Paralelização do ‘BubbleSort’

CPD – Algoritmos Paralelos

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

Introdução

Como caso de estudo deste trabalho temos o *BubbleSort*. Este algoritmo é um algoritmo de ordenação bastante simples. A ideia é percorrer o *array* diversas vezes, e em cada passagem faz flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas de um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

No melhor caso, o algoritmo executa n operações relevantes, onde n representa o número de elementos do vector. No pior caso, são feitas n^2 operações. A complexidade desse algoritmo é de ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

Problema

Paralelizar este algoritmo não parece ser muito fácil à primeira vista. Isto deve-se ao facto de cada uma das passagens efectuadas no *array* são sempre dependentes da passagem anterior, impossibilitando desta forma a paralelização que estivemos habituados a aplicar no semestre passado. Isto faz com que tenhamos de olhar para o problema de outra forma totalmente diferente do que vimos até agora.

Resolução

Locks, são a resposta para o nosso problema. Ao invés de dividir o trabalho dando a cada *thread* uma iteração do ciclo *for* existente no algoritmo original vamos dividir o *array* em blocos obrigando cada *thread* a percorrer uma determinada secção isoladamente. Assim que uma *thread* termine uma determinada secção esta vai bloquear a próxima secção e só de seguida libertar a sua secção actual do repectivo *lock*.

Uma melhoria substancial que pode ser feita no algoritmo é obrigar-lo a executar até que não ser efectuada nenhuma alteração numa passagem completa do *array*. Na maioria dos casos isto é uma excelente melhoria evitando iterações desnecessárias. No caso do *array* dado já estar ordenado são executadas apenas t passagens, em que t é o número total de *threads* existentes.

Na resolução apresentada são sempre assimiladas $t+1$ zonas de bloqueio, em que é criada '+1' devido ao resto da divisão do número total de elementos pelo número de *threads*. Aqui está um ponto que pode ser melhorado em implementações futuras, de modo a equilibrar todas as secções.

A gestão dos *locks* tem de ser feita com bastante cuidado para evitar situações indesejáveis, tais como *deadlocks* ou até mesmo a falha numa passagem completa. As zonas onde iram ocorrer estas trocas são calculadas antes de chegar à zona paralela, e como já foi dito o número de zonas dependerá de t. Como base, a primeira zona é bloqueada antes de entrar no ciclo for e a última zona é libertada a no final. Durante o ciclo, sempre que chegamos ao índice de troca de secção, tenta-se em primeiro lugar

bloquear a próxima zona e só depois é que é libertada a zona actual. É imperativo que esta troca de zona aconteça nesta ordem para garantir a sequencialidade entre as várias *threads*.

Sempre que é feita uma passagem completa é decrementado o tamanho total da lista que é necessária percorrer até ao momento, evitando computação desnecessária.

Testes

Os teste são efectuados no *compute-662-6* que tem as seguintes especificações:

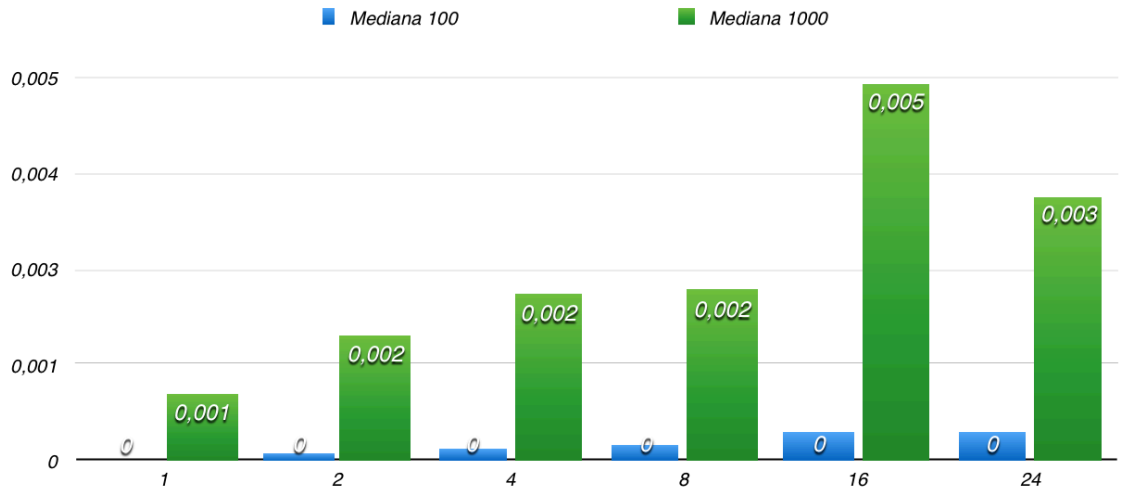
```
papi_avail | grep CPU
Model string and code    : Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz (62)
CPU Revision             : 4.000000
CPUID Info               : Family: 6  Model: 62  Stepping: 4
CPU Max Megahertz        : 2401
CPU Min Megahertz        : 1200
CPUs per Node            : 24
Total CPUs               : 48
```

Para visualizar o comportamento do mesmo para as mais diversas situações é foi feito um script que testa para as varias dimensões um diferente número de *threads*:

```
for size in 100 1000 10000 50000 100000 200000 500000
do
        echo "+++++++++++++++++ Dimensão $size ++++++++++++++++++"
        for nthreads in 1 2 4 8 16 24
        do
                echo "Num threads: $nthreads "
                export OMP_NUM_THREADS=$nthreads
                ./omp $size g
                ./omp $size g
                ./omp $size g
        done
done
```

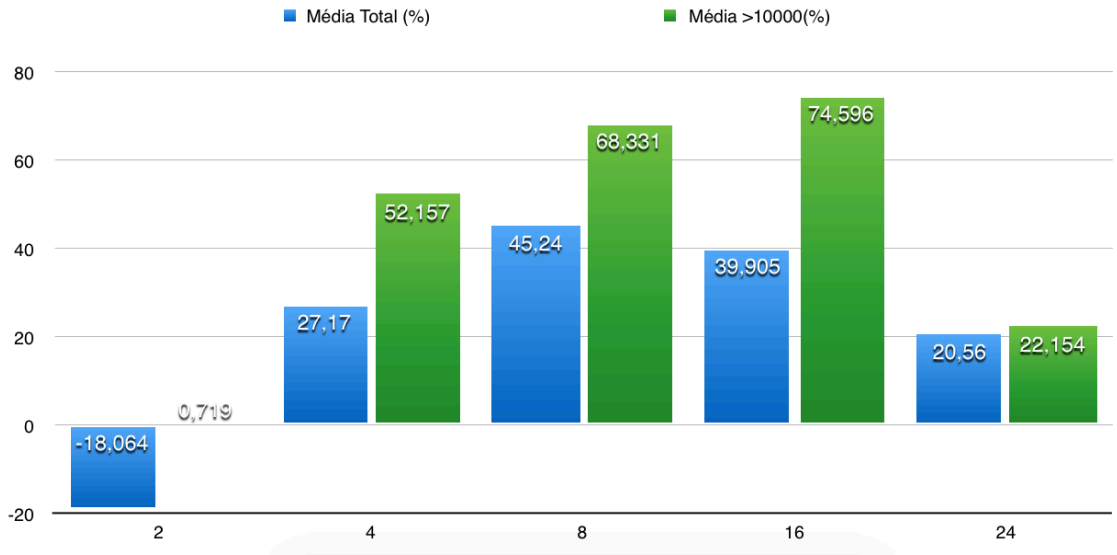
O *output* deste *script* pode ser consultado em anexo em '*resultado.txt*', assim como todos os dados tratados no anexo '*Graphs_01BubbleSort.pdf*'. Agora vamos ver uma pequena análise dos tempos obtidos.

Nos testes das primeiras dimensões observa-se que os tempos pioram à medida que é aumentado o número de threads para a resolução do problema. Este resultado era previsível devido à pequena dimensão do nosso problema. Mas houve uma pequena oscilação no teste da dimensão 1000 para 24 threads, em que o tempo diminuiu e não manteve/aumentou como era regra geral. Isto pode simplesmente dever-se ao facto do nodo não estar somente a ser utilizado para estes testes, mas também ao facto da dimensão ser tão pequena qua bastava uma pequena variação no nodo para termos valores inesperados.



Os restantes dimensões já justificam a paralelização e isso vê-se com a diminuição consecutiva dos tempos de execução à medida que se aumenta do número de *threads* para resolver o problema. Mas infelizmente, não se conseguem obter ganhos de performance lineares. Como se pode ver, na maioria dos casos o algoritmo com duas threads de execução obteve piores valores que o seu sequencial. A boa notícia é que se conseguem obter bons ganhos de desempenho para os restantes testes.

Importante: Consultar restantes gráficos do anexo referido.



Neste gráfico podemos ver a azul os ganhos médios da totalidade dos testes, em que vemos claramente que não compensa executar o programa o com duas threads (-18%). A verde está a média dos ganhos a partir dos testes que já considero que os ganhos sejam relativamente estáveis. Podemos notar que quanto dobramos o poder computacional disponível obtemos melhorias entre $52\% < x <= 74\%$, que apesar de não serem lineares são bastante boas.

Também tive ajuda do ompP para identificar as zonas que ocuparam mais tempo no programa. Por exemplo para o mesmo número de threads podemos ver que para a dimensão 1000, só 88% foi ocupado na zona paralelizada, enquanto que para o de dimensão 100000 tivemos 99% do tempo para a zona paralelizada, entre outras questões, tais como muitos *locks* não estarem balanceados entre eles, possivelmente porque estiveram mais tempo à espera que a próxima zona estive-se disponível. Os relatórios obtidos vão em anexo.

Dimensão: 1000

(%)	Exclusive	(%)						
100.0	0.000466	8.083523	0	[unknown: 8 threads]				
3.816415	0.000220	3.816415	1	R00001	PARALLEL	main.c	41	42
88.100062	0.005080	88.100062	1	R00002	PARALLEL	main.c	180	208

Dimensão: 100000

(%)	Exclusive	(%)						
100.0	0.002766	0.045169	0	[unknown: 8 threads]				
0.003773	0.000231	0.003773	1	R00001	PARALLEL	main.c	41	42
99.951058	6.119897	99.951058	1	R00002	PARALLEL	main.c	180	208