Trabalho Prático Nº 3

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, processos, localidade espacial, localidade temporal, OpenMP, MPI**

I. Introdução

Fazendo uso de um algoritmo de cálculo k-means desenvolvido, a análise e implementação de técnicas de paralelismo deve permitir que em diversas fases do algoritmo que seja possível dividir o processamento por vários fios de execução, sempre com um objetivo final de reduzir o tempo de execução do programa. Uma vez com o objetivo de aumentar o tempo de execução do nosso algoritmo já explorado através de primitivas de OpenMP, decidimos explorar e aplicar novas formas de paralelismo com recurso ao padrão de linguagem de programação MPI que permite que vários processos possam se comunicar e cooperar em conjunto. A metodologia de desenvolvimento de programas paralelos usada para este estudo é a seguinte descrita:

- Análise da Carga Computacional;
- Análise e Estudo de Paralelismo;
- Análise do Sistema;
- Implementação e Análise do Algoritmo Implementado;
- Análise de Desempenho;
- Analise de Escalibilidade do Sistema;

Nota: O código base que serve de suporte ao estudo de paralelismo, já conta com implementação de primitivas de OpenMP fazendo uso de *Threads*.

II. Análise da Carga Computacional

Antes de iniciarmos o estudo do paralelismo do nosso programa, devemos começar por analisar a carga computacional do nosso programa de modo a identificarmos o(s) bloco(s) de código sobre o qual devemos focar a nossa atenção. Através das funcionalidades *perf record* e *perf report* conseguimos obter métricas da carga computacional, percebendo as funções onde o custo de execução é significativo. Como é possível verificar através do Anexo 1, a função *calculateBestCluster* representa massivamente a maioria do tempo de execução do programa (89,27%). Através da testes similares, aumentado o tamanho dos parâmetros de entrada, o overhead desta função aumenta tendo em conta o tempo total de execução.

Overhead	Shared Object	Symbol
89,27%	k_means	calculateBestCluster

Parâmetros Utilizados:

- Pontos: 10000000
- Clusters: 4
- Threads: 6

III. Análise e Estudo de Paralelismo

Uma vez encontrado o bloco de código com elevada carga computacional na execução do programa, vamos analisar este bloco e explorar de que forma podemos melhorar a performance do mesmo.

Este bloco tem como principal objetivo o cálculo do cluster a que cada ponto está associado em cada iteração do algorítmo, pelo que, a cada iteração da função, um ciclo em ordem a N é executado. Aqui reside o motivo da grande carga computacional presente neste bloco de código, sendo o nosso objetivo distribuir esta carga através de técnicas de paralelismo.

De notar, que este bloco já anteriormente tinha sido explorado em termos de paralelismo através do uso de primitivas OpenMP com recurso a *Threads* através da seguinte primitiva:

```
#pragma omp parallel
for num_threads(t) private(i, bClu)
    reduction(+: numPoints[:k])
```

Para explorar o paralelismo deste bloco de código, decidimos usar o MPI (Message Passing Interface). Este é um padrão de linguagem de programação amplamente utilizado em sistemas de computação paralela que permite que vários processos se comuniquem e cooperam em conjunto em um sistema de computação distribuído.

IV. Análise do Sistema

De seguida, estão expostas as características do nodo utilizado para a execução do nosso programa:

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	40
Online CPU(s) list	0-39
Thread(s) per core	2
Core(s) per socket	10
Socket(s)	2
NUMA node(s)	2
Vendor ID	GenuineIntel
CPU family	6
Model	62
Model name	Intel(R) Xeon(R) E5-2670 v2 @ 2.50GHz
Stepping	4
CPU MHz	1237
CPU max MHz	3300
CPU min MHz	1200
BogoMIPS	4999
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	25600K

V. Implementação e Análise do Algoritmo Implementado

A. Introdução ao MPI

De forma a implementarmos técnicas de paralelismo com recurso ao MPI, devemos preparar o ambiente de execução da mesma.

```
MPI_Init(&argc, &argv);

int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

MPI_Init é a função inicializadora do ambiente MPI. *MPI_Comm_size* retorna o número de processos disponivies para comunicação, e a função *MPI_Comm_rank* retorna o identificador único de cada processo (rank). Estas funções são usadas para garantir a comunicação entre os processos paralelos.

Nota: De forma a garantir a correta execução do nosso programa, o **Processo 0** será o nosso processo master, responsável por repartir, reunir e espalhar informação pelos restantes processos.

Deste modo, este será o processo responsável por calcular a lista de pontos a ser utilizada no algoritmo.

```
if (world_rank == 0)
{
   for (int i = 0; i < k; i++)
      numPoints[i] = 0;
   populate(numPoints, pointX, pointY,
      pointCluster, clusterX, clusterY,
      n, k, t);
}
```

B. Implementação do MPI

A nossa função alvo, tal como explicado anteriormente é a função *calculateBestCluster*. Esta permite calcular o cluster a que cada ponto está associado em cada iteração do algoritmo. A cada execução desta, apenas são alterados os arrays que contêm informação sobre o cluster a que cada ponto pertence e o número de pontos que cada cluster contêm.

Desta forma, a principal dificuldade relaciona-se com garantir a correta distribuição dos dados entre os diferentes processos.

Uma vez que os pontos mantém-se inalterados durante toda a execução do algoritmo, estes são passados aos diferentes processos apenas uma vez.

```
MPI_Scatter(pointX, (n / numero_processos)
    + 1, MPI_FLOAT, recv_pointX, (n /
    numero_processos) + 1, MPI_FLOAT, 0,
    MPI_COMM_WORLD);
MPI_Scatter(pointY, (n / numero_processos)
    + 1, MPI_FLOAT, recv_pointY, (n /
    numero_processos) + 1, MPI_FLOAT, 0,
    MPI_COMM_WORLD);
```

Durante o resto do algoritmo, é necessário preparar os dados iterativamente para que os processos executem a função objetivo e finalmente voltar a reunir a informação no processo master.

C. Pré-Função

```
MPI_Scatter(pointCluster, (n /
    numero_processos) + 1, MPI_INT,
    recv_pointCluster, (n /
    numero_processos) + 1, MPI_INT, 0,
    MPI_COMM_WORLD);
   MPI_Scatter(numPoints, k, MPI_INT,
       recv_numPoints, k, MPI_INT, 0,
       MPI_COMM_WORLD);
   MPI_Bcast(clusterX, k, MPI_FLOAT, 0,
       MPI_COMM_WORLD);
   MPI_Bcast(clusterY, k, MPI_FLOAT, 0,
       MPI_COMM_WORLD);
```

Através da primeira chamada da função *MPI_Scatter* dividimos o array de pontos identificativos de cada cluster pelo número de processos disponíveis para execução. Na segunda chamada da mesma função, passamos o array que contém informação sobre o número de pontos de cada cluster para os diferentes processos. Por último, fazemos uso da função MPI_Bcast para espalhar informação sobre os clusters por todos os processos.

D. Pós-Função

```
MPI_Gather(recv_pointCluster, (n /
    numero_processos) + 1, MPI_INT,
    pointCluster, (n / numero_processos) +
    1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Reduce(recv_numPoints, numPoints,
      k, MPI_INT, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

Após a execução da função *calculaBestCluster* em paralelo por vários processos, é necessário reúnir novamente os dados no nosso processo master. Através da função *MPI_Gather*, conseguimos reúnir novamente os dados sobre o nosso array pointCluster. Através da função *MPI_Reduce*, juntamos novamente os dados referentes ao array num_points calculados nos diferentes processos, reúnindo-os com a particularidade de ser efetuada uma operação de redução aos mesmos.

VI. Análise de Desempenho

Nesta secção, iremos expor alguns testes de desempenho ao nosso programa. Para avaliar a eficiência do programa paralelo e a escalabilidade da solução, realizamos vários testes variando o número de pontos, clusters, o número de clusters e threads. Para verificar a eficácia da paralelização, comparamos os resultados obtidos com a versão do programa sequencial. Aquando da execução destes testes, o cluster encontra-se sobrecarregado pelo que os resultados podem apresentar alguma distorção.

Como valores de comparação, iremos usar os seguintes valores obtidos através da execução da versão sequencial do nosso programa:

```
4 Clusters
Tempo Total: 2.43;
Tempo Paralelizvel: 2.27;
Parte do Programa Sequencial: 0.066;

32 Clusters:
Tempo Total: 14.311;
Tempo Paralelizvel: 13.99;
Parte do Programa Sequencial: 0.022;
```

Para o cálculo do valor teórico de speedUp do nosso programa, faremos uso das seguintes fórmula:

Amdahl's law

$$S_{\mathrm{P}} = \frac{1}{f + (1-f)/P} \tag{1}$$

onde f representa a porção de código sequencial e P o número de unidades de processamento.

4 Clusters

$$S_{\mathrm{P}} = \frac{1}{0.066 + (1-0.066)/P} \tag{2}$$

32 Clusters

$$S_{\mathrm{P}} = \frac{1}{0.022 + (1-0.022)/P} \tag{3}$$

A. Número de Pontos e Clusters

Nesta secção iremos expor o nosso programa a vários testes variando tanto o número de pontos como o número de clusters a que o mesmo é sujeito. O número de processos e threads utilizado foram 4 processos e 1 thread respetivamente. (Tabelas I II).

SpeedUp Teórico - 3.33

Número de Pontos	Sequencial	MPI (4 processos)	SpeedUp
100 000	0.049	0.166	0.295
250 000	0.076	0.172	0.441
500 000	0.145	0.233	0.622
1 000 000	0.266	0.279	0.95
5 000 000	1.221	1.021	1.196
10 000 000	2.431	1.302	1.867
25 000 000	5.973	2.763	2.161
50 000 000	13.232	5.892	2.246

TABLE I
Análise da Variação do Nº de Pontos - 4 clusters

SpeedUp Teórico - 3.76

Número de Pontos	Sequencial	MPI (4 processos)	SpeedUp
100 000	0.172	0.229	0.751
250 000	0.393	0.294	1.336
500 000	0.732	0.402	1.820
1 000 000	1.433	0.619	2.315
5 000 000	6.923	2.132	3.247
10 000 000	14.311	3.659	3.911
25 000 000	36.876	10.122	3.643
50 000 000	73.232	21.211	3.452

TABLE II
Análise da Variação do Nº de Pontos - 32 clusters

B. Número de Processos e Threads

Da mesma forma, tentamos averiguar a combinação ideal de processos e threads para a execução do nosso programa. Decidimos começar por manter o número de threads e variar apenas o número de processos. O número de pontos utilizados e clusters foram respetivamente 10 000 000 e 32. (Tabelas III IV)

Processos	Threads	Tempo de Execução	SeedUp Teórico	SpeedUp Prático
2	1	7.232	2.100	1.978
4	1	4.233	3.610	3.380
10	1	2.116	7.110	6.763
20	1	1.322	11.402	10.825
40	1	1.476	16.601	9.696

TABLE III
Análise da variação do nº de Processos - Nº Threads = 1

Processos	Threads	Tempo de Execução	SeedUp Teórico	SpeedUp Prático
2	2	4.154	3.610	3.445
4	2	2.564	6.940	6.763
10	2	1.523	11.402	9.396
20	2	1.432	16.601	9.994

TABLE IV
Análise da variação do nº de Processos - Nº Threads = 2

Por último, vamos testar diferentes combinações dos recursos disponibilizados de forma a testar aquelas que permitem tirar maior proveito dos recursos tendo em conta o tempo de execução do nosso programa.

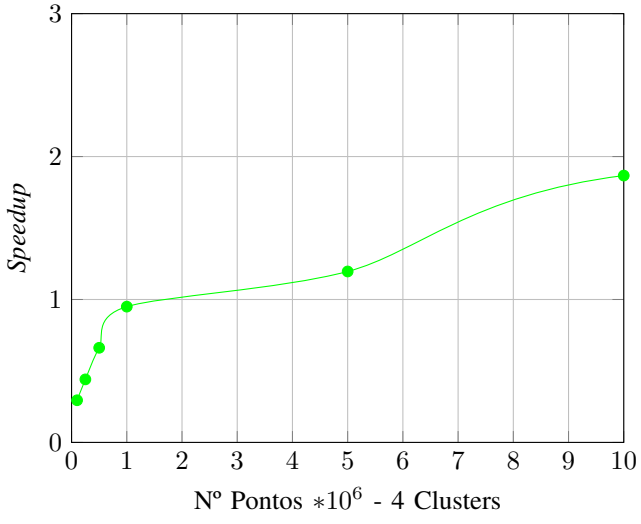
Processos	Threads	Tempo de Execução	SeedUp Teórico	SpeedUp Prático
2	10	2.292	14.28	6.244
4	5	1.482		9.657
5	4	1.789		7.999
10	2	1.432		9.994
20	1	1.415		10.114

TABLE V
ANÁLISE DA VARIAÇÃO DO Nº DE PROCESSOS E THREADS.

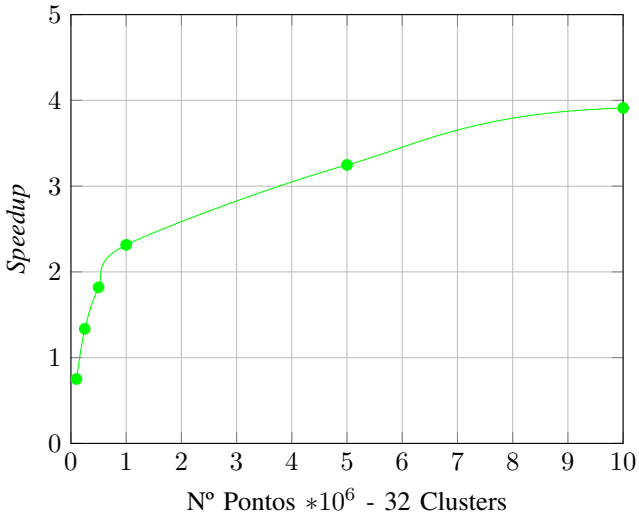
VII. ANÁLISE DE ESCALIBILIDADE DO SISTEMA

Quando são utilizados números de pontos mais baixos, a versão paralela do programa apresenta desempenho inferior em comparação com a versão sequencial. Isso deve-se ao fato de que a criação de processos e a comunicação apresenta um custo computacional superior ao benefício obtido pela paralelização. Conforme o número de pontos aumenta, o benefício começa a ser superior ao custo da criação dos canais de comunicação permitindo um speedUp do nosso algoritmo.

A. Número de Pontos e Clusters



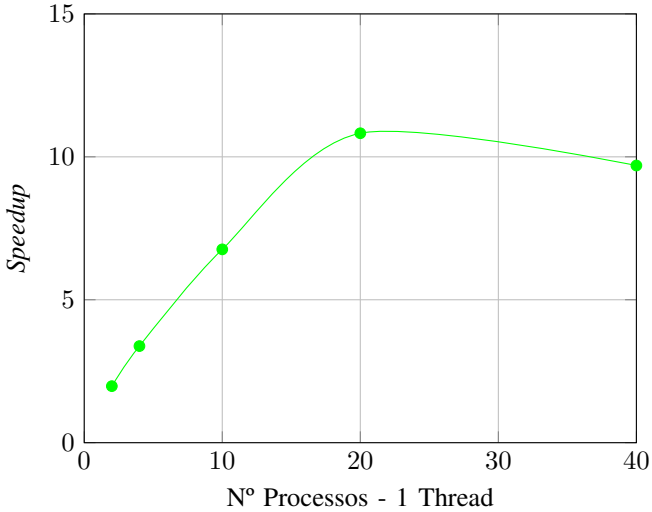
De forma semelhante, é possível verificar que para 32 clusters, a carga de trabalho aumenta significativamente permitindo tirar maior proveito da paralelização do nosso algoritmo.



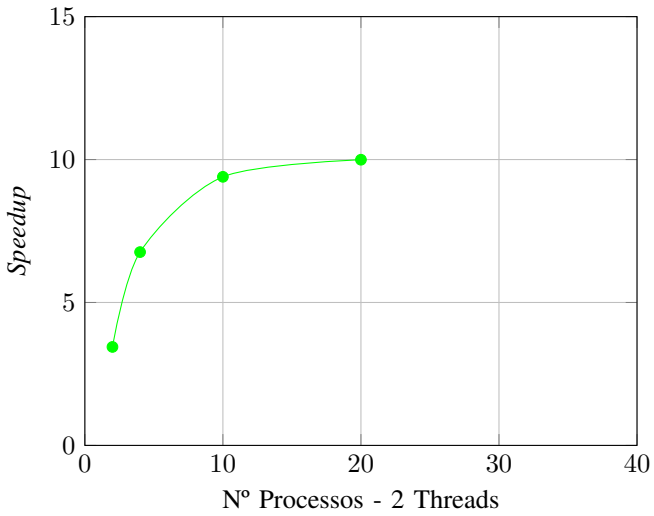
Além disso, o aumento do número de pontos também resulta em mais acessos à memória, seja ela a CPU ou a RAM, o que pode tornar os acessos mais lentos. Portanto, é possível concluir que o uso de processos adicionais é benéfico quando o número de pontos aumenta.

B. Número de Processos e Threads

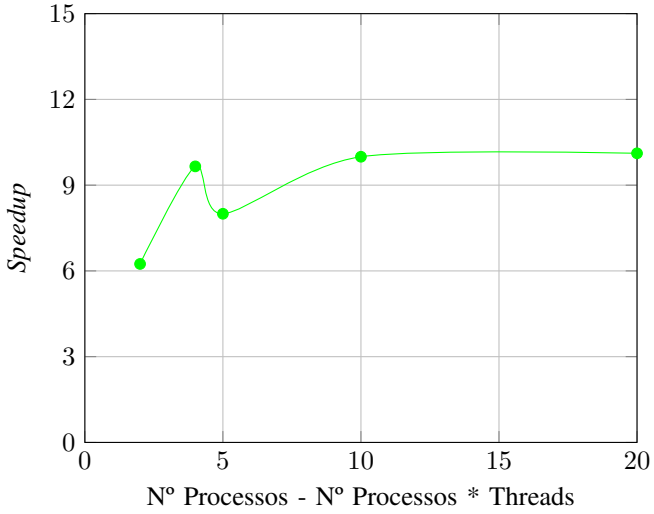
Na etapa inicial de testes, variou-se apenas o número de processos, para avaliar como isso afetaria o desempenho do programa. Com o aumento do número de processos, foi possível observar uma melhoria no desempenho, embora este fosse cada vez mais distante do teórico, possivelmente devido ao overhead da comunicação entre processos.



P



De maneira geral, comparando os tempos medidos entre o uso de um determinado número de processos com 2 threads por processo e o uso do dobro dos processos com uma thread, usar 2 threads traz uma leve melhoria no desempenho do programa, especialmente para valores mais elevados de processos.



Por último, análisamos a utilização de diferentes números de processos e threads tal que **Processos * Threads = 20**. Através do seguinte gráfico conseguimos analisar que os melhores resultados surgem quando são utilizados um maior número de processos, com a execeção da utilização de 5 processos. Por motivos alheios, não conseguimos perceber o motivo pelo decréscimo do speedup para estes valores. Acrescentamos ainda, que deveria ter sido utilizada uma amostra maior de forma a verificar melhor como se comporta a execução do programa variando estes argumentos uma vez que o seguinte gráfico se torna inconclusivo.

VIII. Conclusão

A avaliação da implementação foi realizada em termos de desempenho, comparando o tempo de execução da versão sequencial e paralela do algoritmo tendo em conta a possibilidade de explorar diferentes recursos (Threads, Processos). Os resultados obtidos indicam que a versão paralela do algoritmo apresenta um ganho significativo de desempenho em comparação com a versão sequencial, especialmente em conjuntos de dados de grande escala. A grande afluência ao cluster disponibilizado, os largos periodos de tempo em que este esteve down, e a falta de melhores recursos por parte do mesmo impossibilitaram uma maior amplitude de testes. Contudo, pensamos ter atingido os diferentes objetivos da unidade curricular!

IX. Anexos

```
Samples: 4K of event 'cycles:ppp', Event count (approx.): 86209322563
Overhead  Command  Shared Object      Symbol
  89.27%  k_means  k_means            [.] calculateBestCluster._omp_fn.0
   8.09%  k_means  [unknown]          [k] 0xffffffffa158cbc0
   1.38%  k_means  k_means            [.] calculateCentroid._omp_fn.1
   0.58%  k_means  libc-2.17.so       [.] __random_r
   0.24%  k_means  libc-2.17.so       [.] __random
   0.18%  k_means  [unknown]          [k] 0xffffffffa1195377
   0.17%  k_means  ld-2.17.so         [.] _dl_fixup
   0.03%  k_means  k_means            [.] populate
   0.03%  k_means  libc-2.17.so       [.] rand
   0.01%  k_means  [unknown]          [k] 0xffffffffa0e6d5f8
   0.01%  k_means  [unknown]          [k] 0xffffffffa0eae179
   0.01%  k_means  [unknown]          [k] 0xffffffffa158ba8c
   0.01%  k_means  [unknown]          [k] 0xffffffffa158c48a
   0.00%  k_means  [unknown]          [k] 0xffffffffa1596ed4
   0.00%  k_means  [unknown]          [k] 0xffffffffa158c3ad
   0.00%  k_means  [unknown]          [k] 0xffffffffa0fcdca6
   0.00%  k_means  [unknown]          [k] 0xffffffffa158bb12
   0.00%  k_means  [unknown]          [k] 0xffffffffa0f5ba18
   0.00%  k_means  [unknown]          [k] 0xffffffffa0f5baac
```

Fig. 1. Anexo I - Análise da Carga Computacional