**PRAM**

Algoritmos Paralelos - MEI

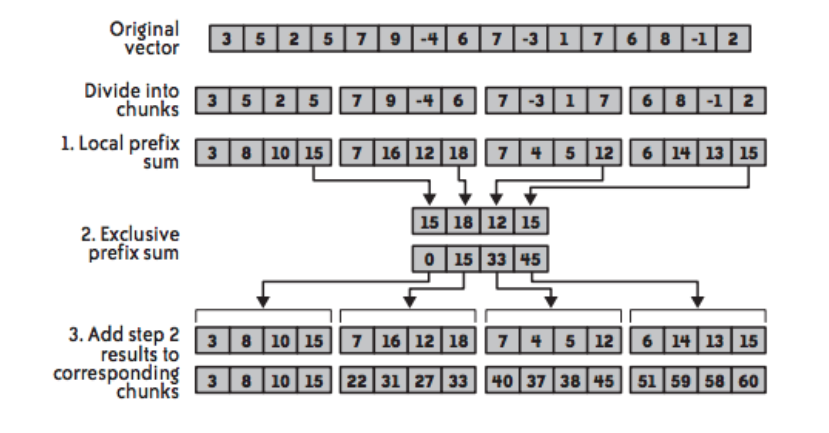
Hélder José Alves Gonçalves – PG28505

**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**

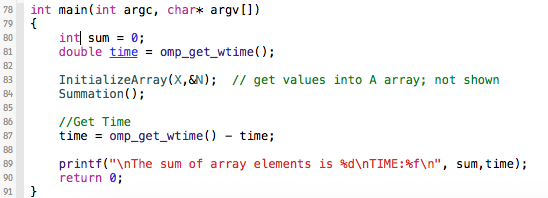
O problema consiste em a partir de um array fornecido, obtermos o somatório inclusivo do mesmo. Para isso recorremos ao seguinte esquema de paralelização estudado na aula.

**Implementações**

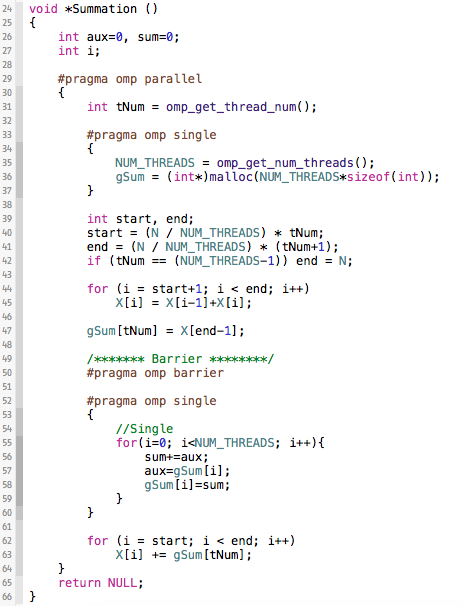
Foram implementadas duas versões deste algoritmo. Uma primeira com o código bastante simples, escrita em *openMP* e depois uma que precisou de um pouco mais cuidado principalmente na implementação de uma barreira.

**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.



As duas versões tem recurso da função ‘omp\_get\_wtime’ para medir os tempos de execução da aplicação, para efeitos de comparação das versões.



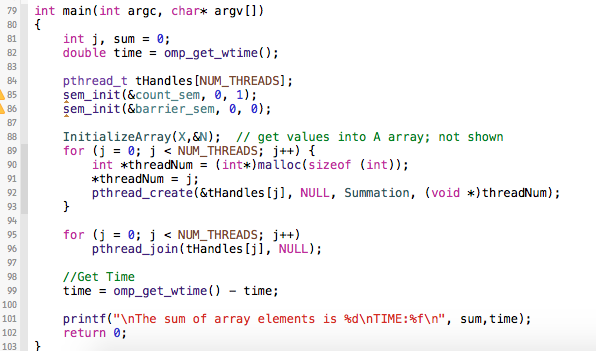
Neste código começamos por obter o *id* da *thread* para esta ser capaz de calcular a sua zona de ação. Depois é importante que apenas uma delas obtenha o número total de *threads* criadas e a seguir aloque a memória necessária para a para a execução central do algoritmo.

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.

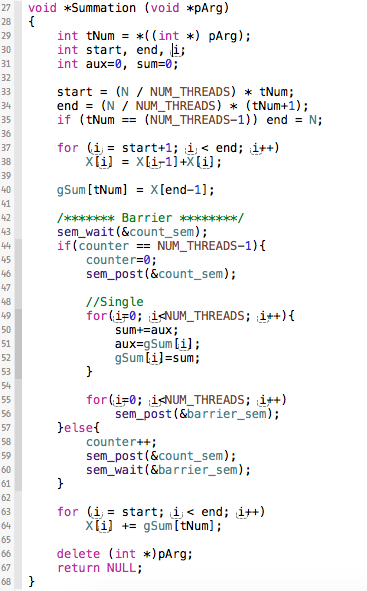


Na *main* temos agora de preparar o lançamento individual de cada *thread*, algo que não era preciso até agora. Além disso é preciso inicializar os semáforos que são necessários para a barreira implementada na função principal.

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.



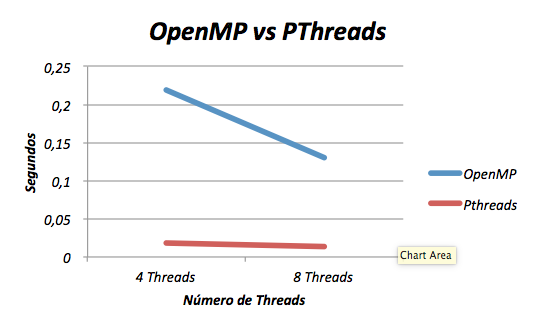
Assim que chega a ultima *thread*, esta coloca o contador a zero e calcula o somatório exclusivo assim como na versão anterior. Assim que esta zona de processamento está concluída, a *thread* lança tantos *sem\_post* quantos necessários para libertar as *threads* que estão à espera na barreira.

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.