Trabalho Prático Nº 2

Computação Paralela, Mestrado Integrado em Engenharia Informática, Universidade do Minho

Júlio Beites Gonçalves
Mestrado em Engenharia Informática
Universidade do Minho
Braga
pg50537@alunos.uminho.pt

João Alexandre Gandra Ferreira
Mestrado em Engenharia Informática
Universidade do Minho
Braga
pg50461@alunos.uminho.pt

***Abstract*—Este trabalho tem como objetivo potencializar e explorar diferentes técnicas de paralelismo, com o objetivo de reduzir o tempo de execução do nosso programa.**

***Index Terms*—Optimização de código, C, paralelismo, threads, localidade espacial, localidade temporal, OpenMP**

I. Introdução

Fazendo uso do algoritmo k-means desenvolvido no TP1, a análise e implementação de paralelismo fazendo uso de primitivas OpenMP deve permitir que em diversas fases do algoritmo que seja possível dividir o processamento por vários fios de execução. A metodologia de desenvolvimento de programas paralelos aplicada é a seguinte descrita:

- Identificar quais os blocos de código com maior carga computacional;
- Apresentar e analisar diferentes alternativas para exploração de paralelismo para cada bloco identificado anteriormente;
- Selecionar a alternativa mais viável justificando com uma análise de escalabilidade;
- Implementação da versão paralela mais adequada ao ambiente de execução (nó do Search da fila cpar);
- Analisar o desempenho da proposta implementada;

Como validação do resultado e output do programa os resultados do nosso programa devem corresponder os seguintes valores:

- N = 10000000, K = 4
- Center: (0.250, 0.750) : Size: 2498728
- Center: (0.250, 0.250) : Size: 2501731
- Center: (0.750, 0.250) : Size: 2499396
- Center: (0.750, 0.750) : Size: 2500145
- Iterations: 20

II. Análise da Carga Computacional

Antes de começarmos a aplicar técnicas de paralelismo, começamos por analisar a carga computacional do nosso programa. Através da funcionalidade *perf record* conseguimos obter métricas da mesma, percebendo as funções onde o custo de execução é significativo e se são passíveis de tratar tendo em conta o objetivo deste trabalho. Através do *Anexo 1*, é possível identificar que a função *calculateBest - 36,41%* juntamente com a função *calculateCentroid - 11,73%* representam uma parte significativa de todo o custo de execução do programa, pelo que a nossa exploração de técnicas de paralelismo vai centrar-se essencialmente no corpo destas duas funções.

III. Exploração do Paralelismo

A. Função "calculateBestCluster"

Esta função têm como objetivo o cálculo e atribuição de um cluster para todos os pontos. No corpo desta função, é juntamente calculado o número de pontos pertencente a cada cluster.

Para explorar o paralelismo desta função, começamos por explorar o construtor de worksharing *for*. Analisando a execução do corpo do ciclo for e de que modos os fios de execução podem processar em simultâneo, retemos o seguinte:

- As variáveis *i* e *bClu* devem ser declaradas privadas, uma vez que foram definidas antes da zona crítica;
- Os arrays *pointCluster, pointX, pointY* podem ser partilhados entre os diferentes fios uma vez que apenas são consultados;
- O array *numPoints* deve sofrer uma redução no final da zona paralela, uma vez que os seus valores são constantemente consultados e alterados nos diferentes fios de execução;

É possivel verificar as diretivas OpenMP implementadas no *Anexo II*.

A implementação de paralelismo neste bloco de código permitiu diminuir o tempo de execução consideravelmente.

B. Função "calculateCentroid"

Esta função têm como objetivo o cálculo do Centroid associado a cada cluster.

Analisando o corpo da função, verificamos que apenas o primeiro construtor de worksharing *for* merece a nossa atenção, uma vez que a complexidade deste ciclo notável. Procedemos de seguida há exploração do paralelismo dentro deste construtor, retendo o seguinte:

- A variável *i* deve ser privada a cada fio. (Implícito na diretiva - *pragma omp for*);

- Os arrays *pointCluster, pointX, pointY* podem ser partilhados entre os diferentes fios uma vez que apenas são consultados;
- Os arrays *xSumCluster, ySumCluster* devem sofrer uma redução no final da zona paralela, uma vez que os os seus valores são constantemente consultados e alterados nos diferentes fios de execução;

É possível verificar as diretivas OpenMP implementadas no *Anexo III*.

A implementação de paralelismo neste bloco de código permitiu diminuir o tempo de execução consideravelmente.

IV. Análise da escalabilidade do programa

Uma vez que o programa pode correr sobre diferentes complexidades, variando o número de amostras e/ou clusters, a análise da escalabilidade do mesmo torna-se importante de modo a percebermos como este se comporta.

Para tal, mantendo a complexidade do programa (N = 10000000 && K == 4 ou K == 32) e variando o número de threads disponíveis para a execução do programa, obtemos algumas métricas sobre o ganho de performance:

Número de Clusters	Número de Threads	Tempo Sequencial	Tempo Paralelo	Ganho Performance
4	2	2.34 s	2.15 s	9%
4	3	2.34 s	1.60 s	46%
4	6	2.34 s	0.92 s	154%
4	10	2.34 s	0.63 s	271%
4	15	2.34 s	0.55 s	325%
4	20	2.34 s	0.51 s	358%
4	32	2.34 s	0.50 s	368%
32	2	14.3 s	13.78 s	4 %
32	3	14.3 s	7.86 s	82 %
32	6	14.3 s	5.25 s	172 %
32	10	14.3 s	3.30 s	333 %
32	15	14.3 s	2.46 s	481 %
32	20	14.3 s	1.91 s	649 %
32	32	14.3 s	1.34 s	967 %

Como possível de verificar na tabela, o aumento do hardware disponível para processamento de informação resulta num ganho de performance em todos os casos.

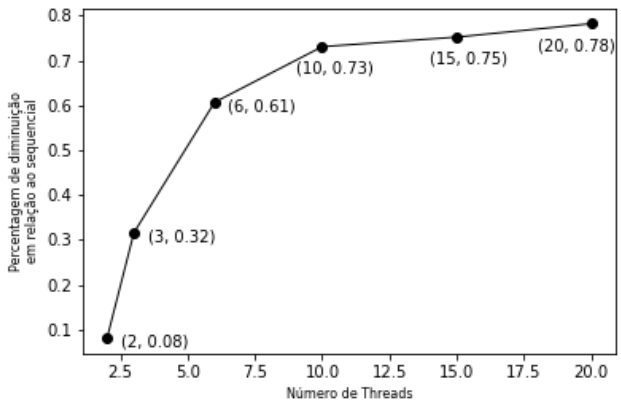

Fig. 1. Análise da Escalabilidade - N = 10000000, K = 4

Com estes dados, conseguimos afirmar que o nosso programa consegue dar resposta a problemas de congestão e sobrecarga, sendo facilmente escalável no futuro.

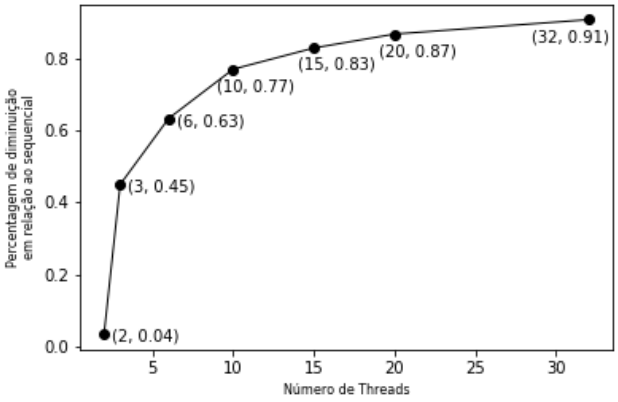

Fig. 2. Análise da Escalabilidade - N = 10000000, K = 32

V. Conclusão e melhorias possíveis

Tendo em conta os resultados apresentados, embora a complexidade do algoritmo estudado e as possíveis explorações de paralelismo sejam limitadas, conseguimos apresentar bons resultados e ganhos de performance consideráveis.

A análise dos diferentes blocos de código e de que forma o paralelismo podia ser explorado dentro destes sem nunca comprometer as escritas e leituras pelos diferentes fios de execução, permitiu-nos inferir conhecimento sobre boas técnicas de escrita de software de modo a permitir que este seja escalável. Do mesmo modo, fazendo uso da API OpenMP conseguimos aprender e compreender os diferentes recursos que aquela oferece no que toca a contextos de paralelismo de software.

VI. Attachments

```
 36.41%  k_means  k_means              [.] calculateBestCluster
 27.86%  k_means  libc.so.6            [.] random
 11.73%  k_means  k_means              [.] calculateCentroid
  7.81%  k_means  libc.so.6            [.] __lll_lock_wait_private
  5.89%  k_means  libc.so.6            [.] random_r
  2.49%  k_means  [unknown]            [k] 0xffffffff9ba00197
  2.13%  k_means  libgomp.so.1.0.0     [.] gomp_barrier_wait_end
  1.84%  k_means  libgomp.so.1.0.0     [.] gomp_team_barrier_wait_end
  1.71%  k_means  libc.so.6            [.] rand
  1.13%  k_means  k_means              [.] populate._omp_fn.0
  0.56%  k_means  libc.so.6            [.] __lll_lock_wake_private
  0.24%  k_means  [unknown]            [k] 0xffffffff9ba018b7
  0.19%  k_means  k_means              [.] rand@plt
  0.01%  k_means  [unknown]            [.] 0000000000000000
  0.00%  k_means  libc.so.6            [.] __default_morecore
  0.00%  k_means  ld-linux-x86-64.so.2 [.] 0x000000000000e7e8
  0.00%  k_means  ld-linux-x86-64.so.2 [.] 0x00000000000167f4
  0.00%  k_means  libgomp.so.1.0.0     [.] gomp_team_barrier_wait
  0.00%  k_means  libgomp.so.1.0.0     [.] omp_get_num_threads
  0.00%  k_means  libgomp.so.1.0.0     [.] gomp_thread_start
  0.00%  k_means  libc.so.6            [.] 0x0000000000108a4d
```

Fig. 3. Anexo I - Análise da Carga Computacional

```
int bClu = -1;
int i;
#pragma omp parallel for num_threads(t) private(i, bClu) reduction(+:numPoints[:k])
for(i = 0; i < n; i++){
    bClu = bestCluster(clusterX, clusterY, pointX[i], pointY[i], n, k, t);   // Calcula o melhor cluster para o ponto
    if(bClu != pointCluster[i]){    // Caso o melhor cluster seja diferente do atribuido anteriormente é atualizado
        if(pointCluster[i] != -1)
            numPoints[pointCluster[i]]--;   // Decrementamos o numero de pontos associado a esse cluster

        numPoints[bClu]++;  // Incrementamos o numero de pontos associado a esse cluster
    }
    pointCluster[i] = bClu;
}
```

Fig. 4. Anexo II - Função CalculateBestCluster

```
#pragma omp parallel num_threads(t) reduction(+:xSumCluster[:k]) reduction(+:ySumCluster[:k])
{
    #pragma omp for //reduction(+:xSumCluster[:k] ySumCluster[:k])
    for(int i = 0; i < n; i++){
        xSumCluster[pointCluster[i]]  += pointX[i];
        ySumCluster[pointCluster[i]]  += pointY[i];
    }
}
```

Fig. 5. Anexo III - Função CalculateCentroid