



UNIVERSIDADE ESTADUAL PAULISTA  
"JÚLIO DE MESQUITA FILHO"  
Campus de São José do Rio Preto



Hethini do Nascimento Ribeiro

## **Paralelização do Algoritmo DIANA com OpenMP e MPI**

São José do Rio Preto  
2018

Hethini do Nascimento Ribeiro

## **Paralelização do Algoritmo DIANA com OpenMP e MPI**

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Campus de São José do Rio Preto.

Financiadora: CAPES

Orientadora: Prof<sup>ª</sup>. Dr<sup>ª</sup>. Roberta Spolon

São José do Rio Preto

2018

Ribeiro, Hethini do Nascimento.

Paralelização do algoritmo DIANA com OpenMP e MPI /  
Hethini do Nascimento Ribeiro. -- São José do Rio Preto, 2018  
66f. : il., tabs.

Orientador: Roberta Spolon

Dissertação (mestrado) – Universidade Estadual Paulista  
(UNESP), Instituto de Biociências, Letras e Ciências Exatas, São  
José do Rio Preto

1. Ciência da computação. 2. Aprendizado por computador. 3.  
Mineração de dados (Computação) 4. Algoritmos paralelos. I.  
Título.

CDU – 518.72

Ficha catalográfica elaborada pela Biblioteca do IBILCE  
UNESP - Câmpus de São José do Rio Preto

Hethini do Nascimento Ribeiro

## **Paralelização do Algoritmo DIANA com OpenMP e MPI**

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Campus de São José do Rio Preto.

Financiadora: CAPES

### **Comissão Examinadora**

---

Profa. Associada Roberta Spolon (Orientadora)  
Universidade Estadual Paulista “Júlio de Mesquita Filho”

---

Prof. Dr. Kelton Augusto Pontara da Costa  
Humber Institute of Technology & Advanced Learning

---

Prof. Dr. Anderson Francisco Talon  
Universidade Estadual Paulista “Júlio de Mesquita Filho”

Bauru  
31 de agosto de 2018

|

*Dedico este trabalho a toda minha família e amigos. Não poderia realizar essa etapa da minha vida sem o apoio e carinho destes.*

## **Agradecimentos**

Agradecimentos principais vão à minha mãe Marlene, meu pai Francisco e minha irmã Tâmizi por serem minha base fortalecedora em todos os momentos dessa jornada. Além, claro, do meu amigo, quase irmão, André Igüera pelo suporte proporcionado durante esse período.

Os agradecimentos especiais são direcionados a minha orientadora, Prof<sup>a</sup>. Dr<sup>a</sup>. Roberta Spolon, pela paciência e ensinamentos compartilhados. Gratidão aos colegas de mestrado, Vinicius Santos Andrade e Wellington Francisco da Silva, não apenas pela companhia durante o desenvolvimento deste trabalho, mas também pelo amparo técnico. Apreço dedicado ao grupo GSPD da Unesp de São José do Rio Preto pela confiança na concessão do *cluster* para a coleta dos testes desta pesquisa. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 1757857.

*“One child, one teacher, one book, one pen can  
change the world”*

— *Malala Yousafzai (2013)*

## Resumo

No início desta década havia cerca de 5 bilhões de telefones em uso gerando dados. Essa produção global aumentou aproximadamente 40% ao ano no início da década passada. Esses grandes conjuntos de dados que podem ser capturados, comunicados, agregados, armazenados e analisados, também chamados de *Big Data*, estão colocando desafios inevitáveis em muitas áreas e, em particular, no campo *Machine Learning*. Algoritmos de *Machine Learning* são capazes de extrair informações úteis desses grandes repositórios de dados e por este motivo está se tornando cada vez mais importante o seu estudo. Os programas aptos a realizarem essa tarefa podem ser chamados de algoritmos de classificação e clusterização. Essas aplicações são dispendiosas computacionalmente. Para citar alguns exemplos desse custo, o algoritmo *Quality Threshold Clustering* tem, no pior caso, complexidade  $O(n^5)$ . Os algoritmos hierárquicos AGNES e DIANA, por sua vez, possuem  $O(n^2)$  e  $O(2^n)$  respectivamente. Sendo assim, existe um grande desafio, que consiste em processar grandes quantidades de dados em um período de tempo realista, encorajando o desenvolvimento de algoritmos paralelos que se adequam ao volume de dados. O objetivo deste trabalho é apresentar a paralelização do algoritmo de hierárquico divisivo DIANA. O desenvolvimento do algoritmo foi realizado em MPI e OpenMP, chegando a ser três vezes mais rápido que a versão monoprocessada, evidenciando que embora em ambientes de memória distribuídas necessite de sincronização e troca de mensagens, para um certo grau de paralelismo é vantajosa a aplicação desse tipo de otimização para esse algoritmo.

**Palavras-chave:** Paralelismo, Algoritmos de Clusterização, Aprendizado de Máquina, Mineração de Dados, DIANA, OpenMP, MPI

## **Abstract**

Earlier in this decade there were about 5 billion phones in use generating data. This global production increased approximately 40% per year at the beginning of the last decade. These large datasets that can be captured, communicated, aggregated, stored and analyzed, also called Big Data, are posing inevitable challenges in many areas, and in particular in the Machine Learning field. Machine Learning algorithms are able to extract useful information from these large data repositories and for this reason their study is becoming increasingly important. The programs that can perform this task can be called classification and clustering algorithms. These applications are computationally expensive. To cite some examples of this cost, the Quality Threshold Clustering algorithm has, in the worst case, complexity  $O(n^5)$ . The hierarchical algorithms AGNES and DIANA, in turn, have  $O(n^2)$  and  $O(2^n)$  respectively. Thus, there is a great challenge, which is to process large amounts of data in a realistic period of time, encouraging the development of parallel algorithms that fit the volume of data. The objective of this work is to present the parallelization of the DIANA divisive hierarchical algorithm. The development of the algorithm was performed in MPI and OpenMP, reaching three times faster than the monoprocessed version, evidencing that although in distributed memory environments need synchronization and exchange of messages, for a certain degree of parallelism it is advantageous to apply this type of optimization for this algorithm.

**Keywords:** Parallelism, Clustering Algorithms, Machine Learning, Data Mining, DIANA, OpenMP, MPI

## Lista de Ilustrações

|                                                                                                                                                                                   |    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| <b>Figura 1.</b> Razões para desenvolver implementações de aprendizado de máquina de alto rendimento. ....                                                                        | 15 |
| <b>Figura 2.</b> Partições realizadas separando todos os 20 objetos em somente um grupo. ....                                                                                     | 20 |
| <b>Figura 3.</b> Partições suaves separando os objetos, podendo existir um elemento que pertença a mais de um grupo. ....                                                         | 21 |
| <b>Figura 4.</b> Fusões feitas pelo método hierárquico AGNES .....                                                                                                                | 21 |
| <b>Figura 5.</b> Divisões dos <i>clusters</i> pelo método hierárquico DIANA.....                                                                                                  | 22 |
| <b>Figura 6.</b> Classificação Flynn-Johnson .....                                                                                                                                | 26 |
| <b>Figura 7. a)</b> Representação de um sistema com memória compartilhada;                                                                                                        |    |
| <b>b)</b> Esquema de um sistema com memória distribuída. ....                                                                                                                     | 27 |
| <b>Figura 8.</b> Comparação entre a CPU e GPU. ....                                                                                                                               | 28 |
| <b>Figura 9.</b> Um dado sendo processado iterativamente por diferentes <i>threads</i> .....                                                                                      | 29 |
| <b>Figura 10. a)</b> Fase de expansão ou envio de dados                                                                                                                           |    |
| <b>b)</b> Fase de recebimento de dados ou contração .....                                                                                                                         | 30 |
| <b>Figura 11.</b> Representação do modelo <i>Bag-of-Tasks</i> em que os dados a serem processados ficam concentrados em um repositório.....                                       | 30 |
| <b>Figura 12.</b> Speedup de acordo com a lei de Amdahl. ....                                                                                                                     | 32 |
| <b>Figura 13.</b> Comportamento do <i>speedup</i> em função de R e N.....                                                                                                         | 35 |
| <b>Figura 14.</b> Comparação do tempo de execução do algoritmo sequencial com o paralelo.....                                                                                     | 39 |
| <b>Figura 15.</b> Exemplo de combinação de pontos, gerando pares únicos com suas respectivas distâncias. ....                                                                     | 42 |
| <b>Figura 16.</b> Esquematização geral da distribuição de dados entre os nós do <i>cluster</i> .....                                                                              | 44 |
| <b>Figura 17.</b> Esquematização geral da distribuição de dados entre os threads locais, eleição do melhor resultado local e então a determinação do melhor resultado global..... | 45 |
| <b>Figura 18.</b> Laço de repetição para percorrer as linhas da matriz de distâncias. ....                                                                                        | 45 |
| <b>Figura 19. a)</b> A variável “pacote” antes da redução contém a distância de cada nó além do índice identificador deste.....                                                   | 47 |
| <b>b)</b> Após a redução da variável “pacote”, a maior distância global é encontrada e a variável é atualizada em todos os nós. ....                                              | 47 |
| <b>Figura 20.</b> “Empacotamento” da distância local de cada nó com o índice deste, além da redução da variável “pacote” na variável “ <i>melhor_eleito</i> ”.....                | 47 |
| <b>Figura 21.</b> Esquematização do reagrupamento do par de pontos seis e três após a redução em MPI. ....                                                                        | 48 |
| <b>Figura 22.</b> Gráfico do <i>speedup</i> previsto segundo a .....                                                                                                              | 52 |
| <b>Figura 23.</b> Gráfico do <i>speedup</i> esperado para a Lei de Gustafson com o aumento de núcleos. ....                                                                       | 54 |
| <b>Figura 24.</b> Comparativo do tempo em segundos com a quantidade de <i>threads</i> .....                                                                                       | 55 |
| <b>Figura 25.</b> Comparativo do <i>speedup</i> com a quantidade de <i>threads</i> .....                                                                                          | 55 |
| <b>Figura 26.</b> Comparativo do <i>speedup</i> medido e previsto. ....                                                                                                           | 56 |

## Lista de Quadros

|                                                                                                                |    |
|----------------------------------------------------------------------------------------------------------------|----|
| <b>Quadro 1.</b> Um resumo dos quatro tipos de medida mais comumente usados em agrupamentos hierárquicos. .... | 24 |
| <b>Quadro 2.</b> <i>Speed-up</i> ganho com OpenMP, MPI e CUDA no algoritmo K-means.....                        | 38 |

## Lista de Tabelas

|                                                                                                         |    |
|---------------------------------------------------------------------------------------------------------|----|
| <b>Tabela 1.</b> Alguns algoritmos que usam aprendizado supervisionado e suas descrições. ....          | 18 |
| <b>Tabela 2.</b> Alguns algoritmos que usam aprendizado não-supervisionado e suas descrições. .         | 19 |
| <b>Tabela 3.</b> Complexidade de alguns algoritmos de clusterização. ....                               | 37 |
| <b>Tabela 4.</b> Um exemplo de uma matriz de distância utilizada em algoritmos hierárquicos. ....       | 41 |
| <b>Tabela 5.</b> Matriz de distâncias simétrica.....                                                    | 41 |
| <b>Tabela 6.</b> Determinação do conjunto de dados e sua distribuição. ....                             | 43 |
| <b>Tabela 7.</b> Maior distância de cada região encontrada por cada thread. ....                        | 46 |
| <b>Tabela 8.</b> Maior distância global encontrada. ....                                                | 48 |
| <b>Tabela 9.</b> Determinação da porcentagem sequencial e paralela do programa monoprocessado<br>.....  | 51 |
| <b>Tabela 10.</b> Ganho de desempenho esperado de acordo com a Lei de Amdahl .....                      | 52 |
| <b>Tabela 11.</b> Ganho de desempenho esperado de acordo com a Lei de Gustafson .....                   | 53 |
| <b>Tabela 12.</b> Tempo em segundos para classificar 11826 pontos para o aumento de <i>threads</i> . .. | 54 |

## Lista de Abreviaturas e Siglas

|              |                                            |
|--------------|--------------------------------------------|
| <b>AGNES</b> | <i>Agglomerative Nesting</i>               |
| <b>ALU</b>   | <i>Arithmetic Logic Unit</i>               |
| <b>CUDA</b>  | <i>Compute Unified Device Architecture</i> |
| <b>DIANA</b> | <i>Divisive Analysis</i>                   |
| <b>DM</b>    | <i>Data Mining</i>                         |
| <b>E/S</b>   | <i>Entrada e Saída</i>                     |
| <b>GPU</b>   | <i>Graphics Processing Unit</i>            |
| <b>KNN</b>   | <i>K-nearest neighbor</i>                  |
| <b>MIMD</b>  | <i>Multiple Instruction Multiple Data</i>  |
| <b>MISD</b>  | <i>Multiple Instruction Single Data</i>    |
| <b>ML</b>    | <i>Machine Learning</i>                    |
| <b>MPI</b>   | <i>Message Passing Interface</i>           |
| <b>QT</b>    | <i>Quality Threshold</i>                   |
| <b>SIMD</b>  | <i>Single Instruction Multiple Data</i>    |
| <b>SISD</b>  | <i>Single Instruction Single Data</i>      |
| <b>SOM</b>   | <i>Self Organizing Maps.</i>               |
| <b>SVM</b>   | <i>Support Vector Machines</i>             |

## Sumário

|                                                                     |    |
|---------------------------------------------------------------------|----|
| Capítulo 1 - Introdução .....                                       | 14 |
| 1 Considerações iniciais .....                                      | 14 |
| 1.1 Motivação e justificativa .....                                 | 14 |
| 1.2 Objetivo .....                                                  | 16 |
| 1.2.1 Objetivo gerais .....                                         | 16 |
| 1.2.2 Objetivo específicos .....                                    | 16 |
| 1.3 Estrutura do trabalho .....                                     | 16 |
| Capítulo 2 - Fundamentação teórica.....                             | 17 |
| 2.1 Considerações iniciais .....                                    | 17 |
| 2.2 Algoritmos de aprendizado de máquina e <i>Data Mining</i> ..... | 17 |
| 2.2.1 Algoritmos supervisionados.....                               | 17 |
| 2.2.2 Algoritmos não-supervisionados .....                          | 19 |
| 2.3 Paralelização .....                                             | 24 |
| 2.3.1 Processamento paralelo.....                                   | 25 |
| 2.3.2 Classificação de sistemas paralelos .....                     | 25 |
| 2.3.3 Programação paralela .....                                    | 27 |
| 2.3.4 Principais Modelos de Programação Paralela .....              | 27 |
| 2.3.5 Modelos de particionamento.....                               | 29 |
| 2.3.6 Análise e Otimização de Desempenho .....                      | 30 |
| 2.3.7 Otimização de código pelo programador .....                   | 36 |
| 2.3.8 Otimização de código pelo compilador .....                    | 36 |
| 2.4 Paralelização dos algoritmos de aprendizado de máquina.....     | 37 |
| 2.5 Considerações finais .....                                      | 39 |
| Capítulo 3 - Materiais e métodos .....                              | 40 |
| 3.1 Considerações iniciais .....                                    | 40 |
| 3.2 Algoritmo DIANA .....                                           | 40 |
| 3.3 O projeto.....                                                  | 41 |
| 3.3.1 Desenvolvimento com MPI .....                                 | 43 |
| 3.3.2 Desenvolvimento com OpenMP .....                              | 44 |
| 3.4 Considerações finais .....                                      | 50 |
| Capítulo 4 - Resultados .....                                       | 51 |
| 4.1 Considerações iniciais .....                                    | 51 |
| 4.2 Resultados experimentais .....                                  | 51 |
| 4.3 Considerações finais .....                                      | 56 |
| Capítulo 5 - Conclusões e trabalhos futuros .....                   | 58 |
| 5.1 Conclusões .....                                                | 58 |
| 5.2 Contribuições .....                                             | 58 |
| 5.3 Trabalhos futuros .....                                         | 59 |
| Referências Bibliográficas .....                                    | 60 |
| Bibliografia.....                                                   | 62 |

|                                                                                                                                  |           |
|----------------------------------------------------------------------------------------------------------------------------------|-----------|
| <b>APÊNDICE A – Relação de artigos encontrados sobre paralelização de algoritmos de clusterização usando MPI ou OpenMP .....</b> | <b>63</b> |
| <b>ANEXO A – Localizações dos usuários do aplicativo MOPSI na Finlândia até 2012 .....</b>                                       | <b>65</b> |

# Capítulo 1 - Introdução

## 1 Considerações iniciais

Devido à diversidade e grande quantidade de dados produzida no mundo, fica praticamente impossível analisá-los. Existe uma grande pressão para entender, sintetizar e tomar decisões em praticamente todos os tipos de dados (BELL, 2015). O grande desafio é transformá-los em informação aproveitável como, por exemplo, para empresas de marketing ou vendas que procuram oferecer produtos a certos grupos de pessoas que apresentam um perfil mais adequado a uma determinada mercadoria ou produto, aumentando assim o sucesso das vendas e mantendo a lealdade do cliente.

Para fazer a análise dessa extensa quantidade de dados e transformá-los em informações proveitosas, utilizam-se algoritmos de agrupamento capazes de identificar padrões e categorizar os elementos analisados em grupos ou classes. Infelizmente realizar essa tarefa não é algo simples devido ao seu alto custo computacional e, uma maneira de alcançar a otimização desses algoritmos é através de técnicas de paralelismo.

### 1.1 Motivação e justificativa

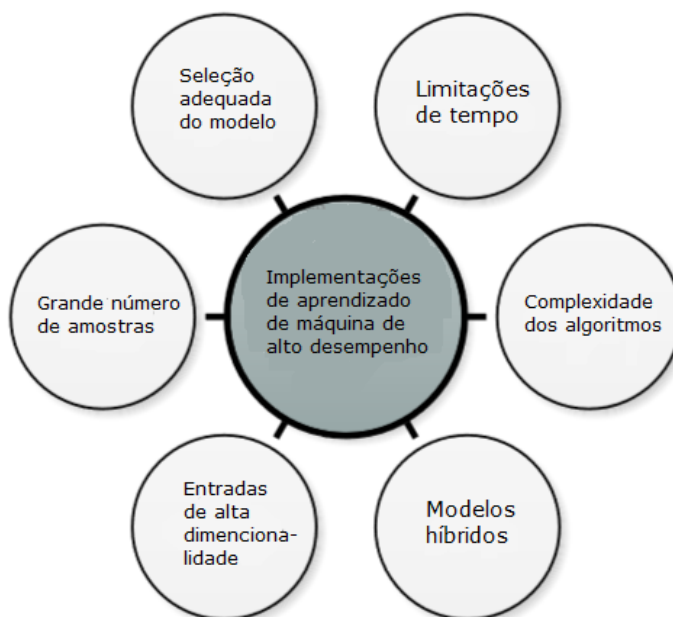
Manyika et al. (2011) definiu Big Data como “[...] conjuntos de dados cujo tamanho está além da capacidade que ferramentas típicas de banco de dados conseguem processar”. Esse é um problema característico em *Machine Learning* (ML).

Ao pensar em motivações para paralelizar algoritmos de aprendizado de máquina, naturalmente, as principais razões referem-se à complexidade computacional dos algoritmos de ML e a necessidade de explorar grandes conjuntos de dados abrangendo um grande número de amostras e/ou características. Além disso, o processo de treinamento dos algoritmos supervisionados é demorado, mesmo com bases de dados pequenas, uma vez que essa atividade pode precisar de várias repetições (LOPES; RIBEIRO, 2015).

Outro exemplo são os problemas complexos de aprendizado de máquina. Estes podem ser resolvidos combinando mais de um algoritmo, resultando numa solução híbrida e muitas vezes com maior custo computacional. Outra razão é a limitação do tempo, seja para desenvolver as aplicações ou na obtenção dos resultados (LOPES; RIBEIRO, 2015).

A Figura 1 apresenta uma esquematização com as razões principais para o desenvolvimento de aplicações de alto desempenho em ML.

**Figura 1.** Razões para desenvolver implementações de aprendizado de máquina de alto rendimento.



**Fonte:** LOPES; RIBEIRO, 2015, p. 6 (adaptado pela autora)

Existem poucas pesquisas feitas nos últimos quatro anos em relação a paralelização dos algoritmos de clusterização (Apêndice A). Embora alguns sejam mais estudados, outros são menos explorados devido à complexidade destes. Conforme Kaufman e Rousseeuw (1990), a alta complexidade de alguns algoritmos implica na exclusão deles em alguns estudos como, por exemplo, o algoritmo DIANA (*divisive analysis*) que possui complexidade  $O(2^n)$ .

Já existem estudos relacionando a paralelização com algoritmos de classificação e agrupamento, como por exemplo, o trabalho feito por Danalis, McCurdy e Vetter (2012) que paralelizaram o algoritmo *Quality Threshold Clustering* com OpenMP e MPI e Bhimani, Leiser e Mi (2015) em que os autores usaram três abordagens:

- Memória compartilhada usando OpenMP;
- Memória distribuída com a passagem de mensagens (MPI); e
- Computação heterogênea com Unidades de Processamento Gráfico (GPUs) NVIDIA programado com CUDA-C.

Os resultados desse trabalho mostraram que as três abordagens foram, em média, 35 vezes mais rápidas que a versão sequencial do algoritmo.

É vantajoso investigar a otimização de algoritmos de aprendizado de máquina através da paralelização não só pelos motivos abordados por LOPES e RIBEIRO (2015) (Figura 1), mas também pelo baixo desenvolvimento acadêmico na área (Apêndice A).

## 1.2 Objetivo

### 1.2.1 Objetivo gerais

Conforme o levantamento de dados apresentado no Apêndice A, há uma baixa exploração acadêmica dos algoritmos de clusterização com paralelismo. Para os algoritmos estudados, os hierárquicos não apresentaram trabalhos envolvendo OpenMP ou MPI. Estes são deixados de lado na literatura devido a sua alta complexidade, especialmente o algoritmo DIANA que possui  $O(2^n)$  (KAUFMAN; ROUSSEEUW, 1990), resultando na associação dos algoritmos hierárquicos com o AGNES, uma vez que este é mais utilizado por sua complexidade de tempo ser  $O(n^2 \log n)$  (JAIN; MURTY; FLYNN, 1999) ou, dependendo da implementação,  $O(n^2)$  (KAUFMAN; ROUSSEEUW, 1990).

Dentro deste contexto, este trabalho tem como objetivos gerais a otimização do algoritmo hierárquico de aprendizado de máquina DIANA através do desenvolvimento de uma versão híbrida com OpenMP e MPI, buscando acelerar o processo de classificação de amostras em bases de dados grandes e mantendo a acurácia do resultado.

### 1.2.2 Objetivo específicos

Com intuito de alcançar e verificar o escopo do trabalho, este foi subdividido em objetivos específicos necessários para que se possa atingi-lo:

- Otimizar o código do algoritmo DIANA procurando ganhar desempenho com o paralelismo; e,
- Paralelizar o algoritmo otimizado com OpenMP e MPI e comparar seu ganho de desempenho com a versão monoprocessada do mesmo.

## 1.3 Estrutura do trabalho

O próximo capítulo refere-se aos conceitos teóricos importantes para a fundamentação e o entendimento dos algoritmos de ML (seção 2.2) e paralelismo (seção 2.3). A seção 2.4 relaciona processamento concorrente para algoritmos de ML. Os métodos e as abordagens utilizadas para o desenvolvimento do projeto são descritos no capítulo 3. No capítulo 4 são apresentados e discutidos os resultados e no capítulo 5 são apresentadas as conclusões e trabalhos futuros.

## Capítulo 2 - Fundamentação teórica

### 2.1 Considerações iniciais

Conseguir extrair uma informação coerente de dados é, além de uma tarefa complexa, as vezes impossível, principalmente quando a quantidade de dados analisado for grande. Os algoritmos de classificação e clusterização auxiliam nessa atividade identificando padrões nos objetos estudados, e transformando os dados sem sentido em informações úteis.

### 2.2 Algoritmos de aprendizado de máquina e *Data Mining*

Algoritmos de classificação e clusterização são muito utilizados em *Data Mining* (DM) e ML. Eles identificam padrões utilizando alguns critérios de semelhança entre os objetos analisados e então determinam a qual grupo ou classe este pertence.

DM, ou mineração de dados, pode ser definida como o “processo de descobrir padrões em dados. O processo deve ser automático ou semiautomático. O padrão descoberto deve ser significativo na medida em que levar a alguma vantagem, geralmente econômica” (WITTEN; FRANK, 2005, p. 5, tradução nossa). A maioria das técnicas que identificam padrões pertencem a um campo conhecido como *Machine Learning*.

ML, ou aprendizado de máquina, é um ramo da inteligência artificial. Usando computação, podemos projetar sistemas que podem aprender ou extrair informações a partir de dados através de um treinamento. Os sistemas podem aprender e melhorar com a experiência, e com o tempo, refinar um modelo que pode ser usado para prever os resultados de perguntas com base em experiências anteriores (BELL, 2015, p. 3, tradução nossa).

Dentro desse contexto, o aprendizado pode ser dividido em dois tipos: Supervisionado e Não-Supervisionado.

#### 2.2.1 Algoritmos supervisionados

Algoritmos de aprendizado de máquina supervisionados são aqueles em que as classificações são pré-definidas, ou seja, para cada objeto  $X_i$  onde  $i = \{i \in \mathbb{N} \mid 0 < i \leq N\}$  existe uma categoria pré-definida  $Y_j$  em que  $j = \{j \in \mathbb{N} \mid 0 < j \leq M\}$ , que pode ser associada àquele objeto (JAMES et al., 2013). O objetivo de uma tarefa de classificação é agrupar corretamente uma coleção de elementos na categoria adequada (PUSTEJOVSKY; STUBBS, 2012).

Os algoritmos *K-nearest neighbor* (KNN), *Naïve Bayes* (PUSTEJOVSKY; STUBBS, 2012), *Support Vector Machine* (SVM) (LOPES; RIBEIRO, 2015), entre outros, são alguns exemplos de algoritmos de classificação (supervisionados). Uma breve descrição destes e outros algoritmos (*Bagging*, *Boosting* e *Perceptron*) é apresentada na Tabela 1.

**Tabela 1.** Alguns algoritmos que usam aprendizado supervisionado e suas descrições.

| Nome Algoritmo     | Descrição                                                                                                                                                                                                                                         | Fórmula                                                                                                                                       |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Bagging</b>     | Baseia-se na criação de <i>bootstraps</i> = amostras diferentes da base de dados que são usadas para aprender hipóteses diferentes.                                                                                                               | $\hat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^B \hat{f}^{*b}(x).$                                                                         |
| <b>Boosting</b>    | No <i>bootstrap</i> , $m$ pontos de dados são selecionados aleatoriamente com substituições do conjunto de dados $X$ . Repetindo este processo de amostragem independentemente $B$ vezes, obtemos $B$ conjuntos de amostras de <i>Bootstrap</i> . | $\hat{\theta}^B = \sum_{j=1}^B \hat{\theta}_j.$                                                                                               |
| <b>KNN</b>         | Consiste em classificar um novo objeto baseado nos exemplos do treinamento e em seus atributos                                                                                                                                                    | $d(X_i, Y_i) = \sqrt{\sum_{i=1}^n  X_i - Y_i ^r}$                                                                                             |
| <b>Naive Bayes</b> | Baseado na teoria probabilística de Bayes de prever a classe de um dado.                                                                                                                                                                          | $\Pr(C = c_j   d).$<br>$c = \arg \max_{c_j} \Pr(C = c_j) \prod_{i=1}^{ A } \Pr(A_i = a_i   C = c_j)$                                          |
| <b>Perceptron</b>  | Perceptron toma um vetor de entradas, calcula uma combinação linear dessas e, em seguida, resulta 1 se o resultado for maior que algum valor de entrada ou -1 caso contrário.                                                                     | $o(\vec{x}) = \text{sgn}(\vec{w} \cdot \vec{x})$<br>$\text{sgn}(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise} \end{cases}$ |
| <b>SVM</b>         | Localiza um hiperplano $h$ que separa as amostras de treinamento                                                                                                                                                                                  | $W \cdot X + b = 0$                                                                                                                           |

**Fonte:** Elaborado pela autora

## 2.2.2 Algoritmos não-supervisionados

A tarefa de classificar dados em categorias já conhecidas é simples, embora ainda dispendiosa computacionalmente. O grande desafio em aprendizado não-supervisionado é não ter as classes pré-definidas, ou seja, o trabalho de análise e separação dos objetos baseado em padrões acontece “às cegas” (JAMES et al., 2013). A Tabela 2 apresenta alguns dos algoritmos de clusterização e uma breve descrição de cada um.

**Tabela 2.** Alguns algoritmos que usam aprendizado não-supervisionado e suas descrições.

| Nome Algoritmo                                    | Descrição                                                                                                                                                    | Fórmula                                                                                                                                                                                |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AGNES                                             | Algoritmo hierárquico que procura agrupar os objetos/clusters através de suas similaridades                                                                  | $d([x, y], z) = \min [d(x, z), d(y, z)].$                                                                                                                                              |
| Clustering as a Mixture of Gaussians (EM)         | Cada cluster pode ser representado matematicamente por uma representação paramétrica (Gaussian/Poisson). O método se baseia na "junção" dessas distribuições | $\mathcal{L}(\mathbf{x}^u   \Phi) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}^u   \mu_k, \Sigma_k)$<br>$\Psi = \{ \{ \pi_k \}_{k=1}^K, \{ \mu_k \}_{k=1}^K, \{ \Sigma_k \}_{k=1}^K \}$ |
| DIANA                                             | Algoritmo hierárquico que procura dividir os objetos entre clusters através de suas dissimilaridades                                                         | $D_i = \frac{1}{n-1} \sum_{j=0, j \neq i}^{n-1} D(x_i, x_j)$                                                                                                                           |
| Fuzzy C-means                                     | Um mesmo objeto pode pertencer a mais de um cluster (grupo)                                                                                                  | $J = \sum_{i=1}^N \sum_{j=1}^C u_{ij}^m d^2(x_i, v_j)$                                                                                                                                 |
| K-Means                                           | Procura distribuir os elementos a um conjunto de k clusters distintos de maneira iterativa.                                                                  | $d(X_i, Y_i) = \sqrt{\sum_{i=1}^n  X_i - Y_i ^r}$                                                                                                                                      |
| QT (Quality Threshold) Clustering                 | O usuário define o tamanho máximo do cluster e novos objetos são adicionados até esse tamanho “estourar”                                                     | $d = \max_{i,j} \{ \sqrt{(x_i - x_j)^2} \}$                                                                                                                                            |
| Redes de Kohonen ou Redes Auto Organizáveis (SOM) | Sistemas que evoluem para uma forma organizada por conta própria                                                                                             | $i(x) = \arg \min_i \ x(n) - w_j\ ,$<br>$j = 1, 2, \dots, L.$<br>$w_j(n+1) = w_j(n) + \eta(n) h_{j,i(x)}(n) (x(n) - w_j(n)).$                                                          |

**Fonte:** Elaborado pela autora

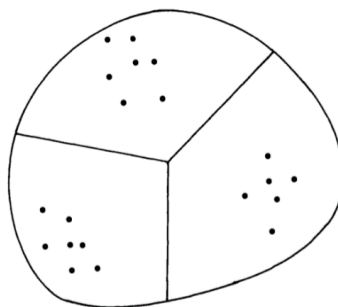
Também conhecidos como algoritmos de clusterização ou agrupamento, estes podem ser divididos em duas categorias: particionados e hierárquicos (XU; WUNSCH, 2009 apud COVÕES, 2015, p. 3). Estes são descritos nas seções seguintes.

Os algoritmos particionados separam os objetos em K grupos distintos, não aninhados e não sobrepostos. Essa partição pode ser *crisp* (rígida) ou soft (suave) (ANDERSON et al., 2010 apud COVÕES, 2015, p. 3).

No caso das partições rígidas os dados pertencem a um único grupo, ou seja, o objeto avaliado possui apenas dois estados: pertence ou não àquele grupo (1 ou 0). Um representante importante dessa categoria é algoritmo K-means que associa N objetos a um número K de grupos pré-determinado. Cada um dos clusters possui um centroide, um ponto de referência para o cálculo das distâncias entre os elementos e os clusters. Os grupos são definidos por um processo iterativo para calcular quais os objetos que estão mais próximos ao centroide. Esses passos se repetem até o ponto onde os objetos não se movem para diferentes centróides, determinando o fim do agrupamento por K-means (BELL, 2015, p. 164, tradução nossa).

A Figura 2 apresenta um esquema da divisão *crisp*. Todos os elementos analisados pertencem a um e somente um único grupo.

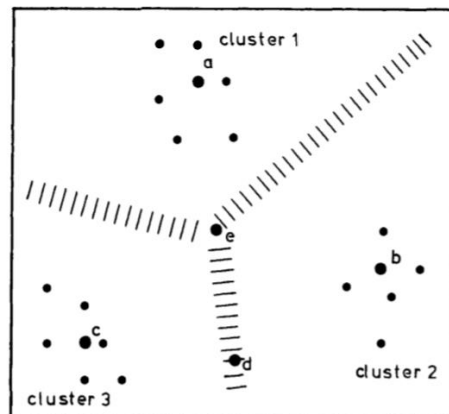
**Figura 2.** Partições realizadas separando todos os 20 objetos em somente um grupo.



**Fonte:** KAUFMAN; ROUSSEEUW, 1990, p. 39

Quando a divisão ocorre de uma maneira soft, a ideia binária da relação entre os dados e os grupos deixa de existir. Essa categoria de partições cria uma relação que varia no intervalo  $[0,1]$ , ou seja, um objeto pode não pertence mais inteiramente a apenas um único grupo (Figura 3).

**Figura 3.** Partições suaves separando os objetos, podendo existir um elemento que pertença a mais de um grupo.

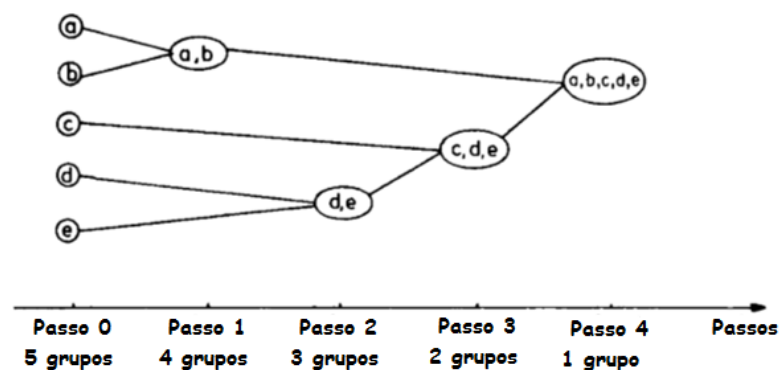


**Fonte:** KAUFMAN; ROUSSEEUW, 1990, p. 43

Os algoritmos do tipo hierárquicos geram várias divisões criando conjuntos e subconjuntos, deixando as partições aninhadas, ou seja, a “partição A pertence a partição B se e somente se A é um subconjunto de B.” (JAIN; DUBES et al., 1988 apud COVÕES, 2015, p. 3). A separação das partições pode ser feita por aglomeração (AGNES) ou por divisão (DIANA).

No método AGNES os objetos analisados começam separados. A cada iteração dois grupos/objetos são mesclados até que reste apenas o menor número possível de grupos com os elementos. A complexidade de tempo desse algoritmo é  $O(n^2 \log n)$  (JAIN; MURTY; FLYNN, 1999) ou, dependendo da implementação, pode ser  $O(n^2)$  (KAUFMAN; ROUSSEEUW, 1990). A Figura 4 demonstra passo a passo como a aglomeração no método AGNES acontece.

**Figura 4.** Fusões feitas pelo método hierárquico AGNES.

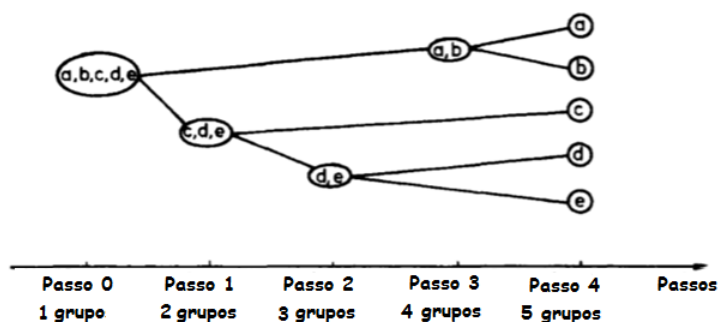


**Fonte:** KAUFMAN; ROUSSEEUW, 1990, p. 205 (adaptado pela autora).

Na Figura 4 observa-se que inicialmente todos os objetos estão sozinhos, ou seja, não foram classificados ainda (passo 0). Iterativamente as amostras são classificadas e agrupadas de acordo com os padrões encontrados, conforme passo 1 em que “a” e “b” foram colocados no mesmo cluster. O mesmo acontece no passo 2, em que “d” e “e” foram agrupados. A busca por padrões e agrupamentos dos elementos é feita até que todos objetos façam parte do mesmo cluster, ou seja, existe apenas um único grupo no sistema (passo 4).

Na direção oposta, iniciando o processo com todos os objetos em um grupo, o método por divisão a cada iteração faz uma segmentação do cluster até que existam N grupos ou apenas os elementos. Este algoritmo é pouco explorado devido a sua complexidade  $O(2^n)$  (KAUFMAN; ROUSSEEUW, 1990). Uma representação do funcionamento do algoritmo DIANA pode ser vista na figura 5.

**Figura 5.** Divisões dos *clusters* pelo método hierárquico DIANA.



**Fonte:** KAUFMAN; ROUSSEEUW, 1990, p. 257 (adaptado pela autora).

Ao contrário do ocorre com o AGNES, o DIANA no passo 0 (Figura 5) inicia com todos os elementos pertencendo a um grupo. A proposta é classificar os objetos em outros clusters a partir do reconhecimento de padrões, ou seja, dividir o grupo principal em subgrupos. Como pode ser observado nos passos 1, 2 e 3, novas classificações são feitas, dividindo os clusters em outros menores até o momento em que todos as amostras tenham classificações diferentes (passo 4).

A decisão de mesclar objetos (AGNES) ou de separá-los (DIANA) é feita baseando-se tanto na similaridade entre os objetos bem como em suas diferenças. O tipo de problema trabalhado determinará qual das duas medidas melhor se adapta.

Para buscar padrões entre as amostras analisadas, muitos fatores podem ser utilizados. Uma maneira muito comum é a distância Euclidiana, a qual é expressa pela Equação 1:

$$\sqrt{(Xa - Xb)^2 + (Ya - Yb)^2} \quad (1)$$

James et al. (2013) cita quatro maneiras principais do uso da Equação 1: *Complete Linkage*, *Single Linkage*, *Average Linkage* e *Centroid Linkage*. Estes utilizam a distância Euclidiana para determinar padrões entre os elementos, porém com algumas diferenças.

Em linhas gerais o *Complete Linkage* e *Single Linkage* aplicam a equação 1 para determinar os padrões entre as amostras, no entanto, o primeiro trabalha com as distâncias maiores (equação 2) enquanto o segundo trabalha com as menores (equação 3).

$$D(x, y)_z = \max(Dxz, Dyz) \quad (2)$$

$$D(x, y)_z = \min(Dxz, Dyz) \quad (3)$$

O *Average Linkage* após o cálculo do intervalo entre os pontos encontra a média dos resultados para então determinar as similaridades e diferenças entre os objetos. A equação 4 apresenta este cálculo, onde  $N_x$  e  $N_y$  são os números de elementos nos clusters X e Y respectivamente e  $D_{xz}$  e  $D_{yz}$  são as distâncias entre os objetos xz e yz, nesta ordem.

$$D(x, y)_z = \frac{(N_x \cdot D_{xz} + N_y \cdot D_{yz})}{N_x + N_y} \quad (4)$$

Já o *Centroid Linkage* procura um ponto central dos clusters para medir as distâncias. A equação 5 expressa este método, onde  $N_x$  e  $N_y$  são os números de elementos nos clusters X e Y, respectivamente e  $D_{xz}$  e  $D_{yz}$  são as distâncias entre os objetos xz e yz, nesta ordem.

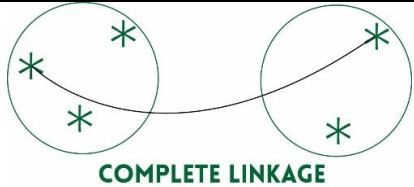
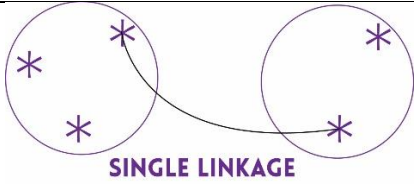
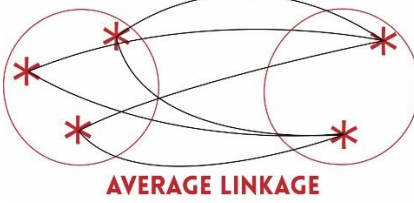
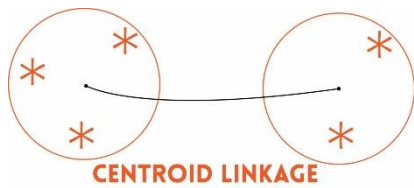
$$D(x, y)_z = \frac{(N_x \cdot D_{xz} + N_y \cdot D_{yz})}{N_x + N_y} - \frac{N_x \cdot N_y \cdot D_{xy}}{(N_x + N_y)^2} \quad (5)$$

O Complete, Single e Average Linkage são mais populares entre os problemas estatísticos. Os Average e Complete são geralmente escolhidos em relação ao Single, pois

tendem a produzir dendrogramas mais equilibrados. Já o Centroid Linkage é frequentemente usado em problemas que envolvem genôma, mas este sofre de uma grande desvantagem, em que uma inversão pode ocorrer, isto é, dois grupos estão fundidos em uma altura abaixo de cada um dos clusters individuais no dendrograma (JAMES ET AL, 2013).

O Quadro 1 apresenta uma breve descrição de cada uma das formas de medição e um esquema dos seus respectivos funcionamentos.

**Quadro 1.** Um resumo dos quatro tipos de medida mais comumente usados em agrupamentos hierárquicos.

| <i>Ligação</i>  | <i>Descrição</i>                                                                                                                                                                             | <i>Ilustração</i>                                                                    |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <i>Complete</i> | Calcula-se todas as dissimilaridades entre as amostras do cluster A e B e registre a maior dessas diferenças.                                                                                |    |
| <i>Single</i>   | Calcula-se todas as dissimilaridades entre as amostras do cluster A e B e registre a menor dessas diferenças.                                                                                |   |
| <i>Average</i>  | Calcula-se todas as dissimilaridades entre as amostras no cluster A e B e registra a média dessas desigualdades.                                                                             |  |
| <i>Centroid</i> | Dissimilaridade entre o centróide do cluster A (um vetor médio de comprimento p) e o centróide do cluster B. O <i>Centroid Linkage</i> pode resultar em inversões indesejáveis no resultado. |  |

**Fonte:** JAMES ET AL, 2013, p.394 (adaptado pela autora).

## 2.3 Paralelização

A ideia de um computador com um único processador está cada dia mais arcaica e a proposta de melhorar o desempenho de uma máquina usando um único processador é

impossível. Tal processador iria consumir uma quantidade de energia inaceitável. Uma saída para o problema é, além da adição de novos núcleos, é a paralelização de código (GEBALI, 2011).

Embora possam existir algumas dependências entre as variáveis, é possível acelerar a execução de um algoritmo desde que algumas subtarefas possam ser executadas simultaneamente, mantendo a coerência na execução e garantindo um resultado satisfatório (GEBALI, 2011).

### 2.3.1 Processamento paralelo

Em computação existem muitas definições para processamento paralelo. A definição de processamento paralelo adotada neste trabalho é a mesma escolhida por Parhami (2002, p. 13, tradução nossa): “[...] o termo processamento paralelo é usado no sentido restrito de ter múltiplos processadores (idênticos geralmente) para o cálculo principal e não para a Entrada/Saída (E/S) ou outras atividades periféricas.”.

### 2.3.2 Classificação de sistemas paralelos

Podem-se dividir os computadores paralelos em duas categorias: fluxo de controle e fluxo de dados. Os processadores que operam com o fluxo de controle são aqueles que se baseiam nos princípios dos computadores sequenciais ou Von Neumann. Já os processadores de fluxo de dados, ou não-von Neumann, seguem na direção contrária. Esses processadores superescalares analisam várias instruções para encontrar dependências de dados, e executar instruções em paralelo que não dependem umas das outras (EIJKHOUT, 2015).

Em 1966, M. J. Flynn criou uma taxonomia que caracteriza arquiteturas pelo fluxo de dados (simples ou múltiplos) e fluxo de controle. Quatro categorias foram criadas: SISD, SIMD, MISD e MIMD (EIJKHOUT, 2015).

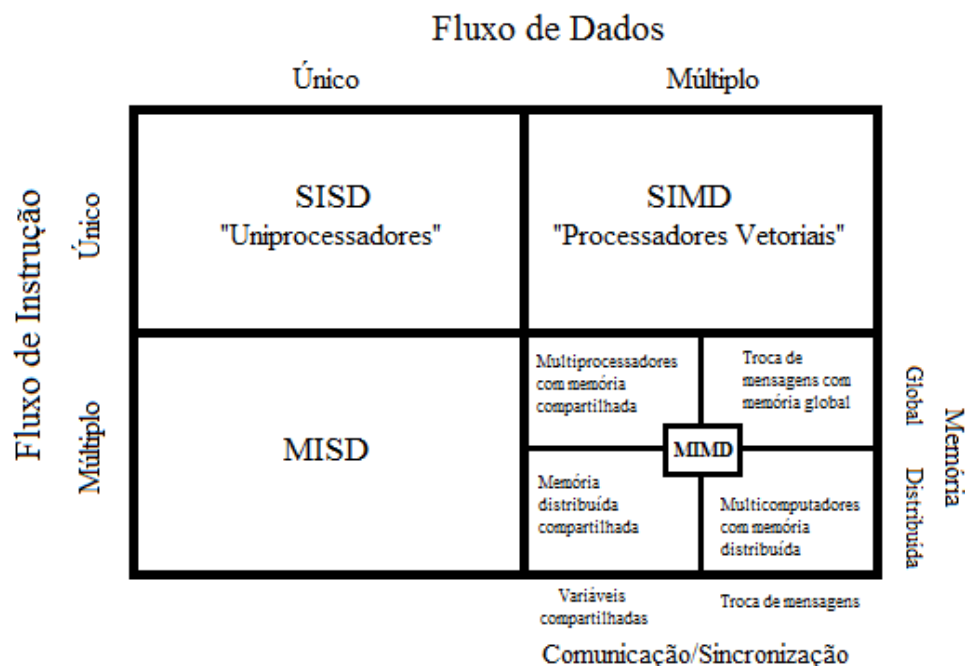
- *SISD (Single Instruction Single Data)*: Esta é a arquitetura tradicional da CPU, em qualquer momento uma única instrução é executada, operando em um único item de dados (EIJKHOUT, 2015).
- *SIMD (Single Instruction Multiple Data)*: Todos os processadores executam a mesma instrução em dados diferentes. Cada um tem seus próprios dados em uma memória local (GEBALI, 2011).

- *MISD (Multiple Instruction Single Data)*: Nenhuma arquitetura que responda a esta descrição existe (EIJKHOUT, 2015). No entanto alguns autores, como o GEBALI (2011), citam alguns exemplos que se assemelham a arquitetura, como as redes neurais e máquinas de fluxo de dados.
- *MIMD (Multiple Instruction Multiple Data)*: Cada processador está executando suas próprias instruções em seus próprios dados locais. Exemplos de tais processadores paralelos são processadores multicore e multiprocessadores multithread em geral (GEBALI, 2011). A maioria dos computadores paralelos atuais são deste tipo (EIJKHOUT, 2015).

E. E. Johnson propôs em 1988 uma classificação das máquinas MIMD baseado na estrutura das memórias (global ou distribuída) e o mecanismo usado para comunicação/sincronização (variáveis compartilhadas ou distribuídas).

A Figura 6 mostra uma concatenação das duas classificações (Flynn-Johnson). Nela há uma relação entre fluxo de instrução e fluxo de dados, apresentando as quatro categorias citadas anteriormente dentro destes grupos de fluxo. Para a arquitetura MIMD, a figura 6 apresenta uma relação entre a comunicação/sincronização para esta arquitetura com o tipo de memória que esta pode utilizar (global ou distribuída).

**Figura 6.** Classificação Flynn-Johnson.



**Fonte:** PARHAMI, 2002, p. 15 (adaptado pela autora).

### 2.3.3 Programação paralela

Adicionar novos núcleos de processamento não é garantia da melhoria de desempenho. Isso acontece porque programas desenhados para executarem sequencialmente em um único processador continuarão trabalhando conforme foram projetados, ignorando completamente a existência de outros processadores (PACHECO, 2011). Dentro desse contexto surge a necessidade da paralelização do software.

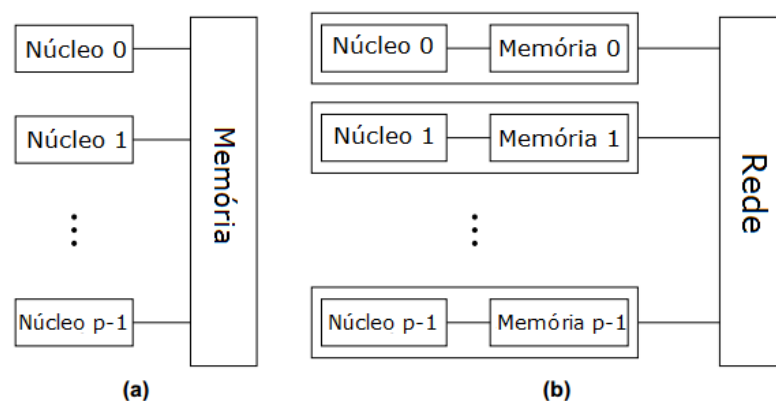
Desenvolver aplicações paralelas não é uma tarefa trivial. Esse tipo de programação pede atenção especial a questões como memória, seja compartilhada ou distribuída (Figura 6), sincronização e balanceamento de tarefas para garantir que todos os núcleos de processamento trabalhem de forma homogênea.

### 2.3.4 Principais Modelos de Programação Paralela

Em um sistema de memória compartilhada os núcleos podem ler e atualizar a memória do computador. Nesse modelo é possível coordenar os processadores através de uma sincronização nas atualizações.

Em uma estrutura com a memória distribuída cada núcleo de processamento tem sua própria memória particular. Neste caso a comunicação acontece de maneira explícita, ou seja, os processadores se comunicam através de trocas de mensagens, fazendo o envio através de uma rede (Figura 7) (PACHECO, 2011).

**Figura 7.** a) Representação de um sistema com memória compartilhada; b) Esquema de um sistema com memória distribuída.



**Fonte:** PACHECO, 2011, p. 9 (adaptado pela autora).

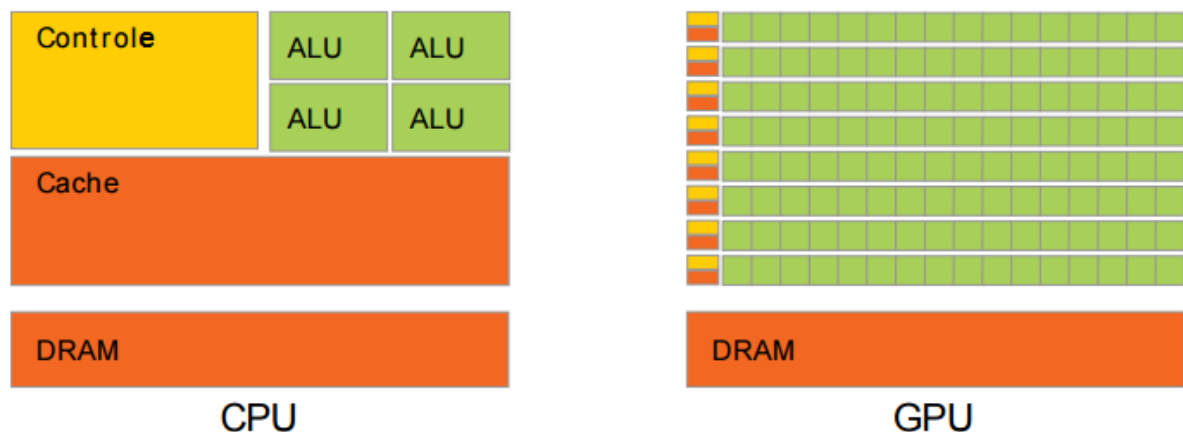
A linguagem C apresenta três extensões diferentes para programação paralela: a interface de passagem de mensagens ou MPI, threads POSIX (Pthreads) e OpenMP (PACHECO, 2011).

Pthreads e OpenMP foram projetadas para sistemas de programação com memória compartilhada e fornecem mecanismos de acesso a esses locais. OpenMP é relativamente uma extensão alto nível para C. Esta pode paralelizar um loop com uma única diretiva. Já o MPI foi concebido para programação em sistemas de memória distribuída, provendo mecanismos para o envio e recebimento de mensagens (PACHECO, 2011).

Em 2006 a NVIDIA apresentou CUDA, uma plataforma de programação paralela de uso geral que utiliza o motor de computação paralela em Unidades de Processamento Gráfico NVIDIA para resolver problemas computacionais complexos de uma maneira eficiente, além de também permitir aos desenvolvedores o uso da linguagem de programação C (NVIDIA, 2016).

As *Graphics Processing Unit* (GPU) são especializadas em computação intensiva massivamente paralela. Foi projetada de tal forma que mais transistores são dedicados ao processamento de dados, em vez de cache de dados e fluxo de controle. A Figura 8 apresenta uma comparação entre a CPU e a GPU, evidenciando a capacidade de processamento através do aumento do número de unidades lógicas e aritméticas.

**Figura 8.** Comparação entre a CPU e GPU.



**Fonte:** NVIDIA, 2016, p. 2.

Como poder ser observado na Figura 8, as Unidades Lógicas Aritméticas, também conhecidas como *Arithmetic Logic Unit* (ALU), são abundantes na GPU, ou seja, o

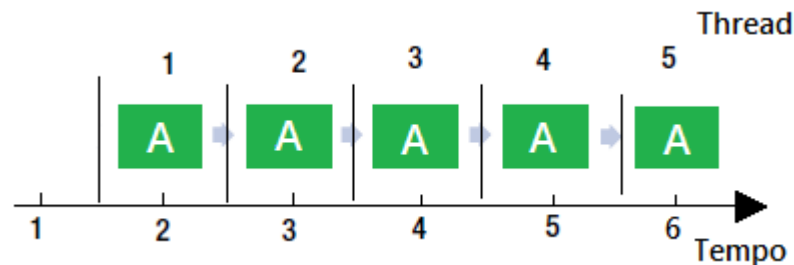
processamento se torna mais rápido graças ao aumento da quantidade das ALU que, por sua vez, são responsáveis pelo processamento dos dados e instruções. Cada fileira com várias ALU possui sua própria Unidade de Controle e memória cache, isto é, para uma ALU na mesma fileira que outra, a memória é compartilhada, enquanto para ALU de fileiras diferentes a troca de dados deve ser explícita.

### 2.3.5 Modelos de particionamento

Os modelos de particionamento são padrões que auxiliam a programação paralela através de definições de como o processamento e a comunicação irá funcionar. A escolha sobre o mais adequado depende das características do problema e da máquina utilizada.

Um padrão comum é o de linha de Produção (*pipeline*), no qual os dados são trabalhados iterativamente, onde cada nova instrução depende do termino da anterior para começar. A figura 9 apresenta o processamento de um elemento no modelo *pipeline*. A medida que o tempo avança cada thread realiza um trabalho no mesmo dado, ou seja, a primeira executa sua função e passa para próxima sucessivamente.

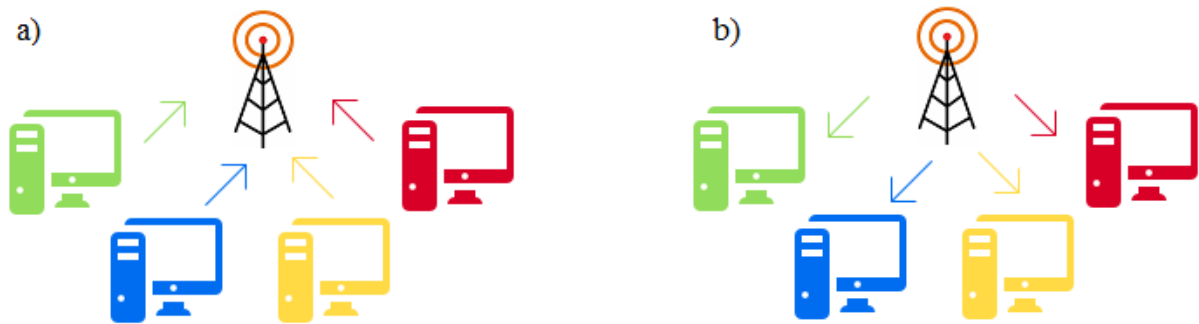
**Figura 9.** Um dado sendo processado iterativamente por diferentes *threads*



**Fonte:** Elaborado pela autora.

Batimento ou *Heartbeat* é outro tipo de particionamento no qual ocorre maior comunicação entre os processadores. Como o próprio nome sugere, este tem um funcionamento semelhante ao do coração, sendo composto por três fases: expansão, contração e processamento. Na primeira os dados são enviados aos núcleos simultaneamente (Figura 10a). O recebimento destes dados acontece na fase dois, contração (Figura 10b), e o processamento ocorre na fase três.

**Figura 10.** a) Fase de expansão ou envio de dados b) Fase de recebimento de dados ou contração



**Fonte:** Elaborado pela autora.

Outro exemplo de partição é o Saco de Tarefas (*Bag-of-Tasks*). Neste padrão as tarefas a serem realizadas são distribuídas na medida em que os processos ficam ociosos, isto é, ao término de um trabalho o processo pode requisitar por mais atividades. O balanceamento de carga deste modelo é uma vantagem pois permite que cada núcleo trabalhe de acordo com sua capacidade. A Figura 11 apresenta um esquema do funcionamento deste modelo, na qual se observa um “saco de tarefas” com todos os dados a serem processados. Na medida em que os processadores precisam de mais, buscam no “repositório”.

**Figura 11.** Representação do modelo *Bag-of-Tasks* em que os dados a serem processados ficam concentrados em um repositório



**Fonte:** Elaborado pela autora

### 2.3.6 Análise e Otimização de Desempenho

Conforme descrito em seu livro, Eijkhout (2015) expõe as dificuldades em quantificar o ganho de desempenho em uma aplicação paralela. Ele elencou dois grandes motivos para o uso de paralelismo: ter acesso a mais memória ou obter maior desempenho.

A determinação da quantidade de memória alcançada pode ser determinada como a soma das memórias individuais no ambiente paralelo.

Uma maneira de mensurar o *speedup* é medindo o tempo de execução do algoritmo sequencial e paralelo, e então comparar os resultados. Ou seja, ganho  $S$  é obtido com a razão entre o tempo sequencial  $T1$  e o paralelo  $T2$  (EIJKHOUT, 2015):

$$S = \frac{T1}{T2} \quad (6)$$

Em um cenário ideal, quanto mais unidades de processamento maior a velocidade adquirida, ou seja, para um tempo sequencial  $T_s$  e um  $T_p$  para um ambiente com  $p$  processadores, tem-se a equação 7:

$$Sp = \frac{T_s}{p} \quad (7)$$

Contudo esse paralelismo linear não é uma regra, pois vários fatores devem ser considerados como, por exemplo, a comunicação. Embora tanto em computação sequencial como na paralela ocorra o ato de ler e gravar dados na memória – algo que também consome tempo de processamento – na programação concorrente adiciona-se o tempo de troca de mensagens entre os processadores (GEBALI, 2011). Claramente, um problema que pode ser resolvido com cálculos independentes, ou seja, sem a comunicação entre os processos. Nesse caso o algoritmo terá um bom desempenho em relação ao *speedup* (EIJKHOUT, 2015).

Para determinar o quão longe o *speedup* obtido está da medida ideal, utiliza-se o conceito de eficiência:

$$Ep = \frac{Sp}{p} \quad (8)$$

O *speedup* linear corresponde a uma eficiência paralela de  $p/p = 1,0$ . Todavia, a eficiência aguardada é sempre inferior (PACHECO, 2011).

Uma razão para um baixo *speedup* se dá em consequência a algumas partes de um código poder ser inerentemente sequenciais (EIJKHOUT, 2015), ou seja, o ganho de

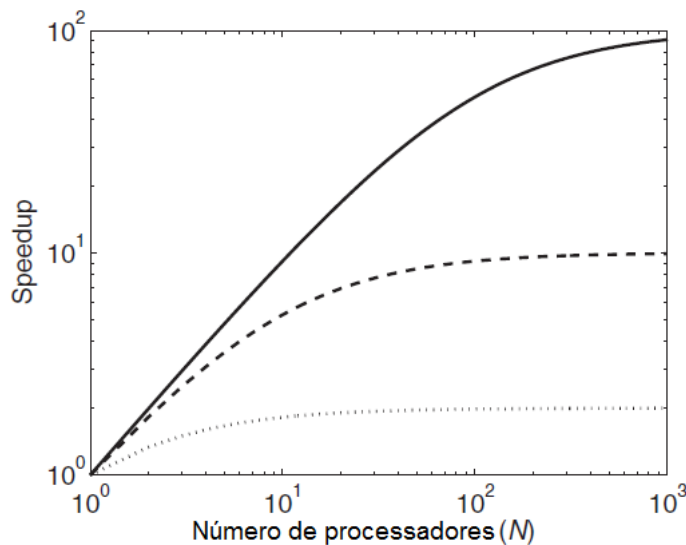
desempenho alcançado com a paralelização de alguns trechos do código pode ficar limitado ao tempo de processamento do fragmento do programa executado em série (RAHMAN, 2013).

Assumindo  $F_p$  e  $F_s$  respectivamente como as frações paralelas e sequenciais do algoritmo, então  $F_p + F_s = 1$ . Declarando  $N$  como o número de processadores paralelos, a lei de Amdahl pode então ser expressa como:

$$Speedup = \frac{1}{F_s + (F_p/N)} \quad (9)$$

A Figura 12 apresenta um comparativo entre três cenários, os quais possuem diferentes porcentagens de porções sequenciais. A linha sólida na figura é para  $F_p = 0,99$ ; A tracejada  $F_p = 0,9$ ; e a linha pontilhada  $F_p = 0,5$ .

**Figura 12.** *Speedup* de acordo com a lei de Amdahl.



**Fonte:** GEBALI, 2011, p. 20 (adaptado pela autora)

Observa-se na Figura 12 que para maiores valores de  $F_p$ , tem-se melhores resultados de *speedup*, ou seja, quanto maior a fração paralelizável do código, maior o ganho de desempenho. No entanto, a aceleração é mais nítida quando a porção  $F_p > 0,5$  (mais de 50% do código é paralelizável). Outra observação é o baixo ganho, quase constante, para um valor grande de  $N$  (GEBALI, 2011). Isto é, quando  $N$  tende ao infinito, a parte paralela pode ser realizada em um tempo muito próximo de zero. Portanto, a aceleração da aplicação não pode

ser feita de forma infinita, incorporando cada vez mais recursos computacionais na resolução ou adicionando mais núcleos de processamento (RAHMAN, 2013).

É importante saber previamente o *speedup* que será alcançado para permitir ao desenvolvedor avaliar o quanto de esforço poderá ser empregado tentando melhorar a velocidade (GEBALI, 2011), poupando não apenas tempo, mas também recursos financeiros.

A lei de Amdahl assume que a fração do código paralelizável é fixa e não depende do tamanho do problema. Porém Gustafson (1988) observou que o paralelismo cresce em uma aplicação quando o tamanho do problema aumenta, portanto há um ajuste entre o cálculo realizado e a adição de mais recursos computacionais. Esta por sua vez pode ser formalizada da seguinte maneira:

$$Speedup = N - Fs.(N - 1) \quad (10)$$

Um programa que responde bem ao aumento de recursos computacionais é chamado de escalável, ou como Blair-chappell e Stokes (2012) explicaram em seu livro: “Um programa perfeitamente escalável é duas vezes mais veloz em uma máquina de quatro núcleos do que em uma de dois, e trabalha quatro vezes mais rápido em um ambiente com 8 processadores.”. Em outras palavras, a proposta é acrescentar processadores/threads e obter ganhos de desempenho na mesma proporção.

A escalabilidade pode ser classificada em dois casos: fortemente escalável e fracamente escalável. Conforme a descrição de Pacheco (2011, p.62, tradução nossa):

Se, ao aumentarmos o número de processos/threads e a eficiência permanece a mesma, sem a necessidade de aumentar o tamanho do problema, diz-se que o programa é fortemente escalável. Agora, se pudermos manter a eficiência constante através do aumento do tamanho do problema na medida que cresce o número de processos/threads, então o programa é fracamente escalável.

Embora essa medida não indique o ganho de velocidade, ela aponta a capacidade de crescimento, ou seja, identifica o momento em que o aumento de recursos computacionais deixa de gerar ganho de desempenho.

No entanto, tanto para aplicações sequenciais como para as paralelas, leitura e escrita dos dados é quase inevitável. A comunicação com a memória consome tempo de

processamento, principalmente quando há comunicação entre os processadores. Infelizmente a troca de mensagens e transferências de dados trazem muitas limitações para o desenvolvimento de aplicações paralelas, tais como as elencadas por Gebali (2011):

“**Atraso da rede de interconexão:** Transmissão de dados pela rede sofre com o atraso de propagação de bit, atraso na transmissão de mensagens/dados, e atraso na fila dentro da rede. Esses fatores dependem da topologia da rede, o tamanho dos dados enviados, a velocidade de operação da rede e assim por diante.

**Largura de banda da memória:** Não importa quão grande seja a capacidade de memória, o acesso a conteúdo da memória é feito usando uma única porta que transfere um dado para dentro ou fora da memória em qualquer ciclo de acesso à memória.

**Colisões de memória:** Dois ou mais processadores tentam acessar o mesmo módulo de memória. Arbitragem deve ser fornecida para permitir que um processador acesse a memória por vez.

**Parede da memória:** A velocidade de transferência de dados na memória é muito mais lenta que a velocidade de processamento. Este problema é resolvido usando hierarquia de memória como:

Registro ↔ cache ↔ RAM ↔ disco eletrônico ↔ disco magnético ↔ disco óptico “

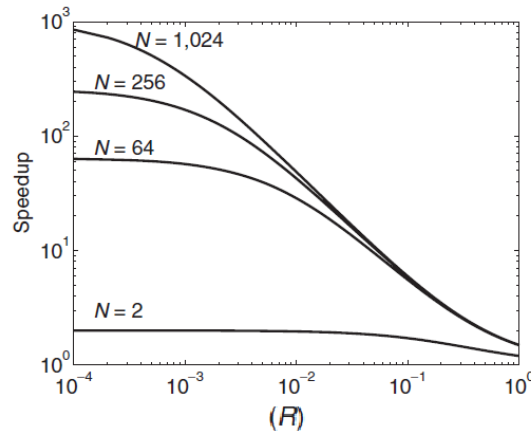
A Figura 13 mostra o efeito de dois parâmetros,  $N$  e  $R$ , na aceleração quando  $\alpha = 1$ , nos quais  $N$  é o número de processadores,  $R$  é a razão do tempo para acessar um bloco de dados na memória ( $T_a$ ) em relação ao tempo de processamento deste ( $T_p$ ) (equação 11) e  $\alpha$  é um fator que leva em conta limitações de acesso à memória compartilhada (GEBALI, 2011):

$$R = \frac{T_a}{T_p} \quad (11)$$

- $\alpha = 1/N$  quando cada processador possui sua própria cópia dos dados.
- $\alpha = 1$  quando os dados são distribuídos para cada processo através de uma memória central.

- $A > N$  quando todos os processadores solicitam dados e colidem uns com os outros.

**Figura 13.** Comportamento do *speedup* em função de  $R$  e  $N$ .



**Fonte:** GEBALI, 2011, p. 19 (adaptado pela autora)

Como pode ser observado na Figura 13, na medida em  $R$  aproxima-se do valor 1 e os valores de  $N$  aumentam (número de processadores), o *speedup* diminui. Em linhas gerais, quanto maior o grau do paralelismo, maior comunicação entre os processos. Mais comunicação requer sincronização dos processos, o que pode gerar gargalos na troca de mensagens ou transferências de dados.

Esses são alguns pontos a serem relevados durante o desenvolvimento de aplicações paralelas, e cabe ao programador identificar e implementar as otimizações concorrentes, escolhendo, por exemplo, o melhor modelo de particionamento para o problema trabalhado, extraíndo desta maneira o máximo de paralelismo. Outras observações a serem feitas pelo desenvolvedor são as otimizações de código.

Conforme descrito por Aho (2007), a “eliminação de instruções desnecessárias, ou a substituição de uma sequência de instruções por outro conjunto mais rápido, que efetua a mesma operação, usualmente é denominada ‘melhoria de código’ ou ‘otimização do código’ ”.

A seguir serão discutidas melhorias de desempenho de código.

### 2.3.7 Otimização de código pelo programador

Os pontos críticos, também conhecidos como gargalos de software, são aqueles em que otimizações de desempenho são viáveis. Estes podem ser classificados em três tipos: linguagens, funções e blocos de repetição.

Gargalos de linguagem levam em conta o uso de estruturas simples e estáticas, dispensando o uso de ponteiros e outras estruturas dinâmicas. No que se refere ao segundo tipo, gargalos de funções decorrem da sobrecarga do sistema com o uso em excesso de modularização e estruturação do código. Para o alcance de melhor desempenho recomenda-se evitar o uso de funções simples que podem ser facilmente substituídas, como por exemplo, a o cálculo matemático  $\text{pow}(x,2)$  da linguagem C, podendo ser escrito apenas como  $x*x$ . Já os gargalos em blocos de repetição podem ser mais explorados, apresentando um conjunto maior de atividades para o melhoramento de desempenho: desdobramento de código, remoção de testes desnecessários, inversão de aninhamentos e fissão ou fusão de laços.

### 2.3.8 Otimização de código pelo compilador

Algumas das otimizações que podem ser realizadas pelo compilador são: propagação de código, renomeação de variáveis, remoção de código morto e desdobramento de constantes.

Em propagação de código ou propagação de cópia, refere-se a casos em que uma variável possui uma cópia do valor de outra, sem outras utilidades, permitindo a eliminação da duplicação.

A renomeação de variáveis é vantajosa por eliminar a dependência entre instruções, permitindo o uso de paralelismo e garantindo um resultado satisfatório:

|              |                |
|--------------|----------------|
| $X = w * w;$ | $X_0 = w * w;$ |
| $Z = X * r;$ | $Z = X_0 * r;$ |
| $X = t - a;$ | $X = t - a;$   |

A propagação de constantes ou o desdobramento de constantes refere-se à substituição de valores constantes nas instruções:

|            |              |
|------------|--------------|
| X = 10;    | X = 10;      |
| Z = 20;    | Z = 20;      |
| Y = Z - X; | X = 10 + 20; |

## 2.4 Paralelização dos algoritmos de aprendizado de máquina

Os algoritmos de clusterização e classificação possuem um alto custo computacional. Jain, Murty e Flynn (1999) em seu trabalho levantaram as complexidades (de tempo e espaço) de vários algoritmos de clusterização.

A tabela 3 elenca alguns algoritmos de clusterização e os relaciona com suas respectivas complexidades de tempo e de espaço. Nela pode-se observar como esses algoritmos são dispendiosos computacionalmente, principalmente em relação ao tempo de processamento.

**Tabela 3.** Complexidade de alguns algoritmos de clusterização.

| Algoritmos de clusterização | Complexidade de tempo | Complexidade de espaço |
|-----------------------------|-----------------------|------------------------|
| leader                      | $O(kn)$               | $O(k)$                 |
| k-means                     | $O(nkl)$              | $O(k)$                 |
| ISODATA                     | $O(nkl)$              | $O(k)$                 |
| shortest spanning path      | $O(n^2)$              | $O(n)$                 |
| single-line                 | $O(n^2 \log n)$       | $O(n^2)$               |
| complete-line               | $O(n^2 \log n)$       | $O(n^2)$               |

**Fonte:** JAIN; MURTY; FLYNN, 1999, p. 293 (adaptado pela autora).

Outro exemplo de um algoritmo de clusterização com alta complexidade de tempo é o *Quality Threshold Clustering* que tem no pior caso a complexidade  $O(n^5)$  (JR., 2015). Os algoritmos hierárquicos AGNES e DIANA possuem  $O(n^2)$  e  $O(2^n)$  respectivamente (KAUFMAN; ROUSSEEUW, 1990).

Dentro desse contexto, a paralelização pode otimizar o código, podendo reduzir e muito o seu tempo de processamento. Danalis, McCurdy e Vetter (2012) paralelizaram o *Quality Threshold Clustering* com um modelo híbrido de MPI e OpenMP e tiveram um *speedup* de 35x.

Bhimani, Leeser e Mi (2015) otimizaram o algoritmo K-means usando CUDA, MPI e OpenMP. Vários testes foram realizados com dois dados de entrada: uma imagem de 300x300 pixels e outra 1164x1200 pixels. No experimento o OpenMP teve melhor desempenho com imagens pequenas e médias enquanto CUDA teve melhor performance com as maiores. Em todos os casos o processamento paralelo teve melhor desempenho que o sequencial. O Quadro 2 apresenta os resultados obtidos, trazendo “K” (quantidade de clusters finais), o número de iterações necessárias para alcançar o resultado e quanto tempo o processo levou, tanto sequencialmente como os modelos paralelos citados.

**Quadro 2.** *Speedup* ganho com OpenMP, MPI e CUDA no algoritmo K-means.

(a) Tempo e *speed-up* (SU) para imagem de 300X300 pixels

| K   | Iter. | Seq. (s) | OpenMP (s) | OpenMP SU | MPI (s) | MPI SU | CUDA (s) | CUDA SU |
|-----|-------|----------|------------|-----------|---------|--------|----------|---------|
| 10  | 4     | 1.8      | 0.1        | 16.51     | 0.13    | 13.84  | 0.16     | 10.77   |
| 30  | 14    | 5.42     | 0.21       | 25.8      | 0.32    | 16.93  | 0.47     | 11.48   |
| 50  | 63    | 30.08    | 1.28       | 23.5      | 1.45    | 20.74  | 2.06     | 14.56   |
| 100 | 87    | 43       | 1.39       | 30.93     | 1.98    | 21.71  | 2.68     | 16.01   |

(b) Tempo e *speed-up* (SU) para imagem de 1164X1200 pixels

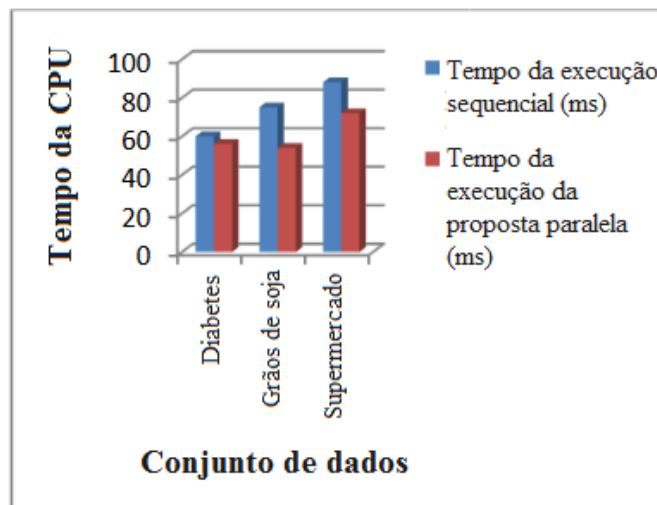
| K   | Iter. | Seq. (s) | OpenMP (s) | OpenMP SU | MPI (s) | MPI SU | CUDA (s) | CUDA SU |
|-----|-------|----------|------------|-----------|---------|--------|----------|---------|
| 30  | 21    | 1187.08  | 49.54      | 23.95     | 60.51   | 19.61  | 46.48    | 25.53   |
| 60  | 49    | 2651.28  | 98.77      | 26.84     | 115.68  | 22.91  | 93.68    | 28.29   |
| 120 | 92    | 4595.96  | 159.97     | 28.72     | 170.22  | 26.99  | 154.36   | 29.77   |
| 240 | 166   | 8897.72  | 300.54     | 29.60     | 315.15  | 28.23  | 294.01   | 30.26   |

**Fonte:** BHIMANI; LEESER; MI, 2015, p. 5 (adaptado pela autora)

Em outro trabalho paralelizando o K-means com OpenMP, Naik, Kumar e Ramakrishna (2013) também tiveram resultados interessantes, diminuindo o tempo de processamento.

A Figura 14 apresenta um gráfico relacionando o tempo de execução (em milissegundos) dos modelos sequencias e paralelos com três conjuntos de dados. Conforme ilustrado pela imagem, o tempo de execução sequencial foi muito maior que o tempo da versão paralela do algoritmo.

**Figura 14.** Comparação do tempo de execução do algoritmo sequencial com o paralelo.



**Fonte:** NAIK; KUMAR; RAMAKRISHNA, 2013, p. 4 (adaptado pela autora)

A paralelização de algoritmos de classificação e clusterização vem se mostrando uma área válida de exploração. No entanto, nem todos os algoritmos foram otimizados por esta técnica ou foram pouco estudados.

## 2.5 Considerações finais

Desenvolvimento de aplicações paralelas são vantajosas, principalmente quando estas utilizam-se de grandes volumes de dados para o processamento. O campo de ML é um exemplo característico dessas aplicações e isto somado a alta complexidade computacional desses algoritmos, tornam o casamento destes dois temas conveniente, especialmente pelo ganho de desempenho produzido a partir de suas combinações, tornando realista o tempo de resposta dos algoritmos.

## Capítulo 3 - Materiais e métodos

### 3.1 Considerações iniciais

Para o desenvolvimento paralelo do algoritmo DIANA com OpenMP e MPI foram feitas adaptações no código e escolhido um modelo de particionamento para o processamento concorrente. Para o aprimoramento do algoritmo foram aplicadas técnicas de otimização propostas no estudo de compiladores expostas no tópico 2.3.7 (Otimização de código pelo programador) e 2.3.8 (Otimização de código pelo compilador), como por exemplo, o abandono de estruturas dinâmicas durante o desenvolvimento do projeto e a eliminação de dependências entre as variáveis. Além disso, parte da lógica do problema foi alterada para aproveitar o máximo de desempenho de cada processador, isto é, ao invés do cálculo sequencial da distância entre os pontos, a nova lógica desenvolvida divide a quantidade total de distâncias igualmente entre os processadores, permitindo deste modo o cálculo simultâneo destas.

### 3.2 Algoritmo DIANA

O algoritmo DIANA consiste em analisar dados e através de padrões encontrados pela distância euclidiana, são criadas subclassificações para os objetos. Essa categorização ocorre de maneira a dividir grupos maiores em menores. Para o projeto propõe-se paralelizar e comparar os resultados do método *Complete Linkage* exposto no Quadro 1. Nesse contexto, a ideia do algoritmo é reconhecer padrões e, deste modo, classificar objetos de acordo com suas semelhanças/diferenças. Para tal, características dos elementos são medidas através de coordenadas (x, y) e comparadas entre si. As similaridades/dissimilaridades são registradas com a construção de uma matriz de distâncias que contém o intervalo entre todos os pontos. Nesta matriz, as linhas e colunas são os pontos do conjunto de dados analisado. Johnson (1967) em seu trabalho expôs um exemplo de uma tabela de distâncias (ou similaridades/dissimilaridades), a qual é apresentada neste trabalho como Tabela 4.

**Tabela 4.** Um exemplo de uma matriz de distância utilizada em algoritmos hierárquicos.

| D | 1    | 2    | 3    | 4    | 5    | 6    |
|---|------|------|------|------|------|------|
| 1 | 0    | 0.31 | 0.23 | 0.31 | 0.23 | 0.23 |
| 2 | 0.31 | 0    | 0.31 | 0.23 | 0.31 | 0.31 |
| 3 | 0.23 | 0.31 | 0    | 0.31 | 0.4  | 0.7  |
| 4 | 0.31 | 0.23 | 0.31 | 0    | 0.31 | 0.31 |
| 5 | 0.23 | 0.31 | 0.4  | 0.31 | 0    | 0.7  |
| 6 | 0.23 | 0.31 | 0.7  | 0.31 | 0.7  | 0    |

**Fonte:** JOHNSON, 1967, p.244 (adaptado pela autora)

Como é possível ser observado na Tabela 4, a matriz formada é do tipo simétrica, ou seja, os elementos opostos em relação a linha diagonal principal são idênticos. Com essa característica parte do trabalho de análise pode ser reduzido, pois apenas um dos lados – acima ou abaixo da linha diagonal – pode ser utilizado para o cálculo, conforme ilustra a Tabela 5.

**Tabela 5.** Matriz de distâncias simétrica.

| D | 1    | 2    | 3    | 4    | 5    | 6    |
|---|------|------|------|------|------|------|
| 1 | 0    | 0.31 | 0.23 | 0.31 | 0.23 | 0.23 |
| 2 | 0.31 | 0    | 0.31 | 0.23 | 0.31 | 0.31 |
| 3 | 0.23 | 0.31 | 0    | 0.31 | 0.4  | 0.7  |
| 4 | 0.31 | 0.23 | 0.31 | 0    | 0.31 | 0.31 |
| 5 | 0.23 | 0.31 | 0.4  | 0.31 | 0    | 0.7  |
| 6 | 0.23 | 0.31 | 0.7  | 0.31 | 0.7  | 0    |

**Fonte:** JOHNSON, 1967, p.244 (adaptado pela autora)

### 3.3 O projeto

Como o algoritmo consiste em realizar o cálculo sequencial das distâncias entre os pontos, e este é feito de forma independente, ou seja, não necessita de outros valores para sua determinação, o processo de busca pela maior distância (*Complete linkage*) pode ser realizado de maneira concorrente.

Para isto o programa desenvolvido lê um arquivo de texto com as coordenadas (x,y) das amostras e então os pares de elementos são distribuídos. Um par de elemento é a combinação entre uma linha e coluna da matriz, ou seja, as distâncias entre os objetos do conjunto provêm da distância euclidiana formada por cada par. A Figura 15 ilustra a formação de pares entre a linha um e as colunas dois, três e quatro.

**Figura 15.** Exemplo de combinação de pontos, gerando pares únicos com suas respectivas distâncias.



**Fonte:** Elaborado pela autora

Cada par criado é visto como um único objeto a ser analisado, uma vez que a versão *Complete Linkage* do algoritmo DIANA procura a maior distância entre os elementos. Como cada par é considerado um único objeto, um mesmo processador pode ter a mesma linha e coluna que outro processador, porém nenhum dos nós de processamento terá a mesma combinação de linha com coluna. Sendo assim, a distribuição dos pares de pontos é feita entre os processadores e a busca pelo melhor resultado é feita paralelamente.

Para este estudo utilizou-se de um cluster homogêneo formado por cinco máquinas com 8Gb de RAM e processador Intel® Core™ i7-3770 em uma topologia do tipo estrela, além de uma versão híbrida OpenMP/MPI do algoritmo DIANA. Os pares dos pontos são distribuídos entre os computadores do cluster, procurando reparti-los de maneira igualitária, evitando desse modo, o ócio dos nós. A repartição é feita somente entre as distâncias de apenas um dos lados – superior ou inferior – da matriz, ou seja, o lado escolhido é o conjunto de dados a ser analisado. A distribuição dos dados é feita pela divisão do número total de pares dividido pela quantidade de nós no *cluster*. Para tal é necessário determinar quantas distâncias serão analisadas. A equação 12 determina o tamanho do conjunto, onde “*totalPontos*” é a quantidade de pontos na base de dados e “*conjunto*” representa a quantidade de distâncias entres esses, ou seja, “*conjunto*” é o triângulo de uma das partes da matriz (superior ou inferior):

$$Conjunto = \frac{(totalPontos^2) - totalPontos}{2} \quad (12)$$

A matriz apresentada por Johnson (1967) (Tabela 4), neste contexto, possui de acordo com a equação 13, um conjunto de dados com valor total de quinze distâncias (pares de pontos) para serem calculadas e analisadas:

$$\text{Conjunto} = \frac{(6^2) - 6}{2} = 15 \quad (13)$$

Com o total do conjunto determinado, a busca concorrente pelo melhor resultado pode ser iniciada. Neste estudo a paralelização foi realizada em duas fases com as ferramentas OpenMP e MPI.

### 3.3.1 Desenvolvimento com MPI

Uma vez que a quantidade total das distâncias em um dos triângulos da matriz foi determinada, este pode ser dividido entre os computadores do *cluster*. A Tabela 6 apresenta um exemplo para três máquinas trabalhando com a matriz de Johnson (1967) (Tabela 4), na qual o conjunto total é dividido entre os nós, ou seja, três regiões (amarela, verde e azul) são criadas com cinco distâncias a serem calculadas e classificadas:

**Tabela 6.** Determinação do conjunto de dados e sua distribuição.

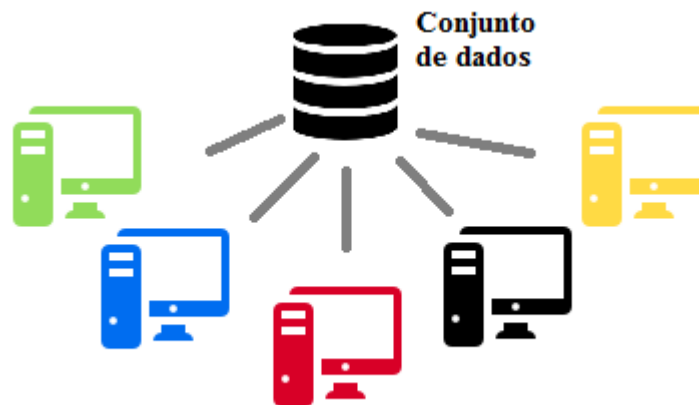
| D | 1    | 2    | 3    | 4    | 5    | 6    |
|---|------|------|------|------|------|------|
| 1 | 0    | 0.31 | 0.23 | 0.31 | 0.23 | 0.23 |
| 2 | 0.31 | 0    | 0.31 | 0.23 | 0.31 | 0.31 |
| 3 | 0.23 | 0.31 | 0    | 0.31 | 0.4  | 0.7  |
| 4 | 0.31 | 0.23 | 0.31 | 0    | 0.31 | 0.31 |
| 5 | 0.23 | 0.31 | 0.4  | 0.31 | 0    | 0.7  |
| 6 | 0.23 | 0.31 | 0.7  | 0.31 | 0.7  | 0    |

**Fonte:** JOHNSON, 1967, p.244 (adaptado pela autora)

Em resumo, o algoritmo DIANA constrói uma matriz com as distâncias (semelhanças/diferenças) entre os pontos e os nós do *cluster* ficam responsáveis pela análise de regiões específicas desta matriz.

Uma vez que as regiões da matriz são determinadas para os nós do *cluster* (Figura 16), o cálculo das distâncias e análise do melhor resultado pode ser feito através do uso de OpenMP.

**Figura 16.** Esquematização geral da distribuição de dados entre os nós do *cluster*.



**Fonte:** Elaborado pela autora

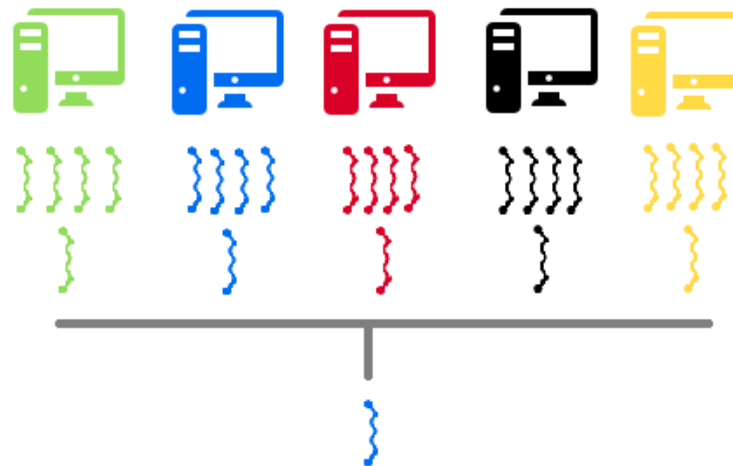
A ideia principal é dividir o conjunto total de dados entre os nós do *cluster* e então, cada *node* por sua vez promove uma distribuição de seus respectivos dados entre seus próprios *threads* locais. Desta maneira, inicia-se a fase de análise com OpenMP.

### 3.3.2 Desenvolvimento com OpenMP

Uma vez que cada computador do *cluster* tem uma região delimitada para análise, ou seja, seu próprio subconjunto de dados, um outro nível de paralelismo pode ser criado. Como cada nó deve procurar dentro da sua região pela maior distância entre seus pontos, e este é um processo sem dependências, a procura pode ser feita concorrentemente com os *threads* do OpenMP. Cada *node* do cluster pode procurar pelo melhor resultado com o uso de *threads*. A Figura 17 traz um esquema ilustrativo da relação entre os nós e seus respectivos *threads* locais.

É importante reforçar que não existe comunicação entre os *threads*, tanto entre aqueles de diferentes computadores como entre os locais de cada *node*. Isso promove ganho de desempenho ao evitar sincronização de comunicação entre eles, além de evitar gargalos. Cada nó tem acesso apenas aos seus pares, assim como os *threads* acessam apenas seus respectivos dados.

**Figura 17.** Esquemática geral da distribuição de dados entre os threads locais, eleição do melhor resultado local e então a determinação do melhor resultado global.



**Fonte:** Elaborado pela autora

A busca dos intervalos entre os pontos é realizada com duas variáveis de controle percorrendo a matriz, onde uma desloca-se pela linha (ponto A) e a segunda pelas colunas (ponto B). A Figura 18 apresenta o código utilizado para caminhar pelas linhas da matriz, na qual a variável “*posA*” refere-se ao número da linha que está sendo analisada, “*inicioA*” guarda o valor do início da região a ser estudada por cada *node*, ou seja, em qual linha cada um deve começar a análise. Já a variável “*fimA*” registra o fim da região da matriz que pode ser analisado por cada *node*, ou seja, a última linha a ser analisada por cada nó.

**Figura 18.** Laço de repetição para percorrer as linhas da matriz de distâncias.

```
#pragma omp declare reduction \
    (maxPar:Par:omp_out=maximo(omp_out, omp_in)) \
    initializer(omp_priv = {10,0})

#pragma omp for schedule (static) reduction(maxPar : maior_finalistas)
for(posA=inicioA; posA< fimA; posA++)
{
```

**Fonte:** Elaborado pela autora

A distribuição paralela é feita usando as diretivas de compilação do OpenMP, conforme Figura 18, incluindo uma redução da variável “*maior\_finalistas*”, ou seja, cada thread encontrará sua própria “maior distância” e irá comparar seu resultado com os outros threads daquele mesmo computador. Através da comparação dos resultados, o melhor é eleito para cada

nó, isto é, cada *node* encontra a maior distância dentro da sua região, conforme esquematização da Figura 17 e o exemplo na Tabela 7.

**Tabela 7.** Maior distância de cada região encontrada por cada thread.

| D | 1    | 2    | 3    | 4    | 5    | 6    |
|---|------|------|------|------|------|------|
| 1 | 0    | 0.31 | 0.23 | 0.31 | 0.23 | 0.23 |
| 2 | 0.31 | 0    | 0.31 | 0.23 | 0.31 | 0.31 |
| 3 | 0.23 | 0.31 | 0    | 0.31 | 0.4  | 0.7  |
| 4 | 0.31 | 0.23 | 0.31 | 0    | 0.31 | 0.31 |
| 5 | 0.23 | 0.31 | 0.4  | 0.31 | 0    | 0.7  |
| 6 | 0.23 | 0.31 | 0.7  | 0.31 | 0.7  | 0    |

**Fonte:** JOHNSON, 1967, p.244 (adaptado pela autora)

Para o exemplo proposto na Tabela 6, os *threads* precisarão fazer uma redução dos valores encontrados, os quais estão guardados na variável “*maior\_finalista*” (Figura 18). A redução de variáveis encontra automaticamente o melhor resultado dentre os *threads*, onde uma comparação é feita entre os valores locais e o melhor resultado é eleito, atualizando o valor em todos os *threads* após a redução. Neste contexto, ao final da análise a maior distância em cada nó é encontrada paralelamente e será reduzida na variável *maior\_finalista*. Para a matriz de Johnson (1967), com a análise paralela em um *cluster* de três *nodes* haveriam três melhores resultados (Tabela 7).

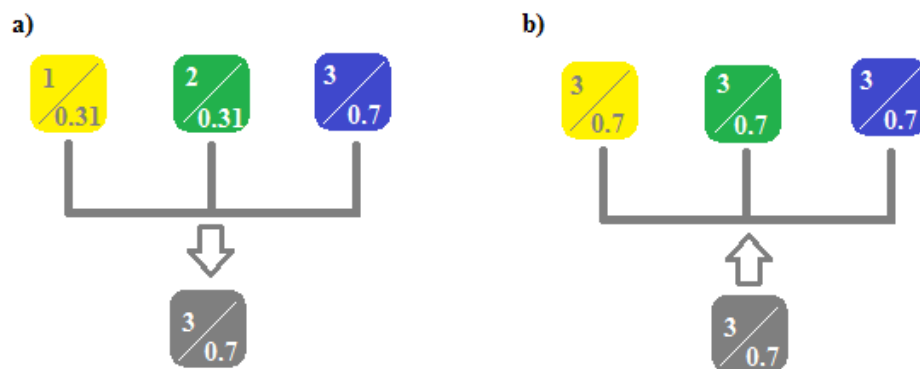
Cada nó é responsável pelo cálculo em uma região da matriz de distâncias. Dentro de cada *node* são disparados *threads* que irão procurar paralelamente – dentro da região delimitada para aquele nó – pelo melhor resultado. Os *threads* de cada nó irão determinar seu melhor resultado local e através de uma redução de variáveis, a maior distância regional é encontrada. Uma vez que cada computador do *cluster* definiu a sua maior distância, uma outra redução – desta vez entre os valores dos nós – pode ser feita determinando deste modo o maior valor da matriz de distâncias, conforme o esquema da Figura 17 apresentado anteriormente.

Para que a comparação dos resultados entre os nós – e por consequência a definição do melhor resultado global – seja realizada, é feito um “empacotamento” com o índice do nó do *cluster* e o valor da maior distância regional para o respectivo *node*. O comando “*MPI\_Allreduce*” compara as distâncias escolhidas dos computadores, elege o melhor resultado e atualiza todos os nós, isto é, ao final da redução a variável “*melhor\_eleito*” guardará a distância global em todos os *nodes*. O esquema apresentado na Figura 19 ilustra o

funcionamento do “*MPI\_Allreduce*”, já o trecho de código exposto na Figura 20 exemplifica como foi realizada a redução.

**Figura 19. a)** A variável “pacote” antes da redução contém a distância de cada nó além do índice identificador deste.

**b)** Após a redução da variável “*pacote*”, a maior distância global é encontrada e a variável é atualizada em todos os nós.



**Fonte:** Elaborado pela autora

**Figura 20.** “Empacotamento” da distância local de cada nó com o índice deste, além da redução da variável “*pacote*” na variável “*melhor\_eleito*”.

```
pacote[0] = maior_finalistas.valor;
pacote[1] = rank;
MPI_Allreduce( pacote, melhor_eleito, 1, MPI_FLOAT_INT, MPI_MAXLOC, MPI_COMM_WORLD );
```

**Fonte:** Elaborado pela autora

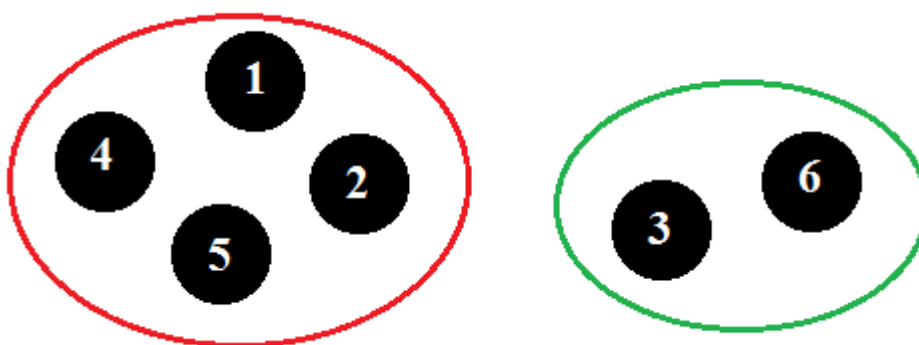
Após a redução em MPI e a maior distância da matriz identificada, o par de pontos correspondente à distância eleita pode ser atualizado, ou seja, para a Tabela 8 os pontos seis e três podem ser retirados do grupo principal. Uma vez que um par deixa de pertencer a um conjunto, passam a fazer parte de um subgrupo deles (Figura 21).

**Tabela 8.** Maior distância global encontrada.

| D | 1    | 2    | 3    | 4    | 5    | 6    |
|---|------|------|------|------|------|------|
| 1 | 0    | 0.31 | 0.23 | 0.31 | 0.23 | 0.23 |
| 2 | 0.31 | 0    | 0.31 | 0.23 | 0.31 | 0.31 |
| 3 | 0.23 | 0.31 | 0    | 0.31 | 0.4  | 0.7  |
| 4 | 0.31 | 0.23 | 0.31 | 0    | 0.31 | 0.31 |
| 5 | 0.23 | 0.31 | 0.4  | 0.31 | 0    | 0.7  |
| 6 | 0.23 | 0.31 | 0.7  | 0.31 | 0.7  | 0    |

**Fonte:** JOHNSON, 1967, p.244 (adaptado pela autora)

**Figura 21.** Esquematização do reagrupamento do par de pontos seis e três após a redução em MPI.



**Fonte:** Elaborado pela autora

A seguir apresenta-se uma versão em pseudocódigo de como a paralelização da análise da matriz de distâncias foi realizada.

```

maior_finalistas.index = -1;
maior_finalistas.valor = 0;
// declaração da redução da variavel maior_finalista para determinar qual é a
maior distancia ENTRE as threads de CADA pc
#pragma omp declare reduction \
    (maxPar:Par:omp_out=maximo(omp_out, omp_in)) \
    initializer(omp_priv = {10,0})
#pragma omp for schedule (static) reduction(maxPar : maior_finalistas)
Para (linha = linhaInicialA; linha < linhaFinalA; linha+1)
//Laço para controle da linha da matriz
{
    //Teste para verificar qual a coluna inicial
    Se(controle == 0)
    {
        controle == 1;
    }
    Se não(controle > 0)
    {
        colunaInicialB = 0;
    }
    Para (coluna = colunaInicialB; coluna < colunaFinalB; coluna+1)
    {
        // Se pontos que estão no mesmo cluster
        Se(Grupo[linha] == Grupo[coluna])
        {
            d = sqrt(pow((xb-xa),2)+pow((yb-ya),2));
            //calcula distância entre os pontos
        }
        Se(d > maior_finalistas.valor)
        // d é maior que a atual maior distancia
        {
            maior_finalistas.index = tid;
            // guarda índice da thread
            maior_finalistas.valor = maior;
            // guarda distancia encontrada
            posicaoA = linha;
            // guarda a linha e a coluna responsável por esta distancia
            posicaoB = coluna;
        }
    }
} // Ao final da execução deste laço é feita a redução calculando qual thread tem
a maior distância

pacote[0] = maior_finalistas.valor; // maior distância encontrada para cada nó
pacote[1] = rank; // índice do nó que encontrou esta distancia
MPI_Allreduce(pacote, melhor_eleito, 1, MPI_FLOAT_INT, MPI_MAXLOC,
MPI_COMM_WORLD);
// redução elegendo a maior distância encontrada entre os nós do cluster
//envio para todos os nós da linha e coluna responsável para aquela distancia
medida e eleita
MPI_Bcast(&posicaoA, 1, MPI_INT, melhor_eleito[1], MPI_COMM_WORLD);
MPI_Bcast(&posicaoB, 1, MPI_INT, melhor_eleito[1], MPI_COMM_WORLD);
agrupar(posicaoA, posicaoB);
// agrupar os pontos da linha e coluna encontrados

```

### 3.4 Considerações finais

O algoritmo DIANA consiste em um algoritmo de aprendizado de máquina não supervisionado, que realiza o reconhecimento de padrões entre os dados de uma base. Para tal, este utiliza-se do cálculo de distâncias euclidianas. Este trabalho tem como melhor resultado a maior distância entre os objetos do conjunto de dados para classificação, também conhecido como método *Complete Linkage*.

O DIANA cria uma matriz de distâncias entre os pontos do conjunto de dados e então procurar o melhor resultado nesta matriz através de uma análise sequencial. Uma versão paralela e híbrida do algoritmo foi desenvolvida combinando OpenMP e MPI a fim de acelerar este processo. Sendo assim, regiões na matriz foram criadas para delimitar a busca feita pelos nós de um *cluster* (MPI). Um outro nível de paralelismo é formado com o OpenMP, uma vez que cada *node* dispara vários *threads* que irão examinar a região demarcada da matriz de forma concorrente. Desta maneira, a classificação da base de dados pode ser otimizada.

## Capítulo 4 - Resultados

### 4.1 Considerações iniciais

Para este trabalho foram realizados três testes com uma base de dados de 11826 pontos referentes a localizações de usuários do aplicativo MOPSI (Anexo A), sem coordenadas repetidas. Esse capítulo apresenta os resultados obtidos e comparados a *speedup* teóricos.

### 4.2 Resultados experimentais

Para a determinação do ganho de desempenho é necessária a comparação do tempo de análise sequencial com a paralela. Para isto, mediu-se o tempo de execução dos trechos e então determinou-se as porcentagens representativas destas no código, conforme a Tabela 9 a seguir:

**Tabela 9.** Determinação da porcentagem sequencial e paralela do programa monoprocessado

| <b>Tempo total<br/>em segundos</b> | <b>Tempo<br/>Seq. em s.</b> | <b>Tempo<br/>Paral. em s.</b> | <b>% Seq.</b> | <b>% Paral.</b> |
|------------------------------------|-----------------------------|-------------------------------|---------------|-----------------|
| 4001                               | 18                          | 3983                          | 0,44989       | 99,55011        |
| 3998                               | 18                          | 3980                          | 0,45023       | 99,54977        |
| 4002                               | 17                          | 3985                          | 0,42479       | 99,57521        |
|                                    |                             |                               | <b>0,44</b>   | <b>99,56</b>    |

**Fonte:** Elaborado pela autora

De acordo com a lei de Amdahl o *speedup* pode ser medido através da equação 14, então para este trabalho teremos:

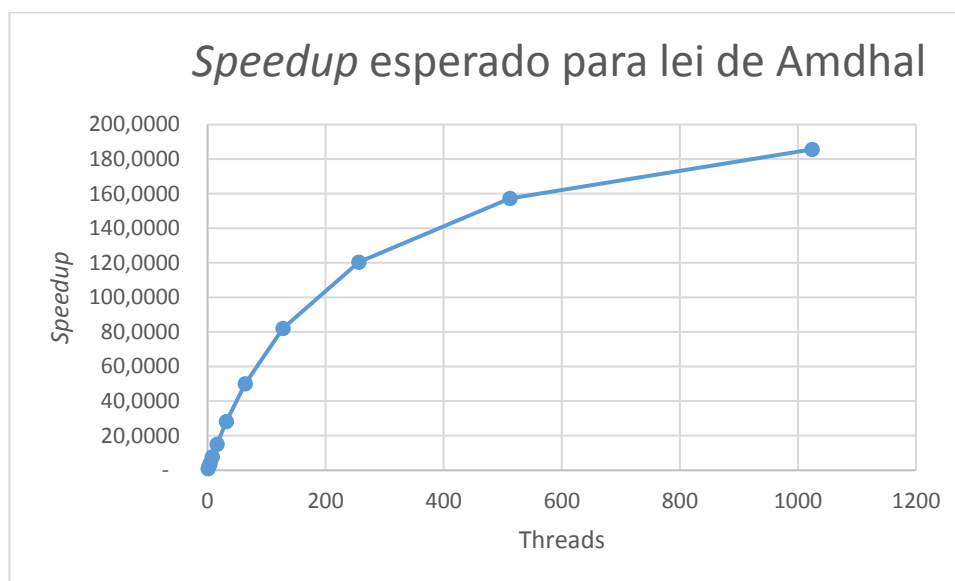
$$Speedup = \frac{1}{0,0044 + (0,9956/N)} \quad (14)$$

Deste modo, para o aumento do número processadores, o *speedup* esperado de acordo com a Lei de Amdahl é apresentado na Tabela 10 e a Figura 22 expõe os valores desta em um gráfico.

**Tabela 10.** Ganho de desempenho esperado de acordo com a Lei de Amdahl

| Lei de Amdahl |          |
|---------------|----------|
| Nº de threads | Speedup  |
| 1             | 1,0000   |
| 2             | 1,9912   |
| 4             | 3,9477   |
| 8             | 7,7601   |
| 16            | 15,0059  |
| 32            | 28,1466  |
| 64            | 50,0693  |
| 128           | 82,0053  |
| 256           | 120,4046 |
| 512           | 157,2121 |
| 1024          | 185,5775 |

**Fonte:** Elaborado pela autora

**Figura 22.** Gráfico do *speedup* previsto segundo a Lei de Amdahl para o aumento de núcleos.

**Fonte:** Elaborado pela autora

Conforme apresentado na Tabela 10 e exposto na Figura 22, observa-se uma expectativa alta de ganho de desempenho uma vez que para 1024 processos é esperado um *speedup* de 185,6.

Como discutido na seção 2.3.6, Amdahl não leva em consideração o tamanho do problema, ao contrário da Lei de Gustafson. Para esta, o desempenho para o problema em questão é expresso pela Equação 15:

$$Speedup = N - 0,44 \cdot (N - 1) \quad (15)$$

A Tabela 11 e a Figura 23 apresentam os resultados para o ganho de desempenho esperado de acordo com a Lei de Gustafson para o trabalho:

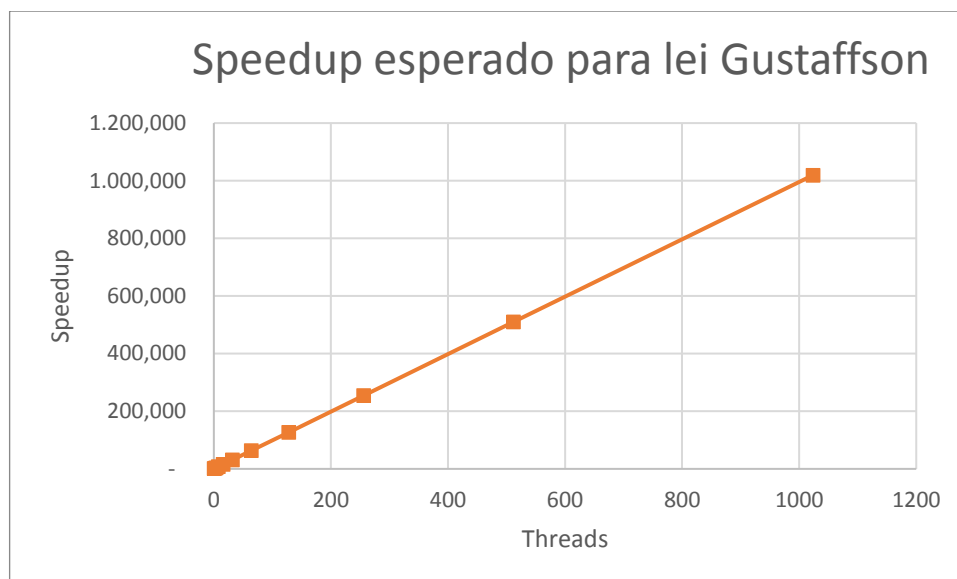
**Tabela 11.** Ganho de desempenho esperado de acordo com a Lei de Gustafson

| Lei de Gustafson |           |
|------------------|-----------|
| Nº de threads    | Speedup   |
| 1                | 1,000     |
| 2                | 1,996     |
| 4                | 3,987     |
| 8                | 7,969     |
| 16               | 15,934    |
| 32               | 31,863    |
| 64               | 63,722    |
| 128              | 127,439   |
| 256              | 254,874   |
| 512              | 509,743   |
| 1024             | 1.019,482 |

**Fonte:** Elaborado pela autora

Conforme a Tabela 11 e a Figura 23, o aumento de desempenho de acordo com Gustafson será vantajosa na medida em que se adiciona mais núcleos de processamento para uma carga maior de dados, apresentando ganhos significativos para quantidades maiores de *threads*, isto é, para 1024 *threads* espera-se um *speedup* de 1.019,482 para problemas cada vez maiores.

O tempo para a análise e classificação dos elementos para o trabalho desenvolvido é descrito na Tabela 12.

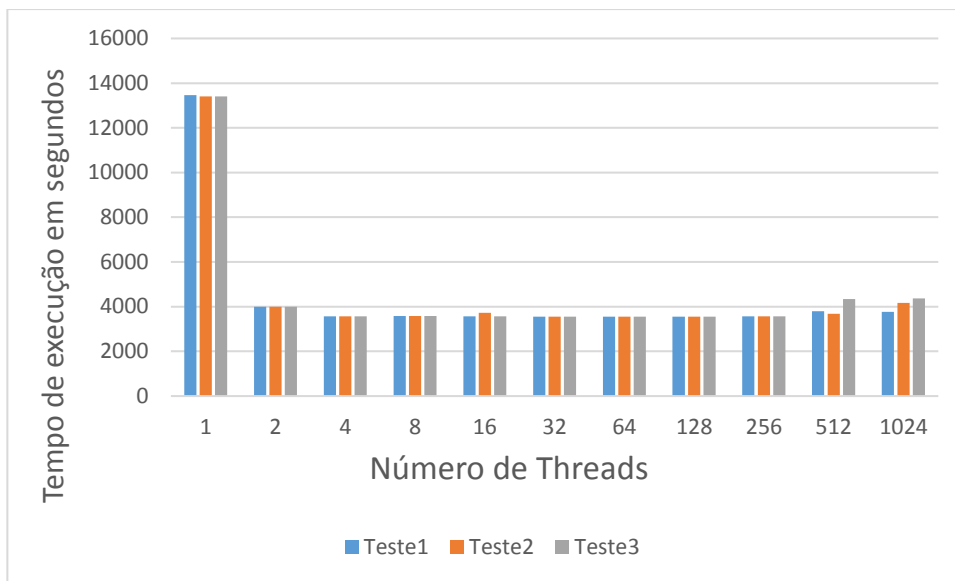
**Figura 23.** Gráfico do *speedup* esperado para a Lei de Gustafson com o aumento de núcleos.**Fonte:** Elaborado pela autora**Tabela 12.** Tempo em segundos para classificar 11826 pontos para o aumento de *threads*.

| Controle de tempo com 11826 elementos |       |         |       |         |       |         |
|---------------------------------------|-------|---------|-------|---------|-------|---------|
| Nº de threads                         | Tempo | Speedup | Tempo | Speedup | Tempo | Speedup |
| 1                                     | 13462 | -       | 13404 | -       | 13402 | -       |
| 2                                     | 4001  | 3,3647  | 3998  | 3,3527  | 4002  | 3,3488  |
| 4                                     | 3567  | 3,7740  | 3566  | 3,7588  | 3564  | 3,7604  |
| 8                                     | 3575  | 3,7656  | 3575  | 3,7494  | 3576  | 3,7478  |
| 16                                    | 3562  | 3,7793  | 3718  | 3,6052  | 3563  | 3,7614  |
| 32                                    | 3558  | 3,7836  | 3558  | 3,7673  | 3559  | 3,7657  |
| 64                                    | 3553  | 3,7889  | 3552  | 3,7736  | 3553  | 3,7720  |
| 128                                   | 3547  | 3,7953  | 3555  | 3,7705  | 3549  | 3,7763  |
| 256                                   | 3572  | 3,7688  | 3573  | 3,7515  | 3572  | 3,7520  |
| 512                                   | 3802  | 3,5408  | 3687  | 3,6355  | 4344  | 3,0852  |
| 1024                                  | 3763  | 3,5775  | 4174  | 3,2113  | 4376  | 3,0626  |

**Fonte:** Elaborado pela autora

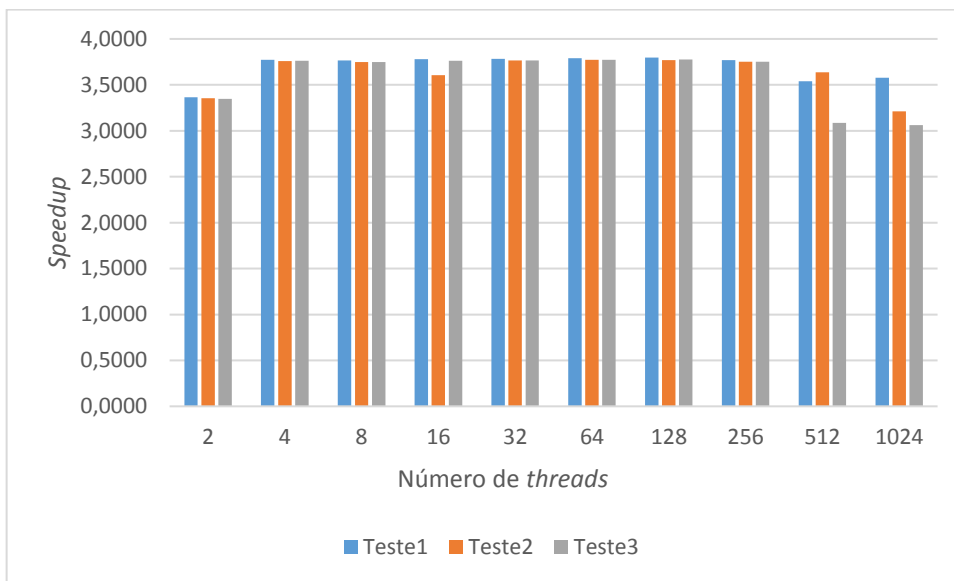
Observa-se a diminuição de tempo ao comparar qualquer resultado paralelo com o monoprocesso, chegando a um *speedup* médio de 3,7542. Outro parecer sobre a Tabela 12 é a estabilização do ganho após quatro *threads* (Figura 24), além de uma queda de desempenho ao atingir 512 *threads* (Figura 25).

**Figura 24.** Comparativo do tempo em segundos com a quantidade de *threads*.



**Fonte:** Elaborado pela autora

**Figura 25.** Comparativo do *speedup* com a quantidade de *threads*.

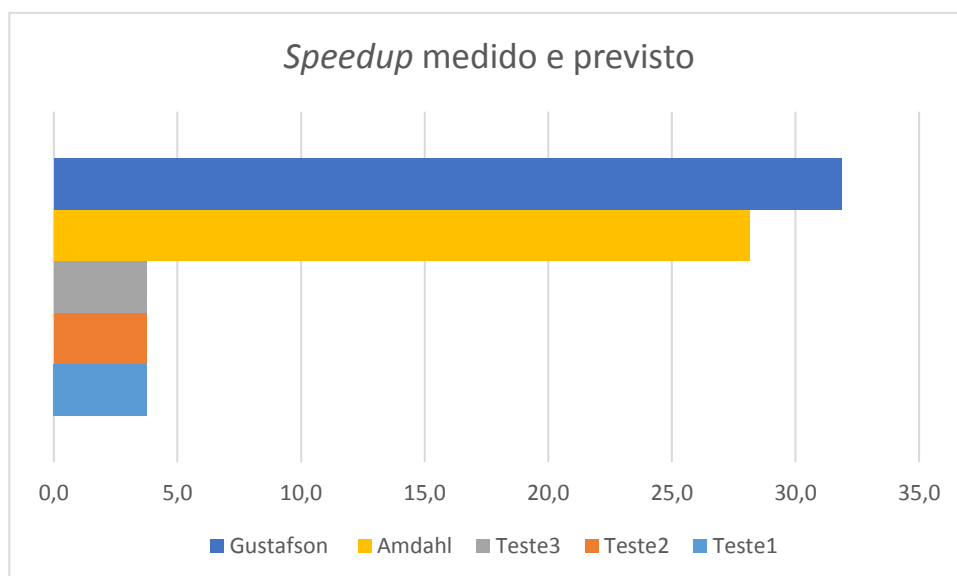


**Fonte:** Elaborado pela autora

Como apresentado pelas figuras 24 e 25, para uma base de dados com 11826 elementos, a adição de mais recursos computacionais a partir de 8 *threads* deixa de ser vantajosa. Conforme o modelo dos processadores i7, que possuem 4 núcleos de processamento e, conforme as instruções do fabricante, 8 *threads* são suportados para cada processador, para o cluster era esperado um ambiente paralelo de 40 *threads* no total. A seguir, a Figura 26

apresenta a comparação de desempenho para o *speedup* previsto para 32 *threads* e o medido para 8 *threads* em cada processador.

**Figura 26.** Comparativo do *speedup* medido e previsto.



**Fonte:** Elaborado pela autora

Para o desempenho aguardado para as duas leis, o *speedup* medido alcançou 13,34% do previsto por Amdahl e 11,78% por Gustafson. Apesar pela limitação causada pela sincronização e comunicação entre os nós e os *threads*, foi possível acelerar a análise e classificação do conjunto de dados.

### 4.3 Considerações finais

Neste capítulo foram apresentados os resultados obtidos para o trabalho desenvolvido, para uma versão paralela híbrida do algoritmo DIANA. Mediu-se o *speedup* deste com os teóricos esperados de acordo as leis de Amdahl e Gustafson e, embora o resultado tenha ficado abaixo do previsto, estas leis não levam em consideração possíveis gargalos na comunicação entre os processadores. Ainda assim o comparativo é importante por enriquecer a pesquisa com outros métodos e valores, ajudando o desenvolvedor a entender os resultados obtidos.

O trabalho apresentou vantagens de uso devido seu ganho de desempenho para analisar e classificar 11826 elementos, gerando um *speedup* médio de 3,7542, ou seja, a versão concorrente é em média 3,7 vezes mais rápida que a versão sequencial do algoritmo.

Para esta pesquisa, neste ambiente estudado, o paralelismo é conveniente até oito *threads*. Isso provém das limitações físicas das máquinas pertencentes ao *cluster*, as quais são capazes de gerenciar até 8 *threads*.

## Capítulo 5 - Conclusões e trabalhos futuros

### 5.1 Conclusões

Como parte deste trabalho, o algoritmo DIANA foi otimizado utilizando técnicas de aprimoramento de código estudadas em compiladores e então paralelizado com a ferramenta OpenMP e MPI. Tal como apresentado pela Tabela 12 e pelas figuras 25 e 26, o algoritmo apresentou um aumento de performance, chegando a ser 3,7542 vezes mais rápido que a versão monoprocessada do programa.

Para a lei de Amdahl esperava-se uma aceleração próxima dos 28,1466 para 32 *threads* rodando simultaneamente no *cluster*. Já pela lei de Gustafson, era previsto um ganho de desempenho quase linear, esperando o *speedup* de 31,863 para 32 *threads* simultaneamente no ambiente. Para este mesmo número de processos, o algoritmo desenvolvido teve aceleração média igual a 3,7542.

A diferença de valores entre o alcançado e o esperado maiores números de *threads* se dá pelo gargalo formado devido as trocas de mensagens entre os processos durante a análise, os quais não são levados em consideração nas equações referentes as leis de Amdahl e Gustafson além do gerenciamento do alto grau de paralelismo, os quais os processadores não estão preparados fisicamente para administrar. Ainda assim é importante a comparação dos *speedup* teóricos com o medido para o enriquecimento da pesquisa.

Conforme a discussão feita na seção “2.3.6 Análise e Otimização de Desempenho”, se há troca de mensagens entre os nós do *cluster*, o tempo de processamento dos dados deve compensar o tempo perdido com a comunicação. No entanto, aumentar o grau do paralelismo para compensar esse gasto não é sempre vantajoso, pois se a velocidade para que os dados sejam entregues para os processadores for pequena, o aumento do grau de paralelismo irá gerar ociosidade, e como consequência, queda de desempenho.

Deste modo, conclui-se que é oportuno o uso de paralelismo para o algoritmo DIANA, uma vez que a versão híbrida apresentou ser 3,7542 vezes mais veloz que a versão sequencial.

### 5.2 Contribuições

Ao final desta pesquisa, as seguintes contribuições podem ser listadas:

- Favorecer o uso do algoritmo DIANA mediante à sua aceleração;
- Facilitação na análise de conjuntos dados mais volumosos e complexos;

- Exploração acadêmica de um algoritmo pouco estudado;
- Contribuições acadêmicas para pesquisas que combinam aprendizado de máquina com computação de alto desempenho;
- Aumento de desempenho do algoritmo através do aumento do *speedup*, contribuindo para a análise e classificação de dados dentro de um tempo realista;
- Facilitação para aplicação do algoritmo em problemas reais.

### 5.3 Trabalhos futuros

Dentre as sugestões de trabalhos futuros, podem-se destacar:

- Desenvolvimento de uma versão híbrida em CUDA e MPI a fins de comparação de desempenho;
- Foco em técnicas para controle e contenção de gargalos em comunicação;
- Testes em clusters formados por computadores capazes de trabalhar com alto grau de paralelismo ou com um número maior de nós;
- Coleta de dados em supercomputadores;
- Análise de desempenho com ferramentas de paralelismo como Intel Parallel Studio XE 2018;
- Desenvolvimento e comparação de desempenho das versões *Single Linkage*, *Average Linkage* e *Centroid Linkage* para o algoritmo DIANA.

## Referências Bibliográficas

- AHO, Alfred V. et al. **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. [s.l.]: Pearson, 2007. 634 p.
- BELL, J. **Machine Learning: Hands-On for Developers and Technical Professionals**. Indianapolis, Indiana: John Wiley & Sons, 2015. 380 p.
- BHIMANI, J.; LEESER, M.; MI, N. **Accelerating K-Means Clustering with Parallel Implementations and GPU computing**. In: High Performance Extreme Computing Conference (HPEC), [S.l.]: IEEE, 2015. Disponível em: <<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7322467>>. Acesso em: 27 out. 2016.
- BLAIR-CHAPPELL, S; STOKES, A. **Parallel Programming with Intel Parallel Studio XE**. [S.l.]: Wrox, 2012. 552 p.
- COVÕES, T. F. **Algoritmos evolutivos para modelos de mistura de gaussianas em problemas com e sem restrições**. 2015. 138 p. (Ciências da Computação e Matemática) - Instituto de ciências da matemática e de computação, Universidade de São Paulo, São Carlos, 2015
- DANALIS, A.; MCCURDY, C.; VETTER, J. S. **Efficient Quality Threshold Clustering for Parallel Architectures**. 2012. 12 p. [S.l.], 2012.
- EIJKHOUT, V. **Introduction to High Performance Scientific Computing**. 2. ed. [S.l.: s.n.], 2015. 534 p.
- GEBALI, F. **Algorithms and Parallel Computing**. [S.l.]: John Wiley & Sons, 2011. 341 p.
- JAIN, A. K.; MURTY, M. N.; FLYNN, P. J. **Data clustering: a review**. ACM Computing Surveys, New York, v. 31, n. 3, p. 265-323, Set., 1999.
- JAMES, G. et al. **An Introduction to Statistical Learning: with Applications in R**. New York: Springer, 2013. 426 p.
- JOHNSON, S.C. HIERARCHICAL CLUSTERING SCHEMES. **Psychometrika**. [S.l.], p. 241-254. set. 1967
- JR., F. L. **Efficient Variations of the Quality Threshold Clustering Algorithm**. 2015. 51 p. (Doutorado em Sistemas de informação computacional) - College of Engineering and Computing, Nova Southeastern University, [S.l.], 2015.
- KAUFMAN, L.; ROUSSEEUW, P. J. **Finding Groups in Data: An Introduction to Cluster Analysis**. [S.l.]: John Wiley & Sons, 1990. 355 p.

LOPES, N.; RIBEIRO, B. **Machine Learning for Adaptive Many-Core Machines: A Practical Approach**. [S.l.]: Springer, 2015. 241 p. v. 7.

MANYIKA, J. et al. **Big data: The next frontier for innovation, competition, and productivity**. [S.l.]: McKinsey & Company's, 2011. 143 p.

NAIK, DS. B.; KUMAR, S. D.; RAMAKRISHNA, S.V. **Parallel Processing Of Enhanced K-Means Using OpenMP**. 2013.4 p. Department of Computer Science and Engineering, Vignan University, Andhra Pradesh, India, 2014.

NVIDIA CORPORATION (Org.). **CUDA C Programming Guide**. [S.l.: s.n.], 2016. 260p. Disponível em: <[http://docs.nvidia.com/cuda/pdf/CUDA\\_C\\_Programming\\_Guide.pdf](http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf)>. Acesso em: 30 out. 2016.

PACHECO, P. S. **An Introduction to Parallel Programming**. Burlington, MA, USA: Morgan Kaufmann Publishers, 2011. 370 p.

PARHAMI, B. **Introduction to Parallel Processing: Algorithms and Architectures**. [S.l.]: Kluwer Academic Publishers, 2002. 532 p.

PUSTEJOVSKY, J.; STUBBS, A. **Natural Language Annotation for Machine Learning**. 1. ed. Sebastopol, CA: O'Reilly, 2012. 326 p.

RAHMAN, Rezaul. **Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers**. [S. l.]: Apress, 2013. 209 p.

WITTEN, I. H.; FRANK, E. **Data mining: Practical machine learning tools and techniques**. 2. ed. [S.l.]: Morgan Kaufmann Publishers, 2005. 525 p.

WU, X.; KUMAR, V. **The top ten algorithms in data mining**. [S.l.]: CRC Press, 2009. 215 p.

## Bibliografia

BLUM, A. **Machine Learning Theory**. 2003. 4 p. Carnegie Mellon University, [S.l.], [200?].

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed systems: Concepts and Design**. 4. ed. [S.l.]: Addison Wesley, 2005. 927 p.

FUJIMOTO, R. M. **Parallel and distributed simulation systems**. [S.l.]: John Wiley & Sons, 2000. 300 p.

GRAMA, A. et al. **Introduction to Parallel Computing**. 2. ed. [S.l.]: Pearson/Addison Wesley, 2003. 636 p.

HAN, J.; KAMBER, M. **Data mining: Concepts and Techniques**. [S.l.]: Morgan Kaufmann Publishers, 2001. 550 p.

MITCHELL, T. **M. Machine Learning**. [S.l.]: McGraw-Hill Science/Engineering/Math, 1997. 432 p.

PEDRYCZ, W. **Knowledge-based clustering: From data to information granules**. [S.l.]: John Wiley & Sons, 2005. 313 p.

ZOMAYA, A. Y.; ERCAL, F.; OLARIU, S. **Solutions to parallel and distributed computing problems: Lessons from biological sciences**. [S.l.]: John Wiley & Sons, 2001. 272 p.

## APÊNDICE A – Relação de artigos encontrados sobre paralelização de algoritmos de clusterização usando MPI ou OpenMP

Foram realizadas pesquisas em três bibliotecas digitais: IEEE Xplore, ScienceDirect e ACM Digital Library. As buscas procuravam artigos que paralelizaram algoritmos de clusterização em MPI ou OpenMP entre 2014 e 2018. Em alguns casos foi necessário aumentar o intervalo do campo “Ano” com a finalidade de trazer mais artigos, resultando em um artigo sobre o *Quality Threshold Clustering*. Outros algoritmos são mais populares dado a sua simplicidade na implementação e por ser relativamente rápido, como o K-means (WU; KUMAR, 2009).

| Algoritmo                                           | Quantidade                              | Ano          |
|-----------------------------------------------------|-----------------------------------------|--------------|
| QT ( <i>Quality Threshold Clustering</i> )          | IEEE: 1;<br>ACM: 0;<br>ScienceDirect: 0 | 2012;        |
| K-Means                                             | IEEE: 4;<br>ACM: 1;<br>ScienceDirect: 2 | 2014 - 2018; |
| Algoritmos Hierárquicos                             | IEEE: 0;<br>ACM: 0;<br>ScienceDirect: 0 | 2014 – 2018  |
| <i>Clustering as a Mixture of Gaussians</i><br>(EM) | IEEE: 0;<br>ACM: 0;<br>ScienceDirect: 0 | 2014-2018    |
| Fuzzy C-means                                       | IEEE: 0;<br>ACM: 0;<br>ScienceDirect: 0 | 2014 – 2016  |
| DBSCAN                                              | IEEE: 1;<br>ACM: 0;<br>ScienceDirect: 0 | 2014-2018    |

**Fonte:** Elaborado pela autora

Como pôde ser observado na tabela, poucos trabalhos foram encontrados nas três bases, totalizando apenas nove artigos. Isso salienta que, embora existam ganhos de desempenho com a paralelização, esse campo ainda é pouco estudado, focando apenas nos algoritmos mais simples e fáceis de implementar, como o K-means.

## ANEXO A – Localizações dos usuários do aplicativo MOPSI na Finlândia até 2012



## TERMO DE REPRODUÇÃO XEROGRÁFICA

Autorizo a reprodução xerográfica do presente Trabalho de Conclusão, na íntegra ou em partes, para fins de pesquisa.

São José do Rio Preto, \_\_\_\_/\_\_\_\_/\_\_\_\_

---

Assinatura do autor