CP - Trabalho Prático 2

Duarte Serrão
UC de Computação Paralela
Mestrado Integrado em Engenharia Informática
Universidade do Minho
a83630@alunos.uminho.pt

Vasco Oliveira
UC de Computação Paralela
Mestrado em Engenharia Informática
Universidade do Minho
pg50794@alunos.uminho.pt

***Abstract*—Este relatório demonstra o processo de paralelismo de um algoritmo por meio de *threads* e cores.**

***Index Terms*—otimização, paralelismo, cores, threads**

I. Introdução

O presente relatório, do segundo trabalho prático da unidade curricular de Computação Paralela, do curso de Mestrado (Integrado) em Engenharia Informática da Universidade do Minho, visa demonstrar a metodologia aplicada aquando a otimização de um dado algoritmo, dando máximo uso às capacidades de paralelismo a nível de *threads*.

II. Método de Otimização

A partir do primeiro trabalho prático, chegou-se à conclusão que a porção mais pesada do programa seria o loop que está aninhado dentro do loop que itera sobre todos os pontos no k_means. Ou seja, o loop que verifica qual o cluster mais próximo de cada ponto.

Como não existem muitos clusters e como essa porção já se encontra vetotrizada, o grupo chegou à conclusão que compensa mais paralelizar o loop que itera sobre os pontos. Como nenhum ponto depende de outro, foi apenas preciso adicionar a seguinta porção de código:

```
#pragma omp parallel num_threads(nThreads)
{
  #pragma omp for
     reduction (+:sum_dist_x[:K])
     reduction (+:sum_dist_y[:K])
     reduction (+:count[:K])
  for(int i = 0; i < N; i++)
  {
    ...
  }
}
```

Como é possível reparar, também se adicionou três reductions diferentes, pois a soma das distâncias e o contador de pontos são arrays partilhados pelas diferentes threads. Teve de se fazer uma reduction para cada um e assim, cada thread terá a sua cópia, somado tudo no final.

Após observar os resultados finais obtidos, o grupo concluiu que não seria necessário alterar mais o código.

III. Resultados

Como métrica de comparação, o grupo utilizou o tempo de execução. Para tentar minimizar fatores externos, compilou-se o programa uma única vez para se anotar os resultados. Em seguimento do primeiro trabalho prático, o grupo continuou a utilizar as mesmas flags, sendo estas `-O2 -fopenmp -ftree-vectorize -msse4 -mavx -funroll-loops`. É de notar que a flag `-fopenmp` é nova e serve para se usar os pragmas da biblioteca openMP.

Para se anotar os resultados, escolheu-se medir o tempo para 1, 2, 8, 20 e 40 threads. À medida que se alterava o número de threads, também se alterava o número de cores, de modo a ficarem iguais, ou seja, todas as threads poderiam correr em paralelo.

A partir da tabela I é possível observar que o tempo de execução diminuiu sempre que se adiciona mais paralelismo, tendo uma tendência para estabilizar, como é possível verificar na figura 1.

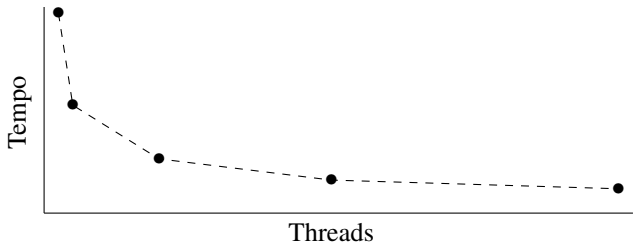

Fig. 1. Evolução do tempo de execução em função do número de threads para 4 clusters.

É de interesse apontar que principalmente com 32 threads, o tempo de execução ao aumentar o número de threads, é sempre aproximadamente metade. Em ambos os casos a diferença maior foi de uma para duas threads.

Quanto ao número de ciclos, era de esperar que aumentasse. O que surpreendeu o grupo, foi o facto de não aumentar de forma linear, sendo que aumentou significativamente de duas para oito threads, mantendo-se de seguida aproximadamente igual.

Quanto ao número de instruções não houveram aumentos significativos, o que resultou num CPI que aumentou similarmente ao número de ciclos.

TABLE I
RESULTADOS OBTIDOS EM VERSÕES DIFERENTES DE COMPILAÇÃO.

#Clusters	#Threads	Tempo de Execução	#CC	#Instruções	CPI	L1-dcache-load-misses
4	1	3.974	11,917,588,179	27,713,555,590	0.4	28,542,684
	2	2.146	11,912,520,373	27,714,629,431	0.4	28,483,789
	8	1.079	19,134,473,046	27,736,485,547	0.7	28,833,592
	20	0.657	19,506,873,880	27,890,723,379	0.7	28,955,015
	40	0.484	19,550,980,977	27,832,828,656	0.7	29,050,370
32	1	18.184	57,837,985,614	110,953,115,319	0.5	31,310,581
	2	9.541	57,837,186,677	110,960,332,564	0.5	31,408,400
	8	4.530	97,190,727,457	111,039,810,646	0.9	33,681,037
	20	2.682	101,658,799,232	113,298,058,638	0.9	36,054,858
	40	1.168	98,595,058,248	111,472,804,393	0.9	33,952,425

Por último, o número de misses na cache L1 aumentaram de forma linear ligeiramente.

IV. CONCLUSÕES E TRABALHO FUTURO

Infelizmente, devido à falta de cooperação do cluster da universidade, foi extremamente difícil obter os resultados, sendo que correr o programa apenas uma vez era capaz de demorar mais de 10 minutos, para depois se descobrir que tinha valores estranhos.

O grupo também gostaria de ter experimentado quantidades diferentes de cores e threads, mas foi impossível, devido à falta de tempo.

Por conclusão, o grupo pode adquirir mais conhecimentos acerca paralelismo e foi interessante observar o impacto que pode ter num programa.

V. ANEXOS

A. Melhores resultados obtidos

```
N = 10000000, K = 4
Center: (0.250, 0.750) 2498613
Center: (0.250, 0.250) 2501832
Center: (0.750, 0.250) 2499167
Center: (0.750, 0.750) 2500388
Iterations: 20 times

 Performance counter stats for './k_means 10000000 4 40':

        29,050,370      L1-dcache-load-misses
    27,832,828,656      inst_retired.any          #      0.7 CPI
    19,550,980,977      cycles

       0.483976668 seconds time elapsed

       6.709806000 seconds user
       0.022961000 seconds sys
```

Fig. 2. Resultado final com melhor tempo de execução para 4 clusters

B. Evolução do tempo de execução para 32 clusters

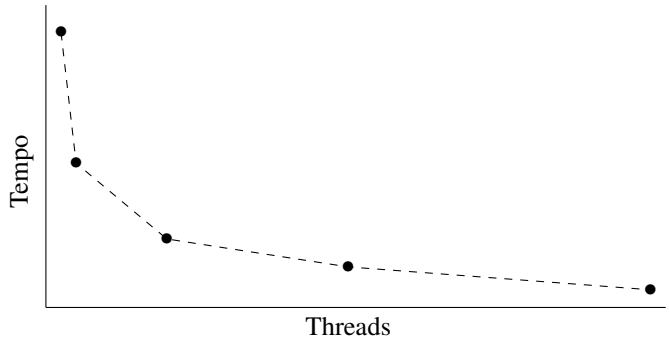

Fig. 3. Evolução do tempo de execução em função do número de threads para 32 clusters.