Computação Paralela
Algoritmo k-Means - Paralelismo

1st Catarina Martins
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50289@alunos.uminho.pt

2nd Joaquim Roque
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50502@alunos.uminho.pt

***Resumo*—O algoritmo *k-Means* é um método de agrupamento de pontos que subdivide um conjunto de N amostras em k *clusters*. É um processo iterativo em que a cada amostra é-lhe atribuído um *cluster*, com base na menor distância euclidiana ao centro do mesmo (referido como centroide). O processo iterativo termina quando se verifica que o algoritmo convergiu, ou seja, nenhum ponto mudou de *cluster*.**

Index Terms*—otimização, *performance*, *openMP*, paralelismo, *threads*, *cluster

I. Introdução

O objetivo deste trabalho consiste em melhorar a performance do algoritmo desenvolvido na fase anterior recorrendo à exploração do paralelismo e às primitivas do *OpenMP*.

II. Identificação

Recorrendo ao algoritmo desenvolvido na fase anterior, o grupo começou por identificar os blocos de código com maior carga computacional. De facto, o ponto crítico do algoritmo está na fase de iteração sobre o vetor das amostras. Como tal, é lógico o uso de *threads* neste ciclo, distribuindo a carga por vários fios de execução. Este foi o único bloco de código identificado como passível de se poder explorar o paralelismo através de *multithreading*. Os outros ciclos (i.e. iterações pelo vetor dos *clusters*) ou contêm dependências de dados do tipo *read-after-write*, que faz com que seja desaconselhado o uso de *threads*, ou reúnem condições para que seja aplicada a auto-vetorização pelo compilador (como explorado na fase anterior, ou ainda porque se tratam de ciclos previsivelmente curtos, e o *overhead* associado à criação de *threads* traduzir-se-ia numa quebra de desempenho.

III. Implementação

A. Versão I

Numa primeira versão, cada *thread* calcula o novo *cluster* para cada amostra do conjunto das mesmas que lhe foi atribuído. Os centroides não se alteram, até à próxima iteração, garantindo assim que não existem *data races*. No entanto, para o posterior cálculo do ponto médio (ou seja, o novo centroide), é necessário garantir exclusão mútua no acesso para escrita à variável que acumula a soma parcial, através da diretiva *critical*. Isto é feito tantas vezes quanto o número de amostras, o que se revelou particularmente custoso em termos de eficiência.

B. Versão II

Na segunda versão, introduzimos alterações às estruturas previamente estabelecidas, mais concretamente no *ClusterVector*. Na versão original, era impossível recorrer a operações de *reduction* sobre as variáveis que representavam os *clusters*, devido ao facto de se usarem estruturas para o efeito, e não *arrays*:

```
typedef struct {

Sample centroid;
size_t size;

} Cluster;

typedef struct {

Cluster* restrict data;
size_t const size;

} ClusterVector;
```

Então, optamos por definir o *ClusterVector* da seguinte forma:

```
typedef struct {

float* xs;
float* ys;
size_t* sizes;
size_t const size;

} ClusterVector;
```

Para um dado *cluster* i, o índice i de cada *array xs, ys* e *sizes* contém a abcissa, a ordenada e o número de amostras no *cluster* em questão, respetivamente.

Desta forma, elimina-se a necessidade de recorrer à diretiva *critical*. As operações de *reduction* garantem que cada fio de execução tenha os seus vetores parciais privados, que são então combinados no final do ciclo. Esta alternativa é mais viável, uma vez que a *Versão I* implica que seja adquirido um *lock* tantas vezes quanto o número de amostras, independentemente do número de *threads*, o que torna esta solução

pouco escalável. Em particular, com o aumento do número de *threads*, poderemos ter várias em simultâneo em espera ao tentar adquirir o *mutex*.

Por outro lado, na *Versão II*, não existe um *overhead* no que toca a variáveis partilhadas entre os vários fios, uma vez que as mesmas são apenas usadas para leitura. No entanto, devido ao uso de *arrays* privados a cada fio, poderá ser notável um aumento do consumo de memória.

IV. Resultados

Todas os executáveis foram compilados com *gcc -O3 -DNDEBUG* (versão 7.2.0), e posteriormente corridos com o comando *srun –partition=cpar –cpus-per-task=$CPUS_PER_TASK perf -e instructions,cycles -M cpi bin/k_means 10,000,000 $CP_CLUSTERS $THREADS*. O comando foi executado 5 vezes em cada caso e registada a média dessas iterações para a análise de resultados que se segue. O critério de paragem usado foi o número de iterações, fixado nas 20, tal como indicado no enunciado.

Tabela I
Resultados

# Clusters	# CPUs per Task	# Threads	Tempo (s)
4	2	1	2,30
		2	2,07
		4	2,05
		8	2,01
	4	1	2,29
		2	1,24
		4	1,21
		8	1,24
	8	1	2,25
		2	1,24
		4	0,77
		8	0,72
32	2	1	13,71
		2	11,34
		4	13,45
		8	12,17
	4	1	13,47
		2	6,83
		4	6,77
		8	6,74
	8	1	12,85
		2	6,89
		4	3,76
		8	3,68

V. Análise de resultados

Os resultados demonstram que o aumento do número de fios de execução tem, no geral, um impacto positivo relativamente a tempo de execução, podendo haver melhorias de cerca de 4 vezes em certos casos. No entanto, é importante salientar que existe um limite até onde se podem aumentar o número de *threads* sem que a *performance* se degrade. Isso pode ser justificado sobretudo por questões de *hardware*, onde, por exemplo, com 2 CPUs por tarefa, a diferença entre o uso de 1 ou 8 fios de execução é apenas residual, comparativamente ao que seria esperado, pelo que não se verificou um grande benefício do uso de *threads* neste caso. É também importante mencionar que questões de sincronização das várias *threads* também poderão justificar a falta de melhorias significativas com o aumento do número das mesmas, nomeadamente quando se dobra o valor de 4 para 8. No nosso caso em particular, isto ocorre nas operações de *reduction*. As mesmas implicam que a *main thread* apenas pode avançar com a execução do código pós-ciclo após todas as outras *threads* terem terminado. Acrescentar ainda que a operação em si também poderá contribuir para a ausência de impacto ao duplicar o número de fios, sendo a mesma possivelmente bastante dispendiosa pelo facto de operar sobre vetores.

VI. Conclusão

Com a realização deste trabalho, pudemos aprofundar os conhecimentos obtidos nas aulas e durante a elaboração da fase anterior do trabalho prático, nomeadamente conceitos de paralelismo com recurso a *multithreading*, análise de *performance* e otimização. Como aspeto negativo, destacamos que apenas foi identificado um bloco de código com maior carga computacional para exploração de paralelismo. Há que reconhecer, porém, que tal advêm da forma como a nossa implementação foi feita e também da natureza do algoritmo em si, pelo que não é legítimo expectar outro desfecho. Por outro lado, há a destacar pela positiva o facto de termos conseguido adaptar, com relativa facilidade, o código de modo a suportar os novos requisitos, bem como a eficácia na escolha e aplicação das diretivas adequadas.