PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Informática

Guilherme Torres Castro

PROPOSTA E AVALIAÇÃO DO ALGORITMO K-MODES EM ARQUITETURAS PARALELAS PARA AGRUPAMENTO DE SEQUÊNCIAS BIOLÓGICAS - UMA ANÁLISE NO CONTEXTO DE PREDIÇÃO DE SÍTIO DE INÍCIO DE TRADUÇÃO DE PROTEÍNAS

Belo Horizonte

2017

Guilherme Torres Castro

PROPOSTA E AVALIAÇÃO DO ALGORITMO K-MODES EM ARQUITETURAS PARALELAS PARA AGRUPAMENTO DE SEQUÊNCIAS BIOLÓGICAS - UMA ANÁLISE NO CONTEXTO DE PREDIÇÃO DE SÍTIO DE INÍCIO DE TRADUÇÃO DE PROTEÍNAS

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Informática.

Orientador: Prof. Dr. Henrique Cota de Freitas

Coorientadora: Prof. Dra. Cristiane Neri Nobre

Belo Horizonte

2017

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Castro, Guilherme Torres
C355p Proposta e avaliação do algoritmo k-modes em arquiteturas paralelas para agrupamento de sequências biológicas – uma análise no contexto de predição de sitio de inicio de tradução de proteínas / Guilherme Torres Castro. Belo Horizonte, 2017.
69 f.: il.

Orientador: Henrique Cota de Freitas
Coorientadora: Cristiane Neri Nobre
Dissertação (Mestrado) - Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática

1. Arquitetura de computador. 2. Processamento paralelo (Computadores). 3. Linux (Sistema operacional de computador). 4. Cluster (Sistema de computador). 5. Algorítmos computacionais. I. Freitas, Henrique Cota de. II. Nobre, Cristiane Neri. III. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática. IV. Título.
SIB PUC MINAS
CDU: 681.3-11

Guilherme Torres Castro

PROPOSTA E AVALIAÇÃO DO ALGORITMO K-MODES EM ARQUITETURAS PARALELAS PARA AGRUPAMENTO DE SEQUÊNCIAS BIOLÓGICAS - UMA ANÁLISE NO CONTEXTO DE PREDIÇÃO DE SÍTIO DE INÍCIO DE TRADUÇÃO DE PROTEÍNAS

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Informática.

Prof. Dr. Henrique Cota de Freitas – PUC Minas (Orientador)

Prof. Dra. Cristiane Neri Nobre – PUC Minas (Coorientadora)

Prof. Dr Luis Enrique Zárate Galvez – PUC Minas

Prof. Dr. José Miguel Ortega – UFMG

Belo Horizonte, 05 de junho de 2017.

AGRADECIMENTOS

Ao Programa de Pós-graduação em Informática da PUC Minas, pelo apoio dos professores e funcionários para o desenvolvimento do projeto.

Agradecimento à FAPEMIG e CNPq pelos recursos financiados para a execução do projeto.

Pesquisa desenvolvida com o apoio do Centro Nacional de Supercomputação (CESUP), da Universidade Federal do Rio Grande do Sul (UFRGS).

“Nós só podemos ver um pouco do futuro, mas o suficiente para perceber que há muito a fazer.”

Alan Turing

RESUMO

A Predição de sítio de início de tradução (SIT) é de vital importância para bioinformática. É pelo uso desse processo que se torna possível entender a formação orgânica e metabólica de todos os organismos. Entretanto, cada molécula de mRNA pode ter muitos AUGs, dentre esses, apenas um deles é o SIT e todos os outros não são SIT. Dessa forma, o custo computacional envolvido para balancear sequências não-SIT, usados no algoritmos para predição de sítio de início de tradução, é muito alto.

Sendo assim, esta dissertação apresenta um versão otimizada do algoritmo *K-modes* para clusterização de sequências de mRNA, e compara com sua versão tradicional, o *K-means*. A solução apresentada faz uso de instruções mais simples e menos recursos computacionais para reduzir o tempo de execução, sem comprometer a qualidade dos resultados de clusterização.

Da versão sequencial do algoritmo *K-modes*, também foram implementadas duas versões para computação heterogênea, uma para GPU com uso de CUDA e a outra para CPU (XeonPhi) com uso de OpenMP. Essas versões foram comparadas com uma versão do algoritmo em MPI/OpenMP que foi executada em um cluster com nove nós *quad-core* e também um cluster formado por Raspberry Pi.

A versão do *K-modes* apresentada nessa dissertação reduziu o tempo de processamento para todas as bases em relação ao *K-means*, chegando, em alguns casos, a ser mais de 73 vezes mais rápida. As versões em paralelo do algoritmo conseguiram reduzir ainda mais o tempo. A versão em GPU apresentou um desempenho até 145 vezes mais rápida em relação ao *K-modes* sequencial, totalizando um *speedup* de mais 10000 vezes em relação ao algoritmo *K-means* original. As outras versões em paralelo também apresentaram um ganho de desempenho em relação a versão sequencial, com a solução para o Raspberry Pi apresentando o pior desempenho computacional (1,4 vezes mais rápido que a versão sequencial), mas em contrapartida apresentou a melhor eficiência energética.

Palavras-chave: K-modes, K-means, Clusterização, Computação em Paralelo, Sequencias de Nucleotídeos, Sítio de Início de Tradução

ABSTRACT

Prediction of the Translation Initiation Site (TIS) is an important problem of molecular biology. Based on this process it is possible to understand the organic and metabolic formation of all organisms. However, each mRNA molecule can have many AUGs, but only one is TIS and all others are non TISs. However, the computational cost for balancing non TISs sequences, used in algorithms to make predictions of initiation sites, is very high.

Thus, this dissertation presents an optimized version of the *K-modes* algorithm for clustering mRNA sequences, and compares it with its traditional version, *K-means*. The proposed version in this dissertation makes use of simpler instructions and fewer computational resources to reduce execution time without compromising the quality of clustering results.

From the sequential version of the K-modes algorithm, two versions for heterogeneous computing were also implemented, one for GPU using CUDA and the other one for CPU (XeonPhi) using OpenMP. Besides, a version of the algorithm in MPI/OpenMP were executed in a cluster with nine quad-core nodes and also a cluster based on Raspberry Pi.

The *K-modes* version presented in this dissertation reduced the processing time for all bases in comparison to *K-means*, in some cases it is 73 times faster. The parallel versions reduced the execution time, mainly the GPU version that achieved a speedup by up to 145 times than sequential *K-modes* version, totaling a speedup of more than 10,000 times in relation to the original *K-means* algorithm. The other parallel versions also showed a performance gain compared to the sequential version, with the solution for Raspberry Pi presenting the worst computational performance (1.4 times faster than the sequential version), but in contrast it is the most energy efficient.

Keywords: K-modes, K-means, Clustering, Parallel Computing, Nucleotides Sequences, Translation Initiation Site

LISTA DE FIGURAS

FIGURA 1 – Modelo de escaneamento da fita de RNA. 30

FIGURA 2 – Consenso de Kozak. 30

FIGURA 3 – Arquitetura de uma GPU. 33

FIGURA 4 – Coprocessador Xeon Phi - Comunicação. 34

FIGURA 5 – Modelo híbrido de computação paralela utilizando MPI/OpenMP. . . . 37

FIGURA 6 – Partição do conjunto em treino e teste para validação cruzada. 46

FIGURA 7 – Exemplo de cálculo de centroide 51

FIGURA 8 – Comparação do desempenho computacional, em segundos, para bases pequenas. 58

FIGURA 9 – Comparação do desempenho computacional, em minutos para bases grandes. 60

FIGURA 10 – Taxa de convergência (quantidade de sequências) no processo de clusterização das bases grandes. 62

FIGURA 11 – Comparação da média energética, por organismo (em watt). 63

LISTA DE TABELAS

TABELA 1 – Relação de trabalhos correlatos referentes à paralelização do algoritmo *k-means* e *k-modes* 42

TABELA 2 – Sequências extraídas 45

TABELA 3 – Distância entre sequências. 49

TABELA 4 – Desempenho computacional, em segundos. 55

TABELA 5 – Resultado das métricas de avaliação utilizadas, em porcentagem. 56

TABELA 6 – Comparação das métricas de avaliação obtidas por Rodrigues et al. (2012) e por essa dissertação 57

TABELA 7 – Número total de iterações, por organismo, para o *k-modes* e *k-means* . 61

TABELA 8 – Consumo total de energia em joules. 64

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface*

CDS *CoDing Sequence*

CPU *Central Processing Unit*

CUDA *Compute Unified Device Architecture*

DNA *Deoxyribonucleic Acid*

GPU *Graphics Processing Unit*

IP *Internet Protocol*

MIC *Many Integrated Core Architecture*

MPI *Message Passing Interface*

OpenCL *Open Computing Language*

OpenMP *Open Multi-Processing*

RefSeq *Reference Sequence - Sequências de referência*

RNA *Ribonucleic acid*

SIMD *Single instruction, multiple data*

SIT *Sítio de Início de Tradução*

SM *Streaming Processor*

SVM *Support Vector Machine*

TCP *Transmission Control Protocol*

SUMÁRIO

1 INTRODUÇÃO **25**
1.1 Problema 27
1.2 Hipóteses 27
1.3 Objetivos 28
1.3.1 Objetivo geral 28
1.3.2 Objetivos específicos 28
1.4 Organização da dissertação 28

2 REVISÃO DA LITERATURA **30**
2.1 Sítio de Início de Tradução 30
2.2 *K-means* e *k-modes* 31
2.3 Computação paralela e distribuída 32
2.3.1 Arquitetura de GPUs e CUDA 32
2.3.2 Intel Xeon Phi - Many Integrated Core Architecture (MIC) 34
2.3.3 OpenMP 35
2.3.4 MPI 36
2.3.5 MPI/OpenMP 36
2.3.6 Raspberry Pi 38
2.4 Trabalhos correlatos 38
2.4.1 Predição de SIT 38
2.4.2 K-modes 39
2.4.3 Paralelização usando computação heterogênea 40
2.4.4 Paralelização dos algoritmos k-means e k-modes 40

3 METODOLOGIA **43**
3.1 Abordagens de paralelização 43
3.2 Ambiente de teste 44
3.3 Métricas para Avaliação de Qualidade e Desempenho 47

4 K-MODES **49**
4.1 *k-modes* em GPU 52
4.2 *k-modes* Híbrido 53
4.3 *k-modes* em XeonPhi 54

5 RESULTADOS E DISCUSSÕES **55**
5.1 Apresentação dos resultados 55
5.2 Comparação de resultados com trabalho correlato do grupo 56
5.3 Comparação de desempenho com trabalho correlato do grupo 57
5.3.1 Convergência 61
5.4 Eficiência energética 62

6 CONCLUSÕES E TRABALHOS FUTUROS **65**

REFERÊNCIAS **66**

1 INTRODUÇÃO

A bioinformática surgiu com o objetivo de mapear sequências genéticas, devido a capacidade de análise, armazenamento e apresentação de grandes volumes de dados biológicos por meios computacionais (PROSDOCIMI, 2007). Estes dados são compostos principalmente de sequências de DNA, RNA e proteínas. Um dos problemas de vital importância para bioinformática, envolvendo a análise de sequências de mRNA, é a predição de sítios de início de tradução (SIT). É através desse processo é possível entender a formação orgânica e metabólica de todos os organismos.

Apesar de toda sequência de mRNA ser transcrita, somente parte da sequência do mRNA carrega informações para codificar proteínas (CDS – *CoDing Sequence*).

O início da tradução de proteínas acontece no momento em que o ribossomo se conecta ao mRNA e inicia a leitura dos códons até encontrar o códon AUG (metionina) com contexto adequado para o início do processo. Esta região é conhecida como sítio de início de tradução (SIT). A partir do momento em que o AUG é encontrado, o ribossomo começa a expressar os códons em proteínas na forma de uma cadeia peptídica. Quando um *stop códon* (UAG, UAA ou UGA) é encontrado, o processo de tradução de proteína é finalizado.

Segundo KOZAK (1984), normalmente a tradução tem início no primeiro códon AUG da sequência. Entretanto, o início de tradução também pode ser iniciado em um códon diferente (HATZIGEORGIOU, 2002). Como não se conhece características determinísticas no processo de identificação de início da tradução, o processo de predição de SIT se torna uma tarefa complexa (SILVA et al., 2011).

A computação pode ser usada para predição de regiões de SIT por meio de aprendizado de máquina, isto é feito através de conhecimento previamente adquirido de outras sequências nas quais possuem esta informação disponível, formando um modelo de classificação capaz de prever se um AUG apresenta o contexto adequado para começar a tradução da proteína.

Entretanto, cada molécula de mRNA pode ter muitos AUGs. Dentre estes, apenas um deles é o SIT e todos os outros não são SIT. Dessa forma, o custo computacional envolvido para balancear sequências não-SIT, usados no algoritmos para predição de sítio de início de tradução, é muito alto.

Para solucionar o problema de desbalanceamento, são usadas técnicas de *oversampling* e *undersampling*. Na técnica de *oversampling* a classe minoritária é replicada sinteticamente. Para o problema de predição de SIT, esta técnica apresenta elevado custo computacional, como demonstrado por Nobre (2007). A técnica de *undersampling* foi

usada por Silva et al. (2011) e Rodrigues et al. (2012), em que é extraída da classe majoritária uma amostra de tamanho similar ao da classe minoritária.

Para implementação do método de balanceamento, Rodrigues et al. (2012) utilizaram o algoritmo *k-means* (MACQUEEN, 1967) construindo grupos de elementos mais homogêneos possíveis e retirando o elemento mais representativo de cada *cluster* formado. Tendo como resultado dois conjuntos balanceados para serem usados como modelo de classificação para identificação de sequências SITs e não-SITs.

Porém, o custo computacional envolvido para balancear sequências não-SIT, usados no algoritmos para predição de sítio de início de tradução é muito alto. E ainda mais, os aprimoramentos de dispositivos individuais de hardware que antes eram capazes de aumentar o desempenho de aplicativos sequenciais de maneira satisfatória, foi interrompido devido aos limites encontrados em relação ao gerenciamento de energia, velocidade de acesso à memória e paralelismo em nível de instrução (ASANOVIC et al., 2006). Esses limites indicam que o crescimento de desempenho deve ser alcançado pelo aumento de sistemas paralelos. Sendo assim, a computação paralela e distribuída tem como meta reduzir o tempo de processamento da aplicação, por meio de um melhor aproveitamento dos recursos de máquina (JONES et al., 2009).

Para treinar e classificar a sequência utiliza-se a *Support Vector Machine* (SVM), proposta por (CORTES; VAPNIK, 1995). Este método baseia-se em técnicas de aprendizado de máquina a partir de métodos estatísticos, recebendo como entrada o conjunto de dados a serem treinados e construindo um modelo de predição capaz de determinar em qual classe um objeto qualquer desse domínio deve ser atribuído.

Neste trabalho foi utilizado como entrada para SVM o conjunto de sequências SITs e o conjunto de sequências não-SIT extraídas da clusterização. Após o treinamento, a SVM é capaz de identificar se uma sequência é ou não SIT.

O custo de balanceamento de sequências é muito grande devido ao tamanho do volume de sequências não-SIT extraídas, associado com o grande esforço computacional envolvido no cálculo das distâncias euclidianas utilizadas no algoritmo *k-means*. Neste trabalho, foi utilizado o algoritmo *k-modes* (HUANG, 1998) a fim de reduzir o custo das operações envolvidas no cálculo das distância, usando a diferença de similaridade entre os objetos como distância e trocando a medida de médias por modas.

Ainda assim, o grande volume de sequências ainda é um problema para programas sequenciais. Por isso, o uso de recursos disponíveis no campo computacional para reduzir o tempo de processamento é importante. Usando a computação paralela e distribuída é possível reduzir o tempo de processamento fazendo melhor aproveitamento dos recursos da máquina.

As Unidades de Processamento Gráficos (GPUs), que antes eram usadas apenas para computação gráfica, têm se tornado um componente de valor para resolver uma grande quantidade de problemas matemáticos. A *GPU* é capaz de reduzir o tempo de execução do *k-means* de forma significativa (KIJSIPONGSE; U-RUEKOLAN, 2012).

Processadores *multi-core* já são amplamente explorados em trabalhos que necessitam um tempo de resposta mais rápido. O aumento de frequência com o objetivo de aprimorar o desempenho dos processadores tem encontrado limites físicos (ASANOVIC et al., 2009), como o alto consumo de energia e dissipação de calor. O aumento dos núcleos tem se tornado a forma mais eficaz de garantir melhor desempenho. Essa tecnologia é tão difundida que chips *multi-core* já são encontrados em praticamente todos os computadores pessoais atuais e em muitos dispositivos como *tablets* e *smartphones*. Computadores *multi-core* conseguem extrair maior desempenho da máquina em trechos de códigos em paralelo.

Outra forma de paralelização de tarefas é o uso de diversas máquinas interconectadas que processam de forma independente a aplicação, sincronizando e trocando informações a partir de mensagens enviadas e recebidas pela rede. Este conjunto de computadores aglomerados são chamados de *cluster*.

Esta dissertação apresenta uma proposta de avaliação do algoritmo *k-modes* para clusterização de sequências não-SIT, usando várias técnicas de processamento paralelo e distribuído, dando continuidade ao trabalho de (RODRIGUES et al., 2012).

1.1 Problema

Cada molécula de mRNA pode ter muitos AUGs, dentre estes, apenas 1 é o SIT e todos os outros não são SITs. O custo para o balanceamento dessas sequências é muito alto para algoritmos sequenciais (RODRIGUES et al., 2012).

Dessa forma, o problema desse artigo pode ser formulado com a seguinte pergunta: Como o tempo de execução do balanceamento entre as sequências SIT e não SIT pode ser reduzido e ainda assim manter a qualidade dos resultados?

1.2 Hipóteses

A hipótese é que a variação do algoritmo *k-means*, conhecida como *k-modes*, possa diminuir os esforços computacionais e o uso de memória, para agrupamento das sequências não-SITs. Uma vez que é possível calcular a distância a partir de operações mais simples, e armazenar as sequências estruturas de dados que ocupam menos memória. Como o algoritmo apresenta pouca dependência de dados, existe a possibilidade de implementar

versões paralelas do mesmo. Este trabalho apresenta a implementação e avaliação do algoritmo *k-modes* em diversas arquiteturas para execução de códigos em paralelo. Dessa forma, será possível extrair desempenho de execução, em diversos contextos de paralelismo. tais como, *threads*, nós, instruções e computação heterogênea.

1.3 Objetivos

1.3.1 Objetivo geral

O objetivo dessa dissertação é propor e avaliar versões do algoritmo *k-modes* em diferentes arquiteturas paralelas para balancear sequências não SIT.

1.3.2 Objetivos específicos

a) Paralelização e avaliação do algoritmo *k-modes* em CUDA para execução em GPUs.
b) Paralelização e avaliação do algoritmo *k-modes* em MPI (passagem de mensagem) e OpenMP (variável compartilhada) para execução em clusters de computadores multi-core.
c) Paralelização e avaliação do algoritmo *k-modes* fazendo uso de vetorização para execução no coprocessador Xeon-Phi.
d) Avaliar o ganho de desempenho e eficiência energética das abordagens paralelas e distribuídas.
e) Avaliar resultados de classificação das bases de organismos processadas.

1.4 Organização da dissertação

Em virtude do que foi descrito, essa dissertação está organizada da seguinte maneira:

O Capítulo 2 apresenta os fundamentos teóricos e trabalhos correlatos das áreas de previsão de SIT e computação paralela de alto desempenho.

O Capítulo 3 apresenta os recursos utilizados para a execução dos testes e o modelo de extração de sequências das bases de genomas para os organismos aqui tratados.

O Capítulo 4 apresenta a metodologia aplicada para a implementação do algoritmo *k-modes*, recursos e técnicas computacionais implementadas para a solução do problema e a descrição das bases utilizadas neste trabalho.

O Capítulo 5 apresenta a análise feita sobre os resultados obtidos pelo processamento das bases em relação ao desempenho computacional, do comportamento da

clusterização e na qualidade dos resultados obtidos a partir da classificação de sequências.

E por fim, o Capítulo 6 discute e conclui sobre os resultados obtidos e trabalhos futuros com base no que foi alcançado neste trabalho. Dessa forma, esse trabalho apresenta uma solução de processamento paralelo e distribuído para a previsão do sítio de início de tradução.

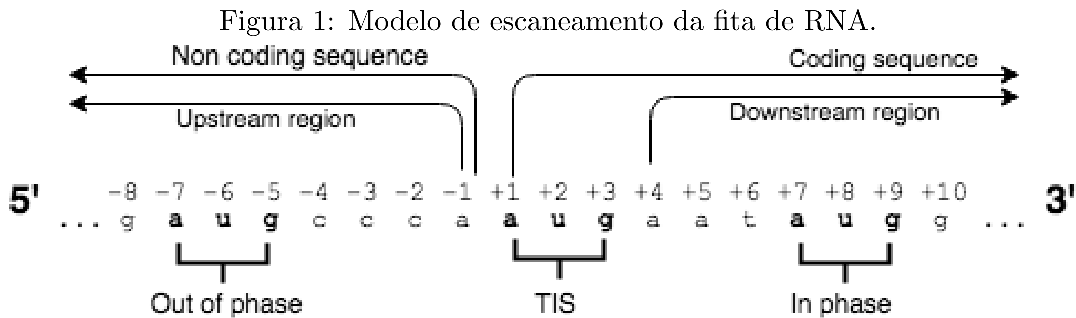
2 REVISÃO DA LITERATURA

Esta seção apresenta os conceitos teóricos do contexto biológico do sítio de início de tradução, o funcionamento do algoritmo *k-means* e sua variante *k-modes*, a arquitetura de GPUs e CUDA, os trabalhos correlatos desenvolvidos para a solução do problema de predição de SIT e outras propostas de paralelização do algoritmo *k-means*.

2.1 Sítio de Início de Tradução

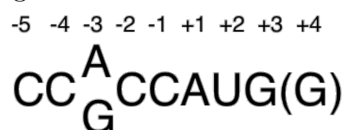
O início de tradução de proteína acontece quando o ribossomo encontra o códon AUG marcado como inicializador de tradução. O escaneamento começa na região 5' da fita do mRNA e percorre em direção a região 3' como ilustrado na Figura 1. Quando encontrado um códon de parada (TAA, TAG ou TGA) a tradução da proteína chega ao fim. Assim, nem todos os nucleotídeos contidos na fita de mRNA são sintetizados. Além disto, cada molécula de mRNA possui apenas um AUG que sinaliza o início de tradução e vários outros que não sintetizam, gerando um problema de desbalanceamento.

Figura 1: Modelo de escaneamento da fita de RNA.



A predição de SIT é um problema já bastante discutido na bioinformática. A primeira tentativa de prever SIT foi feita por (KOZAK, 1984). Nesse estudo, a autora identificou que existem posições relativas ao SIT que são muito conservadoras, tal como a posição -3, apresentando uma purina, nucleotídeo A (Adenina) ou G (Guanina), e as posições -1, -2, -3, -4 e -5 onde um predomínio do nucleotídeo C (Citosina). Este raciocínio determinou o consenso de Kozak, ilustrado na Figura 2.

Figura 2: Consenso de Kozak.



Além da análise estatística realizada por KOZAK (1984), outros métodos para predição de SIT foram propostos baseados em aprendizado de máquina. Um dos primeiros

trabalhos que exploraram o uso de redes neurais foi proposto por Stormo, Schneider e Gold (1982). Outro trabalho de grande importância que também explora o uso de redes neurais foi o de Pedersen e Nielsen (1997). Este trabalho tem grande expressão na área, visto que a base criada é referenciada em outros trabalhos, com o intuito de comparar resultados e validar metodologia (SILVA et al., 2011).

2.2 ***K-means* e *k-modes***

Clusterização é a tentativa de agrupar objetos em grupos, de modo que todos os objetos do mesmo grupo possuam relação natural entre eles, mas ainda assim se diferem (JAIN; MURTY; FLYNN, 1999).

Usando o algoritmo *k-means* é possível agrupar dados em um espaço com d dimensões utilizando características como forma de discriminar o objeto analisado, desde que essas possam ser mapeadas para valores numéricos (MACQUEEN, 1967).

O *k-means* faz essa clusterização criando *clusters*, onde cada *cluster* possui seu centroide formado pela média aritmética dos objetos presentes no *cluster*, e garantindo que todos os pontos pertencentes aquele *cluster* possuem a menor distância possível em relação aos outros centroides. Durante o processo de agrupamento, os valores dos centroides são atualizados a cada iteração.

O *k-modes* é uma extensão do *k-means* que usa dissimilaridade entre os objetos ao invés da distância euclidiana, comumente utilizada no *k-means*. O *k-modes* utiliza também o sistema de frequência (moda) para atualizar os valores dos centroides ao invés da média aritmética (HUANG, 1998).

O pseudocódigo dos algoritmos é ilustrado pelo Algoritmo 1.

Algoritmo 1: Pseudocódigo do algoritmo *k-modes e k-means.*

```
Entrada: Conjuntos de pontos S, Conjuntos de Centroids C
Saída: Clusters
1  Inicialize os centroides C;
2  enquanto HOUVER PONTOS QUE MUDARAM DE CENTROIDE faça
3      Limpe os dados dos centroides temporários tmpC;
4      para cada POINT n EM S faça
5          para cada CENTROID c DO CONJUNTO C faça
6              Calcule a distância entre c e n;
7              se A DISTÂNCIA ENCONTRADA É A MENOR então
8                  Atribua n ao conjunto de centroide c;
9                  Atualize a quantidade de pontos para o centroide c;
10         Atualize a posição do centroide temporário tmpC;
11     para cada CENTROIDE C DO CONJUNTO C faça
12         Posição de c= posição atualizada do centroide tmpC;
```

Neste trabalho, esse algoritmo foi utilizado para o balanceamento de sequências não-SIT. Estas sequências são divididas em k grupos, onde k é o número de sequências SIT. A sequência mais representativa de cada grupo (a que possui a menor distância em relação ao centroide) é selecionada, formando-se um conjunto menor (de mesmo tamanho das sequências SIT) de sequências não-SIT. Este conjunto formará, juntamente com as sequências SIT, o conjunto de treinamento.

2.3 Computação paralela e distribuída

O avanço nos *hardwares* de arquiteturas paralelas também é acompanhado pelo surgimento de ferramentas e bibliotecas que auxiliam o desenvolvimento de *software* em paralelo. Estas ferramentas tem como objetivo extrair o maior desempenho possível dos recursos disponíveis e diminuir a complexidade de implementação de soluções paralelas por meio de abstrações de camadas mais baixas da arquitetura.

Dentre essas, existem ferramentas para o uso em arquiteturas com memória compartilhada como OpenMP (*Open Multi-Processing*), memória distribuída como o MPI (*Message Passing Interface*) e também para arquiteturas heterogênea como GPUs.

A seguir serão apresentados algumas classificações de modelos, arquiteturas e bibliotecas que se inserem no contexto desta dissertação.

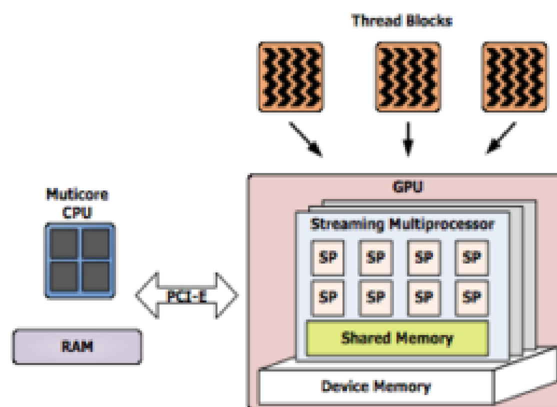
2.3.1 Arquitetura de GPUs e CUDA

Unidades de Processamento Gráficos (GPUs), que antes eram usadas apenas para computação gráfica, têm se tornado um componente de valor para resolver uma grande

quantidade de problemas matemáticos. A GPU é capaz de reduzir o tempo de execução do *k-means* de forma significativa (KIJSIPONGSE; U-RUEKOLAN, 2012).

GPU era inicialmente um *hardware* especializado para cálculos em espaços tridimensionais, consistindo em um grande número de pequenos processadores conhecidos como *Streaming Processor* (SP) que trabalham em paralelo como descrito na Figura 3.

Figura 3: Arquitetura de uma GPU.



Fonte: (KIJSIPONGSE; U-RUEKOLAN, 2012)

Múltiplos *Streaming Processor* são organizados em um *Streaming Multiprocessor* (SM). Cada GPU possui múltiplas SMs para executar *threads* em SIMD (*Single Intruction Multiple Data*). *Threads* são organizadas em blocos, que são agendados para serem executados em uma SM. GPUs são conectadas a um *host*, e para executar operações na GPU é preciso que os dados sejam copiados entre GPU e *host* através de um *PCI-Express bus*.

A memória na GPU pode ser classificada principalmente entre memória do dispositivo e memória compartilhada. A memória do dispositivo pode ser acessada por qualquer *thread* e qualquer bloco. A memória de dispositivo possui um tamanho maior (geralmente *gigabytes* de tamanho), porém o tempo de acesso é maior do que da memória compartilhada, que em contrapartida possui o tamanho reduzido (geralmente *kilobytes*). A memória compartilhada pode ser acessada por qualquer *thread* dentro de um mesmo bloco.

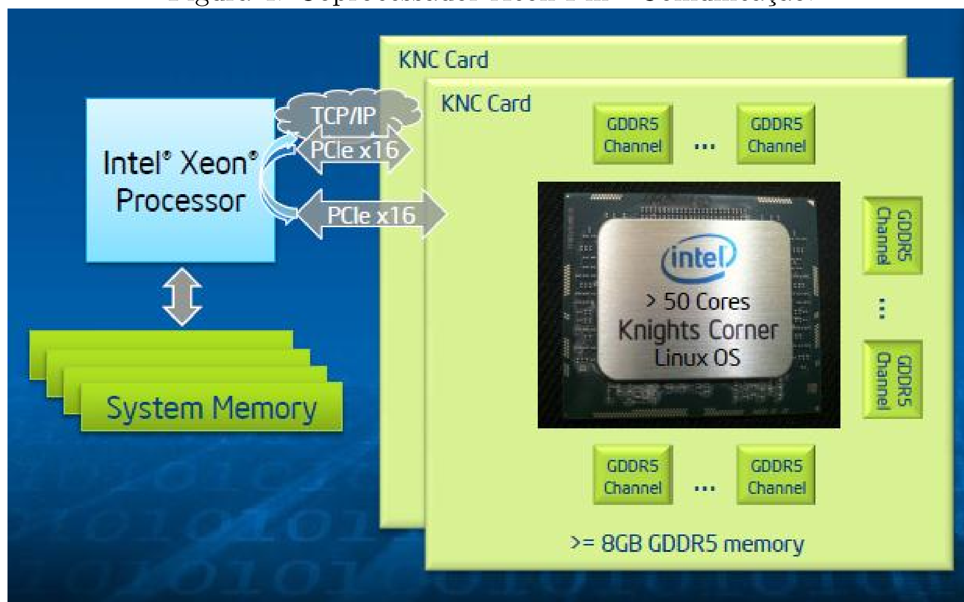
Para facilitar a implementação de algoritmos em GPU, a NVDIA lançou em 2007 uma extensão para linguagem C, chamada *Compute Unified Device Architecture* (CUDA) (NVIDIA, 2017). Em um programa em CUDA partes do código são executadas pela GPU. Essas partes são chamadas *kernels*, sendo codificadas em C e geram códigos *assembly* no formato PTX, para serem executados pela GPU.

Para permitir a evolução da arquitetura, as GPUs da NVIDIA são lançadas e divididas em diferentes gerações, chamadas de capacidade de processamento (*Compute Capability*). A capacidade de processamento é definida por dois números no formato X.Y, onde (X) indica a base da arquitetura, e o (Y) corresponde à melhoria incremental da arquitetura (NVIDIA, 2017).

2.3.2 Intel Xeon Phi - Many Integrated Core Architecture (MIC)

Em busca de um alto desempenho, a Intel lançou em 2012 o coprocessador Xeon Phi (INTEL, 2013a). Trata-se de uma placa, capaz de virtualizar uma conexão com o sistema hospedeiro e rodar um sistema operacional Linux. A Figura 4 mostra como é feita a comunicação entre o coprocessador e o sistema hospedeiro (*host).* Através de um barramento PCI-*Express*, simula-se uma conexão TCP-IP, fornecendo acesso ao sistema do coprocessador.

Figura 4: Coprocessador Xeon Phi - Comunicação.



Fonte: (INTEL, 2013b)

Seu alto desempenho na paralelização destina-se a soluções que envolvem cálculos massivos e complexos, tais como: sequenciamento genético, previsão do tempo, síntese de proteínas, etc. Esta solução de alto desempenho ocorre em decorrência da implementação da técnica de vetorização, que aumenta a velocidade de processamento dos dados. Utilizando a vetorização, é possível executar várias operações em um *array* simultaneamente.

Para o projeto do Xeon Phi, a Intel lançou uma nova arquitetura de computadores capaz de combinar muitos núcleos de uma CPU em um único *chip*; a chamada *Many*

Integrated Core Architecture (MIC) (INTEL, 2013b). Um dos pontos que mais chama a atenção nesta arquitetura é a portabilidade dos códigos: um programa escrito para uma arquitetura Intel MIC pode ser compilado e executado em processadores que seguem o modelo Intel Xeon.

Os cores integrados no *chip* (núcleos) não são dos circuitos mais avançados por ser bastante complexo colocar muitos núcleos em um *chip*. Por isso, estes são mais simples, sendo uma versão otimizada do processador Pentium. A arquitetura oferece suporte às plataformas de programação paralela a seguir: OpenMP, OpenCL, MPI e OFED Overview (INTEL, 2013b).

Cada núcleo é completamente funcional, suportando a busca e decodificação de instrução de quatro *threads* em execução. O núcleo está ligado a um anel de interconexão que contém o controle da *cache* L2 e do diretório de TAG (TD), diretório esse que cada núcleo possui o seu e onde se armazena o mapeamento dos endereços físicos.

A arquitetura conta também com 16 entradas na tabela de páginas e um tamanho de página de 4kB e 2MB, podendo misturar estes dois tamanhos. Sua unidade vetorial também é muito eficiente, suportando 512 bits de um conjunto de instruções SIMD.

2.3.3 OpenMP

O *OpenMP* é uma *Application Programming Interface* (API) que auxilia na criação de códigos paralelos para processadores *multicore*. Disponível para plataformas *unix* e *windows*, essa API permite que trechos de códigos sequenciais possam ser executados em paralelo, com apenas adições de diretivas ao código. É constituída por um conjunto de diretivas de compilador, rotinas de biblioteca, e variáveis de ambiente que influenciam o comportamento do tempo de execução (DAGUM; MENON, 1998).

A estratégia de paralelização usada pelo OpenMP é chamada *fork-join* (CONWAY, 1963). Nesse modelo, uma *thread master* cria um número específico de *threads* escravos (*forks*) e uma tarefa é dividida entre elas. As *threads* são então executadas simultaneamente, com ambiente de execução distribuindo as *threads* para diferentes processadores. Ao término da execução de todas as *threads*, somente a *thread master* continua sua execução em modo sequencial.

Como o OpenMPfaz uso de memória compartilhada, algumas variáveis possuem escopo global, podendo ser acessadas por todas as *threads*. Existem também variáveis que apresentam escopo privado; isto é, cada *thread* cria sua própria cópia da variável. Esse controle de escopo de variável afeta diretamente o desempenho de uma aplicação em paralelo, pois quando há necessidade de escrita em um variável compartilhada, é necessário a garantia de atomicidade da operação para garantir o funcionamento correto do

programa. Esse problema é chamado de *race condition* e para garantir essa atomicidade, uma *thread* faz com que as outras entrem em estado de espera, degradando o desempenho do programa (OpenMP Architecture Review Board, 2008).

2.3.4 MPI

Message Passing Interface (MPI) é um padrão para comunicação de dados em computação paralela. No padrão MPI, um ou mais processos se comunicam por meio de envio e recebimento de mensagens entre os processos. Essa troca de informações ocorre através de mensagens distribuídas pela rede. Diferente do *OpenMP*, o MPI não implementa um modelo de paralelismo, delegando essa responsabilidade para o desenvolvedor. Comumente utiliza-se a estratégia *master-slave*, em que um nó *master* delega para os nós escravos as tarefas e dados que devem ser computados e espera a resposta deles.

Um programa paralelo escrito em MPI possui o seguinte fluxo, segundo (DHILLON; MODHA, 2000):

1. O programa sequencial é escrito em C/C++ ou Fortran e quando compilado seu código é vinculado à biblioteca MPI.
2. Cada processo é responsável por uma parte do conjunto a ser computado.
3. Cada processo é identificado com um código que varia entre 0 e P processos.
4. Todos os nós podem executar o mesmo trecho de código, alterando apenas o conjunto a ser tratado ou partes diferentes da lógica principal do problema.

O MPI permite definir a quantidade de processos a serem criados de forma dinâmica e em tempo de execução. Apesar dessa técnica ser possível, normalmente a quantidade de processos a serem criados é definido durante a inicialização da aplicação e se mantêm estático, diferente do *OpenMP*, em que as *threads* podem ser criadas e destruídas várias vezes durante a execução da aplicação. Como normalmente, durante toda aplicação, existem processos paralelos independentes, a sincronização entre eles é feito por meio de mensagens ponto a ponto ou coletiva (GROPP et al., 1996).

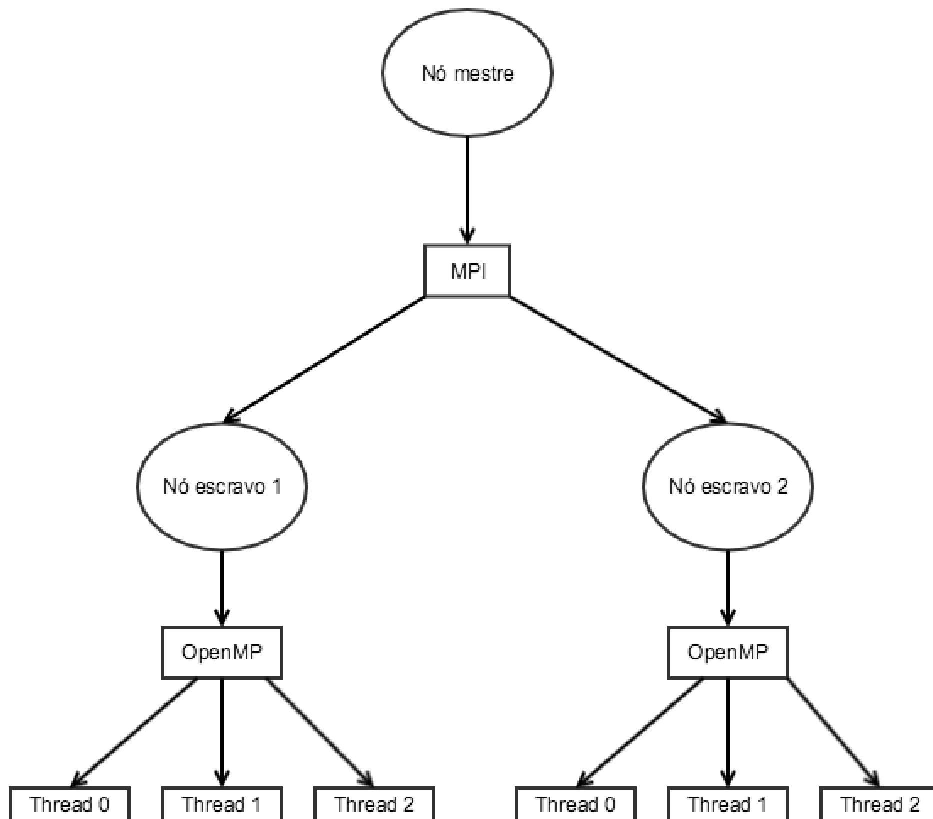
2.3.5 MPI/OpenMP

Originalmente, o *OpenMP* foi desenvolvido para memória compartilhada, e o MPI para memória distribuída. Desse modo, uma estratégia para tirar proveito de ambas as

bibliotecas é a computação híbrida. Ou seja, utilizar o *OpenMP* dentro do nó e o MPI para distribuir essas tarefas dentro do *cluster*.

Criando uma hierarquia de processamento, o MPI distribui as tarefas para os nós, que mais uma vez redistribui em *threads* por meio do OpenMP. A Figura 5 representa a arquitetura de solução.

Figura 5: Modelo híbrido de computação paralela utilizando MPI/OpenMP.



Fonte: Elaborada pelo autor.

Na computação paralela, os problemas podem ser divididos em granularidades. Granularidade fina ou pequena representa um grande número de pequenas tarefas, com um baixo custo de comunicação. Já tarefas com granularidade grossa, ou grande, são um número pequeno de tarefas com uma grande custo de comunicação.

Problemas com granularidade pequena são mais eficazes em soluções de memória compartilhada do que em memória distribuída. Isto devido ao *overhead* criado na troca de mensagens, das quais apresentam um volume proporcional à sua granularidade. Já para volumes de dados muito grandes, o modelo híbrido se torna necessário, pois a partir dele é possível reduzir o número de mensagens criadas, deixando que a maior parte do processamento seja feita pela solução de memória compartilhada. Com isto, é possível

dizer que a abordagem híbrida pode aumentar a escalabilidade e o desempenho de uma aplicação em relação ao uso de uma única abordagem.

Contudo, o uso de soluções híbridas aumenta a complexidade da aplicação. Isso acaba por apresentar diversos obstáculos para sua implementação, tais como:

1. Necessidade de uma configuração de ambiente mais robusto para garantir eficiência.
2. Aumento na dificuldade de implementação do código fonte do aplicativo.
3. O diagnóstico de problemas se torna mais árduo.

2.3.6 Raspberry Pi

O Raspberry Pi é um pequeno computador, com aproximadamente 8,5cm de largura e 5,6cm de comprimento, e com um preço de entrada no mercado de apenas US$35,00. Ele foi criado com o intuito acadêmico, para pesquisas em escolas, faculdade e em casa (PI, 2013).

Com uma arquitetura ARM, o Raspberry Pi vem sem nenhum sistema operacional instalado por padrão, mas boa parte das distribuições Linux possuem suporte para ele. Isso faz com que ele consiga executar praticamente qualquer tarefa que um computador convencional é capaz de executar.

Atualmente existem seis modelos do Raspberry Pi, são eles: *Raspberry Pi A+*, *Raspberry Pi model B*, *Raspberry Pi 2*, *Rasberry Pi Zero*, *Raspberry Pi Zero W*, e *Raspberry Pi 3*. Entre eles, existem pequenas diferenças arquiteturais, sendo a mais significante a presença de processadores *quad-core* nas versões *Raspberry Pi 2* e *Raspberry Pi 3*.

Apesar de possuir um *hardware* bem modesto, devido ao seu baixo custo, O Raspberry Pi tem sido utilizado como uma alternativa econômica para processamento de dados e *cluster* caseiros. Nessa dissertação, o *cluster* de Raspberry Pi é composto por 12 unidades do modelo Raspberry Pi 2, em que cada unidade conta com um processador quad-core Cortex-A7, 1 GB de memória RAM e uma placa de rede *fast ethernet* que alcança até 100 Mbit/s por segundos.

2.4 Trabalhos correlatos

2.4.1 Predição de SIT

Como já apresentado na Seção 2.1, um dos primeiros trabalhos na área de descoberta de SIT foi de (KOZAK, 1984), que descobriu características conservadoras nas

sequências SIT, que são, a posição -3, apresenta uma purina, nucleotídeo A (Adenina) ou G (Guanina), nas posições -1, -2, -3, -4 e -5 existe um predomínio do nucleotídeo C (Citosina). Este raciocínio ilustrado na Figura 2 é chamado consenso de Kozak.

Trabalhando com redes neurais, Pedersen e Nielsen (1997) investigaram uma base de dados de vertebrados. Os autores utilizaram a mesma codificação de Stormo, Schneider e Gold (1982), utilizando quatro bits para codificação dos nucleotídeos (A = 1000, C = 0100, G = 0010 e T = 0001), e janelas com tamanho de 13, 33, 53, 73, 93, 133, 153, 173 e 203 nucleotídeos, Com esta metodologia, os autores obtiveram até 88% de acurácia. Os autores também elaboraram uma análise das sequências que revelou a importância da posição -3 para o desempenho do classificador, o que corrobora com os estudos de Kozak.

Realizando uma análise do problema de predição de SIT focando no aspecto de balanceamento de classes, Silva et al. (2011) apresentaram o *Majority class undersampling based in Clustering* (M-Clus), além de uma nova metodologia que adiciona características às sequências, otimizando o desempenho do classificador a partir da inclusão do conhecimento obtido pelo modelo.

Pérez-Rodríguez, Arroyo-Peña e García-Pedrajas (2014) treinaram uma SVM usando cinco classes ao invés das tradicionais duas classes: positivas (para sequências SITs) e negativas (para sequências não SITs). Além do grupo positivo, eles subdividiram o grupo negativo (não SITs) em mais quatro classes baseadas na região que se encontra a sequência: *exon* (região em que são traduzidas em sequências de aminoácidos), *introns* (região não codificante, mas que é transcrita), região intragênica (que não é transcrita), e um quarto conjunto que abrange a soma das *introns* com a região intragênica. Após a separação, eles treinaram os cinco grupos separadamente. Para o processo de classificação, eles avaliam a resposta de cada um dos classificadores e escolhem a mais adequada para representar a sequência. Os autores testaram a solução em todo o genoma humano e o método conseguiu reduzir o número de falso positivos e falso negativos em até 40%.

2.4.2 K-modes

O uso do *k-modes* para algoritmos em clusterização de sequências genéticas já foi abordado por Zhang et al. (2008), que propuseram um método para clusterização de sequências de sítios de *splice* de DNA usando o algoritmo *Genetic k-modes*. O uso do algoritmo *k-modes* para clusterização de sequências mRNA para predição de SIT, abordado nessa dissertação, não foi encontrando na literatura.

2.4.3 *Paralelização usando computação heterogênea*

Para paralelizar algoritmos de dinâmica molecular, que consiste em simular computacionalmente os movimentos físicos de átomos e moléculas, Harode et al. (2014) otimizaram uma aplicação para tirar total proveito do processador *Xeon-phi*. Explorando sua unidade de aritmética dedicada que possui 512 bits de vetor para executar operações SIMD em cada *core*, eles executaram a versão da aplicação em MPI e compararam com a nova versão OpenMP, que apresenta resultados melhores no processador *Xeon-phi*.

Com apenas uma única GPU, Salomon-Ferrer et al. (2013), ainda no campo de dinâmica celular, alcançaram um desempenho mais do que duas vezes maior do que a versão paralela do algoritmo executando em um *cluster* contendo 16 *Intel Xeon Phi* E5-2670 (totalizando 384 CPUs). Eles também executaram o algoritmo usando até 4 GPUs simultaneamente (em um mesmo nó).

2.4.4 *Paralelização dos algoritmos k-means e k-modes*

Muitas vezes, o processo de clusterização envolve um grande volume de dados. Em virtude disso, e da popularidade do *k-means*, muitos trabalhos relacionados a sua paralelização são encontrados na literatura, com implementações usando tecnologias que estão no mercado desde da década de 90 (KANTABUTRA; COUCH, 2000) e trabalhos com tecnologias recentes (KANG; PARK, 2015). Embora poucos trabalhos associados ao *k-modes* sejam referenciados, os algoritmos são bastante similares. As diferença entre eles estão relacionadas a medida utilizada para o cálculo da distância entre os pontos e o centroide, além do cálculo referente a atualização do centroide.

Mesmo *frameworks* recentes, como o *hadoop*, já dispõe de trabalhos na área. Para agrupar 128000 pontos em 64 centroides, Kang e Park (2015) codificaram o *k-means* com o uso do *hadoop*, que completou a tarefa em 112 segundos, em comparação aos 1618 segundos do algoritmo sequencial. De maneira similar, a versão do *k-modes* de Tao, Xiangwu e Yefeng (2015) alcançou uma performance seis vezes maior em relação ao algoritmo sequencial.

Apesar de conseguirem um resultado satisfatório usando o *hadoop*, os maiores *speedups* encontrados na literatura recente fazem o uso de GPUs.

Para a implementações em GPUs, o CUDA tem sido bastante empregado. Um dos primeiros trabalhos realizados foi o Hong-tao et al. (2009), que conseguiu um ganho de 40 vezes em relação à versão sequencial.

Comparando com uma versão paralela e já altamente otimizada do *k-means* para CPU, (WU; ZHANG; HSU, 2009) obtiveram resultados 10 vezes mais rápido usando GPUs

para agrupar um bilhão de pontos.

Já Li et al. (2010) utilizaram um *cluster* de GPUs capaz de otimizar o acesso de memória em placas gráficas reduzindo o acesso à memória global. Usando 51.200 pontos, alcançaram um *speedup* de 6 vezes, comparado com o processamento em CPUs.

Divergindo dos trabalhos recentes em GPUs, com a biblioteca OpenCl, Dhanasekaran e Rubin (2011) desenvolveram uma versão do *k-means*. Com uma GPU ATI Radeon® HD 5870, sua versão em OpenCl atingiu um *speedup* de 35 vezes em relação a sua versão OpenMP.

Apesar do MPI ser um biblioteca antiga em relação ao CUDA e OpenMP, por sua robustez e confiabilidade, ela continua sendo muito estudada; muitas vezes em conjunto com outras bibliotecas como o OpenMP. Com uma simples abordagem *master slave*, Kantabutra e Couch (2000) atingiram um tempo de execução de até 11 vezes mais rápido que a versão original do programa.

Rao, Prasad e Venkateswarlu (2009) apresentaram uma solução para o algoritmo *k-means* paralelo usando apenas OpenMP e conseguiram reduzir o tempo de execução em até 50% comparando com os métodos sequenciais.

Feita para encontrar o escaneamento envoltório convexo (*Convex Hull Scan*), Waghmare e Kulkarni (2010) elaboraram uma versão do *k-means* utilizando uma arquitetura híbrida com MPI e OpenMP. Os testes realizados revelam que a versão híbrida supera as versões em que o uso do MPI e OpenMP é feito de maneira isolada (não híbrida).

De maneira análoga Rodrigues et al. (2012), também apresentaram uma versão híbrida do algoritmo para clusterização de sequências de mRNA. Os autores alcançaram um *speedup* de até 18.33 vezes em relação o algoritmo sequencial. Diferente da maioria dos trabalhos que envolve a paralelização do algoritmo, os autores trabalharam com base de dados reais, em que os dados utilizados foram coletados do banco RefSeq, e não com dados sintéticos (gerados de forma aleatória).

A Tabela 1 apresenta, em ordem cronológica, a relação dos trabalhos apresentados em função da abordagem utilizada, juntamente com o trabalho proposto nesta dissertação.

Tabela 1: Relação de trabalhos correlatos referentes à paralelização do algoritmo *k-means* e *k-modes*

	OpenMP	MPI	CUDA	MPI/OpenMP	OpenCL	Hadoop
(KANTABUTRA; COUCH, 2000)		X				
(HONG-TAO et al., 2009)			X			
(WU; ZHANG; HSU, 2009)			X			
(RAO; PRASAD; VENKATESWARLU, 2009)	X					
(LI et al., 2010)			X			
(WAGHMARE; KULKARNI, 2010)				X		
(DHANASEKARAN; RUBIN, 2011)	X				X	
(RODRIGUES et al., 2012)	X			X		
(KANG; PARK, 2015)						X
(TAO; XIANGWU; YEFENG, 2015)						X
Dissertação	X		X	X		

Como apresentado nos trabalhos citados, diferentes abordagens podem ser utilizadas para melhorar o tempo gasto para execução e a qualidade dos resultados. Esta dissertação visa estender os estudos de Rodrigues et al. (2012), replicando seus testes em um ambiente similar, e expandido os testes com o acréscimo de versões do *k-modes* em outras arquiteturas paralelas e não convencionais.

3 METODOLOGIA

Este capítulo apresenta os materiais utilizados na execução dos algoritmos, o método da extração de informações das sequências e as métricas utilizadas para a avaliação dos resultados.

3.1 Abordagens de paralelização

Nesta dissertação procura-se analisar algumas técnicas de paralelização do algoritmos *k-means* e *k-modes*. Atendendo-se a isso, os testes realizados por (RODRIGUES et al., 2012), que consistem em uma versão sequencial do *k-means* e uma versão híbrida com o uso de OpenMP e MPI foram refeitos. Ainda mais, foram feitos testes com o algoritmo *k-modes* em sua versão sequencial e híbrida com o uso de OpenMP e MPI, e versões para serem executados por GPUs, pelo processador Xeon-Phi e em um cluster de Raspberry Pi(HALFACREE; UPTON, 2012).

Primeiramente, há o interesse em comparar os resultados do desempenho do algoritmo *k-modes* em relação ao *k-means*, sem levar em conta os aspectos de paralelização e escalabilidade. Sendo assim, a versão do *k-modes* apresentada nesta dissertação será comparada diretamente com a versão sequencial do *k-means* apresentada por Rodrigues et al. (2012).

A escalabilidade do novo algoritmo apresentado neste trabalho também é muito importante. A partir dela é possível estabelecer uma relação de eficiência entre a quantidade de recursos utilizados e a sobrecarga de processamento das soluções em paralelo. Para comparar o comportamento de escalabilidade do *k-modes* em relação à versão paralela apresentada por Rodrigues et al. (2012), foi implementada também uma versão híbrida do *k-modes* usando MPI e OpenMP. O algoritmo possui exatamente as mesmas técnicas de paralelização e distribuição de cargas usadas por (RODRIGUES et al., 2012). Para garantir melhor acurácia dos testes, todos os testes realizados por (RODRIGUES et al., 2012) foram executados no mesmo ambiente que os testes do *k-modes*.

Com o objetivo de analisar o uso de técnicas de paralelização mais recentes, que envolvem a computação heterogênea, algoritmo foi também executado em um processador Xeon-Phi. A versão executada no Xeon-Phi é um versão OpenMP otimizada para o processador.

Ainda com propósito de comparar o desempenho da arquitetura heterogênea do Xeon-Phi com a de GPUs, foi implementada uma versão do algoritmo *k-modes* em CUDA. Desse modo, é possível comparar a eficiência entre essas duas arquiteturas.

Por fim, para comparar a eficiência energética das soluções, o consumo de energia das soluções CUDA, Xeon-Phi e do *cluster* de Raspberry Pi foi mensurado e analisado.

3.2 Ambiente de teste

Os experimentos em GPU foram realizados em uma GPU Nvidia Tesla K20m com 5120 MB. O sistema operacional usado foi o Cent OS 6.6. O código foi compilado usando *Nvidia CUDA compiler driver* (nvcc), versão 5.5, para GPUs com capacidade de processamento maior ou igual a 3.0. Para mensurar a média do consumo energético, o programa foi executado em conjunto com o comando *nvprof*(NVIDIA, 2017).

Após a execução do algoritmo, em sua saída o comando exibe várias informações referentes ao programa e a GPU, e dentre essas informações a médida de consumo energético da placa.

Para os experimentos híbridos (MPI e OpenMP), os testes foram realizados com o *cluster* SGI Altix localizado no Centro Nacional de Supercomputação (Cesup). O *cluster* conta com 64 GB de RAM e 2 processadores dodecacore AMD Opteron (32 unidades com o modelo 6176 SE, de 2.3 GHz e 32 outras com o modelo 6238, de 2.9 GHz de frequência), totalizando 1536 núcleos de processamento e um desempenho teórico de 15.97 TFlops (MEIRA, 2015).

Foram usados 9 (nove) nós e 4 (quatro) processadores por nó do *cluster* SGI Altix, totalizando 36 núcleos. Esta configuração visa replicar um ambiente de execução similar ao de Rodrigues et al. (2012). O ambiente usado pelos autores era composto por um *cluster* homogêneo com nove nós, cada um com um processador Quad Core 2.4 GHz, 4GB de memória RAM.

Os testes realizados no Xeon Phi foram feitos usando um co-processador Intel® Xeon Phi™ Coprocessor 7100, contendendo 61 núcleos de 1.24Ghz cada e 15 GB de memória RAM. Já o *host*, possui um processador Intel® Xeon® Processor E5-2620 v2 e 64 GB de memória RAM. Para mensurar a média do consumo energético durante a execução do algoritmo, periodicamente (uma vez por segundo) o programa executava o comando *micsmc*(KIDD, 2016), que informava o consumo de energia no momento, no coprocessador. Após a execução do comando pelo aplicativo, o mesmo interpretava sua saída e guardava as informações relativas ao consumo de energia, para ao final da execução estabelecer o consumo médio.

Já os testes realizados no Raspberry Pi, foram realizados em *cluster* composto por 12 Raspberry Pi 2 Model B, com processadore quad-core ARM Cortex-A7 (900MHz) e 1 GB de memória RAM. A ligação entre os nós dos *cluster* é feita por um *switch* gigabit (capaz de trafegar até 1 gigabit de dados por segundo). Porém, essa velocidade é

limitada pela placa de rede do Raspberry Pi, que chega apenas até 100 Mbit/s. O cluster conta também com um multímetro DMI T50(ISSO, 2016), capaz de medir o consumo energético dos dispositivos que compõem o *cluster*. O multímetro também fornece uma API para acessar informações de energia através de chamadas http. Para mensurar a média do consumo energético durante a execução do algoritmo, periodicamente (uma vez por segundo) o programa executava uma chamada http para a API do multímetro, que informava o consumo de energia no momento, para o *cluster*. Após a execução da chamada, o mesmo interpretava sua saída e guardava as informações relativas ao consumo de energia, para ao final da execução estabelecer o consumo médio.

O conjunto de dados utilizado nos testes foi o mesmo usado por Rodrigues et al. (2012), sendo extraídas originalmente do banco de dados RefSeq (http://www.ncbi.nlm.nih.gov/genbank) e tratados pela ferramenta PredictTIS (SILVA et al., 2011) para separar as sequências SITs das não-SITs de cada organismo.

Assim como nos trabalhos de Silva et al. (2011) e Rodrigues et al. (2012), as seguintes características adicionais contidas na sequência foram avaliadas durante a clusterização e classificação:

1. Se a sequência possui ou não um AUG em fase;
2. Se a sequência possui ou não um *stop códon* em fase nos próximos 100 nucleotídeos na região *downstream*.
3. Se a sequência possui ou não o códon GAG em fase nos próximos 100 nucleotídeos na região *downstream*.

A Tabela 2 apresenta os organismos utilizados neste estudo, a quantidade de sequências positivas (SIT) e negativas (não-SIT), além da proporção entre sequências positivas e negativas.

Tabela 2: Sequências extraídas

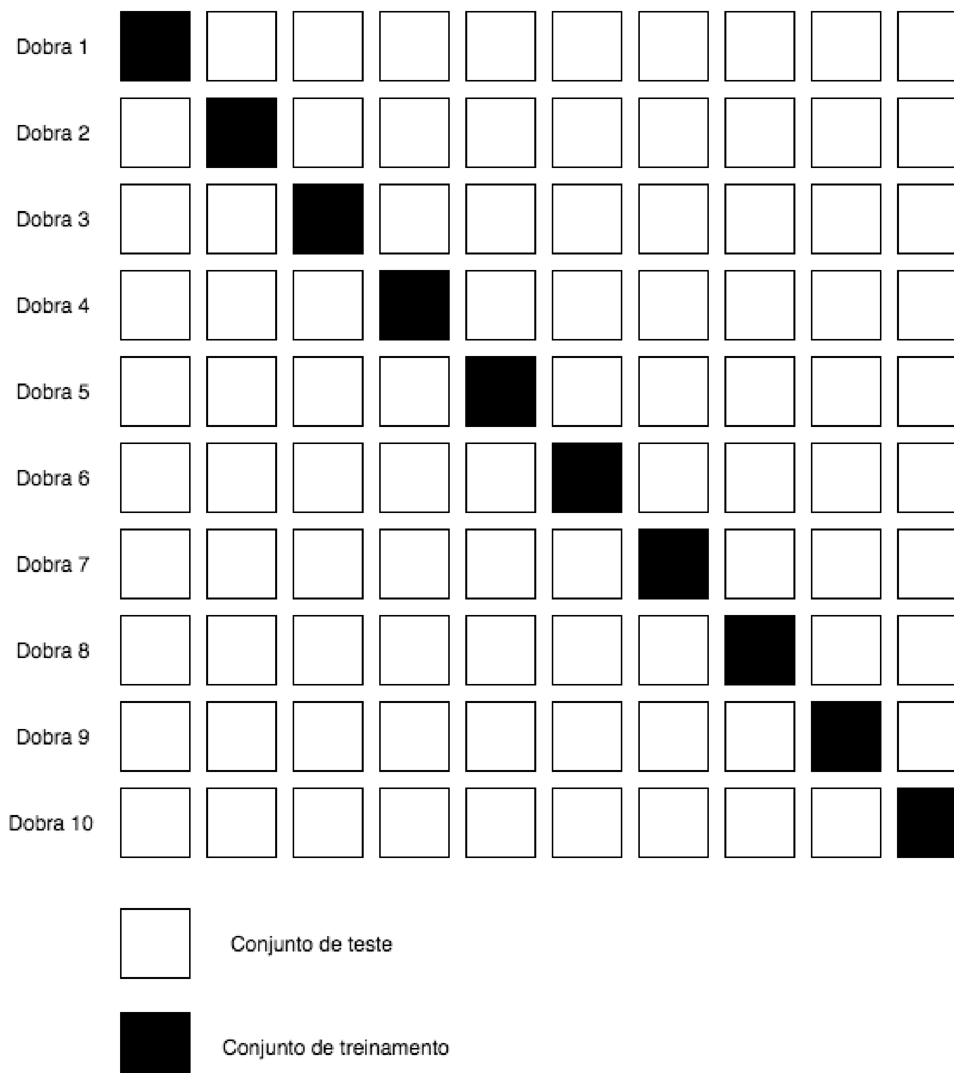
Organismo	**Sit**	**Não-Sit**	**Proporção**
Arabidopsis thaliana	24329	587886	1:24
Caenorhabditis elegans	8763	449240	1:51
Drosophila melanogaster	19782	435892	1:22
Homo sapiens	15845	353606	1:22
Gallus gallus	13	534	1:41
Mus musculus	269	6256	1:23
Rattus novergicus	31	334	1:11

A validação cruzada de 10 dobras foi utilizada como método de amostragem dos dados para validação do modelo, considerada por Kohavi (1995) como sendo o método

mais eficiente. A principal vantagem desse modelo é a garantia de que todos os exemplos do conjunto de dados são eventualmente testados e usados em treinamento.

O processo de validação cruzada utilizada na realização dos experimentos foi o modelo proposto por (MACHADO; LADEIRA, 2007), em que o conjunto de dados desbalanceado é dividido em dez subconjuntos, sendo nove deles destinados a treinamento e apenas um para validação (Figura 6). Para cada dobra o conjunto de treinamento é balanceado usando o algoritmo proposto neste trabalho. Após o balanceamento, os dados são utilizados no SVM para treinamento e validados com o conjunto reservado, sendo esse processo repetido 10 vezes. Ao final são calculados a média e o desvio padrão dos resultados obtidos.

Figura 6: Partição do conjunto em treino e teste para validação cruzada.



3.3 Métricas para Avaliação de Qualidade e Desempenho

Cinco métricas foram utilizadas para avaliar o desempenho da classificação dos modelos, sendo elas: acurácia (Ac), precisão (Pr), sensibilidade (Se), especificidade (Sp) e acurácia ajustada (Adj) (RODRIGUES et al., 2012).

A acurácia diz respeito à proporção de predições corretas (Equação 3.1), onde TP, TN, FP e FN representam a quantidade de sequências verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos, respectivamente.

$$Ac = 100 * \frac{TP + TN}{TP + TN + FN + FP} \tag{3.1}$$

A precisão avalia dentre todas as sequências classificadas na classe quais são realmente daquela classe (Equação 3.2).

$$Pr = 100 * \frac{TP}{TP + FP} \tag{3.2}$$

A sensibilidade (Equação 3.3), também conhecida como razão de verdadeiros positivos, mede a quantidade de sequências corretamente classificadas em cada classe.

$$Se = 100 * \frac{TP}{TP + FN} \tag{3.3}$$

A especificidade apresentada na Equação 3.4, também conhecida como razão de verdadeiros negativos, mensurara a proporção de não-SIT que foi classificada de maneira correta (como não-SIT).

$$Sp = 100 * \frac{TN}{TN + FP} \tag{3.4}$$

A acurácia ajustada é a média aritmética entre a sensibilidade e especificidade, como descrita na Equação 3.5.

$$Adj = \frac{Sensitivity + Specificity}{2} \tag{3.5}$$

Para avaliar o desempenho computacional foi utilizada a métrica *speedup*. O *speedup* é usado para medir o quanto um algoritmo em paralelo é mais rápido que o sequencial (Equação 3.6).

$$Sd = \frac{T_{seq}}{T_p} \tag{3.6}$$

O $Tseq$ representa o tempo total de execução do algoritmo sequencial e Tp é o tempo total de execução do algoritmo em paralelo. Para medir o tempo de execução dos algoritmos sequenciais e paralelo foi utilizado o comando *time*, disponível em todas as versões Unix. Este comando executa um determinado programa e apresenta como saída o tempo gasto na execução do mesmo.

Por fim, para mensurar a eficiência energética dos algoritmos em paralelo, foi utilizada a potência média (em watts) multiplicado pelo tempo gasto na execução do algoritmo (em segundos) (Equação 3.7). Desse modo, é possível saber o consumo total de energia em Joules.

$$J = w * t \tag{3.7}$$

Onde, J representa o consumo total de energia, em joules, durante a execução do algoritmo, w é a potência média em watts e t refere-se ao tempo de execução, em segundos.

4 K-MODES

Tanto o *k-means* quanto o *k-modes* podem ser divididos nas seguintes etapas:

1. Determinar os centroides.
2. Atribuir a cada objeto do grupo o centroide mais próximo.
3. Após atribuir um centroide a cada objeto, recalcular os centroides.
4. Repetir os passos 2 e 3 até que os centroides não sejam modificados.

Com o objetivo de reduzir os custos computacionais executados no passo 2, foi utilizada a distância de *hamming* para medir a similaridade entre as sequências de mRNA. Para conseguir um maior desempenho, as sequências foram codificadas em códigos binários, utilizando-se mesma codificação proposta por (PEDERSEN; NIELSEN, 1997) com A = 1000, C = 0100, G = 0010 e U = 0001. Para representar as características booleanas apresentadas na seção anterior, foi escolhida a codificação 0001 caso seja verdadeiro e 0010 caso seja falso. A representação de um booleano usando quatro bits será discutida posteriormente. Com isso, totaliza-se 172 bits para representação binária da sequência, sendo 160 bits para armazenar os 40 nucleotídeos e 12 para armazenar as características booleanas.

Para se obter a distância de *hamming*, é necessário apenas realizar a soma das três operações de XOR entre os inteiros que representam a sequência. Assim, para cada nucleotídeo que se difere na sequência será contabilizada uma distância de duas unidades como apresentado na Tabela 3, em que a distância entre a sequência ACG e AUG é representada pelos *bits* em negrito.

Tabela 3: Distância entre sequências.

Sequência	**Representação binária**
ACG	1000 0**10**0 0010
AUG	1000 0**0**0**1** 0010

O valor do centroide é representado por uma sequência sintética, onde dentro de uma janela de 4 *bits* (um nucleotídeo), a posição que possui a maior moda (número de bits ativos, naquela posição, dentre as sequências que fazem parte do *cluster*), tem seu bit ativado (1) e as outras posição permanecem com o bit desativado (0).

Para a representação da sequência, diferentemente do trabalho de Rodrigues et al. (2012), que optou por utilizar um *array* de *chars* (1 *byte*) com 163 posições (totalizando

1304 bits por sequência), foi utilizada um estrutura composta por 3 inteiros de 64 bits. Essa estrutura irá garantir um consumo de memória seis vezes menor (192 bits por sequência). Para armazenar 172 bits, computacionalmente são necessários 192 bits, corroborando com a escolha da representação binária em 4 bits.

A estrutura apresentada permite a redução do custo computacional relacionado ao cálculo da distância, removendo um laço interno, e em consequência reduzindo o número de operações. Com apenas três operações XOR e três operações *popCount* (contar o número de bits ativos em um inteiro), é possível calcular a distância entre a sequência e o centroide. A Tabela 3 ilustra o cálculo de distância entre as sequências ACG e AUG.

Portanto, removendo o laço responsável por iterar sob a sequência, o número de operações envolvidos na cálculo da distância foi reduzido consideravelmente, além de remover todas as operações em ponto flutuante por operações simples, que são mais rápidas.

Considerando o número de operações gastos para calcular as distâncias, o custo para o k-means e o k-modes é calculado a partir das Equações 4.1 e 4.2, onde k é a quantidade de centroides, p é o tamanho do conjunto, d é a dimensionalidade do conjunto e l é o tamanho em *bits* do inteiro na arquitetura em execução (que tem tamanho 64 nas arquiteturas utilizadas neste trabalho, com exceção do *Raspberry Pi*, que possui inteiros de 32 *bits)* .

As duas operações que se repetem no cálculo do k-means são: uma operação de subtração e uma de exponenciação para calcular o distância euclidiana (não é necessário a raiz).

Já para o k-modes, Equação 4.2, as duas operações que se repetem são: uma operação de XOR e uma operação *popCount* (contar os bits ativos).

$$CustoKmeans = 2pkd \tag{4.1}$$

$$CustoKmodes = 2pk\frac{d}{l} \tag{4.2}$$

Para atualizar os valores dos centroides, usa-se uma sequência sintética para representar o centroide. Os *bits* que formam essa sequência são escolhidos entre 0 e 1 de acordo com a moda contida naquele grupo.

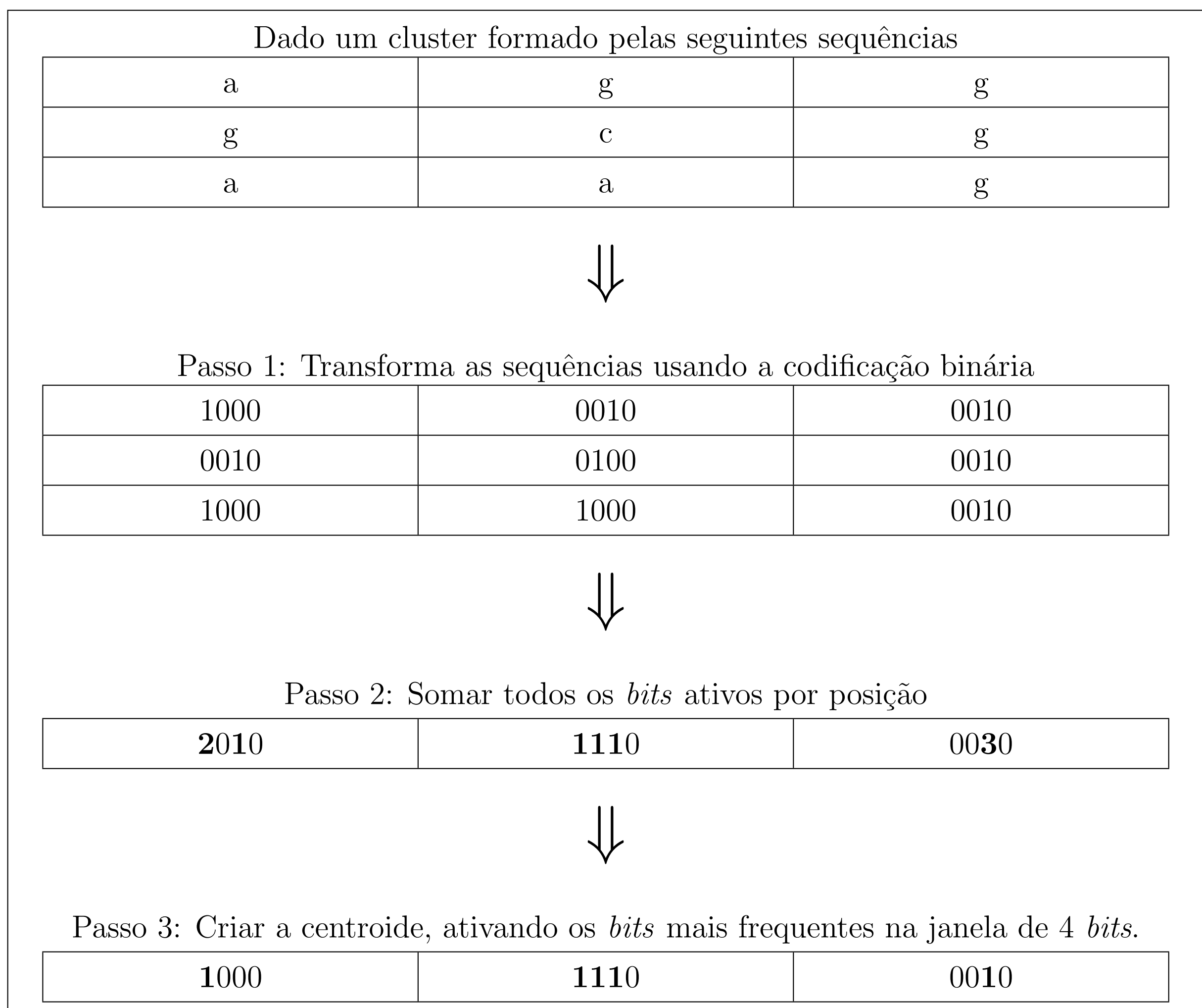
Sendo assim, se a maioria do grupo tiver o bit ativo em determinada posição, o bit naquela posição do centroide também será ativo; caso contrário, o bit que se encontra naquela posição será inativo. Porém, avaliar os bits individualmente pode ser tendencioso; é necessário favorecer algum quando houver empate. Para contornar isso, conta-se o

número de bits ativos com uma janela de 4 *bits* (um nucleotídeo).

Dentre essas quatro posições, somente a posição dominante (que apresenta a maior soma número de bits ativos) será marcada como ativo. Em caso de empate, nenhum bit é ativado. Desse modo, é garantido que nenhuma base (A = 1000, C = 0100, G = 0010 e T = 0001) será favorecida, pois a distância entre todas elas em relação a base nula (0000) é a mesma.

A Figura 7 apresenta um exemplo de cálculo de um centroide, para um grupo composto pelas sequências **agg**, **gcg** e **aag**.

Figura 7: Exemplo de cálculo de centroide



Dado um cluster formado pelas seguintes sequências

a	g	g
g	c	g
a	a	g

⇓

Passo 1: Transforma as sequências usando a codificação binária

1000	0010	0010
0010	0100	0010
1000	1000	0010

⇓

Passo 2: Somar todos os *bits* ativos por posição

2010	**111**0	00**3**0

⇓

Passo 3: Criar a centroide, ativando os *bits* mais frequentes na janela de 4 *bits*.

1000	**111**0	00**1**0

Esse passo no algoritmo exige maior custo de processamento do que as implementações do *k-means* que fazem uso da distância euclidiana. Isso se deve principalmente pela dificuldade encontrada para contar as quantidades de bits ativos em cada posição, e posteriormente acessar essas informações individualmente em janelas

de quatro bits, já que a estrutura (inteiros) não possui forma de acesso por índice diferentemente dos vetores.

Em todo caso, esse passo é executado com uma frequência menor, apenas quando a distância entre todas as sequências e centroides já foram feitas. Ou seja, uma proporção de 1:mnd onde m é o número de *clusters*, n é o número de sequências, e d é o número de dimensões dos dados (163 no caso das sequências analisadas).

4.1 *k-modes* em GPU

GPUs são capazes de executar muitas *threads* simultaneamente. Por isso, as partes do código que são executadas mais vezes foram implementadas em CUDA. Sendo elas o laço referente ao cálculo da distância, linhas 6 a 9 do Algoritmo 1 e atualização dos valores dos centroides linha 10 do Algoritmo 1. A atualização dos valores do centroide exige que o passo as distâncias esteja devidamente calculado. A única maneira de obter essa sincronização usando CUDA é separar os passos em dois *kernels* e esperar o fim da execução do primeiro *kernel* antes de iniciar o segundo *kernel*.

Nenhum dos *kernels* implementados possui dependência de dados. Isso permite uma maior fluidez das *threads*, pois não é necessário em nenhum momento utilizar técnicas para sincronização de *threads* que degradam a execução do algoritmo. O Algoritmo 2 apresenta o código do *kernel* I.

Algoritmo 2: Pseudocódigo do kernel I.

```
Entrada: Conjuntos de pontos S, Conjuntos de Centroides C
Saída: Clusters, tmpC
1 para cada PONTO n EM S faça
2     para cada CENTROID c DO CONJUNTO C faça
3         Calcule a distância entre c e n;
4         se A DISTÂNCIA ENCONTRADA É A MENOR então
5             Atribua n ao conjunto de centroide c;
6             Atualize a quantidade de pontos para o centroide c;
7     Atualize a posição do centroide temporário tmpC;
```

O *kernel* I, responsável pelos cálculos das distâncias entre as sequências e os centroides, apresenta duas concorrências de escrita de dados (*race condition*), sendo uma delas apresentada pela linha 6 do Algoritmo 2. Essa linha é responsável por contabilizar o número de centroides que tiveram seu grupo alterado. A outra concorrência de escrita ocorre na linha 07 do Algoritmo 2. Como ambas as operações concorrentes são aritméticas, esse problema foi contornado usando a função *atomicInc*, disponível em GPUs com capacidade de processamento maior ou igual a 2.0. Esta função garante a atomicidade das operações, eliminando qualquer problema relacionado a *race condition*.

Minimizar a quantidade de transferência de memória entre *host* e *device* é uma forma de alcançar melhor desempenho usando CUDA (NVIDIA, 2017). Uma das técnicas utilizadas foi armazenar todos os dados passados como parâmetros para os kernels na memória do *device*. Desse modo, com exceção da variável delta (linha 6, do Algoritmo 2), que necessita ser acessada pelo *host* ao final de cada interseção, nenhuma outra transferência de memória é efetuada entre *host* e *device* até o fim de execução do algoritmo.

4.2 *k-modes* Híbrido

Para implementação do k-modes híbrido, utilizando MPI e OpenMP, foi utilizada a mesma estratégia de (RODRIGUES et al., 2012). Assim, foi utilizada a estratégia *master-slave*, em que todos os nós (inclusive o *master*) são utilizados para efetuar o processo de clusterização.

O Algoritmo 3 apresenta o pseudo código do k-modes híbrido.

Algoritmo 3: Pseudo código *k-modes* híbrido.

```
   Entrada: Conjunto S
   Saída: Clusters
1  Distribuir o conjunto S para todos os nós ;
2  Selecionar os centroides iniciais de maneira aleatória;
3  Inicializar os centroides temporários tmpC;
4  enquanto ENQUANTO HOUVER ALTERAÇÕES NO CLUSTER faça
5      Limpar os centroides temporários tmpC;
6      para cada PONTO n DO SUB CONJUNTO S PRESENTE NO NÓ faça
7          para cada CENTROIDE c EM C faça
8              Calcular a distância entre c e n ;
9              se A DISTÂNCIA FOR A MENOR ENCONTRADA então
10                 Mover n para o cluster c;
11         Atualizar a posição temporária dos centroides tmpC;
12     Distribuir os centroides tmpC para todos os nós;
13     para cada CENTROIDE c DO SUB CONJUNTO C PRESENTE NO NÓ faça
14         atualizar o valor do centroide C baseado no valor de tmpC
15     Distribuir para todos os nós o novo valor do centroide ;
16 O nó mestre recebe todos valores dos centroides de todos os nós.;
```

A estratégia utilizada divide o processo paralelo em três partes. A primeira divide uma parte do conjunto para todos os nós, Linha 1. Isso é feito com o uso do MPI.

A segunda parte consiste em encontrar o centroide mais perto para cada sequência. Essa operação requer apenas o valor atual do centroide. Portanto, o laço na linha 6 é executado em paralelo, dentro do nó usando o OpenMP. Este laço possui uma concorrência de escrita, no momento de atualizar a menor distância, e o valor do centroide temporário. Para contornar esse problema, foi criada uma cópia das distâncias mínimas locais (para cada *thread OpenMP*) e, em seguida, é feita a redução delas após o término da execução

do laço.

O último passo consiste em calcular a nova posição dos centroides. Neste passo, todos os valores dos centroides *tmpC* já estão atualizados e não existe nenhuma dependência ou concorrência de leitura ou escrita de dados. Logo, o laço na linha 13 também é executado em paralelo usando OpenMP.

4.3 *k-modes* em XeonPhi

Para executar os programas no coprocessador XeonPhi, foi usada uma técnica chamada de *Offload*. Neste modelo de programação, assim como no CUDA, parte do código é executado no coprocessador (*device*) e parte do código no *host*. Neste modelo de computação heterogênea, o *host* e o *device* não compartilham memória. A transferência de memória e sincronização entre *host* e *device* são feitos por meio de diretivas de compilação.

Diferentemente da versão em CUDA, em que é preciso fazer cópia de dados entre o *host* e *device* em toda iteração, para o XeonPhi, após leitura da entrada de dados utilizando o *host*, todos os dados são transferidos para o *device*. Apenas após finalizar toda a execução do k-modes no *device*, o resultado é copiado novamente para o *host*. Desse modo, o *overhead* é muito menor, que devido a natureza de processamento das GPUs, não permite essa mesma flexibilidade.

Além da inclusão das diretivas de compilação para executar o código em *offload*, o código executado no *XeonPhi* é praticamente o mesmo código apresentado pelo Algoritmo 3, com a remoção do código MPI, já que ele é executado em apenas um nó. Sendo assim, os laços, nas linhas 6 e 13 do algoritmo, continuam paralelizados da mesma forma utilizando o OpenMP.

5 RESULTADOS E DISCUSSÕES

Este capítulo apresenta e discute os resultados obtidos a partir do processamento das bases de organismos da Tabela 2 em relação à metodologia utilizada. Para melhor representação dos dados, as bases de dados foram divididas em dois grupos: O primeiro, que consiste das bases com pequena quantidade de sequências, é formado pelos organismos *Gallus gallus* (GG), *Mus musculus* (MM) e *Rattus novergicus* (RN). O segundo grupo, contento bases com grande quantidade de sequências, é formado pelos organismos *Arabidopsis thaliana* (AT), *Caenorhabditis elegans* (CE), *Drosophila melanogaster* (DM) e Homo sapiens (HS).

5.1 Apresentação dos resultados

Para todas as bases, 5 versões do algoritmo *K-modes* foram executadas 10 vezes, sendo elas: a versão sequencial, a versão para GPUs, a versão híbrida MPI/OpenMP, executada no cluster Cesup), a versão OpenMP otimizada para o XeonPhi e a versão híbrida (MPI/OpenMP, executada no cluster de Raspberry Pi). A Tabela 4 apresenta a média aritmética do tempo de execução de todas as versões do *k-modes*, para cada organismo em conjunto com o *Speedup* obtido entre a versão sequencial e a versão em GPU, que são, respectivamente, a versão mais lenta e mais rápida do algoritmo.

Tabela 4: Desempenho computacional, em segundos.

Organismo	**Sequencial**	**CUDA**	**Cesup**	**Raspberry**	**XeonPhi**	***Speedup***
Arabidopsis thaliana	11368,20	55,875	1845,14	8135,03	2446,66	203,45
Caenorhabditis elegans	2110,14	15,5	409,29	1554,88	477,7	136,13
Drosophila melanogaster	6221,25	32,22	779,32	4087,50	1084	193,08
Homo sapiens	3913,76	22,22	597,90	2637,6	893,5	176,13
Gallus gallus	0,09	0,87	0,16	1,01	1.7	0,10
Mus musculus	6,8	0,92	1,81	9,54	3	6,97
Rattus novergicus	0,08	0,87	0,14	1,79	1,7	0,09

Para as bases grandes, o *Speedup* apresenta um ganho no tempo de processamento, indicando que a solução reduz consideravelmente (em mais de 200 vezes) o tempo de processamento. Em contrapartida, para as bases pequenas, com exceção do *Mus musculus*, o algoritmo em paralelo apresentou maior tempo de processamento em relação à versão sequencial. Isso pode ser explicado pelo *overhead* envolvido no custo de comunicação e sincronização de recursos na computação paralela.

A Tabela 6 apresenta os resultados da média aritmética e do desvio padrão das medidas de acurácia, precisão, sensibilidade, especificidade e acurácia ajustada. Vale

lembrar que os resultados apresentados na Tabela 6 são os mesmos valores tanto para versão paralela quanto para versão sequencial do algoritmo *k-modes*.

Tabela 5: Resultado das métricas de avaliação utilizadas, em porcentagem.

Organismo	**Acurácia**	**Precisão**	**Sensibilidade**	**Especificidade**	**Acurácia Ajustada**
Arabidopsis thaliana	91,82 (1,65)	32,24 (18,74)	91,59 (3,77)	91,83 (2,00)	91,71 (0,10)
Caenorhabditis elegans	89,83 (0,40)	14,67 (0,06)	89,58 (0,65)	89,83 (0,04)	89,71 (0,13)
Drosophila melanogaster	93,55 (3,31)	31,06 (54,26)	92,83 (7,15)	93,58 (4,05)	93,21 (0,37)
Homo sapiens	93,78 (4,31)	42,05 (69,16)	92,76 (5,15)	93,82 (5,09)	93,29 (0,36)
Gallus gallus	99,29 (1,28)	80,00 (40,00)	75,00 (40,31)	100,00 (0,0)	87,5 (20,16)
Mus musculus	94,35 (1,32)	42,13 (25,52)	92,12 (25,50)	94,44 (1,32)	93,28 (7,19)
Rattus novergicus	95,42 (10,12)	71,00 (15,69)	56,66 (4,00)	96,06 (4,00)	77,86 (32,72)

Para as bases *Gallus gallus* e *Rattus novergicus*, a precisão é alta em relação as demais. Em contrapartida, seu desvio padrão mostra que este não é um resultado regular, diferente dos apresentados pelas demais bases, onde a sensibilidade é maior, mas o desvio padrão se apresenta menor. Este fato pode ser justificado pelo tamanho das bases pequenas, sugerindo que talvez não seja suficiente para estabelecer um modelo confiável.

O tamanho das bases e razão de desbalanceamento apresentados na Tabela 2 não influenciaram os resultados, corroborando com o objetivo proposto, uma vez que os resultados encontrados se assemelham aos trabalhos de (SILVA et al., 2011) e (RODRIGUES et al., 2012), porém em relação ao tempo de execução, houve um ganho considerável.

5.2 Comparação de resultados com trabalho correlato do grupo

Esta seção apresenta a comparação dos resultados de classificação e desempenho obtidos por (RODRIGUES et al., 2012) e a proposta apresentada nesse trabalho, uma vez que foram usadas as mesmas bases de dados. Como mencionado anteriormente, Rodrigues et al. (2012) apresentaram uma solução híbrida usando MPI e OpenMP, executada em um *cluster* homogêneo com nove nós. Uma comparação entre a qualidade dos resultados apresentados neste trabalho e os obtidos por Rodrigues et al. (2012) é apresentada pela Tabela 6.

Tabela 6: Comparação das métricas de avaliação obtidas por Rodrigues et al. (2012) e por essa dissertação

Organism	Ac-P	Ac-R	Pr-P	Pr-R	Se-P	Se-R	Sp-P	Sp-R	Adj-P	Adj-R
Arabidopsis thaliana	**91.82**	91.3	**32.24**	27.74	**91.59**	83.27	**91.83**	91.25	**91,71**	87.26
Caenorhabditis elegans	89.83	**90.02**	14.67	**14.93**	89.58	**89.71**	**89.83**	89.03	89.71	**89.87**
Drosophila melanogaster	93.55	**95.22**	31.06	**43.01**	**92.83**	90.83	**93.58**	90.47	**93.21**	60.64
Homo sapiens	**93.78**	91.15	**42.05**	39.83	**92.76**	89.11	**93.82**	88.93	**93.29**	89.02
Gallus gallus	99.29	99.3	80.00	80.00	75.00	75.00	100.00	100.00	87.5	87.5
Mus musculus	94.35	**94.61**	42.13	**46.29**	92.12	92.17	**94.44**	91.39	**93.28**	91.78
Rattus novergicus	**95.42**	93.71	**71.00**	45.00	56.66	**92.01**	**96.06**	91.20	77.86	**91.70**

P=proposta e R=Rodrigues et al. (RODRIGUES et al., 2012)

Os resultados de classificação mostram que os valores obtidos pelas duas metodologias são bastante aproximados. Entre as bases grandes, os casos em que houve variações negativas, ainda assim baixas (menor que 2%), foram nas métricas de precisão e acurácia para a base *Homo sapiens*. As bases pequenas, com exceção do *Gallus gallus* (onde o resultado foi igual), houve uma maior variação. Este fato pode ser justificado pelo tamanho das bases não serem grandes o suficiente para estabelecer um modelo confiável, uma vez que as taxas de desvio padrão para essas bases são altas.

A variação quase irrelevante na classificação das bases pode ser explicada devido as seguintes características:

1. A quantidade de valores distintos encontrados nas sequências são poucos (apenas 4), sendo eles a codificação das bases A = 1000, C = 0100, G = 0010 e T = 0001.

2. A variação entre esses valores é muito baixa.

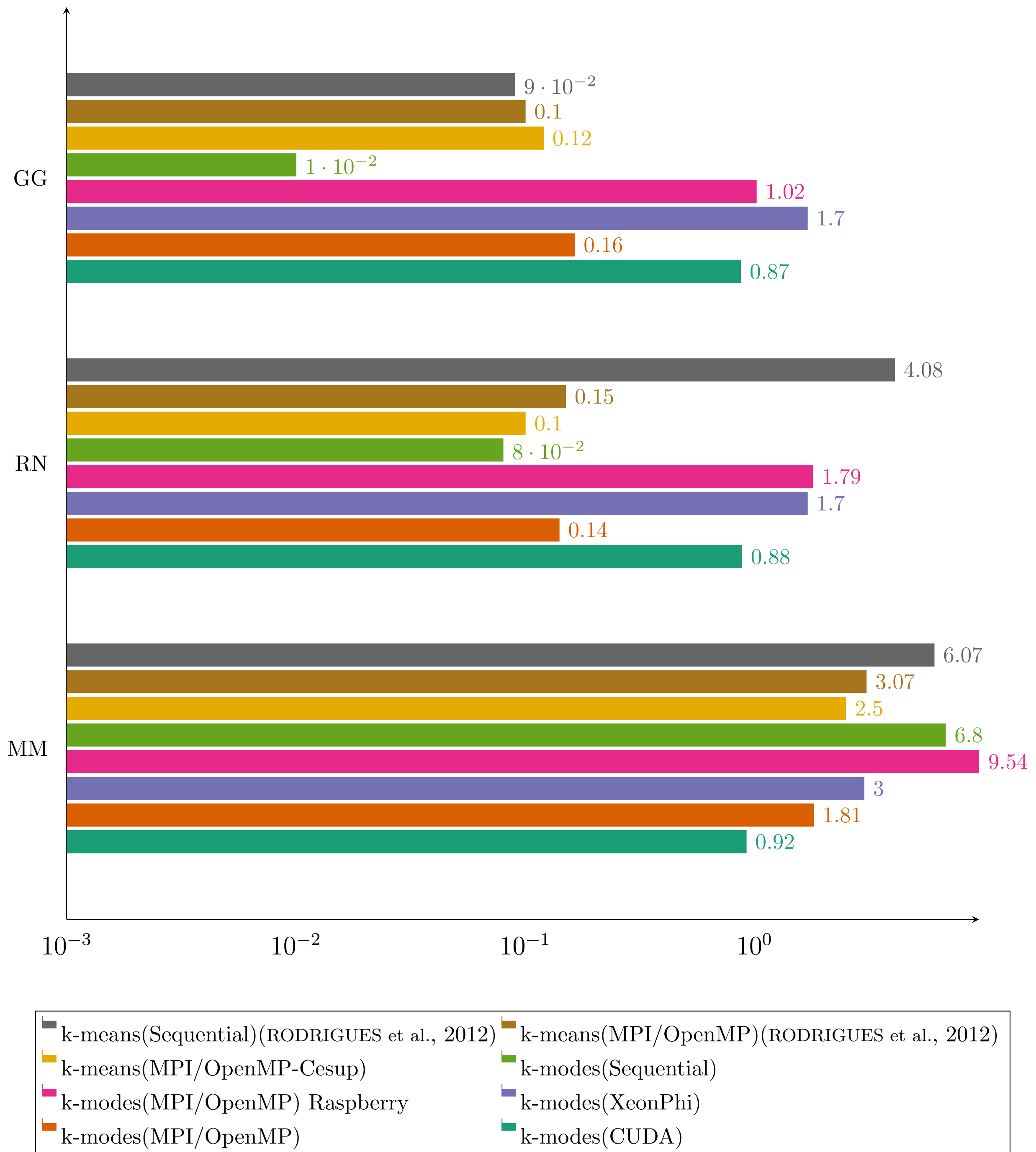
3. De maneira análoga o volume de dados é muito alto, então a frequência desses valores tende a ser muito alta.

Com a junção dessas três características é de se esperar que a moda e a média apresentem um correlação muito forte.

5.3 Comparação de desempenho com trabalho correlato do grupo

A Figura 8 e 9 apresentam a comparação entre os tempos de execução entre a versões propostas por (RODRIGUES et al., 2012) e as versões propostas neste trabalho.

Figura 8: Comparação do desempenho computacional, em segundos, para bases pequenas.



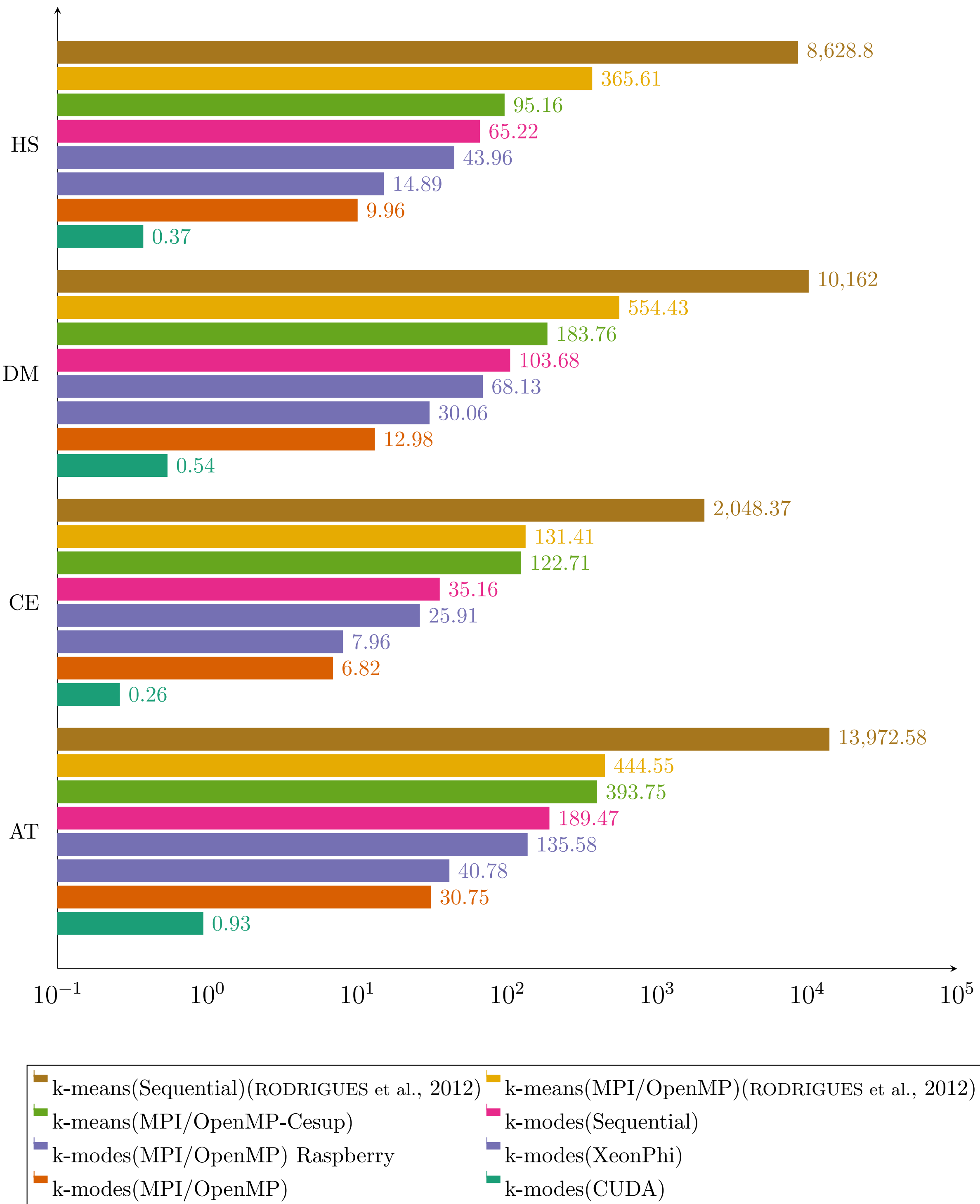
Analisando a Figura 8 é importante comparar o *overhead* entre as diversas implementações do algoritmo. Para a menor base de dados (GG), todas as versões em paralelo foram significativamente mais lentas. Isso significa que as versões paralelas gastam mais tempo gerenciando recursos do que realmente fazendo a computação do algoritmo.

Já em relação ao base RN, que é um pouco maior que a base GG, a versão em MPI/OpenMP obteve um tempo de execução mais aproximado do sequencial, indicando

que é a implementação em paralelo que possui o menor overhead em relação ao ganho de desempenho.

Finalmente, para a base MM, que já pode ser considerada uma base de tamanho médio, todas as versões em paralelo tiveram resultados melhores. Neste caso, a implementação em MPI/OpenMP ainda tem uma modesta vantagem em relação à versão em CUDA. Isso significa que o tamanho da base é próximo do ponto de convergência, em que o CUDA começa a superar todos os algoritmos. isso corrobora com a afirmação que o CUDA possui o maior *overhead* entre as implementações.

Figura 9: Comparação do desempenho computacional, em minutos para bases grandes.



A partir da Figura 9, é possível notar que o algoritmo *k-modes* apresenta um desempenho muito elevado em relação o *k-means*, sendo que a solução sequencial já consegue apresentar um tempo de processamento menor em relação à versão paralela e distribuída, apresentada por (RODRIGUES et al., 2012).

A Figura 8 mostra que o *overhead* para bases pequenas apresentado no CUDA é

maior do que a solução proposta por (RODRIGUES et al., 2012). Isto porque a penalidade de efetuar pequenas transferências de memória entre *host* e *device* é muito alta. Na implementação em CUDA é preciso realizar essa transferência para calcular as distâncias entre os centroides e as sequências e novamente para atualizar a os valores dos centroides. Já na implementação em MPI a comunicação dos nó dessas duas etapas é feita uma única fez. Em contrapartida, a curva de eficiência para a implementação em CUDA parece convergir mais rápido, como observado na base *Mus musculus*.

Mesmo em um ambiente com poder de processamento similar, a execução do mesmo algoritmo feito por (RODRIGUES et al., 2012) apresenta tempos de execução divergentes. Isto devido o custo de comunicação entre nós do Cesup ser menor, em virtude da implementação do MPI usada no ambiente, otimizada pela própria SGI para o *hardware* do gauss(MEIRA, 2015). Porém, quanto maior a base, mais próximos são os tempos de execução. Nesses casos o custo de comunicação é menos relevante em relação ao custo de processamento. Essa variação corrobora com os estudos e resultados de escalabilidade em relação ao custo de comunicação feito por (RODRIGUES et al., 2012).

É possível notar que assim como o trabalho apresentado por (RODRIGUES et al., 2012), a versão do algoritmo em paralelo demonstra ser muito eficiente para reduzir o tempo de clusterização dos dados.

5.3.1 Convergência

A Tabela 7 apresenta a média de iterações totais, por organismo, para os algoritmos *k-modes* e *k-means*. Observa-se que o algoritmo *K-means* tende a convergir mais rápido que algoritmo *k-modes*. Como notado no gráfico, o único organismo onde o *k-modes* converge mais rápido é o *Caenorhabditis elegans*. É importante frisar que, apesar de curva de convergência mais lenta, o *k-modes* ainda é mais rápido do que o *k-means*.

Tabela 7: Número total de iterações, por organismo, para o *k-modes* e *k-means*

Organismo	***K-modes***	***K-means***
Arabidopsis thaliana	50,2	36,7
Caenorhabditis elegans	33,7	57,8
Drosophila melanogaster	42,1	27,0
Homo sapiens	45,0	21,8
Gallus gallus	8,5	5,0
Rattus novergicus	9,9	5,9
Mus musculus	27,0	14,3

A curva de convergência do processo de clusterização para as bases pequenas e grandes dos algoritmos *k-means* e *k-modes* é apresentada pela Figura 10. Para bases

grandes (AT, DM, HS e CE) são exibidas apenas as 15 primeiras iterações do algoritmo.

A partir da Figura 10, é possível avaliar o comportamento dos centroides, em que a curva de convergência é similar para todos os organismos. Também é possível notar que a convergência dos centroides (em outras palavras, a formação dos *clusters*), para todos os organismo é mais rápida nas primeiras iterações para todos os organismos. A maioria das alterações acontecem na primeira iteração, onde os centroides são escolhidos aleatoriamente. Sendo assim, o Gráfico 10 também corrobora com a Tabela 7, indicando que o algoritmo *k-means* tende a convergir mais rápido que o algoritmo *k-modes*.

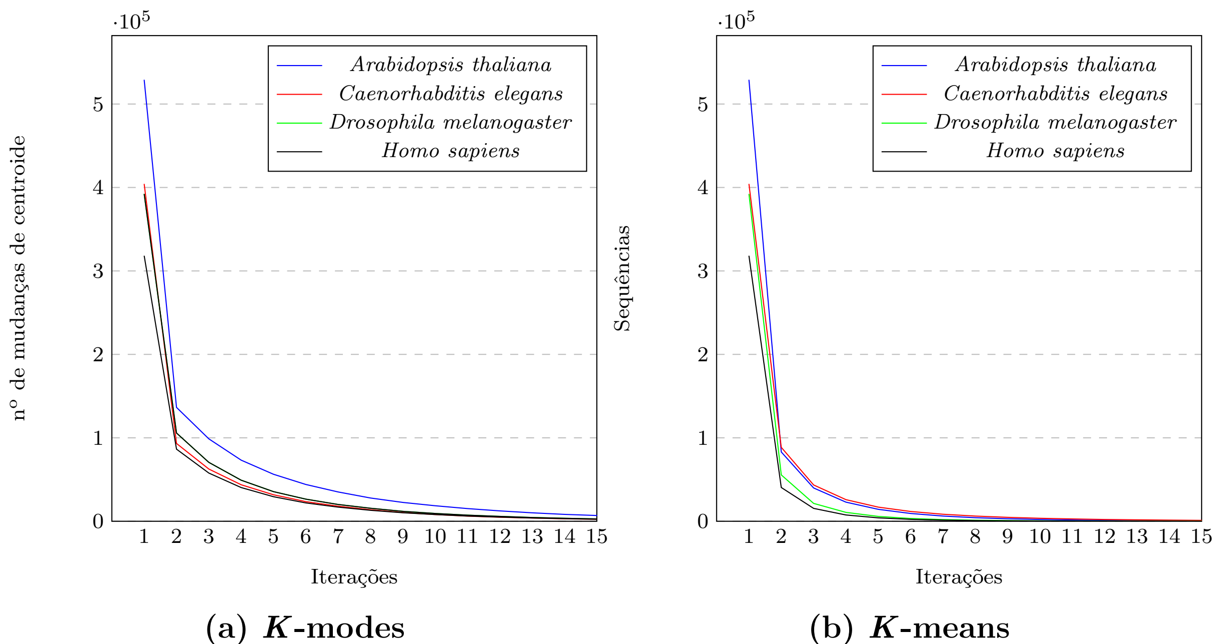
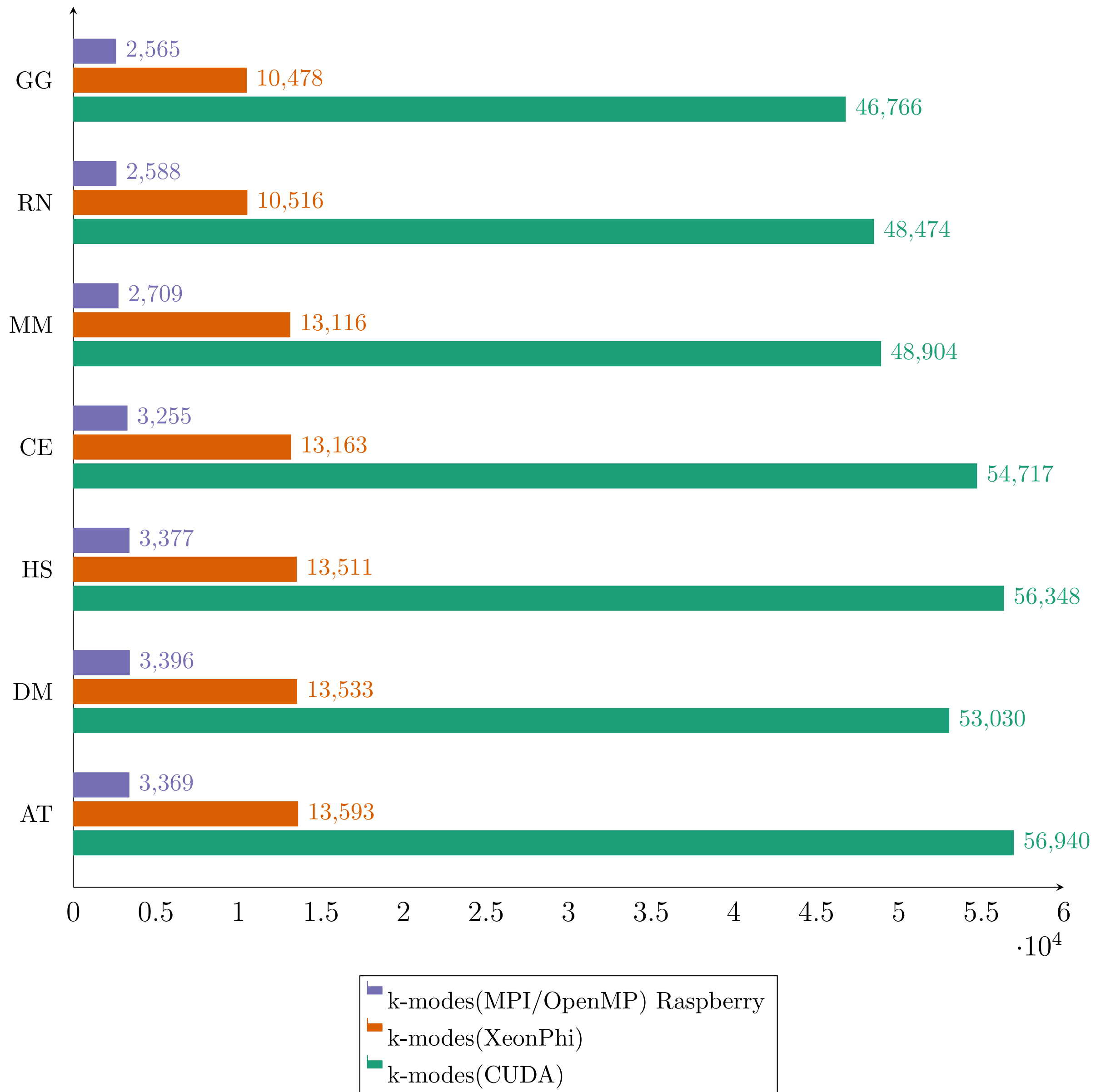


Figura 10: Taxa de convergência (quantidade de sequências) no processo de clusterização das bases grandes.

5.4 Eficiência energética

Além do tempo de execução, a eficiência energética da solução é outra métrica importante para análise de algoritmos em paralelo. O Gráfico 11 apresenta a média de consumo para todas as bases em que foram executados os algoritmos em CUDA, XeonPhi e no cluster Raspberry.

Figura 11: Comparação da média energética, por organismo (em watt).



Analisando a figura, observa-se que as soluções apresentam um consumo médio de energia bem divergente. A solução que apresentou o melhor desempenho de execução (CUDA), também é a solução que apresenta o maior consumo de energia, podendo chegar até a 17 vezes o consumo do Raspberry e 4 vezes o consumo do XeonPhi. A solução que apresentou o pior desempenho de execução dentre as apresentadas no gráfico, também foi a que teve a menor média de consumo. Esse é um comportamento esperado, uma vez que o tempo de execução é diretamente ligado à quantidade de recursos computacionais utilizados.

Outro fator importante apresentado no gráfico é que, assim como o tempo de execução, o tamanho da base de dados também interfere no consumo energético. Apesar

da influência entre o consumo energético e o tamanho das bases não ser forte quanto é no tempo de execução, pode-se observar que o consumo cresce proporcionalmente ao tamanho da base.

Além da média de consumo energético, o consumo total de energia do algoritmo também é de extrema importância, pois com ele é possível ter uma relação de eficiência entre consumo energético e tempo de execução do algoritmo. A Tabela 8 apresenta o consumo total de energia por algoritmo, em joules.

Tabela 8: Consumo total de energia em joules.

Organismo	**CUDA**	**XeonPhi**	**Raspberry**
Arabidopsis thaliana	3181	33257	27406
Drosophila melanogaster	1708	14669	1388
Homo sapiens	1252	12072	8907
Caenorhabditis elegans	848	6287	5061
Gallus gallus	40	11	3
Rattus novergicus	45	40	26
Mus musculus	42	18	5

Observando-se a tabela, é possível notar que, apesar de ter o maior consumo médio energético, o algoritmo em CUDA é o mais eficiente em relação ao consumo de energia para bases grandes (AT, DM, HM e CE), devido ao seu tempo de execução. Já para as bases pequenas (GG, RN, MM), o cluster de Raspberry Pi, possui a melhor média energética. O cluster de Raspberry Pi possui também a melhor eficiência energética dentre os três.

6 CONCLUSÕES E TRABALHOS FUTUROS

Como descrito nesse trabalho, o custo computacional para o balanceamento de sequências não SIT é muito alto, e é preciso investir em pesquisa para conseguir alcançar tempos viáveis para essa tarefa. A implementação do *k-modes*, proposta nesta dissertação para clusterização de sequências não-SITs, reduziu o tempo de processamento para todas as bases testadas em relação à versão sequencial do algoritmo *k-means*, chegando, em alguns casos, a ser mais de 70 vezes mais rápida. Usando a implementação em CUDA foi possível obter um *Speedup* próximo de 190 vezes para algumas bases em relação à implementação sequencial do *k-modes*, totalizando um *speedup* total de até 13000 vezes em relação à versão sequencial proposta por (RODRIGUES et al., 2012). Além do tempo de execução, o algoritmo em CUDA também demonstrou ter uma maior eficiência energética para bases grandes do que os demais algoritmos.

Além de apresentar a solução em CUDA, outras duas soluções em paralelo usando OpenMP em um XeonPhi e MPI/OpenMP em um cluster, foram apresentadas. Dentre essas soluções o cluster MPI/OpenMP obteve melhores resultados. Além disso, foi apresentada uma versão do algoritmo em MPI/OpenMP para o cluster de Raspberry, que obteve resultados um pouco melhor que a sequencial (cerca de 30%, mais rápido), mas demonstrou ser um algoritmo altamente eficiente em relação ao consumo de energia.

A estratégia proposta nesta dissertação, mostrou ser eficiente para predição de SITs, o tempo de processamento foi reduzido e não comprometeu a qualidade das classificações para acurácia, sensibilidade, especificidade e acurácia ajustada.

Ainda é necessário efetuar uma análise aprofundada em relação às variáveis que podem influenciar no desempenho da implementação em CUDA. Esse estudo pode ser feito através do *Visual Profiler* (NVIDIA, 2017). *Visual Profiler* é um programa gráfico que permite ver dados detalhados da execução de um programa CUDA para análise e otimização de código. Entre os dados que são analisados, pode-se destacar a média de número de *cache miss* e número de *cache hit* referentes à memória global e à memória local (sendo essas estatísticas separadas por nível de cache, L1 ou L2).

Melhores resultados também podem ser obtidos fazendo o uso de técnicas avançadas em CUDA como *Pitch Memory* para indexar *arrays* bidimensionais. O uso da *Pitch Memory* permite que um *array* bidimensional seja instanciado de forma que a memória fique alinhada para garantir maior velocidade de acesso (NVIDIA, 2017). Outras estratégias de paralelização como o uso de outras GPUs em um mesmo *host* ou distribuídas também podem aumentar o desempenho da aplicação.

REFERÊNCIAS

ASANOVIC, K. et al. The landscape of parallel computing research: A view from berkeley. n. UCB/EECS-2006-183, Dec 2006. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.

ASANOVIC, K. et al. A view of the parallel computing landscape. Commun. ACM, ACM, New York, NY, USA, v. 52, n. 10, p. 56–67, out. 2009. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1562764.1562783>.

CONWAY, M. E. A multiprocessor system design. ACM, New York, NY, USA, p. 139–146, 1963. Disponível em: <http://doi.acm.org/10.1145/1463822.1463838>.

CORTES, C.; VAPNIK, V. Support-vector networks. Machine Learning, v. 3, n. 20, p. 273–297, 1995.

DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE, IEEE, v. 5, n. 1, p. 46–55, 1998.

DHANASEKARAN, B.; RUBIN, N. A new method for gpu based irregular reductions and its application to k-means clustering. ACM, New York, NY, USA, p. 2:1–2:8, 2011. Disponível em: <http://doi.acm.org/10.1145/1964179.1964182>.

DHILLON, I. S.; MODHA, D. S. A data-clustering algorithm on distributed memory multiprocessors. Springer-Verlag, London, UK, UK, p. 245–260, 2000. Disponível em: <http://dl.acm.org/citation.cfm?id=648035.744385>.

GROPP, W. et al. A high-performance, portable implementation of the mpi message passing interface standard. Parallel computing, Elsevier, v. 22, n. 6, p. 789–828, 1996.

HALFACREE, G.; UPTON, E. Raspberry Pi User Guide. 1st. ed. Wiley Publishing, 2012. Disponível em: <http://www.cs.unca.edu/ bruce/Fall14/360/RPiUsersGuide.pdf>. ISBN 111846446X, 9781118464465.

HARODE, A. et al. Optimization of molecular dynamics application for intel xeon phi coprocessor. p. 1–6, Dec 2014.

HATZIGEORGIOU, A. G. Translation initiation start prediction in human cdnas with high accuracy. Bioinformatics, v. 18, p. 343–350, 2002.

HONG-TAO, B. et al. K-means on commodity gpus with cuda. v. 3, p. 651–655, March 2009.

HUANG, Z. Extensions to the k-means algorithm for clustering large data sets with categorical values. Data Mining and Knowledge Discovery, Kluwer Academic Publishers, v. 2, n. 3, p. 283–304, 1998. ISSN 1384-5810. Disponível em: <http://dx.doi.org/10.1023/A%3A1009769707641>.

INTEL. Intel® Xeon Phi™ Coprocessor 3120A (6GB, 1.100 GHz, 57 core) Product Specifications. 2013. Disponível em: <http://ark.intel.com/products/75797/Intel-Xeon-Phi-Coprocessor-3120A-6GB-1$_1$00 $- GHz - 57 - core$>.

INTEL. O que é o Intel® Xeon Phi™ e como ele atinge o impressionante processamento de 1 TFlops. [s.n.], 2013. Disponível em: <https://software.intel.com/pt-br/articles/o-que-o-intel-xeon-phi-e-como-ele-atinge-o-impressionante-processamento-de-1-tflops>.

ISSO. Manual de ativação DMI. [s.n.], 2016. Disponível em: <http://www.issotecnologia.com/download/115/01_manual_de_ativacao_dmi>.

JAIN, A. K.; MURTY, M. N.; FLYNN, P. J. Data clustering: A review. ACM Comput. Surv., ACM, New York, NY, USA, v. 31, n. 3, p. 264–323, set. 1999. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/331499.331504>.

JONES, C. G. et al. Parallelizing the web browser. USENIX Association, Berkeley, CA, USA, p. 7–7, 2009. Disponível em: <http://dl.acm.org/citation.cfm?id=1855591.1855598>.

KANG, Y.; PARK, Y. The performance evaluation of k-means by two mapreduce frameworks, hadoop vs. twister. p. 405–406, Jan 2015.

KANTABUTRA, S.; COUCH, A. L. Parallel k-means clustering algorithm on nows. NECTEC Technical journal, v. 1, n. 6, p. 243–247, 2000.

KIDD, T. I. Intel® Xeon Phi™ coprocessor Power Management Configuration: Using the micsmc command-line Interface. Intel, Aug 2016. Disponível em: <https://software.intel.com/en-us/blogs/2014/01/31/intel-xeon-phi-coprocessor-power-management-configuration-using-the-micsmc-command>.

KIJSIPONGSE, E.; U-RUEKOLAN, S. Dynamic load balancing on gpu clusters for large-scale k-means clustering. p. 346–350, May 2012.

KOHAVI, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, p. 1137–1143, 1995. Disponível em: <http://dl.acm.org/citation.cfm?id=1643031.1643047>.

KOZAK, M. Compilation and analysis of sequences upstream from the translational start site in eukaryotic mRNAs. Nucleic Acids Research, v. 12, p. 857–872, 1984.

LI, Y. et al. Speeding up k-means algorithm by gpus. p. 115–122, June 2010.

MACHADO, E. L.; LADEIRA, M. Um estudo de limpeza em base de dados desbalanceada e com sobreposição de classes. p. 330–340, 2007.

MACQUEEN, J. Some methods for classification and analysis of multivariate observations. v. 1, p. 281–297, 1967.

MEIRA, L. O Cluster SGI Altix: Guia do Usuário. [s.n.], 2015. Disponível em: <http://dx.doi.org/10.1021/ct400314y>.

NOBRE, C. N. Desenvolvimento de uma Metodologia para Previsão de Sítios de Início de Tradu{ccão. Tese (Doutorado) — Universidade Federal de Minas Gerais.

NVIDIA. CUDA C PROGRAMMING GUIDE. [s.n.], 2017. Disponível em: <http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf>.

OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0. maio 2008. Disponível em: <http://www.openmp.org/mp-documents/spec30.pdf>.

PEDERSEN, A. G.; NIELSEN, H. Neural network prediction of translation initiation sites in eukaryotes: perspectives for est and genome analysis. Ismb. Vol. 5, 1997.

PÉREZ-RODRÍGUEZ, J.; ARROYO-PEÑA, A. G.; GARCÍA-PEDRAJAS, N. Improving translation initiation site and stop codon recognition by using more than two classes. Bioinformatics, Oxford Univ Press, p. btu369, 2014.

PI, R. Raspberry pi. Raspberry Pi, v. 1, p. 1, 2013.

PROSDOCIMI, F. INTRODUÇÃO À BIOINFORMÁTICA. 2007. Disponível em: <http://www2.bioqmed.ufrj.br/prosdocimi/FProsdocimi07_CursoBioinfo.pdf>. Acesso em: 22 Out. 2015.

RAO, S.; PRASAD, E.; VENKATESWARLU, N. A scalable k-means clustering algorithm on multi-core architecture. p. 1–9, Dec 2009.

RODRIGUES, L. et al. Parallel and distributed kmeans to identify the translation initiation site of proteins. p. 1639–1645, Oct 2012.

SALOMON-FERRER, R. et al. Routine microsecond molecular dynamics simulations with amber on gpus. 2. explicit solvent particle mesh ewald. Journal of Chemical Theory and Computation, v. 9, n. 9, p. 3878–3888, 2013. Disponível em: <http://dx.doi.org/10.1021/ct400314y>.

SILVA, L. M. et al. Improvement in the prediction of the translation initiation site through balancing methods, inclusion of acquired knowledge and addition of features to sequences of mrna. BMC genomics, BioMed Central, v. 12, n. 4, p. S9, 2011.

STORMO, G. D.; SCHNEIDER, T. D.; GOLD, L. M. Characterization of translational initiation sites in e. coli. Nucleic Acid Res, v. 10, n. 9, p. 2971–2996, 1982.

TAO, G.; XIANGWU, D.; YEFENG, L. Parallel k-modes algorithm based on mapreduce. p. 176–179, Feb 2015.

WAGHMARE, V.; KULKARNI, D. Convex hull using k-means clustering in hybrid (mpi/openmp) environment. p. 150–153, Nov 2010.

WU, R.; ZHANG, B.; HSU, M. Clustering billions of data points using gpus. ACM, New York, NY, USA, p. 1–6, 2009. Disponível em: <http://doi.acm.org/10.1145/1531666.1531668>.

ZHANG, Q. et al. Genetic k-modes based dna splice site adjacent sequences feature analysis. p. 2065–2070, June 2008.