Computação Paralela - Trabalho Prático nº2

Diogo Manuel Brito Pires
Mestrado em Engenharia Informática - 1ºano
Universidade do Minho - Braga, Portugal
PG50334

Miguel Ângelo Machado Martins
Mestrado em Engenharia Informática - 1ºano
Universidade do Minho - Braga, Portugal
PG50655

***Abstract*—O 2º de 3 trabalhos práticos que serão desenvolvidos nesta UC. Este tem como objetivo avaliar a aprendizagem das técnicas de paralelismo para melhorar a *performance* do programa, através da análise das várias primitivas *OpenMP*.**

I. Mudanças de aspetos da fase anterior

Como não ficamos satisfeitos com o resultado da fase anterior, o grupo decidiu investir algum tempo no melhoramento do progama, antes de avançar para os objetivos desta segunda fase.

Assim sendo, começamos por colocar as nossas estruturas de dados alocadas em memória dinâmica (com o uso de *mallocs*), bem como proceder à substituição da matriz ***clusterPos*** por um *array* (***whichCluster***), que associa cada amostra ao *cluster* ao qual pertence.

Esta mudança permitiu que o número de *misses* na cache de nível 1 diminuisse para menos de metade, aproximando-se da média obtida pelos vários grupos na fase 1, e que a versão anterior superava substancialmente. Na figura 1 dos anexos pode-se observar efetivamente que esse valor melhorou de 298×10^6 para 134×10^6.

II. Análise da carga computacional

Para escolher as partes do programa que devíamos paralelizar, decidimos estudar qual o impacto que cada uma delas tinha na execução. Assim, saberíamos onde seria mais eficaz implementar primitivas de paralelismo. Para tal, usamos os comandos: ***perf record*** e ***perf report*** (Figura 2), e no fim vimos a proporção de tempo computacional utilizado pela função em causa relativamente ao tempo total, e acompanhamos com explicações para tais resultados:

- ***attribution*** (85%) - Esta função, tem um custo elevado pelas várias operações que efetua. Esse custo é proporcional ao produto do número de amostras com o número de clusters ($N \times K$). Assim, para cada amostra, temos de fazer duas multiplicações, duas subtrações, e uma soma, entre valores do tipo *float*. Analisando as instruções com maior custo, com o *perf report*, verificamos que são instruções do tipo *movaps*, *mulss* e *subss*, todas elas instruções sobre *floats*.
- ***geometricCenter*** (7.6%) - Esta função tem um custo que evolui linearmente com o número de amostras. Cada uma destas iterações realiza duas somas com valores do tipo *float*. Após a análise do *perf report*, verificamos que são instruções como *adss* e *movss*, operações sobre *floats*, que mais influenciam o elevado custo de execução desta função.
- ***random*** (5.52%) - Este custo é constante no nosso algoritmo, visto que a inicialização de cada uma das amostras implica a evocação desta função duas vezes, para a componente *x* e *y*. Logo, quanto maior for a proporção deste custo, melhor é o nosso algoritmo, visto que é um custo fixo base, que não conseguimos melhorar.

III. Alternativas para exploração de paralelismo

O critério para escolhermos quais as funções que devíamos paralelizar foi o peso computacional das mesmas no programa total, e para isso utilizámos a análise da secção anterior.

De seguida, explicamos as várias alternativas que consideramos para implementar paralelismo no nosso programa nas regiões mencionadas anteriormente.

- ***attribution*** : Um dos propósitos da alteração da versão entregue relativamente à fase anterior foi a utilização de paralelismo na função de atribuição. Com o método atual, o cálculo da atribuição de cada amostra não origina qualquer dependência de dados entre as várias amostras. Assim, é perfeitamente possível utilizar paralelismo no ciclo que tem complexidade linearmente proporcional ao número de amostras, através da primitiva *#pragma omp parallel for*. Também seria possível ter K *threads* a calcular as distâncias da amostra ao centro geométrico de cada *cluster*, porque não existe dependência de dados, mas o *overhead* de gestão de *threads* iria retirar a vantagem possível.
- ***geometricCenter***: O custo nesta função está na análise de todas as amostras, cuja informação do *cluster* a que pertencem está armazenada no *array wichCluster*. Nesta função, a leitura do *cluster* a que cada amostra pertence não gera problemas de concorrência na leitura de dados. No entanto, o cálculo dos centros geométricos é guardado nos mesmos $K \times 2$ endereços de memória (duas componentes, *x* e *y*, de cada centro geométrico). Assim, podem ser utilizados dois tipos de controlo a esta zona crítica: diretivas do tipo *single* ou *critical*; ou *reduce*. Iremos analisar este último caso por ser o mais eficiente nesta situação. Isto porque, caso utilizássemos uma das primeiras duas opções, acabaríamos por manter a execução do código sequencial. Com o *reduce*, guardámos o resultado das adições em variáveis

temporárias, associadas às *threads*, e assim podem ser executadas paralelamente. No fim, o resultado do programa pode sofrer alterações pois a ordem de adição das amostras e os arredondamentos afetam o resultado final, e com paralelismo a ordem não é garantida.

- ***random*** : Nós assumimos que o custo da operação de *random* se refere à inicialização, e uma das hipóteses para explorar paralelismo seria utilizar *threads* para a inicialização de cada amostra. No entanto, teríamos de garantir que os pares se mantêm, implicando usar um *#pragma omp single*, ou *critical*, após a instrução de *for* (Nesta análise foi importante ter em conta que o *srand* gera uma sequência determinística). Mas, ao fazermos isto, a execução da inicialização manter-se ia sequencial. Para além de se manter sequencial, teríamos o *overhead* de gestão de *threads*.

Para além disto, pensamos se compensaria implementar paralelismo em ciclos que tivessem um número de iterações igual ao número de *clusters*. Mas tal não faria sentido, porque o *overhead* associado à sincronização, e gestão de *threads*, não compensaria o paralelismo. Por outro lado, as instruções destes ciclos não são tão complexas ao ponto de o paralelismo ser recomendável, mas se o número de *clusters* fosse muito maior, já seria interessante analisar o ganho em implementar paralelismo nestas situações.

De seguida, avaliámos as diferentes formas de *scheduling* de *threads* às iterações dos ciclos *for* nas funções **attribution** e **geometricCenter**. Nós testamos as mesmas atribuições para as duas funções pois possuem caraterísticas semelhantes, nomeadamente: terem uma carga computacional entre iterações aproximadamente igual; e só uma *thread* chegar àquela zona.

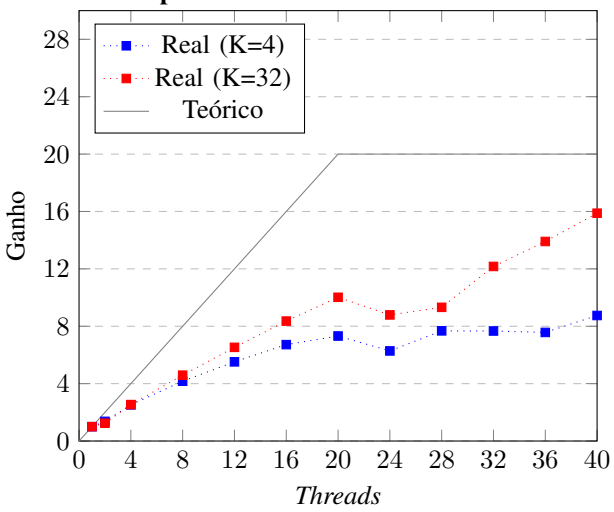
- ***Schedule* estático (default)** - É a melhor alternativa no nosso caso, porque evitamos analisar qual a *thread* que executa o corpo do *for* a cada iteração. Como todas as *threads* têm um custo aproximadamente igual, faz sentido dividir o número total de iterações pelo número de *threads* disponíveis.
- ***Schedule* dinâmico** - Piora relativamente ao anterior, porque no fim de cada iteração procura as *threads* livres, e atribui-lhes operações. Assim, existe mais *overhead* na atribuição de computação a *threads*. Conforme previámos, aumenta em 45% o número de instruções (instruções de sincronização), e 51% o número de ciclos (espera de outras *threads*).
- ***Guided Schedule*** - Apesar de ser um escalonamento melhor que o dinâmico, consideramos que no nosso caso não justifica a sua utilização, porque cada iteração tem um custo aproximadamente igual. Ao contrário do que pensávamos, melhora muito ligeiramente o tempo de execução do programa, relativamente ao escalonamento estático. Mas o aumento de complexidade associado a este escalonamento não justifica a sua utilização.

Assim, decidimos implementar paralelismo na função de atribuição e cálculo dos centros geométricos, com escalonamento estático. No centro geométrico, usamos a primitiva *reduce* para os arrays que guardavam os resultados das iterações.

IV. Otimização do desempenho

Para analisar a evolução do desempenho e escalabilidade do programa, apresentamos o seguinte gráfico, que possui 3 curvas distintas:

- **Curva teórica**, que representa a evolução do ganho esperada. Esta começaria por ser linear até 20 e depois estagnaria, visto que a máquina do *Search* possui 20 *cores* físicos;
- **Curva real para 4 *clusters***;
- **Curva real para 32 *clusters***.



Analisando as curvas reais, podemos concluir que à medida que o número de *threads* vai aumentando, o ganho na versão com o maior número de *clusters* vai-se tornando mais substancial.

Na função *geometricCenter* a zona que é paralelizada possui complexidade $O(N)$, e portanto o número de clusters não influencia a complexidade dessa zona, ou seja, um maior número de *threads* não representa um ganho diretamente proporcional ao número de *clusters*.

Contudo, na função *attribution* a zona que é paralelizada possui complexidade $O(K \times N)$. Desta forma, o ganho será mais significativo quanto maior for o número de *clusters*, uma vez que a percentagem de instruções executadas em paralelo será maior.

Por último, para ambas as curvas podemos observar que o ganho máximo ocorre com o número máximo de *threads* possíveis de utilizar (40), ultrapassando os 20 cores físicos. Logo, podemos concluir que a utilização de *Hyper-threading* funciona melhor quando temos mais clusters, porque a percentagem de instruções sequenciais executadas diminui relativamente ao total. A diferença entre as curvas do ganho ideal e do real deve-se ao número de instruções executadas sequencialmente ser relevante, impossibilitando assim que todo o código seja paralelizado, e o ganho ser máximo (***Lei de Amdahl***). Para além disso, o número de instruções realizadas também aumenta, influenciando o tempo de execução.

V. Anexos

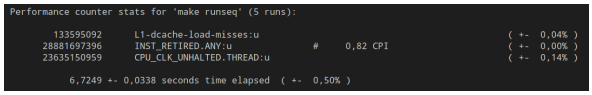


Fig. 1. Novas estatísticas da versão completa

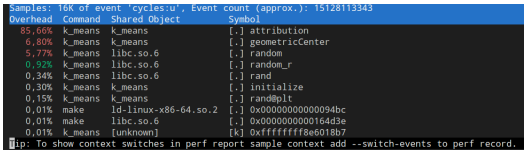


Fig. 2. Visão geral do ***perf report*** na versão sequencial

	Clusters (K)			
	4		32	
	Tempo	Ganho	Tempo	Ganho
Versão Sequencial	5.240 s	—	18.519 s	—
2 *Threads*	3.815 s	1.37	14.953 s	1.24
4 *Threads*	2.077 s	2.52	7.280 s	2.54
8 *Threads*	1.258 s	4.17	4.036 s	4.59
12 *Threads*	0.950 s	5.52	2.836 s	6.53
16 *Threads*	0.780 s	6.72	2.215 s	8.36
20 *Threads*	0.716 s	7.32	1.848 s	10.02
24 *Threads*	0.835 s	6.28	2.107 s	8.79
28 *Threads*	0.682 s	7.68	1.987 s	9.32
32 *Threads*	0.683 s	7.67	1.522 s	12.17
36 *Threads*	0.692 s	7.57	1.331 s	13.91
40 *Threads*	0.599 s	8.75	1.166 s	15.88

TABLE I
TABELA QUE RELACIONA O NÚMERO DE *threads* COM OS TEMPOS DE EXECUÇÃO, E RESPETIVOS GANHOS.

References

[1] : *The schedule clause*
https://learn.microsoft.com/en-us/cpp/parallel/openmp/d-using-the-schedule-clause?view=msvc-170