Paralelização do algoritmo K-means

Marco Martins[1], Paulo Shirley[2]

[1]Universidade Aberta de Portugal, Lisboa, Portugal, marcopaulomartins@hotmail.com
[2]Universidade Aberta de Portugal, Lisboa, Portugal, Paulo.Shirley@uab.pt

Resumo

Neste artigo são exploradas as vantagens da paralelização do algoritmo K-means. O algoritmo é implementado na linguagem de programação C com a utilização de tarefas (*threads*) POSIX e são analisadas as consequências ao nível do desempenho pela utilização de programação multitarefa. O algoritmo K-means é essencialmente iterativo e a paralelização deste permite tirar partido do máximo de processadores disponíveis e com isso reduzir consideravelmente o tempo de execução. Este ganho no tempo total de execução permite um investimento no aprimorar do algoritmo de forma a obter resultados mais precisos, ou processar um maior volume de dados e manter a execução em tempo útil.

Palavras-chave: k-means, *pthreads*, linguagem de programação C, programação paralela multitarefa.

Title: Parallelization of the k-means algorithm

Abstract: This article explores the advantages of parallelizing the K-means algorithm. The algorithm will be implemented in the C programming language using POSIX Threads and the consequences of the use of multi-thread programming in terms of performance will be analysed. The K-means algorithm is essentially iterative, and its parallelization will make it possible to take advantage of as many processors as possible and thereby considerably reduce the execution time. This gain in the total execution time allows an investment in improving the algorithm to obtain more accurate results, or to process a larger volume of data and keep the execution time limited.

Keywords: k-means, pthreads, C programming language, multitasking parallel programming.

1. Introdução

Este trabalho foi desenvolvido no âmbito do projeto de final de curso da Licenciatura em Engenharia informática da Universidade Aberta e pretende-se, utilizando a linguagem de programação C, programar a paralelização do algoritmo K-means utilizando a biblioteca *pthreads*, a *interface* de tarefas POSIX (*Portable Operating Systems based on unIX*), explorando as vantagens que dai resultam.

K-means é um algoritmo de *Machine Learning* com aprendizagem não-supervisionada, pertencente ao grupo de algoritmos de *clustering* com uma abordagem gananciosa. Este algoritmo tem como objetivo agrupar entidades em vários conjuntos (*clusters*) seguindo critérios de similaridade previamente definidos. No entanto este algoritmo tem alguns pontos que podem ser otimizados se recorremos a programação paralela como a definição do número de *clusters* a usar, a escolha dos centros dos *clusters*, e a decomposição da execução em série, habitual do k-means.

1.1. Linguagem C

A linguagem C foi criada em 1972 nos Bell Telephone Laboratories por Dennis Ritchie com a finalidade de permitir a escrita de um sistema operativo (o unix), utilizando uma linguagem de relativo alto nível, evitando assim o recurso ao *assembly* que é uma linguagem de baixo nível onde cada instrução corresponde a uma instrução de código máquina e requer que o programador tenha conhecimentos sobre a arquitetura e recursos do processador uma vez que as instruções *assembly* são para determinado processador. Esta linguagem de programação não está vinculada a nenhum sistema operativo ou máquina, no entanto é conhecida por alguns como "linguagem de programação de sistema" pois tem sido usada na criação de compiladores e de sistemas operativos [Marques 2009].

O facto da linguagem C não ser de alto nível não é negativo, é simplesmente por lidar diretamente com caracteres, números e endereços, que são os objetos que a maioria dos computadores lida. Esta linguagem é relativamente pequena, o que a torna simples podendo ser aprendida rapidamente e é razoável que um programador tenha a expectativa de conhecer, entender e usar toda a linguagem. É rápida, conseguindo-se obter performances idênticas ao *assembly*, permite um desenvolvimento modular agilizando a divisão de projetos por módulos independentes, permite a utilização de macros e existe um vasto conjunto de bibliotecas que disponibilizam funções úteis para todo o tipo de tarefas. É utilizada por todo o mundo, a documentação existe em quantidade e qualidade e com o padrão ANSI definido tornou-se extremamente portátil, o que significa que os compiladores estão bem definidos e o código pode ser compilado em qualquer arquitetura de computadores sem ter de ser alterado.

1.2. Tarefas (*Threads*) POSIX

Para o desenvolvimento de programação paralela multitarefa é fundamental ter bem presente o conceito de tarefa (*thread*). As tarefas são ramificações simplificadas do fluxo de execução. A sua criação é eficiente e são executadas de forma independente, mas mantendo o contexto comum entre si, o que as torna num excelente mecanismo para gerir múltiplas atividades em paralelo.

Uma boa definição de tarefa é encontrada em [Marques 2009] *"Uma tarefa é um fluxo de atividade que se executa no âmbito de um processo já existente, executando uma ou várias funções do programa associado ao processo e partilhando o mesmo espaço de endereçamento, ou seja, os dados do processo, os ficheiros, etc"*.

As tarefas POSIX correspondem ao standard POSIX 1003.1c-1995 e são conhecidas como *Pthreads*. Os conceitos fundamentais para usar tarefas de forma produtiva são a concorrência (*concurrency*), sincronização (*synchronization*) e agendamento (*scheduling*). Com as *Pthreads* estes conceitos são aplicados de forma organizada, eficiente e portátil.

Vantagens da programação multitarefa são a exploração do paralelismo em sistemas multi-processador, a exploração mais eficiente de programação concorrente e uso de programação modular com relações claras entre eventos independentes no mesmo programa.

Usar um modelo multitarefa nem sempre é vantajoso, é necessário ter em consideração o tempo de sincronização das tarefas, quando há muita sincronização o desempenho pode diminuir com a utilização de tarefas. A complexidade do código quando se utiliza o modelo multitarefa também aumenta, surgem problemas como impasses (*deadlocks*), inversão de prioridades e condições de corrida (*races*). O processo de depuramento (*debug*) também se torna mais complexo neste modelo [Marques 2009].

Com a *interface* POSIX, para criar uma tarefa é necessário ter uma variável do tipo pthread_t para guardar o identificador da tarefa a ser criada e a sua criação é feita recorrendo à função pthread_create. Na criação de uma tarefa fica logo definido qual a função que ela irá executar. As funções a serem executadas por tarefas recebem um único argumento do tipo void* e devem retornar um valor do mesmo tipo. Para obter o valor de retorno de uma tarefa ou para saber quando a tarefa terminou utiliza-se a função pthread_join. Para transferir o controlo a *interface* POSIX disponibiliza a função pthread_yield e para terminar uma tarefa a função a utilizar é pthread_exit. As tarefas, em qualquer instante da sua vida, encontram-se num de quatro estados básicos, podem estar no estado *Ready* o que significa que estão prontas para executar o que lhe esta atribuído, no estado *Running* ou seja está nesse momento a executar a sua tarefa, no estado *Blocked* não pode correr por estar à espera de algum recurso, ou no estado *Terminated* significando que a tarefa terminou.

Como as tarefas normalmente trabalham em conjunto para atingir um objetivo comum, podem necessitar de aceder a recursos partilhados, para gerir os acessos de forma sincronizada aos recursos partilhados recorre-se a um *mutex* POSIX que consiste numa estrutura de dados do tipo pthread_mutex_t e utiliza-se para que os acessos à memória de um mesmo dado seja feita com exclusão mútua, ou seja para garantir que num determinado instante, só uma tarefa tem acesso, por exemplo a uma variável. Para criar um *mutex* utiliza-se a função pthread_mutex_init, para fechar o *mutex* é a função pthread_mutex_lock, para abrir o *mutex* existe a função pthread_mutex_unlock e para terminar o *mutex* a função pthread_mutex_destroy.

1.3. Algoritmo K-means

A mineração de dados compreende os algoritmos principais que permitem obter "*insights*" e conhecimentos fundamentais de dados massivos. É um campo interdisciplinar que combina conceitos de áreas afins, como sistemas de bases de dados, estatísticas, aprendizagem de máquina e reconhecimento de padrões. Na verdade, a mineração de dados faz parte de um processo maior de descoberta de conhecimento, que inclui tarefas de pré-processamento, como extração de dados, limpeza de dados, fusão de dados, redução de dados e construção de recursos, bem como etapas de pós-processamento, como padrão e interpretação de modelos, confirmação e geração de hipóteses, etc. Este processo de descoberta de conhecimento e mineração de dados tende a ser altamente iterativo e interativo.

A análise exploratória de dados tem como objetivo explorar os atributos numéricos e categóricos dos dados individualmente ou em conjunto para extrair características-chave da amostra de dados por meio de estatísticas que fornecem informações sobre a centralidade, dispersão e muito mais. Permitem ainda extrair relações complexas entre os dados, desde que tenhamos uma maneira de medir a semelhança entre os pares de dois objetos abstratos.

O algoritmo K-Means é um algoritmo de *clustering*, não supervisionado, que classifica ou agrupa os dados recorrendo à análise e comparação dos seus valores.

Clustering é a tarefa de particionar os pontos em grupos naturais chamados *clusters*, de modo que os pontos dentro de um mesmo grupo são muito semelhantes, enquanto os pontos entre grupos são tão diferentes quanto possível. Dependendo dos dados e das características desejadas do *cluster*, há diferentes tipos de paradigmas de agrupamento, como baseados em representantes, hierárquicos, agrupamento baseado em densidade, baseado em gráfico e espectral.

K- MEANS (D, k, E):

t = 0

Randomly initialize k centroids: $u_{1,}^{t} u_{2,}^{t} \ldots, u_{k}^{t}, \in \mathbb{R}^{d}$

repeat

t ← t + 1

$C_j \leftarrow \emptyset$ for all $j = 1,\cdots,k$

// Cluster Assignment Step

foreach $x_j \in D$ **do**

// Assign x_j to closest centroid

$j* \leftarrow argmin_i\{\|x_j - u_i^{t-1}\|^2\}$

$C_{j*} \leftarrow C_{j*} \cup \{x_j\}$

// Centroid Update Step

foreach i = 1 to k do

$u_i^t \leftarrow \frac{1}{|C_i|}\sum_{x_j \in C_i} x_j$

until $\sum_{i=1}^{k} \|u_i^t - u_i^{t-1}\|^2 \leq E$

Figura 1. Pseudo-código Algoritmo K-means [Zaki 2014]

O algoritmo K-means é um algoritmo guloso que minimiza o erro quadrático dos pontos de suas respetivas médias de *cluster*, e realiza "*hard clustering*", ou seja, cada ponto é atribuído a apenas um cluster. A figura 1 mostra o algoritmo em pseudo-código.

2. Implementação

O Algoritmo K-means paralelizado foi implementado na linguagem de programação C recorrendo a tarefas POSIX e compilado com GCC [http://gcc.gnu.org/] numa máquina de quatro processadores com o sistema operativo Ubuntu Server 20.04.1 LTS [https://ubuntu.com/download/desktop]. Para controlo de versões foi utilizado o GIT e colocado o repositório privado no *github* [https://github.com/MarcoPSM/kmeans].

Os ficheiros de entrada são ficheiros de texto simples, onde cada linha corresponde a uma entidade e os atributos estão separados por espaço. O ficheiro de saída e idêntico ao de entrada, mas com um atributo adicional que contem o identificador do grupo (*cluster*) a que cada entidade ficou associada. Os atributos das entidades são valores numéricos do tipo *double*. A cada entidade são adicionados dois atributos extra, um para valores relacionados com cálculos de distância e outro para o identificador de grupo. Os centros (*centroids*) são inicializados com a técnica de inicialização para o algoritmo generalizado de Lloyd proposto por Katsavounidis, Jay Kuo e Zhang [Katsavounidis 1994]. As partes do algoritmo que foram paralelizadas foram: a inicialização dos centros, a atribuição das entidades aos grupos e o cálculo dos novos centros.

2.1 Descrição

O programa desenvolvido recebe por argumento o ficheiro de entrada, número de grupos (*clusters*) pretendidos, o limite de convergência e o número de tarefas a utilizar.

Exemplo de chamada ao programa passando os argumentos:

```
./kmeans -f datafiles/ds4d1milhoes.txt -k 32 -e 0.0001 -t 4
```

É verificada a dimensão dos dados de entrada, o número de entidades e o número de atributos, criada uma matriz para o *dataset*, com espaço para dois atributos adicionais que são necessários para dados relativos a distâncias e ao grupo associado. São criadas duas matrizes para os centros, uma para centros calculados e outra para centros a calcular, com n linhas e m colunas onde n corresponde ao número de grupos pretendido e m corresponde ao número de atributos que as entidades possuem. São inicializados os centros para a matriz dos novos centros e iniciado o ciclo do k-means, os centros são copiados para a matriz dos centros a associar, são associados os centros às entidades e calculados novos centros, determinada a convergência e se ainda não tiver atingido o limite definido é repetido o ciclo. Quando atinge o limite de convergência definido, é gravada a saída que contém o *dataset* mais o identificador do grupo a que cada entidade ficou associada. A figura 2 mostra o fluxograma do programa.

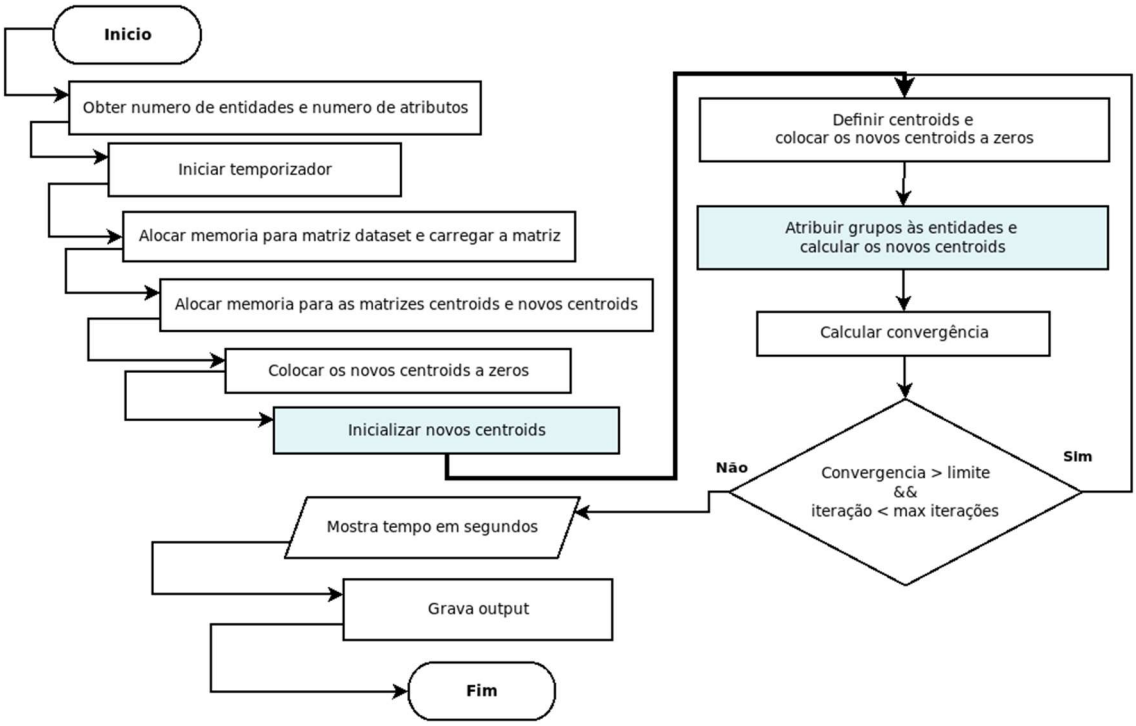

Figura 2. Fluxograma do programa desenvolvido.

2.2 Inicialização dos centros

A implementação da inicialização de centros recorre à técnica de inicialização para o algoritmo generalizado de Lloyd proposto por Katsavounidis, Jay Kuo e Zhang [Katsavounidis 1994]. O conceito base desta técnica é inicializar com entidades que estejam mais distantes entre si, pois é provável que venham a pertencer a *clusters* diferentes. A técnica consiste em seguir três passos. Primeiro calcula-se a norma de todas as entidades e fica a que tiver maior norma definida como o primeiro centro. De seguida calcula-se a distância de todas as entidades ao primeiro centro definido e a entidade mais distante será o segundo centro. No terceiro passo, que define os restantes centros, calcula-se a distância de cada entidade aos centros já definidos e escolhe-se a menor distância, depois verifica-se qual ficou com uma distância maior e essa entidade servirá como centro.

2.3 Paralelização

Sendo o algoritmo K-means um algoritmo iterativo a paralelização é feita a específicas partes do algoritmo. Optou-se por aplicar a paralelização na inicialização dos centros dos clusters, na atribuição das entidades a um cluster e no cálculo de novos centros. A forma como a paralelização foi implementada consiste na divisão do *dataset* em blocos e cada tarefa processa um bloco. É dividido o número de entidades pelo número de tarefas e é esse o tamanho do bloco, caso o resto da divisão não seja resto zero é adicionado mais um a cada bloco. A função da tarefa recebe sempre por argumento um índice identificador de tarefa e o tamanho do bloco. No ciclo na tarefa é verificado também se não ultrapassa o número de entidades, pois como pode ter sido adicionado mais um a cada bloco, a soma dos blocos pode ser maior que a do *dataset*.

```
// Definir número de entidades que cada thread vai processar (ceil)
int blockSize = ( nLines / nThreads) + ((nLines % nThreads) != 0);

// Ciclo for que cada thread percorre
for (int i = index*blockSize; i < index*blockSize + blockSize && i < nLines; i++) {
        …
};
```

Figura 3. Excerto de código para divisão e processamento de blocos.

3. Resultados

Os dados utilizados para entrada foram quatro *datasets* com um milhão de entidades. O algoritmo foi aplicado a cada *dataset* com um número diferente de tarefas e um número diferente de processadores. Estas execuções tiveram o número de *clusters* definido a dezasseis e o valor de limite de convergência de 0.001. Cada execução é repetida três vezes e calculada a média dos tempos obtidos.
Na tabela 3.1 pode-se ver os tempos de execução numa máquina com apenas 1 processador e sem recorrer a multitarefa (programa sequencial).

Tabela 3.1: Resultados para 1 processador

Processadores	PThreads	Entidades	Dimensões	clusters	Iterações	Tempo obtido (segundos)	Tempo (média)
1	0	1000000	1	16	2	3 5 4	4
1	0	1000000	2	16	20	19 19 19	19
1	0	1000000	3	16	85	77 83 81	80
1	0	1000000	4	16	59	57 56 51	55

Na tabela 3.2 estão os resultados de execução numa máquina com 2 processadores. Verifica-se que numa máquina com 2 processadores se não se recorrer a tarefas obtêm-se tempos de execução idênticos aos de numa máquina com apenas 1 processador. Com a utilização de duas tarefas a diferença dos tempos de execução vai aumentando consoante o número de dimensões do *dataset*. Por exemplo no *dataset* de apenas uma dimensão não se verifica redução no tempo de execução, mas no *dataset* de quatro dimensões nota-se uma redução no tempo de execução.

Tabela 3.2: Resultados para 2 processadores

Processadores	PThreads	Entidades	Dimensões	clusters	Iterações	Tempo obtido (segundos)	Tempo (média)
2	0	1000000	1	16	2	4 4 3	4
2	2	1000000	1	16	2	4 4 4	4
2	0	1000000	2	16	20	19 17 18	18
2	2	1000000	2	16	20	12 11 13	12
2	0	1000000	3	46	85	81 79 75	78
2	2	1000000	3	16	85	45 47 47	46
2	0	1000000	4	16	43	57 53 58	56
2	2	1000000	4	16	43	34 35 33	34

Os resultados da execução numa máquina com quatro processadores estão registados na tabela 3.3. Verifica-se novamente que apenas por ter mais processadores não há redução em tempos de execução, ou seja, os registos com zero ou duas tarefas têm os tempos idênticos aos obtidos anteriormente e registados nas tabelas anteriores. Com o uso de quatro tarefas, não se verifica melhoria com o *dataset* de uma dimensão, mas com os restantes *datasets* há uma redução de tempo de execução. Comparando a utilização de quatro tarefas com a implementação sequencial (zero tarefas) verifica-se a maior redução de tempo de execução, com o *dataset* de 3 dimensões. Esta redução é de 65%. Comparando a utilização de quatro tarefas com a utilização de duas tarefas, a diferença maior ocorre com o *dataset* de quatro dimensões onde se obtém uma redução de 38% no tempo de execução.

Tabela 3.3: Resultados para 4 processadores

Processadores	Threads	Entidades	Dimensões	clusters	Iterações	Tempos obtidos (segundos)	Tempo (média)
4	0	1000000	1	16	2	4 4 4	4
4	2	1000000	1	16	2	4 4 4	4
4	4	1000000	1	16	2	5 4 6	5
4	0	1000000	2	16	20	20 18 20	19
4	2	1000000	2	16	20	12 12 13	12
4	4	1000000	2	16	20	11 10 10	10
4	0	1000000	3	16	85	85 85 84	85
4	2	1000000	3	16	85	47 47 47	47
4	4	1000000	3	16	85	31 28 32	30

Processadores	**Threads**	**Entidades**	**Dimensões**	**clusters**	**Iterações**	**Tempos obtidos (segundos)**	**Tempo (média)**
4	0	1000000	4	16	43	58 59 59	59
4	2	1000000	4	16	43	36 33 32	34
4	4	1000000	4	16	43	23 21 19	21

Com estes resultados e focando no caso com mais iterações vemos um ganho, em segundos, considerável. No caso do *dataset* de três dimensões, que é o que corre com mais iterações, verificamos que numa máquina de um processador, sem tarefas adicionais tem um tempo médio de execução de 80 segundos e numa máquina com quatro processadores com quatro tarefas o tempo medio é de 30 segundos. Vemos um ganho de 50 segundos que corresponde a um ganho de 62% de tempo. No caso de pequenos *datasets* que requerem poucas iterações o ganho é negativo, ou seja, neste caso perde-se tempo em utilizar multitarefa. Isto verifica-se com o *dataset* de apenas uma dimensão, que corre com apenas duas iterações e o tempo médio utilizando quatro tarefas é superior ao tempo médio sem tarefas adicionais.

Tabela 4: Ganhos com a utilização de multitarefa

Pthreads	Iterações	Ganho segundos	Ganho percentagem
2	80	34	42.5%
4	80	50	62.5%
2	2	0	0%
4	2	-1	-1.25%

É também bastante interessante a análise às execuções onde o foco é a quantidade de *clusters*, ou seja, são analisadas as vantagens de tarefas com a variante de número de *clusters* num ambiente com número de processadores fixo em quatro e utilizando sempre o *dataset* de quatro dimensões. Os resultados destas execuções estão registados na tabela 3.4 e deixam claro que quanto maior é o número de clusters pretendido maior é a vantagem da paralelização do algoritmo. O algoritmo foi aplicado para a obtenção de 2, 4, 8, 16 e 32 clusters e em todos os casos se verificou redução no tempo de execução por recorrer a multitarefa. Comparando a paralelização com quatro tarefas e a execução sequencial verifica-se uma redução de dois segundos no caso de dois *clusters*, uma redução de 66% mas num intervalo curto. Conforme o número de *clusters* vai aumentando o intervalo da diferença de tempo é maior e percebe-se uma melhoria crescente no desempenho do algoritmo paralelizado.

Tabela 3.4: Resultados para diferentes quantidades de clusters

Processadores	Threads	Entidades	Dimensões	clusters	Iterações	Tempos obtidos (segundos)	Tempo (média)
4	0	1000000	4	2	6	3 2 3	3
4	2	1000000	4	2	6	2 2 1	2
4	4	1000000	4	2	7	1 1 1	1
4	0	1000000	4	4	30	11 11 11	11
4	2	1000000	4	4	30	7 5 5	6
4	4	1000000	4	4	30	5 4 6	5
4	0	1000000	4	8	42	29 29 29	29
4	2	1000000	4	8	42	16 20 19	18
4	4	1000000	4	8	42	13 13 13	13
4	0	1000000	4	16	43	59 58 59	59
4	2	1000000	4	16	43	33 35 34	34
4	4	1000000	4	16	43	21 21 22	21
4	0	1000000	4	32	100	252 252 258	254
4	2	1000000	4	32	100	137 140 143	140
4	4	1000000	4	32	100	82 79 71	77

A referida crescente melhoria de desempenho é percetível numa forma clara no gráfico da figura 4.

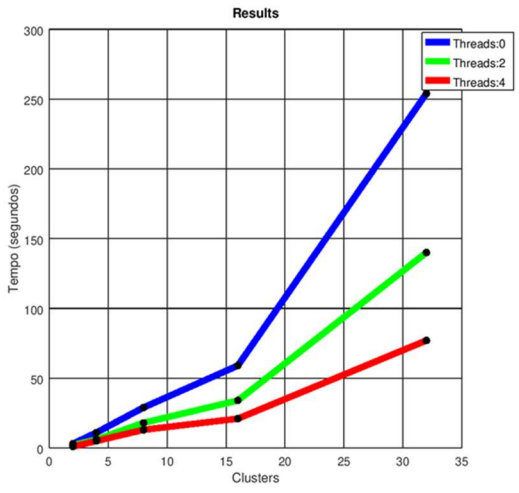

Figura 4. Tempos de cálculo variando o número de *clusters* e de tarefas.

Uma nota em relação ao limite de convergência, este valor é fundamental para a precisão do algoritmo. Quanto mais perto de zero mais precisa será a saída do algoritmo. Para verificar a relevância do limite de convergência na precisão do resultado foi executado o algoritmo num ambiente com quatro processadores, quatro tarefas e dezasseis clusters, variando apenas o limite de convergência entre os valores 0.001, 0.0001 e 0. Neste caso foi utilizado o *dataset* de quatro dimensões com distribuição uniforme. A precisão dos resultados é percetível nos gráficos da figura 5.

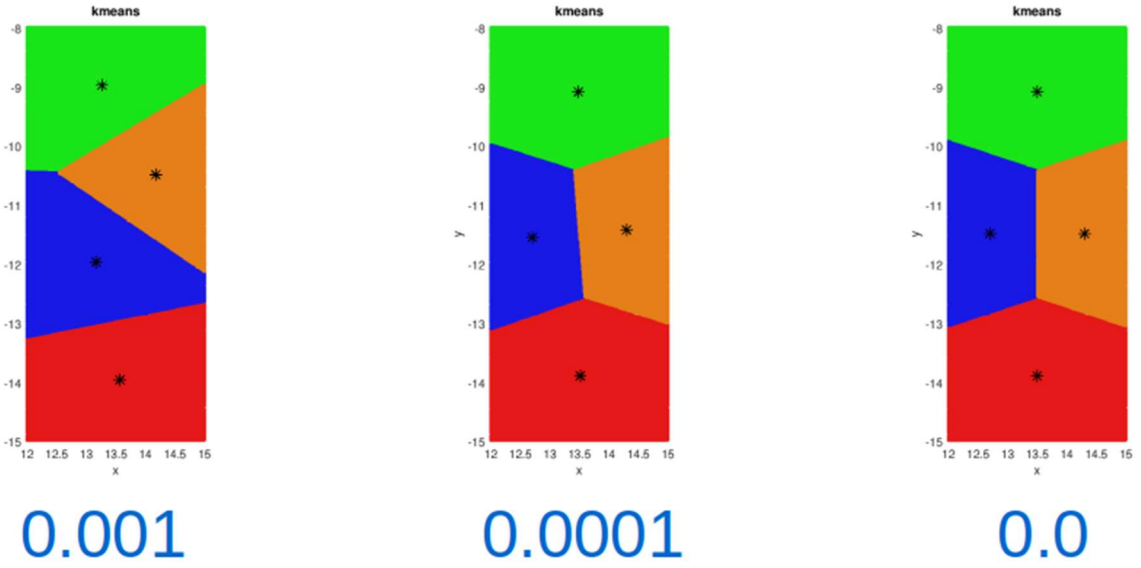

Figura 5. Precisão dos resultados variando o limite de convergência.

4. Conclusões

A paralelização do algoritmo mostra-se vantajosa em todos os casos, e quanto mais complexo é o *dataset* (maior número de entidades e/ou atributos) ou maior o número de *clusters*, mais percetível é a vantagem da paralelização do algoritmo K-means recorrendo a multitarefa.

Comparando a utilização de quatro tarefas em quatro processadores com a implementação sequencial, dos casos estudados verifica-se que o maior ganho de desempenho ocorre com o *dataset* de 3 dimensões, obtendo-se uma redução de 65% no tempo de execução, ou seja, o tempo de execução paralelo é cerca de um terço do tempo sequencial.

REFERÊNCIAS

Damas L. (2007), Linguagem C, LTC.

Katsavounidis I., Kou J., Zhang Z. (1994), A new initialization technique for generalized Lloyd iteration, IEEE Signal Processing Letters, 1(10), pp.144-146.

Kernighan B.W., Ritchie D.M. (1988), The C Programming Language, Prentice Hall.

Marques J. A., Ferreira P., Ribeiro C., Veiga L., Rodrigues R. (2009), Sistemas Operativos, FCA.

Zaki M.J., Meira Jr. W. (2014), Data Mining and Analysis, Cambridge University Press.

Marco Martins, profissionalmente desempenha tarefas de programador nas linguagens PHP, JavaScript, Java e Python, administrador de sistemas linux e administrador de bases de dados Postgresql. Licenciado em Engenharia Informática em 2020 pela Universidade Aberta de Portugal. Tem como áreas de interesse, sistemas de informação geográfica.

Paulo Shirley, Professor Auxiliar no Departamento de Ciências e Tecnologia (DCeT), Secção de Informática, Física e Tecnologia (SIFT). Licenciado em Engenharia Electrotécnica e de Computadores em 1988 pelo IST-UTL. Obteve os graus de Mestre (perfil de Controlo e Robótica) e de Doutor em Eng. Electrotécnica e de Computadores, pelo IST-UTL em 1993 e 2003 respetivamente. Tem como áreas de interesse, a intersecção da Informática (Computer Science) com a área do Controlo Automático, nomeadamente a área da "Computação de Alto Desempenho" aplicada a problemas de otimização.