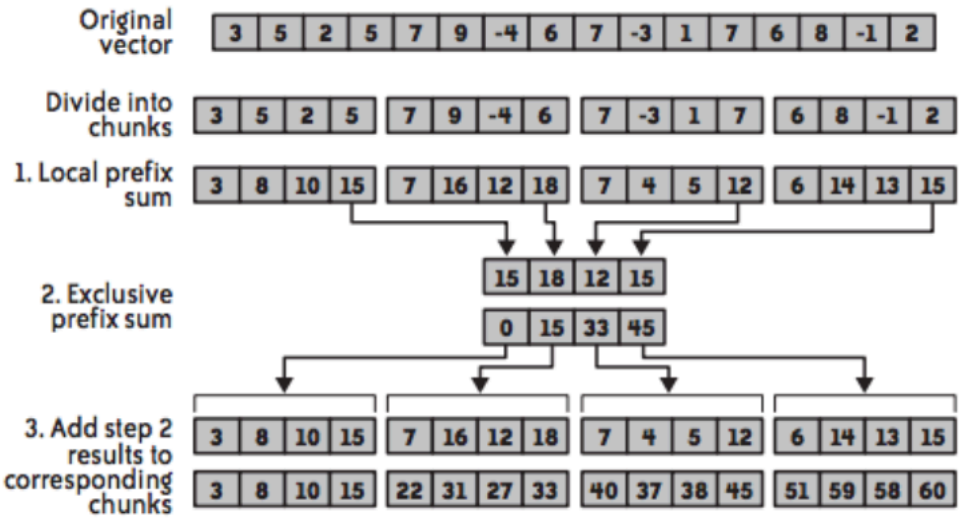
Introdução

O uso de *pthreads* normalmente tem uma vantagem acrescida em relação ao *OpenMP*. Deve-se devido ao mais baixo nível de programação exigida no uso de *pthreads*, depois a outra vantagem é que é evitada uma camada de abstração existente no *OpenMP*, pois este uso *pthreads* na sua camada inferior.

Para isso neste relatório é efectuada uma pequena comparação da mesma aplicação, em que uma usa exclusivamente o *pthreads* e o outro usa só *OpenMP.*

Problema

O problema consiste em a partir de um array fornecido, obtermos o somatório inclusivo do mesmo. Para isso recorremos ao seguinte esquema de paralelização estudado na aula.



Implementações

Foram implementadas duas versões deste algoritmo. Uma primeira com o código bastante simples, escrita em *openMP* e depois uma que precisou de um pouco mais cuidado principalmente na implementação de uma barreira.

OpenMP

Esta versão é bastante simples, resumindo-se à inicialização do array 'X' e à chamada da função responsável pela função principal que será vista em mais detalhe de seguida.

```
78  int main(int argc, char* argv[])
79  {
80      int sum = 0;
81      double time = omp_get_wtime();
82
83      InitializeArray(X,&N);  // get values into A array; not shown
84      Summation();
85
86      //Get Time
87      time = omp_get_wtime() - time;
88
89      printf("\nThe sum of array elements is %d\nTIME:%f\n", sum,time);
90      return 0;
91  }
```

As duas versões tem recurso da função 'omp_get_wtime' para medir os tempos de execução da aplicação, para efeitos de comparação das versões.

```
24  void *Summation ()
25  {
26      int aux=0, sum=0;
27      int i;
28
29      #pragma omp parallel
30      {
31          int tNum = omp_get_thread_num();
32
33          #pragma omp single
34          {
35              NUM_THREADS = omp_get_num_threads();
36              gSum = (int*)malloc(NUM_THREADS*sizeof(int));
37          }
38
39          int start, end;
40          start = (N / NUM_THREADS) * tNum;
41          end = (N / NUM_THREADS) * (tNum+1);
42          if (tNum == (NUM_THREADS-1)) end = N;
43
44          for (i = start+1; i < end; i++)
45              X[i] = X[i-1]+X[i];
46
47          gSum[tNum] = X[end-1];
48
49          /******* Barrier ********/
50          #pragma omp barrier
51
52          #pragma omp single
53          {
54              //Single
55              for(i=0; i<NUM_THREADS; i++){
56                  sum+=aux;
57                  aux=gSum[i];
58                  gSum[i]=sum;
59              }
60          }
61
62          for (i = start; i < end; i++)
63              X[i] += gSum[tNum];
64      }
65      return NULL;
66  }
```

Neste código começamos por obter o *id* da *thread* para esta ser capaz de calcular a sua zona de ação. Depois é importante que apenas uma delas obtenha o número total de *threads* criadas e a seguir aloque a memória necessária para a para a execução central do algoritmo.

São inicializadas as variáveis privadas *start* e *end* que correspondem à zona de cálculo. O for seguinte corresponde ao calculo do somatório inclusivo de

cada uma das afetadas por cada uma, e depois e depositado em um *array* auxiliar o valor do último índice de cada uma das zonas afectadas. Agora é preciso calcular o somatório exclusivo com base no *array* auxiliar calculado, mas para isso é preciso garantir que o *array* anterior já esteja calculado, para isso é usado uma barreira.

Cada índice do *array* auxiliar corresponde a um *thread*, e agora basta somar o seu valor à zona afectada por cada *thread*. É de notar que esta zona é ótima para vetorizar apesar de esta funcionalidade não ser explorada.

Pthreads

Esta versão é um pouco mais complexa mas a base do algoritmo mantem-se.

```
79  int main(int argc, char* argv[])
80  {
81      int j, sum = 0;
82      double time = omp_get_wtime();
83
84      pthread_t tHandles[NUM_THREADS];
85      sem_init(&count_sem, 0, 1);
86      sem_init(&barrier_sem, 0, 0);
87
88      InitializeArray(X,&N);  // get values into A array; not shown
89      for (j = 0; j < NUM_THREADS; j++) {
90          int *threadNum = (int*)malloc(sizeof (int));
91          *threadNum = j;
92          pthread_create(&tHandles[j], NULL, Summation, (void *)threadNum);
93      }
94
95      for (j = 0; j < NUM_THREADS; j++)
96          pthread_join(tHandles[j], NULL);
97
98      //Get Time
99      time = omp_get_wtime() - time;
100
101     printf("\nThe sum of array elements is %d\nTIME:%f\n", sum,time);
102     return 0;
103 }
```

Na *main* temos agora de preparar o lançamento individual de cada *thread*, algo que não era preciso até agora. Além disso é preciso inicializar os semáforos que são necessários para a barreira implementada na função principal.

Assim que se tem o *array* inicializado basta arrancar com as *threads* com a função pretendida. Para finalizar fazemos um *join* para esperar que cada uma das *threads* acabe a sua execução.

Ao contrário da versão anterior todas as variáveis locais são privadas, mas a grande diferença está na barreira implementada, todo o resto é igual à versão anterior.

Esta versão evita o famoso busy-wait, evitando assim o desperdício de ciclos de relógio. A seu segredo está nos seus semáforos (barrier_sem e count_sem). Primeiro usamos o count_sem que é inicializado a 1 para permitir a passagem de uma thread de cada vez que vai incrementando o contador de threads que passou até ao momento. Após incrementar o contador é libertado este lock para permitir a entrada de novas threads na zona critica e ficam à espera que a thread mais lenta chegue. Esta zona é necessária para conseguir identificar quando é que a última thread é atingida.

```
27 void *Summation (void *pArg)
28 {
29     int tNum = *((int *) pArg);
30     int start, end, i;
31     int aux=0, sum=0;
32
33     start = (N / NUM_THREADS) * tNum;
34     end = (N / NUM_THREADS) * (tNum+1);
35     if (tNum == (NUM_THREADS-1)) end = N;
36
37     for (i = start+1; i < end; i++)
38         X[i] = X[i-1]+X[i];
39
40     gSum[tNum] = X[end-1];
41
42     /******* Barrier ********/
43     sem_wait(&count_sem);
44     if(counter == NUM_THREADS-1){
45         counter=0;
46         sem_post(&count_sem);
47
48         //Single
49         for(i=0; i<NUM_THREADS; i++){
50             sum+=aux;
51             aux=gSum[i];
52             gSum[i]=sum;
53         }
54
55         for(i=0; i<NUM_THREADS; i++)
56             sem_post(&barrier_sem);
57     }else{
58         counter++;
59         sem_post(&count_sem);
60         sem_wait(&barrier_sem);
61     }
62
63     for (i = start; i < end; i++)
64         X[i] += gSum[tNum];
65
66     delete (int *)pArg;
67     return NULL;
68 }
```

Assim que chega a ultima *thread*, esta coloca o contador a zero e calcula o somatório exclusivo assim como na versão anterior. Assim que esta zona de processamento está concluída, a *thread* lança tantos *sem_post* quantos necessários para libertar as *threads* que estão à espera na barreira.

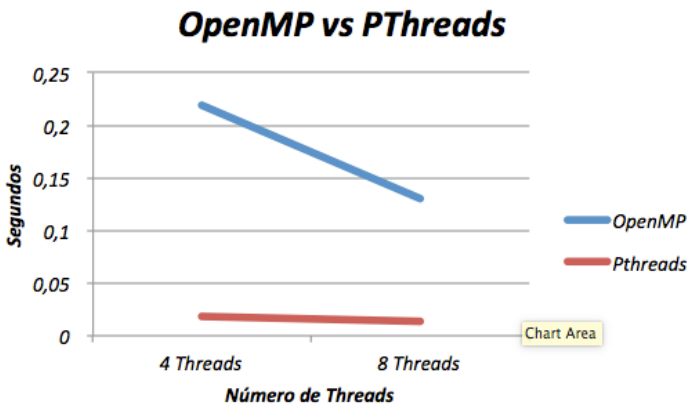
Agora o processamento é todo igual à versão anterior.

Comparação de resultados

Aqui comparamos os resultados de ambas as versões. São efectuados 4 testes e daí é retirada a mediana. São feitos dois tipos de testes, para quatro e oito *threads*, mas principal questão em estudo são as diferenças entre as duas ferramentas usadas.

Pela análise do gráfico seguinte podemos reparar que a versão com *pthreads* obteve um resultado cerca de dez vezes mais rápido que a versão *OpenMP*, o que nos leva a ver que existem grandes vantagens neste tipo de programação apesar de ser parecer um pouco mais minuciosa de implementar.

Já quando comparamos a execução de quatro contra a de oito threads repara-se que em ambas as versões houveram ganhos. Estes ganhos notam-se mais na



versão *OpenMP* mas não chegam para rivalizar a sua adversária, apesar de ter ganhos inferiores tem um tempo de execução bastante melhor.

Conclusão

Trabalho interessante para explorar novas ferramentas, desta vez com maior incidência em semáforos, ao contrário do exercício anterior que esteve mais focado em *mutex's*.

Ficou mais que claro as vantagens em usar Pthreads em relação à sua rival.