MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

THIAGO MAIA SANTOS

Acelerando operações filogenéticas usando programação paralela

Trabalho de conclusão de curso apresentado à Universidade Federal de Goiás como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Wellington Santos Martins

Aprovado em 4 de Julho de 2017.

BANCA EXAMINADORA

Prof. Dr. Wellington Santos Martins
Universidade Federal de Goiás
Instituto de Informática

Prof. Me. Elias Batista Ferreira
Universidade Federal de Goiás
Instituto de Informática

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

THIAGO PEREIRA MAIA DOS SANTOS

Acelerando operações filogenéticas usando programação paralela

Goiânia
2017

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE TRABALHO DE CONCLUSÃO DE CURSO EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Acelerando operações filogenéticas usando programação paralela

Autor(a): Thiago Pereira Maia dos Santos

Goiânia, 10 de Julho de 2017.

Thiago Pereira Maia dos Santos – Autor

Wellington Santos Martins – Orientador

THIAGO PEREIRA MAIA DOS SANTOS

Acelerando operações filogenéticas usando programação paralela

Trabalho de Conclusão apresentado à Coordenação do Curso de Sistemas de Informação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Área de concentração: Computação Paralela.

Orientador: Prof. Wellington Santos Martins

Goiânia
2017

Dedico este trabalho a todos os meus amigos e familiares.

Agradecimentos

Primeiramente agradeço a todos os meus amigos e familiares que estiveram comigo em todos os momentos, principalmente na reta final onde tive alguns problemas familiares referentes ao desaparecimento do meu pai. Obrigado a todos pelo apoio.

Agradeço também ao meu orientador, Professor Wellington Santos Martins, que me ajudou extremamente durante esse um ano, com seu conhecimento e sua paciência de sempre. Me ajudou também no processo seletivo do mestrado nos EUA, sendo de suma importância para meu sucesso. Serei sempre grato.

Agradeço a todos os outros envolvidos na construção desse projetos, Elias e Evandro. Sem eles, a execução do projeto seria inviável.

Agradeço a minha mãe, que sempre fez o possível e impossível para mim ter uma educação digna e uma oportunidade que ela nunca teve.

Agradeço ao meu pai, que sempre me apoio e que sempre dizia que com os pés no chão e cabeça no lugar, eu iria onde quisesse.

Por fim, agradeço a minha namorada Tenshi, que mesmo de bem longe sempre me ajudou e me deu forças para continuar em frente, a nunca desistir dos meus sonhos.

"Feliz aquele que transfere o que sabe e aprende o que ensina"

Cora Coralina,
Wikipedia: a enciclopedia livre.

Resumo

Santos, Thiago. **Acelerando operações filogenéticas usando programação paralela**. Goiânia, 2017. 50p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

A mescla entre um grande poder computacional e procedimentos eficazes, possibilitou a simulação de análises filogenéticas baseadas em um grande número de árvores, com o intuito de possibilitar a reconstrução e/ou atualização da árvore da vida. Trabalhos anteriores possibilitou essas simulações com um número grande de árvores filogenéticas, abrindo então a possibilidade de lidar com grandes conjuntos de dados em análises evolucionárias. Todavia, apenas a possibilidade de se trabalhar com um grande volume de dados não é suficiente para facilitar as análises feitas pelos biólogos, é também de suma importância haver cálculos de coeficientes estatísticos, possibilitando assim uma verificação de correlação evolutiva de uma determinada característica da espécies. Neste trabalho, apresentamos uma implementação altamente paralela e eficiente para o cálculo estatístico I de Moran, assim também como outros cálculos e funções não paralelas bastante eficientes. O uso da computação em GPUs e de um número elevado de threads, resultou em ganhos de desempenho de até 380 vezes quando comparado a uma implementação sequencial dos mesmos procedimentos. Nossos resultados gera a possibilidade de lidar com uma massa de dados grande e promove um índice que meÃ§a a quantidade de incerteza filogenética presente em árvores incompletas, facilitando assim análises evolucionárias.

Palavras–chave

Computação Paralela, GPU, CUDA, Filogenia, Árvore Filogenética

Abstract

Santos, Thiago. **Accelerating comparative phylogeny operations using parallel programming** . Goiânia, 2017. 50p. Graduation Final Project. Instituto de Informática, Universidade Federal de Goiás.

The mixture between a great computational power and effective procedures, allowed the simulation of phylogenetic analyses based on a large number of trees, in order to enable the reconstruction and / or updating of the tree of life. Recently works has enabled simulations with a large number of phylogenetic trees. Thus, it has opened the possibility of dealing with large datasets in evolutionary analyses. However, the possibility of working with a large volume of data by itself is not sufficient to facilitate the analysis made by the biologists, it is also of the utmost importance to have statistical coefficient calculations, thus enabling an evolutionary correlation check of a given species characteristic. In this work, we present a highly parallel and efficient implementation for the Moran Autocorrelation Coefficient calculation, as well as other non-parallel calculations and quite efficient functions. The use of GPU computing and a high mass of threads, resulted in performance gains of up to 380 times when compared to a sequential implementation of the same procedures. Our results generate the possibility of dealing with a large data mass and promote an index that measures the amount of phylogenetic uncertainty present in incomplete trees, thus facilitating evolutionary analyses.

Keywords

Parallel computing, GPU, CUDA, Phylogeny, Phylogenetic tree

Sumário

Lista de Figuras **9**

Lista de Tabelas **10**

Lista de Algoritmos **11**

Lista de Códigos de Programas **12**

1 Introdução **13**

2 Filogenia **15**
- 2.1 Método de Comparação Filogenético 15
- 2.2 Dados Utilizados 16
  - 2.2.1 Árvore Filogenética 16
  - 2.2.2 Formato Newick 17
- 2.3 Coeficiente I de Moran 17

3 Computação Paralela **20**
- 3.1 Visão Geral 20
  - 3.1.1 Paralelismo 21
  - 3.1.2 Taxonomia de Flynn 23
  - 3.1.3 Arquiteturas de computação paralela 24
  - 3.1.4 Arquitetura heterogênea 28
  - 3.1.5 CPU x GPU 29
  - 3.1.6 Métricas de desempenho 29
    - Eficiência 30
    - Aceleração(speedup) 30
    - Custo 31
    - Granularidade 31
    - Escalabilidade 31
- 3.2 Ambientes de programação 31
  - 3.2.1 CUDA 32
  - 3.2.2 Estrutura de um programa em CUDA 33
  - 3.2.3 Desafios Programação em CUDA C 33

4 Proposta de Implementação **35**
- 4.1 Pré-processamento 36
- 4.2 Inserção de espécies 38
- 4.3 Cálculo de matriz de distância 38

4.4 Cálculo de I de Moran 38

5 Resultados **43**
5.1 Cálculo de I de Moran 43

6 Conclusão **47**

Referências Bibliográficas **49**

Lista de Figuras

2.1 Árvore filogenética [4] 17
2.2 Árvore filogenética e suas formas de representação [4] 18

3.1 Parallel x Serial programming representation [3] 22
3.2 SISD Representation [17] 24
3.3 SIMD Representation [17] 24
3.4 MISD Representation [17] 25
3.5 MIMD Representation [17] 25
3.6 SIMD Distributed Memory Architecture [13] 27
3.7 SIMD Shared Memory Architecture [13] 27
3.8 MIMD Shared Memory Architecture [13] 28
3.9 MIMD Distributed Memory Architecture [13] 28

4.1 Estrutura para representação dos dados [4] 41

5.1 Speedup para 128 ávores 45
5.2 Speedup para 1024 ávores 45
5.3 Speedup para 4096 ávores 46
5.4 Speedup para 8192 ávores 46

Lista de Tabelas

5.1 Tempo de execução I de Moran para 128 árvores 44
5.2 Tempo de execução I de Moran para 1024 árvores 44
5.3 Tempo de execução I de Moran para 4096 árvores 44
5.4 Tempo de execução I de Moran para 8192 árvores 44

Lista de Algoritmos

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

Árvore filogenéticas, ou árvore evolutiva, representa as relações evolutivas entre um conjunto ou grupo de organismos, chamadas taxa. As pontas das árvore, ou folhas como são chamadas em grafos, representam grupos de taxa descendentes(espécies mais frequentes, novas) e os nós internos da árvore representam os antepassados comuns daqueles descendentes. Tais informações são de crucial importância nas pesquisas biológicas, e com a mescla de uma abordagem moderna da biologia molecular com um eficiente poder computacional de baixo custos, há uma significativa expansão na disponibilidade de sequências genômicas em bases de dados públicas. [8] Além disso, o estudo da evolução e reconstrução da árvore da vida têm se tornado cada vez mais eficiente. Estudiosos utilizam a informação filogenética para estudar como a seleção natural tem eventualmente moldado às características das espécies, bem como suas variações. Além disso, o estudo da evolução e reconstrução da árvore da vida têm se tornado cada vez mais eficiente. Estudiosos utilizam a informação filogenética para estudar como a seleção natural tem eventualmente moldado às características das espécies, bem como suas variações.

Muitas árvores filogenéticas têm uma única linhagem na base representando um antepassado comum. Os cientistas chamam essas árvores de "enraizadas", o que significa que há uma única linhagem ancestral a que todos os organismos representados no diagrama se relacionam. As árvores não-arraigadas não mostram um antepassado comum, mas mostram relações entre as espécies.[1] Em uma árvore enraizada, a ramificação indica relações evolucionárias. O ponto onde ocorre uma divisão, chamado de ponto de ramificação, representa onde uma única linhagem evoluiu para uma nova distinta. Uma linhagem que evoluiu cedo da raiz e permanece sem ramificação é chamada de taxon basal. Quando duas linhagens se originam do mesmo ponto de ramificação, elas são chamadas táxons irmãos. Um ramo com mais de duas linhagens é chamado de politomia e serve para ilustrar onde os cientistas não determinaram definitivamente todas as relações. É importante notar que, embora os táxons irmãos e a politomia compartilhem um antepassado, isso não significa que os grupos de organismos se dividiram ou evoluíram entre si. Os organismos em dois táxons podem ter se dividido em um ponto de ramificação específico, mas nenhum deles deu origem ao outro. [2] As árvores filogenéticas enraizadas

podem servir como um caminho para a compreensão da história evolutiva. O caminho pode ser rastreado a partir da origem da vida para qualquer espécie individual, navegando através dos ramos evolutivos entre os dois pontos. Além disso, começando com uma única espécie e rastreando de volta para o "tronco"da árvore, pode-se descobrir ancestrais dessa espécie, bem como onde as linhagens compartilham uma ascendência comum. Além disso, a árvore pode ser usada para estudar grupos inteiros de organismos.[8]

Na reconstrução da árvore da vida, é feito o uso de métodos de simulação, onde há a repetição aleatória de amostras para modelar fenômenos com significativa incerteza nos dados de entrada. Todavia, tais simulações e comparações traz consigo um alto custo de replicação computacional. Tal barreira, impediram os biólogos por muito tempo de empregar, de forma plena, os métodos de simulação em estudos evolucionários. Para se considerar a incerteza filogenética em análises estatísticas, é necessário um grande conjunto de árvores, as quais possuem as espécies com incertezas filogenéticas(PUT, phylogenetic uncertain taxa) disponíveis, e as quais deveriam ser distribuídas aleatoriamente para as árvores parcialmente conhecidas.[12] Todavia, sabe-se que o ponto de inserção de cada novo PUT não é totalmente aleatório dentro da filogenia, pois cada espécie deve ser inserida em uma parte específica da árvore. Dessa maneira, é necessário estabelecer, para cada PUT, o MDCC(most derived consensus clade). Podemos entender então o MDCC como uma subárvore, que é conhecida, e na qual é possível inserir novas espécies, restringindo assim o âmbito de atribuição aleatória.

A inserção das novas espécies é de suma importância, pois é dessa maneira que é feita a reconstrução da árvore da vida. Sendo assim, temos que as informações contidas na topologia de cada árvore são de grande interesse para os biólogos e suas análises, visto que inferência de processos evolutivos são derivadas a partir de relacionamentos filogenéticos entre espécies.[8] Tais relacionamentos podem ser observados a partir da distância filogenética entre os pares de espécies e armazenados em uma matriz de distância, ou como chamamos, matriz de distância patrística. Com os dados armazenados, é possível então realizar diversos cálculos e análises, como o I de Moran, o qual é o foco deste trabalho, assim como o Parse.

O presente trabalho tem como foco, contribuir no processo de análises de árvores já replicadas, com suas devidas matrizes de distância calculadas, visando entregar para o biólogo, da forma mais simples e completa, o índice de I de Moran de cada árvore, visando assim facilitar o reconhecimento sobre qual seria a árvore com a representação mais real de uma árvore da vida atualizada, com os novos PUTs inseridos nas posições mais adequadas. Utilizaremos como abordagem, novas técnicas de computação paralela, apresentando algoritmos paralelos, que fazem de uma forma rápida e eficiente, todos os cálculos necessários.

CAPÍTULO 2

Filogenia

Desde os tempos de Darwin, métodos de comparação tem sido a principal ferramenta utilizada para o entendimento de padrões evolutivos e processos baseados na diversidade atual. Nesse processo, temos que a origem da similaridade é a ancestralidade comum, criando então uma dependência entre espécies, não sendo possível analisar estatisticamente uma espécie de forma isolada.[9]

Geralmente, ecologistas preferem depender de uma única árvore filogenética para estudar os processos evolutivos que afetam os padrões macroecológicos. Todavia, na maioria das vezes, estudos ecológicos e biogeográficos envolvem centenas ou milhares de espécies e árvores, representando linhagens recentes e antigas. Sendo assim, dado um conjunto de espécies e informações que caracterizam diferentes grupos de organismos, por exemplo o formato e tamanho das asas de aves, como podemos retirar informações sobre a relação entre as espécies as quais estas características foram observadas? É de extrema raridade que as relações entre espécies e ancestrais sejam observáveis diretamente.[10]

2.1 Método de Comparação Filogenético

Os métodos comparativos filogenéticos usam informações sobre as relações históricas de linhagens, chamadas filogenias ou árvore filogenética, para testar hipóteses evolutivas, padrões de variação e outras análises entre espécies. O método comparativo tem uma vasta história na biologia evolutiva. Em "A Origem das Espécies", Charles Darwin usou diferenças e semelhanças entre as espécies como uma grande fonte de evidência para provar sua teoria; Essa realização inspirou o desenvolvimento de métodos de comparação explícitos.[5] Os biólogos usam esse método para aplicar informações filogenéticas para estudar como a seleção natural tem no decorrer dos anos, moldado características das espécies, bem como suas variações. Além disso, o uso da filogenia pode ser importante para a reconstrução e compreensão os fatos da história.

Métodos comparativos filogenéticos geralmente são divididos em dois tipos principais de abordagens:

1. Inferem a história evolutiva de algum caráter(característica ou genética) através de uma filogenia.
2. Inferem o processo de ramificação evolutiva em si.

Normalmente, a árvore que é usada em conjunto com o processo de comparação foi estimada independentemente, de tal forma que tanto as relações entre linhagens quanto o comprimento dos ramos que as separam são assumidas como sendo conhecidas.[19]

2.2 Dados Utilizados

Na Biologia, é comum dividir os organismos de interesse em partes diferentes, sejam estas partes, comportamentos, aspectos, características, funções, entre outros. Todavia, tais partes isoladas não necessariamente apresentam significado biológico. Dessa forma, tem-se que os dados comparativos podem ser obtidos em estudos da Biologia, seja na Ecologia, Fisiologia, Genética ou qualquer outra. Tais dados são então utilizados para observações em um dado nível da hierarquia biológica.

Para se utilizar o método comparativo, como fizemos em nossa aplicação, é necessário se ter uma árvore filogenética, ou árvore da vida. É necessário também as espécies com filogenia incerta(PUT) junto com seu devido MDCC e também um vetor de características das espécies, o qual é usado para o cálculo do índice de I de Moran. Tais requerimentos são imprescindíveis para a realização do método.

2.2.1 Árvore Filogenética

Uma árvore evolutiva é um diagrama de ramificação, ou simplesmente árvore, como é chamado na ciência da computação, que mostra as relações evolucionárias inferidas entre diversas espécies biológicas, baseado em semelhanças e diferenças em suas dadas características, físicas ou genéticas. As árvores filogenéticas são de suma importância para o campo da filogenética e para o entendimento da evolução como um todo.[6]

Em uma árvore filogenética, cada nó com descendentes representa o antepassado comum mais recente dos descendentes, e os comprimentos de ramo em algumas árvores podem ser interpretados como estimativas de tempo entre o surgimento de cada espécie. Os ramos das árvores que se bifurcam representam então a existência da evolução que transformou aquela espécie em um ponto de cladogênese. A figura 1.1 ilustra graficamente a estrutura de uma árvore filogenética. Cada nó é chamado uma unidade taxonômica. Nódulos internos são geralmente chamados de unidades taxonômicas hipotéticas, pois não podem ser observados diretamente. As árvores são úteis em campos da biologia como bioinformática, sistemática e métodos comparativos filogenéticos.[14]

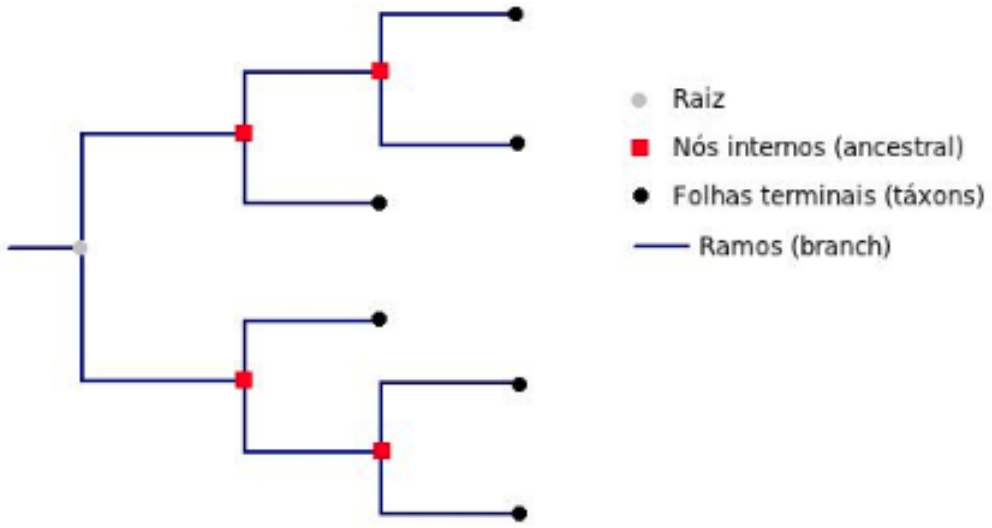


Figura 2.1: *Árvore filogenética [4]*

2.2.2 Formato Newick

Na ciência da computação, uma árvore é um tipo particular de grafo, ou seja, uma estrutura que contém nós conectados por arestas. Para que as árvores filogenéticas possam então ser usadas em sistemas computacionais é necessário criar uma alternativa a apresentação gráfica. No ramo da bioinformática, um formato de representação bem usado é o padrão Newick. Tal padrão é bastante utilizado por programadores, por conta da sua simplicidade e portabilidade. Todas as informações, como espécies, descendentes, comprimento de ramos e identificação do nó raiz são facilmente identificados e obtidos, bastando apenas fazer uma varredura na estrutura de dados.

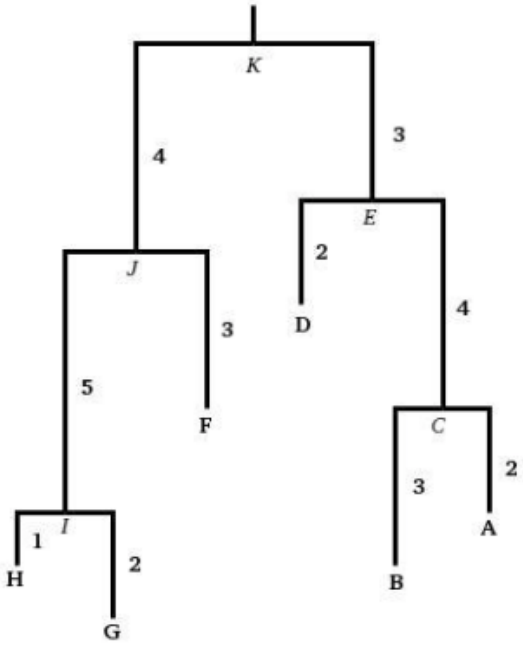
A árvore termina sempre com um ponto-e-vírgula. Os nós interiores são representados por um par de parênteses. Entre eles, estão os nós que são imediatamente descendentes desse nó, separados por vírgulas. Comprimentos de ramificação podem ser incorporados em uma árvore, colocando um número real, com ou sem ponto decimal, após um nó e precedido por dois-pontos. Isso representa o comprimento do ramo imediatamente abaixo desse nó. A figura 2.2 representa graficamente as formas de representação de uma árvore filogenética gráfica e textual.[11]

2.3 Coeficiente I de Moran

O I de Moran é um coeficiente de correlação, que mede a autocorrelação espacial do conjunto de dados. Nesse caso, é possível analisar os padrões filogenéticos através desse índice. Em outras palavras, é possível medir como uma espécie é semelhante às outras que a rodeiam. O índice de Moran, dado na equação 1, é utilizado para auxiliar

```
(((A:2,B:3)C:4,D:2)E:3,(F:3,(G:2,H:1)I:5)J:4)K;
```

(a) *Representação no formato Newick*



(b) *Representação gráfica*

Figura 2.2: *Árvore filogenética e suas formas de representação [4]*

na compreensão dessa semelhança. A autocorrelação espacial é multidirecional e multidimensional, tornando-a útil para encontrar padrões em conjuntos de dados complicados. É semelhante aos coeficientes de correlação, tem um valor de -1 a 1. No entanto, enquanto outros coeficientes medem correlação perfeita com nenhuma correlação, Moran é ligeiramente diferente (devido aos cálculos espaciais mais complexos):[16]

- -1 é o agrupamento perfeito de valores diferentes (você também pode pensar nisso como dispersão perfeita).
- 0 não é autocorrelação (aleatoriedade perfeita.)
- +1 indica agrupamento perfeito de valores semelhantes (é o oposto da dispersão).

Formula:

1. $I = (\frac{n}{S}) \frac{(\sum_{i=1}^{n}(\sum_{j=1}^{n}(Wij(yi-ylinha)*(yi-ylinha))))}{\sum_{i=1}^{n}(yi-ylinha)^2}$

Onde:

- n: é o número de espécies
- y: representa a variável analisada
- ylinha: é a média de y

- Wij: é o valor 1 ou zero, que representa a conectividade.
- S: é a soma dos elementos de conectividade

Os cálculos do I de moran Moran são baseados em uma matriz ponderada(matriz de distância), com unidades i e j. As similaridades entre unidades são calculadas como o produto das diferenças entre yi e yj com a média geral.

CAPÍTULO 3

Computação Paralela

3.1 Visão Geral

Tradicionalmente, software tem sido escrito para uma computação serial, ou em outras palavras, a execução é feita de forma sequencial. Nessa linha de pensamento, temos que o problema é quebrado em uma série de instruções, as quais são executadas sequencialmente, uma após a outra. Tais instruções são executas em um único processador, e embora não conseguimos perceber, devido a velocidade de processamento, apenas uma única instrução é executada naquele exato momento. [3]

No decorrer dos anos, o computador tem passado por modificações e atualizações constantes, principalmente no que diz respeito a velocidade de processamento. Saímos de uma capacidade de processamento de CPU que girava em torno de 0,05 GHz em 1991 para notáveis 32 GHz em 2011. [7] Um crescimento que já chegou a ser mais de 60% ao ano, variando bastante na maioria das vezes entre 20% a 40%.

Atualmente, atividades recentes dos principais fabricantes de chips, como a NVIDIA, nos faz acreditar ainda mais que projetos futuros de microprocessadores e grandes sistemas HPC serão híbridos / heterogêneos por natureza. Tais sistemas heterogêneos dependerão fortemente da integração de dois tipos principais de componentes em proporções variáveis: [18]

- Multi-Core e Many-Core CPU: Certamente, o número de núcleos continuará a aumentar, devido ao desejo de se colocar cada vez mais componentes em um único chip, evitando ao mesmo tempo sua sobrecarga.
- Hardware de uso especial e operações paralelas massivas: Por exemplo, as GPUs da NVIDIA ultrapassaram o padrão das CPUs em desempenho de ponto flutuante nos últimos anos. Além disso, elas tem se tornado tão fáceis quanto, ou até mesmo mais fácil de se programar do que os CPUs multicore.

Tem-se que o equilíbrio relativo entre esses tipos de componentes em projetos futuros não é claro e provavelmente variará ao longo do tempo. Todavia, parece não haver

dúvidas de que as futuras gerações de sistemas informáticos, que vão desde computadores portáteis a supercomputadores, consistirão numa composição de componentes heterogêneos. Mesmo assim, os problemas e desafios para os desenvolvedores nessa nova paisagem computacional de processadores híbridos permanecem assustadores.[18]

3.1.1 Paralelismo

No decorrer dos últimos anos, as GPUs vem sendo evoluídas e seu uso cada vez mais popular e necessário. Em suas origens, ela era vista apenas como um processador gráfico especializados, que poderiam rapidamente processar e produzir imagens para uma unidade de exibição. Tal uso era bem comum em computadores gamers e consoles. Todavia, o conceito e uso das GPUs mudou, não limitando apenas a produção de imagens, mas também altamente ligada ao uso de processamento ultra-rápido de dados.[13] Como dito anteriormente, as GPUs tem sido cada vez mais ligadas a CPUs, com o intuito de acelerar a capacidade de computação, no chamado sistema heterogêneo. Para se ter uma melhor visão de sua necessidade, podemos perceber que nos dias atuais, GPUs são configuradas em cluster de computação, supercomputadores e até mesmo em muitos sistemas desktop. Seu principal papel se dá como fornecedor de grandes quantidades de poder computacional, devido a sua grande velocidade de processamento de grandes dados. Assim, GPUs têm permitido avanços importantes não apenas na ciência e engenharia, mas também em outras áreas, como a medicina. [13] Isso se dá pelo motivo de que com as GPUs, é possível colocar diversos núcleos de computação para trabalhar em paralelo.

Em uma forma simples de se explicar, podemos dizer que a computação paralela é a utilização simultânea de múltiplos recursos computacionais para resolver um problema computacional. Para isso, temos que, um problema é dividido em partes distintas que podem ser resolvidas simultaneamente, cada parte então é dividida em uma série de instruções, e as mesma são executadas simultaneamente em diferentes processadores. Para controlar tudo isso, é definido um mecanismo de controle global. Veja a figura 2.1

Um exemplo simples para se entender melhor como funciona seria o incremento de valores em um vetor. Se utilizarmos um algoritmo sequencial, provavelmente usaremos um "for"para percorrer todo o vetor e assim incrementar o valor de cada elemento, um a um sequencialmente. Entretanto, ao paralelizar esse problema, teríamos vários processadores disponíveis, nos possibilitando a construção de um algoritmo que particione cada posição do vetor para um processador, ou seja, cada thread terá o valor de uma posição do vetor. Sendo assim, cada processador poderá fazer o incremento de seu valor, simultaneamente com os outros.[3]

Temos que muitos problemas são tão grandes e complexos, que é impraticável ou impossível resolvê-los em um único computador. Percebe-se então que o grande objetivo

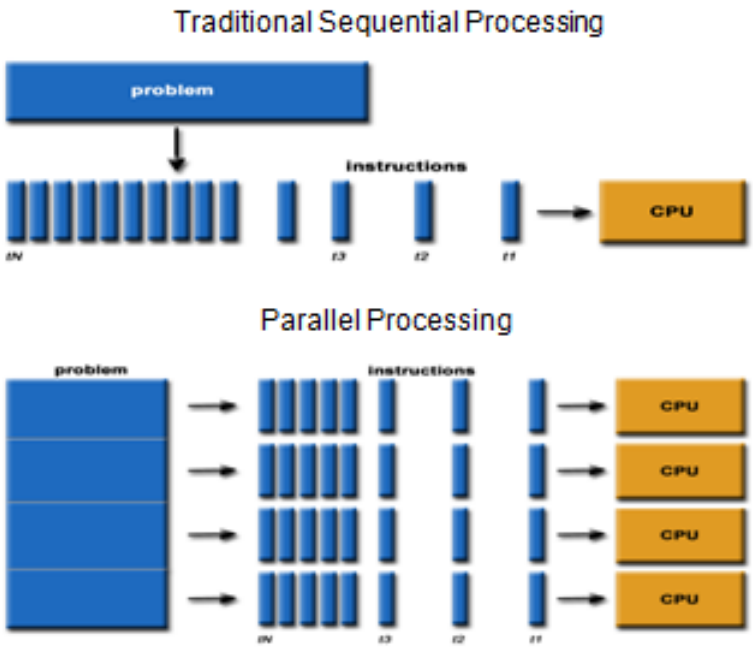


Figura 3.1: *Parallel x Serial programming representation [3]*

da computação paralela é resolver problemas complexos em um menor tempo, com um custo mais baixo, afinal os computadores paralelos podem ser construídos a partir de componentes de mercadorias de preços razoáveis e acessíveis.[3]

Para se ter uma paralelização eficiente, é preciso saber como mapear os cálculos simultâneos feitos na GPU, afinal, a chave da computação paralela é explorar de forma correta e eficiente a concorrência. Sabemos de fato que, a computação paralela geralmente envolve duas áreas distintas de tecnologias de computação:[13]

- Arquitetura de computadores(aspecto hardware): Se concentra em suportar o paralelismo em a nível de arquitetura. É necessário fornecer uma plataforma que suporte a execução simultânea de vários processos ou múltiplos threads, para se obter uma execução paralela em software.
- Programação paralela(aspecto software): Se concentra em resolver o problema, simultaneamente, explorando totalmente o poder computacional da arquitetura do computador e da GPU.

Hoje em dia, a computação paralela está se tornando onipresente, e seu uso essencial em diversas aplicações. Existem dois tipos fundamentais de paralelismo nas aplicações:[13]

- Paralelismo de tarefas: Surge quando há muitas tarefas ou funções que podem ser executadas independentemente e em grande parte paralelamente. Ou seja, há uma concentração na distribuição de funções em vários núcleos.

- Paralelismo de dados: Surge quando há muitos itens de dados que podem ser operados ao mesmo tempo. Sendo assim, cada tarefa executa uma mesma série de cálculos sobre diferentes dados. Ou seja, há uma concentração na distribuição dos dados em vários núcleos.

3.1.2 Taxonomia de Flynn

Existem várias maneiras de se classificar uma arquitetura de computador. Nos dias atuais, um esquema de classificação amplamente utilizado é a taxonomia de Flynn, definida em 1966[17], que classifica arquiteturas em quatro tipos diferentes, a partir de como as instruções e os dados fluem através de núcleos. Flynn propôs a arquitetura com as seguintes categorias:

- Única Instrução, único dado (SISD - single instruction, single data): Este é o tipo mais antigo de computador, uma arquitetura serial. Existem apenas um núcleo no computador. Em qualquer momento, apenas um fluxo de instruções é executado e as operações são operadas em um fluxo de dados. A figura 2,2 ilustra graficamente a estrutura e visão da arquitetura.
- Única instrução, múltiplos dados (SIMD - single instruction, multiple data): Refere-se a um tipo de arquitetura paralela. Existe apenas uma unidade de controle, porém vários núcleos no computador. Todos os núcleos executam o mesmo fluxo de instruções a qualquer momento, cada um operando em fluxos de dados diferentes. A maioria dos computadores modernos empregam a arquitetura SIMD. Uma grande vantagem desse modelo é que, ao escrever o código na CPU, os programadores podem continuar a pensar de forma sequencial, e ao mesmo tempo, obter um bom ganho de velocidade paralela de operações, devido os detalhes serem responsabilidade do compilador. A figura 2.3 ilustra graficamente a estrutura e visão da arquitetura.
- Múltiplas instruções, único dado (MISD - multiple instruction, single data): Cada núcleo opera no mesmo dato atravês de instruções distintas. É uma arquitetura incomum e de pouco uso prático. A figura 2.4 ilustra graficamente a estrutura e visão da arquitetura.
- Múltiplas intruções, múltiplos dados (MIMD - multiple instruction, multiple data): Refere-se a um tipo de arquitetura paralela, assim como SIMD. Porém nesse modelo, múltiplos núcleos opera em múltiplos dados, cada um executando instruções independentes. Muitas das arquiteturas MIMD inclui também SIMD. A figura 2.5 ilustra graficamente a estrutura e visão da arquitetura.

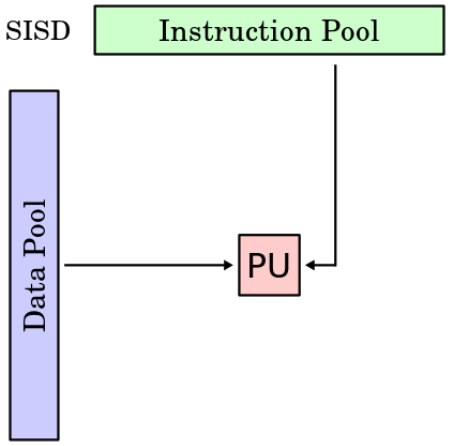


Figura 3.2: *SISD Representation [17]*

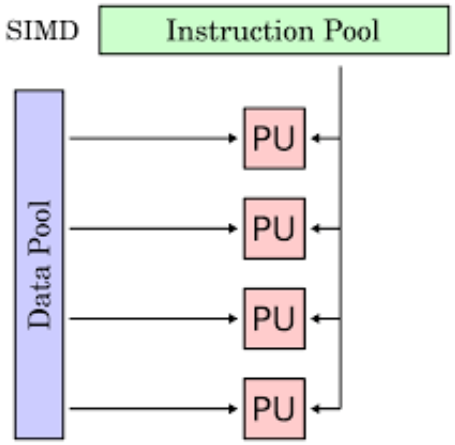


Figura 3.3: *SIMD Representation [17]*

3.1.3 Arquiteturas de computação paralela

A nível de computação paralela, os modelos de arquiteturas mais usados são SIMD e MIMD. Quando há a necessidade de haver apenas uma unidade de controle, onde todos os processadores executam a mesma instrução, então a arquitetura SIMD é usada. Todavia, quando se tem mais de um processador onde cada um tem sua própria unidade de controle e pode consequentemente executar diferentes instruções sobre diferentes dados, então a arquitetura MIMD é usada.

O sistema SIMD compreende uma das três classes de computadores paralelos com maior sucesso comercial. Uma série de factores contribuíram para este êxito, incluindo: [17]

- Simplicidade de conceito e programação.
- Regularidade da estrutura.
- Escalabilidade fácil de tamanho e desempenho

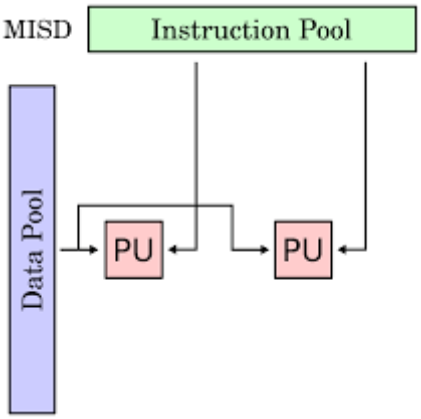


Figura 3.4: *MISD Representation [17]*

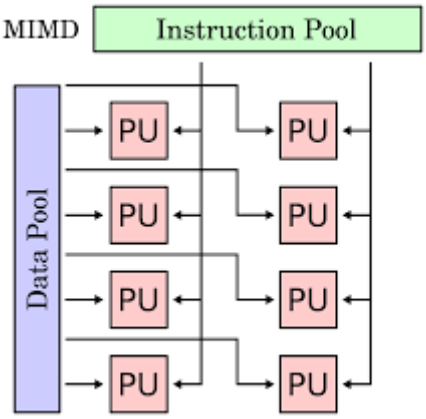


Figura 3.5: *MIMD Representation [17]*

- Aplicabilidade direta em vários campos que exige paralelismo para atingir o desempenho necessário.

Nesse modelo, existe um conjunto bidimensional de elementos de processamento, cada um conectado a seus quatro vizinhos mais próximos, todos os processadores executam a mesma instrução simultaneamente e cada processador incorpora memória local. Além disso, os processadores são programáveis, isto é, eles podem executar uma variedade de funções e os dados podem propagar rapidamente através da matriz.

Há duas principais maneiras de se configurar máquinas de arquitetura SIMD.[13]

- Arquitetura SIMD com memória distribuída:
  - Possui uma unidade de controle que interage com cada elemento de processamento na arquitetura.
  - Cada processador possui sua própria memória local, conforme observado na Fig 2.6.

  - Os elementos do processador são utilizados como uma unidade aritmética onde as instruções são fornecidas pela unidade de controle. Para que um elemento de processamento se comunique com outra memória na mesma arquitetura, para obter informações por exemplo, ele terá que adquiri-la através da unidade de controle.
  - A grande inconveniência é com o tempo de desempenho, pois a unidade de controle tem de lidar com as transferências de dados.

- Arquitetura SIMD com memória compartilhada: necessário.
  - Nesta arquitetura, um elemento de processamento não tem uma memória local, mas em vez disso está conectado a uma rede onde ele pode se comunicar com um componente de memória. A Figura 2.7 mostra todos os elementos de processamento conectados à mesma rede, o que lhes permite compartilhar seu conteúdo de memória com outros.
  - A desvantagem nesta arquitetura é que, se houver a necessidade de expandir essa arquitetura, cada módulo (elementos de processamento e memória) deve ser adicionado separadamente e configurado.
  - No entanto, esta arquitectura é ainda benéfica, uma vez que melhora o tempo de desempenho e as informações podem ser transferidas mais livremente sem a unidade de controlo.

O sistema MIMD, que significa Multiple Instruction, Multiple Data é a forma mais simples e possivelmente a mais básica de processamento paralelo. Sua arquitetura consiste em uma coleção de N processadores independentes, cada um com uma memória, que pode ser comum a todos os outros processadores. Há duas principais maneiras de se configurar máquinas de arquitetura MIMD.[20]

- Arquitetura MIMD com memória compartilhada:
  - Cria um conjunto de processadores e módulos de memória.
  - Qualquer processador pode acessar diretamente qualquer módulo de memória através de uma rede de interconexão conforme observado na Fig 2.8.
  - O conjunto de módulos de memória define um espaço de endereço global que é compartilhado entre os processadores.

- Arquitetura MIMD com memória distribuída:
  - Ela Replica os pares de processador/memória e os conecta através de uma rede de interconexão. O par de processador/memória é chamado de elemento de processamento.
  - Cada elemento de processamento podem interagir entre si através do envio de mensagens, como pode ser observado na figura 2.9.

Nos últimos anos, a nível de arquitetura, foram feitos muitos avanços para atingir objetivos como: [13]

- Diminuir a latência: É o tempo que leva para uma operação iniciar e concluir, e é comumente expressa em microssegundos. Ou seja, ela mede o tempo para concluir uma operação.
- Aumentar a largura de banda(bandwidth): É a quantidade de dados que podem ser processados por unidade de tempo, normalmente expressos em megabytes/s ou gigabytes/s.
- Aumentar o rendimento(throughput): É a quantidade de operações que podem ser processadas por unidade de tempo, comumente expressa como gflops. Ou seja, mede o número de operações processadas em uma determinada unidade de tempo.

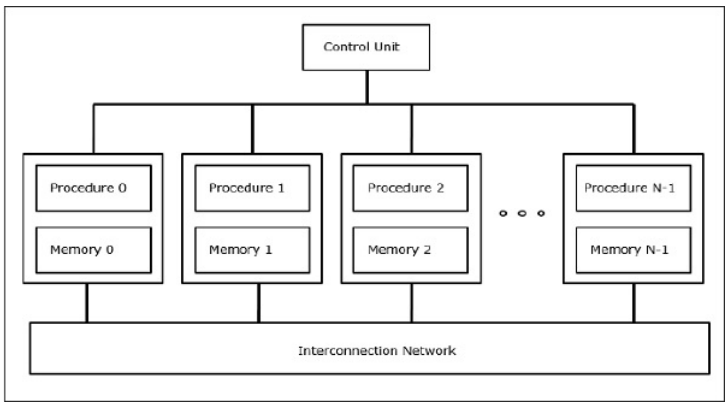


Figura 3.6: *SIMD Distributed Memory Architecture [13]*

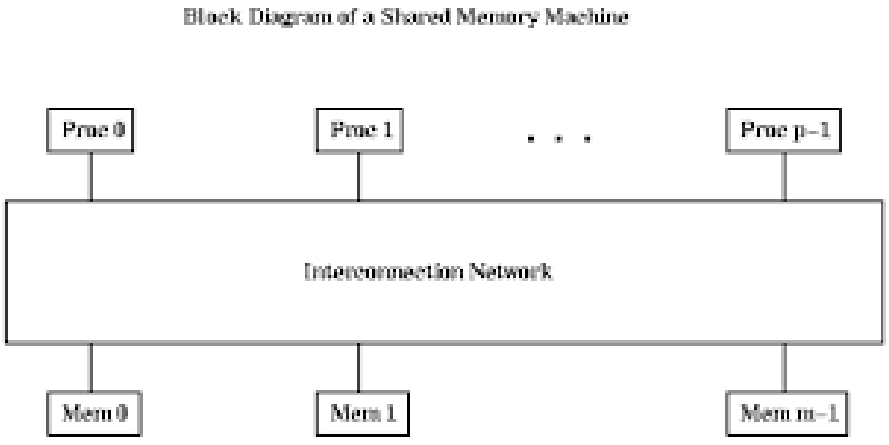


Figura 3.7: *SIMD Shared Memory Architecture [13]*

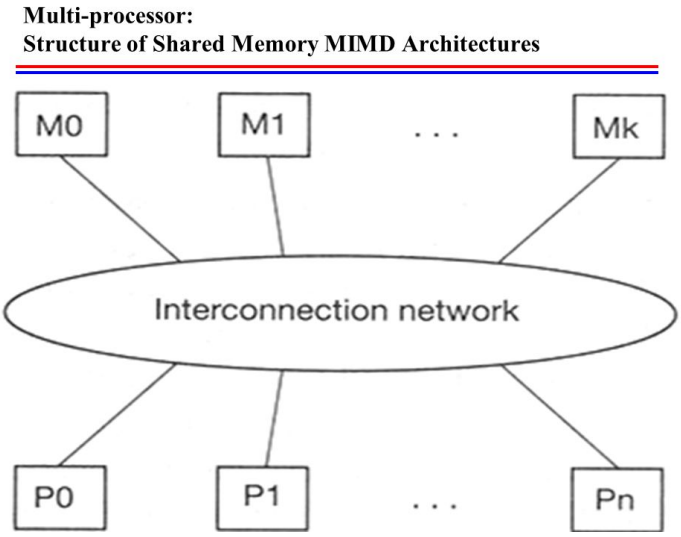


Figura 3.8: *MIMD Shared Memory Architecture [13]*

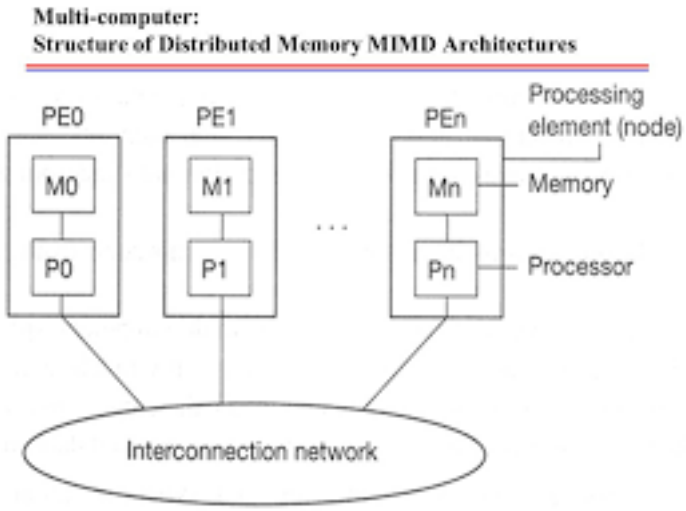


Figura 3.9: *MIMD Distributed Memory Architecture [13]*

3.1.4 Arquitetura heterogênea

Atualmente, a computação heterogênea consiste em dois sockets de CPU multi-core e dois ou mais GPUs de múltiplos núcleos. Hoje em dia, uma GPU não é uma plataforma autônoma, mas um co-processador para uma CPU. Portanto, as GPUs devem operar em conjunto com um host baseado em CPU através de um barramento PCI-Express. Por esse motivo, em termos de programação e conceitos, a CPU é chamada de host e a GPU é chamada de dispositivo(device).[13]

Como dito anteriormente, uma aplicação heterogênea consiste em 2 partes, código do host e código do dispositivo. O código host roda exclusivamente nas CPUs e código do dispositivo nas GPUs. Um aplicativo executando em uma plataforma heterogênea é tipicamente inicializado pela CPU. O código da CPU é responsável por gerenciar o ambiente, o código e os dados do dispositivo antes de carregar tarefas de computação

intensiva no dispositivo(GPU). Com aplicações computacionais intensivas, as seções do programa frequentemente exibem uma grande quantidade de paralelismo de dados. As GPUs são usadas para acelerar a execução desta porção de paralelismo de dados. Quando um componente de hardware que é fisicamente separado da CPU é usado para acelerar seções computacionalmente intensivas de um aplicativo, é referido como um acelerador de hardware. GPUs são, sem dúvida, o exemplo mais comum de um acelerador de hardware.[17]

3.1.5 CPU x GPU

A computação em GPU não se destina a substituir a computação em CPU. Cada abordagem tem vantagens para certos tipos de programas. A computação da CPU é boa para tarefas de controle intensivo, e a computação GPU é boa para tarefas de computação intensiva de dados paralelas. Quando CPUs são colocadas juntas com GPUs, é gerado uma combinação poderosa. A CPU é otimizada para cargas de trabalho dinâmicas marcadas por curtas sequências de operações computacionais e controles imprevisíveis; E GPUs visam a outra extremidade do espectro: cargas de trabalho que são dominadas por tarefas computacionais com controle simples de execução(flow). Existem duas dimensões que diferenciam o escopo de aplicativos para CPU e GPU: Nível de paralelismo e tamanho dos dados.[18]

Se um problema tem um tamanho de dados pequeno, lógica de controle sofisticada e / ou paralelismo de baixo nível, a CPU é uma boa escolha por causa de sua capacidade de lidar com lógica complexa e paralelismo de nível de instrução. Se o problema na mão, em vez disso, processa uma enorme quantidade de dados e exibe um paralelismo de dados maciço, a GPU é a escolha certa porque tem um grande número de núcleos programáveis, pode suportar multi-threading massivo e tem uma largura de banda de pico maior comparada com a CPU.

As arquiteturas de computação paralela heterogêneas de CPU + GPU evoluíram porque a CPU e a GPU possuem atributos complementares que permitem que os aplicativos desempenhem melhor com os dois tipos de processadores. Portanto, para um ótimo desempenho, você pode precisar usar tanto a CPU como a GPU para sua aplicação, executando as partes sequenciais ou as partes paralelas da tarefa na CPU e partes intensivas de dados paralelos na GPU.

3.1.6 Métricas de desempenho

Quando desenvolvemos algoritmos a serem executados em uma arquitetura paralela, é sempre esperando uma boa eficiência. Por exemplo, quando é criado um algoritmo paralelo em uma arquitetura que contém 8(oito) processadores, é então esperado

um desempenho de pelo menos 8(oito) vezes mais rápido se comparado a uma arquitetura mono-processada. Entretanto, nem sempre é assim que ocorre, podendo inclusive ocorrer o contrário, um desempenho inferior. Isso se dá porque o fato de se ter mais processadores não te garante uma maior eficiência, pois há uma redução no tempo de execução, devido o grau de dependência existentes entre os processadores e as tarefas paralelizadas.

Sendo assim, ao se pensar no desempenho de um algoritmo paralelo, tem que se levar em conta todas as situações e cenários. Por exemplo, o problema pode ser totalmente paralelizável? Como vai ser a comunicação entre os processadores? Qual o tamanho de memória compartilhada será usada? Sendo assim, é usado alguns conceitos e métricas para a análise da eficiência e desempenho computacional do algoritmo paralelo, sao elas: eficiência, aceleração, custo, granularidade e escalabilidade.[20]

Eficiência

A eficiência relaciona o speedup com o número de processadores, ou seja, identifica a utilização do processador. Todavia, o comportamento ideal não é conseguido porque ao executar um algoritmo paralelo, os elementos de processamento não podem dedicar 100% de seu tempo aos cálculos do algoritmo. Em um sistema paralelo ideal a eficiência é igual a um porém, na prática, a eficiência está entre zero e um. [20]

Aceleração(speedup)

Com um problema dado e paralelizado, quanto foi o ganho de performance se comparado a uma implementação sequencial? Speedup é uma medida que capta o benefício relativo em se resolver um problema em paralelo.

Speedup, S, pode ser definido como a razão entre o tempo gasto para resolver um problema em um computador serial e o tempo necessário para resolver o mesmo problema em um computador paralelo. Por exemplo, considerando o exemplo do bubble sort, assumimos que:

- Versão serial do bubble sort de 105(MATH elevado - falta fazer) registros leva 150 s
- Uma versão quicksort serial pode resolver o mesmo problema em 30 s.
- Uma versão paralela do bubble sort Leva 40 segundos em 4 núcleos.

Parece que a versão paralela do algoritmo resulta em uma aceleração de 150/40 = 3,75. Todavia, esta conclusão é enganosa. O algoritmo paralelo resulta em uma aceleração de 30/40 = 0,75 em relação ao melhor algoritmo serial. Quando é medido o Speedup, deve se usar a melhor versão serial do problema, para se ter um ganho real.

Custo

Na computação paralela, custo é definido pelo produto entre o tempo de execução paralelo e o número de processadores. O custo reflete a soma do tempo que cada processador gasta para resolver o problema.

Granularidade

Uma chave para alcançar um bom desempenho paralelo é escolher a granularidade certa para a aplicação. Granulosidade na computação paralela pode ser definida como o tamanho das unidades de trabalho submetidas aos processadores, ou em algumas vezes, como a quantidade de trabalho realizado entre as iterações do processador.

A granulosidade pode ser fina, média ou grossa. Na granulosidade fina ou fine-grain identificamos que houve a decomposição em um grande número de pequenas tarefas, normalmente implicando em considerável comunicação entre os processadores. Já na granulosidade grossa ou coarse-grain, a decomposição é feita num pequeno número de grandes tarefas, o que pode implicar em baixa comunicação entre os processadores. Ao desenvolver um algoritmo em paralelo, devemos analisar qual o melhor caminho. Se granularidade é muito fina, o desempenho pode sofrer de sobrecarga de comunicação. Se a granularidade é muito grosseira, então o desempenho pode sofrer de desequilíbrio de carga. O objetivo é determinar a granularidade certa (geralmente maior é melhor) para tarefas paralelas, evitando o desequilíbrio de carga e sobrecarga de comunicação para obter o melhor desempenho.[20]

Escalabilidade

Por escalabilidade, entendemos como a capacidade de aumentar o desempenho á medida que a complexidade do problema aumenta, isso a nível de hardware ou ao software. A nível de hardware, escalável é uma característica de um arquitetura onde a ampliação do seu tamanho (inclusão de mais sistemas computacionais) tem consequência no aumento do seu desempenho. Já a nível de software, o algoritmo é escalável quando pode acomodar o aumento do tamanho do problema com um aumento baixo dos passos de computação a ser realizados.

3.2 Ambientes de programação

As primeiras GPUs foram projetadas como aceleradores gráficos, suportando somente tubulações específicas de função fixa. Começando no final dos anos 90, o hardware tornou-se cada vez mais programável, culminando com o primeiro GPU da NVIDIA em 1999. Mas GPGPU estava longe de ser fácil na época, mesmo para aqueles

que sabiam linguagens de programação gráfica, como OpenGL.[15] Em 2003, uma equipe de pesquisadores liderada por Ian Buck revelou Brook, o primeiro modelo de programação amplamente adotado para estender C com construções de dados paralelos. Usando conceitos como fluxos, kernels e operadores de redução, o compilador Brook e o sistema runtime expuseram a GPU como um processador de propósito geral em uma linguagem de alto nível. Mais importante ainda, os programas do Brook não eram apenas mais fáceis de escrever do que o código GPU manualmente ajustado, eram sete vezes mais rápidos do que o código existente semelhante. A NVIDIA sabia que o hardware incrivelmente rápido precisava ser acoplado a ferramentas intuitivas de software e hardware e convidou Ian Buck para se juntar à empresa e começar a desenvolver uma solução para executar C na GPU. Juntando o software e o hardware, a NVIDIA revelou a CUDA em 2006, a primeira solução do mundo para computação geral em GPUs.[17]

3.2.1 CUDA

CUDA® é uma plataforma de computação paralela e modelo de programação inventado pela NVIDIA. Permite aumentos dramáticos no desempenho de computação, aproveitando a potência da unidade de processamento gráfico (GPU). Com milhões de GPUs CUDA espalhadas pelo mundo até agora, os desenvolvedores de software, cientistas e pesquisadores estão encontrando amplos usos para computação GPU com CUDA e obtendo grandes avanços em suas aplicações e pesquisas.[15] Em linhas gerais, podemos resumir CUDA como uma hierarquia de threads mapeadas para os processadores de uma GPU.

- Parcelas paralelas da aplicação são executadas no dispositivo(device) como kernels
  - 1(um) kernel é executado de cada vez
  - Muitas threads executam cada kernel
- Threads em CUDA são extremamente leves
  - Muito pouca sobrecarga de criação
  - Comutação rápida
- CUDA usa 1000s de threads para alcançar uma boa eficiência
- Um kernel CUDA é executado por um array de threads
  - Todos as threads executam o mesmo código
  - Cada thread tem um ID que ele usa para calcular endereços de memória e tomar decisões de controle
- 1(um) SM (streamming multiprocessor) executa um ou mais blocos de threads e os CUDA Cores (ou SPs) e outras unidades de execuc?a?o que compo?em um SM

executam as threads. O SM executa threads em grupos de 32 threads, chamado warp.

3.2.2 Estrutura de um programa em CUDA

Um programa CUDA consiste de uma mistura das duas partes, código host que é executado pela CPU e código do dispositivo que é executado na GPU, afinal CUDA roda em um sistema heterogêneo.[13]

O compilador CUDA nvcc da NVIDIA separa o código do dispositivo do código do host durante o processo de compilação. O código do host é um código C padrão e é compilado com compiladores C. O código do dispositivo é escrito usando CUDA C estendido com palavras-chave para rotular funções paralelas de dados, chamadas kernels. O código do dispositivo é compilado usando nvcc. Durante o estágio de link, as bibliotecas de tempo de execução CUDA são adicionadas para chamadas de procedimento do kernel.[18]

Em um típico programa NVIDIA CUDA C é comum a intercalação de código sequencial e paralelo. Geralmente, é indicado a seguir a seguinte estrutura para a criação do código usando CUDA:

- Alocar memória para a GPU.
- Copiar os dados de memória da CPU para a memória da GPU.
- Invocar o kernel CUDA para executar a computação específica do programa.
- Copiar os dados da memória da GPU para a memória da CPU.
- Destruir memórias alocadas para a GPU.

3.2.3 Desafios Programação em CUDA C

A principal diferença entre a programação da CPU e a programação da GPU é o nível de exposição do programador aos recursos arquitetônicos da GPU. Pensar em paralelo e ter uma compreensão básica da arquitetura GPU permite que você escreva programas paralelos que escalam centenas de núcleos tão facilmente quanto você escreve um programa sequencial.

Para escrever um código eficiente, é preciso ter um conhecimento básico de arquiteturas de CPU. Por exemplo, a localidade(locality) é um conceito muito importante na programação paralela. Localidade refere-se à reutilização de dados de modo a reduzir a latência de acesso à memória. As arquiteturas de CPU modernas usam caches grandes para otimizar aplicações com boa localidade espacial e temporal. É responsabilidade do programador projetar seu algoritmo para usar eficientemente o cache da CPU. Os programadores devem lidar com otimização de cache de baixo nível, mas não têm

introspecção em como os segmentos estão sendo agendados na arquitetura subjacente porque a CPU não expõe essas informações.[18]

É preciso também estar ciente sobre os diferentes tipos de memórias usadas na programação paralela, tais como memória compartilhada e local. Memória compartilhada é exposta pelo modelo de programação CUDA e pode ser considerada como um cache gerenciado por software, que fornece grande velocidade, conservando a largura de banda na memória principal. Com a memória compartilhada, você pode controlar a localidade do seu código diretamente. Já a memória local tem escopo somente de uma thread e fica localizada na memo?ria global. Estrutura de vetores que ocupam grande quantidade de registradores sa?o tipos de varia?veis candidatas a serem colocadas nesta memo?ria. Por residir na memo?ria global, essa memo?ria possui a mesma late?ncia e largura de banda dessa memo?ria.

CAPÍTULO 4

Proposta de Implementação

O problema de filogenia é complexo e exige um alto custo computacional. Com isso, temos que desenvolver uma solução eficaz e eficiente, que realize todos os passos computacionais necessários no menor tempo possível, sempre mantendo a qualidade e persistência dos dados e dos resultados. Para a criação da solução, o problema foi dividido em quatro partes, todavia, como dito anteriormente eu entrei no meio do projeto e participei então de apenas duas das quatro etapas, pré-processamento e cálculo de I de Moran.

- Pré-Processamento: Um biólogo nos disponibiliza uma árvore (arquivo formato Newick). A partir desse arquivo, iremos ler, validar, e gerar vetores com todas as informações necessárias para fazer o processamento em paralelo.
- Inserção de espécies e geração de árvores(réplicas): Essa etapa consiste na inserção de espécies novas, as quais não se tem muitas informações, sob uma árvore já conhecida. Tal inserção se deve a partir do arquivo put, que contém todas as novas espécies que devem ser inseridas em alguma sub árvore. É gerado então variações/réplicas das árvores, mudando sempre, aleatoriamente, o local de inserção da nova espécie naquela sub árvore, a partir do seu MDCC.
- Cálculo de matriz de distância: Nessa parte da solução, é calculado a matriz de distância evolutiva, isto é, a matriz contendo as distâncias entre todos os pares de espécies, para cada árvore construida no passo de inserção de espécies. As matrizes de distância são base para extrair informações estatísticas sobre filogenia.
- Cálculo I de Moran: O índice I de Moran é um índice geral para medir autocorrelação. Ele é uma extensão de Pearson Coeficiente de correlação produto-momento para uma série univariável (Moran - Autocorrelation Coefficient in Comparative Methods). Para realizar esse cálculo é necessário ter a matriz de distância, pois ela é usada como base. Sendo assim, para cada matriz de distância se tem um valor de índice de correção por classe de distância.

4.1 Pré-processamento

Toda a estrutura de resolução do problema, como vamos tratar o problema, qual abordagem iremos seguir, começa no pré-processamento. Essa etapa é fundamental para obtermos uma boa paralelização, pois é aqui que preparamos os dados de entrada em uma estrutura adequada e pensada, para uma futura manipulação dos dados na GPU. No pré-processamento, temos que a entrada de dados é composta por:

- Uma árvore filogenética, mundialmente conhecida pelos biólogos como Newick.
- PUT (phylogenetically uncertain taxa), é o arquivo que contém todas as espécies desconhecidas e faltantes na árvore Newick.
- Traits. Representa as características das espécies, tanto as que já estão no Newick como as do PUT. Essas características são usadas no cálculo de I de Moran.

Todo o conjunto de dados(Newick, PUT e Traits) devem ser lidos por uma função, e tratados da melhor forma possível, como dito anteriormente. Para isso, fazermos a realização do parsing. No ambiente de computação, temos que o parsing é o processo de analisar uma sequência de entrada para, posteriormente, ser representado em outro formato, o qual melhor atende a aplicação. No parsing, todos os arquivos de entrada são carregados e mapeados em uma estrutura definida por nós, que é apropriada para usarmos na computação paralela.

Para um melhor entendimento, usaremos em nossos exemplos um modelo de árvore simplificada, que está representada a partir do formato Newick. Usaremos então uma árvore pequena, pois assim conseguimos demonstrar melhor, de uma forma real, como é estruturado a árvore na nossa aplicação, após o parsing.

Nós escrevemos um parsing em C++, assim como todo o resto da aplicação. O código faz a leitura de todos os arquivos de entrada e realiza o devido processamento. No caso do Newick, a leitura é feita linha por linha e então colocada em uma estrutura interna, um array de String, isso facilita o manipulamento futuro. Na manipulação, fizemos uma leitura pós-ordem reversa e processamos a árvore da forma como ele se apresenta no formato Newick. Isso facilita tanto na manipulação, quanto na organização. Por exemplo, no arquivo exemplo temos uma sequência A, B, C, D, E, F, G, H, I, J e K; Isso significa que os filhos sempre são lidos antes dos pais e a raiz sempre será a última a ser lida e processada. Para realizar esse processamento pós-ordem, utilizamos a técnica de expressão regular(regex em C++), pois assim conseguirmos retirar uma parte específica da árvore, sem modificar sua estrutura. Esse processo fica mais evidenciado no algoritmo 3.1

Ao sair do parsing, temos uma estrutura de vetores, pois achamos essa a melhor ideia, devido favorecer o acesso aglutinado a memória. Sendo assim, cada vetor terá partes

das informações sobre a árvore. Por exemplo, um vetor terá a informação dos nomes dos nós, outra vetor o comprimento dos ramos, sobre o pai de cada nó, seus dois sucessores, etc. Outra informação importante e que armazenamos em um vetor separado, são os dados referentes a característica da espécie. Isso facilitará quando for feito o cálculo de I de Moran. Observe a figura 3.1 onde é mostrado como as informações ficariam organizadas em vetores separados. Além disso, observe que fizemos uma separação no vetor, para facilitar o entendimento e uso. Os ancestrais nós armazenamos nos índices mais baixos do vetor e informações sobre os ancestrais nos índices mais altos do vetor, sendo a raiz o último elemento do vetor. Dessa forma, através de uma simples comparação do índice do vetor com o número espécies determinamos se o elemento é uma espécie ou um ancestral.

Podemos ver, através da figura 3.1, que as espécies A, B, D, F, G e H foram mapeadas para as seis primeiras posições do vetor que representa a árvore (espécies iniciais). Já, os ancestrais K, E, C, J e I foram mapeados para as últimas 5 posições (ancestrais iniciais). Observe que utilizamos o valor “-2“, no vetor “filho esquerda“ e “filho direita“, para indicar que o nó não tem filhos, se o valor for “-1“ (vetor pai) indica que este é o nó raiz, que não tem pai (ancestral). [4] Observe que alguns espaços são reservadas para as novas espécies a serem incluídas. Essas espécies são armazenadas nas posições seguintes as espécies originais da árvore (Figura 3.1, posições 6 e 7), visualizada na estrutura como novas espécies. Posições correspondentes são reservadas para os novos ancestrais (Figura 3.1, posições 9 e 10), na metade superior do vetor, ao lado dos ancestrais originais da árvore, chamado novos ancestrais. Essa pré-alocação de memória é possível devido a leitura anterior dos arquivos (Newick + PUTs), que nos permite saber a quantidade de espécies a árvore terá. Ou seja, sabendo que a árvore é binária e conhecendo o total de espécies, o total de ancestrais será 1 (um) a menos que o total de espécies, então o tamanho do vetor será 2 vezes a quantidade de espécies menos 1. Entretanto, em nossa solução, mantivemos esta posição a mais para identificação da árvore.[4]

Observamos que existe correspondência entre os dados dos vetores, ou seja, se o vetor espécie na posição 2 contém a espécie D, então os outros vetores possuem as informações sobre a espécie D na posição correspondente. Assim, o pai de D está na posição 14 (E), o comprimento do ramo que relaciona D ao seu antecessor é 2, a característica a ser avaliada durante o cálculo do I de Moran vale 6.0 e esse nó não possui filhos, ou seja, é um nó folha - o que pode ser observado pelo valor negativo “2“, nos vetores “filho esquerda“ e “filho direita“. [4]

4.2 Inserção de espécies

A parte de inserção eu não participei em nenhum momento, ficando sob responsabilidade do outro integrante do trabalho, Evandro. De forma resumida, temos que a inserção consiste em receber as estruturas de dados as árvores a inserir em posições aleatórias. Entretanto, tais pontos não são totalmente aleatórios, pois existe um ancestral de consenso (MDCC). Nesse caso, sabemos a sub árvore onde deve ser inserido aquela nova espécie em específico. As simulações construídas chegam a gerar milhares de réplicas de forma concorrente. Para mais detalhes, os quais não tenho total conhecimento, basta ler o artigo dos autores Evandro e Elias. [4]

4.3 Cálculo de matriz de distância

Assim como a etapa anterior, eu não participei em nenhum momento da parte do cálculo de matriz de distância. Sendo assim, não posso dar muitos detalhes sobre a implementação e abordagem utilizada pelo Evandro. Resumidamente, nessa etapa temos que é calculado a similaridade filogenética entre todos os pares de espécies. Sendo n espécies, então a matriz de distância possui tamanho n X n. Temos que a matriz é simétrica, sendo assim, não é necessário calcular toda a matriz, apenas a metade dos elemento - a diagonal, pois a mesma é trivial, com todos os valores iguais a zero. Para o cálculo é preciso somar todos os comprimentos de ramos que conectam duas espécies. Sendo assim, precisamos sempre encontrar o ancestral comum mais baixo entre os dois elementos a serem cálculados. Para mais detalhes, os quais não tenho total conhecimento, basta ler o artigo dos autores Evandro e Elias. [4]

4.4 Cálculo de I de Moran

Sabemos que geralmente, o I de Moran é aplicado sobre classes de distância relacionadas no tempo, ou seja, iremos comparar características de espécies que estão a uma mesma faixa de distância no tempo. Para isso, os biólogos definem as classes de distância, de forma mais aleatória possível. Para se ter uma melhor análise e resultados, são definidas poucas faixas de distâncias. Em nossa solução, usaremos 4 classes de distância. As faixas entre elas é definida de forma automática, pois usamos a diferença entre o maior e menor valor de distância de matriz.

Em nossa solução, criamos um kernel que aloca uma árvore para cada bloco, deixando todas as threads do bloco trabalhando no cálculo do I de Moran para aquela árvore específica. Para isso, para cada bloco são lançadas quantas threads forem o total de espécies. Assim, cada thread do bloco trabalha em uma área exclusiva de valores da

matriz de distância. O kernel é lançado uma única vez, gerando b blocos e t threads, podendo gerar milhões de threads, conforme o tamanho da árvore.

Antes de executar o algoritmo paralelo, é feito o cálculo para obter a média das características, o valor da variância, e também uma cópia do vetor de matriz, retirando os elementos que não farão parte do cálculo de I de Moran. O algoritmo paralelo tem os seguintes passos:

1. Cada thread verifica se as distâncias pertencem a aquela classe de distância (estamos usando 1-4).
2. Caso afirmativo, acumulam o produto da diferença entre a característica de cada espécie e a média.
3. Quando cada thread concluir os passos anteriores, temos que uma área da matriz foi verificada. O valor calculado até aqui é então acumulado, atomicamente, em uma variável compartilhada. Embora há um gasto computacional e de tempo para executar essa soma atomicamente, é de suma importância, pois várias threads podem concorrer pela variável compartilhada.
4. Após a execução dos passos anteriores, a thread então fica no aguardo das outras, devido a uma barreira de sincronização.
5. Quando todas concluírem, é então calculado o I de Moran daquela árvore, para aquela classe de distância.
6. Todos os passos anteriores são repetidos até a conclusão de todas as classes de distância.

Ao final do processo anterior, cada classe de distância terá seu Índice de Moran. Todavia cada árvore gera seu próprio Moran, ou seja, teremos milhares de árvore e consequentemente de índices calculados. Para resolver esse problema, é calculado a média do Índice de Moran, por classe. Veja o algoritmo 3.2 para um melhor entendimento do processo paralelo.

Algoritmo 3.1: Função Parse (Usando expressões regulares)

Entrada: String newick
Vector com os Puts(novas espécies)

Saída: Estrutura de vetores: espécie, pai, filho esquerda, filho direita, comprimento ramo e características.

```
1  Criar variáveis para percorrer e salvar partes da string
2  Criar variáveis Regex, para usar expressões regulares.
3  While ( tiver folhas (usando expressão regular)  ) faça
4  |    vetor_especie ←— nome folha
5  end
6  While ( tiver folhas (usando expressão regular)  ) faça
7  |    vetor_especie ←— nome nó interno
8  end
9  While (tiver nós) faça
10 |    Usar expressão regular para encontrar o próximo nó folha
11 |    Encontrar o index da falha no vetor espécie
12 |    Encontrar o index do filhos no vetor espécie
13 |    vetor_pai ←— index folha
14 |    vetor_filho_esquerda←— index filho esquerda
15 |    vetor_filho_direita ←— index filho direita
18 end
19 for (todos novos puts) faça
20 |    buscar onde será inserido
21 |    buscar novo pai
22 |    vetor_especie ←— novo put
23 |    vetor_pai ←——— index put
24 end
25 for (todas as espécies) faça
26 |    realizar busca em profundidade e somar o comprimento até a raiz
27 |    vetor_comprimento_ramo ←——— comprimento ramo
28 end
```

		espécies iniciais					novas espécies			novos ancestrais		ancestrais iniciais				
índice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
espécie	A	B	D	G	H	F	*X*	*Y*	-	*?*	*?*	I	J	C	E	K
pai	13	13	14	11	11	12	*15*	*12*	-	*?*	*?*	12	15	14	15	-1
filho esquerda	-2	-2	-2	-2	-2	-2	*-2*	*-2*	-	*?*	*?*	3	11	0	13	14
filho direita	-2	-2	-2	-2	-2	-2	*-2*	*-2*	-	*?*	*?*	4	5	1	2	12
comprimento do ramo	2	3	2	2	1	3	*?*	*?*	-	*?*	*?*	5	4	4	3	0
característica (traits)	3,3	2,4	6,0	3,4	5,6	2,1	*4,3*	*5,5*	-	*0,0*	*0,0*	0,0	0,0	0,0	0,0	0,0

Figura 4.1: *Estrutura para representação dos dados [4]*

Algoritmo 3.2: Kernel Cálculo de I de Moran

Entrada: Estrutura Dtree com todas as árvores
Matriz de distância
Número de classes de distância
Vetor de classes de distância
Vetor com características de cada espécie

Saída: Vetor com a média dos Índices de Moran por classe de distância

```
1  Criar duas variáveis compartilhadas para guardas somas
2  Criar um vetor compartilhado dinamicamente, e armazena o vetor com as classes de
distância.
3  Realizar uma barreira de sincronização (sync_threads).
4   For (todas as threads em cada bloco) faça em paralelo
5   If (index_thread < total_threads) então
6       Inicializar variáveis de soma
7       For ( classe ←— 1  até número total de classes de distância) faça
8       For (IndexMatrix ←— 1 até elemts thread +(elemts thread * indexThread) ) faça
9           If ( distância entre as espécies pertence àquela classe) faça
10              Buscar e associar o elemento do vetor de distância no vetor de espécies
11              Calcular e guardar o produto entre a diferença das espécies
12          end
13      end
14      Realizar atomicamente, o somatório dos valores calculados por todas as threads
15      Realizar uma barreira de sincronização (sync_threads).
16      If ( index thread = 0) faça
17          Calcular I de Moran para a classe em especifico
18          Reinicializar variáveis compartilhadas
19      end
20      Realizar uma barreira de sincronização (sync_threads).
21    end
22   end
23  End
```

CAPÍTULO 5

Resultados

Como descrito em capítulos anteriores, foi criado um algoritmo paralelo para a execução do cálculo de I de Moran. Todavia, para fins de comparação, usaremos também uma versão sequencial em nossos testes para fins de comparação e análise de desempenho. Para realizar os cálculos de I de Moran, usaremos 3 árvores:

- Dummy 160: Grupo de espécies artificiais, em uma filogenia completamente balanceada, com 160 espécies no total onde 32 delas são novas espécies(PUTs).
- Dummy 400: Grupo de espécies artificiais e também completamente balanceada, com 400 espécies no total onde 272 são novas espécies.
- Hummingbirds: Contém uma filogenia com 304 espécies, onde 158 são novas espécies a serem inseridas.

Os experimentos foram realizados diversas vezes, no mínimo 10, onde foi extraído então a média dos tempos de execução. Para realizar os cálculos, realizamos simulações para gerar 128,1024,4096 e 8192 árvores filogenéticas.

Para a realização dos experimentos, utilizamos um Intel Xeon E5 core i7, NVIDIA GeForce GTX Titan Black, com sistema operacional CentOS 6. A Titan Black possui 2880 núcleos CUDA com memory clock de 7.0 Gbps com 6 GB de memória global do tipo GDDR5. O programa foram implementados em C++ e CUDA C/C++.

5.1 Cálculo de I de Moran

Antes de realizar o cálculo de I de Moran, é necessário realizar alguns passos, como por exemplo transferir os dados da memória principal para a GPU. Nessa etapa, uma cópia da árvore e enviada várias vezes a GPU. Para a realização do cálculo, criamos um Kernel que possui 1 bloco por árvore a ser replicada, ou seja, cada bloca será responsável por calcular o I de Moran daquela árvore em específico. Além disso, cada bloco irá trabalhar com um número de threads igual ao número de folhas, pois é a quantidade certa de espécies a serem calculado o I de Moran. É possível perceber nas tabelas e nos gráficos, que foi possível conseguir ótimos resultados de desempenho e speedup, chegando a

Tabela 5.1: *Tempo de execução I de Moran para 128 árvores*

Espécies	Sequêncial	Paralelo
Dummy160	0,26	0,0013
Dummy400	1,84	0,004
Hummingbirds	1,14	0,005

Tabela 5.2: *Tempo de execução I de Moran para 1024 árvores*

Espécies	Sequêncial	Paralelo
Dummy160	2,55	0,0095
Dummy400	17,60	0,08
Hummingbirds	9,55	0,064

Tabela 5.3: *Tempo de execução I de Moran para 4096 árvores*

Espécies	Sequêncial	Paralelo
Dummy160	8,2	0,0363
Dummy400	100,35	0,26
Hummingbirds	28,88	0,42

Tabela 5.4: *Tempo de execução I de Moran para 8192 árvores*

Espécies	Sequêncial	Paralelo
Dummy160	21,12	0,072
Dummy400	303,50	0,789
Hummingbirds	75,40	1,02

ser mais de 300x mais rápido quando comparado a versão sequencial, como pode ser observado na figura 5.3. É possível perceber também que a medida que a quantidade de árvore a serem simuladas vão aumentando, o speedup vai melhorando.

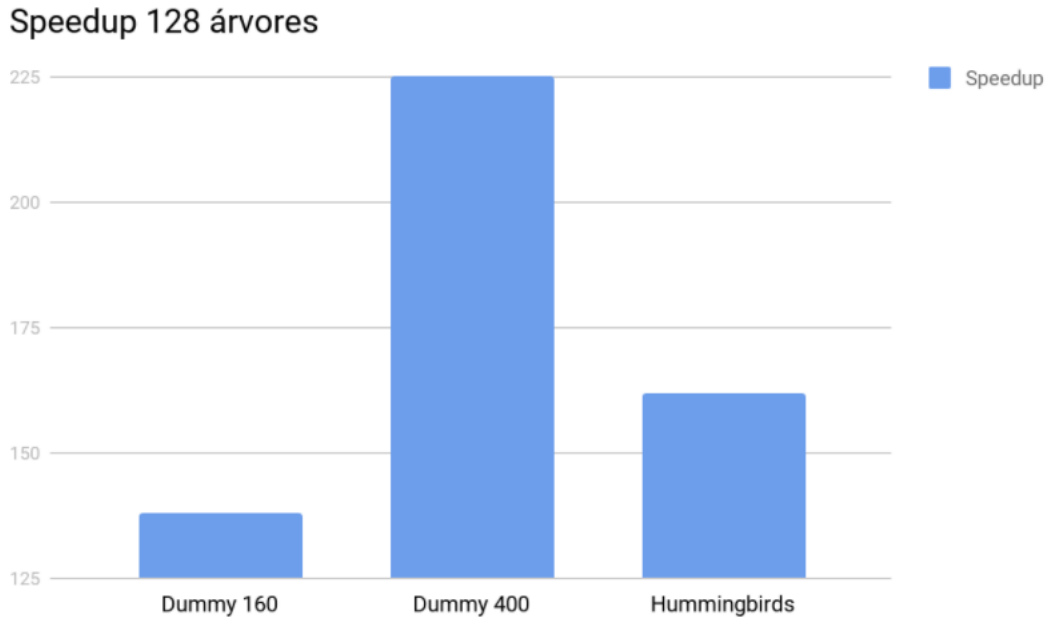


Figura 5.1: *Speedup para 128 árvores*

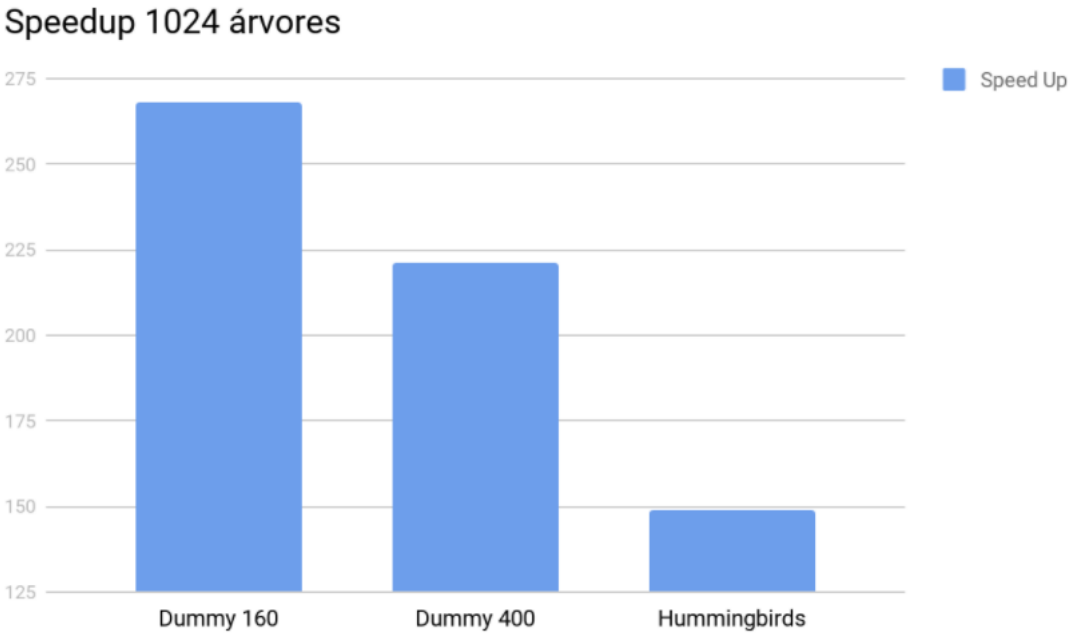


Figura 5.2: *Speedup para 1024 árvores*

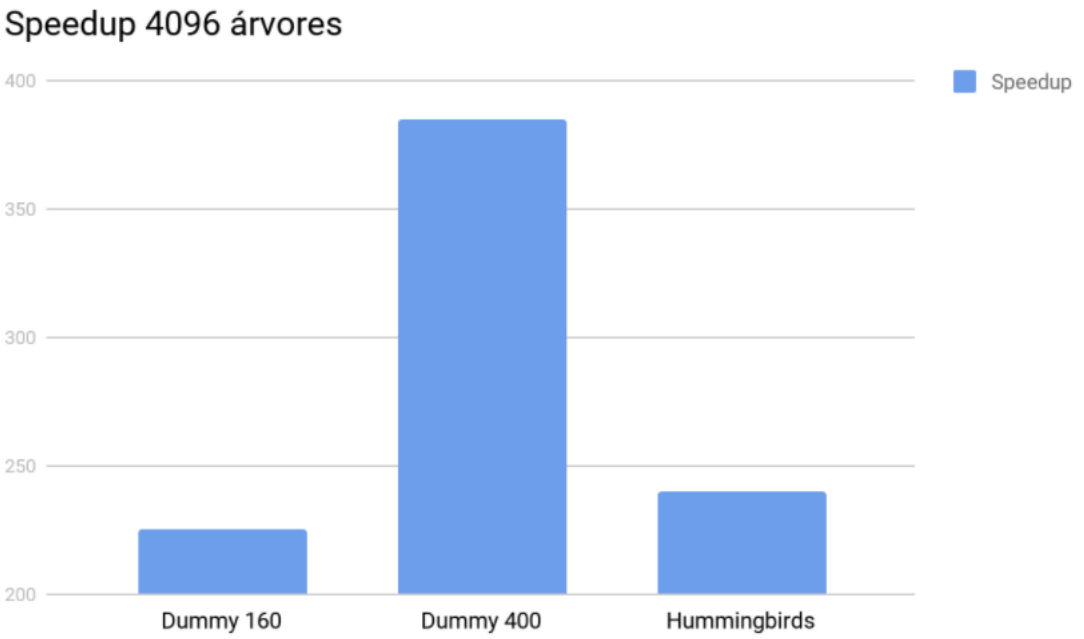


Figura 5.3: *Speedup para 4096 árvores*

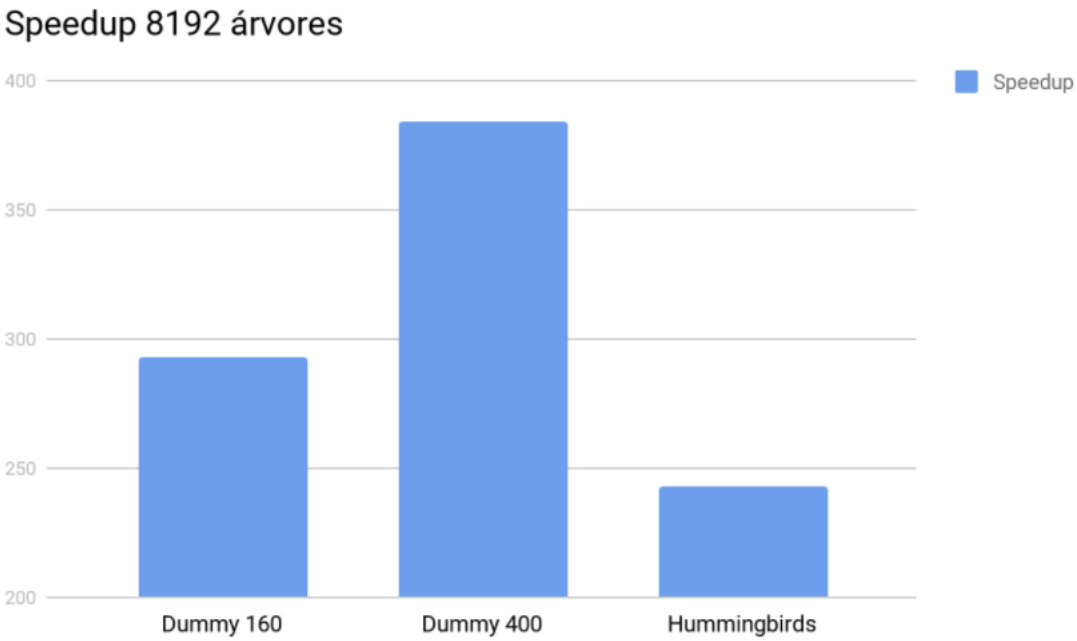


Figura 5.4: *Speedup para 8192 árvores*

CAPÍTULO 6

Conclusão

Embora eu tenha entrado com o trabalho já em andamento, eu participei de alguma forma de todas as etapas do projeto. Na fase inicial, buscamos identificar a melhor forma de processar e representar os dados de entrada(Newick, PUTs e Traits) . Sendo assim, construímos uma estrutura orientada a objeto, com a maior organização possível. Sendo assim, ao criar um Kernel, não seria necessário passar diversos parâmetros e dados, apenas o ponteiro para uma estrutura. Isso nos permitiu, junto a algoritmos altamente paralelos, utilizar bastante da potência da computação em GPU.

Para criar a estrutura orientada a objeto, criamos uma função Parse, a qual é responsável por receber todos os dados de entrada(Newick, PUTs e Traits) e os transformar em dados processados, armazenados em seus devidos vetores. O algoritmo foi realizado de forma sequencial, pois o mesmo requer muitas dependências e tomadas de decisão, o que não favorece um algoritmo paralelo, podendo inclusive ter resultados inferiores ao sequencial.

Após a execução do Parse, toda a estrutura e montada e organizada, favorecendo assim a execução das próximas etapas (Inserir Espécies e Matriz de Distância), as quais eu não tive uma participação direta, apenas o aluno Evandro.

Para a criação do I de Moran, utilizamos de forma considerável a concorrência na execução de tarefas, podendo chegar a ter milhões de threads em execução simultânea. Na criação do Kernel, é lançado um bloco por árvore e cada bloco possui tantas threads quanto forem o número de espécies, assim cada bloco é responsável por calcular o I de Moran daquela árvore em específico. Para o cálculo, utilizamos o acesso rápido a memória compartilhada para armazenar informações sobre classes de distância e variáveis de soma. Como o I de Moran é dividido por classe de distância, essa informação são consultadas constantemente por cada thread, garantindo então o acesso instantâneo, o que gera um impacto positivo no desempenho do algoritmo. Para calcular o índice de correção espacial, cada thread fica responsável por calcular uma faixa de valores na matriz de distância, o que pode gerar uma alta concorrência pelos dados. Sendo assim, embora possa desfavorecer um pouco o desempenho em geral do algoritmo, teve a necessidade do uso de operações atômicas e barreiras de sincronização.

Apresentamos algumas possibilidades e soluções de melhorias, as quais foram possíveis graças ao uso da computação em GPU. Foi possível então alcançar um elevado speedup se comparado a versão seriais, graças ao uso da computação paralela baseado em GPU, aliado a uma boa estrutura orientada a objeto e bem planejada para a arquitetura. Com a base desse projeto, é possível avançar muito mais, criando diversos outros métodos estatísticos que requerem a matriz de distância patrística como entrada, o I de Moran foi apenas um dos vários que é possível ser calculado.

Referências Bibliográficas

[1] BAUM, D. **Reading a Phylogenetic Tree: The Meaning of Monophyletic Groups**. preprint, 2008.

[2] BAUM, D. **Phylogenetics:Morphology Protocol**. preprint, 2011.

[3] BLAISE, BLAISE BARNEY. **Introduction to Parallel Computing**. preprint, 2016.

[4] ELIAS, E. B. **Processamento Paralelo Aplicado a Metodos Filogeneticos Comparativos**. *Instituto de InformÃ¡tica UFG*, 1:1–8, 2012.

[5] FELSENSTEIN, J. **Phylogenies and the Comparative Method**. *The American Naturalist*, 1:120–126, 1985.

[6] GOOLSBY, E. **Article Navigation Phylogenetic Comparative Methods for Evaluating the Evolutionary History of Function-Valued Traits**. *Systematic Biology*, 1:2–6, 2015.

[7] HENK, HENK POLEY. **A Look Back at Single-Threaded CPU Performance**. preprint, 2012.

[8] HILLS, D. **Ribosomal DNA: Molecular Evolution and Phylogenetic Inference**. *Biology Journal*, 3:2–7, 2012.

[9] THIAGO, T. F. **PAM - Phylogenetic Analysis in Macroecology** . *UFG*, 9:1–3, 2012.

[10] THIAGO, T. F. **Phylogenetic Uncertainty in Macroecological Analyses** . *American Naturalist*, 1:2–8, 2012.

[11] JOHN. **The Newick tree format**. preprint, 2011.

[12] JOHN. **Phylogenetic tree**. Enclyclopedia Britannica, England, 2012.

[13] JOHN CHENG, MAX GROSSMAN, T. M. **Professional CUDA C Programming**. John Wiley and Sons, Inc, Indianapois, IN, 2014.

[14] KHAN. **Phylogenetic trees**. preprint, 2014.

[15] NVIDEA. **NVIDIA CUDA Programming Model Overview**. NVIDEA Corporation, Indianapois, IN, 2013.

[16] PARADIS, E. **Moran - Autocorrelation Coefficient in Comparative Methods**. *Biology Journal*, 1:2–6, 2012.

[17] REWINI, H. **Advanced Computer Architecture and Parallel Processing**. Wiley, New Jersey, NY, 2005.

[18] SANDERS, J. **CUDA By Example - An Introduction to General-Porposo GPU Programming**. Jack Dongarra, Ann Arbor, Michigan, 2010.

[19] WAYNE. **Estimating a Binary Character's Effect on Speciation and Extinction**. *Systematic Biology*, 1:2–9, 2007.

[20] KAUR, M. K. **A Comparative Analysis of SIMD and MIMD Architectures**. *International Journal of Advanced Research in Computer Science and Software Engineering*, 1:1–8, 2013.