Computação Paralela - Trabalho Prático nº1

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 1º de 3 trabalhos práticos que serão desenvolvidos na UC de Computação Parelela. Este tem como objetivo avaliar a aprendizagem das técnicas de otimização de código, explorando as técnicas/ferramentas de análise de código para inferir essas mesmas otimizações.**

I. Introdução

Neste trabalho apresentamos a nossa implementação de um algoritmo de *k-means*, tendo como objetivo uma performance melhorada. Para tal utilizamos várias estratégias de otimização de acessos a memória e de processamento.

II. Armazenamento de dados

Desde o início da implementação tivemos o cuidado de armazenar os dados de forma a otimizar os acessos à memória, conforme será explicado no capítulo que refere a localidade espacial.

Para começar, armazenamos todas as coordenadas das amostras em 2 *arrays*, um com os valores do eixo das abcissas (x), e outro com os valores do eixo das ordenadas (y).

Para armazenar as amostras em *clusters* utilizámos uma matriz. Nessa matriz, cada linha corresponde a um *cluster*. Em cada posição da linha, está armazenado a posição do ponto nos *arrays* mencionados anteriormente, conforme está também descrito na imagem em anexo.

Num *array* à parte está armazenado o número de amostras que cada *cluster* possui. Aí, o índice do *array* indica qual o *cluster*, e o valor armazenado carresponde ao seu tamanho.

Finalmente, armazenamos o centro geométrico de cada um dos *clusters* em dois *arrays*, cada um com uma componente do centro (valores das abcissas e das ordenadas).

III. Localidade espacial e temporal

De seguida, descrevemos como as várias componentes do programa, aliadas à estratégia de armazenamento de dados, possuem localidade espacial e temporal:

A. Função de atribuição

- Quando é lida uma componente da amostra, as seguintes também são lidas. Por isso, quando na função de atribuição se vai buscar uma componente da amostra, as seguintes também são guardadas em cache (**localidade espacial**);
- Também as várias componentes dos centros geométricos são armazenadas em cache, pois estão em posições consecutivas na memória (**localidade espacial**);
- Na função de atribuição também temos **localidade temporal**, porque a mesma amostra é utilizada para todos os *clusters*, no ciclo mais interno. Isto acontece porque é necessário calcular a distância ao centro de todos os *clusters* para saber a qual vai ser atribuído.

B. Cálculo do centro geométrico

- Também existe **localidade espacial** no acesso às várias amostras que cada *cluster* contém. No ciclo mais interno, quando acedemos à posição da amostra, as posições seguintes também são guardadas em cache, porque o *cluster* é o mesmo. Assim, existe um custo elevado ao ler a primeira componente da amostra, mas em contrapartida as seguintes já estão armazenadas.

IV. Dependências de dados

Analisando o código, conseguimos inferir facilmente que não existem muitas dependências de dados no nosso programa. Seguidamente, estão enumeradas as dependências de dados identificadas, bem como a função na qual estão presentes:

- Na função ***attribution*** existem dependências *Read after Write* (**R**a**W**) entre as instruções das linhas 191/121, e 144/146. Além disso, também existe dependências *Write after Read* (**W**a**R**) entre as instruções das linhas 121/122, e 146/147.
- Na função ***geometricCenter2*** não existem dependências pois estas foram antecipadas e prevenidas, como se pode observar nas porções de código das linhas 184-187, visto que alternamos a escrita das variáveis, não colocando nem a escrita das abcissas, nem a das ordenadas seguidas. Neste caso impediu-se uma dependência *Write after Write* (**W**a**W**)

V. Otimizações gerais

Agora iremos descrever algumas otimizações gerais, e a forma como melhoraram a perfomance do nosso programa:

- Na função *attribution*, só verifica se existem mudanças nos *clusters* até encontrar uma. Quando já sabe que existem diferenças, para de comparar. Esta otimização acabou por ser ponderada ainda antes de qualquer tipo de testagem, pois pareceu vantajosa para reduzir o número de instruções;

- Na função *initialize*, em vez de dividirmos sempre pelo número máximo, calculamos uma vez o inverso, e depois multiplicamos sempre pelo inverso. Decidimos fazer isto porque as operações de divisão têm um custo muito elevado em termos de tempo de execução.
- No cálculo das distâncias da função *attribution*, não é necessário estar a calcular a raiz quadrada, pois as distâncias mantém o significado da diferença quer estejam ou não ao quadrado.
- Também na função *attribution*, se calcularmos nós os valores ao quadrado em vez de utilizarmos a função *pow* da biblioteca *math*, poupámos em tempo de execução.

Estas últimas 3 otimizações refletem-se na versão 2 na tabela colocada em anexo, que podem ser comparadas com os tempos da versão 1 da referida tabela. Comparando os melhores tempos de cada uma das versões, pode-se constatar que houve uma melhoria de 33% no tempo de execução.

VI. O0 vs. O2 vs O3

De seguida descrevemos as principais diferenças entre as três otimizações usadas acima, com todas as melhorias efetuadas nas otimizações gerais. Os tempos de execução, CPI e falhas no acesso à memória de nível 1 estão descritos na tabela dos anexos, e foram obtidas a partir da média de 5 testagens, usando-se o seguinte comando:

```
srun --partition=cpar perf stat -r 5 -e
L1-dcache-load-misses -M cpi make run
```

E em termos de código *assembly* gerado:

- Em O2, temos menos instruções por comparação a O0, passando de 541 para 368. Isto acontece porque em O2 são feitas muitas otimizações ao nível das instruções, que diminuem o número de instruções, e consequentemente o tempo de execução.
- Em O2, o compilador tem o cuidado de alinhar a memória, como indica a instrução *p2align 4* evitando falhas no acesso à memória, reduzindo o CPI, e consequentemente o tempo de execução.
- Em O3 existem otimizações na gestão de memória, pois aparece a instrução *call memset*, e isto não acontecia com os outros níveis de otimização.
- Apesar de a otimização O3 poder fazer vetorização, tal não aconteceu no nosso programa. Isso está explicado na secção relativa à vetorização.
- Por outro lado, apesar de o ficheiro O3 ser maior, é mais rápido por conseguir ter um CPI baixo.

VII. *Loop-Unrolling*

Para analisarmos a execução do programa com *loop_unrolling,* decidimos compará-lo com o código otimizado com a versão O3, e todas as otimizações analizadas na secção "Otimizações gerais" (versão 2 na tabela).

Para começar, utilizando a *flag* de *loop_unrolling* melhora-se pouco o tempo de execução, e piora o CPI, baixando para 1, como se pode observar nos valores obtidos na versão 3 na tabela. De seguida, tentámos explicar algumas das razões para essa perda de *perfomance*:

- Como nem sempre é útil fazer *loop_unrolling*, podemos ver que partes do código são muito parecidas à versão sem a *flag*. No entanto, em anexo colocamos um exemplo interessante da aplicação de *loop_unrolling*, quando utilizadas as otimizações O0 e O3, separadamente, com *loop_unrolling*.
- Como é observável na versão 3 na tabela, testamos *loop_unrolling* com vários níveis de otimização.
- Também é possível observar nas instruções geradas pelo *loop_unrolling* com flag *-O0*, que são geradas instruções de *no operation* no *assembly*, que apesar de aumentarem o número de instruções, diminuem ligeiramente o tempo de execução. (Optamos por omitir os valores obtidos com a *flag -O0* pelo motivo indicado em anexo).

Por outro lado, e tendo também sempre em conta a legibilidade do código, procurámos fazer manualmente *loop_unrolling* (versão 4 na tabela). Escolhemos fazer essa alteração na função de cálculo dos centróides, porque possui um ciclo com complexidade N, e os cálculos realizados em cada iteração são simples. A melhoria observada é bastante baixa nos tempos de execução.

Assim, podemos concluir que o *loop_unrolling* não melhora substencialmente a *perfomance* deste programa.

VIII. Vetorização

Ao estudarmos as caraterísticas que possibilitam a vetorização de um programa, descobrimos que o nosso programa possuía certas caraterísticas que o prejudicaram :

- Na *attribution* existem verificações condicionais dentro da iteração, que impedem a vetorização.
- Na *geometricCenter*, o acesso às amostras dos *clusters* tem de ser calculado individualmente, o que não permite a vetorização.

Os resultados otidos no *perf* estão representados na versão 5 na tabela, e demonstram que não se conseguiu obter um ganho de *performance* apreciável a usar *flags* de vetorização.

IX. Conclusões

Como reflexão final, consideramos que aprendemos vários conceitos, nomeadamente:

- Importância dos acessos à memória nos tempos de execução;
- Otimização do armazenamento de dados para melhorar a perfomance dos níveis de memória (*cache*);
- Analisar e melhorar a localidade espacial e temporal;
- Analisar a dependência de dados, e como evitar;
- Utilizar *loop_unrolling*, e estudar a sua perfomance;
- Vetorização.

Como trabalho futuro, pretendemos melhorar alguns aspetos do código, nomeadamente a forma como se guardaram os dados, fazendo a transição de alocação estática para dinâmica. Além disso, antes de inserirmos *multithreading* queremos melhorar os tempos de execução obtidos e tentar colocar o código vetorizável.

X. Anexos

Exemplo de *loop_unrolling* com a flag O0, , com comentários, onde se nota que instruções são repetidas:

```
movl    clusterCurrentPos(,%rax,4), %eax
        cltq
        movl    -20(%rbp), %edx
        movslq  %edx, %rdx
        imulq   $10000000, %rdx, %rdx
        addq    %rdx, %rax
        movl    clusterPos(,%rax,4), %eax
        cltq
        movss   x(,%rax,4), %xmm0
        movss   %xmm0, -40(%rbp)
        movl    -20(%rbp), %eax
        cltq
 // Inicia repetição
        movl    clusterCurrentPos(,%rax,4), %eax
        cltq
        movl    -20(%rbp), %edx
        movslq  %edx, %rdx
        imulq   $10000000, %rdx, %rdx
        addq    %rdx, %rax
        movl    clusterPos(,%rax,4), %eax
        cltq
        movss   y(,%rax,4), %xmm0
        movss   %xmm0, -44(%rbp)
        movl    -20(%rbp), %eax
        cltq
```

Exemplo de *loop_unrolling* com a *flag* O3, onde é visível a utilização de outra estratégia.

```
        subq    %rax, %r10
        subq    $4, %r10
        shrq    $2, %r10
        addq    $1, %r10
        andl    $7, %r10d
        je      .L58
        cmpq    $1, %r10
        je      .L84
        cmpq    $2, %r10
        je      .L85
        cmpq    $3, %r10
        je      .L86
        cmpq    $4, %r10
        je      .L87
        cmpq    $5, %r10
        je      .L88
        cmpq    $6, %r10
        jne     .L98
```

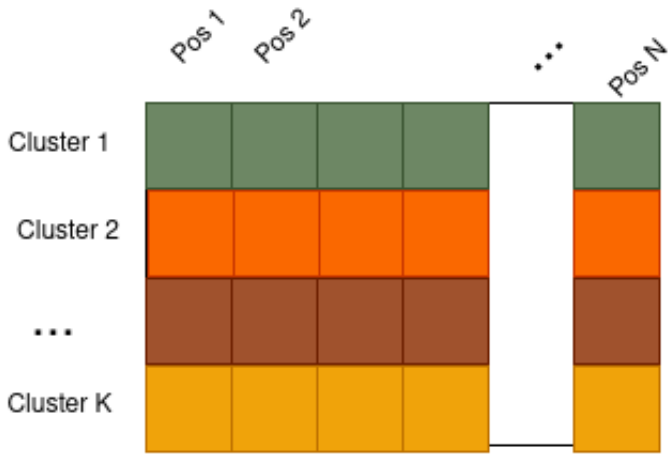

Fig. 1. Exemplo de estruturação de dados.

Nesta tabela fomos progressivamente retirando níveis de otimização nas testagens, deixando apenas os que produziam melhores tempos e seriam, portanto, mais úteis para estabelecer comparações.

TABLE I
ESTATÍSTICAS DE EXECUÇÃO

Versão		**Informação do *Perf***				
		Time(s)	CPI	#I	L1 Cache Misses	Cycles
1	O0	64.222 +- 0.334	0.8	256,693,448,361	303,807,653	208,700,475,982
	O1	21.58 +- 1.11	1.4	48,702,413,425	294,752,258	68,028,811,028
	O2	16.11 +- 1.03	1.2	44,626,603,771	293,440,733	51,550,525,873
	O3	15.669 +- 0.133	1.3	34,292,387,314	293,567,858	46,017,501,110
	Os	16.305 +- 0.975	1.0	49,390,586,791	293,553,991	51,756,963,334
	Ofast	14.4092 +- 0.0673	1.4	31,549,594,843	29,333,4091	43,580,678,139
2	O0	19.053 +- 0.603	0.6	91,994,595,591	301,903,970	55,344,393,049
	O1	9.676 +- 0.324	0.9	32,608,369,961	299,463,108	30,537,521,966
	O2	9.088 +- 0.249	0.9	32,122,002,027	299,352,382	27,639,477,383
	O3	7.1616 +- 0.0506	1.0	24,425,672,230	298,853,918	23,465,892,569
	Os	9.4208 +- 0.0137	0.9	31,831,546,610	299,389,903	27,320,675,813
	Ofast	7.8313 +- 0.0526	0.9	24,380,169,400	298,949,126	23,156,631,468
3	O1	9.176 +- 0.353	1.2	23,175,215,808	299,451,101	27,159,124,110
	O2	9.192 +- 0.627	1.1	23,454,513,157	299,368,954	26,753,431,544
	O3	9.368 +- 0.633	1.2	23,382,705,384	299,170,584	27,334,500,692
	Ofast	9.408 +- 0.436	1.2	23,312,575,665	299,270,153	28,552,744,387
4	O1	7.4552 +- 0.0443	1.0	23,172,518,951	298,686,886	22,455,682,158
	O2	7.0892 +- 0.0743	1.0	23,447,110,803	298,565,887	22,541,185,687
	O3	8.273 +- 0.430	1.0	23,387,799,789	298,931,848	24,552,914,161
5	O1	7.601 +- 0.736	1.0	23,175,667,484	298,745,702	24,052,451,701
	O2	7.380 +- 0.398	1.0	23,451,983,632	298,798,185	24,160,279,370
	O3	7.0656 +- 0.0352	1.0	23,381,730,987	298,516,528	23,168,519,569

[1]Versão base.
[2]Versão com uso da inversa e sem uso das funções da biblioteca *math*.
[3]Versão só com as *flags* de *loop unrooling*.
[4]Versão de *loop unrooling* com otimizações no código.
[5]Versão com as *flags* de ção.