UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

ELIAS BATISTA FERREIRA

Processamento Paralelo Aplicado a Métodos Filogenéticos Comparativos

Goiânia
2012

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE DISSERTAÇÃO 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: Processamento Paralelo Aplicado a Métodos Filogenéticos Comparativos

Autor(a): Elias Batista Ferreira

Goiânia, 02 de Agosto de 2012.

Elias Batista Ferreira – Autor

Wellington Santos Martins – Orientador

ELIAS BATISTA FERREIRA

Processamento Paralelo Aplicado a Métodos Filogenéticos Comparativos

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Computação.

Área de concentração: Computação Paralela.

Orientador: Prof. Wellington Santos Martins

Goiânia
2012

ELIAS BATISTA FERREIRA

Processamento Paralelo Aplicado a Métodos Filogenéticos Comparativos

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Computação, aprovada em 02 de Agosto de 2012, pela Banca Examinadora constituída pelos professores:

Prof. Wellington Santos Martins
Instituto de Informática – UFG
Presidente da Banca

Prof. Edson Norberto Caceres
Universidade Federal de Mato Grosso do Sul – UFMS

Prof. Thiago Fernando Rangel
Universidade Federal de Goiás – UFG

Prof. Thierson Couto Rosa
Universidade Federal de Goiás – UFG

Elias Batista Ferreira

Graduou-se em Ciência da Computação na PUC-GO - Pontifícia Universidade Católica de Goiás. Pós-graduado MBA em Gestão de Software pelo Centro Universitário de Goiás (Uni-Anhanguera). É desenvolvedor de software, atuando como programador, analista de sistemas e administrador de banco de dados ao longo da carreira. Também atuou em curso de formação profissional na área de tecnologia. Atualmente é Professor de curso superior na Faculdade de Tecnologia Senac Goiás.

Aos meus pais, irmãos, esposa e meus filhos

Agradecimentos

Primeiramente agradeço aos meus filhos que, em momentos difíceis, transformaram seus olhares inocentes e carinhosos em força para continuar. Obrigado Lucas, Giovanna, Arthur e Henrique.

Agradeço ao meu orientador, Professor Wellington Santos Martins, pelos seus inúmeros ensinamentos, por sua paciência e contribuição para o desenvolvimento deste trabalho.

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) da qual obtive bolsa de estudo.

Aos servidores técnico-administrativos do Instituto de Informática da UFG, em especial ao Ricardo, Miriam e Edir pelo apoio e presteza no atendimento.

Agradeço ao meu irmão José Olímpio Ferreira, por seu apoio e incentivo para iniciar o mestrado.

Agradeço ao meu Pai, Manoel, que incentivou seus filhos ao estudo e a Sebastiana, minha Mãe, que sempre esteve por perto.

Finalmente, agradeço a minha esposa Terezinha, meus irmãos, irmã, familiares e amigos pelo auxílio nessa jornada.

Ciência da computação tem tanto a ver com o computador como a Astronomia com o telescópio, a Biologia com o microscópio, ou a Química com os tubos de ensaio. A Ciência não estuda ferramentas, mas o que fazemos e o que descobrimos com elas.

Edsger Dijkstra,
Wikipedia: a enciclopédia livre.

Resumo

Ferreira, Elias Batista. **Processamento Paralelo Aplicado a Métodos Filogenéticos Comparativos**. Goiânia, 2012. 94p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Algumas análises filogenéticas comparativas fazem uso de procedimentos de simulação que utilizam um grande número de árvores filogenéticas para estimar correlações evolutivas. Devido à grande demanda computacional requerida para processar centenas de milhares de árvores, a menos que este procedimento seja eficaz, as análises serão de pouca utilidade prática. Neste trabalho, apresentamos uma implementação altamente paralela e eficiente para gerar aleatoriamente e processar árvores filogenéticas.O uso do poder da computação das GPUs e de um grande número de threads resultou em ganhos de desempenho de até 225x quando comparado a uma implementação sequencial dos mesmos procedimentos. Novas estruturas de dados e algoritmos são também apresentadas de forma a processar eficientemente estruturas de dados irregulares baseadas em ponteiros, como árvores. Em particular, uma implementação paralela, baseada em GPU, do problema do menor ancestral comum (LCA) é apresentada. Além disso, a implementação faz uso intensivo de bitmaps de forma codificar eficientemente caminhos para nós da árvore, e otimizar as operações de memória por trabalhar com estruturas de dados que favorecem acessos aglutinados à memória. Nossos resultados abrem a possibilidade de lidar com grandes conjuntos de dados em análises evolucionárias e ecológicas.

Palavras–chave

Computação Paralela, GPGPU, GPU, CUDA, Filogenia, Incerteza Filogenética.

Abstract

Ferreira, Elias Batista. **Parallel Processing Applied to Phylogenetic Comparative Methods**. Goiânia, 2012. 94p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Some phylogenetic comparative analyses rely on simulation procedures that use a large number of phylogenetic trees to estimate evolutionary correlations. Because of the computational burden of processing hundreds of thousands of trees, unless this procedure is efficiently implemented, the analyses are of limited applicability. In this work, we present a highly parallel and efficient implementation for randomly generating and processing phylogenetic trees. By using the power of GPU computing and a massive number of threads we are able to achieve performance gains up to 225x when compared to a sequential implementation of the same procedures. New data structures and algorithms are also presented so as to efficiently process irregular pointer-based data structures such as trees. In particular, a GPU-based parallel implementation of the lowest common ancestor (LCA) problem is presented. Moreover, the implementation makes intensive use of bitmaps to efficiently encode paths to the tree nodes, and optimize memory transactions by working with data structures that favors coalesced memory accesses. Our results open up the possibility of dealing with large datasets in evolutionary and ecological analyses.

Keywords

Parallel computing, GPGPU, GPU, CUDA, Phylogeny, Phylogenetic uncertainty.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

CAPÍTULO 1

Introdução

As informações a respeito das relações genealógicas sobre os dados filogenéticos das espécies são de crucial importância nas pesquisas biológicas O advento das abordagens modernas na biologia molecular e genômica, aliado a um eficiente poder computacional de baixo custo e as ferramentas de informática, tem rapidamente expandido a disponibilidade de sequências genômicas em bases de dados públicas [44]. Simultaneamente, o aumento da filogenia, com suas ferramentas estatísticas para reconstruções filogenética a partir de sequências de DNA, tem causado uma mudança radical na perspectiva científica sobre a Árvore da Vida. Os biólogos evolucionários utilizam a informação filogenética para estudar como a seleção natural tem gradualmente moldado as características das espécies, bem como as variações entre as espécies. Além disso, ecologistas utilizam a filogenia para reconstruir a história, identificar causas e consequências nas relações entre as espécies, bem como compreender fatores históricos que afetaram suas abundâncias e distribuições. Na verdade, praticamente todos os campos da biologia usufruem da disponibilidade das informações filogenéticas e estão sendo afetados por essa silenciosa revolução que está em curso.

Árvores filogenéticas são afirmações hipotéticas sobre o relacionamento evolucionário entre as espécies e os métodos para construir essas hipóteses são normalmente baseados em algoritmos de pesquisa que tentam maximizar o ajuste entre a topologia da árvore e os dados, considerando um modelo evolucionário [26]. Embora os biólogos considerem cada vez mais a filogenia em seus estudos (e.g. [28]), as incertezas filogenéticas são habitualmente ignoradas. Uma fonte importante de incertezas filogenéticas surge quando biólogos fazem inferências evolucionárias sobre um grupo de espécies com filogenia incompleta [25, 46]. Neste caso, a hipótese básica é que o processo evolucionário capturado através das espécies amostradas pode ser extrapolado para espécies desaparecidas, apesar que esta suposição raramente é testada e pode, potencialmente, afetar conclusões sobre o processo natural [17]. Contudo, mesmo na ausência total de informações genômicas sobre as espécies, as características morfológicas, comportamentais e ecológicas podem fornecer pistas importantes sobre a relação evolutiva com outras espécies [27]. Essas informações de origem não genética não são precisas, visto que são insuficientes para

indicar as espécies próximas ou a distância filogenética entre elas. De qualquer maneira, essas fontes adicionais de informação podem ser utilizadas de forma confiável em estudos evolutivos contanto que a **incerteza estatística** possa ser estimada apropriadamente [25, 27].

Métodos de simulação contam com a repetição aleatória de amostras para modelar fenômenos com significativa incerteza nos dados de entrada [17, 27]. Todavia, a vantagem de estimar erros estatisticamente, associado com inferências baseadas na incerteza de dados traz consigo um alto custo de replicação computacional. A demanda computacional requerida para replicar analises estatísticas que se baseiam em informações filogenéticas impediram os biólogos de empregar, de forma plena, os métodos de simulação em estudos evolucionários. Particularmente, para considerar a incerteza filogenética em análises estatísticas, teríamos de construir um grande conjunto de árvores filogenéticas, na qual as ***espécies* com incertezas filogenéticas** (PUT, phylogenetic uncertain taxa) deveriam ser distribuídas aleatoriamente para árvores parcialmente conhecidas (e.g. construídas a partir de dados moleculares). Contudo, certo conhecimento filogenético sobre PUT está, normalmente, disponível; o ponto de inserção de cada PUT não é totalmente aleatório dentro da filogenia. Assim, para cada PUT é preciso estabelecer o **clado de consenso** (MDCC, *most derived consensus clade*) que, inequivocamente, contém cada PUT, utilizando todas as informações biológicas disponíveis e, quando necessário, a classificação conforme a filogenia disponível. Então, podemos entender o MDCC como uma **subárvore**, que é conhecida, e na qual podemos incluir as novas espécies - restringindo assim o âmbito de atribuição aleatória.

A técnica de colocar os *táxons* com incertezas filogenéticas (PUT, phylogenetically uncertain taxa) em uma árvore construída a partir de filogenias moleculares começa com a aleatorização na sequência em que esses PUT serão adicionados a árvore. Em seguida, para cada espécie que compõem o PUT, iremos identificar o seu respectivo MDCC. No âmbito do MDCC a espécie será atribuída em um ramo qualquer, escolhido ao acaso. O comprimento do ramo para o PUT, depois de ter sido colocado conforme descrito anteriormente, é simplesmente a distância a partir do ponto de fixação até uma ponta do seu próprio ramo. Depois que um PUT foi inserido na árvore, seu próprio ramo pode servir como um potencial ponto de inserção para espécies subsequentes, a serem atribuídas a partir de uma lista de PUT. Iremos construir um algoritmo que repetirá até que cada PUT tenha sido adicionado ao seu MDCC apropriado, produzindo assim uma completa e totalmente resolvida (sem politomias) filogenia. Deste modo, a variação na topologia das árvores é causada por esta incerteza nas relações entre PUTs e todas as outras espécies. Portanto, essas várias filogenias alternativas, construídas a partir desta técnica, podem ser utilizadas para quaisquer propósitos que se baseiam em informações filogenéticas, desde que o número de replicas sejam suficientes para mensurar o erro filogenético [27].

As informações contidas na topologia de uma árvore filogenética são de grande interesse para os biólogos, visto que inferências de processos evolutivos são derivadas a partir de relacionamentos filogenéticos entre espécies. Esses relacionamentos podem ser capturadas pela distância filogenética entre os pares de espécies; e quando múltiplas espécies são incluídas na analise, tais informações podem ser convenientemente armazenadas em uma matriz de distância [17], a qual chamamos de **matriz de distância patrística**. Uma distância patrística é a soma de todos os comprimentos de ramos entre duas espécies pertencentes a árvore. Alguns desses métodos estatísticos utilizados para estudar o processo evolucionário requerem a matriz de distância em estado natural (e.g. Estimativa da diversidade filogenética), enquanto outros métodos requerem a transformação da matriz de distância em uma matriz de covariância e variância (e.g. Análise de regressão linear). Em qualquer caso, se há uma incerteza filogenética no relacionamento entre as espécies, simulações podem ser realizadas para estimar a incerteza filogenética, por meio da geração aleatória de múltiplas árvores, colocando as espécies com incertezas filogenéticas na árvore. Assim, além da carga computacional de inserção dos PUT dentro da árvore, deve-se computar uma matriz de distância entre os pares de espécies para cada árvore filogenética gerada aleatoriamente. O próximo passo da análise comparativa filogenética consiste na replicação dos métodos estatísticos sobre cada uma das matrizes filogenéticas. A média filogenética calculada por meio da replicação das analises captura a melhor estimativa para um parâmetro real (e.g. diversidade filogenética das espécies), enquanto que a variância do cálculo filogenético reúne os erros devido ao tamanho da amostra (números de espécies) e incertezas filogenéticas na análise estatística.

Uma variedade de algoritmos computacionais vem sendo aplicados para análises filogenéticas. Entretanto, a maioria desses algoritmos (e.g. baseados em matriz de distância, máxima parcimônia, máxima verossimilhança e analise *bayesiana*) tentam inferir os relacionamentos filogenéticos entre as espécies estudadas. Por exemplo, eles constroem uma árvore filogenética a partir de um conjunto de genes, espécies ou outros *taxa*. Esses métodos vem sendo paralelizados para serem capazes de lidar com árvores filogenéticas maiores e com mais precisão e, também, para reduzir a carga computacional. Algumas propostas de paralelismo fazem uso de *grids*, *clusters* e até GPUs [52, 4, 51, 45, 57]. Por outro lado, análises comparativas filogenéticas não buscam fazer inferências filogenéticas, mas utilizam árvores filogenéticas para comparar espécies. Apesar de haver algoritmos desenvolvidos para realizar análises comparativas, somente à pouco tempo simulações envolvendo grande conjunto de dados (tipicamente 1000 ou mais) estão sendo desenvolvidas. Estas simulações são realizadas, geralmente, com ferramentas de comparações filogenéticas tradicionais, que não fazem uso das novas arquiteturas da computação paralela [35, 55, 20].

O presente trabalho visa contribuir no processo de análises comparativas filo-

genéticas em alta escala, utilizando como abordagem novas técnicas de computação paralela, e a criação de estruturas de dados que facilite o uso de dessas novas técnicas. Apresentaremos algoritmos paralelos, que fazem uso dessas novas estruturas e aproveitam as vantagens dessa nova forma de trabalhar com paralelismo, para gerar diversas árvores aleatoriamente, com a inserção de *espécies* com incertezas filogenéticas (PUT), calcular a matriz de distância patrística, realizar cálculos de coeficientes estatísticos, como o cálculo do *I de Moran*, que permitem verificar a correlação evolutiva de uma determinada característica da espécie.

Essa dissertação está organizada da seguinte forma, no capítulo 2 faremos uma introdução aos conceitos relacionados aos métodos comparativos filogenéticos e também uma breve revisão sobre filogenias e sobre algumas ferramentas de apoio computacional disponíveis no meio científico. No capítulo 3, abordaremos o processamento paralelo, discutiremos sobre conceitos das arquiteturas multinúcleos, computação em GPGPU (General Purpose Graphics Processing Unit), a solução CUDA da NVIDIA sobre essa arquitetura, OPENCL e como a computação vem sendo utilizada junto a biologia para auxiliar seus métodos científicos. No capítulo 4, apresentaremos nossa solução, descrevendo os algoritmos e estruturas de dados propostas. No capítulo 5, os resultados obtidos com nossa solução são apresentados, analisados e discutidos. Finalmente, no capítulo 6, apresentamos a conclusão e trabalhos futuros.

CAPÍTULO 2

Métodos Comparativos Filogenéticos

> O método comparativo vem sendo, desde os tempos de Darwin, uma das principais maneiras de estudar os padrões e processos da evolução biológica. Na verdade, a comparação de diferentes espécies com objetivo de inferir processos biológicos pode ser considerada uma das grandes subdivisões da própria Biologia[14].

A filogenia é o termo utilizado para definir hipóteses de relações evolutivas, ou seja, relações filogênicas, de um grupo de organismos. O princípio básico é que a origem da similaridade (medida de semelhança ou diferença) é a ancestralidade comum. O estudo da filogenia tem por objetivo reconstruir entidades biológicas por suas semelhanças, estimar o tempo de formação das espécies após compartilharem um ancestral comum e detalhar os eventos ocorridos durante esta formação[23].

Dado um conjunto de informações que caracterizem diferentes grupos de organismos - por exemplo, sequências de DNA ou de proteínas, ou estruturas de proteínas, ou formato dos dentes de diferentes espécies de animais - como podemos derivar informações sobre a relação entre os organismos nos quais estas características foram observadas? É raro que as relações entre as espécies e a ancestralidade sejam observáveis diretamente. Árvores evolucionárias determinadas com dados genéticos são frequentemente baseadas em inferências a partir de padrões de similaridade, o que é tudo que pode ser observado hoje entre as espécies vivas. A partir das relações entre as características desejamos inferir padrões de ancestralidade: a topologia das relações filogenéticas[34].

Devido ao crescimento de técnicas de biologia comparada, tornou-se possível a comparação de amplas quantidades de dados morfológicos, ecológicos e comportamentais com a informação oriunda do DNA ou sequências de aminoácidos. Os caracteres analisados são codificados em uma matriz e, partindo de diferentes premissas, são concebidos diagramas denominados filogenias ou árvores filogenéticas.

2.1 O Método Comparativo

Aos Métodos Comparativos estão associados um conjunto de práticas e técnicas para comparação de espécies. A aplicação desses métodos é conhecida em estudos de

biologia evolutiva, como as inferências filogenéticas [13]. A Biologia Evolutiva tem como principais objetivos descrever a variabilidade e a associação entre traços biológicos e interpretar como os padrões observados se originam em consequência da interação de fatores internos e externos.

Na Biologia Evolutiva, o principal objetivo do método comparativo sempre foi inferir, a partir da relação entre diversidade fenotípica das espécies e variação ambiental, o processo de adaptação biológica. Esta adaptação pode ser entendida como a mudança em uma característica fenotípica por ação de uma pressão seletiva atuando diretamente sobre ela (seleção natural) [14].

A forma de estabelecer a relação filogenética entre as espécies foi um dos principais problemas da Biologia Comparada. No entanto, após meados do século XX, a sistemática filogenética ou análise cladística, destacou-se como uma metodologia mais objetiva e eficiente para realizar análise dos padrões de diferenciação entre os organismos. Mesmo consciente da existência de problemas e discussões envolvendo os métodos para reconstrução filogenética e os tipos de dados mais adequados para realizar tal tarefa, já é possível utilizá-los de forma relativamente simples a fim de obter uma boa aproximação das relações entre as espécies a serem estudadas. Isso abriu espaço para os métodos comparativos filogenéticos.[14]

2.2 Método Comparativo Filogenéticos

Os métodos comparativos filogenéticos utilizam as relações filogenéticas (árvore filogenética) como base para analisar estatisticamente os padrões de variação e/ou covariação de caracteres morfológicos, ecológicos, fisiológicos e comportamentais entre as espécies[14]. Informações a respeito das relações genealógicas de espécies - dados filogenéticos - são de crucial importância em pesquisa biológica. Os biólogos evolucionários, aplicam informações filogenéticas para estudar como a seleção natural tem gradualmente moldado características das espécies, bem como a variação entre as espécies. Além disso, ecologistas usam filogenia para reconstruir e compreender os fatos da história. Na verdade, virtualmente todos os campos da Biologia beneficiam-se de disponibilidade de informações filogenéticas.

As espécies possuem ancestrais comuns no tempo, elas compartilham semelhanças por causa das características presentes no próprio ancestral. A consequência disso é que passa a existir um padrão filogenético nas características das espécies atuais e a existência do padrão filogenético implica que essas espécies não fornecem, individualmente, uma evidência independente para explicar as variações no carater, já que uma parte dessa variação provém precisamente do(s) ancestral(is).[14]. Os métodos compara-

tivos filogenéticos são utilizados para analisar essa variação, pois os valores de um carater qualquer não estão distribuídos ao acaso nas espécies estudadas.

As análises comparativas permitem integrar dados fenotípicos com filogenias moleculares a fim de inferir processos evolutivos. Esses métodos têm se tornado populares em função do desenvolvimento de novas ferramentas computacionais e da disponibilidade de filogenias moleculares.[14]

O aumento das filogenias e o uso de ferramentas estatísticas para reconstruir essas filogenias a partir de sequencias de DNA, tem causado uma mudança drástica na perspectiva científica sobre a Árvore da Vida. Muitos campos da Biologia podem tirar proveito do conhecimento construído à partir de dados moleculares e filogenéticos. Os biólogos evolucionários aplicam informações filogenéticas para estudar como a seleção natural tem gradualmente moldado características das espécies, bem como a variação entre as espécies.

2.3 Que dados utilizar

Os dados comparativos podem ser obtidos em estudos da Ecologia, Fisiologia, Morfologia, Genética ou qualquer outra área da Biologia dos organismos e são utilizados para observações em um dado nível da hierarquia biológica. Expressam, geralmente, uma tendência central - média, mediana ou frequência - ou simplesmente definindo qualitativamente uma característica dessas espécies[14].

A utilização do método comparativo requer uma árvore filogenética (também conhecida como árvore da vida), sendo que esta árvore pode ter sido concebida através de métodos moleculares. Além da árvore, as espécies com filogenia incerta e um vetor de características das espécies são imprescindíveis.

As filogenias expressam exatamente as ligações de ancestralidade entre as espécies em diversos níveis e são a base dos métodos compratívos. Na maior parte das vezes, as filogenias devem ser consideradas hipóteses sobre as relações entre os organismos, sujeitas a revisões em função do aumento de conhecimento sobre os próprios organismos ou sobre os métodos utilizados no estabelecimento das relações[14].

O ideal para os métodos comparativos é que a filogenia contenha tanto informações sobre o parentesco das espécies quanto sobre o comprimento dos “ramos” ligando as espécies.

2.3.1 Árvore Filogenética

As relações filogenéticas são representadas graficamente como **árvores filogenéticas**, também designada por Árvore da Vida. Elas representam as relações de an-

cestralidade e descendências, que consiste em linhas que se bifurcam de acordo com a existência no passado de um evento que transformou uma espécie em duas novas espécies. O ponto de bifurcação identifica o ancestral comum e as linhas que interligam as espécies, representam os comprimentos dos ramos, que são estimativas do tempo evolutivo. A figura 2.1 ilustra graficamente a estrutura de uma árvore filogenética com raiz. Nas árvores filogenéticas *enraizadas*, a raiz representa o ancestral comum a todas as espécies da árvore. Contudo, nem sempre é possível identificar esse ancestral, puramente por falta de informação, então constrói-se uma arvore que não possui orientação, dita *sem raiz*. As árvores sem raízes apenas posicionam as sequências umas em relação às outras sem, no entanto, mostrar a direção evolutiva, ou seja, mostrar a topologia da relação, mas não o padrão de descendência.

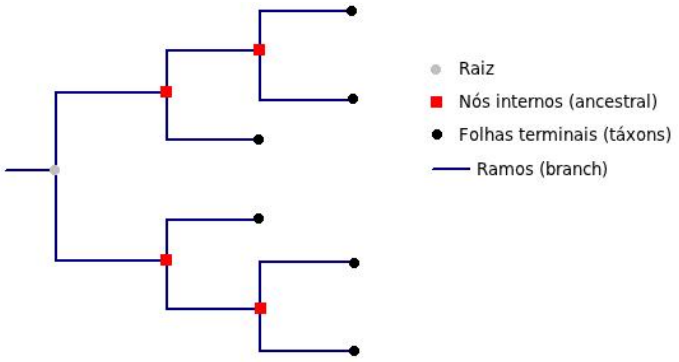


Figura 2.1: *Árvore filogenética*

Na ciência da computação, uma árvore é um tipo particular de grafo. Um grafo é uma estrutura que contém nós (pontos abstratos) conectados por arestas (representadas como linhas entre os pontos). Um **caminho** entre um nó e outro é um conjunto de arestas consecutivas, começando em um ponto e terminando em outro. Um **grafo conexo** é um grafo contendo ao menos um caminho entre dois pontos quaisquer. A partir deste conceito, podemos definir uma árvore: um grafo conexo, no qual há exatamente um caminho entre cada dois pontos. Um nó particular pode ser selecionado como a **raiz**, uma árvore com raiz, na qual cada nó tem dois descendentes, é chamada de **árvore binária**. Outro tipo especial de grafo é o *grafo orientado*, onde cada aresta é uma via de mão única. Árvores filogenéticas com raiz são, implicitamente, grafos orientados, sem ciclos, com a relação ancestral-descendente determinando o sentido de cada aresta [34].

Com já mencionado, as arestas de um grafo (nesse caso uma árvore filogenética) representam a "distância" entre as nós conectados por essa aresta. Um valor numérico é adicionado sobre a aresta para indicar essa distância. Em árvores filogenéticas, os comprimentos das arestas significam uma medida da dissimilaridade entre duas espécies, ou o tempo decorrido desde a sua separação. Podemos descobrir a distância evolutiva entre duas espécies quaisquer da árvore, realizando a soma das arestas que as interligam.

Para que as árvores sejam utilizadas em sistemas computacionais é necessário criar uma alternativa a representação gráfica, que expresse todas as informações descritas anteriormente, sem ambiguidades. Na bioinformática, um formato de representação de árvore na forma de texto muito utilizado é o padrão *Newick*. Esse é um padrão utilizado por muitos programas que tratam sobre filogenias, permitindo facilmente a portabilidade de dados entre softwares e plataformas distintas. Espécies, descendências, ascendências, comprimento de ramos e indicação de nó raiz são facilmente obtidos simplesmente fazendo uma varredura sobre o conjunto de dados. Esse formato é apresentado na seção 2.3.1.

Formato padrão Newick

O formato Newick foi idealizado em 1857 pelo matemático inglês Arthur Cayley. Newick é um formato simples utilizado para representar árvores em arquivos textos. Apesar de difícil leitura por seres humanos, é de fácil interpretação por programas computacionais. É uma notação padrão para representar árvores através de uma cadeia de caracteres, contendo parênteses aninhados, delimitada ao final por um ponto-e-vírgula.

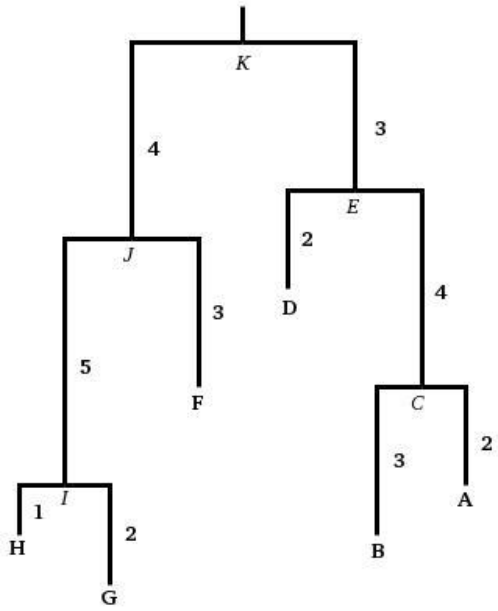
Os nós, internos ou externos, são separados por vírgulas, com parênteses indicando o agrupamento em sub-árvores. Cada par de parênteses corresponde a um nó interno ou a raiz da árvore. Nós internos podem ser identificados, neste caso o rótulo deve aparecer logo após o par de parênteses correspondente. O comprimento de cada ramo é informado por um sinal de ":" seguido de um número (inteiro ou real), esse valor indica o comprimento do ramo que o une ao seu ancestral.

A figura 2.2 mostra as formas de representação de uma árvore filogenética: gráfica e textual. Por exemplo, na figura 2.2(a) podemos identificar que as espécies *A* e *B* são derivadas da espécie *C*, simplesmente interpretando o jogo de parenteses, virgulas e rótulos que permitem visualizar essa dependência: (A:2,B:3)C:4. Também, podemos observar o comprimento do ramo (ou aresta) que interliga cada nó ao seu ancestral, através do número que sucede o simbolo de ":", que poderá conter tanto valores inteiros como fracionários. O formato *newick* é de fácil interpretação por um programa computacional, entretanto é complexo para leitura humana, principalmente à medida que a filogenia conta com mais espécies. Então, o formato gráfico, representando por uma árvore, figura 2.2(b), é utilizado para compreensão visual.

Existem diversos programas para visualização de árvores filogenética que fazem uso do formato Newick para ler ou exportar uma árvore ([41, 35]). Vários pacotes de softwares que utilizam de árvores filogenéticas para realizar suas computações, fazem leitura de árvores no formato Newick.

```
(((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: *Exemplo de uma árvore filogenética e suas formas de representação.*

2.3.2 Phylogenetically Uncertain Taxa (PUT)

O Método de Monte Carlo, o qual é largamente utilizados em vários campos da ciência, apoia-se na aleatorização de amostragens para modelar fenômenos com significativa incerteza nos dados de entrada. Contudo, as estimativas estatísticas e os erros associados com inferência baseada na incerteza de dados, vem ao custo de cálculos altamente replicados. A demanda computacional para replicar as análises estatísticas que dependem de informações filogenéticas tem impedido os biólogos de empregar completamente o método de Monte Carlo em estudos evolutivos. Em particular, para quantificar a incerteza filogenética em análises estatísticas, seria necessário construir um grande conjunto de árvores, nas quais as *espécies com incertezas filogenéticas* (PUT) seriam distribuídos aleatoriamente em árvores parcialmente conhecidas (exemplo: construídas a partir de dados moleculares). Entretanto, esta aleatorização não é realizada em qualquer ponto da árvore, algum conhecimento filogenético sobre os PUT estão geralmente disponíveis. Assim, o ponto de inserção para cada PUT não será totalmente aleatório através da filogenia. Para cada PUT, é preciso determinar o MDCC (most derived consensus clade, ou ancestral que é consenso), que inequivocamente contem cada PUT, utilizando toda informação biológica e, quando necessário, classificação baseada na melhor taxonomia disponível. Assim, o MDCC define a subárvore conhecida para incluir a espécie e restringe o alcance

da aleatorização para alocação do PUT.

Os PUT's devem vir associados ao seus ancestral. Esta ancestralidade ao qual a espécie esta vinculada (MDCC), diz que a espécie pesquisada pertence a algum ramo da subárvore representada por este ancestral. Então, ao distribuir as espécies aleatoriamente pela árvore, deve-se considerar apenas a **subárvore** ou clado de referência para este ancestral.

2.3.3 Vetor de características

As **características** (ou *traits*) abordadas, referem-se a medidas funcionais das espécies envolvidas na análise. Essas informações são utilizadas para calcular o índice de correlação espacial entre cada espécie da filogenia. Na estrutura do *vetor de características* temos duas informações em cada entrada: a espécie e a característica. Todas as espécies possuem uma entrada nesta estrutura, inclusive as espécies com incertezas filogenéticas (PUTs). A seguir vemos uma estrutura comum para este vetor, onde a característica a ser analisada é o comprimento do corpo.

Espécie	Comprimento do corpo
A	0.2966208518
B	0.440878001
C	0.2111513391
D	0.6851080745
E	0.4753307621

2.4 Coeficiente I de Moran

Os padrões filogenéticos podem ser analisados através de índices de correlação. Esses índices expressão as variações na semelhança entre os pares de espécies em função do tempo de divergência entre elas. O Índice de Moran, dado na equação 2-1, é utilizado para auxiliar na compreensão dessa semelhança. O índice de Moran (I de Moran) é uma medida estatística para autocorrelação espacial. Esse índice realiza uma medida geral que indica o grau de associação espacial presente no conjunto de dados. Os valores do índice de Moran variam entre "-1.0" e "1.0", sendo estes valores seus máximos para para correlação negativa e positiva. Valores de I de Moran próximos a 1.0 indicam que as espécies ligadas em um dado nível de conexão filogenética tendem a ser, para o caráter estudado, similares entre si, enquanto que valores próximos a -1.0 indicam que essas espécies tendem a dissimilaridade [15]. Portanto, uma variável é auto-correlacionada se uma medida feita num ponto carrega informação sobre outra medida daquela variável registrada em outro ponto, localizado a determinada distância. Dessa forma, os valores

não são independentes do ponto de vista estatístico, violando um pressuposto importante dos testes tradicionais, i.e., o da independência dos dados entre os pontos de amostragem. [12]

$$I = (\frac{n}{S}) \left[\frac{\sum(\sum(W_{ij}(y_i - \bar{y}) \cdot (y_j - \bar{y}))}{\sum(y_i - \bar{y})^2} \right] \quad (2\text{-}1)$$

Onde:

- n: é o número de espécies
- y: representa a variável analisada (representa uma característica ou *traits*)
- $\bar{y}$: é a média de y
- W_{ij}: é o valor 1 ou zero, representa a conectividade. Indica se o par de distâncias da matriz simétrica esta ou não dentro da classe.
- S: é a soma dos elementos de conectividades em cada classe de distância

Normalmente, o *I de Moran* é aplicado sobre classes de distância no tempo, ou seja, para comparar uma determinada característica entre espécies que estão a uma mesma faixa de distância no tempo. Assim, os biólogos definem as classes de distância, de forma arbitrária, que possuem relevância para uma determinada análise. Além disso, são definidas poucas faixas de distâncias, aproximadamente $\log_2 n$ (seja "n" a quantidade de espécies).

A utilização da classe de distância é necessária para restringir o grupo de espécies a serem comparadas. Uma classe informa que somente espécies que se encontram na faixa de distância dessa classe, são utilizadas nos cálculos do *I de Moran*.

2.5 Como são usados

Analises evolucionárias e ecológicas geralmente dependem de uma única árvore filogenética de consenso a fim de estudar os processos evolutivos. Entretanto, a maioria das árvores filogenéticas, as quais incluir aquelas baseadas em dados moleculares, são incompletas no que diz respeito a amostragem, então o que geralmente ocorre é que apenas algumas poucas espécies de genealogia mais importantes são estudadas.

Uma solução é adicionar as espécies faltantes nas árvores previamente definidas. A inclusão das espécies (PUTs) ocorre de forma *parcialmente aleatória*, pois algum conhecimento já existe sobre essas espécies. Normalmente já sabemos o ponto na árvore à partir do qual podemos inserir uma espécie, ou seja, o ancestral comum (MDCC) de um determinado *clado*. A espécie pode ser incluída em qualquer ponto dentro desse *clado*, mas somente dentro dele, nunca fora. Um *clado* é um grupo de organismos originados de um único ancestral comum.

A inserção dos PUTs (phylogenetically uncertain taxa) em uma árvore consiste na aleatorização de um ponto onde a espécie será adicionada (ponto esse dentro do clado) e no cálculo do comprimento do ramo. Uma vez que o PUT tenha sido inserido, seu próprio ramo deve servir como um potencial ponto de inserção para espécies subsequentes. O comprimento do ramo (ou *branch*) com base no comprimento do *taxa* irmão. Em seguida, á arvore, agora *completa*, da origem a uma matriz de distância patrística, que serve como base para diversos cálculos aplicados sobre filogenias.

O vetor de características (ou *traits*) é utilizado para calculo do coeficiente estatístico "I de Moran" que mede o quanto espécies dentro de uma classe de distância filogenética possuem semelhança para uma dada característica.

2.6 Apoio computacional

A ciência da computação tem importante participação nas descobertas científicas atuais. O poder computacional crescente, oferecidos por novas tecnologias de hardware e melhores soluções de softwares, permite aos cientistas realizarem experimentos com extraordinária velocidade e acurácia. Sem os avanços nas áreas de hardware e software, não conseguiríamos alcançar os resultados apresentados pela ciência atualmente [24].

A bioinformática é o uso de técnicas computacionais e conceitos da matemática, da estatística e da informática para resolver problemas biológicos [56]. Um dos problemas da biologia que é alvo de grandes estudos refere-se ao alinhamento de sequências. A comparação de sequência é um problema de interesse da ciência da computação. Algoritmos para fazer alinhamento de sequenciais buscam realizar comparações que permitam descobrir aquela que corresponda ao maior grau de similaridade. Além da comparação de sequências a computação também é aplicada a problemas da biologia evolutiva, ecologia, microbiologia, etc. O uso da computação de alto desempenho (HPC) e de baixo custo, conseguido através das GPUs, surge como forma de disponibilizar aos pesquisadores poder computacional equivalente aqueles fornecidos por *clusters* e/ou *grids*.

A obtenção e a análise de informações exigem diversos tipos de algoritmos, desde os mais simples até os mais sofisticados. Uma variedade de algoritmos computacionais tem sido aplicados para análises filogenéticas. No entanto, a maioria destes algoritmos (baseados na matriz de distância, máxima parcimônia, máximo verossimilhança e análises Bayesiana) tentam inferir as relações genéticas entre as espécies em estudo, isto é, eles constroem a árvore filogenética a partir de um conjunto de genes, espécies ou outros grupos taxonômicos (taxa). Estes métodos tem sido paralelizados para ser capaz de lidar com árvores filogenéticas maiores e mais exatas e, também, para reduzir a carga computacional. Algumas das propostas fazem uso de *grids* computacionais, *clus-*

ters, e mesmo GPU's ([4, 45, 51, 52, 57]. Por outro lado, análises filogenéticas comparativas não tentam inferir filogenias, ao invés disso, usa árvores filogenéticas para comparar espécies. Embora, algoritmos computacionais tenham sido desenvolvidos para analises comparativas, somente recentemente, simulações envolvendo grande conjunto de dados (tipicamente 1.000 ou mais) tem sido utilizados. Estas simulações são usualmente realizadas utilizando ferramentas de comparação filogenéticas tradicionais, que não exploram novas arquiteturas computacionais paralelas ([20, 35, 55]). Algumas dessas ferramentas são discutidas nas próximas seções.

2.6.1 Phylocom

Phylocom é um software escrito na linguagem de programação ANSI C, publicado sobre licença open source, licença BSD, e pode ser compilado sobre qualquer sistema operacional [55]. É uma software desenvolvido para calcular uma série de métricas referentes a uma comunidade filogenética e analisar características de similaridade entre essas espécies. Diferente de outros softwares existentes, o Phylocom fornece um pacote de funcionalidades, que permite realizar diversas análises, como a medida de um sinal filogenético e correlacioná-lo à evolução dessa característica na comunidade das espécies.

Uma característica chave do Phylocom é a incorporação da análise evolutiva de um carácter dentro de uma comunidade filogenética e, também, os algoritmos para comparar sinais filogenéticos e correlacionar um determinado carácter. Além disso tem habilidade para manipular politomias e trabalhar com árvores com milhares de nos terminais [55]. Como diversas analises estatísticas são realizadas a partir de um matriz de distância filogenética, é requisito que sejam calculadas as distâncias entre todos os pares de espécies da filogenia.

O formato para entrada de dados no Phylocom é através de arquivos textos delimitados, bem como a saída de dados que é escrita em arquivos textos que podem facilmente serem importadas para planilhas, processadores de textos, etc. As filogenias são repassadas em um formato largamente conhecido, o formato Newick. Outro formato largamente utilizado é o formato *Nexus*.

2.6.2 Patristic

Patristic é um programa, em java, que calcula a **matriz de distâncias patrística** das árvores carregadas na forma de arquivos de entradas, inclusive filogenias no formato *newick*. O **patristic** fornece diversas visualizações gráficas dos resultados. Essa matriz pode ser utilizada pelo programa *Phylip*, que faz parte do pacote de ferramentas do

phylocom, e por outros softwares. Ele também importa informações do pacote de software *phylocom* [20].

2.6.3 Compare

O **Compare** [35] é um pacote de programas utilizado para realizar inferências sobre evolução à partir de dados comparativos. Foi desenvolvido em Java, e realiza analise de dados sobre um contexto evolutivo. Também aplica vários métodos filogenéticos comparativos (por exemplo: autocorrelação espacial e outros).

É um software que pode ser utilizado diretamente pela web, na forma de uma *applet java*, ou podemos baixar uma versão para ser utilizada localmente. O código fonte esta disponível e pode ser baixado para alteração, se necessário.

2.6.4 FigTree

FigTree é um programa de visualização de árvore filogenética através de uma interface gráfica. Podemos carregar arquivos no formato Newick e Nexus e realizar exportações no formato PDF. No decorrer dos estudos utilizamos este software para nos auxiliar na avaliação e visualização dos resultados gerados em cada fase do trabalho, por possuir uma interface limpa e de fácil utilização e, também, por permitir configurar as informações a serem exibidas na árvore. Não é um programa utilizado para realizar comparações filogenéticas.

Este software esta disponível para os ambientes MacOS, Microsoft Windows e Linux. Pode ser obtido, gratuitamente, em http://tree.bio.ed.ac.uk/software/figtree/.

CAPÍTULO 3

Processamento Paralelo

A computação paralela tem causado um tremendo impacto em diversas áreas, que vão desde simulações computacionais para aplicações científicas e aplicativos para mineração de dados e processamento de transações. O custo-benefício de do paralelismo, juntamente com os requisitos de desempenho exigidos pelas aplicações atuais são argumentos convincentes em favor da computação paralela [22].

3.1 Visão Geral

Tradicionalmente, o computador tem sido visto como uma máquina sequencial. A maioria das linguagens de programação requer que o programador especifique um algoritmo como um sequência de instruções. Os processadores executam programas por meio de execução sequêncial de instruções de máquina. Cada instrução é executada como um sequência de operações (busca de instrução, busca de operandos, execução da operação, armazenamento de resultados)[50].

Há algumas décadas a tecnologia de microprocessadores tem registrado grandes avanços. A taxa de clock alcançou valores superiores a 2.0 Ghz. Ao mesmo tempo, processadores são capazes de executar múltiplas instruções em um mesmo ciclo de clock [22]. Soluções com vários processadores em uma única placa já estão consolidadas e processadores com vários núcleos são a nova tendência tecnológica na atualidade. A combinação destas duas tecnologias resulta em um arquitetura multiprocessada híbrida, onde vários processadores com múltiplos núcleos podem ser instalados em conjunto.

Os processadores atuais têm seguido dois caminhos distintos, que tendem a se encontrar num futuro próximo [38]. De um lado temos processadores com alguns poucos núcleos com grande poder de processamento, os chamados *multicore*. Esses processadores são uma evolução natural dos processadores de um único núcleo, que visam otimizar a latência de memória e permitem um uso moderado de linhas de execução (threads). São processadores que dedicam grande área da pastilha (chip) para a inclusão de memória *cache*, lógica de previsão de desvios, pipeline com vários estágios,

entre outras técnicas que favorecem a exploração de paralelismo ao nível de instruções (ILP). Do outro lado, temos os processadores com dezenas ou centenas de núcleos mais simples, os chamados *manycore*. Esses são otimizados para uma maior vazão (throughput) na execução de instruções, permitindo um uso intenso (milhares) de *threads*. Nesses processadores, grande parte da pastilha é dedicada a unidades de execução (lógica/aritmética) em detrimento de memória cache e técnicas de ILP.

Os processadores *multicore* são hoje uma realidade não somente nos computadores servidores e de mesa, mas também em diferentes dispositivos móveis como computadores portáteis (laptops), pranchetas eletrônicas (tablets), celulares (smartphones) e até em algumas máquinas fotográficas profissionais. Por outro lado, os processadores manycore são comuns em plataformas voltadas para jogos eletrônicos, dispositivos embarcados e até em dispositivos de chaveamento em redes (switches). Um caso especial de sucesso é o da evolução das Unidades de Processamento Gráfico (GPUs), responsáveis pela geração de imagens realísticas e em tempo real nos jogos. Essas passaram de processadores manycore de propósito específico (gráfico) para unidades de processamento de uso geral. O recente anúncio da arquitetura Kepler da NVIDIA confirma a importância desse tipo de solução. Outras iniciativas de processadores *manycore* também vêm ganhando atenção, como as soluções das empresas AMD, Tilera, Azul e Adapteva [1].

O uso da concorrência com objetivo de acelerar a resolução de problemas computacionais tem sido aplicado há várias décadas. Diversas plataformas paralelas foram e estão sendo construídas para permitir a solução de problemas maiores e, também, cada vez mais rápidos. Segundo Cacéres, Mongelli e Song [7], um Sistema de Computação Paralela e Distribuída consiste de uma coleção de elementos de computação (processadores), geralmente do mesmo tipo, interconectados de acordo com uma determinada topologia para permitir a coordenação de suas atividades e troca de dados. Esses processadores trabalham simultaneamente, de forma coordenada, com o objetivo de resolver um problema específico. Esses sistemas surgiram em função da necessidade de solucionar problemas onde a computação sequencial não consegue obter uma solução dentro de tempos razoáveis.

3.1.1 Paralelismo e classe de problemas

A chave da computação paralela é explorar a concorrência. A concorrência existe dentro de um problema computacional quando o problema pode ser decomposto em subproblemas que podem, seguramente, executar ao mesmo tempo. No entanto, para utilizar a concorrência o código deve ser estruturado para explorá-la, permitindo que os subproblemas sejam realmente executados simultaneamente [36].

A maioria dos grandes problemas computacionais podem ser explorados con-

correntemente. O trabalho do programador é explorar esta concorrência, construindo um algoritmo paralelo e utilizando um ambiente de desenvolvimento para programação paralela. Quando um programa paralelo é executado sobre um sistema com múltiplos processadores a quantidade de tempo que iremos aguardar pode ser reduzida. Além disso, é possível executar problemas maiores do que seria possível com um único processador[36].

Um exemplo simples seria a soma de uma grande conjunto de valores. Se utilizarmos um algoritmo sequencial cada número é adicionado ao outro de forma sequencial, até que o cálculo este concluído. Contudo, se vários processadores estiverem disponíveis, podemos construir um algoritmo que particione o conjunto de valores distribuindo-os para cada processador, de tal forma que esse processador realize a soma de seu subconjunto de valores. Então, os resultados parciais são combinados para obtermos o resultado final. Além disso, se cada processador tiver sua própria memória, podemos particionar os dados para cada processador de tal forma que problemas maiores sejam executados [36].

Podemos observar que os objetivos da computação paralela é permitir a resolução de problemas em um menor tempo e/ou resolver problemas maiores do que seria possível utilizando apenas uma único sistema computacional. Os passos para construir um programa paralelo são: identificar a concorrência, construir um algoritmo que explore esta concorrência e implementar a solução, fazendo uso de ambiente de desenvolvimento que propicie a construção de programas paralelos. Em seu livro [19], Ian Foster diz que a maior parte dos problemas tem várias soluções paralelas, sua metodologia é constituída de quatro passos (veja também a figura 3.1:

- **Particionamento**: aqui a ideia é expor oportunidades de paralelismo, buscando encontrar uma grande quantidade de pequenas tarefas. Existe a abordagem centrada nos dados (decomposição do domínio) e outra centrada no processamento (decomposição funcional). Na decomposição do domínio, analisamos os dados buscando dividi-los em partes que possam ser executadas separadamente. Já na decomposição funcional, daremos atenção ao processamento, buscando dividi-lo em partes e depois definindo como associar dados a estas partes.
- **Comunicação**: consiste em determinar estruturas e algoritmos necessários para comunicação. Apesar das tarefas poderem ser executadas separadamente, seus resultados precisam ser associados a outras tarefas. Então, devemos definir quais canais que ligam as tarefas que precisam de dados (consumidoras), com as tarefas que possuem dados (produtoras). A comunicação é parte da **sobrecarga paralela** dos algoritmos paralelos, porque tarefas sequências não sofrem deste problema.
- **Aglomeração**: após identificar o máximo de paralelismo possível, sem a preocupação com a plataforma onde o programa será utilizado, devemos voltar a atenção para o agrupamento das tarefas paralelas em processadores físicos, adaptando o al-

goritmo genérico para máquina paralela. É necessário construir uma solução que permita a redução da sobrecarga paralela, mantenha a escalabilidade e reduza o custo de engenharia de software.

- **Mapeamento**: o mapeamento tem por objetivo maximizar a taxa de utilização de processadores e minimizar a comunicação interprocessador. As tarefas que podem ser executadas concorrentemente são atribuídas a diferentes processadores, para aumentar a concorrência e, por outro lado, tarefas que se comunicam frequentemente são alocadas no mesmo processador, permitindo o aumento da localidade.

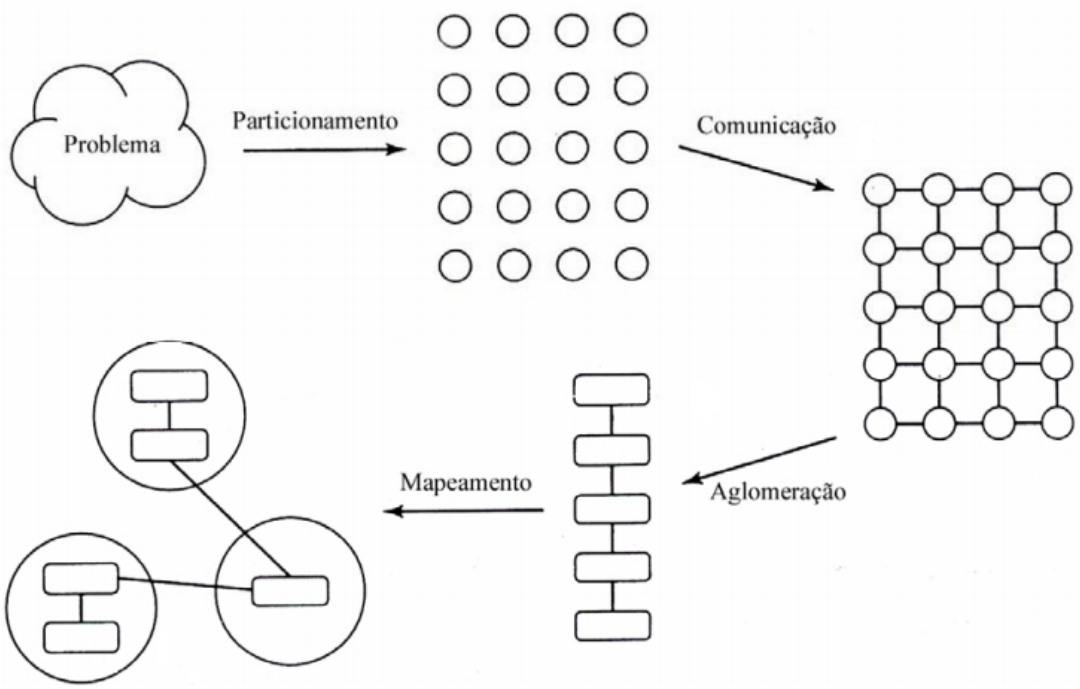


Figura 3.1: *Métodologia de Foster para desenvolvimento de algoritmo paralelo*

Aplicar paralelismo para solucionar problemas computacionais nos coloca diante de um grande desafio. Muitas vezes, as tarefas a serem realizadas simultaneamente apresentam dependências, que devem ser identificadas para que sejam devidamente tratadas. A ordem na qual as tarefas são executadas pode alterar as respostas. Por exemplo, no somatório paralelo descrito anteriormente, uma soma parcial não pode ser combinada com outras até que sua própria computação tenha sido concluída. O algoritmo deve impôr uma ordem sobre as tarefas, em nosso exemplo, as somas parciais devem ser concluídas antes de serem combinadas. Fazendo uma análise mais refinada, nos resultados da soma podem haver pequenas variações, causadas pela aritmética de ponto flutuante; deve-se ter cuidado para que essa variação não afete a qualidade da resposta final [36]. A construção de programas paralelos seguros exige um considerável esforço do programador.

Mesmo quando um programa paralelo esta correto, ele ainda pode falhar se entregar, ainda que explorando a concorrência, uma solução que não apresente as melhorias de

desempenho esperadas. Isso pode ocorrer devido a sobrecarga adicionada a um programa paralelo ao gerenciar a concorrência (ordem de execução, divisão dos subconjunto de dados, etc). Deve-se ter o cuidado para que a gerência da concorrência não sobrecarregue o tempo para execução do programa. Além disso a eficácia de um algoritmo paralelo pode ser muito boa para uma arquitetura e ter um desempenho sofrível em outra.

A aplicação do paralelismo se divide em duas vertentes: dados e funcional. No **paralelismo de dados** cada tarefa executa uma mesma série de cálculos sobre diferentes dados, dizemos que a decomposição é realizada no domínio dos dados. devemos pensar na sua divisão em conjuntos que podem ser processados de forma em paralelo, em seguida, identificamos quais operações devem ser realizadas sobre cada conjunto de dados.

Já no **paralelismo funcional**, cada tarefa executa diferentes cálculos sobre um conjunto de dados (podem ser os mesmos ou diferentes) para resolução de um problema. Neste caso o enfoque é no processamento a ser realizado, dividindo-os em tarefas independentes e aplicando, cada uma das tarefas, sobre o conjunto de dados identificados para a tarefa.

Taxonomia de Flynn

Muitos tipos de computadores paralelos já foram propostos e construídos ao longo dos anos, sendo natural buscar uma maneira de categorizá-los em uma taxonomia [53]. O projeto de uma arquitetura de computadores paralelos deve considerar dois conceitos, o fluxo de dados e fluxo de controle. O fluxo de controle em computadores paralelos são baseados nos mesmos princípios do computador de von Neumann, exceto que várias instruções podem ser executadas no mesmo instante [22]. De forma semelhante, no fluxo de dados podemos encontrar um único caminho de *chegada* dos dados ou vários caminhos, ou seja, consiste em um conjunto de operandos que podem ser manipulados por instruções de controle. A taxonomia introduzida por Flynn [18] é a forma mais comum de classificação sendo utilizada e, ainda assim, é uma aproximação muito grosseira[53]. Flynn propôs as seguintes categorias de sistema computação (Figura 3.2):

- **Única instrução, único dado** (SISD – single instruction, single data): é o classico computador de von Neumann, um único processador executa uma única sequência de instruções, usando dados armazenados em uma única memória. Um sistema uniprocessador pertence a esta categoria[50].
- **Única instrução, múltiplos dados** (SIMD - single instruction, multiple data): têm uma única unidade de controle que emite uma instrução por vez, mas elas tem multiplas ALU's (Unidade Lógica e Aritmética) para executá-la em vários conjunto de dados simultaneamente[53]. Os processadores vetoriais e matriciais pertencem a essa categoria.

- **Múltiplas instruções, único dado** (MISD - multiple instruction, single data): uma sequência de dados é transmitida para um conjutno de proessadores, cada um dos quais executa uma sequência de instruções diferentes. Essa estrutura tem pouco interesse prático [50].
- **Múltiplas instrução, múltiplos dado** (MIMD - multiple instruction, multiple data): um conjunto de processadores executa simultaneamente sequências diferentes de instruções, sobre conjunto de dados distintos. Segundo Stallings (2002), os clusters, SMP (multiprocessadores simétricos) e sistemas NUMA (acesso não-uniforme à memória) pertencem a essa categoria.
  Uma variação do modelo MIMD é o SPMD (single program multiple data, único programa e múltiplos dados). Um estilo de programação popular para sistemas de computador maciçamente paralelos [22, 32]. Em um sistema SPMD, as unidades de processamento paralelo executam o mesmo programa sobre várias partes dos dados.

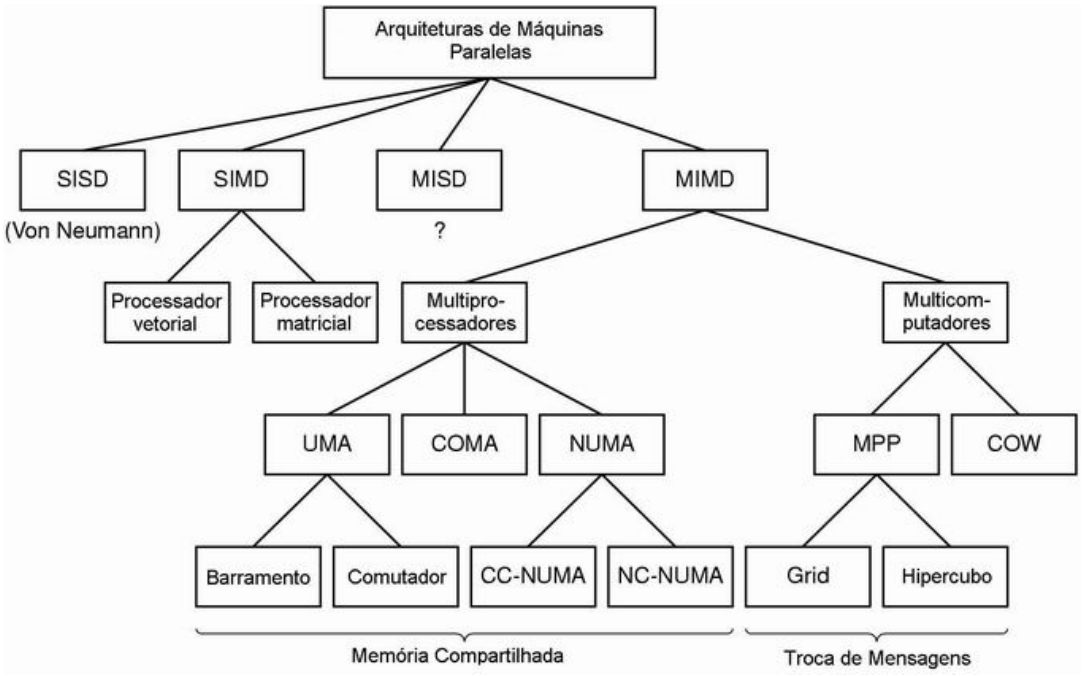


Figura 3.2: *Taxonomia de computadores paralelos [53]*

Arquiteturas paralelas

Computadores paralelos, geralmente, seguem os modelos SIMD ou MIMD. Quando há apenas uma unidade de controle e todos os processadores executam a mesma instrução de forma sincronizada, então a maquina paralela é classificada como SIMD. Nas maquinas MIMD cada processador tem sua própria unidade de controle e pode executar diferentes instruções sobre diferentes dados.

O Modelo SIMD de computação paralela consiste em duas partes: um computador front-end estilo maquina de von Neumann e um arranjo de processadores (Figura 3.3). O conjunto de processadores é formada por vários elementos de processamento idênticos capazes de executar, simultaneamente, a mesma operação sobre diferentes dados. Cada processador dentro do arranjo possui uma pequena quantidade de memória local onde os dados ficam armazenados enquanto são processados. Na arquitetura SIMD o paralelismo é explorado aplicando, simultaneamente, uma operação em um grande conjunto de dados. Este paradigma é útil quando aplicado, exaustivamente, sobre dados para realização de cálculos numéricos [30].

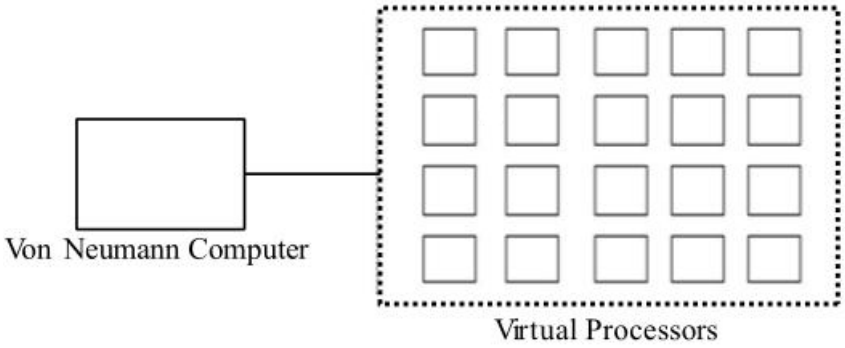


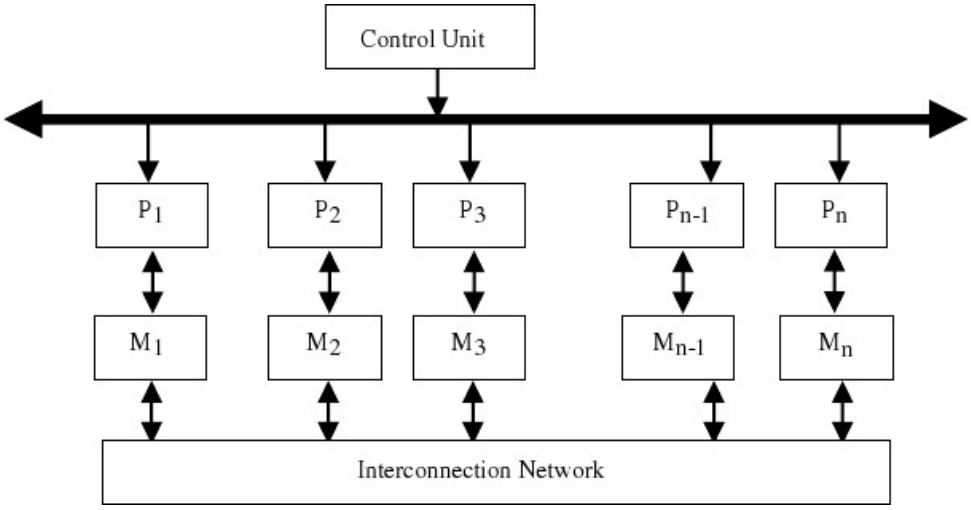
Figura 3.3: *Modelo de arquitetura SIMD [30]*

Há duas configurações que podem ser utilizadas com máquinas de arquitetura SIMD (Figura 3.4). No primeiro esquema (figura 3.4(a)) cada processador tem sua própria memória local, os computadores comunicam entre si através de uma rede de comunicação. Se a rede de comunicação não fornecer conexão direta entre um dado par de de processadores, esse par pode trocar informações fazendo uso de um intermediário. Já no segundo modelo(3.4(b)), processadores e módulos de memória comunica-se um com os outros através da rede de comunicação. Dois processadores podem trocar dados através dos módulos de memória [30].

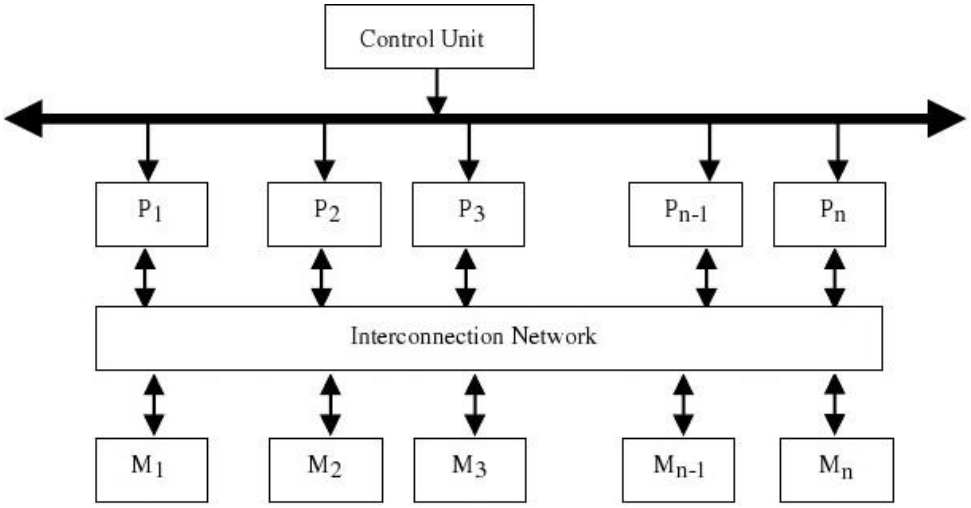
A categoria MIMD pode ser subdividida em multiprocessadores (máquinas de memória compartilhada) e multicomputadores (máquinas de troca de mensagens) - (Figura 3.2 e 3.5).

Nas máquinas com memória compartilhada os processadores acessam dados e programas armazenados nesta memória e se comunicam com os outros processadores por meio dessa memória. A forma mais comum de sistemas desse tipo é conhecida como multiprocessador simétrico (SMP). Nesta arquitetura, cada processador tem igual oportunidade de leitura/escrita na memória, assim como o tempo de acesso a qualquer região da memória é aproximadamente o mesmo para cada processador. Esta plataforma também é conhecida como UMA (uniform memory access, acesso uniforme a memória).

Outro desenvolvimento na categoria das máquinas com memória compartilhada são as máquinas NUMA (non-uniform memory access, acesso não-uniforme à memória).



(a)

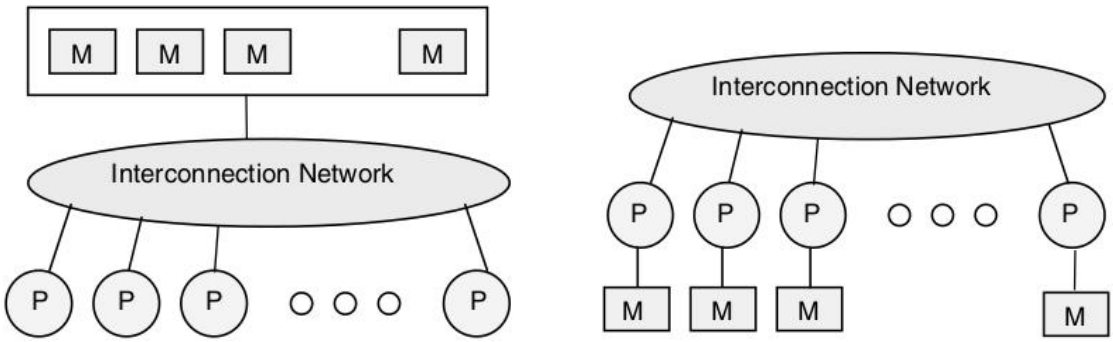


(b)

Figura 3.4: *(a) e (b) representam dois esquemas de acesso a memória da arquitetura SIMD[30].*

Neste modelo o tempo de acesso a diferentes regiões de memória podem ser diferentes, pois, muitas vezes há um módulo de memória próximo a cada processador e acessar aquele módulo de memória, que está fisicamente ligado a ele (local) é mais rápido do que acessar os módulos mais distantes, ligados a outros processadores (remoto).

A distinção entre as plataformas NUMA e UMA é importante sobre o ponto de vista de projeto de algoritmos. Máquinas NUMA requerem que os algoritmos explorem o princípio da localidade para alcançar melhor desempenho, pois, se acessar a memória local é mais barato que acessar a memória global, então os algoritmos devem utilizar estruturas de dados que permitam explorar a localidade das informações. Além disso, a leitura e escrita dos dados são mais difíceis de programar do que as operações somente de leitura, uma vez que operações de leitura/escrita requerem uso da exclusão mútua para acessos simultâneos.



(a) *Arquitetura MIMD - Memória compartilhada* (b) *Arquitetura MIMD - Passagem de mensagens*

Figura 3.5: *Arquiteturas de Memória compartilhada e Passagem de mensagem[30].*

As arquiteturas com passagem de mensagens (Figura 3.5(b)) consiste em uma plataforma de troca de mensagens entre **maquinas lógicas** que possuem "p" nós de processamento, onde cada nó tem seu próprio espaço de endereçamento exclusivo. Cada nó de processamento pode ser um processador único ou um multiprocessador com espaço de endereçamento de memória compartilhada, uma tendência nos computadores paralelos. Nessa plataforma, as interações entre processos que estão executando em diferentes nós devem ser realizadas através de mensagens, daí o nome **troca de mensagem**. As trocas de mensagens são utilizadas para transferir dados, tarefas e ações de sincronização entre os processos[22].

Os multicomputadores (ou máquinas de troca de mensagens) podem ser divididos em duas categorias. A primeira categoria contém os MPPs (Massively Parallel Processors – processadores de paralelismo maciço) que são supercomputadores caros que consistem em muitas CPUs fortemente acopladas por uma rede de interconexão proprietária de alta velocidade [53]. A outra categoria consiste em PCs ou estações de trabalho comuns, possivelmente montados em estantes e conectados por tecnologia de interconexão comercial. Em termos de lógica não há muita diferença de redes de PCs montadas pelos usuários por uma fração do preço de um MPP. Essas máquinas caseiras são conhecidas por vários nomes, entre eles NOW (Network of Workstations – rede de estações de trabalho), COW (Cluster of Workstations – grupo de estações de trabalho), ou, às vezes apenas **clusters** [53]. O sucesso dessa solução levou algumas empresas a comercializarem plataformas mais robustas, voltadas ao segmento empresarial.

Modelo PRAM

A utilização de modelos computacionais nos permite comparar algoritmos, indicando o nível de complexidade de uma determinada solução. À partir do uso de modelos podemos relacionar soluções paralelas com soluções sequências, enfatizando

características estruturais do problema e do processamento concorrente. Detalhes de sincronização e comunicação são, geralmente, ignorados nesta análise.

Os modelos permitem realizar uma descrição abstrata de uma máquina paralela. Possibilitando assim, capturar somente características essenciais para um determinado algoritmo. Simulando algoritmos sobre os modelos, é possível argumentar sobre a eficiência dos algoritmos, analisar os limites de complexidade e o máximo de paralelismo possível.

A definição de um modelo geral para computadores paralelos que considere todos as possíveis variáveis, inclusive os custos de comunicação, parece difícil de ser alcançada [43]. O Modelo PRAM (Parallel Random Access Machine) é um modelo geral e impreciso para computação paralela, que ignora os custos envolvidos com comunicação, sincronização e outras limitações físicas existentes em situações reais (como limites de memória e restrições de processadores). No entanto, torna possível a classificação de algoritmos e obtenção de suas complexidades.

O modelo PRAM, é uma extensão ao modelo RAM do computador sequencial. É um modelo ideal constituído de uma memória central compartilhada, largura de banda infinita e "p" processadores. O custo de acesso a memória global é $O(1)$, assim como a execução de operações basicas (adição, multiplicação, conjunção, disjunção, etc), que também são realizadas a um custo de $O(1)$. O modelo não limita o tamanho da memória compartilhada nem o número de processadores.

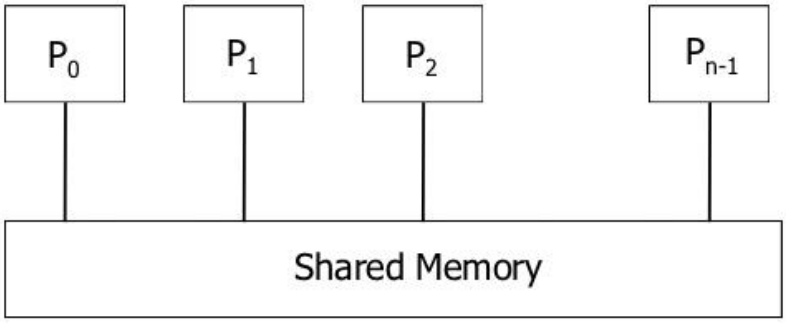


Figura 3.6: *Modelo de computação PRAM (Parallel RAM)*

Uma importante questão relacionada ao modelo PRAM diz respeito ao que acontece quando mais de um processador tenta, ao mesmo tempo, acessar a mesma célula da memória global (leitura e/ou escrita). As diferentes formas de resolver os conflitos de leitura ou escrita definem as subdivisões do modelo PRAM [7]:

- Exclusive-read, exclusive-write (EREW) PRAM: não permite qualquer tipo de acesso simultâneo a uma posição da memória compartilhada.
- Concurrent-read, exclusive-write (CREW) PRAM: permite que acessos simultâneos a uma posição da memória sejam feitos exclusivamente para leitura. No entanto, vários acessos de escrita a uma mesma localização de memória são serializados.

- Exclusive-read, concurrent-write (ERCW) PRAM: múltiplos acessos de escrita são permitidos para um mesmo local de memória, mas multiplos acessos de leitura são serializados.
- Concurrent-read, concurrent-write (CRCW) PRAM: nessa classe múltiplos acessos de leitura e escrita a um local de memória comum são permitidos. As escritas concorrentes ao mesmo local de memória requer uma política para decidir qual processador irá efetuar a escrita:
    1. Commom CRCW: comum, neste modelo é permitido a escrita concorrente se, e somente se, o valor for o mesmo para todos os processadores.
    2. Arbitrary CRCW: arbitrário, neste modelo um processador arbitrário terá sucesso na escrita.
    3. Priority CRCW: prioridade, cada processador deve possuir uma prioridade, aquele que possuir a prioridade mais alta tera sucesso na escrita.
    4. Combining CRCW: neste modelo o valor armazenado é uma combinação dos valores a serem escritos. Um operador de combinação deve ser escolhida (por exemplo adição, máximo, mínimo, etc). Se o operador **soma** for escolhido, então a soma dos valores de cada processador é realizada para, em seguida, ser armazenada na célula de memória.

Limites da computação paralela

A Complexidade de um algoritmo consiste na quantidade de *trabalho* necessária para sua execução, buscando analisar o as operações fundamentais realizadas para resolver o problema, considerando desde as instruções do algoritmo até o volume de dados. Podemos encontrar dois tipos de complexidade de algoritmos: espacial e temporal. Na complexidade espacial avalia-se a quantidade de recursos utilizados para resolver um problema. Enquanto isso, a complexidade temporal o fator *tempo* é verificado, ou seja, qual o número de instruções necessárias para resolver determinado problema. Todos os problemas tem sempre uma entrada de dados, medida por tamanho n, o tamanho dessa entrada tem impacto direto no tempo de resposta de um algoritmo, então ao avaliar a complexidade de um algoritmo, o tamanho da entrada deve ser considerado.

Quando descobrimos a complexidade de um algoritmo, torna possível avalia-lo para que saibamos se essa é a melhor solução, ou seja, é aquela que tem a melhor *performance*. A performance é extremamente importante na informática, então há uma busca constante de melhorar dos algoritmos. E, mesmo com o aumento da velocidade dos computadores, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, devido ao aumento constante do "tamanho" do problemas a serem resolvidos.

Nessa análise, o número de passos necessário para resolver um problema (tempo) é o principal recurso avaliado - o algoritmo. Os algoritmos podem ser classificados

quanto a ordem de complexidade, podendo ser: constante, linear, logarítmica, quadrática, cúbica, exponencial, entre outras. Problemas de **complexidade linear**, são resolvidos em tempo proporcional ao seu *tamanho n*, ou seja, uma operação é realizada em cada elemento de entrada. Já problemas de complexidade constante, são algoritmos onde a complexidade independente do tamanho da entrada. Os problemas de complexidade logarítmica, normalmente dividem o problema e problemas menores, reduzindo assim sua complexidade [58].

Na teoria da computação podemos classificar os problemas como **Indecidíveis** (problemas impossíveis de serem solucionados), **intratáveis** (são problemas que só poderiam ser solucionados com recursos ilimitados, porém impossível uma solução com recursos limitados) e **tratáveis** (classe de problemas solúveis, mesmo com recursos limitados, ou seja, existe uma solução polinomial para o mesmo). Essas classes de problemas são bastante conhecidas na computação. A classe $\mathcal{P}$, que refere-se aos problemas tratáveis, ou seja, podem ser **resolvidos** em tempo polinomial e a classe $\mathcal{NP}$, problemas intratáveis, consiste nos problemas que podem ser **verificados** em tempo polinomial. Outra classe bastante conhecida é a classe de problemas $\mathcal{NP}$–completo, que são um subconjunto dos problemas de $\mathcal{NP}$. Um problema é $\mathcal{NP}$–completo se todos os problemas de busca são redutíveis a ele. A Figura 3.7 mostra a relação entre as classes de problemas citadas, consulte [49] para mais informações sobre o assunto.

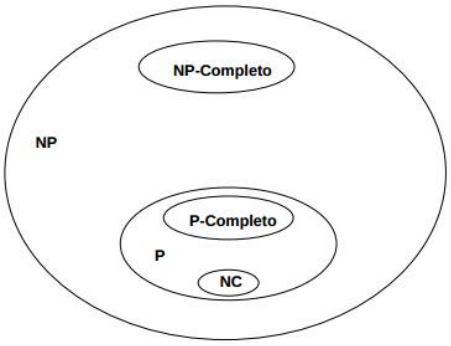


Figura 3.7: *Classificação da complexidade dos problemas computacionais*

Sabendo que o paralelismo consegue grandes melhorias na execução dos algoritmos, então os cientistas da computação se motivam a buscar soluções melhores, inclusive que possam obter melhoras para problemas indecidíveis. Mesmo assim, problema que são indecidíveis no mundo "sequencial", também o são quando convertidos para paralelo, isso porque algoritmo paralelo pode ser simulado sequencialmente por um único processador. No entanto, podemos pensar que, utilizando o modelo PRAM, os algoritmos paralelos podem tornar problemas da classe $\mathcal{NP}$ em problemas tratáveis. No entanto, devemos fazer algumas considerações a respeito: é necessário um número de processadores exponenciais para resolver estes problemas, além disso, não sabemos responder se um problema é

realmente intratável ($\mathcal{P} = \mathcal{NP}$), e também há de se considerar sobre o gargalo da comunicação para uma máquina com arranjo de processadores super-polinomial.

Será que o paralelismo pode resolver problemas, em quantidade razoável de tempo, que não sejam possíveis de resolver, de forma sequencial (em tempo aceitável)? Essa é uma questão que está em aberto. Dizemos que um problema é paralelizável se o mesmo puder ser resolvido muito rapidamente com quantidade moderada de processadores. Devemos entender como "muito rapidamente", resolvidos em tempo poli-logarítmico, isto é, $T(n) = O((\log n)^k)$ para alguma constante k e uma entrada de tamanho n. A classe $\mathcal{NC}$ (Nick's Class, homenagem a Nickolas Pippener) representam estes problemas poli-logarítmicos, ou seja, *NC* consiste em todos os problemas que podem ser resolvidos por algoritmos paralelos de tempo poli-logarítmicos sobre um modelo PRAM utilizando um número polinomial de processadores.

Existem alguns problemas, chamados $\mathcal{P}-$completo (Figura 3.7), sobre os quais não é conhecida uma solução eficiente (poli-logarítmica) para resolve-los. Isso nos leva a uma questão, ainda em aberto, será $\mathcal{NC} = \mathcal{P}$? A maioria das pessoas acreditam que não, ou seja, $\mathcal{NC} \not\subset \mathcal{P}$. De forma oposta, sabemos que $\mathcal{NC} \subset \mathcal{P}$, uma vez que podemos pegar qualquer algoritmo em $\mathcal{NC}$ e converte-lo em termo polinomial em um algoritmo sequencial.

A situação $\mathcal{NC} \subset \mathcal{P}$ é similar ao problema entre as classes $\mathcal{P}$ e $\mathcal{NP}$. Se pudermos provar que há um algoritmo de tempo polinomial para um problema $\mathcal{NP}-$completo, então temos algoritmos de tempo polinomial para todos os problemas $\mathcal{NP}-$completo. Da mesma maneira, se encontrarmos um algoritmo de tempo poli-logarítmico para ao menos um problema $\mathcal{P}-$completo, então teremos algoritmos de tempo poli-logarítmico para todos os problemas $\mathcal{NP}-$completo.

3.1.2 Métricas de desempenho

Ao desenvolver algoritmos que vão executar sobre uma arquitetura paralela cria-se a expectativa quanto a sua eficiência. Quando um algoritmo paralelo executa sobre uma arquitetura que possua 4 (quatro) processadores, esperamos que alcance um desempenho 4 (quatro) vezes mais rápido que uma arquitetura mono-processada. Todavia, na maioria das vezes esse ganho esperado não acontece, podendo inclusive ocorrer o contrário, a agregação de mais processadores implicar na redução no tempo de execução, devido o grau de dependência existentes entre as partes paralelizadas.

Diversos fatores podem interferir no desempenho de um algoritmo paralelo. Por exemplo, o código é totalmente paralelizável? a comunicação necessária entre os processadores é realizada em tempo razoável? fatores como o tempo de decomposição e sincronização entre os processadores devem ser considerados. Alguns conceitos e

métricas são utilizados para a análise da eficiência e desempenho computacional, são elas: granulosidade, aceleração (speedup), eficiência, custo, *overhead* e escalabilidade [42].

Granulosidade

Granulosidade pode ser definida como o tamanho das unidades de trabalho submetidas aos processadores ou ainda como a quantidade de trabalho realizado entre iterações do processador [33]. É uma medida da razão entre a quantidade de computação realizada em uma tarefa paralela e a quantidade de comunicação necessária. A medida de granulosidade auxilia na medida de desempenho de uma determinada arquitetura.

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 feito num pequeno número de grandes tarefas, o que pode implicar em baixa comunicação entre os processadores. Ao desenvolver um algoritmo paralelo devemos avaliar se o tempo de execução das atividades compensam os custos para criação, comunicação e sincronização ao mesmo tempo que temos que manter a ocupação dos processadores no nível mais alto possível. Estes, são fatores conflitantes, considerando que ao atender uma das partes, então estaremos potencialmente afetando negativamente a outra. Ou seja, teremos, basicamente, duas variáveis principais: numero de processadores *versus* tempo de comunicação. A modificação em uma dessas variáveis tem implicação direta na outra.

Em linhas gerais, devemos buscar um meio termo, onde a solução paralela apresente ganhos consideráveis se comparado com a solução sequencial. Assim, quando temos um algoritmo com granularidade grossa, podemos tentar fazer uso de mais processadores na expectativa de aumentar a eficiência do algoritmo. Por outro lado, se a granulosidade é fina, o número ótimo de processadores pode necessitar de redução, para amenizar à latência (comum quando há alto índice de comunicação), e os ganhos sejam consideráveis [42].

Speedup e eficiência

Speedup e *eficiência* são dois parâmetros utilizados para medir as vantagens na utilização da computação paralela [33].

Speedup pode ser definido como aceleração ou ganho de desempenho. É uma medida que tem por objetivo determinar a relação existente entre entre uma arquitetura sequencial e a arquitetura paralela. Considere P um problema computacional, sendo n o seu tamanho, então denotamos o a complexidade sequencial de P por $T_s(n)$, ou seja, existe um algoritmo sequencial que resolve P em tempo $T_s(n)$. Da mesma forma, podemos ter

um algoritmo paralelo que resolve P em tempo $T_p(n)$ em um computador paralelo com p processadores [42]. Assim, podemos calcular o ganho de desempenho (speedup) obtido, com a equação 3-1:

$$S_p(n) = \frac{T_s(n)}{T_p(n)} \tag{3-1}$$

O valor $S_p(n)$ mede a aceleração obtida pelo algoritmo paralelo com p processadores disponíveis para utilização. Quando o *speedup* for igual a "p" indica que o caso ótimo foi atingido, ou seja, o aumento da capacidade de processamento é diretamente proporcional ao número de processadores. Podemos obter três resultados distintos para o *speedup*:

- $S_p(n) < 1$: este é o pior caso, onde a solução paralela é pior que a solução sequencial.
- $1 < S_p(n) \leq p$: normal, são os casos mais comuns. O objetivo é projetar algoritmos para esta classificação.
- $S_p(n) > p$: conhecido como ***speedup* super linear**, não ocorre com frequência. Normalmente, o *speedup* máximo que pode ser alcançado com p processadores é função linear de p. Porém, em algumas situações podem ocorrer acima desse valor, o que chamamos de *Speedup super linear*. Um dos motivos no qual esse ganho pode acontecer refere-se a quantidade de recursos disponíveis. Por exemplo, um problema cujo os dados ocupam além da memória disponível em uma máquina sequencial, pode ter ganhos acima do esperado quando divido em varias partes e, assim, puder usufruir de recursos extras para computação.

O objetivo é projetar algoritmos paralelos que alcancem $S_p(n) \cong p$, mas podemos encontrar diversos fatores que introduzem ineficiência nos algoritmos paralelos. Dentre esse fatores destacamos partes não paralelizáveis do algoritmo, atrasos introduzidos pela comunicação, granulosidade inadequada, a sobrecarga ocorrida na sincronização das atividades dos vários processadores e no controle do sistema[7].

Outra medida para análise da execução do algoritmo é a **eficiência**, definida na equação 3-2:

$$E_p(n) = \frac{S_p(n)}{p} \tag{3-2}$$

A eficiência relaciona o *speedup* e número de processadores, ou seja, identifica a utilização do processador. O ideal é que consigamos atingir **eficiência** igual a 1, indicando que o ganho é proporcional ao números de processadores utilizados, entretanto, a situação mais comum é encontrar a eficiência inferior a 1 e o *speedup* inferior a "p", que chamamos do caso normal.

Custo

O custo de uma computação paralela é definida pelo produto entre o tempo gasto para execução da solução paralela e o número de processadores (equação 3-3). Dizemos que um algoritmo paralelo é ótimo se o seu **custo** estiver na mesma classe de complexidade do algoritmo sequencial ótimo.

$$C = T_p(n) \cdot p \tag{3-3}$$

Escalabilidade

O termo escalabilidade indica a habilidade de um sistema em manipular uma porção crescente de trabalho de forma uniforme, este conceito pode ser aplicado ao hardware ou ao software (algoritmo). Quando falamos de hardware, ser 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. Na escalabilidade do **algoritmo** entende-se que o mesmo é escalável quando pude acomodar o aumento do tamanho do problema a ser tratado com um aumento baixo e limitado dos passos de computação [42]. Entendemos, passos da computação as etapas da metodologia de Ian Foster, que pode ser vista na figura 3.1.

Overhead

Overhead é considerado qualquer processamento ou armazenamento em excesso, seja de tempo de computação, de memória, de largura de banda ou qualquer outro recurso que seja requerido para ser utilizado ou gasto para executar uma determinada tarefa. O *overhead* é fator limitante do *speedup* e podem ser identificados pelo tempo necessário para comunicação, período de tempo no qual os processadores ficam ociosos e processamento que surgem na versão paralela e não existem na versão sequencial.

3.1.3 Paralelismo e Lei de Amdahl

O *speedup* de um programa que utiliza vários processadores na computação paralela é limitado pela fração sequencial deste programa. Na figura 3.8, é mostrado a forma de execução serial e paralela de um algoritmo, onde a fração de tempo da parte serial é representada por T_s e a fração de tempo da parte paralelizável é representada por T_p.

Podemos concluir que $T(p) = T_s + \frac{T_p}{p}$, onde $T(p)$ representa o tempo de execução do algoritmo paralelo, fazendo uso de "p" processadores. O primeiro termo, no lado esquerdo da equação, representa o tempo que o processador precisa para executar a

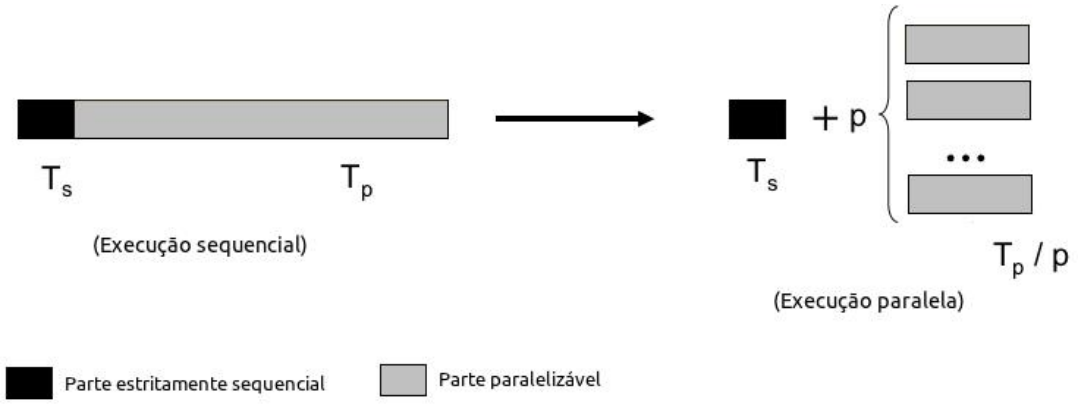


Figura 3.8: *Lei de Amdahl: parte serial x parte paralelizável*

parte serial e o segundo termo, no lado direito da equação, é o tempo que o processador precisa para executar a parte paralela.

Considerando que o algoritmo é composto de uma fração, obrigatoriamente, serial (f_s) e outra paralelizável ($f_p = 1 - f_s$), podemos redefinir o *speedup*, segundo a lei de Amdahl conforme equação 3-4:

$$S(p) = \frac{1}{f_s + \frac{1}{p}(1 - f_s)} \tag{3-4}$$

Como o lado direito desta equação é uniformemente crescente em "p", o limite superior de $S(p)$ ocorre quando p tende a infinito. Então, segundo a lei de Amdahl's, podemos definir um limite para o *speedup* (figura 3.9) conforme a equação 3-5.

$$S(p) \leq \frac{1}{f_s} \tag{3-5}$$

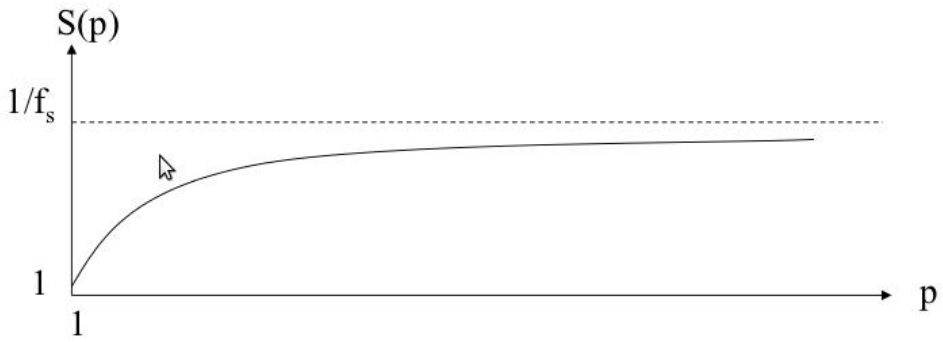


Figura 3.9: *Limite sobre o speedup*

3.1.4 Paralelismo e Lei de Gustafson-Barsis's

De acordo com a lei de Amdahl's, as previsões sobre o ganho de *speedup* são pessimistas. Segundo Gustafson o paralelismo em uma aplicação tende a aumentar,

à medida que cresce o tamanho do problema[21]. Amdahl analisa o desempenho do algoritmo assumindo tamanho fixo da entrada e T_s constante para um determinado número de processadores, enquanto Gustafson observa que T_s é dependente do tamanho da entrada e, provavelmente, irá diminuir para grandes conjuntos de dados.

Enquanto a lei de Amdahl parte do tempo de execução sequencial para estimar o maior *speedup* possível fazendo uso de vários processadores, a lei de Gustafson-Barsis faz o inverso, ou seja, parte do tempo de execução em paralelo para estimar o *speedup* máximo comparado com a execução sequencial. O *speedup* segundo a lei de Gustafson-Barsis pode ser representado pela equação 3-6:

$$S(p) \leq p + T_p \cdot (1 - p) \tag{3-6}$$

3.2 Arquiteturas multinúcleos: GPGPU

Antes do surgimento dos processadores gráficos dedicados, haviam hardwares chamados "placas de vídeo" que encarregavam-se de preparar as imagens a serem exibidas no monitor. Os primeiros modelos dos chips gráficos possuíam pouca memória e baixo poder de processamento, são considerados chips de pouca complexidade. Com o passar do tempo, a evolução dessas "placas de vídeos" torna-se notável, empresas como NVIDIA, ATI, 3dfx e outras produziam modelos que se superavam a cada nova geração. Tecnologias de aceleração 2D e 3D, *pipeline* gráfico, barramentos ISA, VLB, PCI e AGP, SLI (tecnologia que permite que duas placas de vídeos sejam interligadas), adição de diversos chips para processamento em paralelo e aumento da memória dedicada de vídeo fizeram parte da revolução que vem transformando as placas de vídeo em potentes processadores de alta performane (HPC, High-performance computing), chamados de GPU (Graphics Process Unit). Hoje a computação com GPU, não se resume a exibição de informações (gráficas ou textuais) em monitores de vídeo, seu alto poder de processamento e capacidade de execução de tarefas em paralelo, permite à GPU ser utilizada na computação de proposito geral, como um coprocessador da CPU (Central Process Unit).

Desde 2003, a indústria de semicondutores tem estabelecido duas trajetórias principais para o projeto de microprocessador[39]. A trajetória de múltiplos núcleos (*multicore*) busca manter a velocidade de execução dos programas sequenciais enquanto se move para múltiplos núcleos[32]. Os múltiplos núcleos começaram como processadores de dois núcleos, sendo que o número de núcleos praticamente dobra a cada nova geração de semicondutor. Por outro lado, a trajetória baseada em muitos núcleos (*many-core*) busca executar várias aplicações paralelas, a unidade de processamento gráfico (GPU – Graphics Processing Unit), é representante desta arquitetura[32]. A figura 3.10, exibe, lado-a-lado, como são organizados os chips de processamento das tecnologias multinú-

cleos da CPU (multicore) e GPU (many-core). Observe que, na arquitetura *many-core*, também há uma memória DRAM de grande capacidade e exclusiva para uso das GPUs. Além disso, essa arquitetura possui maior número de transistores, reservados para processamento (cores) de um grande número de instruções por unidade de tempo, enquanto a arquitetura *multicore* dedica boa parte do *chip* para *caching* de dados (inclusive com vários níveis de cache) e controle de fluxo. Essa característica permite as arquiteturas *many-core* explorar o processamento paralelo de forma intensa, principalmente para cálculos matemáticos.

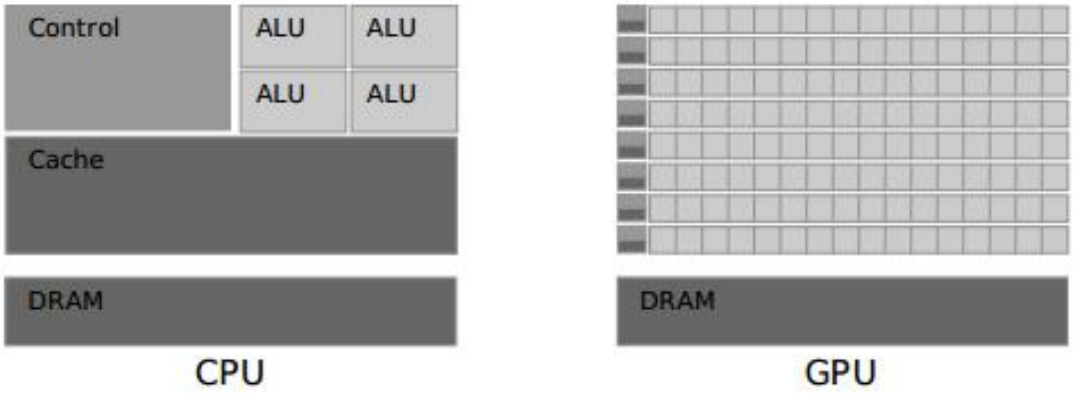


Figura 3.10: *Comparativo: CPU x GPU ([32])*

O modelo de computação *many-core* esta com boa aceitação no mercado, prova disso é o uso destes processadores para construção de máquinas de alto desempenho, como exemplo, no **Top 500 supercomputer sites** (www.top500.org), na classificação dos 500 computadores mais rápidos do mundo, alguns computadores como o **Nebulae** e **Tianhe-1A** usam GPUs e figuram na lista dos computadores mais rápidos [1].

3.2.1 Computação em GPGPU

Nos últimos anos, a indústria de jogos eletrônicos tem desenvolvido unidades de processamento gráfico cada vez mais poderosas, fruto do grande poder computacional exigido pelos jogos atuais. A capacidade de processamentos destas placas gráficas (GPU) chamou a atenção dos pesquisadores, que fazem uso das mesmas para construir algoritmos maciçamente paralelos. O cenário atual para processamento de alto desempenho (HPC, High-performance computing), com novos processadores *many-core*, tem se tornado muito atraente devido ao excelente desempenho e baixo custo.

Esses novos processadores são conhecidos como GPGPU (General-purpose computing on graphics processing units), computação de propósito geral em unidade de processamento gráfico. A sigla GPGPU define uma arquitetura computacional, maciçamente paralela, que utiliza a GPU não apenas para tarefas de renderização gráfica. Seu grande poder computacional e baixo custo, estão motivando muitos desenvolvedores de

aplicação e cientistas a mudar e/ou construir trechos de códigos, que necessitem de computação intensa, para execução sobre as GPUs.

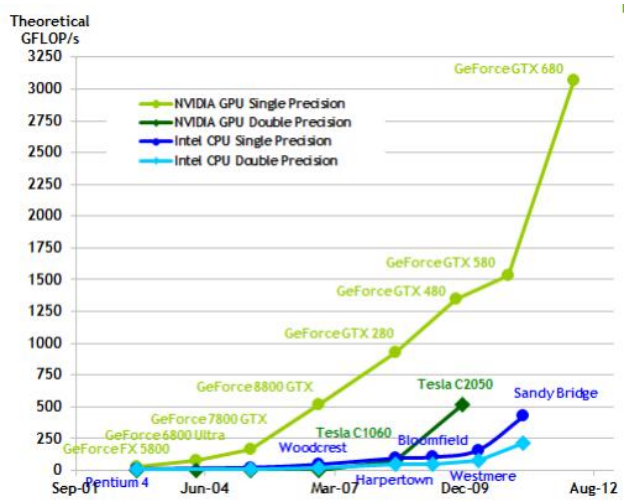
O projeto de *hardware* de uma GPU concentra-se em incluir a maior quantidade possível de núcleos de processamento, capazes de realizar muitos cálculos matemáticos simultaneamente[16]. A título de comparação, o número de núcleos de uma CPU comercial girar em torno de 1 (uma) dezena, em alguns modelos de GPU podemos chegar a milhares de núcleos de processamento [11, 2].

As placas gráficas, responsáveis pelas imagens dos jogos eletrônicos, avançaram tanto que superam até as mais rápidas CPU (Central Process Unit). Áreas como as de exames de diagnósticos médicos por imagens, entretenimento, investimentos financeiros, exploração de petróleo, bioinformática e qualquer outra área que faça uso intensivo de cálculos numéricos podem aproveitar da tecnologia de computação em GPU para acelerar o processamento de suas aplicações[16]. A computação com GPU é o uso de uma unidade de processamento gráfico como um coprocessador para acelerar as CPUs para computação de engenharia e científica de propósito geral [11]. O coprocessamento refere-se ao uso de um acelerador, neste caso a GPU, para aliviar a carga da CPU a fim de aumentar a eficiência computacional.

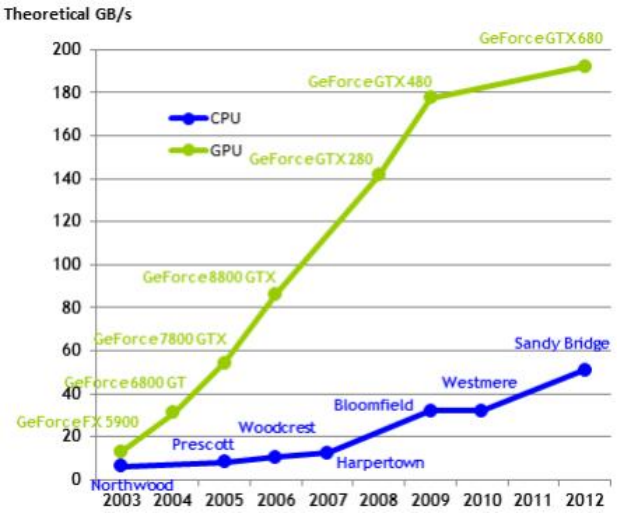
Processadores com muitos núcleos, especialmente as GPUs, têm liderado a corrida do desempenho em ponto flutuante desde 2003 (figura 3.12 e 3.11(a)). Embora o ritmo de melhoria de desempenho do micro-processadores de uso geral tenha desacelerado significativamente, o das GPUs continua a melhorar de maneira continua [32]. A diferença de desempenho entre GPUs com muitos núcleos e CPUs com múltiplos núcleos é explicada nas filosofias de projeto para cada tipo de processador (figura 3.10). A arquitetura de uma CPU é otimizada para o desempenho de código sequencial, para uma (ou algumas poucas) *threads*. O uso de técnicas de pipeline, cache, predição de desvios e execução fora de ordem possibilitaram reduzir as latências de acesso as instruções e dados na memória e, desta forma, permitir uma alimentação continua ao processador.

Outro ponto importante é a largura de banda da memória, figura 3.11(b). Os chips gráficos têm operado com largura de banda bem superiores aos dos chips de CPU disponíveis na mesma época. Isso se deve ao fato dos processadores de uso geral precisarem satisfazer os requisitos dos sistemas operacionais, aplicações e disposítivos de E/S, o que torna a largura de banda da memória mais difícil de aumentar. Ao contrário, com modelos de memória mais simples e menos restrições associadas, os projetistas da GPU podem alcançar uma largura de banda de memória mais alta com facilidade[32].

A filosofia de projeto das GPUs busca otimizar a vazão de execução de um número maciço de *threads*. O hardware tira proveito de um grande número de threads de execução para encontrar trabalho para fazer, enquanto alguns deles estão esperando por acesso à memória de longa latência. As GPUs são projetadas como mecanismos de



(a) *Operações de ponto flutuante por segundo*



(b) *Largura de banda CPU x GPU*

Figura 3.11: *Desempenho CPU x GPU, Fonte: [9].*

cálculo numérico, e elas não funcionarão bem em algumas tarefas em que as CPUs são naturalmente projetadas para funcionar bem; portanto, é de se esperar que a maioria das aplicações usará tanto CPUs quanto GPUs, executando as partes sequenciais na CPU e as partes numericamente intensivas nas GPUs.

Entretanto, além do desempenho, as GPUs contam com importante fator para seu sucesso, a "presença de mercado"; ter forte presença no mercado é primordial para o sucesso de uma arquitetura paralela, o que não aconteceu com outras arquiteturas paralelas tradicionais, devido ao seu alto custo. Atualmente, existem centenas de milhões de GPUs presentes no mercado, grande parte dos PC's possuem GPUs, permitindo, pela primeira vez na história, que a computação maciçamente paralela seja viável [32]. Outros fatores que impulsionam o sucesso das GPUs são custo, pois são economicamente atraentes, e suporte para o padrão de ponto flutuante estipulado pelo *Institute of Eletrical*

and Electronics Engineers (IEEE).

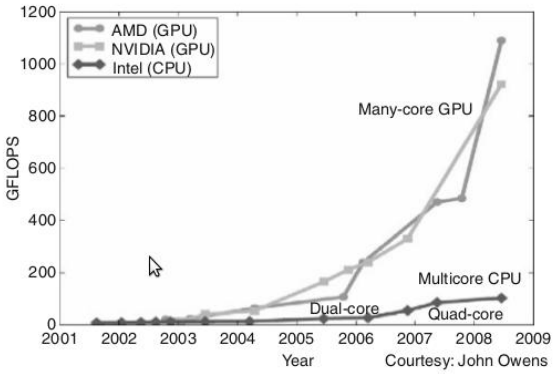


Figura 3.12: *Diferencial de performance entre GPU's e CPUs[32]*

É aconselhável implementar uma solução que exponha uma quantidade de paralelismo superior aos recursos disponíveis no hardware, pois a GPU tem suporte eficiente para *threading*. Cada duplicação da quantidade de núcleos da GPU oferece mais recursos de execução do hardware, que exploram mais do paralelismo exposto para obter maior desempenho, ou seja, o modelo de programação paralela da GPU foi projetado para que tenha uma escalabilidade transparente [32].

NVIDIA e ATI desenvolvem GPUs que disputam o mercado de computação paralela sobre GPUs. Foram criadas linguagens específicas para a chamada GPGPU (General Purpose computation on Graphics Processing Unit), a mais conhecida, e projetada pensando em GPGPU e foi proposta pela própria NVIDIA, e chama-se CUDA (Computer Unified Device Architecture). A ATI deseja explorar o mercado com uso da sua linha de produtos **ATI Stream** e como uso da linguagem de programação paralela OpenCL. o OpenCL é uma abordagem aberta e compatível com ambas tecnologia. No entanto, CUDA é um produto com maior visibilidade, mais maduro e com muitos desenvolvedores adeptos. Por essa maior aceitação do mercado, adotaremos a arquitetura da NVIDIA para computação sobre GPUs e a linguagem CUDA.

3.2.2 Arquitetura da NVIDIA

CUDA é uma plataforma de computação paralela e, também, um modelo de programação paralela inventados pela NVIDIA. A arquitetura CUDA é suportada por todas as placas da NVIDIA a partir do chipset G80 (processador gráfico da NVIDIA que possui aproximadamente 700 milhões de transistores). As GPUs são extremamente paralelas, possuem muitos núcleos e uma imensa quantidade de ULAs menos sofisticados que uma CPU, diversas caches e uma memória dedicada que é compartilhada por todos os núcleos.

A figura 3.13 mostra a arquitetura de uma GPU típica, organizada em uma matriz de *stream multiprocessor* (SMs). Nessa arquitetura, dois SMs formam um bloco, no entanto, o número de SMs em um bloco pode variar de uma geração de GPUs CUDA para outra geração (ver arquitetura Fermi, na figura 3.14). Além disso, cada SM tem um certo número de *streamming processors* (SPs) que compartilham a lógica de controle e a cache de instruções. Atualmente, cada GPU vem com até 6 gigabytes de memória, chamadas de *memória global* [32]. O acesso a memória global é considerado lento, no entanto, por suportar uma grande quantidade de *threads*, essa latência de memória pode ser ocultada através do chaveamento que ocorre entre grupos (blocos/*warps*) de threads.

Ainda sobre a figura 3.13, podemos observar que a mesma consiste em 16 multiprocessadores de *streaming* (SMs), contendo 8 processadores de streaming (SPs). Os 8 núcleos (SPs) executam as instruções das *threads* segundo o modelo SIMD, executando uma *warp* (unidade mínima de escalonamento da arquitetura NVIDA, composta de 32 *threads*), a cada 4 ciclos. Isso permite que cada processador possua apenas uma unidade de instrução, que envia dados para todos os 8 núcleos.

O *warp* é um grupo de 32 *threads* que executam em SIMD. Durante a execução as *threads* de um bloco são agrupadas em *warps* de 32 threads, que são a unidade de escalonamento do multiprocessador. Os *warps* são formados por faixas contínuas de *threads* dentro de um bloco. O conhecimento dos *warps* permite a criação de várias otimizações. O multiprocessador é capaz de disparar somente uma instrução de cada vez para todas as *threads* de um *warp*. Caso *threads* de um mesmo *warp* sigam caminhos diferente durante a execução, o multiprocessador terá que executar o *warp* em várias passadas, executando a cada vez as *threads* que seguiram um mesmo caminho [8, 32].

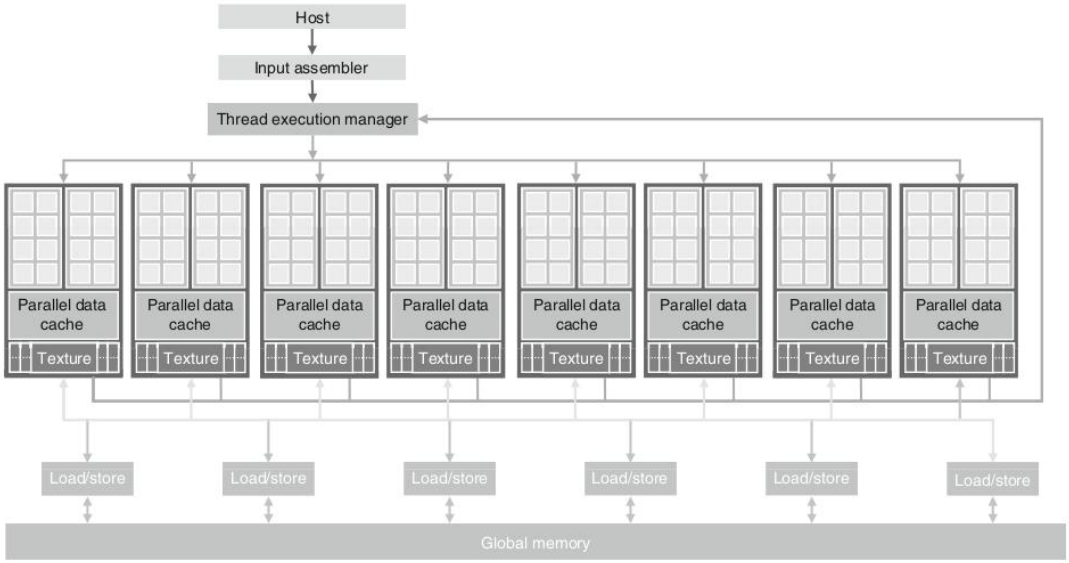


Figura 3.13: *Arquitetura de uma GPU (NVIDIA CUDA - [32])*

O G80, GT200 e a Fermi (GF100) são arquiteturas construídas pela NVIDIA, um

quadro resumo das principais características destes chips são apresentados na tabela 3.1. A linha de processadores da NVIDIA criada para dar suporte ao modelo de programação paralela é formada por modelos GeForce™, Quadro™e Tesla™. As GPUs Tesla™são projetadas e otimizados para computação científica, engenharia e gráficas, que exigem intensos cálculos de ponto flutuante e que requeirem alto desempenho. Os modelos GeForce™é voltado para o mercado de placas gráficas e aceleradoras, é o modelo com maior representatividade. O modelo Quadro™é uma solução gráfica profissional, foi programado para trabalhar principalmente com softwares CAD (Computer-Aided Design), criação de conteúdo digital e outros, são modelos poderosos e indicados para aplicativos gráficos mais exigentes.

Na tabela 3.1 são apresentados algumas das principais características das arquiteturas NVIDIA. Nota-se que a quantidade de transistores vem dobrando a cada nova geração de arquitetura, permitindo que mais unidades de processamento sejam adicionados a GPU. A quantidade de SPs (streamming processors ou, como é conhecido nas arquiteturas mais recentes, "cuda cores"), cresce na mesma proporção da quantidade de transistores, isso permite que mais *threads* sejam lançadas e o paralelismo seja melhor explorado. Outro fator importante é a capacidade da memória global, o crescimento desta memória colabora para que porções maiores de um problema sejam carregas nessa memória, fazendo que soluções mais eficazes seja construída. Além disso, melhorias para tratamento de número de ponto flutuante vem sendo realizadas a cada geração, atualmente o modelo Fermi™é compatível com aritmética de ponto flutuante de dupla precisão, conforme padrão IEEE 754.

Tabela 3.1: *Modelos de arquitetura da NVIDIA*

	G80	**GT200**	**GF100 (Fermi)**
Exemplo de produto	GeForce 8800 GTX	Tesla C1060	Tesla C2050
Transistores (bilhões)	0,681	1,4	3
SMs	16	30	16
Quantidade SPs por SM	8	8	32
Total de SPs (cuda cores)	128	240	512
Giga Flops	500	1.063	1.581
Threads por SM	768	1.024	1.536
Memória global (GB)	0,75	4	6

A arquitetura Fermi™trouxe diversas melhorias para o modelo NVIDIA CUDA. A figura 3.14 mostra o modelo Fermi, constituída de 3 bilhões de transistores, possui 512 núcleos CUDA, cada núcleo executa uma instrução de ponto flutuante ou inteiro por *thread*. Esses núcleos cuda (CUDA core) são organizados em 16 multiprocessadores (cada um com 32 núcleos CUDA), compartilhando um cache (L2), seis interfaces DRAM de 64-bits e uma interface PCI-Express para comunicação com o a CPU (host). O escalonador

GigaThread distribui os blocos de threads entre os multiprocessadores (SM) disponíveis, realizando o balanceamento da carga de trabalho na GPU e, se possível, executando múltiplos kernels em paralelo[10].

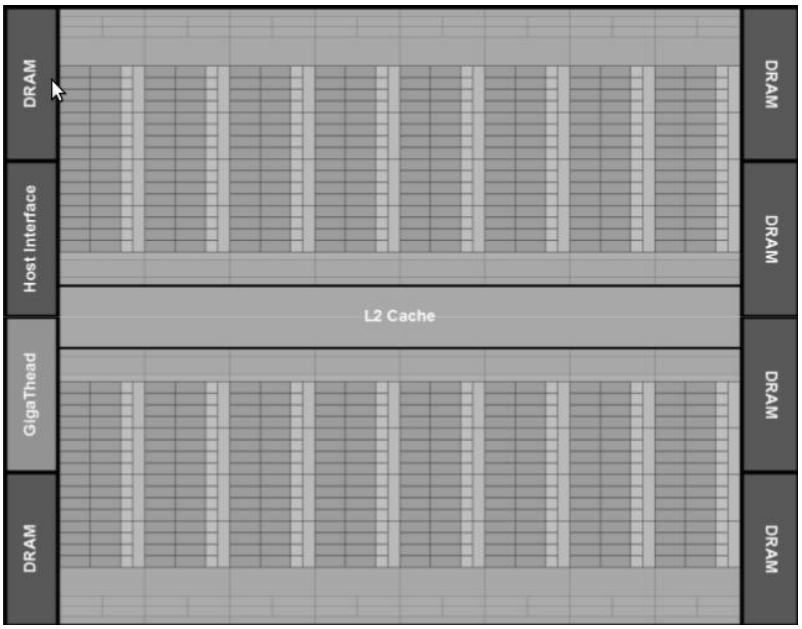


Figura 3.14: *Arquitetura Fermi: 16 Streamming Processors com 32 cuda cores [10].*

Os *Streamming Multiprocessors* (SMs) são elementos de processamento escaláveis, sob o qual um conjunto uniforme de dados podem ser manipulados em paralelo. Tem modelo semelhante ao SIMD (ver 3.1.1), da Taxonomia de Flynn, onde é possível a execução de várias threads. Os multiprocessadores podem conter diversos CUDA cores, de acordo com a geração de sua arquitetura. Na arquitetura Fermi (ver figura 3.15), cada SM possui 32 CUDA cores, e cada núcleo de processamento tem uma unidade lógica e aritmética (ULA) e uma unidade de ponto flutuante (FPU). As GPUs anteriores utilizavam aritmética de ponto flutuante segundo o padrão IEEE 754-1985, o que não ocorre com a Fermi, que utiliza o padrão IEEE 754-2008 para manipular números de ponto flutuante de precisão simples (32 bits) e de precisão dupla (64 bits), que faz uso do FMA (Fused Multiply-add). O FMA traz melhorias nas funções de adição e multiplicação pelo fato de não truncar os dígitos dos números, e com isso mantem o nível de precisão na operação [10]. Aritméticas de precisão dupla são o coração das aplicações HPC (High Performance Computation), por isso a arquitetura Fermi busca melhorar a performance para operações com este tipo de dado. Quatro unidades de funções especiais (SFUs), que executam funções transcendentais, como *seno* e *cosseno*, compõem um SM dessa arquitetura. Cada SFU executa uma instrução por thread para cada pulso, são necessário 8 pulsos para executar uma *warp* (grupos de 32 *threads* gerenciados e executados pela GPU) completa.

Outras características são que um SM Fermi possui 16 unidades de leitura e escrita, o que permite o cálculo de endereços para dezesseis *threads* por pulso; há dois escalonadores (warp scheduler) por SM que permitem manter duas *warps* (grupos de 32 *threads*) em execução simultaneamente, 64 KB para a memória compartilhada e cache L1.

Uma das mais importantes tecnologias da arquitetura Fermi é o escalonamento de threads em dois níveis. No nível de chip, o mecanismos de escalonamento distribui os blocos de threads para os vários SMs existentes, enquanto que no nível do SM, os escalonadores de *warp* distribui as 32 threads de cada *warp* para as unidades de execução. Além de conseguir melhorias quanto a quantidade de threads executadas (throughput), o escalonador GigaThread^{TM}, consegue mais agilidade na troca de contexto, execução concorrente de kernels e melhoria quanto ao escalonador de bloco de threads.

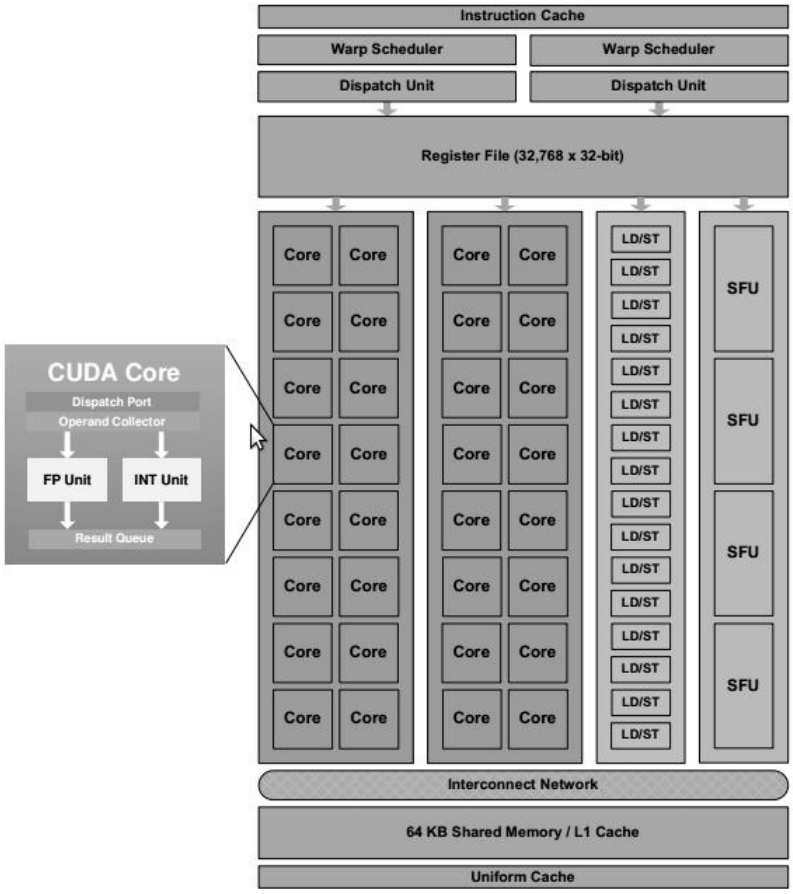


Figura 3.15: *Fermi: Streamming Multiprocessor (SM) [10].*

Memórias da GPU

CUDA admite vários tipos de memórias que podem ser usados para alcançar altas velocidades de execução para suas funções [32]. A figura 3.16 mostra o modelo de memória utilizado em CUDA. Uma GPU trabalha com os seguintes tipos de memórias:

global, constante, registradores, compartilhada e de textura (não exibida nessa figura, mas pode ser vista na figura 3.13). Essas memórias podem ser lidas e escritas, possuem vantagens e desvantagens, um ponto que requer atenção especial ao desenvolver aplicativos para GPUs e voltado ao acesso à memória, uma implementação que faça uso otimizado desses diferentes tipo de memória conseguirá melhor performance para suas aplicações. A solução mais fácil, pois requer, um menor esforço é a utilização dos dados em memória global. Esta implementação é simples de ser desenvolvida, mas sua performance pode ser sofrível, pois os tempos de acesso aos dados serão altos devido a latência de acesso a esse tipo de memória, tornando os ganhos na implementação paralela baixo. É considerada a *memória de trabalho* principal, onde a GPU armazena os resultados, antes de devolve-los a memória principal da CPU. Todas as *threads* tem acesso a esta memória.

Apesar do baixo desempenho, sua latência pode ser ocultada, através da forma de execução de *threads* da GPU NVIDIA. Um multiprocessador gasta 4 ciclos para requisitar dados da memoria para um *warp*. Acessar a memoria global resulta em 400 a 600 ciclos adicionais de latência de memória [8]. A maioria da latência da memoria global pode ser escondida definindo um grande número de blocos de *threads* e utilizando o máximo possível de registradores, memoria compartilhada e memória de constante para realizar o acesso aos dados. Normalmente o número de *warps* necessárias para manter o escalonador ocupado depende do código do *kernel* e do nível de paralelismo. Em geral, mais *warps* são necessárias se o número de instruções que realizam operação aritméticas for baixo [9, 8]. Em geral esses valores ficam em torno de 6 a 10 warps.

A memória nas GPUs não difere muito das memórias DRAM que são tipicamente utilizadas em computadores pessoais. Elas são conhecidas como GDDR (Graphics Double Data Rate). O padrão utilizado atualmente é o GDDR5 (quinta geração das memórias GDDR) e trabalham com 8 transferências por ciclo de clock e frequência acima de 900 MHz. Atualmente, da arquitetura Fermi^{TM}em diante, temos GPUs com capacidade de memória superior a 4 GB (gigabytes).

A memória compartilhada possui um tamanho restrito (48 KB na arquitetura Fermi^{TM}), mas possui acesso extremamente rápido. Os acesso a essa memória é definido pelo desenvolvedor e a mesma é compartilhada por todas as *threads* do SM, por isso é necessário uma manipulação de forma diferenciada, o que pode tornar sua lógica de utilização mais complexa que a global. É uma memória que pode ser acessada somente pelo seu próprio SM. Já os registradores, assim como a memória compartilhada, também são memórias integradas ao chip da GPU, são extremamente rápidas, *unitárias* e de utilização extremamente *limitados*.

Na memória constante, os dados são recebidos da memória RAM da CPU (ou *host*, como é normalmente conhecida) e podem ser lidos por qualquer SM, no entanto não podem ser alterados - possui alta velocidade de acesso. Por fim, temos a memória

de textura, um outro tipo de memória somente para leitura. A memória de textura foi especialmente projetada para aplicações gráficas onde o padrão de acesso a memória são previsíveis e exibem grande localidade espacial[47] (*threads* fazem leituras de endereços próximos), também possui controle de cache transparente e eficiente.

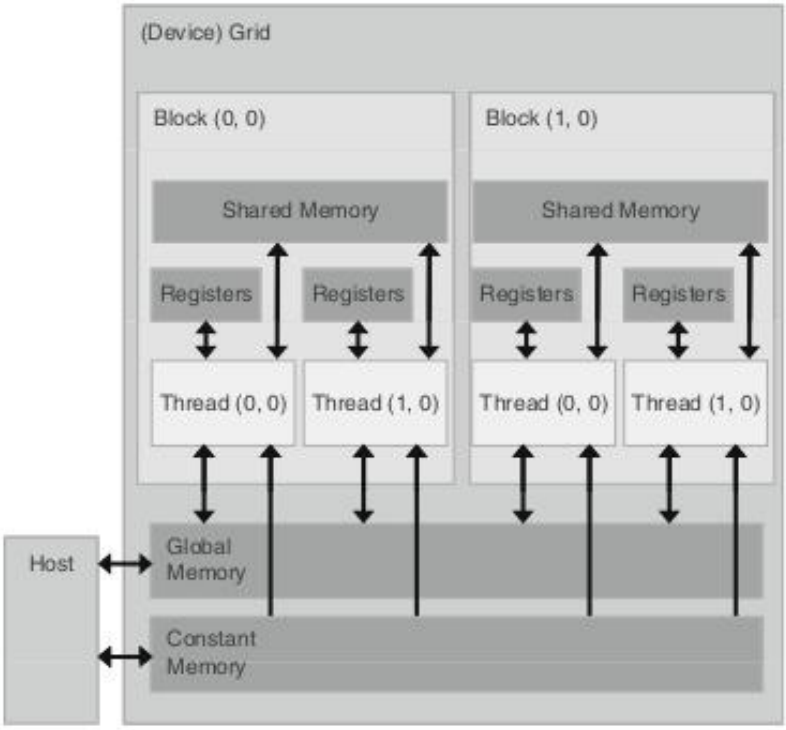


Figura 3.16: *Organização das memórias*

A arquitetura Fermi trouxe algumas inovações ao subsistema de memória da GPU. Por exemplo, o NVIDIA Parallel DataCache^{TM}traz a cache L1 configurável e cache L2 unificada. Alguns algoritmos fazem o mapeamento dos dados naturalmente para a memória compartilhada, outros não conseguem usufruir desta característica (muitas vezes, devido à pequena capacidade dessa memória) e se beneficiam melhor de uma memória cache. Uma forma atender essas duas abordagens é permitir a configuração da capacidade dessas memórias. Existem 64 KB de memória para cada SM, que podem ser divididos de duas formas distintas. Na primeira forma podemos privilegiar o uso da memória compartilhada, definindo 48 KB para a mesma e 16 KB para a cache L1. Na segunda forma, privilegiamos a cache L1, invertendo a distribuição das memórias.

A arquitetura Fermi trouxe as primeiras GPUs com suporte a memória com código de correçaõ de erros (ECC-Error Correcting Code), uma solicitação antiga dos desenvolvedores para computação de alto desempenho. Outra melhoria implementada foi a aceleração das operações atômicas, graças a adição de unidades especificas para realizar essas operações, e a cache L2, trouxe melhorias de até 20x mais rápidas que arquiteturas anteriores.

3.3 Ambientes de Programação

Até 2006, os chips gráficos eram muito difíceis de usar para aplicações não gráficas, pois os programadores tinham que usar o equivalente das funções da API gráfica para acessar os núcleos processadores, significando que técnicas de OpenGL©ou Direct3D© eram necessárias para programar esses chips. Em 2007, com o lançamento da CUDA, a NVIDIA©dedicou área do silício para facilitar a comodidade da programação paralela; hardware adicional foi acrescentado ao chip [32].

A NVIDIA desenvolveu o compilador CUDA C/C++, bibliotecas e softwares de apoio para permitir que os programadores rapidamente acessassem o novo modelo de computação paralela de dados e desenvolvessem aplicações[32]. Por outro lado, foi desenvolvido o OpenCL™, um padrão aberto que é suportado por NVIDIA, ATI, Intel e Outros. OpenCL é uma API padronizada, multiplataforma, para computação paralela baseada na linguagem C [32]. OpenCL é gerido pelo consórcio tecnológico Khronos Group. As próximas seções detalham estes ambientes de programação.

3.3.1 CUDA

CUDA™é uma plataforma de computação paralela e um modelo de programação criado pela NVIDIA. Ela permite aumentos significativos de performance computacional ao aproveitar a potência da unidade de processamento gráfico (GPU). As GPUs GeForce™, Tesla™e Quadro™da NVIDIA oferecem suporte ao modelo de programação paralela NVIDIA CUDA™. No ambiente de programação CUDA, o sistema computacional distingue entre o que é executado na CPU, chamado de *host* (hospedeiro), e o que é executado na GPU, chamada de *device* (dispositivo).

Em linhas gerais, podemos resumir CUDA como uma hierarquia de *threads* mapeadas para os processadores de uma GPU; uma GPU executa um ou mais *kernels*; um SM (streamming multiprocessor) executa uma ou mais blocos de threads e os CUDA Cores (ou SPs) e outras unidades de execução que compõem um SM executam as *threads*. O SM executa *threads* em grupos de 32 *threads*, chamado *warp*. Um programador pode ignorar o conceito de *warp* e desenvolver pensando na execução de apenas uma *thread*, entretanto, é possível conseguir melhorias de performance agrupando *threads* em uma *warp*, desde que seus códigos sigam caminhos de execução não divergentes pelos mesmos caminhos e acessem endereços de memórias próximos [10].

3.3.2 Estrutura de um programa em CUDA

Um programa em CUDA consiste em partes de código executados no *host*, normalmente partes que não exigem paralelismo, e outras partes que apresentam

grande quantidade de paralelismo, executam no *device*. É tarefa do compilador C da NVIDIA^{TM}(*nvcc*) separar os dois códigos durante a compilação. Como já mencionado, o código em CUDA é uma extensão do C, onde são utilizadas algumas palavras chaves que rotulam as *funções paralelas*, chamadas **kernels**, e suas estruturas de dados associadas. [32].

Em um típico programa NVIDIA CUDA C é comum a intercalação de código sequencial e código paralelo. Inicialmente, ocorre a preparação dos dados, em seguida, o código paralelo é acionado através do *kernel* CUDA. O seguinte fluxo pode ser identificado diversas vezes no mesmo programa:

1. um conjunto de dados são inicializados pelo *host*
2. os dados são copiados da memória do *host* para a memória do *device* (GPU)
3. a CPU invoca a execução do *kernel*
4. GPU divide os dados e realiza cálculos em paralelo
5. os dados modificados são copiados de volta para a memória do *host*

A implementação de uma função para CUDA é chamada *kernel*, o código escrito nesta função será executado na GPU por todas as *threads* lançadas pela aplicação[9]. Como todas essas *threads* executam o mesmo código, a programação CUDA e um caso do conhecido estilo de programação de único programa e múltiplos dados (SPMD – Single-Program, Multiple-Data, ver seção 3.1.1), um estilo de programação popular para sistemas de computador maciçamente paralelos [32].

Kernels e hierarquia de threads

As funções *kernel* normalmente geram grande número de *threads* para explorar o paralelismo de dados. Devido ao suporte eficiente do hardware, essas *threads* são de peso muito mais leve do que aquelas das CPUs, utilizando pouquíssimos ciclos de clock para serem geradas e escalonadas. A figura 3.17 mostra a execução de um programa em CUDA, como citado anteriormente o primeiro passo consiste no *host* (CPU) que faz a chamada ao *kernel*, repassando o controle para o *device* (GPU). Todas as *threads* geradas por um *kernel* durante uma chamada são conhecidas coletivamente como *grid* (grade). Veja na figura 3.17 que após as *threads* de um *kernel* completar sua execução o controle volta para o *host*, que poderá lançar um novo *grid*.

Em geral, CUDA estende as declarações de função C com três palavras-chave qualificadoras: *__global__*, *__device__* e *__host__*. A palavra-chave *__global__* indica que a função sendo declarada é uma função de *kernel* CUDA. A função será executada no *device* e só pode ser chamada pelo *host* para gerar uma grade de *threads* em um *device*. O uso da palavra-chave *__device__* indica que a função sendo declarada é uma função de *device* CUDA, isto é, é uma função executada no device e só pode ser chamada por uma

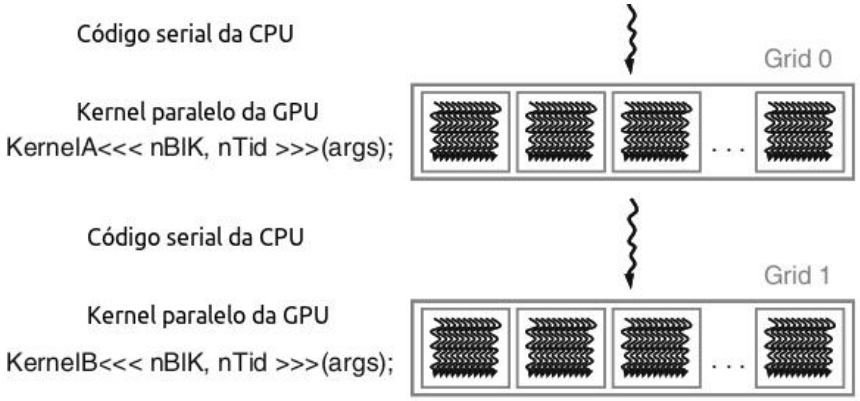


Figura 3.17: *Execução de um programa em CUDA, fonte: [32]*

.

função de *kernel* ou outra função de *device*. Por fim, a palavra-chave *__host__* indica que a função sendo declarada é uma função de *host*, ou seja, só pode ser chamada e executada no próprio host.

Quando um *kernel* é disparado, ele é executado como uma grade de *threads* paralelas. As *threads* em uma grade são organizadas em uma hierarquia de dois níveis, conforme ilustra a figura 3.18. Nó nível mais alto, cada grade consiste em um ou mais **blocos de *threads***. Todos os blocos em um *grid* tem o mesmo número de *threads*. Na figura 3.18 podemos ver um *grid* organizado como um matriz 2*x*3, com seis blocos. Cada bloco tem uma coordenada bidimensional exclusiva, dada pelas palavras-chave *blockIdx.x* e *blockIdx.y* [32].

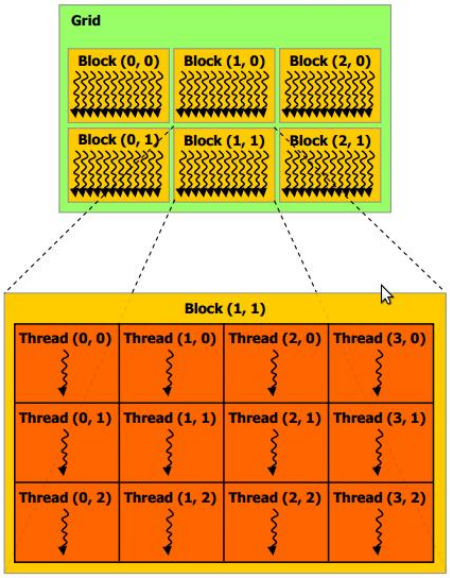


Figura 3.18: *Grade com vários blocos de threads, fonte: [9]*

.

Cada bloco de *threads*, por sua vez, é organizado como uma matriz tridimensional de *threads*, como um tamanho total de até 512 *threads*. As coordenadas das *threads*

em um bloco são definidas de modo exclusivo por três índices de *thread*: *threadIdx.x*, *threadIdx.y* e *threadIdx.z* [32]. Não é necessário fazer uso de todas as dimensões, o uso das dimensões pode estar relacionado a definição da estrutura dos dados. Na figura 3.18, cada bloco de *thread* é organizado em um matriz bidimensional 4*x*3 *threads*, ou seja, 12 *threads* por bloco. Isso dá um *grid* com 72 *threads*, para o exemplo da figura 3.18.

Memória no CUDA

Os dados a serem processados por todas as *threads* lançadas pelo *grid* são transferidos da memória do *host* para a memória global do *device* (tipo de memória DRAM, ver 3.2.2). As *threads* acessam sua parte dos dados a partir da memória global. Entretanto, o acesso a memória global costuma ter atrasos, pois o congestionamento de tráfego nos caminhos de acesso à memória global dificultam as threads de fazerem progresso. Para contornar esse congestionamento, CUDA oferece métodos de acesso a outras memórias da GPU, o que pode reduzir o tráfego e melhorar o desempenho.

As *threads* CUDA podem acessar dados à partir de multiplos espaço de memória durante a execução de um kernel, veja 3.16. Todas as *threads* tem acesso a uma mesma memória global. Cada bloco de *threads* possui uma memória compartilhada por todas as *threads* do bloco e com o mesmo tempo de vida do bloco. Cada *thread* possui uma memória privada, que são os registradores.

Os registradores e a memória compartilhada são memórias no *chip*, chamadas *on-chip*. As variáveis que residem neste tipo de memória podem ser acessadas em velocidade muito alta de maneira altamente paralela. Normalmente, os registradores são utilizados para manter variáveis acessadas frequentemente, enquanto que a memória compartilhada é um mecanismo eficiente para as *threads* de um bloco cooperarem compartilhando seus dados e resultados intermediários.

Uma abstração de memória utilizada em CUDA é a memória *local*. Esta memória tem escopo somente de uma *thread* e fica localizada na memória global. Estrutura de vetores que ocupam grande quantidade de registradores são tipos de variáveis candidatas a serem colocadas nesta memória. Por residir na memória global, essa memória possui a mesma latência e largura de banda dessa memória. Nas arquiteturas Fermi^{TM}e superiores as variáveis dessa memória são armazenadas nas *caches* L1 e L2.

Além destas memória, CUDA pode utilizar também memórias constantes e de textura. Na tabela 3.2 são destacas algumas características importantes para cada tipo de memória, por exemplo, memória *on-chip* fica localizada dentro da GPU, logo são mais rápidas. Normalmente o acesso a memória global é 100 vezes mais lento que o acesso a uma memória *on-chip*.

Compute Capability

A NVIDIA distingue a capacidade de computação de cada GPU no ambiente CUDA através de uma classificação chamada *Compute capability*. Essa classificação deve ser conhecida pelo desenvolvedor, pois existem limitações da arquitetura CUDA que são especificados em cada revisão.

O número de revisão do processador gráfico é divido em duas partes. A primeira parte, chamado número de revisão principal, identifica a arquitetura da GPU (1-Tesla, 2-Fermi e 3-Kepler). a segunda parte, número de menor revisão, corresponde a melhorias implementadas por uma determinada arquitetura, possivelmente novas funcionalidades [9].

3.3.3 OpenCL

O OpenCL™(*Open Computing Language*) é um padrão aberto, mantido pelo Khronos Group, que permite além do uso de GPUs, outras opções como o processador Cell. O OpenCL também suporta tanto o CUDA quanto o Brook+, permitindo o desenvolvimento de aplicativos para ser executado em qualquer plataforma. É uma API padronizada, multiplataforma, para computação paralela baseada na linguagem C.

O OpenCL foi projetado para permitir o desenvolvimento de aplicações paralelas portáveis para sistemas com *devices* de computação heterogêneos. O desenvolvimento de OpenCL foi iniciado pela Apple®e desenvolvido pela Khronos Group, o mesmo grupo que administra o padrão OpenGL®. O modelo de execução OpenCL é semelhante ao utilizado em CUDA, tanto que existe uma tabela de correspondência de um-para-um sobre os principais recursos [32].

Por ter um modelo de gerenciamento para a portabilidade multiplataformas e multifornecedores, OpenCL também é mais complexo. Programas OpenCL devem ser preparados para lidar com uma diversiade de hardware muito maior e, assim, exibirão mais complexidade. Além disso, muitos recursos do OpenCL são opcionais e podem não ser aceitos em todos os devices, de modo que um código OpenCL portável deverá evitar o uso desses recursos opcionais, recursos esses que podem trazer melhor desempenho[32]. Como resultado, um código OpenCL portável pode ter seu desempenho degradado.

Um programa OpenCL consiste em duas partes: *kernels* que executam em um ou mais *devices* OpenCL e um programa host que gerencia a execução dos *kernels*. Ao disparar uma função do *kernel*, é definido um espaço de indexação (*index space*) conhecido como *NDRange*, o código será executado por *work itens* (*threads* em CUDA). Os itens de trabalho formam *work groups* (blocos em CUDA). Assim como em CUDA, podem haver sincronismos através do uso de barreiras, apenas dentro do *grupo de trabalho* (work groups). A única forma de sincronização entre *grupos de trabalho* é a

finalização de um *kernel* para início de um novo. A figura 3.19 apresenta o modelo de execução de um programa em OpenCL.

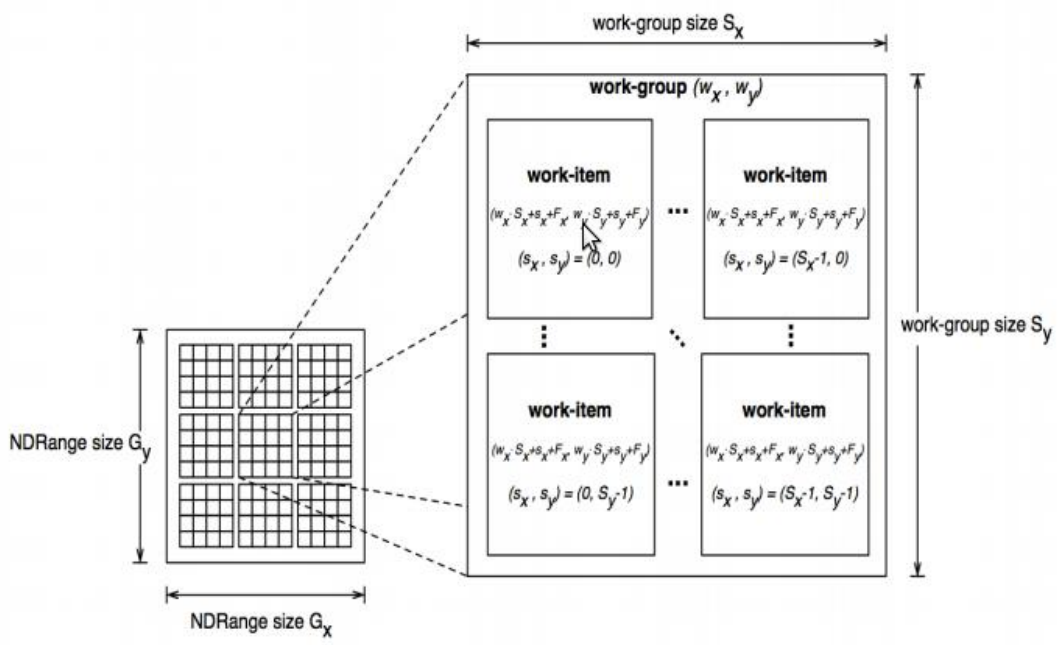


Figura 3.19: *NDRange, Work-groups e Work-item, fonte: [31]*

.

No modelo de memória em OpenCL, os *work-item(s)* executados pelo kernel tem acesso a quatro tipos de regiões de memória [31]:

1. Memória global: é uma região de memória que permite acesso de leitura e escrita para todos os *work-items* em todos os *work-groups*.
2. Memória constante: é uma região da memória global que permanece constante durante toda a execução do *kernel*. É função do *host* alocar e inicializar a memória constante.
3. Memória local: é uma região de memória utilizada por um *work-group*. Pode ser utilizada para alocar variáveis compartilhadas por todos *work-items* que fazem parte do *work-group*.
4. Memória privada: é uma região de memória reservada para cada *work-item*. Variáveis definidas para um *work-item*, não estará disponível para outros *work-items*.

Tabela 3.2: *Características de cada tipo de memória*

Tipo de memória	Escopo	Tempo de vida	On/off chip
Registrador	thread	thread	on
Local	thread	thread	off
Compartilhada	bloco	bloco	on
Global	grade	aplicação	off
Constante	grade	aplicação	off
Textura	grade	aplicação	off

CAPÍTULO 4

Proposta de Implementação

> Algoritmos e arquiteturas de multiprocessamento estão intimamente ligados. Não podemos pensar em um algoritmo paralelo, sem pensar no hardware paralelo que vai apoiá-lo. Por outro lado, não podemos pensar em hardware paralelo, sem pensar no software paralelo que irá conduzi-lo. O paralelismo pode ser implementada em diferentes níveis em um sistema de computação usando técnicas de hardware e software [21].

Frente ao problema de filogenia e do sistema de computação disponível (computação paralela sobre GPU), temos que desenvolver uma solução que realize todos os passos computacionais necessários sobre o conjunto de dados no menor tempo possível, mantendo a acurácia na geração dos dados. Inicialmente, dividimos o problema em quatro etapas principais (ver figura 4.1):

1. preprocessamento: à partir da árvore (formato Newick) iremos ler, consistir e gerar arquivo pronto para ser processado em paralelo.
2. inserção de espécies/geração de árvores: consiste na inserção de espécies incertas sob uma árvore já conhecida. Dada uma árvore filogenética e uma lista de outras espécies faltantes/perdidas, isto é *espécies com incertezas filogenéticas* (PUT, phylogenetically uncertain taxa), iremos gerar variações desta árvore na qual as novas espécies são aleatoriamente posicionadas a partir de um dado nó consensual (MDCC - most derived consensus clade).
3. cálculo da matriz de distância: calcular a matriz de distância evolutiva, isto é a matriz contendo as distâncias entre todos os pares de espécies, para cada árvore construída no passo de inserção de espécies. As matrizes de distância são base para extrair informações estatísticas sobre filogenia.
4. cálculo do índice estatístico “I de Moran”: o índice I de Moran, é um índice geral para medir auto-correlação. O cálculo desse índice é realizado sobre a matriz de distância, gerada na etapa anterior, calculando para cada uma dessas matrizes o índice de correção por classe de distância.

Como pode ser visto na figura 4.1, as etapas são dependentes, então o início da próxima etapa deve aguardar a conclusão da anterior. Além disso, nossa solução permite

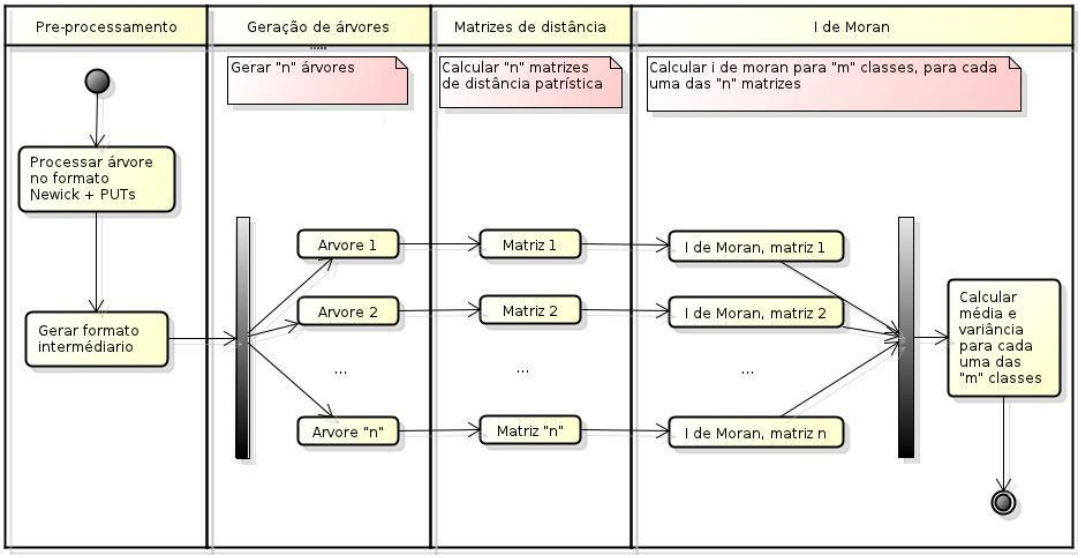


Figura 4.1: *Etapas da Implementação*

reaproveitar as estruturas de dados armazenadas na memória da GPU, i.e., os resultados obtidos em uma etapa permanecem em memória entre as chamadas dos *kernel* e são reaproveitados. Os processos descritos nestas etapas, assim como as estruturas de dados, os algoritmos e as otimizações utilizadas, estão detalhadas nas próximas seções.

4.1 Preprocessamento

Antes de realizar o processamento paralelo, a implementação tem que cuidar de algumas tarefas de preprocessamento, realizada sobre um conjunto de dados de entrada. Essa atividade serve de preparação dos dados de entrada em uma estrutura adequada para manipulação pelo sistema computacional em GPGPU. A entrada de dados é composta de:

- uma árvore filogenética, parcialmente conhecida e representada no formato Newick [41]. Normalmente construída à partir de dados moleculares.
- PUT (phylogenetically uncertain taxa), que são conjunto de espécies faltantes na árvore de entrada.
- *traits*, são características das espécies que se deseja comparar. São utilizados nos cálculos de correlação filogenética.

4.1.1 Etapas do preprocessamento e definição das estruturas de dados

Já vimos que a entrada de dados corresponde a uma árvore, PUTs e *trais*. A árvore é representada no formato Newick, enquanto as outras duas entradas na forma de arquivo texto comum, esse conjunto de dados deve ser lido para realização do *parsing*.

Em ciência da computação, o *parsing* é o processo de analisar uma sequência de entrada para, posteriormente, ser representado em outro formato requisitado por um programa. O *parsing* é responsável por carregar os arquivos de entrada e mapeá-los em uma estrutura de dados apropriada para computação paralela, que permita o uso de paralelismo de dados.

Em nossos exemplos, utilizaremos um modelo de árvore simplificada (figura 2.2(b)), que esta representada no formato Newick (2.2(a)). As figuras 2.2 e 4.2, mostram dados de exemplo fictícios, que vão ser utilizados para demonstrar o funcionamento da nossa solução. Optamos por uma árvore exemplo pequena, em detrimento de dados reais que poderiam sobrecarregar a visualização e compreensão das informações. Essa escolha, irá facilitar nossas explicações e exemplificação, deixando que nossa atenção fique voltada para as estruturas e algoritmos propostos.

O arquivo contendo as novas espécies, figura 4.2(a), é um arquivo texto com dois valores, separados pelo marcador de tabulação e por um marcador de final de linha. O primeiro valor é o nome da nova espécie a ser inserida e o segundo valor é o nome do ancestral que representa o ponto de inserção (MDCC). De forma equivalente, os *traits*, que podem vistos na figura 4.2(b), representam as características de cada espécies e serão utilizados na fase de cálculo do I de Moran. Essas informações estão gravadas em um arquivo texto, onde cada linha possui: o nome de uma espécie e o valor da características a ser analisada. O símbolo de tabulação é utilizado para separar esses dois dados.

Nos escrevemos um *parsing* em C++ que faz a leitura da árvore filogenética à partir do arquivo no formato Newick e converte para uma *representação interna*. O *parsing* faz uma leitura pós-ordem reversa e processa a árvore da forma como ela se apresenta no formato Newick. Por exemplo, para a ilustração da figura 2.2(a), a ordem seria A, B, C, D, E, F, G, H, I, J e K. Isso significa que os filhos sempre são lidos antes dos pais e, em particular, que a raiz será a última. Além disso, a árvore filogenética de entrada pode conter politomias (nós com mais de dois ramos), então o preprocessamento irá tratar essa incerteza filogenética, assegurando que a árvore resultante seja uma estrutura binária. Transformamos todas as politomias em dicotomias utilizando o mesmo processo de randomização utilizado para incluir as novas espécies na árvore filogenética de entrada. Esse processo é detalhado mais tarde, quando descrevemos a geração aleatória das árvores (ver seção 4.2).

Para favorecer o acesso aglutinado (*coalesced access*) a memória nos armazenamos as informações em uma estrutura de vetores (SoA - structure of vetor), onde cada vetor acomodará partes das informações sobre a árvore. Além dos nomes dos nós, comprimento dos ramos, seu antecessor (nó pai) e seus dois sucessores (nós filhos), também calculamos e armazenamos sua altura e o número de nós abaixo dele. Outra informação pertinente, e que armazenamos em outro *vetor*, são os dados referentes à característica da espécie que desejamos realizar a comparação. Além disso, asseguramos que as infor-

put	mdcc
X	K
Y	J

(a) *PUT (phylogenetically uncertain taxa)*

name	body
A	3.3
B	2.4
D	6.0
F	2.1
G	3.4
H	5.6
X	4.3
Y	5.5

(b) *Características das espécies*

Figura 4.2: *Formato dos arquivos de espécies faltantes (a) e características usadas na comparação (b).*

mações relacionadas a espécies e ancestrais (nós internos) estão facilmente acessíveis, armazenando informações sobre as espécies nos índices mais baixos do vetor e informações sobre os ancestrais nos índices mais altos 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. Também é utilizado um índice negativo para um ponteiro nulo, o que nos permite identificar facilmente se o índice refere-se a uma espécie (dados na metade inferior do SoA), um ancestral (dados na metade superior do SoA) ou para nulo (valores negativos).

Podemos ver, através da figura 4.3, que as especies 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).

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 4.3, posições 6 e 7), visualizada na estrutura como *novas espécies*. Posições correspondentes são reservadas para os novos ancestrais (Figura 4.3, 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.

	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 4.3: *Estrutura para representação dos dados*

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".

Ao representar as *novas espécies*, devemos interpretar de forma diferente os valores destes *vetores* . Por estarmos tratando com espécies perdidas/faltantes, não conhecemos o verdadeiro antecessor (*pai*) dessas espécies. Por consequência, não temos a informação do *comprimento do ramo*, que conecta espécie ao seu antecessor, pois o antecessor ainda é desconhecido. Assim, utilizamos o *vetor pai* para armazenar o nó MDCC dessa espécie, por exemplo, a espécie X, tem como MDCC o nó 15 (K), enquanto a espécie Y tem o nó 12 como MDCC (J). Essa informação será utilizada na etapa de **inserção de espécies/geração de árvores**, após essa etapa o nó pai será atualizado para o ancestral *escolhido* como antecessor da espécie. O comprimento do ramo será calculado, de forma semi-aleatória, nessa mesma etapa - ao final da inserção esses *vetores* conterão os valores esperados.

4.2 Inserção de espécies

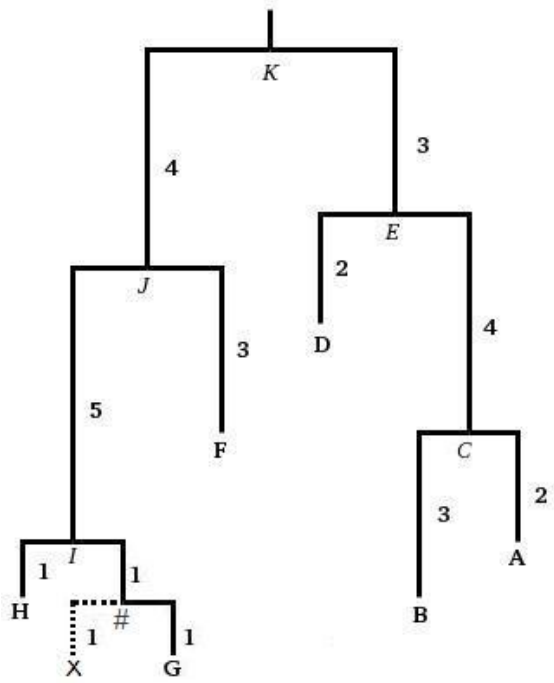
A inserção consiste em receber as estruturas de dados com as árvores e inserir em posições aleatórias da árvore, todas as espécies com incertezas filogenéticas. No entanto, temos conhecimento que os pontos de inserção não são totalmente aleatórios, pois existe um ancestral de consenso (MDCC, most derived consensus clade). À partir deste ponto, ou seja, dentro de uma posição descendente do ancestral de consenso (MDCC), será escolhida, de forma aleatória, um local para *pendurarmos* esta espécie.

O *Kernel* CUDA foi construído para gerar as **árvores aleatórias**, que derivam de uma árvore de entrada e considerando as novas espécies (PUT). Inicialmente, a árvore de entrada é replicada o número de vezes que for necessário. Em seguida, utilizamos diversos blocos de *threads* e, para cada *thread* será associada uma árvore. Cada *thread* é responsável por incluir as espécies faltantes na árvore, sorteando uma posição à partir do ponto identificado como MDCC. As simulações construídas chegam a gerar milhares de árvores de forma concorrente.

A inserção de novas espécies foi codificada utilizando uma representação binária para caminhar pelas árvores. Inicialmente geramos um número aleatório entre 0 (zero) e o maior valor permitido para um inteiro sem sinal (32 bits). A representação binária deste número é utilizada para percorrer a árvore até encontrar o ponto de inserção escolhido. Começamos a partir do bit mais significativo e caminhamos para baixo na árvore, considerando cada bit do número. A direção é dada pelo valor do bit, para o bit 1 seguiremos o caminho para o nó descendente à esquerda (filho esquerda), e o bit 0, diz que o caminho a seguir é o descendente a direita (filho à direita). Um outro número aleatório é utilizado para limitar a profundidade que deve ser percorrida. No caso do percurso atingir uma espécie (nó folha) antes do procedimento descrito concluir, então o último ancestral será utilizado como ponto de inserção.

Por exemplo, considere a árvore da figura 2.2 e o nó *K* como o ponto de inserção (MDCC). Assuma que uma nova espécie *X* precisa ser inserida na árvore e que somente 4 bits são utilizados para caminhar pela árvore. Caso os dois números randômicos gerados sejam 12 (1100_2) e 2, então iriamos descer na árvore duas vezes caminhando pela esquerda e paramos, ou seja, a nova espécie será inserida abaixo do nó "I". Visto que o próximo bit é 0 (1100_2), criaremos um novo nó antecessor à direita de I, mantendo G a direita e inserindo a nova espécie *X* a esquerda. A árvore resultante e as estruturas atualizadas são exibidas na figura 4.4.

Uma questão importante ao inserir uma nova espécie é como determinar o comprimento do ramo (*branch*) para essa nova espécie(X) e seu antecessor. A regra que deve ser respeitada é manter a distância original entre espécies e ancestrais inalteradas. Deste modo, o comprimento do ramo para o novo ancestral é definido como um percentual do comprimento do ramo do ancestral escolhido como ponto de inserção (em nosso exemplo, nó **I**) e o percentual restante é definido para o comprimento do ramo para a nova espécie ou grupo abaixo do novo ancestral. Para a nova espécie, o tamanho do ramo é definido para o mesmo tamanho da sua espécie irmã. No entanto, se não for uma espécie e sim um clado (grupo de organimos originados de um ancestral comum), o comprimento do ramo para a nova espécie deve ser calculado com um procedimento similar ao descrito anteriormente para determinar o ponto de inserção. Isto é, um número aleatório é gerado e os bits são utilizados para percorrer um caminho do clado. Iniciaremos a partir do nó



(a) *Árvore após a inserção de um novo nó*

	espécies iniciais						novas espécies			novos ancestrais		ancestrais iniciais				
indice	*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	*10*	11	12	*10*	*12*	-	*?*	*11*	12	15	14	15	-1
filho esquerda	-2	-2	-2	-2	-2	-2	*-2*	*-2*	-	*?*	*6*	3	11	0	13	14
filho direita	-2	-2	-2	-2	-2	-2	*-2*	*-2*	-	*?*	*3*	4	5	1	2	12
comprimento do ramo	2	3	2	*1*	1	3	*1*	*?*	-	*?*	*1*	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

(b) *Estrutura de dados atualizada após a inserção de um novo nó*

Figura 4.4: *Inserção de um nó (espécie).*

raiz do clado e continuaremos até uma espécie seja alcançada. O comprimento do ramo da nova espécie é a soma acumulada das distâncias percorridas. A figura 4.4 ilustra a situação onde o percentual utilizado para o novo comprimento do ramo ancestral é 50% e a irmã da nova espécie inserida é uma espécie (G) e não um clado. Observe na figura 4.4(b) o destaque nas posições que sofreram atualização de informações.

O algoritmo 4.1 é executado por cada *thread*. Note que antes de iniciarmos a geração das árvores, cada *thread* realiza uma cópia da árvore original, junto com as novas espécies a serem inseridas, para uma área da memória global que será de acesso exclusivo para essa *thread*. Dessa forma, cada *thread* irá gerar sua própria árvore e as árvores resultantes são mantidas na memória global para uso do próximo *kernel*.

Algoritmo 4.1: *kernelInserirEspecies*(*SoA*)

Entrada: Struct of Array (SoA) construídas no pré-processamento.
Saída: SoA atualizadas.

```
1  for todas as threads do in parallel
2      faça uma copia da árvore original para area de acesso exclusivo
3      while houver espécies a serem inseridas do
4          definir, randomicamente, a posição para inserir a nova espécie
5          inserir a nova espécie e o novo ancestral
6          if irmã da nova espécie também é uma espécie then
7              definir o comprimento igual ao da sua espécie irmã
8          else
9              examinar o clado irmão até chegar a uma de suas espécies
10             utilizar a distância acumulada para definir o comprimento do
               ramo
11         end
12     end
13 end
```

4.3 Cálculo da matriz de distância

Sejam n espécies em uma árvore, então a matriz de distância patrística possui tamanho $n \times n$, e descreve a similaridade filogenética entre todos os pares de espécies. Essa matriz é simétrica, então, precisaremos calcular somente a metade dos elementos da matriz, por exemplo a metade superior. A diagonal é trivial, com todos valores iguais a zero. Contudo, para calcular uma única distância entre um par de espécies, nós precisamos somar todos os comprimentos de ramos que conectam essas duas espécies. Um caminho para realizar este cálculo é, primeiramente, encontrar o ancestral comum mais *baixo* (lowest commom ancestor - LCA), ou seja, o nó mais próximo na árvore que tem as duas espécies como descendente. Então a distância entre as espécies é dada pela soma das distâncias de cada espécie até a raiz, subtraindo duas vezes a distância do LCA até a raiz. A figura 4.5 ilustra essa situação para as espécies H e F. O LCA é J, cuja a distância até a raiz é 4. A distância de H e F até a raiz é 10 e 7, respectivamente. Assim, calculamos a distância entre H e F como $10+7-8$ (o valor 8 foi obtido multiplicando-se a distância do LCA, J, por dois), cujo resultado é 9.

Logo, para realizar o cálculo da matriz de distância filogenética, precisaremos encontrar o LCA (lowest common ancestor) entre cada par de espécies. Conforme descrevemos anteriormente, essa operação pode ser facilmente implementada. Encontramos

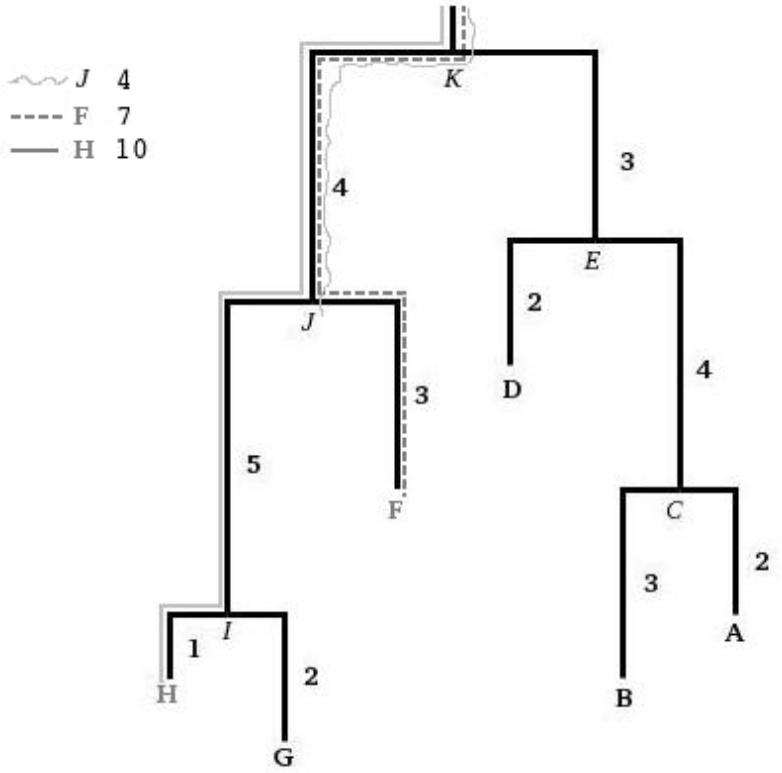


Figura 4.5: *Cálculo da distância*

alguns algoritmos propostos na literatura para calcular o LCA, incluindo alguns algoritmos paralelos [3, 5, 48]. Entretanto, não conhecemos nenhuma implementação paralela do LCA para GPUs. Sendo assim, propomos um método que trabalha em dois passos. Primeiro, nós calculamos a distância e o *caminho* de todos os nós (espécies e ancestrais) até a raiz da árvore. Isto é realizado através do caminhamento até a raiz da árvore, seguindo o ramo que interliga a *espécie* ao seu antecessor (link ascendente) e acumulando o comprimento de cada ramo à medida que sobimos a árvore. Durante esse processo, iremos manter o caminho trilhado conforme a direção que percorremos (direita ou esquerda) em formato númerico e codificado em binário. Este *caminho númerico* será utilizado como chave, junto ao valor da distância acumulada, numa tabela *hash*. Através da comparação de dois caminhos binários é possível encontrar o caminho do LCA e, conseguentemente, o próprio LCA e a distância até a raiz.

Para ilustrar esse processo, considere novamente a espécie H e F na figura 4.5. O caminho binário para H e F é 1111 e 101, respectivamente. Assumimos que 0 indica que viemos a partir do ramo da direita e 1 em caso contrário. Além disso, um bit extra "1", indicando a raiz, é utilizado como prefixo para o *caminho binário*. Dessa maneira, o *caminho binário* para a espécie F é definido inicialmente com 1 (prefixo) e, a medida que caminhamos para cima na árvore, os bits são deslocados para a esquerda e bits adicionais são incluídos, 0 para a direita e 1 para a esquerda, resultando em 101. Para encontrar qual o ancestral comum mais baixo (LCA) entre H e F, podemos comparar

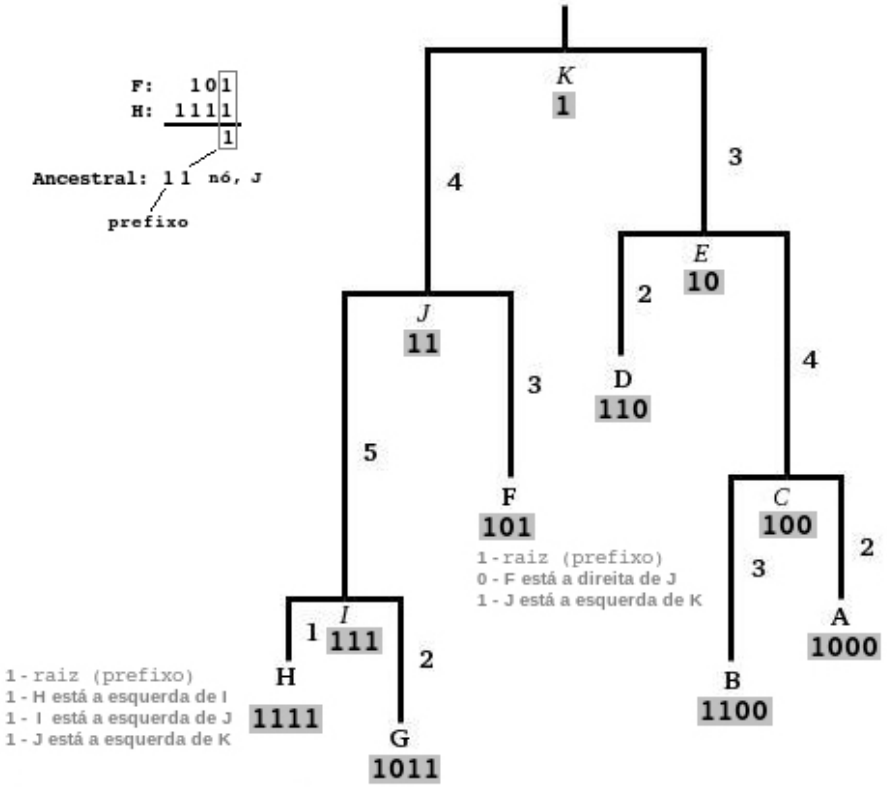


Figura 4.6: *Árvore exemplo: caminhos binários (chaves)*

os dois caminhos binários, 1111 and 101, iniciando a partir do bit menos significativo e parando quando os bits divergirem. O caminho binário do LCA é igual à 1 (prefixo) seguido dos bits coincidentes, neste exemplo identificamos apenas um bit coincidente, de valor "1" (ver figura 4.6). Esse resultado será o caminho binário 11, o qual é a chave da tabela *hash*, que nós permite acessar o valor da distância do ancestral *J*. Sendo assim, conhecendo o caminho binário de dois nós, é possível encontrar o LCA entre eles através de uma simples operação de comparação *bit-a-bit*, como pode ser visto na figura 4.6, canto superior esquerdo.

A tabela 4.1 mostra os caminhos binários (chaves ou *keys*) e as distâncias (valores) para todos nós da árvore. Essa tabela foi implementada na forma de tabela *hash*, na qual consultas podem ser processadas rapidamente. Optamos por implementar o método *open addressing* (endereçamento aberto), visto que conhecemos o número de entradas a ser armazenadas, que é duas vezes o número de espécies menos 1. Então podemos fazer uso de posições vazias na tabela para resolver problemas de colisões, i.e., nesse método todas as chaves são armazenadas na própria tabela, inclusive as que sofreram colisões, evitando assim o uso de apontadores explícitos. Quando houver colisões, elas serão tratadas através o uso da técnica *quadratic probing*, na qual o índice correspondente é incrementado por um polinômio quadrático, até que uma posição disponível seja encontrada. Asseguramos que o tamanho da tabela *hash* seja um número

Tabela 4.1: *Árvore exemplo: distâncias dos nós*

Nó	Chave	Distância
K	1	0
J	11	4
E	10	3
I	111	9
F	101	7
D	110	5
C	100	7
H	1111	10
G	1011	11
B	1100	10
A	1000	9

primo superior a duas vezes o números de espécies e consideramos que o fator de preenchimento dessa tabela será menor que 50%. Todo o processo para gerar a matriz de distância patrística foi implementado com o *kernel* do algoritmo 4.2.

Agora, vamos explicar, passo-a-passo, como o método apresentado no algoritmo 4.2 foi implementado em paralelo dentro da GPU. Visto que a distância entre todos os pares de espécies necessitam ser calculadas para cada árvore, nos atribuímos cada árvore filogenética a um bloco de *threads* e deixamos as *threads* dentro do bloco cooperarem para calcular as distâncias da matriz. Isso nos permite explorar um nível adicional de paralelismo, com muitas *threads* trabalhando numa única árvore. Nesse ponto tentamos obter ganhos de desempenho trazendo a árvore para a memória compartilhada, entretanto o limite de capacidade dessa memória impede a alocação necessária para armazenarmos nossas estruturas de dados e, por consequência, não obtivemos bons resultados. Sendo assim, voltamos a ideia inicial, com utilização da memória global. Devido nossa solução utilizar grande quantidade de *threads* trabalhando simultaneamente, podemos ocultar a latência ao buscar dados na memória global.

Parte do trabalho para calcular a matriz de distância é definir a tabela *hash*, que representa a distância de cada nó até a raiz. A tabela exibida na figura 4.5 o resultado obtido com essa etapa. Para computar esses valroes, nós associamos cada espécie e cada ancestral para uma *thread*. A medida que as *threads* seguem para o link do ancestral, elas acumulam a distância até chegar a raiz e constroem o caminho binário correspondente [linhas 2 a 6]. Este cálculo foi implementado em duas fases. Na primeira as *threads* trabalham com as espécies, que estão na primeira metade do *vetor*. Em seguida, depois da barreira de sincronização, os ancestrais (nós internos) são processados. Todos os nós (espécies e ancestrais) poderiam ser processados simultaneamente, mas isto iria requisitar mais *threads* por bloco, o que poderia limitar ainda mais o tamanho da árvore ao limites de *threads* por bloco (atualmente 512 nos modelos G80 e 1024 na arquitetura

Fermi). Dependendo do tamanho da árvore, poderíamos ajustar um grupo de *threads* para trabalhar sobre partes dos nós da árvore e programar a realização de um *rodízio* para processar todas as espécies.

Algoritmo 4.2: *kernelCalcularMatrizPatristica*(*SoA*)

Entrada: Struct of Array (SoA)

Saída: matriz de distância, mapeada como vetor unidimensional.

```
1  for todas as threads do in parallel
2      associar a thread com uma espécie ou ancestral
3      while nó pai não for a raiz do
4          acumular a distância e o caminho percorrido (na forma binária)
5          seguir para o nó pai
6      end
7      armazenar a distância e o caminho binário em uma tabela hash
8      sync_threads (barreira de sincronização )
9      associar uma thread com um elemento (espécie) do vetor
10     for k ← 1 to (quantidade de espécies/ 2) do
11         associar um elemento do vetor para espécie da matriz
12         consultar na tabela hash a distância das espécies até a raiz
13         comparar os caminhos binários do par de espécies
14         encontrar o LCA e consultar na tabela hash a distância do mesmo
           até a raiz
15         calcular a distância entre o par de espécies
16         distância do 1º + distancia do 2º - 2 vezes a distância do LCA
17         armazenar a distância em um vetor que representa a matriz
18     end
19 end
```

Após o armazenamento dos caminhos binários e as distâncias correspondentes até a raiz em uma tabela *hash* (linhas 7 e 8), a matriz de distância filogenética pode, finalmente, ser calculada. Como já mencionado, a matriz é simétrica, então precisaremos calcular somente a metade dos elementos da matriz. Entretanto, as linhas tem diferentes números de elementos e atribuir cada linha a uma *thread* será ineficiente. Para balancear a carga, os elementos que estão acima da diagonal principal da matriz foram armazenados fisicamente em um vetor. O comprimento desse vetor foi definido conforme o número de espécies, através da seguinte fórmula: $(n*(n-1)/2)$, seja n a quantidade de espécies na filogenia. Então, para obter o balanceamento de carga pretendido, atribuímos a cada *thread* o mesmo número de elementos, adotamos o número de espécies dividido por

dois [linhas 9-10]. Uma função é utilizada para mapear o índice do vetor para os correspondentes índices (linhas e coluna) da matriz (linha 11). A figura 4.7 ilustra este esquema para uma árvore contendo oito espécies. Por meio dos índices (linhas e coluna) da matriz, agora a *thread* pode consultar a tabela *hash* para descobrir as distâncias das espécies até a raiz e os caminhos binários correspondentes (chaves). Os caminhos binários são comparados para determinar o LCA das duas espécies. Então, o caminho binário do LCA é utilizado para consultar a tabela *hash* e recuperar a distância até a raiz. Finalmente, a distância entre as duas espécies é calculada e o resultado armazenado dentro do vetor que representa a matriz [linhas 13-19].

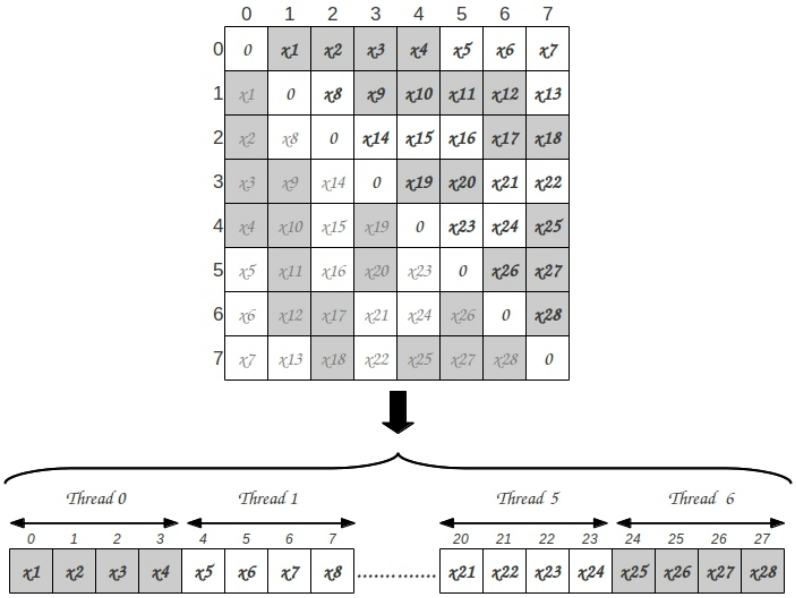


Figura 4.7: *Mapeamento da matriz de distância para threads*

4.4 Cálculo do I de Moran

Normalmente, o I de Moran é aplicado sobre classes de distância no tempo, ou seja, iremos comparar uma determinada característica entre espécies que estão a uma mesma faixa de distância no tempo. Assim, os biólogos definem as classes de distância, de forma arbitrária, conforme seus interesses nos grupos de análise. Além disso, são definidas poucas faixas de distâncias, aproximadamente $\log_2 n$ (seja "n" a quantidade de espécies).

O algoritmo paralelo proposto aloca uma árvore para cada bloco, deixando o trabalho de calcular o *I de Moran* distribuído entre todas as *threads* do bloco. Para cada bloco, são lançadas tantas *threads*, quanto forem o total de espécies. Cada *thread* do bloco trabalha em uma área exclusiva de valores da matriz de distância patrística (a divisão é realizada conforme o modelo da figura 4.7). Então, o *kernel* é lançado uma única vez,

gerando *b* blocos com *t threads*, o que poderá alcançar milhões de *threads*, conforme o tamanho da árvore.

Primeiramente, é realizado o cáclulo para obter a média das características. Em seguida, as *threads* verificam se as distâncias pertencem a primeira classe e, em caso afirmativo, acumulam o produto da diferença entre a característica de cada espécie e a média $((y_i - \bar{y}) \cdot (y_j - \bar{y}))$. Quando cada *thread* concluir sua parte, uma área da matriz de distância foi verificada, este valor é acumulado, atomicamente, em um variável compartilhada. A atomicidade é necessária, pois várias *threads* podem concorrer por esse variável compartilhada. Na sequência, as *threads* que concluíram sua área de verificação, encontram uma *barreira de sincronização* que as forçam a aguardar as outras. No ponto que todas concluírem, então podem calcular o *I de Moran* para aquela classe de distância. Esse processo se repete paras as próximas classes de distância, conforme podem pode ser visto no algoritmo 4.3.

Ao final, cada classe terá seu *Índice de Moran* calculado, porém cada árvore gera seu próprio conjunto de *Índice de Moran*, ou seja, teremos centenas ou milhares de árvores e, consequentemente, de índices calculados. Esses valores são mantido sem uma estrutura de vetor. Então, o próximo passo será calcular a média do *Índice de Moran*, por classe, e em seguida, também devemos calcular a variância como forma de avaliar a dispersão entre valores encontrados para cada árvore. Optamos por calcular a média e a variância sequencialmente, devido ao baixo volume de dados e exigência reduzida de operações matemáticas necessárias para utilização nessa estrutura.

O armazenamento das variáveis de *soma* e do *vetor das classes de distância* na memória compartilhada, além de *compartilhar* os valores entre todas as *threads*, também permite conseguir acesso rápido a estas informações, reduzindo a latência de memória gerada ao acessar a memória global. Por se tratar de variáveis com pouca quantidade de dados, as mesmas não sofrem restrições quanto à capacidade reduzida da memória compartilhada.

Pudemos observar que o uso de operações atômicas degradam o desempenho total do *kernel*, no entanto são operações essenciais para garantir confiabilidade e veracidade ao valor dessas variáveis. Verificamos outras soluções para amenizar o número de operações atômicas, uma delas foi agrupar as *threads* por classe de distância e, por consequência, ampliar a faixa de valores de exclusividade de cada *thread*, reduzindo assim o número de operações atômicas. Entretanto, conforme a quantidade de classes, algumas *threads* do último bloco podem ficar paradas, pois não é possível utilizar divisão exata do trabalho entre as *threads*. Além disso, com uma área maior para responsabilidade da *thread*, reduzimos o paralelismo.

Algoritmo 4.3: *kernelCalcularIdeMoran*(*distance*, *traits*)

Entrada: Matriz de distância patristica
Vetor com características de cada espécie
Vetor de classes de distância

Saída: Média e variância dos Índice de Moran por classe

```
1  armazenar o vetor com as classes de distância na memória compartilhada
2  criar e inicializar variáveis compartilhadas, utilizada nas somas parciais
3  calcular a media das características
4  for todas as threads do in parallel
5      for classe ← 1 to (número de classes) do
6          for k ← 1 to (quantidade de espécies / 2) do
7              if distância entre as espécies pertence a classe then
8                  associar o elemento do vetor de distância as espécies da
                   matriz
9                  calcular e acumular o produto entre a diferença das espécies
                   e a média
10             end
11         end
12         realiza, atomicamente, o somatório para calcular o I de Moran
           (numerador da fórmula 2-1)
13         sync_threads (barreira de sincronização)
14         apenas uma thread calcula o I de Moran para a classe
15         inicializar variáveis compartilhadas
16         sync_threads (barreira de sincronização)
17     end
18 end
```

CAPÍTULO 5

Análise de Resultados

Além do algoritmo paralelo descrito no capítulo 4, também foi desenvolvida uma versão sequencial desse mesmo algoritmo. Iremos utilizar essa versão sequencial em nossos testes para fim de comparação e análise de desempenho. Também avaliaremos a execução do programa **phylocom** [55], que é utilizado para calcular a matriz patrística. O programa **phylocom** não realiza todas as etapas descritas em nossa implementação, ele calcula somente a matriz patrística, então iremos utiliza-lo para comparação apenas da matriz de distância. O programa Phylocom calcula a matriz de distância uma única vez, mas precisamos que seja realizadas várias vezes, conforme o número de simulações necessárias. Então, modificamos o código do Phylocom para que se adequasse a esta necessidade.

Dividimos essa capítulo duas partes. Na primeira, apresentamos os resultados obtidos para calcular a matriz de distância sobre as quatro filogenias disponíveis (*DummysA, DummysB, Carnivores e Hummingbirds*. Na segunda parte desse capítulo, analisaremos os resultados obtidos para executar *todas as etapas*. Entretanto, não encontramos um programa que realiza todas as etapas do nosso programa, principalmente no módulo que trata as incertezas (PUTs), então faremos um quadro comparativo as versões sequenciais e paralela do nosso algoritmo. Além disso, a filogenia *Carnivores* não será utilizada, pois não possui PUTs. Por fim, iremos analisar os resultados e verificar *speedup up*. Assim, na primeira parte utilizaremos análise de três algoritmos: *phylocom*, solução serial e solução paralela. Já na segunda etapa, a solução serial e paralela são utilizadas para medir desempenho de todas as etapas dos cálculos (ver figura 4.1 para saber quais as etapas).

Quatro filogenias foram disponibilizadas para testes e simulações. Duas delas, foram construídas artificialmente, somente para uso em nosso trabalho, de forma a realizar testes e medir desempenho. Essas filogenias artificiais são identificadas no texto como *DummyA* e *DummyB*. A seguir temos uma breve descrição dessas filogenias:

- **DummysA**: Grupo de espécies *artificiais*, em uma filogenia completamente balanceada. O *trait* também foi criado artificialmente. São 160 espécies no total, sendo 32 delas PUTs.

- **Carnivores**: Filogenia dos carnívoros. A característica (ou *trait*) disponível para análise de auto-correlação é o tamanho de corpo. Essa filogenia contém 209 espécies, porém nenhuma é perdida/desconhecida, ou seja, não há PUT's a serem inseridos.
- **Hummingbirds**: contém a filogenia para 146 espécies de beija-flores e mais 158 PUT's a serem adicionados a filogenia, totalizando 304 espécies. A característica disponível é o tamanho do corpo.
- **DummysB**: Grupo de espécies *artificiais*, em uma filogenia completamente balanceada. O *trait* também foi criado artificialmente. São 400 espécies no total, sendo 272 delas PUTs.

Todos os experimentos realizados, nos três programas, foram executados no mínimo 10 vezes, então extraímos a média e calculamos o desvio padrão para construirmos as tabelas, e gráficos realizar as análises de desempenho. As simulações foram realizadas para gerar 128, 1024 e 8192 árvores filogenéticas em memória.

Os experimentos fora conduzindo utilizando um Intel Core2Duo 1.6GHz, 2 GB RAM, NVIDIA Tesla C1060, com sistema operacional Linux (Ubuntu 11.04 de 32 bits). A Tesla consiste em um conjunto de 30 processadores de *streaming* (SMs), cada um deles possui 8 núcleos de processamento (SPs) com velocidade igual a 1.4 GHz, totalizando 240 núcleos. A Tesla ainda possui 4 GB de memória global do tipo GDDR e 16 KB de memória compartilhada. Nossos programas foram implementados em C/C++ e CUDA C/C++ 4.0.

5.1 Cálculo da matriz de distância

Nesta seção analisaremos o desempenho dos programas *Phylocom*, solução serial e solução paralela para realizar o calculo da matriz de distância. As tabelas 5.1, 5.2 e 5.3 mostram os resultados alcançados por cada programa e para as filogenias disponíveis. Os valores numéricos nas células das tabelas, correspondem ao tempo médio gasto para calcular a matriz de distância e os valores fracionários dentro dos parênteses correspondem ao desvio padrão. A unidade de medida utilizada é o **segundos**. Quando o *desvio padrão* for inferior a *centenas de segundos* (3 ou mais casas decimais) iremos omiti-los, por questões de otimização de espaço.

O gráfico da figura 5.1 mostra o ganho de desempenho (speedup) para calcular a matriz de distância. Como pode ser observado no gráfico 5.1(c), conseguimos atingir um *speedup* aproximado de $225x$, se comparado ao programa Phylocom, sobre a filogenia dos *carnívoros*, para gerar 8.192 árvores.

Já com relação a nossa solução sequencial, tivemos *speedup* máximo de 204 vezes, quando geramos as mesmas 8.192 árvores, mas desta vez para a filogenia artificial

DummyB. Entretanto, os ganhos com relação a solução sequencial ficam entre 60 até 100 vezes. A justificativa para esse *pico do speedup* se deve ao fato da CPU ter que fazer paginação, causando queda no desempenho da execução do programa sequencial. O mesmo não foi identificado com o Phylocom devido o mesmo não manter as cópias em memória. No *Phylocom*, ao calcular uma nova matriz de distância, sobrepomos o valor do calculo anterior. Enquanto que a solução sequencial e paralela mantem todas as cópias em memória.

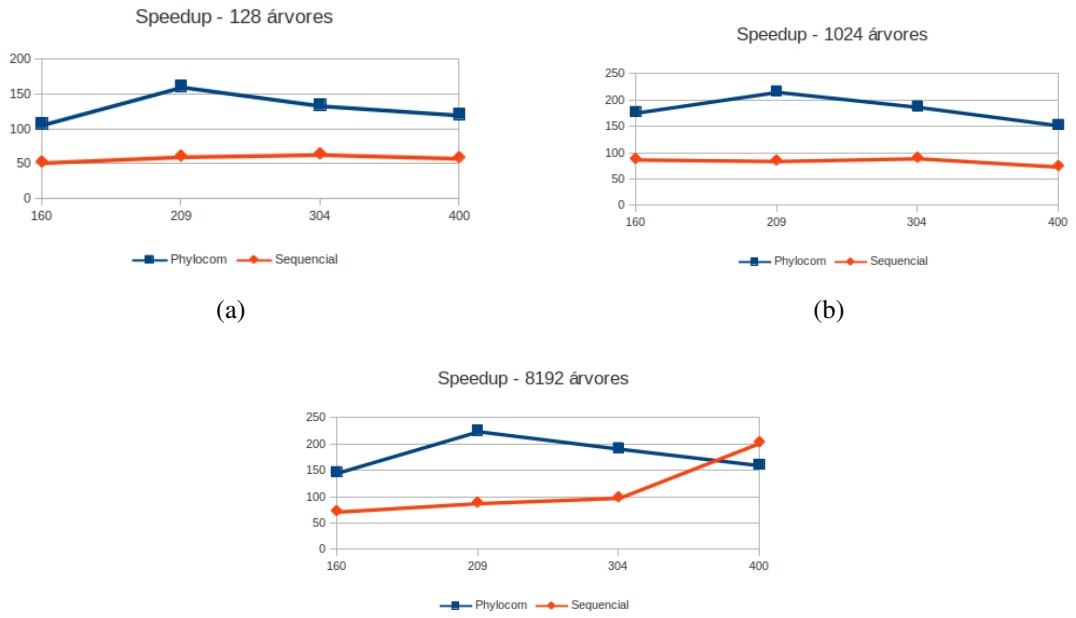


Figura 5.1: *Speedup alcançados com o calculo da matriz de distância (incluso tempo de cópia dos dados para a GPU).*

5.2 Cálculo para todas as etapas

Agora, iremos avaliar a execução de todas as fases do programa. No entanto, utilizamos apenas as soluções sequencial e paralela, pois o *Phylocom* não realiza o calculo do índice de Moran e, também, não realiza simulações sobre incertezas filogenéticas. A solução sequencial realiza as mesmas etapas do programa paralelo, entretanto, executando sobre a CPU. Isso o permite fazer uso das técnicas de previsão de desvio, cache, pipeline e outras disponíveis nessa arquitetura. É importante lembrar que o paralelo realizada uma etapa a mais que o programa sequencial, que é a transferência dos dados para a memória da GPU. Na solução sequencial essa etapa é desnecessária, já que os dados estão disponibilizados na memória *do host*.

O processo de cópia/transferência dos dados da memória principal (RAM) para a GPU é realizado da seguinte forma: uma cópia da árvore e enviada várias vezes para a GPU, árvore-por-árvore, ou seja, primeiro pré-alocamos todo os espaço necessário na GPU, em seguida, preenchemos com os dados da filogenia. Outras alternativas foram utilizadas e estão disponíveis no programa, como o envio de uma única cópia da filogenia para o *device* e a própria GPU se encarrega de replica-la em memória. A replicação poderia ser realizada direto na memória global ou com uso da memória compartilhada. Para gerar *n* árvores, utilizamos *n threads*. Na primeira opção, cada *thread* copia um elemento da árvore original, acessando a memória global, e depois copia o dado para outra área de memória correspondente a outra árvore. A segunda solução, utilizando a memória compartilhada, fazemos uma cópia da árvore para essa memória e, em seguida, cada *thread* do bloco lê os dados à partir da memória compartilhada e replica para a posição reservada na memória global, com isso reduzimos o número de acesso a memória global. Ambas estratégias tiveram desempenho superior a primeira, com resultados expressivos proporcionalmente, entre 20% e 30% de melhoria. Entretanto os valores são pouco expressivos, sendo que, o tempo para cópia na primeira estratégia esta na casa dos centésimos de segundos.

Nas tabelas 5.4, 5.5 e 5.6 mostramos detalhadamente os resultados alcançados em cada etapa de execução. Vale salientar que a coluna **cópia** corresponde a soma do tempo de transferência dos dados da CPU para a GPU com tempo gasto para alocação da memória da GPU. Da mesma maneira que adotamos para as tabelas da seção anterior, também omitiremos o desvio padrão quando for inferior a *centenas de segundos* (3 ou mais casas decimais). Além disso, o desvio padrão para a tabela com 8.192 espécies foi registrado separadamente, na tabela 5.7, devido ao volume de informações a serem exibidas.

Quando observamos as tabela 5.4, 5.5 e 5.6, podemos verificar que o processo de **Inserir espécies** não obteve ganhos, inclusive com valores inferiores ao sequencial. Atribuímos esses fraco desempenho a dois fatores: baixo nível de paralelismo e acesso não-aglutinado a memória. Primeiramente, o paralelismo aplicado para realizar as inserções das espécies faltantes ocorre apenas em nível de árvore, ou seja, cada *thread* é responsável por adicionar todas as espécies em uma determinada árvore. O processo de inserção é sequencial para cada árvore.

Além disso, o segundo e principal fator que influenciou no baixo desempenho foi o uso da memória global. O acesso a esta memória é lento e necessita de um padrão de acesso a memória que favoreça a aglutinação. Entretanto, nossa solução utiliza acesso não aglutinado, onde cada *thread* trabalha em uma faixa diferente dos dados. Para visualizar esse tipo de acesso, é necessário relembrar como os dados estão disponíveis na memória. Vimos que, todas as árvores estão na memória na forma de um único vetor (ver figura 4.3),

Tabela 5.1: *Tempos para gerar matriz de distância* - 128 *árvores*

Espécies	**Phylocom**	**Sequencial**	**Paralelo**	
			Execução	**Cópia**
DummyA (160)	0,64 (0,05)	0,31 (0,01)	0,01	0,00
Carnivores (209)	1,42 (0,06)	0,57 (0,04)	0,01	0,00
Hummingbirds (304)	2,86 (0,07)	1,41 (0,09)	0,02	0,00
DummyB (400)	4,63 (0,07)	2,25 (0,03)	0,04	0,00

Tabela 5.2: *Tempos para gerar matriz de distância* - 1.024 *árvores*

Espécies	**Phylocom**	**Sequencial**	**Paralelo**	
			Execução	**Cópia**
DummyA (160)	4,85 (0,01)	2,43 (0,02)	0,04	0,01
Carnivores (209)	10,95 (0,03)	4,35 (0,01)	0,05	0,00
Hummingbirds (304)	22,40 (0,07)	10,85 (0,61)	0,12	0,01
DummyB (400)	36,86 (0,05)	18,09 (0,67)	0,25	0,01

Tabela 5.3: *Tempos para gerar matriz de distância* - 8.192 *árvores*

Espécies	**Phylocom**	**Sequencial**	**Paralelo**	
			Execução	**Cópia**
DummyA (160)	38,83 (0,02)	19,39 (0,03)	0,26	0,02
Carnivores (209)	87,47 (0,06)	34,71 (0,10)	0,41	0,03
Hummingbirds (304)	178,83 (0,07)	92,49 (2,79)	0,93	0,05
DummyB (400)	294,60 (0,10)	372,83 (28,04)	1,82 (0,02)	0,07

Tabela 5.4: *Tempo de execução para* 128 *árvores*

Espécies	**Sequêncial**			**Paralelo**			
	Inserir	Matriz	I Moran	Inserir	Matriz	I Moran	Cópia
160	0,00	0,31(0,01)	0,34	0,01	0,01	0,01	0,00
304	0,01	1,41(0,08)	1,25(0,07)	0,02	0,02	0,02	0,00
400	0,02	2,25(0,03)	2,25(0,06)	0,03	0,04	0,06	0,00

Tabela 5.5: *Tempo de execução para* 1.024 *árvores*

Espécies	**Sequêncial**			**Paralelo**			
	Inserir	Matriz	I Moran	Inserir	Matriz	I Moran	Cópia
160	0,02	2,43(0,02)	2,70(0,01)	0,14	0,04	0,04	0,01
304	0,08	10,85(0,61)	10,08(0,82)	0,17	0,12	0,14	0,01
400	0,12	18,09(0,67)	18,03(1,11)	0,20	0,24	0,31(0,08)	0,01

e cada uma delas é armazenada sequencialmente nessa estrutura. Assim, considerando n a quantidade de espécies, quando a primeira *thread* de uma *warp* estiver acessando a posição 1 de um vetor de n posições, então a segunda *thread* da mesma *warp* irá acessar outra espécie na posição $n+1$ do vetor e assim sucessivamente. Essa forma de acesso não favorece a aglutinação e, por consequência, não consegue evitar a latência de acesso a memória global. O que explica o fraco desempenho da solução paralela, nessa etapa do programa.

Também construímos uma solução onde todas as *threads* do bloco são associadas a uma espécies perdida (PUT). Durante a inserção as *threads* deviam utilizar operações atômicas, impedindo que outra mudança seja realizada simultaneamente em um mesmo nó. Essas operações degradavam o desempenho, tornando os resultados inferiores ao programa sequencial. O uso da memória compartilhada também foi considerado como opção, mas se tornou proibitivo devido à sua capacidade, que limitava em demasia o tamanho da filogenia.

Tabela 5.6: *Tempo de execução para* 8.192 *árvores*

Espécies	**Sequêncial**			**Paralelo**			
	Inserir	Matriz	I Moran	Inserir	Matriz	I Moran	Cópia
160	0,14	19,39	21,58	0,33	0,26	0,39	0,02
304	0,66	92,49	78,90	0,78	0,93	1,08	0,05
400	0,95	372,83	305,60	1,02	1,82	2,18	0,07

Tabela 5.7: *Desvio padrão para os tempos da tabela 5.6*

Espécies	**Sequêncial**			**Paralelo**			
	Inserir	Matriz	I Moran	Inserir	Matriz	I Moran	Cópia
160	(0,00)	(0,03)	(0,02)	(0,00)	(0,00)	(0,01)	(0,00)
304	(0,04)	(2,79)	(4,67)	(0,00)	(0,00)	(0,04)	(0,00)
400	(0,07)	(28,04)	(33,86)	(0,00)	(0,02)	(0,02)	(0,00)

Os gráficos da figura 5.2, mostram o quanto ganhamos em performance ao utilizar a solução paralela. Os gráficos exibem a relação *speedup* versus o número de espécies. Um fato curioso, ou melhor uma anomalia, pode ser observado no gráfico 5.2(c), no qual o *speedup* sobre a filogenia *DummyB* atinge *speedup* $133x$. Valor que chega a superar, em até em $5x$, os ganhos atingidos com outas filogenias. Acreditamos que se trata de um caso isolado, ocorrido devido o uso intenso da paginação. Para este caso, o uso da paginação foi necessária devido ao tamanho limitado da memória principal e ao alto número de espécies na filogenia. Se desconsiderarmos esse *speedup*, devido a anomalia, então a solução paralela possui *speedup* máximo de $60x$, quando aplicado sobre a filogenia dos *hummingbirds*, na figura 5.2(b).

É possível perceber nos gráficos, que a medida que a quantidade de árvores a serem simuladas vão aumentando, os ganhos de performance vão melhorando. Para a filogenia dos *beija-flores*, ou *hummingbirds*, os *speedups* foram de 39, 47 e 59 quando utilizados para gerar 128, 1024 e 8192 árvores, respectivamente. Podemos considerar que a medida que o número de árvores aumentam, melhores são os ganhos. Além disso, para filogenias maiores (com mais espécies), os ganhos são mais significativos se trabalharmos com mais simulações.

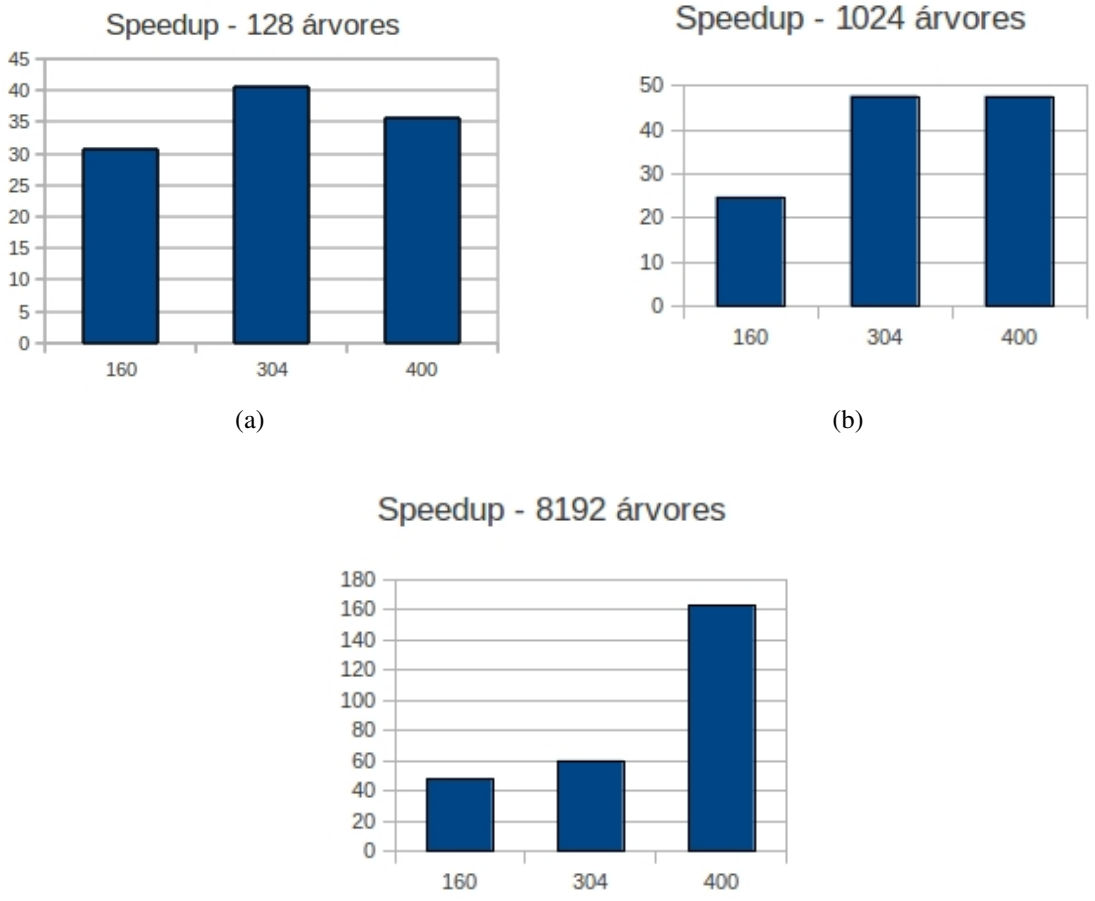


Figura 5.2: *Speedup alcançados ao executar todas as fases (incluso tempo de cópia dos dados para a GPU).*

Os ganhos de *performance* alcançados se justificam, principalmente, devido ao alto grau de paralelismo alcançado nas etapas que calculam a **matriz de distância** e o **I de Moran**. Nessas etapas, quando instanciamos o *kernel* para computar as 8.192 árvores, sendo cada árvore com 400 espécies (DummyB), lançamos milhões de threads (8.192×400), expondo um nível de paralelismos, até então, inimaginável para um sistema computacional do porte de uma GPU. Nas GPUs atuais, podemos trabalhar com filogenias

que possuem até 1.024 espécies, sem necessidade de fazer rodízio para utilizar a GPU, então alcançaríamos um número ainda maior de *threads* em execução.

Entretanto, devemos ter cautela quanto aos ganhos oferecidos por esse modelo de computação. Barreiras, como a limitação de recursos, por exemplo: memória e taxa de transferência de dados, podem ser impeditivos para trabalhar com modelos de simulações crescentes. Ao mesmo tempo, a tendência é que facilidades utilizadas no modelo sequencial seja trazidas para a arquitetura GPU, como é caso da memória cache, já presente na arquitetura Fermi. Além da limitação da arquitetura, temos também limites de implementação. Por utilizar variável inteira de 32 bits para representar o caminho binário, também limitamos a 32 a altura da árvore filogenética. Entretanto, é uma altura considerável, que permite acomodar uma árvore completamente balanceada com aproximadamente 4 bilhões de espécies.

CAPÍTULO 6

Conclusões e trabalhos futuros

No decorrer da pesquisa, buscamos identificar a melhor forma para representar os dados de entrada (árvore filogenética com suas relações de ancestralidade, PUTs e *Traits*). Conseguimos construir uma estrutura de dados que, em conjunto com algoritmos altamente paralelos, nós permitiu utilizar muito da potência da computação em GPU. Programas paralelos necessitam de acesso fácil e rápido ao conjunto dados. Propusemos algoritmos diferentes para cada uma das três etapas do programa, o que proporcionou abordar diferentes técnicas de paralelismo oferecidas pela arquitetura da GPU.

Na primeira etapa, tratamos as incertezas filogenéticas (ou PUTs), construindo um algoritmo que utiliza uma *thread* para cada árvore modificada. Mesmo conseguindo um nível razoável de paralelismo, proporcional ao número de árvores, não conseguimos superar a solução sequencial, que obteve excelentes resultados atualizando 8 mil árvores em menos de 1 segundo. Nossa solução paralela, obteve resultados superiores aos alcançados no algoritmo sequencial, atingindo o tempo de pouco mais de 1 segundo para gerar a árvore. Construímos outras alternativas de paralelização, como utilizar diversas *threads* para construir uma árvore. Contudo, nessa abordagem foi necessário o uso operações atômicas a cada atualização na árvore, que tornaram os resultados modestos. Essa tarefa, demonstrou ser essencialmente sequencial, conseguindo bons resultados nesse formato.

A próxima etapa, na qual construímos a matriz de distância, exploramos um nível de paralelismo extremamente alto, podendo chegar a ter milhões de *threads* em execução. Esse foi o caso da filogenia dos beija-flores (*hummingbirds*), que possui 304 espécies. Ao simularmos 8.192 árvores, conseguimos lançar um *grid* com 377.290.752 (mais de 377 milhões) de *threads*. Em seguida, dividimos as *threads* entre todos os multiprocessadores de *streamers* disponíveis na GPU. A criação de múltiplos blocos de *threads*, que são associados aos SMs, permitiu que atingíssemos bom desempenho, e ainda nos garante alta escalabilidade, com perspectiva de melhores *speedups* ao migrar para outros modelos de GPUs com maior capacidade de processamento.

A última etapa, realiza o cálculo do *Índice de Moran* e, assim como a segunda, aproveitou consideravelmente a concorrência na execução de suas ações, sendo capaz de lançar o mesmo número de *threads*. Além disso, utilizamos acesso rápido a memória

compartilhada para armazenar um pequeno conjunto de dados: classes de distância e variáveis acumuladoras. Essas classes, são consultadas constantemente por cada *thread*, sendo assim a garantia de acesso *instantâneo* a esses dados tem impacto positivo no desempenho da programa. Essa etapa lança uma bloco por árvore e cada bloco possui tantas *threads* quanto forem o número de espécies, da mesma foram que a segunda etapa. As *threads* colaboram para calcular o índice de correção espacial, I de Moram, ficando cada uma responsável por uma faixa de valores na matriz de distância. Entretanto, a colaboração cria necessidade do uso de operações atômicas e barreiras de sincronização, que podem degradar o desempenho do sistema. Mesmo assim, devido ao bom uso do paralelismo de dados e acesso aglutinado a memória, conseguimos obter bons resultados com essa solução.

Na segunda etapa, referente ao cálculo da matriz de distância, pudemos utilizar os resultados dos experimentos para comparar os programa *phylocom*, solução sequencial e solução paralela. Mostramos que, a utilização da computação paralela sobre GPU, pode trazer ganhos de performance acima de 200x, se comparado ao programa sequencial *Phylocom* ou até 100x quando comparado a solução sequencial. Também realizamos experimentos envolvendo as três etapas e o comparamos com a solução sequencial, onde alcançamos *speedups* de 60x. A solução paralela se mostrou eficiente, com ganhos consideráveis nos dois casos.

Alcançamos esses *speedups*, graças ao uso do modelo de computação paralela baseado em GPU, aliado a uma estrutura de dados planejada especialmente para essa arquitetura. Com o uso de algoritmos otimizados, para acesso aos dados dessa estrutura, foi possível alcançar um alto desempenho. É importante observar que, os ganhos foram obtidos com o uso de um modelo de computação paralela acessível a praticamente todos os nichos do mercado. Com pouco esforço de programação, podemos utilizar a computação paralela sobre GPU em nossas casas, sem a necessidade da construção de *clusters* e *grids* caros e de difícil implementação.

Apresentamos algumas possibilidades de melhorias em nossa solução, através da adição de uma estratégia para aleatorização das árvores filogenéticas utilizando simulações em alta escala. Além disso, pretendemos utilizar diversos outros métodos estatísticos que requerem a matriz de distância patrística como entrada. É importante avançar no sentido de implementarmos um sistema completo de simulação (utilizando o método de Monte Carlo) para estudos comparativos filogenéticos, incluindo o cálculo de diferente coeficientes (além do I de Moran) e análises estatísticas.

Referências Bibliográficas

[1] A ENCICLOPÉDIA LIVRE, W. **Multi-core processor**. http://en.wikipedia.org/wiki/Multi-core_processor. Accessado em 28 de maio de 2012.

[2] ADVANCED MICRO DEVICES, I. **Amd firestream 9350 / 9370 gpu compute accelerators**. http://www.amd.com/uk/products/server/processors/firestream/firestream-9370-9350/Pages/firestream-9370-9350.aspx. Accessado em 27 de maio de 2012.

[3] AHO, A. V.; HOPCROFT, J. E.; ULLMAN, J. D. **On finding lowest common ancestors in trees**. In: *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, p. 253–265, New York, NY, USA, 1973. ACM.

[4] AYRES, D. L.; DARLING, A.; ZWICKL, D. J.; BEERLI, P.; HOLDER, M. T.; LEWIS, P. O.; ADN FREDRIK RONQUIST, J. P. H.; SWOFFORD, D. L.; CUMMINGS, M. P.; RAMBAUT, A.; SUCHARD, M. A. **Beagle: an application programming interface and high-performance computing library for statistical phylogenetics**. *Systematic Biology*, 1;61(1):170–3, 2012.

[5] BENDER, M. A.; FARACH-COLTON, M. **The lca problem revisited**. In: *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, LATIN '00, p. 88–94, London, UK, UK, 2000. Springer-Verlag.

[6] BRODTKORB, A. R.; HAGEN, T. R.; SAETRA, M. L. **Graphics processing unit (gpu) programming strategies and trends in gpu computing**. *Journal of Parallel and Distributed Computing*, 2012.

[7] CÁCERES, E. N.; MONGELLI, H.; SONG, S. W. **Algoritmos Paralelos Usando CGM/PVM/MPI: Uma Introdução**. Sociedade Brasileira de Computação, 2001.

[8] CORPORATION, N. **Cuda c best practices guide**. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf. Accessado em 02 de maio de 2012.

[9] CORPORATION, N. **Nvidia cuda c programming guide**. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Accessado em 27 de junho de 2012.

[10] CORPORATION, N. **Nvidia fermi compute architecture whitepaper**. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessado em 02 de julho de 2012.

[11] CORPORATION, N. **O que é computação com gpu?** http://www.nvidia.com.br/object/tesla-supercomputing-solutions-br.html. Accessado em 01 de julho de 2012.

[12] DE CARVALHO, F. M. V.; FERREIRA, L. G.; LOBO, F. C.; DINIZ-FILHO, J. A. F.; BINI, L. M. **Padrões de autocorrelação espacial de índices de vegetação MODIS no bioma cerrado**. *Revista Árvore*, 32:279 – 290, 04 2008.

[13] DE SOUZA AMORIM, D. **Fundamentos de Sistemática Filogenética**. Holos, first edition, 1997.

[14] DINIZ FILHO, J. A. F. **Métodos Filogenéticos Comparativos**. Holos, Ribeirão Preto, São Paulo, Brasil, first edition, 2000.

[15] DINIZ FILHO, J. A. F.; VIEIRA, C. M. **Padrões e processos na evolução do tamanho do corpo em carnívoros (Mammalia) da América do Sul**. *Revista Brasileira de Biologia*, 58:649 – 657, 11 1998.

[16] FARIAS, R.; BENTES, C. **Placas gráficas: O novo cérebro veloz dos computadores**. *Ciência Hoje*, 47(281):32–35, 2011.

[17] FELSENSTEIN, J. **Confidence limits on phylogenies: an approach using the bootstrap**. Society for the Study of Evolution, first edition, 1985.

[18] FLYNN, M. J. **Some computer organizations and their effectiveness**. *IEEE Trans. Comput.*, 21(9):948–960, sep 1972.

[19] FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[20] FOURMENT, M.; GIBBS, M. J. **Patristic: a program for calculating patristic distances and graphically comparing the components of genetic change**. *BMC Evolutionary Biology*, 6:1, 2006.

[21] GEBALI, F. **Algorithms and Parallel Computing**. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, 2011.

[22] GRAMA, A.; GUPTA, A.; KARYPIS, G.; KUMAR, V. **Introduction to Parallel Computing**. Addison Wesley, Edinburgh, Harlow, England, second edition, 2003.

[23] GRAUR, D.; LI, W. **Fundamentals of Molecular Evolution**. Sinauer, 2000.

[24] HEY, T.; TANSLEY, S.; TOLLE, K.; BECK, L. **Quarto Paradigma, O: Descobertas Cientificas na Era da ESCIENCE**. OFICINA DE TEXTOS, 2011.

[25] HOLDER, M.; LEWIS, P. O. **An approach to the analysis of comparative when a phylogeny is unavailable or incomplete**. *Systematic Biology*, 43:117–123, 1994.

[26] HOLDER, M.; LEWIS, P. O. **Phylogeny estimation: traditional and Bayesian approaches**. *Nature Reviews Genetics*, 2003.

[27] HOUSWORTH, E. A.; MARTINS, E. P. **Random sampling of constrained phylogenies: conducting phylogenetic analyses when the phylogeny is partially known**. *Systematic Biology*, 50:628–639, 2001.

[28] HUELSENBECK, J. P.; RANNALA, B. **Detecting correlation between characters in a comparative analysis with uncertain phylogeny**. *Evolution*, 57(6):1237–1247, 2003.

[29] JONES, N. C.; PEVZNER, P. A. **An introduction to bioinformatics algorithms**. The MIT Press, Cambridge, MA, USA, first edition, 2004.

[30] KAI HWANG.; BRIGGS, F. A. **Computer architecture and parallel processing**. McGraw-Hill computer communications series. McGraw-Hill, 1984.

[31] KHRONOS GROUP. **Opencl - the open standard for parallel programming of heterogeneous systems**. `http://www.khronos.org/opencl/`. Accessado em 28 de junho de 2012.

[32] KIRK, D. B.; HWU, W.-M. W. **Programando para Processadores Paralelos**. Elsevier, Rio de Janeiro, RJ, BRA, first edition, 2011.

[33] KIRNER, C. **Arquitetura de Sistemas Avancados de Computacao**. SOCIEDADE BRASILEIRA DE COMPUTACAO, Rio de Janeiro, 1991.

[34] LESK, A. M. **Introdução a Bioinformática**. Artmed, Porto Alegre, RS, BRA, second edition, 2008.

[35] MARTINS, E. P. **Compare, version 4.6b. computer programs for the statistical analysis of comparative data**. Department of Biology, Indiana University, Bloomington IN.

[36] MATTSON, T. G.; SANDERS, B. A.; MASSINGILL, B. **Patterns for parallel programming**. Software patterns series. Addison-Wesley, first edition, 2005.

[37] MCGUIRE, J. A.; WITT, C. C.; ALTSHULER, D. L.; REMSEN, J. V. **Phylogenetic systematics and biogeography of hummingbirds: Bayesian and maximum likelihood analyses of partitioned data and selection of an appropriate partitioning strategy**. *Systematic Biology*, 56(5):837–856, 2007.

[38] MCINTOSH-SMITH, S. **The gpu computing revolution**. 2011.

[39] MEI HWU, W.; KEUTZER, K.; MATTSON, T. G. **The concurrency challenge**. *IEEE Design and Test of Computers*, 25:312–320, 2008.

[40] NICKOLLS, J.; DALLY, W. J. **The gpu computing era**. *IEEE Micro*, 30(2):56–69, Mar. 2010.

[41] OLSEN, G. **"Newick's 8:45"Tree Format Standard**, 1990.

[42] PALIN, M. F. **Técnicas de decomposição de domínio em computação paralela para simulação de campos eletromagnéticos pelo método dos elementos finitos**. Doutorado em sistemas de potência, São Paulo, SP, Brasil, 2007.

[43] PARHAMI, B. **Introduction to Parallel Processing: Algorithms and Architectures**. Plenum Series in Computer Science. Plenum Press, first edition, 1999.

[44] PARR, C. S.; GURALNICK, R.; CELLINESE, N.; PAGE, R. D. **Evolutionary informatics: unifying knowledge about the diversity of life**. *Trends in Ecology e Evolution*, 27(2):94 – 103, 2012. <ce:title>Ecological and evolutionary informatics</ce:title>.

[45] PETZOLD, E.; MERKLE, D.; MIDDENDORF, M.; VON HAESELER, A.; SCHMIDT, H. A. **Phylogenetic parameter estimation on cows**. *Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies*, 2005.

[46] RANNALA, B.; HUELSENBECK, J. P.; YANG, Z.; NIELSEN, R. **Taxon sampling and the accuracy of large phylogenies**. *Systematic Biology*, 47:702–710, 1998.

[47] SANDERS, J.; KANDROT, E. **CUDA by example: an introduction to general-purpose GPU programming**. Pearson Education, Boston, MA, USA, first edition, 2011.

[48] SCHIEBER, B.; VISHKIN, U. **On finding lowest common ancestors: simplification and parallelization**. *SIAM J. Comput.*, 17(6):1253–1262, Dec. 1988.

[49] SIPSER, M. **INTRODUÇÃO A TEORIA DA COMPUTAÇÃO: 2a EDIÇÃO NORTE-AMERICANA**. THOMSON PIONEIRA, 2007.

[50] STALLINGS, W. **Arquitetura e Organização de computadores**. Prentice Hall, São Paulo, SP, Brasil, fifth edition, 2002.

[51] STAMATAKIS, A. **Parallel and distributed computation of large phylogenetic trees**. *Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies (ed A. Y. Zomaya), John Wiley & Sons, Inc.*, 2005.

[52] SUCHARD, M. A.; RAMBAUTS, A. **Many-core algorithms for statistical phylogenetics**. *Bioinformatics*, 25:1370–1376, 2009.

[53] TANENBAUM, A. S. **Organização estruturada de computadores**. Pearson Prentice Hall, São Paulo, SP, Brasil, fifth edition, 2007.

[54] TELLES, M. P. D. C.; DINIZ-FILHO.; F., J. A.; COELHO.; AD LAZARO J. CHAVES, A. S. **Autocorrelação espacial das frequêcia alélicas em subpopulações de cagaiteira (Eugenia dysenterica DC., Myrtaceae) no sudeste de Goiá!** *Revista Brasileira de Botânica*, 24:145 – 154, 06 2001.

[55] WEBB1, C. O.; ACKERLY, D. D.; KEMBEL, S. W. **Phylocom: software for the analysis of phylogenetic community structure and trait evolution**. *Bioinformatics*, 24:2098–2100, 2008.

[56] WIKIPEDIA. **Bioinformatics**. `http://en.wikipedia.org/wiki/Bioinformatics`. Accessado em 28 de junho de 2012.

[57] WILLIAMS, T. L.; BADER, D. A.; MORET, B. M. E.; YAN, M. **High-performance phylogeny reconstruction under maximum parsimony**. *Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies*, 2005.

[58] ZIVIANI, N. **PROJETO DE ALGORITMOS: COM IMPLEMENTAÇOES EM PASCAL E C**. CENGAGE, 2004.