Universidade do Estado do Rio de Janeiro

Centro de Tecnologia e Ciências

Instituto de Matemática e Estatística

Leandro Rouberte de Freitas e Gabriela da Cruz Paranhos

Estudo da paralelização do algoritmo de alinhamento global de sequências biológicas Needleman-Wunsch em arquiteturas multicore e manycore

Rio de Janeiro

2013.1

Leandro Rouberte de Freitas e Gabriela da Cruz Paranhos

Estudo da paralelização do algoritmo de alinhamento global de sequências biológicas Needleman-Wunsch em arquiteturas multicore e manycore

Projeto Final apresentado ao Instituto de Matemática e Estatística da Universidade do Estado do Rio de Janeiro, para obtenção do grau de bacharel em Ciência da Computação

Orientador: Prof. Dr. Leandro Augusto Justen Marzulo
Coorientador: Prof. M.Sc. Alexandre Solon Nery

Rio de Janeiro
2013.1

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC-A

SXXX Rouberte, Leandro; Cruz, Gabriela.
Estudo da paralelização do algoritmo de alinhamento global de sequências biológicas Needleman-Wunsch em arquiteturas multicore e manycore / Leandro Rouberte de Freitas e Gabriela da Cruz Paranhos. – 2013.
138f. : il.
Orientador: Prof. Dr. Leandro Augusto Justen Marzulo
Coorientador: Prof. M.Sc. Alexandre Solon Nery

Projeto final (Bacharel em Ciência da Computação) - Universidade do Estado do Rio de Janeiro, Instituto de Matemática e Estatística.
1. XXXXXXXXXX 2. XXXXXXXX. I. Marzulo, Leandro. II. Nery, Alexandre. III. Universidade do Estado do Rio de Janeiro. Instituto de Matemática e Estatística. III. Título.

CDU XXX.XX

_____ _____

Assinatura Data

_____ _____

Assinatura Data

Leandro Rouberte de Freitas e Gabriela da Cruz Paranhos

Estudo da paralelização do algoritmo de alinhamento global de sequências biológicas Needleman-Wunsch em arquiteturas multicore e manycore

Projeto Final apresentado ao Instituto de Matemática e Estatística da Universidade do Estado do Rio de Janeiro, para obtenção do grau de bacharel em Ciência da Computação.

Aprovada em 03 de maio de 2013.

Banca Examinadora:

_____

Prof. Dr. Leandro Augusto Justen Marzulo - Orientador
Instituto de Matemática e Estatística - UERJ

_____

Prof. M.Sc. Alexandre Solon Nery - Coorientador
Instituto de Matemática e Estatística - UERJ

_____

Prof.ª Dra. Maria Clícia Stelling
Instituto de Matemática e Estatística - UERJ

_____

Prof. M.Sc. Tiago Assumpção de Oliveira
COPPE - UFRJ

Rio de Janeiro
2013

RESUMO

ROUBERTE, Leandro; CRUZ, Gabriela. ***Estudo da paralelização do algoritmo de alinhamento global de sequências biológicas Needleman-Wunsch em arquiteturas multicore e manycore.*** 20xx. xx f. Projeto final (Bacharelado em Ciência da Computação) - Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2013.

Alinhamento de sequências biológicas é uma ferramenta de análise computacional utilizada no campo da Bioinformática para avaliar semelhanças entre duas ou mais sequências. Diariamente milhões de sequências são comparadas ao redor do mundo para auxiliar acadêmicos em pesquisas científicas, tais como evolução das espécies, curas de doenças, estudo de genômas de seres vivos, dentre outros. Os algoritmos utilizados para essa tarefa demandam um imenso poder computacional. Desta forma, se torna necessário criar ou aperfeiçoar sistemas computacionais de alto desempenho, assim como algoritmos de alinhamento de sequências. O paradigma de programação paralela tem sido uma prática bastante abordada para melhoria de desempenho de problemas computacionais. Com o avanço das arquiteturas multicore e manycore, esse paradigma, que até então era restrito aos programadores de alto desempenho, se tornou mais popular. O algoritmo de alinhamento global de sequências Needleman-Wunsch foi o primeiro a empregar a técnica de programação dinâmica para encontrar um alinhamento ótimo entre duas sequências. Neste trabalho é proposto um estudo do paradigma de programação paralela em arquiteturas de GPU e memória compartilhada de forma a avaliar o desempenho do algoritmo de Needleman-Wunsch nessas arquiteturas.

Palavras-chave: Needleman-Wunsch. Alinhamento de sequências biológicas. GPU. CUDA. OpenMP. Programação Paralela.

ABSTRACT

Alignment of biological sequences is a tool of computational analysis used in bioinformatics to evaluate similarities between two or more sequences. Every day millions of sequences are compared around the world to assist academics in scientific research, such as the evolution of species, cures for diseases, study of living organisms genomes , among others. The algorithms used for this task require has become enormous computational power. Thus, it becomes necessary to create or enhance high performance computing systems, as well as algorithms of sequence alignment. The parallel programming paradigm has been a well known practice to improve performance of computational problems. With the advances on multicore and manycore architectures, this paradigm, which was restricted to developers of high-performance, became more popular. The Needleman-Wunsch algorithm was the first to employ dynamic programming techniques to find an optimal global alignment between two sequences. This paper proposes a study of the parallel programming paradigm on GPU architectures and shared memory to evaluate the performance of the Needleman-Wunsch algorithm on such architectures.

Keywords: Needleman-Wunsch. Alignment of biological sequences. GPU. CUDA. OpenMP. Parallel Programming.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

DNA –	Ácido Desoxirribonucleico
RNA –	Ácido Ribonucléico
CPU ou UCP –	Unidade de Processamento Central
UC ou CU –	Unidade de Controle
ALU ou ULA –	Unidade Lógica e Aritmética
CI ou PC –	Contador de Instruções ou *Program Counter*
AP ou SP –	Apontador da Pilha ou *Stack Pointer*
GPU –	Unidade de Processamento Gráfico
API –	*Application Programming Interface* (Interface de Programação de Aplicativos)
CUDA –	*Computer Unified Device Architecture*
OPENMP –	*Open Multi-Processing* (Multi-processamento aberto)
MSA –	Alinhamento de Sequências Múltiplas
DP –	Programação Dinâmica
PAM –	*Point Accepted Mutations* ou *Percent of Accepted Mutations*
BLOSSUM –	*Blocks Amino Acid Substitution Matrices*

NW – Needleman-Wunsch

MPI – *Message Passing Interface* (Interface de Passagem de Mensagens)

BSP – Bulk Synchronous Parallel Model

GCM – *Coarse Grained Multicomputers*

EARTH – Eficient Architecture for Running Threads (Arquitetura Eficiente para Execução de *Threads*)

PSW – *Program Status Word* (Registrador de Status)

MAR – *Memory Address Register* (Registrador de Endereço de Memória)

MBR – *Memory Buffer Register* (Registrador de Dados de Memória)

RAM – *Random Access Memory* (Memória Principal)

ROM – *Read-Only Memory*

EPROM – *Erasable Programmable* ROM

PCB – *Process Control Block* (Bloco de Controle do Processo)

SO – Sistema Operacional

SISD – *Single Instruction Single Data*

SIMD – *Single Instruction Multiple Data*

MIMD – *Multiple Instruction Multiple Data*

MISD – *Multiple Instruction Single Data*

SM ou SMP – Multiprocessador de *Streaming*

SP – Processador de *Streaming* ou CUDA *core*

ID – *Identification* (Identificação)

BLAST – *Basic Local Alignment Search Tool*

FASTA – *Fast Alignment Sequence Algorithm*

SUMÁRIO

INTRODUÇÃO

Com o descobrimento da molécula de DNA e o advento da biologia molecular, foram revelados os blocos construtores da vida. Capazes de fornecer uma descrição precisa de cada componente de um ser vivo, o DNA tem sido alvo de grandes pesquisas com objetivo de encontrar curas para doenças genéticas, técnicas de aperfeiçoamento biológico de seres vivos e determinação de parentesco evolutivo, dentre outros. (WATSON; BERRY, 2003)

O campo da biologia molecular convergiu para o campo da computação devido à necessidade de armazenar e analisar quantidades massivas de dados gerados pelo sequenciamento moderno de moléculas de DNA, RNA e aminoácidos. Dessa forma surgiu um novo campo de estudo conhecido como Bioinformática. (MOUNT, 2004)

No campo emergente da Bioinformática, o alinhamento de sequências é uma técnica computacional que faz uso de algoritmos - conhecidos como Algoritmos de Alinhamento de Sequências – a fim de se descobrir similaridades entre duas ou mais sequências de DNA, RNA ou aminoácidos. Esses alinhamentos podem ser globais, isto é, pesquisar similaridades com base em uma sequência completa, ou locais, os quais consistem em procurar semelhanças entre dois ou mais fragmentos de diferentes sequências. (SHARMA, 2009)

Com o avanço de sequenciadores de moléculas biológicas, números expressivos de sequências genéticas são gerados e armazenados diariamente em bancos de dados biológicos públicos. Em média uma sequência genética possui milhões de pares bases. Desta forma, alinhar duas ou mais sequências gera uma necessidade de computadores com grande capacidade de processamento e armazenamento, como também, algoritmos eficientes capazes de alinhar sequências biológicas em um período aceitável de tempo. (ZOMAYA, 2006)

Devido à evolução crescente e rápida de sistemas computacionais, hardwares com processadores cada vez mais potentes foram desenvolvidos. Os fabricantes desses processadores baseavam-se em obter maiores frequências de clock para atingir um desempenho desejável, porém esta estratégia atingiu os limites físicos dos componentes da CPU. Este fato acarretou no desenvolvimento de novos processadores capazes de efetuar mais tarefas em paralelo para gerar

processadores melhores sem haver necessidade de quebrar os limites físicos do chip de processamento. (KIRK; HWU, 2010)

A Bioinformática se une a programação paralela com o objetivo de desenvolver aplicações, sistemas e algoritmos capazes de auxiliar no processamento de dados massivos de forma eficiente. Essa eficiência é necessária por conta da grande informação biológica de seres vivos. Através dela novas descobertas sobre a vida na terra podem ser feitas. (ZOMAYA, 2006)

Atualmente existem diversos algoritmos criados para alinhamento de sequências por programação dinâmica: tanto globais, como locais. O objetivo deste trabalho consiste em fazer uma análise do algoritmo de alinhamento global conhecido como "Algoritmo de Needleman-Wunsch", e criar versões paralelizadas deste utilizando duas arquiteturas distintas. A primeira consiste em uma arquitetura baseada em GPUs (processadores gráficos) da fabricante NVIDIA utilizando a linguagem CUDA. A segunda é uma arquitetura de memória compartilhada e vários processadores multicores, manipulada pela API OpenMP.

Este trabalho está organizado da seguinte forma: o primeiro capítulo trata-se de uma breve introdução aos assuntos da Biologia Molecular, o segundo disserta sobre uma ferramenta de análise muito utilizada na Bioinformática conhecida como alinhamento de sequências, o terceiro capítulo descreve conceitos de programação paralela e arquitetura de sistemas computacionais utilizados neste trabalho. O quarto capítulo refere-se ao problema de paralelização do algoritmo de Needleman-Wunsch, enquanto o capítulo cinco descreve as versões paralelizadas do algoritmo implementadas neste trabalho e avalia os seus desempenhos. Por fim, o capítulo seis apresenta as conclusões sobre este projeto e descreve trabalhos futuros.

1 **CONCEITOS DE BIOLOGIA MOLECULAR**

A comparação de sequências de DNA, RNA e proteínas é um dos principais recursos utilizados por biólogos a fim de que se possam realizar pesquisas no campo da biologia molecular. Esse capítulo explica a estrutura do DNA, RNA e proteínas, assim como suas funções, para que se possa entender melhor o estudo desse trabalho.

1.1 **Estrutura e funções do DNA**

Os seres vivos são compostos por células e foram gerados por divisões celulares de uma única célula. Quando são compostos por uma única célula, são chamados unicelulares, e os compostos por várias células, multicelulares. A célula, portanto é a portadora da informação hereditária que define as espécies e especificada nessas informações estão as instruções para construir cópias da célula original com a informação hereditária completa. (ALBERTS et al., 2008)

Todas as células guardam sua informação hereditária na forma de moléculas chamadas DNA (ácido desoxirribonucleico), as quais são constituídas por duas fitas formando uma estrutura helicoidal. O DNA é composto por quatro tipos de monômeros, isto é, pequenas moléculas capazes de ligar-se a outros monômeros, também conhecidos como nucleotídeos. Estes são representados por um alfabeto de quatro letras A, T, C e G. Eles estão ligados em uma longa sequência química linear que codifica a informação hereditária assim como os valores 1 e 0 codificam um arquivo de computador. Pode-se colocar um pedaço do DNA humano em uma bactéria ou vice-versa, e essa informação será lida, interpretada e copiada com sucesso. Cientistas através de métodos químicos podem fazer leituras de sequências de monômeros em qualquer molécula de DNA, que pode se estender por milhões de nucleotídeos, e assim decifrar a informação hereditária de um organismo vivo. (ALBERTS et al., 2008)

Cada nucleotídeo é composto por duas partes: uma pentose (molécula de açúcar) chamada desoxirribose ligado a um grupo fosfato, e uma base nitrogenada,

a qual pode ser uma adenina (A), guanina (G), citosina (C) ou timina (T). As bases nitrogenadas são classificadas em purinas (adenina e citosina) e pirimidinas (timina e guanina). (ALBERTS et al., 2008)

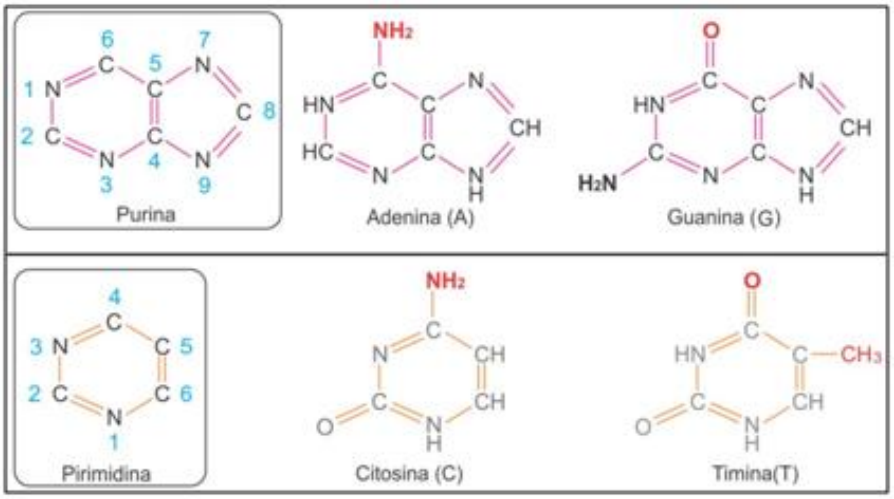


Figura 1: Estrutura molecular das bases nitrogenadas que compões os nucleotídeos da molécula de DNA.
Fonte:
http://www.facom.ufms.br/~tmcomparisons/genetica.htm
(Acessado em 20/04/2013)

O mecanismo que torna a vida possível depende da estrutura tridimensional de dupla hélice do DNA, formada por bases de nucleotídeos ligados através de pontes de hidrogênio, onde a Adenina é sempre ligada à Timina por duas pontes de hidrogênio, e a Guanina à Citosina por três pontes de hidrogênio. Cada faixa de uma molécula de DNA possui uma sequência de nucleotídeos que é exatamente complementar à de sua fita vizinha. (ALBERTS et al., 2008, pg. 199)

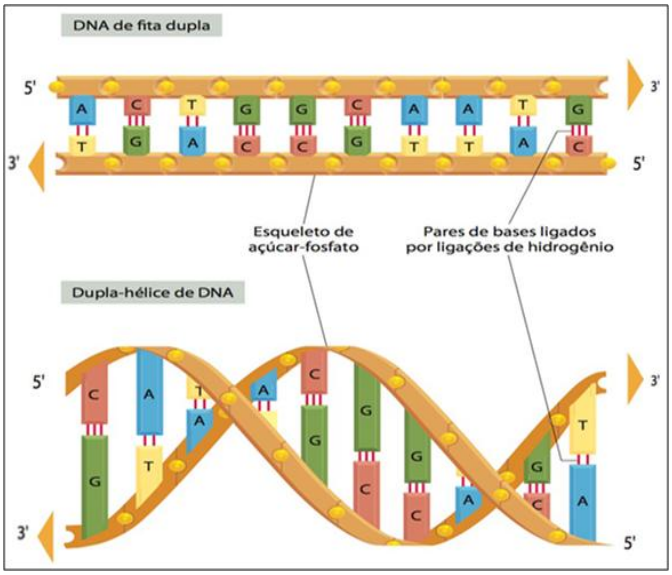


Figura 2: Molécula de DNA com sua orientação 5' - 3' mostrada em sua forma tradicional de dupla-hélice e na sua forma desespiralizada.
Fonte: ALBERTS et al., 2008.

Genes são informações biológicas formadas por sequências de nucleotídeos, as quais servem de instrução para a construção de proteínas através de um procedimento chamado expressão gênica. Esse é o procedimento no qual a célula converte sequências de nucleotídeos de determinado gene em uma sequência nucleotídica de uma molécula de RNA, e depois a converte em uma sequência de aminoácidos que comporão uma determinada proteína. O conjunto completo de informações em um organismo é chamado de genoma. Diferentes genomas produzem diferentes organismos. (ALBERTS et al., 2008)

As informações genéticas precisam ser copiadas e passadas para a próxima geração. Para alcançar este objetivo, deve ser feito uma cópia da molécula de DNA. Inicialmente ocorre a separação das suas duas fitas do DNA. Então, cada fita serve de molde para construção de sua fita complementar, formando no final do processo uma cópia da molécula de DNA. (ALBERTS et al., 2008)

1.2 **Estrutura e funções do RNA**

A molécula de RNA (ácido ribonucleico) difere da molécula de DNA em três aspectos importantes:

1) A estrutura principal do RNA contém uma ribose como açúcar ao invés da desoxirribose do DNA.
2) Contém a Uracila (U) como uma de suas bases nitrogenadas no lugar da Timina (T) do DNA.
3) É geralmente encontrada em seres vivos como uma cadeia polinucleotídica singular.

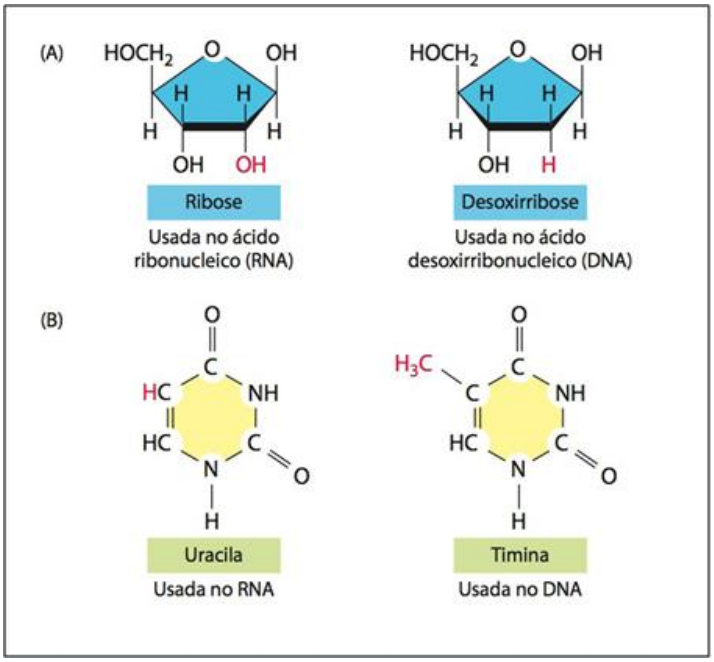


Figura 3: (A) O RNA contém o agrupamento OH em sua molécula de açúcar - a ribose. (B) A base timina é substituída no RNA pela base uracila, a qual se diferencia quimicamente por não possuir o grupo CH3. Fonte: ALBERTS et al., 2008.

O RNA é fabricado a partir de moléculas de DNA através de um processo conhecido como transcrição. A informação na molécula de RNA é escrita essencialmente na mesma linguagem do DNA - a linguagem de uma sequência de quatro tipos diferentes de nucleotídeos. O processo de transcrição apresenta certas similaridades em relação ao processo de replicação do DNA. Esse processo começa com a desespiralização de uma pequena porção da dupla-hélice de DNA, o que

expõe as bases nitrogenadas em cada fita de DNA. Uma das fitas serve então como molde para a síntese de uma molécula de RNA.

As células produzem diversos tipos de RNA, porém a maioria dos genes carregados no DNA das células específica a sequência de aminoácidos de proteínas; as moléculas de RNA que são copiadas a partir desses genes (e que definem a síntese de proteínas) são chamadas de moléculas de RNA mensageiro ou mRNA. O produto final de uma minoria de genes, entretanto, é o próprio RNA. (ALBERTS et al., 2008)

1.3 **Estrutura e funções de proteínas**

Proteínas são moléculas constituídas por uma longa cadeia de aminoácidos concatenados por uma ligação peptídica covalente. Por este motivo as proteínas são conhecidas também como cadeias polipeptídicas ou polipeptídios. Cada tipo de proteína possui uma sequência exclusiva de aminoácidos. (ALBERTS et al., 2008)

Todos os aminoácidos possuem uma estrutura básica formada de um átomo de carbono geral, chamado de C alfa (Cα), ligado a um grupamento amínico (-NH2) ou íminico (-NH-). No caso do aminoácido prolina, um grupamento carboxílico (-COOH) é ligado a um grupamento variável, denominado R ou cadeia lateral. Cada proteína é composta por 20 aminoácidos, podendo estes ser agrupados em quatro cadeias: ácidas, básicas, polares não-carregadas e apolares. (ALBERTS et al., 2008) (ZAHA et al., 2003)

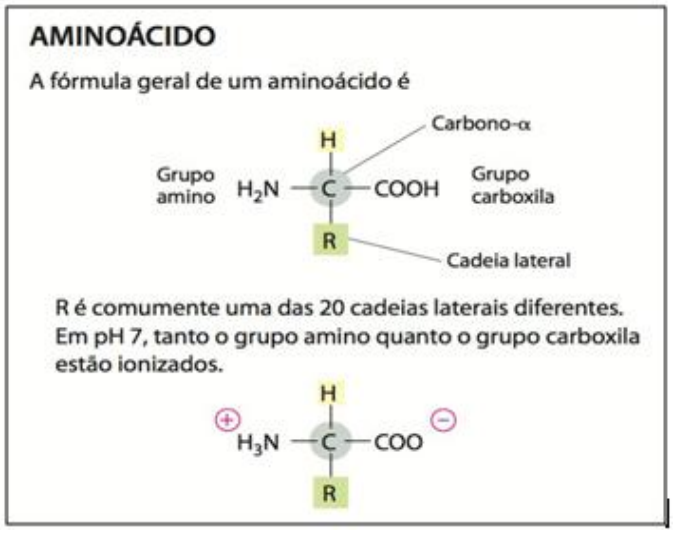


Figura 4: Ilustração da fórmula geral de um aminoácido.
Fonte: ALBERTS et al., 2008.

A união de aminoácidos para formar uma molécula protéica produz-se através de uma ligação entre o grupamento carboxílico de um aminoácido com o grupamento amínico de outro aminoácido, com perda simultânea de uma molécula de água. A este tipo de ligação denomina-se ligação peptídica. (ZAHA et al., 2003)

As proteínas diferenciam-se em quatro níveis de organização estrutural. A estrutura primária é formada pela sequência de aminoácidos, ou seja, a ordem na qual os aminoácidos estão interligados para formarem uma cadeia peptídica. A sequência de aminoácidos é a primeira etapa unidimensional na especificação da estrutura tridimensional e determina o nível estrutural mais importante da molécula. (ZAHA et al., 2003)

A organização espacial dos aminoácidos que se encontram próximos na cadeia peptídica central define a estrutura secundária de uma proteína, já a forma como a cadeia polipeptídica está dobrada, incluindo arranjo tridimensional de todos os átomos desta molécula, define a estrutura terciária da proteína. Este nível estrutural se estabelece quando diferentes estruturas secundárias dispõem-se entre si. Por último, a estrutura quaternária de uma proteína é definida pela disposição das subunidades protéicas que formam a molécula. (ZAHA et al., 2003)

Proteínas possuem grande importância em organismos vivos, constituem mais de metade do peso seco de uma célula, desempenham inúmeras funções biológicas, tais como catalisar reações químicas, controlar permeabilidade das

membranas, proporcionar movimento, controlar a expressão gênica, etc.. e determinar a forma e estrutura da célula. (ZAHA et al., 2003)

1.4 **Processo de síntese de proteínas**

Proteínas são cadeias polipeptídicas constituídas por aminoácidos em séries determinadas pelo DNA. A relação entre a sequência de bases no DNA e a sequência correspondente de aminoácidos, na proteína, recebe a denominação de código genético. O código genético é lido em códons, ou seja, grupos de três nucleotídeos que correspondem a um aminoácido. Um gene inclui uma série de códons que é lida sequencialmente a partir de um ponto de iniciação em uma das extremidades da molécula, até o ponto de terminação, localizado na outra extremidade. (ZAHA et al., 2003)

A síntese de proteínas não é diretamente direcionada pelo DNA, mas faz utilização do RNA como uma molécula intermediária. Quando a célula necessita de uma proteína específica, a sequência de nucleotídeos da região apropriada de uma molécula de DNA imensamente longa em um cromossomo é inicialmente copiada sob a forma de RNA (por meio do processo de transcrição). Estas cópias de RNA a partir de segmentos de DNA são utilizadas diretamente como moldes para direcionar a síntese de proteínas (em um processo denominado tradução). O fluxo de informação genética nas células é, portanto, de DNA para RNA, e para proteína. Todas as células, desde bactérias até seres humanos, expressam sua informação genética desta maneira - a este princípio denomina-se dogma central da biologia molecular. (ALBERTS et al., 2008)

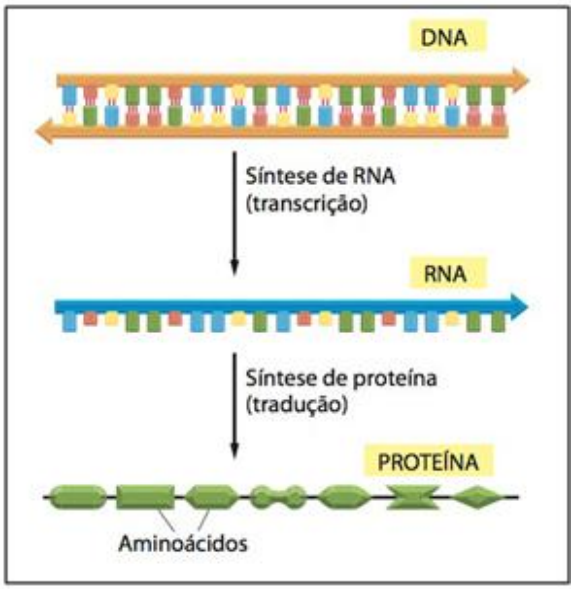


Figura 5 : Ilustração do processo geral de síntese de uma proteína.
Fonte: ALBERTS et al., 2008.

O primeiro passo executado por uma célula para expressar suas informações genéticas é a cópia de uma parcela específica de sequências de nucleotídeos do DNA - um gene - sob a forma de uma sequência de nucleotídeos de RNA. Esta produção de RNA pode ser feita de acordo com a demanda da célula por proteínas, ou seja, podem ser feitas várias cópias de uma molécula de RNA caso haja uma necessidade maior de uma proteína que seja traduzida a partir daquela fita de RNA. Esta molécula sintetizada a partir de uma fita molde de DNA que codifica uma proteína é chamada de mRNA ou RNA mensageiro.(ALBERTS et al., 2008)

2 **ALINHAMENTO DE SEQUÊNCIAS BIOLÓGICAS**

Alinhamento de sequências biológicas é uma ferramenta de suma importância no campo da Bioinformática. Através da similaridade obtida com o alinhamento de sequências de DNA, RNA e proteínas é possível determinar ancestrais em comum, cura para doenças, e informações genéticas de grande importância para estudos.

Nesse capítulo será abordado esse assunto para melhor compreensão do estudo dos algoritmos de alinhamento de sequências.

2.1 **Descrição de Alinhamento de Sequências Biológicas**

Alinhamento de sequências é o processo de alinhar duas ou mais sequências e compará-las, a fim de se obter semelhanças (também chamadas *matches*) entre elas. Quando duas sequências são alinhadas, o alinhamento é chamado *pairwise alignment*, ou seja, alinhamento par a par. Quando mais que duas sequências são alinhadas e examinadas, esse alinhamento é chamado *multiple-sequence alignment*, ou seja, alinhamento múltiplo de sequências. (SHARMA, 2009)

A similaridade entre as sequências apresentadas pode ser baseada em relações evolutivas, estruturais ou funcionais entre elas. Semelhanças encontradas entre as sequências de nucleotídeos são também chamados *identity* (identidade). *Conservation* (Conservação) refere-se a alterações em uma posição específica de uma sequência de aminoácidos que preservam as propriedades físico-químicas do resíduo original. (SHARMA, 2009)

Se duas sequências de diferentes organismos são semelhantes, elas podem ter uma sequência ancestral comum, e as sequências são então definidas homólogas. (MOUNT, 2004) (HENIKOFF et al., 1997)

Quando duas ou mais sequências são alinhadas e ligadas a um antepassado comum, e diferenças (também chamadas *mismatches*) são encontradas no alinhamento, essas diferenças podem ser consideradas mutações pontuais. Lacunas (chamadas *gaps*) nas sequências podem ser vistas como *indels*, termo usado para

se referir à inserção (quando há uma adição de nucleotídeos em uma sequência ancestral) ou deleção (quando há perda de nucleotídeos). (SHARMA, 2009)

2.2 **Objetivos do uso de Alinhamento de Sequências Biológicas**

Alinhamento de sequências é útil para descobrir informações funcionais, estruturais e evolutivas em sequências biológicas através do alinhamento “ideal” obtido. As sequências que são muito semelhantes, ou “similares” na linguagem de análise de sequências, provavelmente têm a mesma função, quer seja um papel regulador no caso de moléculas de DNA semelhantes, ou uma função bioquímica e estrutura tridimensional semelhantes, no caso de proteínas. (HENIKOFF et al., 1997)

O alinhamento indica as alterações que possam ter ocorrido entre as duas sequências homólogas e uma sequência ancestral comum durante a evolução. (HENIKOFF et al., 1997) Um exame mais detalhado dos alinhamentos pode ajudar a resolver possíveis origens evolutivas entre sequências semelhantes. (TATUSOV et al., 1997)

A descoberta de genes e seu papel nos mecanismos de alguma doença podem ser atingidos através de um alinhamento de sequência. (SHARMA, 2009)

A partir da semelhança entre as sequências de proteínas podemos determinar o grau de conservação entre elas. Conservação de pares de bases de DNA ou RNA podem indicar papéis funcionais e estruturais semelhantes. (SHARMA, 2009)

O objetivo do alinhamento de sequências é poder selecionar duas ou mais sequências e compará-las para determinar a similaridade. O grau de similaridade é uma medida usada para tirar conclusões sobre a existência de homologia entre duas sequências. (SHARMA, 2009)

2.3 **Exemplos do uso de Alinhamento de Sequências Biológicas**

Alinhamento de sequências pode ser a chave para encontrar a cura para doenças autoimunes. Nesse tipo de doença ao invés da aniquilação orientada ao

vírus acusado pelo sistema imunológico, o sinal de dentro do paciente desencadeia o ataque das suas próprias células. Quando a distribuição de sequência do vírus acusado e a distribuição de sequência nas células no sítio da doença no paciente são idênticas, o sistema imunológico ataca o próprio organismo, danificando assim suas células. Uma vez que esta semelhança é determinada através do alinhamento de sequências, medicamentos podem ser projetados cuja ação terapêutica pode alterar a expressão do gene, realizando assim a cura. (SHARMA, 2009)

Outra aplicação é na esclerose múltipla, onde as células T do sistema imunológico atacam as células nervosas do paciente. As proteínas que foram sequenciadas foram combinadas num banco de dados de proteínas com sequências semelhantes bacterianas e virais, e os testes foram realizados para determinar se as células T atacaram as proteínas com a mesma sequência que as proteínas bacterianas e virais. O resultado foi a identificação de certas proteínas bacterianas e virais que foram confundidas com as proteínas relacionadas a esse problema. O mecanismo molecular desta doença pode ser entendido usando alinhamento de sequências. (SHARMA, 2009)

Também podemos usar o alinhamento de sequências na obtenção de herança comum e na construção da árvore evolutiva a partir do conhecimento adquirido. Com base na suposição de que os organismos intimamente relacionados possuem sequências similares, foi descoberto que Chimpanzé e Homo sapiens possuem um ancestral comum, e as asas de morcegos e as de borboletas evoluíram de forma independente. (SHARMA, 2009)

Alinhamento de sequências múltiplas (MSA) é usado no estudo de doenças genéticas. (SHARMA, 2009)

2.4 **Tipos de Alinhamento de Sequências**

Existem dois tipos de alinhamento de sequências: global e local.

No alinhamento global, é feita uma tentativa de alinhar toda a sequência, usando o máximo de caracteres possível até as suas extremidades, a fim de se obter tantos *matches* quanto possíveis. As sequências que são bastante semelhantes e possuem aproximadamente o mesmo comprimento são candidatas

para o alinhamento global. O sistema mais simples de pontuação consiste em uma penalidade por alinhar um símbolo e um espaço, uma pontuação para alinhar dois símbolos diferentes, e ainda uma pontuação para alinhar dois símbolos idênticos. (MOUNT, 2004)

No alinhamento local, trechos de sequência com a maior densidade de *matches* são alinhados, o que gera subalinhamentos nas sequências alinhadas. Alinhamentos locais são mais adequados para o alinhamento de sequências que são semelhantes em algumas partes e diferentes em outras, sequências que diferem em comprimento, ou sequências que partilham uma região conservada ou de domínio. Traços indicam sequências não incluídas no alinhamento. Este tipo de alinhamento é favorecido ao se encontrar padrões de nucleotídeos conservados, sequências de DNA, ou padrões de aminoácidos nas sequências de proteína. (MOUNT, 2004)

Quando duas sequências são alinhadas, o alinhamento é chamado *pairwise alignment*, ou seja, alinhamento par a par. O alinhamento de duas sequências é feito utilizando três métodos: (i) análise de matriz de pontos, (ii) algoritmos de programação dinâmica (DP) e (iii) métodos de palavras, como utilizado pelo FASTA e BLAST. (MOUNT, 2004)

Quando mais que duas sequências são alinhadas e examinadas, esse alinhamento é chamado *multiple-sequence alignment*, ou seja, alinhamento múltiplo de sequências. A partir de um alinhamento múltiplo de três ou mais sequências de proteínas, os resíduos altamente conservados que definem os domínios estruturais e funcionais em famílias de proteínas podem ser identificados. Novos membros dessas famílias podem ser encontrados através de pesquisa às bases de dados de sequência para outras sequências com esses mesmos domínios. (MOUNT, 2004)

2.5 **Algoritmos de Alinhamento de Pares de Sequência**

2.5.1 Algoritmos de Alinhamento por Programação Dinâmica

Programação Dinâmica é um paradigma algorítmico muito poderoso no qual o problema é solucionado identificando-se uma coleção de subproblemas, resolvendo-os um por um, começando pelos menores. As respostas desses problemas são usadas para ajudar na solução dos problemas maiores, até que todos eles sejam solucionados. (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2006)

2.5.2 Alinhamento Global: Algoritmo Needleman-Wunsch

Neste trabalho será proposto uma avaliação de desempenho deste algoritmo.

O método de programação dinâmica fornece um alinhamento global de sequências, como descrito por Needleman e Wunsch (1970), mas também foi demonstrado matematicamente e estendido para incluir um sistema de pontuação melhorado por Smith e Waterman (1981). O algoritmo Needleman-Wunsch maximiza o número de *matches* entre as sequências ao longo de todo o comprimento das sequências.

Esse algoritmo consiste em construir uma matriz, onde a primeira sequência será sua linha e a segunda será sua coluna. Primeiramente é realizado o alinhamento trivial das sequências a serem alinhadas, que consiste em preencher a primeira linha e coluna dessa matriz com a pontuação atribuída à *gaps*. (NEEDLEMAN; WUNSCH, 1970)

Essa inicialização é feita atribuindo zero ao primeiro elemento da matriz $[0,0]$, preenchendo o restante da primeira linha $[0, j]$ com o valor de j multiplicado pela pontuação atribuída à *gaps* e o restante da primeira coluna $[i, 0]$ com o valor de i multiplicado pela pontuação atribuída à *gaps*. (NEEDLEMAN; WUNSCH, 1970)

Essa etapa é descrita na fórmula abaixo (*gap* é a penalidade por inserção de espaços vazios na sequência):

$$S[0,0] = 0 \quad (1)$$

$$S[i, 0] = \ i \times gap, onde\ i \in [1, comprimento(A)] \quad (2)$$

$$S[0, j] = \ j \times gap, onde\ j \in [1, comprimento(B)] \quad (3)$$

O restante da matriz é preenchido com a pontuação dos alinhamentos ótimos de cada subsequência, até que o alinhamento ótimo das duas sequências inteiras é encontrado. Esse procedimento é feito para cada $[i,j]$ da matriz, escolhendo sempre o maior valor entre $[i-1,j-1]$ somado com a pontuação referente a um *match* ou *mismatch*, $[i-1,j]$ e $[i,j-1]$ somados com a penalidade por inserção de um *gap* nas sequências. (NEEDLEMAN; WUNSCH, 1970)

O elemento $[i-1,j-1]$ é somado com a pontuação retornada pela função $f(a_i,b_j)$, onde a_i representa o símbolo i da sequência a e b_j representa o símbolo j da sequência b. Essa função retorna o resultado da comparação entre dois caracteres de sequências (a_i,b_j) sendo um *match* (somando o valor de 1) ou um *mismatch* (somando o valor de 0). A pontuação de $[i-1,j]$ e $[i,j-1]$ são dados pela soma com a penalidade de *gap* (somando o valor de -1). *Gaps* podem também estar presentes nas extremidades das sequências, caso haja sequência extra de remanescentes após o alinhamento. (NEEDLEMAN; WUNSCH, 1970)

$$S[i,j] = \max_{i,j\geq 1}\begin{Bmatrix} S[i-1,j-1] + f(a_i,b_j), \\ S[i-1,j] - gap, \\ S[i,j-1] - gap \end{Bmatrix} \quad (4)$$

A fórmula acima descreve o preenchimento da matriz no algoritmo de Programação Dinâmica Needleman-Wunsch.

Quando todas as posições da matriz forem preenchidas, a pontuação do alinhamento ótimo global será definida pelo valor do elemento $[n,m]$ da matriz de programação dinâmica, onde n e m representam o comprimento das sequências comparadas. (NEEDLEMAN; WUNSCH, 1970)

Para determinar um alinhamento ótimo das sequências a partir da matriz de pontuação, uma segunda etapa denominada *traceback* é usada. O *traceback* rastreia as posições na matriz de pontuação que contribuíram para a maior pontuação geral ser encontrada. (SMITH; WATERMAN, 1981)

O uso do método de programação dinâmica requer um sistema de pontuação para a comparação de pares de símbolos (nucleotídeos para sequências de DNA ou RNA, e aminoácidos para sequências de proteínas), e um sistema de penalidades para inserção de *gaps*. Uma vez que esses parâmetros forem definidos, o alinhamento resultante de duas sequências deve ser sempre o mesmo. As matrizes

de pontuação mais comumente usadas para alinhamentos de sequências de proteínas e nucleotídeos são as matrizes da família PAM e exclusivamente para sequências de proteínas as matrizes da família BLOSUM. (SMITH; WATERMAN, 1981)

2.5.3 Alinhamento Local: Algoritmo Smith-Waterman

Alinhamentos locais geralmente são mais significativos do que matches globais porque incluem padrões que são conservados nas sequências. Eles também podem ser usados em vez do algoritmo de Needleman-Wunsch para combinar duas sequências que podem ter uma região de *match* que é somente uma fração dos seus comprimentos, sequências que têm comprimentos diferentes, sequências que se sobrepõem, ou no caso de uma sequência ser um fragmento ou subsequência da outra. (MOUNT, 2004)

Na implementação do algoritmo de Smith-Waterman as regras para o cálculo dos valores da matriz de pontuação são um pouco diferentes pois inclui pontuação negativa para *mismatches*, e quando o valor de uma matriz de pontuação de programação dinâmica torna-se negativo, esse valor é definido como zero. Os alinhamentos são produzidos a partir de posições de maior pontuação na matriz de pontuação e seguem um caminho de rastreamento a partir dessas posições até onde a pontuação for zero. (MOUNT, 2004)

Nesse algoritmo a inicialização é feita da mesma forma que no Needleman-Wunsch, atribuindo zero ao primeiro elemento da matriz $[0,0]$. Porém, o preenchimento do restante da primeira linha $[0,j]$ e do restante da primeira coluna $[i,0]$ é feito atribuindo zero. (SMITH; WATERMAN, 1981)

O restante da matriz é preenchido da mesma forma que no algoritmo de Needleman-Wunsch, onde cada $[i,j]$ da matriz recebe o maior valor entre $[i-1,j-1]$ somado com a pontuação referente a um *match* ou *mismatch*, $[i-1,j]$ e $[i,j-1]$ somados com a penalidade por inserção de um *gap* nas sequências. O diferencial é que o algoritmo de Smith-Waterman não permite a inclusão de valores negativos na matriz, substituindo esses valores por zero. (SMITH; WATERMAN, 1981)

O algoritmo pode ser escrito em forma matemática como mostrado na figura abaixo:

$$S[i,j] = \max_{i,j\geq 1} \begin{Bmatrix} S[i-1,j-1] + f(a_i, b_j), \\ S[i-1,j] - gap, \\ S[i,j-1] - gap, \\ 0 \end{Bmatrix} \quad (5)$$

Essa fórmula descreve o preenchimento da matriz no algoritmo de Programação Dinâmica Smith-Waterman.

2.5.4 Alinhamentos resultantes de Algoritmos de Alinhamentos Global e Local

Apesar de um programa de computador baseado no algoritmo de alinhamento local de Smith-Waterman ser usado para a produção de um alinhamento ótimo, este recurso sozinho não assegura que um alinhamento local será produzido. A matriz de pontuação de *match* ou pontuações e *mismatch* e as penalidades de *gap* escolhidas também influenciam se um alinhamento local será obtido.

Do mesmo modo, um programa baseado no algoritmo de alinhamento global de Needleman-Wunsch pode também retornar um alinhamento local, dependendo do peso de *gaps* de extremidade e de outros parâmetros de pontuação. Muitas vezes, pode-se simplesmente inspecionar o alinhamento obtido para ver quantos *gaps* estão presentes.

Se as regiões de *match* são longas e cobrem a maior parte das sequências e, obviamente, depende da presença de muitos *gaps*, o alinhamento é global. Já um alinhamento local tende a ser mais curto e não inclui muitos *gaps.* (MOUNT, 2004)

Um alinhamento global atinge *matches* na maior parte da sequência, enquanto a pontuação negativa de *mismatches* e de penalidades de *gaps* é muito pequena, de modo a proporcionar um alinhamento de comprimento. Assim sendo, apenas o número de *matches* tem de ser proporcional ao comprimento.

Teremos um alinhamento global ao utilizarmos uma matriz de pontuação que dá a média da pontuação para cada posição alinhada e uma penalidade de *gap*

pequena o bastante para permitir a extensão do alinhamento de regiões mal combinadas. (MOUNT, 2004)

Um alinhamento local tem uma pontuação negativa *mismatches* e de penalidades de *gaps* escolhida para balancear a contagem de *matches* e prevenir o alinhamento crescente em regiões não coincidentes. A matriz de pontuação, neste caso dá um valor negativo para as posições de *match*, e a penalidade de *gap* será suficientemente grande para evitar *gaps* ao longo do alinhamento.

Teremos um alinhamento local quando a pontuação de alinhamento local de sequências aleatórias não aumentar proporcionalmente com o comprimento da sequência, porque a pontuação positiva de *match* é compensada pela pontuação de *mismatch* e de penalidades. (MOUNT, 2004)

Pequenas mudanças no sistema de pontuação podem mudar abruptamente um alinhamento de local para global. (MOUNT, 2004)

2.6 **Matrizes de Pontuação**

Matrizes de comparação analisam as substituições de aminoácidos ou nucleotídeos que ocorrem nos alinhamentos de sequências. Com bases nessas substituições, uma pontuação é atribuída a cada aminoácido ou nucleotídeo. Os dois tipos mais utilizados de matrizes são a PAM (*Point Accepted Mutation*) e a BLOSUM (*Blocks Substitution Matrix*). (MOUNT, 2004)

Quanto maior o valor da pontuação, maior é o número de nucleotídeos ou aminoácidos alinhados, e, por conseguinte, maior a similaridade destas sequências. (MOUNT, 2004)

2.6.1 PAM

As matrizes PAM (*Point Accepted Mutations* ou *Percent of Accepted Mutations*) são calculadas a partir da observação das diferenças em proteínas ou nucleotídeos proximamente relacionadas. A família de matrizes PAM foi

desenvolvida para comparar duas sequências separadas por uma distância especificada em unidades PAM, que corresponde a uma medida de evolução calculada por 100 aminoácidos ou nucleotídeos. Ou seja, se a sequência A e a sequência B divergem de uma unidade PAM se as mesmas se diferenciam em 100 aminoácidos ou nucleotídeos. (DAYHOFF; SCHWARTZ; ORCUTT, 1978).

A numeração na matriz PAM utilizada na comparação entre duas sequências indica em quantas unidades PAM elas estão separadas. A matriz é preenchida com as pontuações, onde cada elemento $[i, j]$ da matriz PAM representa a pontuação para substituir um aminoácido ou nucleotídeo a_i por um b_j. (DAYHOFF; SCHWARTZ; ORCUTT, 1978)

2.6.2 BLOSUM

As matrizes BLOSUM (*Blocks Amino Acid Substitution Matrices*) são usadas nos alinhamentos de sequências divergentes, ou seja, sequências de relacionamento distante. Essas matrizes são baseadas em blocos, representando as regiões de uma família de proteínas melhor conservadas. Esses blocos são obtidos pelo grupamento de sequências de proteínas relacionadas entre elas. (HENIKOFF; HENIKOFF, 1992)

A família de matrizes BLOSUM é baseada nas mudanças ocorridas em cada bloco. Dentro de cada bloco as sequências são agrupadas no mesmo grupo par a par se a porcentagem de identidade delas for semelhante a da matriz BLOSUM utilizada. Ou seja, se a porcentagem de identidade entre duas sequências for igual ou superior a $X\%$ e a matriz utilizada for a BLOSUM X, essas duas sequências serão agrupadas em um mesmo grupo dentro de determinado bloco. (MOUNT, 2004)

A matriz é preenchida com as pontuações, onde cada elemento $[i, j]$ da matriz BLOSUM representa a pontuação para substituir um aminoácido a_i por um b_j.

Quanto mais baixa a pontuação nas matrizes BLOSUM, esta será mais indicada para alinhamentos de sequências divergentes. (HENIKOFF; HENIKOFF, 1992)

2.6.3 Matrizes de Pontuação para Alinhamento de Sequências de Nucleotídeos

Através de análises de mutações feitas em moléculas de DNA foi comprovado que transições (substituições entre os nucleotídeos A e G - purinas - ou entre os nucleotídeos C e T - pirimidinas) são mais comuns que transversões (substituições de purinas por pirimidinas ou pirimidinas por purinas). (LI; GRAUR, 2000)

As matrizes de transição ou transversão podem ser usadas para produzir alinhamentos globais ou locais de sequências de DNA. Essas matrizes são usadas para pontuar alinhamentos de nucleotídeos. O principal benefício no uso dessas matrizes de pontuação é que elas são baseadas em um modelo evolucionário definido.

Também são utilizadas matrizes identidade como matrizes de pontuação para alinhamento de sequências de nucleotídeos.

3 **PROGRAMAÇÃO PARALELA**

A estratégia de programação paralela tem sido bastante utilizada para ganho de desempenho a partir da utilização de diversas unidades de processamento. No princípio, programas eram executados sequencialmente onde somente era utilizado um núcleo do processador. Essa estratégia, no decorrer do tempo, não conseguiu atender as demandas de novas aplicações que necessitavam de mais poder de processamento. Fez-se necessário investir em novas arquiteturas, as quais pudessem processar várias instruções ao mesmo tempo. A fim de que o potencial destas fosse completamente utilizado, se fez necessário o desenvolvimento de um novo paradigma de programação conhecido como programação paralela.

Neste capítulo serão abordados conceitos básicos das arquiteturas paralelas utilizadas para paralelização do algoritmo de Needleman-Wunsch neste trabalho, e também conceitos básicos do próprio paradigma de programação paralela.

3.1 **O advento da programação paralela**

Microprocessadores baseados em uma única unidade de processamento central (CPU) buscaram um aumento rápido de desempenho e redução de custos em aplicações de computadores por mais de duas décadas. Esta busca incansável por melhoria de desempenho permitiu que aplicações possuíssem mais funcionalidades, melhores interfaces de usuário e resultados mais úteis. Os usuários exigiram então ainda mais melhorias, criando um ciclo positivo para a indústria de computadores. (KIRK; HWU, 2010)

Embora os usuários continuassem a procurar computadores mais performáticos, a partir de 2003 a indústria de processadores atingiu o limite físico de números de transistores para fazer projetos que desempenhassem melhor sem aquecer operando em alta frequência. Esses problemas no consumo de energia e dissipação de calor limitaram o aumento da frequência de *clock* e a quantidade de atividades realizadas por ciclo de *clock* dentro de uma única CPU. Fornecedores de microprocessadores migraram então para modelos com múltiplos chips de

processamento, ou seja, múltiplos núcleos de processador utilizados em cada CPU, para aumentar a capacidade de processamento. (KIRK; HWU, 2010)

Desta forma, programas sequenciais, isto é, que executam de forma serializada em somente um núcleo do processador, não poderão usufruir das novas tecnologias desenvolvidas pelas fabricantes de processadores. Nos programas paralelos, nos quais múltiplas unidades processadoras (*threads* ou processos) cooperam para que mais tarefas sejam concluídas ao mesmo tempo, é possível melhorar o tempo de execução de uma determinada aplicação. (KIRK; HWU, 2010)

Com o passar do tempo foi estabelecido no mercado à procura por processadores com maior capacidade de paralelismo, isto é, mais núcleos. Desta forma a técnica de se programar em paralelo, a qual antes era restrita a alguns poucos programadores de alto desempenho, agora se torna uma prática a ser utilizada por todos os desenvolvedores de *software* e aplicações.

3.2 **O paradigma de programação paralela**

A programação paralela consiste em utilizar múltiplas unidades de processamento para executar partes de um mesmo programa em paralelo, ou seja, simultaneamente. (KIRK; HWU, 2010)

Os programas de computador originalmente foram escritos para a computação em série, ou seja, para ser executados em um computador contendo uma única unidade de processamento (CPU). Um único problema é dividido em instruções distintas, as quais eram executadas de forma sequencial por um processador.

Computação Paralela é o uso simultâneo de múltiplos recursos a fim de solucionar um problema computacional. Nesse caso, um problema é dividido em pequenas partes que possam ser solucionadas de forma segura ao mesmo tempo usando múltiplas unidades de processamentos (múltiplas CPU's, *threads*, máquinas ligadas em rede, entre outros). (KIRK; HWU, 2010)

O principal objetivo de se programar em paralelo é reduzir o tempo total de processamento, ou seja, o *wallclock time*. Com o uso de programação paralela atinge-se o aumento de desempenho de um único processador com múltiplos

núcleos, fazendo com que a velocidade de execução seja muito maior que uma programação sequencial. Dessa forma é garantido que as aplicações desfrutem de um aumento contínuo de velocidade em gerações futuras de *hardware*. (KIRK; HWU, 2010)

Computadores paralelos são melhor utilizados por aplicações que processem grandes quantidades de dados e possuam um número elevado de tarefas independentes. Entretanto deve-se buscar o melhor equilíbrio entre custos de sincronização e comunicação entre unidades processadoras, e o número de tarefas em um problema que podem ser efetivamente executadas em paralelo.

Problemas que são subdivididos em muitos problemas menores independentes possuirão um maior grau de paralelismo, isto é, número de tarefas capazes de serem executadas simultaneamente. Para suprir essa execução de pequenos problemas em paralelo será necessário um maior número de unidades processadoras, sejam estas, *threads* ou processos. Haverá então um maior *overhead* (gastos de recursos computacionais) para administração destas *threads* ou processos e comunicação entre diferentes unidades de processamento ou núcleos de processadores. Enquanto que problemas com menor quantidades de subproblemas possuirão menor grau de paralelismo, mas em compensação menos gastos com administração de unidades processadoras e comunicação entre processadores ou núcleo de processadores.

Deve-se também manter um equilíbrio entre a carga de trabalho para cada unidade processadora, de forma que algumas unidades de processamento não fiquem sobrecarregadas, enquanto outras ficam ociosas.

Um dos principais desafios é garantir a correta funcionalidade e confiabilidade, o que constitui um problema sutil em computação paralela. Com o modelo de programação paralela com paralelismo de dados, pode-se obter alto desempenho e alta confiabilidade em aplicações distintas. (KIRK; HWU, 2010)

3.3 **Conceitos básicos de programação paralela e *hardware***

3.3.1 Sincronização

A sincronização na programação paralela é necessária para coordenar a troca de informações entre tarefas, impedir casos como condição de corrida e atualização perdida. Isso pode consumir tempo de processamento, pois um processador terá que aguardar o término de execução de tarefas em outros processadores, causando a perda de *speedup* (aceleração).

Um bloqueio (*lock*) é o mecanismo básico para assegurar exclusão mútua. Um bloqueio de exclusão mútua garante que apenas uma *thread* ou um processo possa acessar uma região crítica (região do código que é acessada por vários processos ou *threads*) de cada vez. Se outra *thread* ou outro processo quiser acessar uma região crítica e esta estiver ocupada, então estes são bloqueados até que sejam notificados para tentarem novamente. (HERLIHY; SHAVIT, 2008) (TANENBAUM, 2007)

Um semáforo é uma generalização de bloqueios de exclusão mútua. Em vez de permitir que somente uma *thread* de cada vez possa acessar uma região crítica, um semáforo permite que x *threads* a façam, onde x é a capacidade determinada quando o semáforo é inicializado. No semáforo uma *thread* acessa um método para pedir permissão para acessar a região crítica, e outro método para anunciar que está deixando a região crítica. O semáforo em si é apenas um contador: ele mantém o controle do número de *threads* que receberam permissão para entrar. Se o semáforo está prestes a exceder a capacidade x, a *thread* de chamada está suspensa até que haja espaço. Quando uma *thread* sai da região crítica, ela chama um método para notificar a outra *thread* que estava em espera que agora há espaço. (HERLIHY; SHAVIT, 2008) (TANENBAUM, 2007)

Monitores são formas estruturadas de combinar sincronização e dados. Consistem em combinar métodos, exclusão mútua e objetos de condição. Essa sincronização é feita de forma modular através de uma fila com o seu próprio bloqueio interno, adquirido por cada método quando ele é chamado, e liberado quando ele retorna. Se uma *thread* tenta enfileirar um item para uma fila que já está cheia, então um método pode detectar o problema, suspender o chamador, e retomar o chamador quando a fila tiver espaço. (HERLIHY; SHAVIT, 2008)

O problema referente à sincronização de tarefas em um ambiente paralelo não é uma tarefa trivial de ser solucionada. Programadores comumente

superestimam as regiões sincronizadas de um programa, o que acarreta na perda de desempenho do mesmo.

3.3.2 Lei de Amdahl

A lei de Amdahl dita que o *speedup* S alcançado por uma melhoria na aplicação ou no *hardware* será limitado pela parte da aplicação que não foi melhorada. Em programação paralela, isso significa que para obter o melhor ganho possível em uma aplicação ou algoritmo, é necessário paralelizá-los como um todo. Esta lei é formalizada pela seguinte função matemática:

$$S(n) \leq \frac{1}{(1-p)+\frac{p}{n}} \quad (6)$$

Na fórmula acima (6), no contexto de programação paralela, p representa a porção do programa paralelizado, $(1-p)$ representa a porção do programa na forma sequencial e (p/n) é o ganho obtido pela paralelização da porção p utilizando n processadores. $S(n)$ representa o *speedup* máximo alcançado pela paralelização usando n processadores.

3.4 **Classificação de arquiteturas computacionais**

No início dos anos 70, Flynn (1972) desenvolveu uma taxonomia para classificar arquiteturas de computadores de acordo com o número de instruções distintas que estas podiam executar simultaneamente, e o número de elemento de dados que poderiam ser processados por elas. Este método de classificação é denominado de Taxonomia de Flynn.

Existem quatro classificações para arquiteturas computacionais baseadas nesse método:

1) SISD - *Single Instruction Single Data*: Consiste na arquitetura de computadores tradicionais antigos onde as instruções eram executadas de forma sequencial sobre um único dado.
2) SIMD - *Single Instruction Multiple Data*: Esta arquitetura envolve um único fluxo de instruções, porém sendo executada simultaneamente para diferentes elementos de dados. As arquiteturas baseadas em GPU utilizam-se deste modelo de arquitetura.
3) MIMD - *Multiple Instruction Multiple Data*: Consiste em diversos fluxos de instruções sendo executados em paralelo e processando elementos de dados. Arquiteturas de memória compartilhada, as quais representam os computadores *multicores* atuais, utilizam este modelo.
4) MISD - *Multiple Instruction Single Data*: Neste modelo vários fluxos de instruções são executados simultaneamente sobre um mesmo elemento de dados.

Apesar de ser um método relativamente antigo de classificação, este ainda pode ser utilizado para classificar arquiteturas computacionais no sentido macro.

Das classificações existentes nessa taxonomia, serão utilizadas neste trabalho arquiteturas que são classificadas como MIMD e SIMD para paralelização do algoritmo de Needleman-Wunsch.

3.4.1 Modelo SIMD - *Single Instruction Multiple Data*

O paradigma SIMD de programação paralela é baseado no conceito de uma mesma instrução sendo executada para diversos dados diferentes. Por exemplo, várias *threads* executarem uma soma de diferentes elementos.

A arquitetura SIMD é apropriada para aplicações que utilizam paralelismo orientado a instrução. Nessa arquitetura todas as unidades devem receber a mesma instrução no mesmo instante de tempo para que todas as unidades possam executá-la simultaneamente, ou seja, é síncrona. Utiliza o conceito de determinismo, ou seja, só existe uma tarefa sendo executada por todas as unidades em cada instante de tempo.

Por ter controle simplificado, essa arquitetura permite um grande número de unidades de processamento possibilitando que uma única instrução manipule muitos itens de dados em paralelo. Pode-se afirmar também que modelos SIMD possuem um único fluxo de instrução e múltiplos fluxos de dados. Um exemplo de sistema SIMD é quando se tem um co-processador em sistemas de alto desempenho em uma forma um pouco restrita, por exemplo, uma unidade de processamento gráfico (GPU). Outra subclasse dos sistemas SIMD são os processadores vetoriais (*vectorprocessors*), que por sua vez agem em vetores de dados semelhantes. Quando os dados podem ser manipulados por estas unidades de vetor, os processadores vetoriais os executam de uma forma quase paralela, mas apenas quando a execução está no modo vetorial. Neste caso, eles são mais rápidos do que quando executados em modo escalar convencional. (Netlib.org)

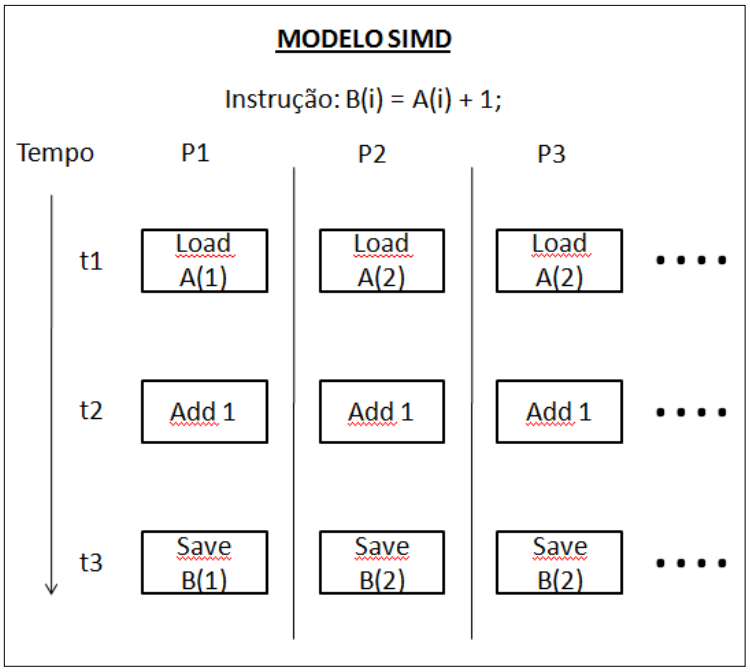


Figura 6: Exemplo do modelo SIMD para a instrução $B(I) = A(I) + 1$ sobre diferentes dados. Cada unidade de processamento P irá computar as mesmas instruções necessárias para resolver o problema.

3.4.2 Modelo MIMD - *Multiple Instruction Multiple Data*

O modelo MIMD pode ser usado para classificar arquiteturas de memória compartilhada, onde estas são compostas por um ou vários processadores *multicores* (processadores com mais de um núcleo) que compartilham a memória principal do sistema.

Máquinas MIMD (*Multiple Instruction Multiple Data*) são arquiteturas que suportam diferentes fluxos de instruções executando sobre diferentes dados de forma simultânea. Essa capacidade deve-se ao fato de que são construídas a partir de vários processadores ou núcleos independentes, cada qual com suas próprias unidades de controle e ALUs, podendo assim processar duas ou mais aplicações de forma paralela ou concorrente, e consequentemente encurtar o tempo de execução de uma aplicação. (Netlib.org)

Ao contrário do modelo SIMD, arquiteturas MIMD são normalmente assíncronas, isto é, cada processador pode executar diferentes instruções em tempos distintos dos demais. (PACHECO, 2011)

Essa definição deixa margem para que várias topologias de máquinas paralelas e de redes de computadores sejam enquadradas como MIMD. A diferenciação entre as diversas topologias MIMD é feita pelo tipo de organização da memória principal, memória *cache* e rede de interconexão. (Netlib.org)

Esta arquitetura pode ser melhor utilizada quando a natureza do problema permite que instruções diferentes sejam executadas simultaneamente para dados diferentes.

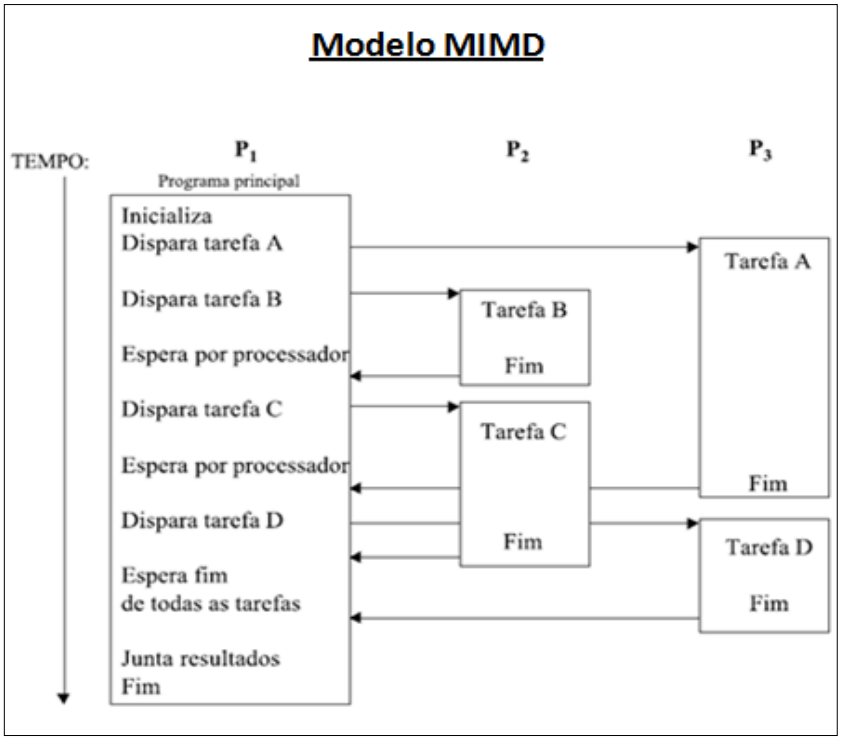


Figura 7: Modelo de divisão de tarefas usando paradigma MIMD.

Fonte: Simone de Lima Martins; 2002.

3.5 **Arquitetura paralelas utilizadas neste trabalho**

Desde 2003, a indústria de semicondutores tem investido em duas principais trajetórias para a concepção de microprocessadores *multicores*. (KIRK; HWU, 2010)

A trajetória *multicore* de CPUs procura manter a velocidade de execução de programas sequenciais utilizando vários núcleos. Inicialmente havia dois núcleos por processador, e esse número tem dobrado a cada geração. (KIRK; HWU, 2010)

A trajetória *multicore* de GPUs procura manter o maior *throughput* de execução possível em aplicações paralelas, ou seja, mais instruções executadas em um determinado período. (KIRK; HWU, 2010)

Enquanto a melhoria de desempenho de propósito geral de *multicores* CPUs tenha abrandado significativamente, os *multicores* GPUs têm melhorado continuamente. (KIRK; HWU, 2010)

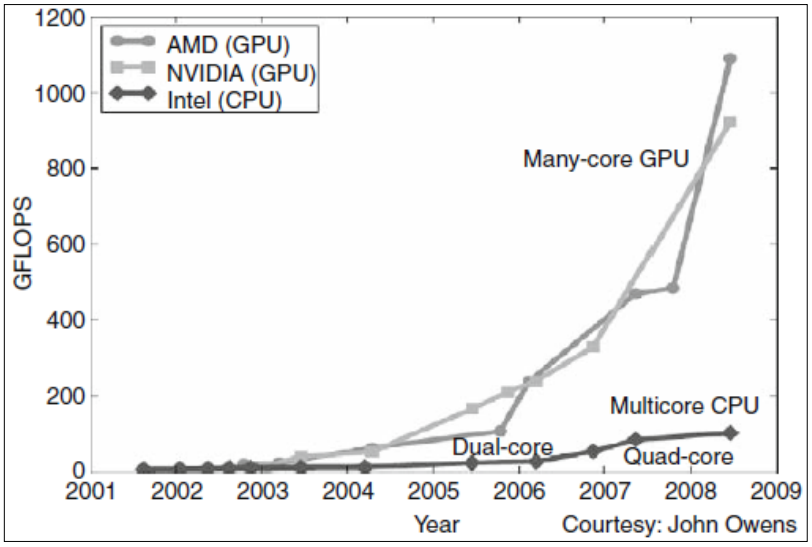


Figura 8: Diferença de desempenho entre GPUs e CPUs ao longo dos anos.
Fonte: KIRK; HWU, 2010.

Neste trabalho essas duas arquiteturas são utilizadas para implementar versões paralelas do algoritmo de Needleman-Wunsch.

3.5.1 Arquitetura de memória compartilhada com processadores *multicores*

Em sistemas de memória compartilhada uma coleção de processadores autônomos estão conectados a um sistema de memória, e cada processador consegue acessar locais diferentes deste sistema. Por este motivo a comunicação entre processadores é efetuada com base em operações implícitas em estruturas de dados compartilhadas.

No caso de ambientes de memória compartilhada que usam um ou mais *chips multicore*, o sistema de memória compartilhado é a própria memória RAM do computador conectados às CPUs da máquina geralmente por um barramento de dados.

Núcleos em *chips* diferentes se comunicam através de operações de *load e store* na memória principal da máquina, enquanto núcleos de um mesmo *chip* se comunicam através de memórias *caches* que se localizam próximos a estes na CPU.

Protocolos de coerência são então aplicados para manter a integridade dos dados nestas memórias.

Memórias *caches*, por possuírem maior velocidade de acesso do que a memória RAM, são utilizadas também para reduzir a latência de acesso à dados das aplicações em execução.

O design de uma CPU *multicore* é otimizado para melhor desempenho de códigos serializados. A CPU faz uso de um controle lógico sofisticado para permitir instruções de uma única *thread* executar em paralelo ou ainda fora da sua ordem sequencial enquanto mantém a aparência de uma execução tradicional. (KIRK; HWU, 2010)

3.5.2 Arquitetura baseada em processadores gráficos NVIDIA (GPU CUDA)

A arquitetura de uma GPU moderna é organizada em uma *matriz* de multiprocessadores de *streaming* (SMs) com capacidade de execução de muitas *threads*. O número de SMs pode variar conforme a geração da GPU. Cada SM tem um número de processadores de streaming (SPs ou *CUDA cores*) que compartilham a lógica de controle e instruções. (KIRK; HWU, 2010)

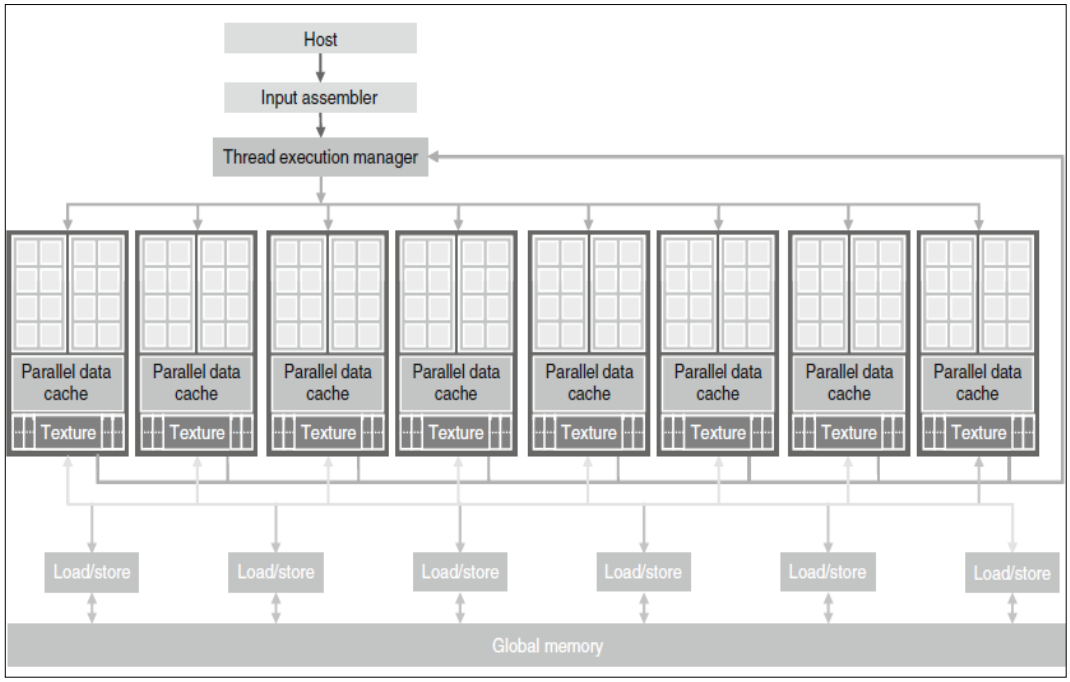


Figura 9: Exemplo de arquitetura CUDA GPU com 8 SMs que por sua vez possuem 16 CUDA cores cada um.
Fonte: KIRK; HWU, 2010.

Com uma GPU que possua a configuração esquematizada na figura 11 seria possível executar efetivamente em paralelo 128 *threads*, porém de forma concorrente seria possível executar um número de *threads* muito mais elevado. Isso acontece pois quando *threads* efetuam um acesso de grande latência, como no caso de acessar a memória global, elas são colocadas em espera até receberem todos os dados necessários para prosseguir. Enquanto os dados são lidos pelas *threads*, novas *threads* ganham direito de usar os CUDA *cores* e podem prosseguir com suas execuções. O agente escalonador de *threads* responsável por agendar a execução das mesmas é denominado de *warp scheduler*.

As CUDA *threads* são hierarquicamente organizadas em blocos de uma, duas ou três dimensões. Pode-se pensar que blocos de *threads* disputam acesso aos SPs, enquanto *threads* disputam acesso aos *cuda cores.*

Uma placa gráfica NVIDIA possui um sistema próprio de memória separado da memória principal utilizada pelos processadores tradicionais. Os tipos de memória mais comuns em uma GPU NVIDIA são:

1) Memória Global: Esta memória possui maior capacidade de armazenamento, porém seu tempo de acesso é muito elevado. Nesta

memória são alocados pela CPU todos os dados necessários para computar um problema a ser resolvido na GPU. Todas as *threads* instanciadas durante a execução possuem acesso de leitura e escrita a essa memória.

2) Memória Compartilhada: Esta unidade de armazenamento possui uma capacidade muito inferior a encontrada em memórias globais, entretanto seu tempo de acesso é muito mais rápido que da memória global, pois memórias compartilhadas são *caches* que estão localizadas dentro dos chips SMs. Essas memórias podem ser acessadas por todas as *threads* de um determinado bloco, tanto para leitura como para escrita.
3) Registradores: Estas unidades de armazenamento possuem tempo de acesso muito baixo e capacidade de armazenamento reduzida. Dados armazenados nessas unidades são copiados para cada *thread* instanciada em uma execução na GPU, isto é, cada *thread* possuirá uma cópia de determinado dado.
4) Memória Constante: Esta memória possui tempo de acesso inferior ao da memória global, porém a GPU não tem permissão para escrever nessa memória, possui somente acesso de leitura. A tarefa de escrever nesta memória é atribuída à CPU. Todas as *threads* instanciadas possuem acesso de leitura a essa memória.
5) Memória de textura: Memórias de textura são semelhantes às memórias constantes, ambas só podem ser lidas por *threads* na GPU e possuem tempo de acesso inferior ao de uma memória global.

A GPU otimiza o número de instruções executadas em um determinado período com uso de um grande número de *threads*. O *hardware* tira proveito de um grande número de *threads* para executar outras instruções quando algumas *threads* estão à espera de acessos de longa latência de memória, minimizando o controle lógico exigido para cada *thread* de execução. Pequenas memórias *cache* são fornecidas para ajudar a controlar os requisitos de largura de banda desses aplicativos permitindo que várias *threads* que acessam os mesmos dados da memória não precisem acessar a memória global. Como resultado, uma área muito maior do chip é dedicada aos cálculos de ponto flutuante. (KIRK; HWU, 2010)

3.6 **Linguagens e APIs utilizadas neste trabalho**

Este trabalho propõe a implementação do algoritmo NW em duas arquiteturas paralelas distintas: processadores *multicores* com memória compartilhada e processadores gráficos.

Para poder manipular a arquitetura de memória compartilhada foi utilizada a API OpenMP. No caso da paralelização do algoritmo em processadores gráficos da fabricante NVIDIA foi utilizada a linguagem de programação CUDA.

Nesta seção a API OpenMP e a linguagem CUDA são brevemente descritas.

3.6.1 API OpenMP

OpenMP é uma API para se programar em um ambiente de memória compartilhada usando paradigma MIMD de programação paralela. Essa API é projetada para sistemas em que cada *thread* pode ter acesso a toda a memória disponível, ou seja, o sistema com OpenMP é como uma coleção de núcleos com acesso a memória principal. (PACHECO, 2011)

OpenMP permite que o programador indique o bloco de código a ser executado em paralelo através de diretivas, enquanto as funções e *threads* que devem executá-lo são determinadas pelo compilador e pelo sistema em tempo de execução.O objetivo do OpenMP é fornecer uma API padrão e portátil para escrever programas paralelos de memória compartilhada. (PACHECO, 2011) (CHANDRA et al., 2001)

Essa API é dividida em três categorias: estruturas de controles paralelos, ambientes de dados e sincronização. (CHANDRA et al., 2001)

Estruturas de controle são construtores que são capazes de alterar o fluxo de controle em um programa. O OpenMP tem dois tipos de construtores: um que provê a criação de múltiplas *threads* de execução que podem ser executadas

concorrentemente e outro que provê a divisão de tarefas simultaneamente entre threads existentes. (CHANDRA et al., 2001)

O ambiente de dados do OpenMP é associado com uma única *thread* de controle, a chamada *master thread*. A comunicação entre threads em OpenMP é representada por operações de leitura e escrita nas variáveis compartilhadas do programa. A sincronização entre *threads* é feita através de exclusão mútua e sincronização de eventos. A exclusão mútua de construtores é usada para controlar o acesso de uma variável compartilhada por uma *thread*. Essa sincronização é feita de forma que uma só uma *thread* tenha acesso exclusivo a uma variável compartilhada durante a execução da aplicação. (CHANDRA et al., 2001)

3.6.2 A linguagem CUDA - *Computer Unified Device Architecture*

Um programa CUDA consiste em uma ou mais fases que são executadas em um *host* (CPU) ou um *device* (GPU - Unidade de Processamento Gráfico). As fases que exibem quase nenhum paralelismo de dados são implementadas em um código que irá ser processado pela CPU. As fases que apresentam grande quantidade de paralelismo de dados são implementados em um código que irá ser processado pela GPU através de funções denominadas *kernels*.

Paralelismo de dados refere-se à propriedade do programa pelo qual muitas operações aritméticas podem ser realizadas com segurança em estruturas de dados de forma simultânea, o que torna a execução muito mais rápida. (KIRK; HWU, 2010)

O compilador C da NVIDIA (NVCC) separa o código da CPU e o código da GPU durante o processo de compilação. O código da GPU é escrito usando ANSI C estendida com palavras-chave para a rotulagem de funções de dados paralelos chamado de *kernels*, assim como suas estruturas de dados associadas. Os *kernels* normalmente geram um grande número de *threads* para explorar paralelismo de dados. As *threads* de CUDA são mais rápidas que as *threads* da CPU, levando menos ciclos de *clock* para executar suas instruções. (KIRK; HWU, 2010)

A execução de um programa em CUDA começa com a execução no host (CPU). Quando uma função de *kernel* é chamada ou iniciada, a execução é

transferida para um dispositivo (GPU), onde um grande número de *threads* é gerado para tirar vantagem do paralelismo de dados. Todas as *threads* que são geradas durante a execução de um *kernel* são chamadas coletivamente de um *grid*. Quando todas as *threads* de um *kernel* terminam sua execução, o *grid* correspondente termina, e a execução continua no *host* até que outro *kernel* seja chamado. Todas as *threads* em um *grid* são organizadas em blocos. Cada *thread* e blocos possuem determinados índices para permitirem sua identificação durante a execução de um *kernel*. (KIRK; HWU, 2010)

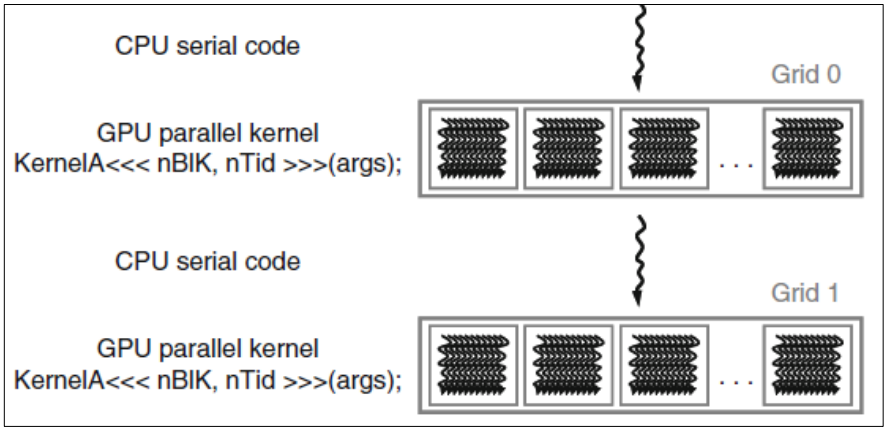


Figura 10: Exemplo de aplicação CUDA em execução. CPU computa serialmente parte do problema com pouco paralelismo, depois copia os dados para GPU. A GPU inicia o kernel processando a mesma instrução em paralelo para diferentes dados. Por fim a CPU copia os dados da memória da GPU e a disponibiliza para uma próxima etapa.
Fonte: KIRK; HWU, 2010.

Para que um *kernel* seja executado em um *device*, o programador precisa alocar memória e transferir os dados da memória do *host* (CPU) para a memória do *device* (GPU). O mesmo ocorre quando um *kernel* termina sua execução: os dados da memória da GPU precisam ser transferidos para a memória da CPU, e então

Tabela 1: Tabela de permissão dos códigos do *device* e do *host* no modelo de memória do CUDA *device*.

deve ser efetuada a liberação da memória da GPU.

Permissão	*Código do device (GPU)*	*Código do host (CPU)*
Leitura/Escrita nos registradores por thread	✔	✘
Leitura/Escrita na memória local por thread	✔	✘
Leitura/Escrita na memória compartilhada por blocos	✔	✘
Leitura/Escrita na memória global por grid	✔	✘
Leitura na memória constante por grid	✔	✘
Transferência de dados entre as memórias global e constante por grid	✘	✔

As memórias do *host* e do *device* são separadas. Cada *host* pode invocar um determinado número de *kernels,* e cada kernel é invocado com um número fixo de blocos e *threads.* Cada bloco possui *threads* que cooperam entre si, sincronizando suas execuções e compartilhando dados usando a memória compartilhada, a qual possui uma latência muito inferior a da memória global.

A) B)

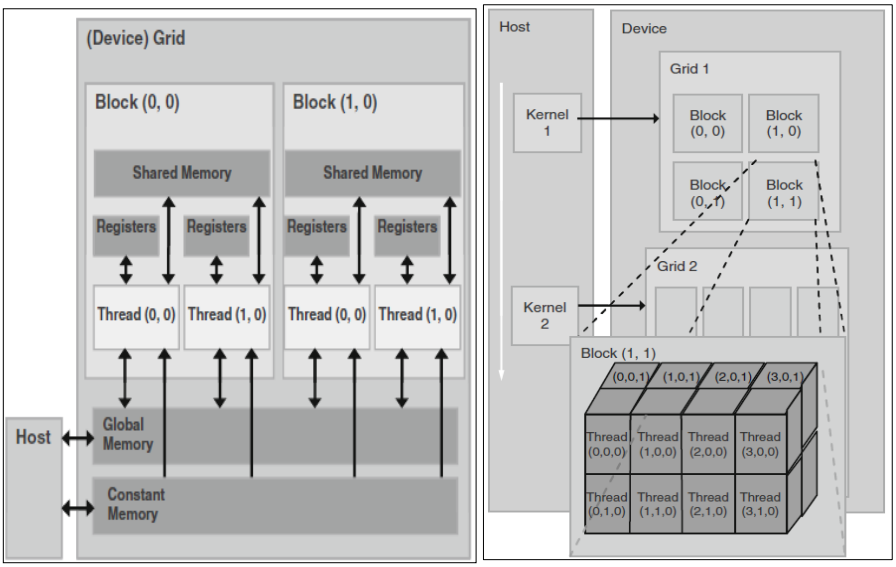


Figura 11: Exemplo do modelo de memória e organização de threads de uma GPU NVIDIA. A) Organização do acesso às memórias da GPU pelas threads. B) Organização tridimensional de threads por blocos em um grid.

Fonte: KIRK; HWU, 2010.

O modelo de memória CUDA possui funções em uma API para auxiliar o manuseio de dados nas memórias. A função *cudaMalloc()* aloca objetos na memória global da GPU (*device*). Ela possui dois parâmetros: endereço de um ponteiro (para alocar o objeto) e tamanho (do objeto alocado em bytes). A função *cudaFree()* libera objetos na memória global da GPU. Analogamente, existem as funções *cudaHostMalloc()* e *cudaFreeHost()*, as quais possibilitam alocar na memória do *host* páginas fixas que não sofreram nenhum tipo de realocação por parte do SO. (SANDERS; KANDROT, 2010)

Existe também a função *cudaMemcpy()* que é responsável pela transferência de dados entre *hosts* (CPUs), entre *devices* (GPUs) ou entre *host* e *device* (CPU - GPU) de forma síncrona. A função *cudaMemcpyAsync()* possui a mesma funcionalidade de cópia de dados entre as memórias do sistema, porém ela o faz de forma assíncrona, isto é, enquanto os dados estão sendo copiados outras atividades estão sendo efetuadas pelo *kernel*. (SANDERS; KANDROT, 2010)

4 **PARALELIZAÇÃO DO ALGORITMO NEEDLEMAN-WUNSCH**

Alinhamento global de sequências é uma ferramenta de análise muito utilizada no campo da Bioinformática. Milhões de alinhamentos são feitos diariamente ao redor do mundo. Para que se supra a necessidade do mundo acadêmico, computadores mais potentes e algoritmos melhores desenvolvidos e implementados devem ser criados. Paradigmas de programação paralela têm sido vastamente utilizados para a melhora desses algoritmos.

Neste capitulo abordaremos o algoritmo de alinhamento global descrito por Needleman-Wunsch apresentando uma visão detalhada sobre este, trabalhos anteriores utilizando arquiteturas diversas para paralelizá-lo, e as estratégias utilizadas para as implementações propostas nesse estudo.

4.1 **Uma visão mais detalhada do algoritmo**

A tarefa de alinhar pares de sequências consiste em comparar duas sequências, a fim de que se possa encontrar um trecho comum a elas, possivelmente introduzindo espaços conhecidos como *gaps* nessas sequências. Caso os elementos das duas sequências em uma mesma posição sejam iguais, ocorre um *match* (acerto), caso contrário, ocorre um *mismatch* (erro). (MOUNT, 2004)

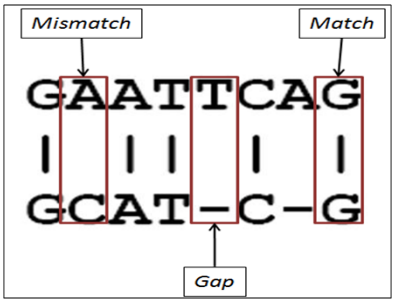


Figura 12: Exemplo de um alinhamento de uma pequena sequência genética.

Alinhamento de sequências biológicas é um importante e fundamental problema na área de bioinformática e constitui uma ferramenta crítica de suporte a pesquisas no campo da biologia molecular. Algoritmos de alinhamento de sequências biológicas tipicamente fazem uso de técnicas de programação dinâmica, na qual uma ou mais tabelas de tamanho $[(m+1) \times (n+1)]$ são preenchidas, onde m e n representam o comprimento das duas sequências a serem alinhadas. (RAJKOY; ALURU, 2004)

Existem dois métodos de alinhamento de pares de sequências, global e local. O alinhamento global compara as duas cadeias como um todo, enquanto o alinhamento local analisa somente trechos similares das duas sequências. (MOUNT, 2004)

O primeiro algoritmo a tratar esse problema de forma a conseguir o melhor alinhamento global exato de duas sequências se chama Needleman-Wunsch. (RAJKOY; ALURU, 2004)

Formalizado e criado pelos doutores Saul B. Needleman e Christian D. Wunsch, este algoritmo de programação dinâmica consiste em preencher uma matriz bidimensional de tamanho $[(m+1) \times (n+1)]$ com a pontuação de todos os possíveis alinhamentos entre um par de sequências A e B, onde m e n representam o comprimento da cadeia A e B respectivamente. Essa matriz é conhecida como matriz de comparações ou matriz de programação dinâmica. (NEEDLEMAN; WUNSCH; 1970)

Para efetuar o preenchimento da matriz de comparações, o algoritmo se utiliza de uma fórmula recursiva onde cada elemento da matriz é preenchido com a pontuação de um de três elementos adjacentes à célula corrente da matriz somado com uma penalidade ou bônus, excluindo-se a primeira linha e coluna, as quais são inicializadas no início da execução do algoritmo. (NEEDLEMAN; WUNSCH, 1970)

A matriz de comparações ou similaridades é inicializada com o elemento $[0,0]$ recebendo o valor 0, a primeira linha recebe os valores da penalidade pela inserção de espaços (*gaps*) na sequência representada pelas colunas da matriz de comparações, analogamente a primeira coluna receberá os valores das penalidades pela inserção de espaços na sequência representada pelas linhas da matriz de comparações. (NEEDLEMAN; WUNSCH, 1970)

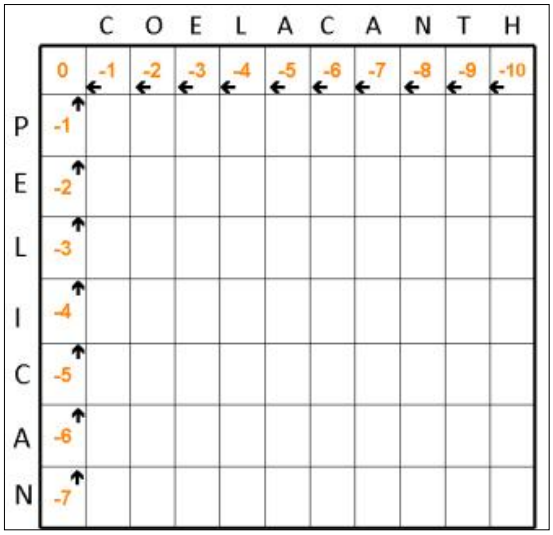


Figura 13: Exemplo de matriz de comparações inicializada para alinhar duas sequências de caracteres. Neste caso a penalidade pela inserção de um gap corresponde (a-1).
Fonte: http://www.nature.com/scitable/topicpage/basic-local-alignment-search-tool-blast-29096 (Acessado em 21/04/2013)

A entrada $[i,j]$ na matriz de programação dinâmica do algoritmo depende dos valores $[i-1,j-1]$, $[i,j-1]$, $[i-1,j]$. No caso do elemento $[i-1,j-1]$, este é somado com um bônus caso o símbolo de uma sequência seja igual ao simbolo de outra sequência (*match*), ou poderá receber uma penalidade se os símbolos forem diferentes (*mismatch*). Os elementos $[i-1,j]$ e $[i,j-1]$ são somados à uma penalidade chamada *gap* – inserção de um espaço em uma das sequências alinhadas. A atribuição do valor do elemento $[i,j]$ se dá pelo valor máximo obtido pela aplicação das operações descritas acima para cada elemento $[i-1,j-1]$, $[i-1,j]$ e $[i,j-1]$. (NEEDLEMAN; WUNSCH, 1970)

$$SM[i,j] = \max\begin{Bmatrix} SM[i,j-1]+gp, \\ SM[i-1,j-1]+ss, \\ SM[i-1,j]+gp \end{Bmatrix} \quad (7)$$

A fórmula acima é utilizada para preencher células da matriz conforme descrito no parágrafo anterior. A sigla gp corresponde à penalidade de inserção de um *gap* e ss é a pontuação associada a um determinado *match* ou *mismatch*.

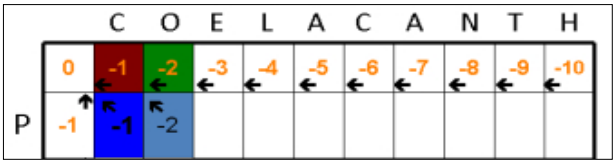


Figura 14: Preenchimento da matriz de comparações para o exemplo da figura 13, utilizando fórmula acima (7).
Fonte: http://www.nature.com/scitable/topicpage/basic-local-alignment-search-tool-blast-29096 (Acessado em 21/04/2013)

O segundo passo desse algoritmo efetua um procedimento na matriz denominado *traceback*, cujo objetivo é determinar o alinhamento ótimo percorrendo um caminho na matriz de comparações. O ponto inicial deste caminho se localiza na célula $[n, m]$, onde n e m representam o comprimento das duas sequências a serem comparadas. A partir desta célula é verificado de acordo com a fórmula recursiva apresentada acima, de qual dos três elementos adjacentes à célula atual recebeu seu valor. Se o valor da célula $[i, j]$ for proveniente do elemento $[i-1, j-1]$, significa que o símbolo na posição i da sequência representada pelas linhas da matriz de comparações deve ser escrito acima do elemento j da sequência representada pelas colunas. Caso a célula atual tenha seu valor proveniente do elemento $[i-1, j]$, é inserido um espaço vazio na sequência representada pelas colunas, de forma análoga se o valor recebido pela célula atual for proveniente do elemento $[i, j-1]$, um espaço vazio é inserido na sequência representada pelas linhas da matriz de comparações. (NEEDLEMAN; WUNSCH, 1970)

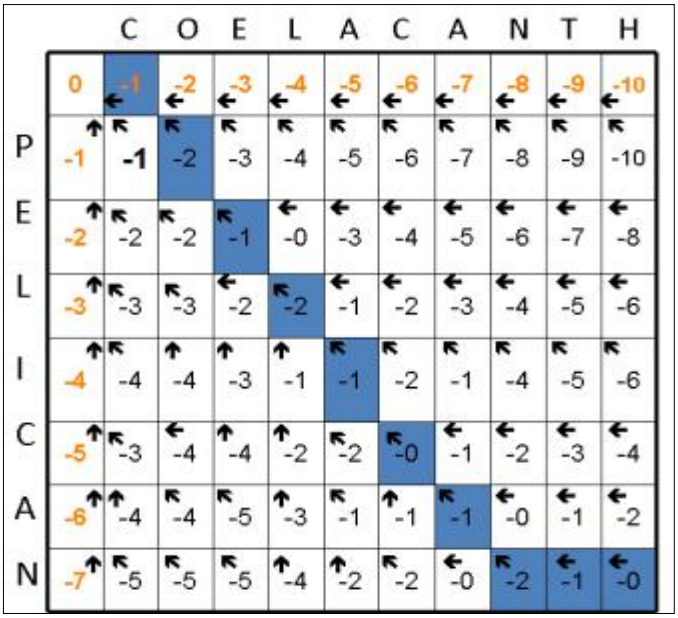


Figura 15: Procedimento de Traceback para determinar o melhor alinhamento global.
Fonte: http://www.nature.com/scitable/topicpage/basic-local-alignment-search-tool-blast-29096 (Acessado em 21/04/2013)

Durante o procedimento de *traceback* pode ser que haja empates sobre qual elemento da matriz deve ser incluído no caminho para encontra o alinhamento ótimo, neste caso um caminho é escolhido arbitrariamente, desta forma a matriz de programação dinâmica preenchida pela fórmula recursiva do algoritmo Needleman-Wunsch apresenta vários alinhamentos ótimos, porém somente retorna um resultado, isto é, um alinhamento ótimo. (MOUNT, 2004)

Para o exemplo proposto na figura 14 o alinhamento global ótimo seria:
C O E L A C A N T H
_ P E LI C A N _ _

O algoritmo de alinhamento global possui uma complexidade de espaço $O(nm)$, pois o mesmo precisa criar uma matriz com todos os alinhamentos possíveis entre cada subsequência pertencentes às duas sequências a serem comparadas. Sua complexidade de tempo também é $O(nm)$, pois será necessário preencher a

matriz $[n \times m]$ toda, a fim de que se possa descobrir o alinhamento ótimo. (NEEDLEMAN; WUNSCH, 1970)

A técnica apresentada por Needleman-Wunsch em seu algoritmo para alinhar cadeias de caracteres, originalmente utiliza o valor de $+1$ caso ocorra um *match* e 0 caso ocorra um *mismatch*. Esse esquema de pontuação pode ser representado pela matriz identidade $[4 \times 4]$ quando são alinhadas duas sequências de nucleotídeos, ou $[20 \times 20]$ quando são alinhadas duas sequências de aminoácidos. (ZOMAYA, 2006)

O algoritmo de alinhamento global criado por Needleman-Wunsch oferece uma técnica exata para encontrar alinhamentos ótimos entre duas sequências de nucleotídeos ou aminoácidos utilizando matrizes de pontuação. Devido ao seu grande custo de espaço e tempo por conta de sua complexidade $O(nm)$, algoritmos heurísticos como o BLAST são empregados. (ZOMAYA, 2006)

O algoritmo BLAST, muito utilizado em banco de dados biológicos para encontrar semelhanças entre sequências consultadas e sequências armazenadas no banco de dados, faz uso de heurísticas para reduzir seu consumo de tempo. Dessa forma, a precisão de seus resultados é pior que os produzidos por métodos exatos, como pelo algoritmo de Needleman-Wunsch. (BEZERRA; MAGALHÃES; 2006)

4.2 **Desafios para paralelização do algoritmo Needleman-Wunsch**

A Bioinformática é uma área de estudo que cada vez mais necessita dos esforços de uma variedade de áreas científicas para descobrir novas formas de organizar, armazenar, manipular, analisar e interpretar a crescente quantidade de informações biológicas geradas pelo método científico de análise de genomas. Ela necessita do desenvolvimento de ferramentas de *software* para administrar a enorme quantidade de dados provenientes do sequenciamento de genomas e proteínas. Também envolve o desenvolvimento de aplicações revolucionárias e algoritmos inovadores capazes de analisar esta grande massa de dados e fornecer conclusões sobre os estudos biológicos de forma sucinta. (ZOMAYA, 2006)

A grande maioria dos estudos feitos pelo campo da Bioinformática exige o uso de supercomputadores com capacidades muito além do esperado em simples

computadores pessoais e novos paradigmas de programação para analisar de forma otimizada os dados gerados pelas pesquisas genômicas. Isto se torna verdade já que, atualmente, quantidades massivas de dados biológicos são geradas diariamente. Portanto, é esperado que o paradigma de programação paralela aliado com sistemas computacionais de alta performance tenham um papel decisivo em assistir à comunidade científica no campo da Bioinformática a explorar possibilidades que, em um passado não muito distante, pareciam impossíveis. (ZOMAYA, 2006)

Estes desafios gerados pelos emergentes campos da Bioinformática e Biologia computacional possibilitam uma nova gama de problemas computacionais a serem resolvidos por cientistas da Computação. (ZOMAYA, 2006)

Muitos esforços têm sido feitos para agregar informações de sequências biológicas em banco de dados públicos. O grande problema está no fato desses repositórios biológicos serem demasiadamente grandes e atualmente apresentarem taxas de crescimento exponencial. Portanto comparação de sequências biológicas é um dos problemas mais importantes e básicos no campo da Biologia computacional e Bioinformática, dado que o número e diversidade de sequências biológicas são altíssimos, e também o número de vezes que alinhamentos precisam ser resolvidos diariamente em todo o mundo. (ZOMAYA, 2006)

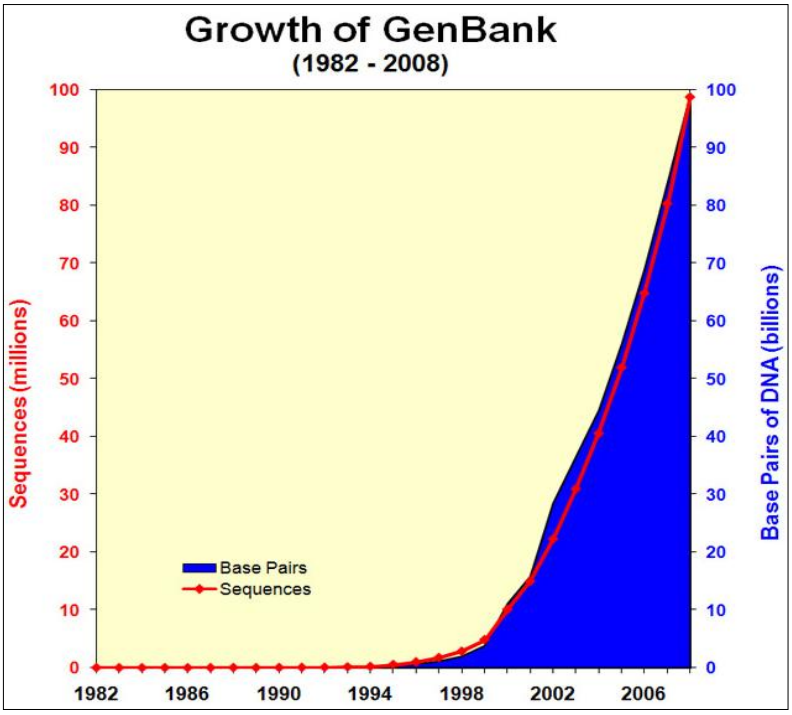


Figura 16: Crescimento do GENBANK mensurado até 2008. Fonte: http://www.ncbi.nlm.nih.gov/genbank/genbankstats-2008/ (Acessado em 21/04/2013)

O algoritmo de alinhamento global desenvolvido por Needleman-Wunsch, descrito em detalhes na seção anterior, possui complexidade de tempo e espaço $O(nm)$, o que o torna inviável para comparação de grandes genomas. Nesse cenário seria necessário alocar uma matriz $[(m+1) \times (n+1)]$ em memória, e se for considerado uma implementação do algoritmo em um paradigma sequencial, o tempo para retornar um alinhamento ótimo seria muito grande. (RAJKOY; ALURU; 2004)

Tabela 2: Relação de espaço em memória necessário para alocar uma matriz $[(n+1) \times (n+1)]$ e quantidade de nucleotídeos em uma amostra. Para esta tabela foi considerado que cada elemento da matriz é um inteiro de 4 bytes.

Nucleotídeos	***Memória (Gb)***
10000	0,37
40000	5,96
160000	95,37
640000	1.525,88

Por esta razão variações desse algoritmo foram propostas, cujo objetivo principal é reduzir o custo de tempo e espaço gasto pelo alinhamento de sequências. Um algoritmo com a mesma aplicação geral que execute em uma complexidade de tempo menor que o de NW (Needleman-Wunsch) ainda não é conhecido. (ZOMAYA, 2006)

D.S Hirschberg (1975) propôs um algoritmo exato que calcula a maior subsequência comum a duas sequências distintas usando complexidade quadrática de tempo, porém, complexidade linear de espaço. O algoritmo de Hirschberg consiste em uma modificação inteligente do algoritmo de NW. Para Hirschberg, o problema a ser resolvido pelo algoritmo de NW era quebrado em problemas menores, estes eram então quebrados em problemas ainda menores recursivamente, caracterizando assim uma solução por divisão e conquista. Isto era feito dividindo as sequências pela metade e as comparando para se encontrar em qual partição estava o caminho a ser seguido para encontrar a maior subsequência comum. No entanto, este método dobra o consumo de tempo no pior caso. (ZOMAYA, 2006) (HIRSCHBERG, 1975)

O método proposto por Fickett (1984) é uma modificação do algoritmo de NW exclusivo para sequências similares. Neste caso, pode-se inferir que o melhor alinhamento será a principal diagonal, caso não seja inserido nenhum espaço vazio (*gap*) em nenhuma das duas sequencias comparadas. Entretanto, se um *gap* for inserido em uma das sequências, por elas serem similares o alinhamento ótimo estará próximo a diagonal principal. Então, para se encontrar o melhor alinhamento é necessário somente guardar em memória uma pequena faixa próxima a diagonal principal da matriz de comparações. Este algoritmo possui complexidade $O(kn)$, onde k é a distância entre o elemento da sequência a ser comparada e o elemento que compõe a diagonal principal na mesma linha. (ZOMAYA, 2006) (FICKETT, 1984)

Para se criar uma versão paralelizada do algoritmo de NW deve-se primeiro considerar os três pontos principais, que para Kirk e Hwu (2010) são: resolver um dado problema em menos tempo, resolver problemas maiores em um tempo aceitável, e alcançar melhores soluções para um dado problema em um período de tempo razoável. Na prática, o uso de programação paralela pode ser proveniente da combinação desses três pontos.

Aplicações que são boas candidatas para computação paralela tipicamente estão relacionadas a problemas de grande porte e alta complexidade de modelagem. Isto é, essas aplicações provavelmente processam grande quantidade de dados, fazem uso de um número elevado de iterações, ou até mesmos os dois. Para tal problema ser resolvido com o paradigma de computação paralela, o problema deve ser reformulado de forma a ser decomposto em problemas menores que possam ser computados ao mesmo tempo de forma segura. (KIRK; HWU, 2010)

No caso do algoritmo NW, é possível notar que devido ao seu grande consumo de memória $O(nm)$ gerado pela matriz de comparações, a qual é originada devido a longas sequências de nucleotídeos ou aminoácidos, é um algoritmo que processa bastante quantidade de dados. Como sua complexidade de tempo envolve $O(nm)$, pois se faz necessário preencher a matriz de comparações elemento por elemento, e posteriormente, achar o melhor caminho nessa matriz para que se descubra o alinhamento ótimo, pode-se considerar que este algoritmo também possui um número elevado de iterações, tornando-o assim um excelente candidato para paralelização.

Encontrar paralelismo em problemas computacionais de grande porte, em sua grande maioria, é conceitualmente simples, mas pode se tornar desafiador na prática. O objetivo principal é identificar o trabalho que pode ser feito por cada unidade de execução paralela, esta podendo ser uma *thread* ou até mesmo um processador, para que o paralelismo inerente do problema em questão seja bem utilizado. (KIRK; HWU, 2010)

O padrão de acesso apresentado pelo cálculo da matriz de comparações feito pelo algoritmo de programação dinâmica NW é não uniforme e já foi exaustivamente estudado na literatura sobre programação paralela relacionada à Bioinformática. (ZOMAYA, 2006)

A maioria dos algoritmos de alinhamento de sequências implementados em paralelo preenche a matriz de comparações por diagonais, pois os elementos requeridos para se preencher uma diagonal estão presentes nas duas diagonais anteriores. Como não existem dependências entre os elementos de uma mesma diagonal, estes podem ser computados em paralelo. Quando a n-ésima diagonal for calculada, onde n representa o comprimento da maior sequência, o grau de paralelismo máximo será alcançado.

A este método de processamento da matriz de comparações se dá o nome de *wavefront method*. Este nome é proveniente da forma que a matriz é preenchida. Esse preenchimento é feito em diagonais, onde a cada iteração o grau de paralelismo cresce até atingir um valor máximo, lembrando o comportamento de uma onda (em inglês, *wave*). (RAJKOY; ALURU, 2004)

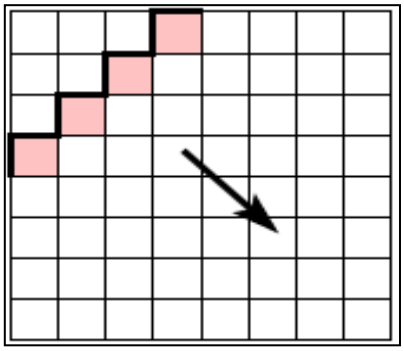


Figura 17: Exemplo de preenchimento da matriz de comparações utilizando o wavefront method.
Fonte: SCHIMIDT; CHEN, 2003.

O segundo método baseia-se em uma estratégia de *pipeline*, onde cada linha da matriz de comparações é computada sequencialmente por uma unidade de processamento paralelo. Cada uma dessas linhas deve ter sua execução paralisada enquanto os elementos necessários da linha superior não estiverem disponíveis para serem utilizados na fórmula recursiva do algoritmo NW para preencher o elemento corrente da matriz. Isto permite a execução do algoritmo NW de forma a preencher a matriz de programação dinâmica como um *pipeline* contínuo. (CHEN; YU; LEN, 2006)

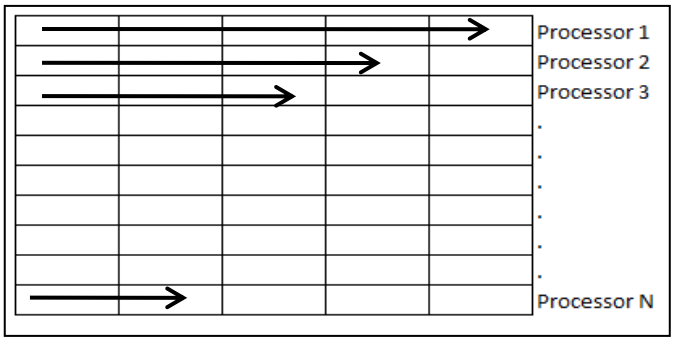


Figura 18: Preenchimento em paralelo utilizando pipeline, onde o elemento computado por um processador em uma linha é utilizado como entrada pelo processador que irá computar a linha imediatamente abaixo.
Fonte: Chen; Yu; Len, 2006.

Existe ainda outro desafio presente na paralelização de algoritmos de comparação de sequências por programação dinâmica, além da implementação de seu paralelismo inerente, como foi demonstrado pelo método *wavefront* anteriormente. Esse desafio está intimamente relacionado à complexidade espacial $O(mn)$ do algoritmo, já que esta complexidade indica que a matriz de comparações de tamanho $[m \times n]$, onde m e n são os comprimentos das sequências, que será alocada em memória. Como mencionado anteriormente o custo do crescimento de memória cresce exponencialmente, necessitando assim de modificações no algoritmo original, uso de heurísticas ou técnicas de economia de memória, para que se torne possível alinhar sequências biológicas de tamanho considerável. (BEZERRA; MAGALHÃES, 2006)

Existe uma real necessidade de algoritmos que consigam encontrar um alinhamento ótimo em curto espaço de tempo e pequenos gastos de memória. Entretanto ao empregar o uso de programação paralela, não significa que por distribuir a execução de um determinado problema em n processadores ou n *threads*, será alcançado um aumento de desempenho n vezes mais rápido. Isso ocorre pois muitos problemas computacionais não podem ser efetivamente paralelizados sem gastos de comunicações internas dos processadores e sincronia das tarefas em paralelo. (KIRK; HWU, 2010)

4.3 **Trabalhos anteriores sobre paralelização do algoritmo NW**

Nesta seção é realizada uma breve descrição sobre trabalhos relacionados a paralelização do algoritmo de Needleman-Wunsch (NW) em diferentes arquiteturas paralelas.

4.3.1 Implementação em paralelo do algoritmo NW em Clusters

Chen, Yu e Len (2006), em seu artigo, propuseram primeiramente uma comparação entre diversos algoritmos de alinhamento de sequências biológicas por programação dinâmica, e posteriormente apresentaram um algoritmo paralelo para este problema. Esse algoritmo foi implementado em um sistema de clusters utilizando MPI (*Message Passing Interface).*

Neste artigo os pesquisadores optam por utilizar o método *wavefront* para paralelizar sua aplicação. O *wavefront method*, muito comum em estratégias de paralelização de algoritmos de alinhamento de sequências, consiste em preencher a matriz de comparações por suas diagonais. Como cada elemento necessitará somente dos elementos pertencentes a duas diagonais anteriores, a cada iteração, mais elementos poderão ser preenchidos em paralelo.

A estratégia usada foi dividir a matriz de comparações em faixas por linhas, e blocos por colunas. Cada faixa foi concedida a um processador via uma alocação balanceada. Obedecendo a ordem das diagonais, cada processador calcula o bloco pertencente a sua faixa de forma concorrente. Devido a dependência de dados presente no algoritmo, era necessário que os processadores gastassem uma certa quantidade de tempo com comunicações entre eles para transferirem dados pertencentes às linhas da matriz que formavam a fronteira entre as diferentes faixas.

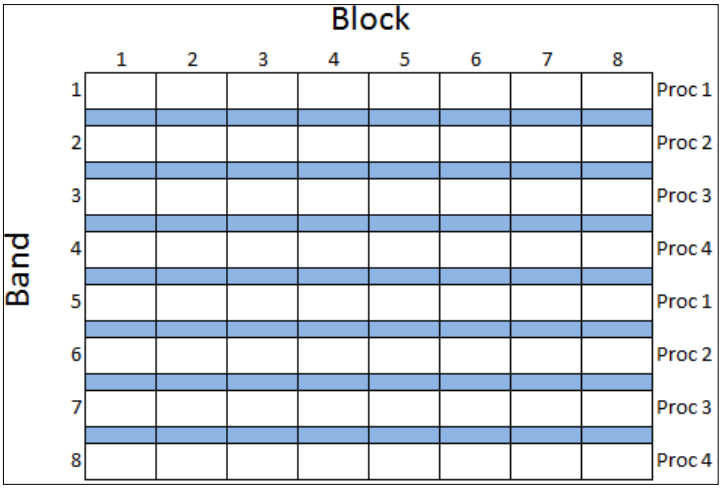


Figura 19: Esquema de particionamento proposto por Chen, Yu e Len (2006). Cada faixa (Band) é entregue a um processador, os quais processam os blocos (Block) da matriz. A pequena faixa em azul representa as comunicações feitas em entre os processadores.
Fonte: Chen; Yu; Len, 2006.

Aplicando estas técnicas de paralelismo, afirmaram que a complexidade de tempo do algoritmo foi reduzida para $O(n)$ quando n processadores eram utilizados. Também constataram que alterações no tamanho das faixas e dos blocos afetavam diretamente o ganho de desempenho atingido por sua implementação.

Por fim, com a implementação deste algoritmo paralelo feita em um sistema de cluster, os pesquisadores puderam observar que para sequências biológicas com grande comprimentos o ganho de desempenho era vantajoso em relação a implementação sequencial. Porém conforme a quantidade de processadores fosse aumentando, esse ganho era bastante reduzido. O motivo disso ocorrer é que há um grande desperdício de tempo devido às comunicações entre processadores ocasionadas pelo fato de haver troca de dados entre eles. (CHEN; YU; LEN, 2006)

4.3.2 Abordagem teórica para paralelização do algoritmo NW

Neste artigo Seguel e Torres (2011) basearam-se no método de divisão e conquista explorado no algoritmo de Hirschberg (1975). Isso foi feito com o intuito de

criar uma versão paralela do algoritmo de alinhamento global de sequências proposto por Needleman-Wunsch com complexidade de espaço também $O(m)$.

O algoritmo de Hirschberg é utilizado para encontrar a maior subsequência comum a duas sequências utilizando complexidade de espaço $O(m)$, onde m é o tamanho da menor sequência a ser comparada. (HIRSCHBERG, 1975)

Os pesquisadores em sua busca por uma versão paralela do algoritmo de alinhamento de sequências propuseram uma alternativa não recursiva à etapa conhecida como *backtrack* do algoritmo NW. Essa etapa é responsável por retornar o alinhamento ótimo entre um par de sequências. (SEGUEL; TORRES, 2011)

Esta alternativa proposta por Seguel e Torres (2011) é baseada nas propriedades simétricas provenientes da matriz de comparações gerada pelo algoritmo de Needleman-Wunsch. Estas propriedades admitem que ao comparar uma matriz D com uma matriz $D*$, onde D é a matriz de programação dinâmica gerada pela comparação de duas sequências, e $D*$ é a matriz gerada pela comparação das mesmas sequências de D, porém revertidas, obtém-se os mesmos alinhamentos, porém revertidos um em relação ao outro.

Este método que faz uso do paradigma de divisão e conquista, o qual consiste em dividir um problema complexo em subproblemas mais simples até que um problema trivial seja alcançado, utiliza a etapa de preenchimento da matriz de comparações presente no algoritmo NW repetidamente. A cada iteração essa etapa é repetida para sequências de tamanhos aproximadamente iguais à metade das sequências utilizadas nas iterações anteriores com o propósito de dividir o problema de comparação de sequências em subproblemas, até que um problema de tamanho predeterminado seja resolvido. (SEGUEL; TORRES, 2011)

Somente a última coluna de cada matriz de comparações D e $D*$ geradas pelos subproblemas intermediários devem ser mantidas temporariamente em memória para se determinar a pontuação do melhor alinhamento, e assim dividir a sequência com maior pontuação do subproblema anterior em duas. Este processo de dividir uma sequência em duas, denominado pelos autores como “fase de divisão”, é repetido para cada subproblema até que uma subsequência de tamanho predeterminado seja descoberta. A fase de divisão termina com a aplicação do algoritmo NW para cada subsequência encontrada de tamanho predeterminado. (SEGUEL; TORRES, 2011)

Os autores exploraram em um modelo teórico todas as aplicações computacionais independentes feitas nas fases de divisão e conquista existentes no algoritmo proposto. Estas são, em suma, a computação da última coluna da matriz de programação dinâmica para o alinhamento global ótimo de cada par de subsequências, a computação do caminho na matriz de comparações para cada subsequência de comprimento menor ou igual ao valor de comprimento máximo predeterminado e a produção dos alinhamentos correspondentes. (SEGUEL; TORRES, 2011)

O paradigma usado pelos pesquisadores para implementação do algoritmo proposto em paralelo utiliza o paradigma de mestre/trabalhadores, onde existe uma unidade de processamento paralelo (mestre) responsável por coordenar outras unidades que irão descrever a execução em paralelo (trabalhadores). (SEGUEL; TORRES, 2011)

Primeiramente, a unidade de processamento paralelo mestre envia tarefas para dois trabalhadores. Quando essas tarefas terminam, os dois trabalhadores enviam subtarefas para outros dois trabalhadores, totalizando quatro trabalhadores para esta implementação. Todos os quatro trabalhadores processam suas subtarefas em paralelo. Então a etapa de divisão das sequências irá gerar uma árvore de altura igual a dois, onde cada nó representa uma unidade de processamento. Portanto esta árvore terá sua raiz como sendo a unidade de processamento mestre. (SEGUEL; TORRES, 2011)

Posteriormente a etapa para apresentar o alinhamento ótimo global terá os quatro trabalhadores concatenando os alinhamentos ótimos encontrados para os casos onde a subsequência possuir um tamanho pré-determinado, e por fim, enviando o resultado dessa concatenação para a unidade mestre de processamento. (SEGUEL; TORRES, 2011)

Os autores se basearam então em um modelo simplístico de desempenho, e com ele demonstraram que o algoritmo paralelo proposto poderia obter um ganho de performance considerável em relação ao tempo de execução do algoritmo de NW. Assim, demonstraram matematicamente que para este modelo o ganho de desempenho poderia ser calculado aproximadamente pela fórmula abaixo, onde P é o número de passos requeridos na fase de divisão do algoritmo paralelo.

$$S = \frac{1}{\left[1-\frac{P}{2}\right]} \quad (8)$$

4.3.3 Implementação paralela do algoritmo NW usando computação em *Grid*

No artigo intitulado "*Parallel Needleman-Wunsch Algorithm for Grid*", ou "Algoritmo Needleman-Wunsch paralelizado em *Grades"*, os autores Naveed, Siddiqui e Ahmed (2005) utilizam o *software* conhecido como "*Alchemi Grid*" para implementarem uma versão em paralelo do algoritmo NW.

AlchemiGrid é um *software* de código aberto que permite de forma simples agregar poder computacional de máquinas conectadas em redes e transformá-las em um único supercomputador virtual. (http://www.cloudbus.org/~alchemi/, Acessado em 28/03/2013)

A versão em paralelo do algoritmo de NW feita pelos pesquisadores Naveed, Siddiqui e Ahmed (2005) utiliza vários processadores para inicializarem, calcularem e preencherem em paralelo a matriz de comparações. Essa matriz contém a sequência de DNA e seus valores calculados, além da matriz de ponteiros. A matriz de ponteiros armazena também as sequências de DNA, porém armazena as direções a serem seguidas na etapa de *backtrack*.

O algoritmo proposto aloca as estruturas de dados a serem utilizadas (a matriz de comparações e de ponteiros) em uma memória global que permite acesso simultâneo por todos os processadores do *grid* para que estes possam efetuar inicializações e cálculos sobre essas estruturas.

Os passos de inicialização das sequências de DNA podem ser feitos em paralelo para as duas matrizes, assim como a inicialização das penalidades por inserção de um espaço vazio nas sequências (*gap*). (NAVEED; SIDDIQUI; AHMED, 2005)

Eles utilizam o *wavefront method* para preencher a matriz de comparações em paralelo e demonstram que quanto mais unidades de processamentos disponíveis, mais elementos da matriz poderão ser preenchidos de forma paralela. Nesta implementação cada processador é responsável pelo preenchimento de um elemento da diagonal corrente. Caso haja mais elementos na diagonal do que o

número de processadores, então unidades de processamento calcularão mais de um elemento da matriz de programação dinâmica em paralelo.

Para esta implementação em *grid* utilizaram um conjunto de computadores conectados por uma estrutura de redes, o que acarretou em problemas de tráfego intenso na rede devido a grande troca de informações entre os diferentes nós proveniente das grandes sequências de DNA utilizadas nos experimentos. A fim de que esse tráfego fosse reduzido, os autores utilizaram técnicas para calcular o número de *threads* a serem criadas baseando-se na carga de trabalho que cada *thread* haveria de ter e o número de elementos presentes na diagonal corrente. (NAVEED; SIDDIQUI; AHMED, 2005)

O algoritmo proposto pelos pesquisadores alcançou uma redução na complexidade do algoritmo sequencial de NW original de $O(nm)$ para $O(n+m)$. O sistema *Alchemi* foi utilizado para demonstrar o ganho de desempenho pelo algoritmo proposto quando comparadas grandes cadeias de DNA. (NAVEED; SIDDIQUI; AHMED, 2005)

Dessa forma conseguiram implementar as etapas de inicialização das duas matrizes utilizadas em seu algoritmo em paralelo. Os autores também afirmaram que para esta implementação quanto mais CPUs houvessem, mais cálculos seriam processados em paralelo, gerando uma maior ganho de desempenho.

4.3.4 Implementação paralela do algoritmo NW utilizando o modelo BSP/CGM

Neste artigo, Alves et al. (2003) apresentam uma implementação em paralelo do algoritmo de comparações de sequências de Needleman-Wunsch utilizando computadores com memória distribuída, onde cada máquina possui P processadores e $O((m+n)/P)$ espaço disponível em memória, onde m e n são os comprimentos das duas sequências a serem comparadas. O algoritmo proposto pelos autores requer $O(P)$ etapas de comunicações e complexidade de tempo de $O(mn/P)$ para processamento local.

A característica inovadora do algoritmo desenvolvido baseia-se em uma relação entre a carga de trabalho de cada processador e o número de etapas de comunicações requeridas, esta razão é expressa por um parâmetro denominado α.

O algoritmo é baseado em função deste parâmetro que pode ser melhorado para obter o melhor ganho geral em uma determinada implementação em paralelo. (ALVES et al., 2003)

Os pesquisadores utilizaram a característica de comunicação proveniente do método de preenchimento em paralelo da matriz de programação dinâmica do algoritmo NW denominado *wavefront method*, no qual cada processador deve apenas se comunicar com outros dois processadores.

Para Alves et al. (2003) o uso do modelo BSP/CGM de programação paralela apresentava uma maior vantagem para a implementação de suas soluções para o problema de alinhamento global de sequências.

Um algoritmo que segue o modelo BSP, ou *Bulk Synchronous Parallel Model*, consiste em uma sequência de etapas denominadas super passos separados por uma barreira de sincronização. Em um super passo, cada processador executa uma série de instruções independentes usando dados locais disponíveis em cada processador no início de um super passo, assim como comunicações consistindo em envios e recebimentos de mensagens. Uma relação-h em um super passo corresponde ao envio e recebimento de mensagens de pelo menos h mensagens em cada processador.

Entretanto, os autores preferiram utilizar uma variação do modelo BSP chamada *Coarse Grained Multicomputers* ou CGM. Neste modelo, P processadores são conectados por qualquer rede de interconexão. O termo *Coarse Grained* - Granularidade Grossa - deriva do fato de que o tamanho do problema em cada processador (n/P) é considerado maior que o número de processadores disponíveis. Um algoritmo implementado utilizando o modelo CGM consiste em uma sequência de super passos, onde cada super passo é dividido em uma fase de comunicação global e uma fase de computação local. (ALVES et al., 2003)

Segundo os autores, os algoritmos implementados utilizando o modelo CGM apresentam ganho de desempenho similar a ganhos preditos em modelos teóricos.Por fim, apresentaram um algoritmo paralelo utilizando o modelo CGM de programação paralela que necessita de $O(P)$ etapas de comunicação e $O((m+n)/P)$ espaço disponível em memória para cada processador, onde m e n são os comprimentos das duas sequências a serem comparadas, e P é o número de processadores. (ALVES et al., 2003)

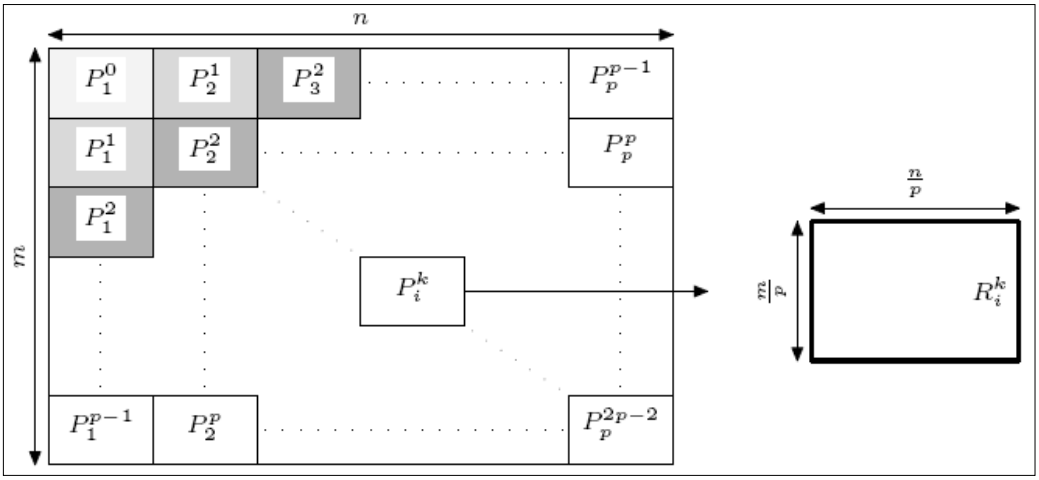


Figura 20: Exemplo de preenchimento da matriz de comparações utilizando P processadores. Para este exemplo o algoritmo possui complexidade espacial O(mn/P), porém se mantidas em memória somente as últimas colunas do bloco essa complexidade decresce para O((m + n)/P).
Fonte: ALVES et al., 2003.

O algoritmo proposto reduz a complexidade de tempo do original $O(mn)$ para $O(mn/P)$ de tempo de computação local. Os pesquisadores também demonstraram que o parâmetro α mencionado anteriormente, quando combinado com o tamanho de blocos no qual a matriz de comparações era dividida, podia ser ajustado para obter o melhor ganho de desempenho em uma determinada implementação. Dessa forma atingiram um *speedup* de 50 em relação ao algoritmo sequencial. (ALVES et al., 2003)

4.3.5 Implementação paralela do algoritmo NW usando *multithreads*

Neste artigo Gao et al. (2001) propuseram uma abordagem para o problema de paralelização de alinhamento global por programação dinâmica utilizando uma plataforma de computação paralela generalizada baseada em um modelo de execução de *multithreads*, onde o problema de alinhamento abordado utilizava uma granularidade fina de execução em paralelo direcionada por eventos.

O algoritmo proposto foi implementado na arquitetura EARTH – *Eficient Architecture for Running Threads* ou Arquitetura Eficiente para Execução de *Threads*

-, a qual consiste em um modelo de *multithreads* de granularidade paralela fina comandadas por eventos. Esta arquitetura foi adaptada pelos autores para ser utilizada agregando diversos computadores com processadores tradicionais. (GAO et al., 2001)

Os pesquisadores, com o objetivo de criar um versão paralela do algoritmo de NW, optaram por paralelizar a etapa de preenchimento da matriz de comparações. Como muitos outros trabalhos abordando o problema de alinhamento global, escolheram utilizar o *wavefront method* para preencher a matriz de comparações gerada pelo algoritmo NW.

Apesar de este método expor o paralelismo desse algoritmo, Gao et al. (2001) levantaram alguns desafios a serem abordados para implementar uma versão paralelizada do algoritmo de NW utilizando a arquitetura proposta.

O primeiro desafio está relacionado aos tamanhos das diagonais da matriz que variam durante a execução do algoritmo. Esse fator gera um trabalho desbalanceado entre os processadores. Por exemplo, na primeira iteração o processador $P1$ iria calcular o elemento $[1,1]$ da matriz de programação dinâmica. Na segunda iteração o processador $P1$ e $P2$ estariam computando respectivamente o elemento $[1,2]$ e $[2,1]$. Pode-se afirmar que para P processadores, onde o valor de colunas é M, tal que $M > P$, seriam necessárias P iterações para se utilizar todos os processadores, e logo após a iteração de número P já haveria pelo menos um processador ocioso se for considerado o pior caso, onde o número de colunas é igual ao número de linhas.

O segundo desafio identificado está relacionado ao número de elementos a serem computados por cada processador em cada iteração. Se for considerado que cada processador será responsável por preencher um elemento da matriz, para problemas de alinhamento de sequências biológicas reais onde o número de elementos possui uma ordem de grandeza elevada, seria necessário que houvessem muitos processadores compondo o sistema. Além do mais, essa implementação geraria um desperdício de recursos do sistema já que requereria uma quantidade excessiva de comunicações entre processadores ativos em uma determinada iteração.

Com o intuito de anular esses problemas provenientes da paralelização do algoritmo NW, Gao et al. (2001) propuseram uma divisão da matriz de comparações em blocos, onde cada bloco necessitaria apenas dos elementos imediatamente

superiores ao bloco e dos elementos imediatamente à esquerda do bloco. Com essa solução a razão de comunicação entre processadores cairia em aproximadamente 81%, porém haveria certa perda do grau de paralelismo já que haveriam partes sequenciais no algoritmo para preencher cada bloco de forma independente.

Ainda para aumentar a utilização dos processadores, os blocos seriam agregados em faixas de blocos, e cada faixa seria entregue a um processador para ser computada. Essa técnica aumentaria a utilização dos processadores em até 75% de acordo com os autores.

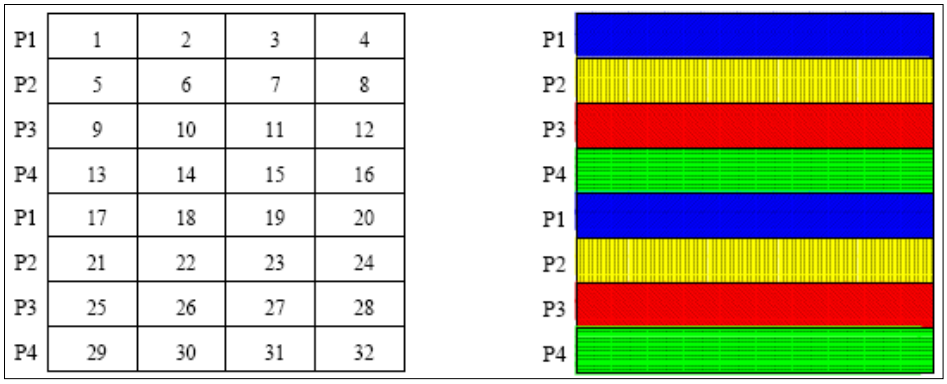


Figura 21: Exemplificação da divisão da matriz de comparações proposta pelos pesquisadores. A matriz é dividida primeiramente em blocos, e posteriormente em faixas de blocos. Cada faixa é entregue a um processador.
Fonte: GAO et al., 2001.

Na implementação utilizando arquitetura EARTH e o modelo de *multithreads* proposto por Gao et al. (2001), cada nó do sistema possui duas *threads* instanciadas, e cada thread possui duas *fibers* ou fibras. Cada faixa é entregue a uma *thread* para processamento, e as duas *fibers* de uma *thread* são instanciadas repetidamente para calcularem um bloco por vez. Somente uma das duas *fibers* pode estar ativa por vez.

Com essa implementação afirmaram que obtiveram um *speedup* de 90 usando um sistema composto por 120 máquinas com processadores comuns. (GAO et al., 2001)

4.4 **Estratégias abordadas neste trabalho**

O problema relacionado ao desempenho de algoritmos de alinhamento de sequências biológicas tem sido um assunto abordado exaustivamente na literatura de programação paralela relacionada à Bioinformática. (ZOMAYA, 2006)

O objetivo principal dos trabalhos propostos pelos diversos pesquisadores de diferentes centros de pesquisas ao redor do mundo é obter um ganho de desempenho por parte dos algoritmos já conhecidos através da implementação em paralelo, tanto por *software* quanto por *hardware*. Dessa forma, o tempo de execução seria decrescido em relação ao tamanho das sequências comparadas, já que estas, tratando-se de cadeias biológicas, possuem comprimentos consideráveis. Para estes casos podem-se ser citados os trabalhos de Li, Ranka e Sahni (2012) e Gao et al. (2001).

Uma segunda abordagem para o problema consiste na criação de novos algoritmos que utilizavam ideias dos anteriormente propostos para se alcançar um alinhamento ótimo com consumo de tempo e memória aceitáveis. Para estas abordagens podem-se citar como exemplo os trabalhos de Rajkoy e Aluru (2004) e Bezerra e Magalhães (2006).

Este trabalho propõe um estudo do algoritmo NW de forma a entender os mecanismos do paradigma de programação paralela possíveis para desenvolver uma solução para o problema de alto consumo de tempo requerido para execução desse algoritmo, e ainda assim obter resultados exatos e ótimos sem necessidade de perder precisão com uso de heurísticas.

Neste trabalho não são aplicadas modificações no algoritmo NW para reduzir consumo de memória. As implementações paralelizadas são baseadas na versão original do algoritmo de alinhamento global criado por Needleman–Wunsch.

A versão paralela do algoritmo NW implementada neste trabalho pode ser utilizada para fazer comparações de sequências biológicas de DNA e RNA somente. Esta seção disserta sobre as estratégias abordadas para implementar uma versão paralelizada do algoritmo de NW.

4.4.1 Análise do consumo de tempo proveniente da execução do algoritmo NW

Primeiramente, antes de se fazer qualquer tipo de implementação em paralelo, é necessário fazer uma análise do algoritmo em sua implementação serial para descobrir quais partes do problema podem ser computadas em paralelo, e quais partes apresentam um ganho vantajoso ao serem paralelizadas. Para isso, leva-se em consideração o trabalho realizado pelo programador, os recursos computacionais gastos pela nova implementação em paralelo e o ganho de desempenho da nova implementação em relação à antiga. (KIRK; HWU, 2010)

Com este objetivo foi realizada uma implementação tradicional do algoritmo NW na linguagem C, e esta foi utilizada como base de comparação com todas as outras implementações em paralelo feitas nesse projeto para medir o ganho de desempenho ou *speedup* alcançado. O algoritmo desenvolvido por Needleman-Wunsch para encontrar um alinhamento ótimo global, é constituído de basicamente duas etapas.

A primeira etapa refere-se ao preenchimento da matriz de comparações de $[(n+1) \times (m+1)]$, a qual é uma matriz de programação dinâmica que guarda todos os alinhamentos entre as subsequências das duas sequências a serem comparadas, onde m e n representam o comprimento das duas sequências. Este preenchimento é feito com base em uma função recursiva onde o elemento $[i,j]$ da matriz recebe o valor máximo dos elementos $[i-1,j-1]$, $[i,j-1]$, $[i-1,j-1]$ somado com penalidades ou bônus pré-determinados. (NEEDLEMAN; WUNSCH, 1970)

A segunda etapa, conhecida também como *backtrack,* refere-se a um procedimento de encontrar o melhor caminho nesta matriz partindo do elemento $[n,m]$ até alcançar o elemento $[0,0]$. A cada iteração desta etapa é verificado de qual das três células da matriz o elemento corrente recebeu seu valor. Supondo que o elemento $[i,j]$ seja o elemento corrente, será verificado se o valor dele é proveniente dos elementos $[i-1,j-1]$, $[i,j-1]$, $[i-1,j-1]$. Caso seja o elemento da diagonal $[i-1,j-1]$, o símbolo da sequência na coluna j é escrito sobre o símbolo da sequência da linha i. Se o valor do elemento corrente vier da célula imediatamente acima $[i-1,j]$, é inserido um espaço vazio na sequência representada pelas

colunas. Se vier do elemento esquerdo, é inserido um espaço vazio na sequência representada pelas linhas da matriz de comparações. (NEEDLEMAN; WUNSCH, 1970)

Em nossa implementação instrumentamos o código fonte para medir o tempo de execução das duas etapas separadamente para descobrir o comportamento temporal do algoritmo NW. A execução da implementação sequencial feita neste trabalho em uma máquina *Quad-Core* Intel com 8GB de memória RAM e sistema operacional Linux, demonstrou que em média 88,61% do tempo gasto durante a execução do algoritmo estava na etapa de preenchimento da matriz de comparações, como mostram os resultados coletados na tabela 3 abaixo e resumidos no gráfico da figura 22. A coluna "Preenchimento" consiste na porcentagem de tempo gasto para preencher a matriz de comparações. A coluna "*Traceback"* é preenchida com a porcentagem de tempo gasto pela etapa de *traceback* do algoritmo. A coluna "Outros", representa o tempo gasto pela implementação para alocar espaço na memória, inicializar variáveis, ler e escrever arquivos, entre outros.

Tabela 3: Resultados de testes do consumo de tempo do algoritmo em porcentagem dado as dimensões da matriz de comparações.

Linhas	***Colunas***	***Preenchimento%***	***Traceback%***	***Outros %***	***Total***
1001	2001	82,56%	1,49%	15,95%	0,040031
4001	6001	84,66%	0,21%	15,13%	0,465284
7001	10001	86,43%	0,11%	13,46%	1,345382
10001	14001	87,01%	0,09%	12,90%	2,717167
13001	18001	85,70%	0,07%	14,24%	4,476925
16001	22001	87,86%	0,05%	12,08%	6,721583
19001	26001	89,36%	0,05%	10,59%	9,478643
22001	30001	90,61%	0,04%	9,34%	12,62466
25001	34001	97,56%	0,04%	2,41%	16,094509
28001	38001	97,58%	0,03%	2,39%	20,289435
31001	42001	97,69%	0,03%	2,28%	24,940113
34001	46001	97,78%	0,03%	2,20%	29,565598
	Média	88,61%	0,05%	11,34%	

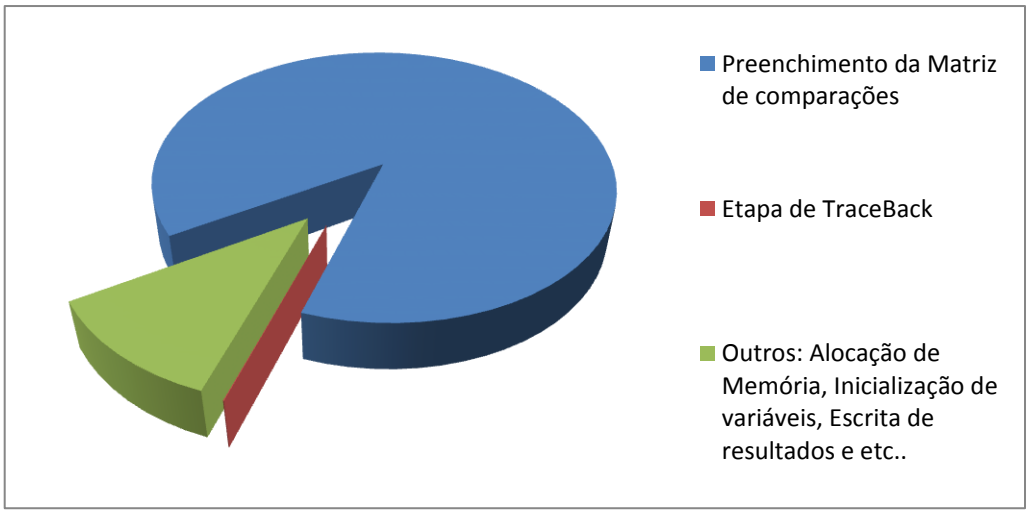


Figura 22: Gráfico que demonstra o comportamento do algoritmo de NW.

Esse comportamento já era esperado devido à complexidade temporal da etapa de preenchimento da matriz de comparações ser $O(nm)$, enquanto a segunda

etapa de *backtrack* possui apenas complexidade de tempo $O(n+m)$. (STEINFADT; SCHERGER; BAKER, 2006)

Por este motivo, foi escolhido neste trabalho o estudo da paralelização da primeira etapa do algoritmo de Needleman-Wunsch.

A paralelização somente da primeira etapa dos algoritmos de comparações de sequências é uma abordagem bastante utilizada para este problema, e pode ser verificada em trabalhos de paralelização de algoritmos de alinhamento global e local como os de Steinfadt, Scherger e Baker (2006), Yap, Frieder e Martino (1998), entre outros.

4.4.2 Analisando a natureza paralela do algoritmo NW

A primeira fase do algoritmo de Needleman-Wunsch consiste em preencher a matriz de comparações, a qual contém a pontuação de todos os alinhamentos entre as subsequências das sequências a serem comparadas. No elemento $[m,n]$ haverá a pontuação do alinhamento entre as duas sequências, onde n e m são os comprimentos das sequências. (NEEDLEMAN; WUNSCH, 1970) Este preenchimento é feito com base em uma função recursiva descrita nas seções anteriores.

A matriz de comparações pode ser preenchida de três formas distintas: por linhas, por colunas ou por diagonais. Existem duas estratégias quanto à forma de paralelizar o preenchimento desta matriz.

A primeira, menos comum, refere-se a computar a matriz na forma de um *pipeline*, onde o resultado da computação de um elemento é o dado necessário para se computar o outro. Para esta estratégia, cada linha ou coluna da matriz deve ser entregue a uma unidade de processamento em paralelo. Estas unidades terão suas execuções paralisadas se o dado necessário da linha ou coluna imediatamente anterior não estiver processado. (CHEN; YU; LEN, 2006)

A segunda, mais comumente usada na paralelização dos algoritmos de alinhamento, preenche a matriz por diagonais (*wavefront*). A cada iteração, a quantidade de elementos da matriz de comparações que podem ser computados em paralelo cresce. Quando o algoritmo estiver em sua n-ésima iteração, onde $n = m$, o

nível de paralelismo máximo será alcançado. Posteriormente na iteração $n+1$, a taxa de elementos computados em paralelo começa a decrescer até chegar novamente a 1 elemento, o qual será a célula $[m,n]$ da matriz de programação dinâmica. (LI; RANKA; SAHNI, 2012) (YAP; FRIEDER; MARTINO, 1998) (NAVEED; SIDDIQUI; AHMED, 2005)

Neste trabalho será utilizado o *wavefront method* para preencher a matriz em paralelo, pois de acordo com trabalhos anteriores, este método é capaz de oferecer ganhos de performance razoáveis em tipos variados de arquiteturas e modelos de programação paralela. (LI; RANKA; SAHNI, 2012) (YAP; FRIEDER; MARTINO, 1998) (NAVEED; SIDDIQUI; AHMED, 2005) (CHEN; YU; LEN, 2006) (GAO et al., 2001)

4.4.3 Utilização de matrizes de pontuação neste trabalho

Matrizes de pontuação são utilizadas em alinhamentos de sequências biológicas para melhorar o significado do alinhamento retornado pelo algoritmo. Estas servem como uma tabela a ser consultada durante a execução do algoritmo para bonificar *matches*, isto é, ocasiões onde os símbolos em uma posição $[i,j]$ da matriz de comparações são iguais, ou penalizar *mismatches*, isto é, símbolos em uma posição $[i,j]$ da matriz que são diferentes. (MOUNT, 2004)

Matrizes como PAM, BLOSUM e suas variações são mais comumente utilizadas em alinhamentos de proteínas. Alinhamentos de moléculas de DNA utilizam matrizes identidade para suas comparações. Entretanto existem outras matrizes de pontuação para o alinhamento de DNA além da matriz identidade. (MOUNT, 2004)

Com o objetivo de possibilitar o uso desses tipos de tabela na versão paralelizada do algoritmo NW proposta neste trabalho, e ainda assim evitar instruções de decisões para acessar a matriz de pontuações, foi criada uma estrutura vetorial onde os índices do vetor de pontuação são os valores em código ASCII dos nucleotídeos.

Para melhorar o entendimento desta solução é proposto o seguinte exemplo considerando o uso da linguagem C:Considere o alinhamento de duas sequências, sequência A = 'ATCTG' e sequência B ='CATCGT', e considere a matriz de pontuação abaixo utilizada para determinar alinhamentos de sequências distantes:

	A	T	C	G
A	0	5	5	1
T	5	0	1	5
C	5	1	0	5
G	1	5	5	0

Figura 23: Matriz de pontuação utilizada para determinar alinhamentos de sequências que possuem parentesco evolutivo distante.
Fonte: http://bioinformatics.istge.it/bcd/Curric/PrwAli/nodeD.html (Acessado em 21/04/2013)

Para esta matriz o valor referente a comparação do nucleotídeo “A” com “C”, o qual gera um *mismatch*, é uma pontuação de valor 5. Este *mismatch* está representado nesta matriz no elemento de coordenadas $[0,2]$.

Durante a execução do algoritmo, deverá existir uma condição para descobrir qual índice da matriz de pontuação referenciará determinada combinação de nucleotídeos. Por exemplo:

1) Se elemento corrente da sequência A for igual a 'A' então $i = 0$.
2) Se elemento corrente da sequência B for igual a 'T' então $j = 1$.
3) Logo, a pontuação somada ao elemento da célula da matriz de comparações deverá ser o valor do elemento da matriz de pontuação $[0,1]$.

Considerando que existamquatro nucleotídeos, seriam necessários oito tipos de cláusulas condicionais (*If-Else*) para descobrir as coordenadas na matriz de pontuação de dois nucleotídeos comparados.

Pode-se observar que é necessário conhecer apenas 10 elementos da matriz de pontuação, isto é, os elementos da diagonal principal e os elementos da parte inferior ou superior a esta diagonal. Isto ocorre pelo fato de não haver diferenças

entre alinhar um nucleotídeo 'A' com 'T' ou alinhar o nucleotídeo 'T' com 'A', porque o alinhamento entre combinações de nucleotídeos caracteriza uma operação comutativa.

Com objetivo de evitar diversos testes para determinar qual posição da matriz acessar, o que causaria perda de desempenho durante a execução, foi proposto neste trabalho a criação de um vetor que possui por índices a soma dos valores ASCII dos nucleotídeos. Estes valores são calculados com base nas conversões implícitas de tipos primitivos existentes na linguagem C.

Tabela 4: Valor decimal do código ASCII para os caracteres A, T, C e G.
Fonte: http://www.acm.uiuc.edu/webmonkeys/book/c_guide/a.html
(Acessado em 21/04/2013)

Caractere	***Decimal***
A	65
T	84
C	67
G	71

A estrutura de dados vetorial criada para não utilizar as cláusulas condicionais (*If-Else*) teria o número de elementos igual a maior soma entre as letras A,T,C e G adicionado do valor 1, pois o elemento $[0]$ é considerado. Haveria então $(84 + 84 + 1) = 169$, ou seja, $('T' + 'T' + 1)$ elementos no vetor. Esta situação geraria um grande desperdício de espaço já que desses elementos somente seriam utilizados 10 espaços do vetor. Por este motivo foi criada a fórmula $\left\lfloor \frac{(N1+N2-130)}{2} \right\rfloor$, onde $N1$ e $N2$ são os nucleotídeos comparados, para diminuir o número de elementos vazios no vetor.

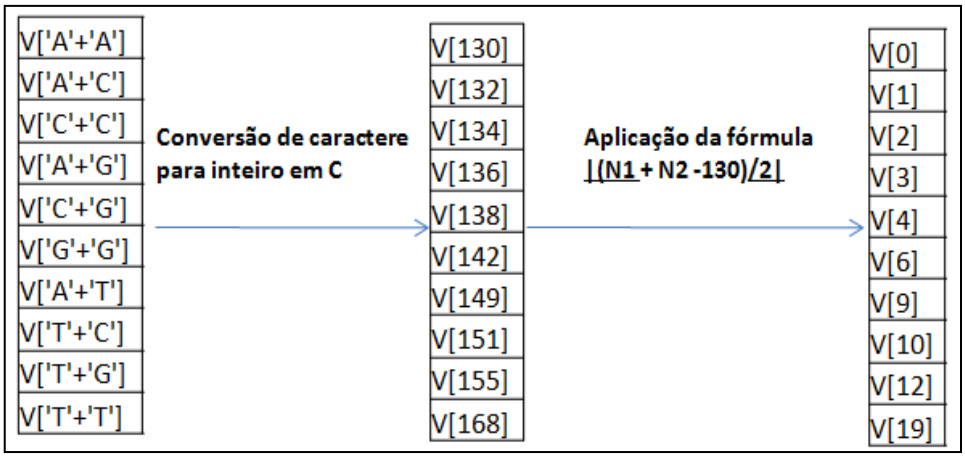


Figura 24: O primeiro vetor representa a soma feita entre os caracteres A, T, C e G. O segundo vetor mostra o resultado das somas baseando-se na tabela ASCII. O terceiro mostra o resultado da subtração da soma entre os dois caracteres por 130, dividido por 2, e tendo este resultado arredondado para baixo.

Desta forma é garantido que não haverá resultados iguais, o que geraria ambiguidade na atribuição de valores, pois um mesmo elemento do vetor poderia ter dois resultados distintos. Essa implementação também diminuirá o espaço desperdiçado por elementos do vetor não utilizados de 159 $(169 - 10 = 159)$ para 10 $(20 - 10 = 10)$. Se for comparada à matriz de pontuação $[4 \times 4]$, a qual aloca 16 elementos em memória com a estrutura vetorial proposta nesta implementação, a qual ocupa 20 elementos na memória, poderá ser concluído que o vetor ocupará na memória 25% de espaço a mais que a matriz. Porém não haverá necessidade da execução de várias cláusulas condicionais, pois o valor da comparação de dois nucleotídeos poderá ser acessado diretamente no vetor como demonstrado no esquema da figura 24.

4.4.4 Utilização do *wavefront method* para preenchimento da matriz de comparações

O *wavefront method* consiste em preencher a matriz de comparações $[m \times n]$, onde m e n representam os comprimentos das sequências, por suas diagonais. Desta forma cada elemento de uma diagonal da matriz poderá ser preenchido

independentemente dos demais em paralelo, pois cada diagonal necessita apenas das duas diagonais imediatamente anteriores para ser preenchida.

O grau de paralelismo para este método de preenchimento cresce com razão igual a 1 para cada iteração. Quando o número de iterações é igualado a n, onde n representa a menor dimensão da matriz, o grau de paralelismo máximo é atingido. Este nível máximo de paralelismo é mantido para as próximas $(m-n)$ iterações, onde m representa a maior dimensão da matriz. Após este número de iterações, o grau de paralelismo começa então a decrescer de forma constante com razão igual a -1, depois de $[n-1]$ iterações o preenchimento da matriz termina com o mesmo grau de paralelismo que havia no início de seu preenchimento.

Se for somado o número de iterações necessárias para percorrer todas as diagonais da matriz será encontrada a fórmula $(m+n-1\,(n\,+m-n+\,n-1))$, onde m e n representam as dimensões das matrizes. Esta fórmula pode ser utilizada para descobrir o número de diagonais presentes em uma matriz conhecendo as suas dimensões e pode ser facilmente provada utilizando indução matemática.

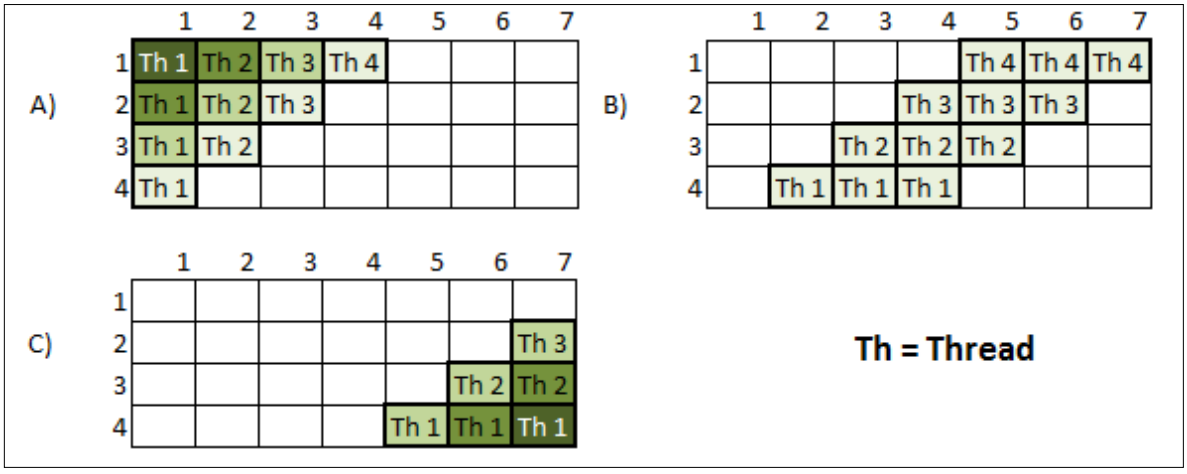


Figura 25: (A) Primeira etapa do método wavefront. O grau de paralelismo cresce com razão 1, isto é, na primeira iteração só um elemento da matriz poderá ser calculado por uma thread, na segunda iteração haverá duas threads para calcular dois elementos em paralelo, e assim sucessivamente. (B) Nesta segunda etapa o grau máximo de paralelismo se mantém até o número de iterações ser igual ao de colunas da matriz. (C) A terceira etapa exemplifica o decrescimento do grau de paralelismo, até chegar à última célula da matriz.

Para preencher a matriz de comparações pelas suas diagonais são necessários dois laços de repetições: um laço externo para percorrer as diagonais e outro para percorrer a quantidade de elementos em uma diagonal. O número de repetições do primeiro laço pode ser determinado pela fórmula $(\mathrm{m}+\mathrm{n}-1)$, já que este indica o número de diagonais presentes em uma matriz. O segundo laço possui um número de iterações que varia em função do número da iteração do primeiro laço. Isto é, na iteração 1 do primeiro laço, haverá uma iteração para o laço interno que é a quantidade de elementos presentes na diagonal corrente. Na segunda iteração do laço externo, o laço interno terá duas iterações, e assim sucessivamente. Quando o primeiro laço chegar a sua n-ésima iteração, o segundo laço fará n iterações, onde n é o valor da menor dimensão da matriz. Após isso, a quantidade de iterações do laço interno se manterá constante para as próximas $(\mathrm{m}-\mathrm{n})$ iterações, e começará a decrescer constantemente com razão -1 até chegar ao ultima iteração do laço externo.

A Formalização matemática do comportamento do laço interno quando se preenche uma matriz por suas diagonais é descrita na fórmula abaixo:

$$\left\{\begin{array}{c} Q(i)=i, \quad se \quad 1 \leq i \leq n-1, \\ Q(i)=n, \quad se \quad n \leq i \leq m, \\ Q(i)=m+n-i, \ se \ m+1 \leq i \leq m+n-1 \\ onde\ i,m,n \in \mathbb{N}\ e\ m \geq n \geq 1. \end{array}\right\} \quad (9)$$

Dado um elemento id de uma diagonal d, para se encontrar o valor deste elemento em coordenadas de linhas e colunas de uma matriz M, deve-se aplicar a transformação $[\mathrm{id}+\mathrm{c}, \mathrm{d}-\mathrm{id}+\mathrm{c}]$, onde d é o identificador da diagonal, id é o identificador do elemento de uma diagonal e c é o número da coluna onde essa diagonal começa.

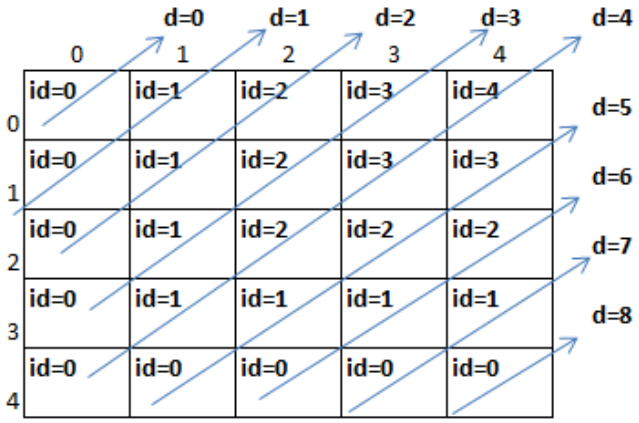


Figura 26: Exemplificação do esquema de numeração das diagonais e de seus elementos.

Tabela 5: Demonstração da fórmula descrita para diagonais distintas em cada fase do algoritmo.

			Coluna	***Linha***
d	***Id***	***L***	***id+l***	***d-id-l***
0	0	0	0	0
3	0	0	0	3
3	1	0	1	2
3	2	0	2	1
3	3	0	3	0
4	0	0	0	4
4	1	0	1	3
4	2	0	2	2
4	3	0	3	1
4	4	0	4	0
8	0	4	4	4

Para implementar o procedimento de preenchimento da matriz pelo método *wavefront* em linguagem C é necessário utilizar cláusulas condicionais (*if-else*) para determinar como será o comportamento do laço mais interno, o qual percorre os

elementos da diagonal corrente. Porém, o uso dessas cláusulas para implementar a versão paralela do algoritmo NW na arquitetura CUDA geram degradação de desempenho.

4.5 **Considerações finais sobre o capítulo**

O algoritmo de Needleman-Wunsch é o primeiro algoritmo de alinhamento global por programação dinâmica a ser formulado. Este algoritmo atualmente tem sido alvo de fortes estudos acadêmicos no campo da Bioinformática unida à programação de alto desempenho.

Avaliando-se o algoritmo de Needleman-Wunsch e os vários estudos em arquiteturas distintas no meio acadêmico, pôde-se deduzir que para todos esses trabalhos, por mais que sejam utilizadas abordagens estruturais diferentes, a essência para se explorar o paralelismo desse algoritmo é utilizar o método *wavefront*.

Para as estratégias propostas nesse trabalho deve-se evitar ao máximo uso de cláusulas condicionais e alocação de estruturas de dados grandes em memória, excetuando-se a matriz de comparações.

5 **VERSÕES PARALELAS IMPLEMENTADAS NESTE TRABALHO**

Neste capítulo são abordadas as versões paralelizadas do algoritmo NW propostas pelos autores, assim como experimentos e comparações entre as mesmas.

Este trabalho propõe implementações em paralelo do algoritmo de Needleman-Wunsch usando dois paradigmas de programação paralela. O primeiro, denominado MIND, será implementado usando uma arquitetura de memória compartilhada manipulada pela API OpenMP. O segundo implementará o algoritmo com base no paradigma SIMD, utilizando uma arquitetura de GPU manipulada pela linguagem CUDA.

5.1 **OpenMP**

Para a implementação em OpenMP das versões em paralelo do algoritmo NW, foram propostas duas versões distintas. A primeira possui uma granularidade fina, isto é, o problema original de preenchimento da matriz de comparações é decomposto em muitos subproblemas a serem computados em paralelo. A segunda versão do algoritmo em paralelo utiliza o conceito de particionamento da matriz de comparações em blocos, com o objetivo de aumentar a granularidade dos subproblemas.

5.2 **Implementação em OpenMP utilizando granularidade fina**

Nesta versão do algoritmo a etapa de preenchimento da matriz de comparações utilizada pelo algoritmo de NW, foi decomposta em muitos subproblemas capazes de serem computados em paralelo.

Conforme descrito em seções anteriores, a forma mais comum de se extrair o paralelismo de problemas de alinhamento de sequências por programação dinâmica

é preencher a matriz de comparações por suas diagonais utilizando um método conhecido como *wavefront method.* Neste método, cada elemento de uma diagonal poderá ser processado em paralelo.

Para preencher a matriz de comparações por suas diagonais deve-se aplicar o procedimento descrito no capítulo anterior. Nesse procedimento são criados dois laços para percorrerem a matriz. O primeiro laço varia de $0\ (zero)$ até o número de diagonais $(m + n - 1)$. O segundo laço possui um número variável de iterações que pode ter seu comportamento deduzido com base na diagonal que se encontra.

Este método de preenchimento pode ser decomposto em três etapas distintas para matrizes não quadráticas. Na primeira, o número de elementos cresce para cada iteração do laço externo. Na segunda, o número de iterações para o laço interno se mantém para as próximas $(m - n)$ iterações, onde m é a maior dimensão da matriz e n é a menor. Na última etapa, o número de iterações do laço interno decresce até chegar a $1\ (um)$, isto é, quando o laço externo chegar até $(m + n - 1)$.

Para matrizes quadradas o comportamento dos laços internos pode ser decomposto em apenas duas etapas, onde na primeira o número de iterações para este laço cresce, e na segunda ele diminui até atingir a célula $[m \times n]$ da matriz.

Com o objetivo de se evitar o uso de cláusulas condicionais para determinar como seria o comportamento do laço interno, a matriz teve sua dimensão menor acrescida até possuir o mesmo valor da maior dimensão. Ou seja, a matriz de comparações foi transformada em uma matriz quadrada para evitar atrasos provenientes de desvios. Esses atrasos na implementação em OpenMP não causariam muita perda de desempenho, mas teriam grande impacto na implementação em CUDA.

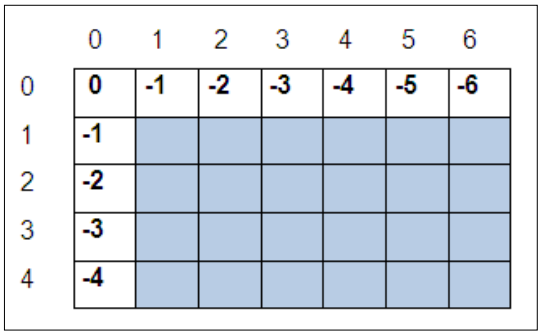
Esta tática aumenta a complexidade espacial do algoritmo NW de $O(nm)$ para $O(n^2)$, onde n é a maior dimensão da matriz. Porém, diminui o tempo gasto pela implementação em OpenMP e viabiliza uma possível implementação em GPU.

Esta complexidade não apresenta um desperdício de espaço muito grande, pois como esta implementação visa uma versão paralela do algoritmo de alinhamento global, está ira comparar sequências de tamanhos semelhantes. (MOUNT, 2004)

O método *wavefront* só pode ser utilizado para preencher a parte interna da matriz, ou seja, todas as linhas e colunas excetuando-se a primeira linha e coluna.

Isto ocorre porque estas são preenchidas durante a etapa de inicialização com valores referentes à penalização de inserção de *gaps*.

Por este motivo, as iterações do preenchimento em diagonais devem ser percorridas somente para a matriz mais interna. Ao invés de se fazer um laço que variasse de 0 a $(m+n-1)$, o laço variaria de 0 até $(m+n-3)$, o que seria aplicar a fórmula anterior para uma matriz de dimensões $[(m-1)\times(n-1)]$.



	0	1	2	3	4	5	6
0	**0**	**-1**	**-2**	**-3**	**-4**	**-5**	**-6**
1	**-1**						
2	**-2**						
3	**-3**						
4	**-4**						

Figura 27: Exemplo da matriz de comparações inicializada com valores de *gaps* igual a -1. A parte em azul representa a matriz mais interna.

Portanto, para percorrer a matriz de comparações por completo serão necessárias duas combinações de dois laços: um externo para percorrer as diagonais e outro interno para percorrer os elementos das diagonais. A primeira combinação terá seu laço externo variando de 0 até $(m-1)$ (quantidades de diagonais na parte superior da matriz, com a diagonal principal da matriz inclusa). O laço interno dessa primeira combinação terá seu número de iterações acrescido até o fim da execução do laço externo.

Na segunda combinação, o laço externo irá variar de m até $(m+n-3)$, e a cada iteração deste laço, o número de iterações do laço interno diminuirá até chegar a apenas um elemento.

Nesta primeira implementação foi utilizada uma versão de granularidade fina que decompunha o problema original em muitos subproblemas, onde cada subproblema é uma diagonal da matriz de comparações a ser computada em paralelo por várias *threads*. Neste caso, cada *thread* instanciada é responsável por preencher um elemento da diagonal corrente. Estas *threads* são instanciadas para

cada iteração do laço externo pelo sistema operacional, ou seja, a cada processamento de uma nova diagonal.

Para avaliar esta implementação foram criados três casos de testes: um caso de teste classificado como pequeno, onde eram usadas sequências de 18.000 nucleotídeos; outro denominado médio, onde foram usadas sequências de 45.000 nucleotídeos; e o último caracteriza um caso grande, onde são usadas sequências de 100.000 nucleotídeos.

Todos os casos de testes foram executados variando a quantidade de *threads*. No primeiro cenário, os casos de testes foram executados com 2 *threads.* Já no segundo e terceiro cenários, foram utilizadas 4 e 6 *threads* respectivamente. Posteriormente, o número de *threads* foi acrescido em 6 para cada execução até chegar ao número de 24 *threads.*

Este procedimento foi feito para avaliar a performance da versão paralelizada usando 4 processadores de 6 núcleos cada (24 *threads* em paralelo), 3 processadores de 6 núcleos cada (18 *threads* em paralelo), 2 processadores de 6 núcleos cada (12 *threads* em paralelo) e 1 processador de 6 núcleos cada (2, 4 ou 6 *threads* em paralelo).

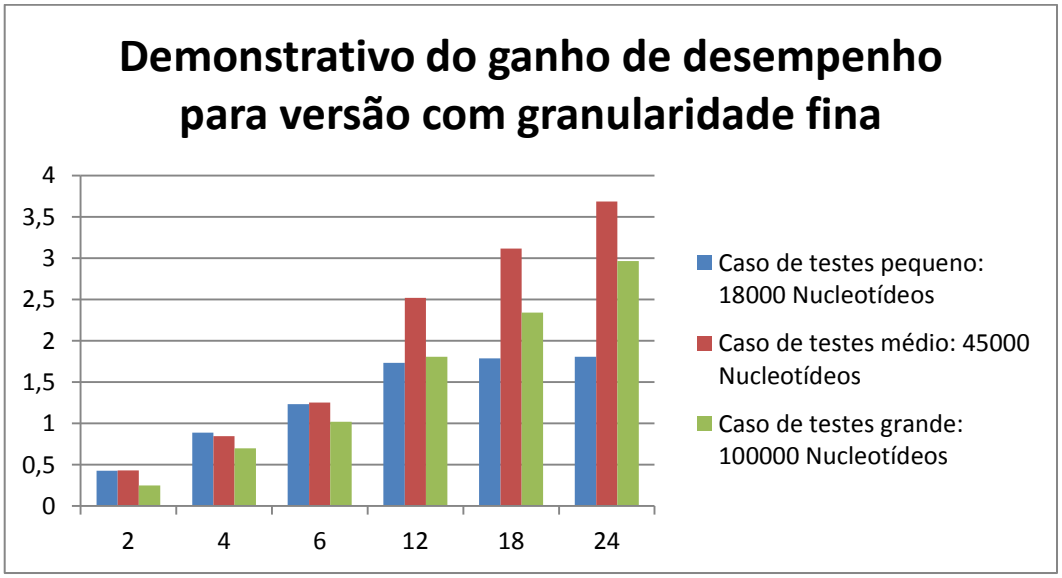


Figura 28: Gráfico demonstrando ganho de desempenho para paralelização do algoritmo utilizando granularidade fina. Cada barra representa um caso de teste, o eixo y representa o *speedup* atingido para cada execução, enquanto o eixo x representa a quantidade de *threads* utilizada em cada execução do algoritmo paralelizado.

Baseado nos resultados coletados, os quais foram demonstrados no gráfico da figura 28, pôde-se observar que no caso de testes médios para esta implementação, o *speedup* alcançado era superior ao alcançado no caso de testes grande. Este comportamento é esperado, pois para cada diagonal haverá criação e destruição de *threads,* o que gera perda de desempenho para os dois casos de testes: médio e grande. No entanto, o número de diagonais em uma matriz gerada pela comparação de duas sequências de 45.000 nucleotídeos é bastante inferior a quantidade de diagonais geradas pela matriz de comparações para sequências de 100.000 nucleotídeos.

Analogamente, haverá também uma quantidade de elementos superior para as diagonais pertencentes à matriz de comparações gerada no caso de testes grandes. Esta implementação em paralelo do algoritmo possui um melhor ganho de desempenho para casos médios do que para casos grandes, pois apresentava uma melhor razão entre o tempo gasto para criação e destruição *threads* e o tamanho de elementos a serem processados na matriz de comparações.

Com base no gráfico (Figura 28), pode-se observar que somente partir da utilização de 6 *threads* houve algum tipo de ganho de desempenho. Isto ocorre por consequência da granularidade fina dessa implementação. Para cenários com menos de 6 *threads*, a execução da versão paralelizada se aproximava de uma execução sequencial. Devido aos gastos de tempo para se criar e destruir *threads* a cada nova diagonal preenchida (o que não ocorre em uma versão sequencial), o tempo de execução dessa versão paralelizada foi superior ao tempo de execução da versão sequencial.

Ao se criar 12, 18 ou 24 *threads,* apesar do aumento do número de *threads* que deveriam ser criadas e destruídas para cada diagonal da matriz de comparações, mais elementos da diagonal poderiam ser preenchidos em paralelo. Dessa forma, o ganho de desempenho aumenta e o tempo de execução diminui.

Nesta primeira implementação os resultados de desempenho foram inferiores à versão original do algoritmo de Needleman-Wunsch para os cenários com 2 e 4 *threads*. Isto ocorre apesar de cada elemento de uma diagonal corrente poder ser preenchido por uma *thread* independentemente, pois a cada iteração do laço, as *threads* são recriadas e o grau de paralelismo para estes cenários é inferior aos

demais. Nos cenários de 18 e 24 *threads* houve um *speedup* considerável para os casos médio e grande. Como o caso médio apresenta melhor equilíbrio em relação ao número de elementos a serem processados em paralelo e tempo de criação e destruição de *threads,* este apresentou um melhor ganho em relação aos testes com sequências de 100.000 nucleotídeos.

5.3 **Implementação em OpenMP utilizando blocos**

Para superar as dificuldades impostas pela implementação anterior e obter um ganho maior de desempenho, foi proposta a decomposição da matriz de preenchimento mais interna (sem contar com a primeira linha e coluna) em submatrizes de tamanho pré-determinados denominadas blocos. Um bloco consiste em uma partição de tamanho pré-definido da matriz de comparações, onde as linhas e colunas deste bloco são iguais.

Desta forma, foi possível transformar a matriz de comparações em uma matriz virtual com um número menor de elementos a serem processados. Por sua vez, esses elementos menores da matriz virtual (blocos) possuem o mesmo número de elementos em suas linhas e colunas.

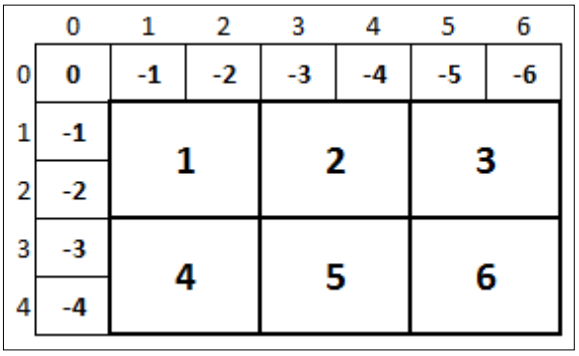


Figura 29: Exemplo do particionamento de uma matriz de comparações em blocos de dimensão $[2 \times 2]$.

Ao particionar uma matriz em blocos de dimensões $[b \times b]$, a matriz de comparações não ficará dividida em partes iguais caso as dimensões da mesma não

forem múltiplas de b. Por este motivo, para dimensões da matriz de comparações que não são múltiplas de b, é acrescido um valor de $(b - (m \; mod \; b))$, onde m é a dimensão da matriz que não era múltipla de b e mod representa o operador de módulo.

Durante a etapa de preenchimento essa matriz é processada por completo, porém o procedimento de *backtrack* somente começa do elemento da matriz que armazenara a pontuação para o alinhamento.

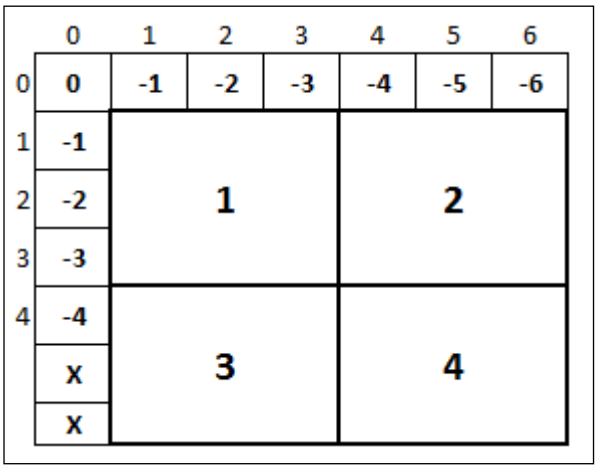


Figura 30: Exemplificação da matriz de comparações da figura 29 quando esta tenta ser particionada em blocos de tamanho $[3 \times 3]$. O número de colunas da matriz interna é múltiplo de 3 e não precisa ser acrescido, porém o número de linhas não. Desta forma é necessário criar duas linhas extras na matriz para que essa configuração de blocos seja usada $((3 - (4 \; mod \; 3)) = 2)$.

Assim como implementado na versão anterior de granularidade fina, para eliminar cláusulas condicionais que determinam o comportamento do laço mais interno, a matriz virtual também foi transformada em uma matriz quadrada de dimensões $[(m + b - (m \; mod \; b)) \times (m + b - (m \; mod \; b))]$, onde m representa a maior dimensão não múltipla de b da matriz de comparações e mod representa a operação de módulo.

Utilizando esta técnica, a complexidade espacial do algoritmo aumenta para $((m + b - (m \; mod \; b))^2)$. O método *wavefront* é então aplicado para a matriz virtual,

isto é, os blocos pertencentes a uma diagonal desta matriz são processados em paralelo por *threads* diferentes.

Os elementos dos blocos são computados de forma sequencial como seria em uma versão tradicional do algoritmo Needleman-Wunsch. Desta forma o número de vezes que *threads* são instanciadas durante a execução do algoritmo decresce para $\left(\frac{m+n-3}{b}\right)$, e trabalho realizado por cada uma é maior, o que gera uma queda significativa no tempo de execução do algoritmo.

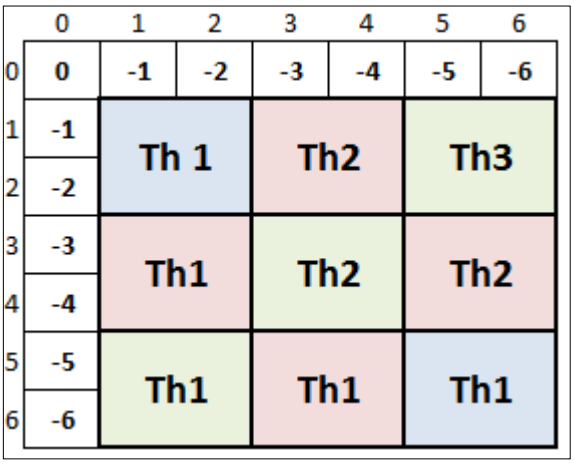


Figura 31: Exemplificação do método *wavefront* para uma matriz de blocos. A cada nova diagonal mais blocos poderão ser preenchidos em paralelo por *threads* distintas, porém ao passar da diagonal principal o grau de paralelismo decresce simetricamente inverso a forma como antes cresceu.

A versão em paralela implementada usando OpenMP do algoritmo NW proposta neste trabalho foi executada em uma máquina com quatro *chips* AMD *Six-Core* Opteron™ 8425 HE 2100 MHz (total de 24 núcleos) e 64 GB de RAM DDR-2 667MHz (16 x 4GB). O sistema operacional usado foi o GNU/Linux (*kernel* 2.6.31.5-127 64bits).

Para avaliação de performance dessa implementação, primeiramente foram criados três casos de testes: o primeiro caso para sequências de tamanho pequeno – 1.000 pares de bases –, o segundo para sequências de tamanho médio – 45.000

pares de bases –, e o terceiro para sequências de tamanho grande – 100.000 pares de bases.

Depois de definidos os casos de teste, foram feitas execuções da versão em paralelo do algoritmo para várias *threads,* variando-se o tamanho de bloco utilizado.

A primeira bateria de testes foi feita com 24 *threads* para todas as configurações de blocos, ou seja, os quatro processadores disponíveis eram utilizados nesta primeira execução. A segunda bateria de testes utilizava 18 *threads* também para todas as configurações de blocos, isto é, desta vez só eram utilizados três processadores dos quatro disponíveis. A terceira e quarta bateria de testes seguia a lógica das anteriores e executavam o algoritmo para todas as configurações de blocos utilizando 12 *threads* (2 processadores foram usados) e 6 threads (apenas 1 processador foi usado). A quinta e sexta bateria foram feitas com 4 *threads* e 2 *threads* respectivamente.

Desta forma foi possível confirmar qual configuração de blocos possui um melhor ganho de desempenho para determinado cenário e determinar a variação de ganho de desempenho em função do número de *threads* utilizado. Este procedimento foi efetuado para os três casos de testes.

5.3.1 Caso de teste pequeno: 18.000 nucleotídeos

Para o primeiro caso de testes de 18.000 nucleotídeos, foi escolhido pelos autores particionar a matriz de comparações em blocos de dimensões $[18 \times 18]$ para a primeira execução do algoritmo e $[360 \times 360]$ para a última. As dimensões dos blocos foram acrescidas a uma taxa constante de 18 unidades para cada execução do algoritmo, totalizando 20 execuções $(360 \div 18 = 20)$ da mesma versão em paralelo do algoritmo para diferentes configurações de blocos. Estas execuções foram então repetidas para os diferentes cenários de *threads*.

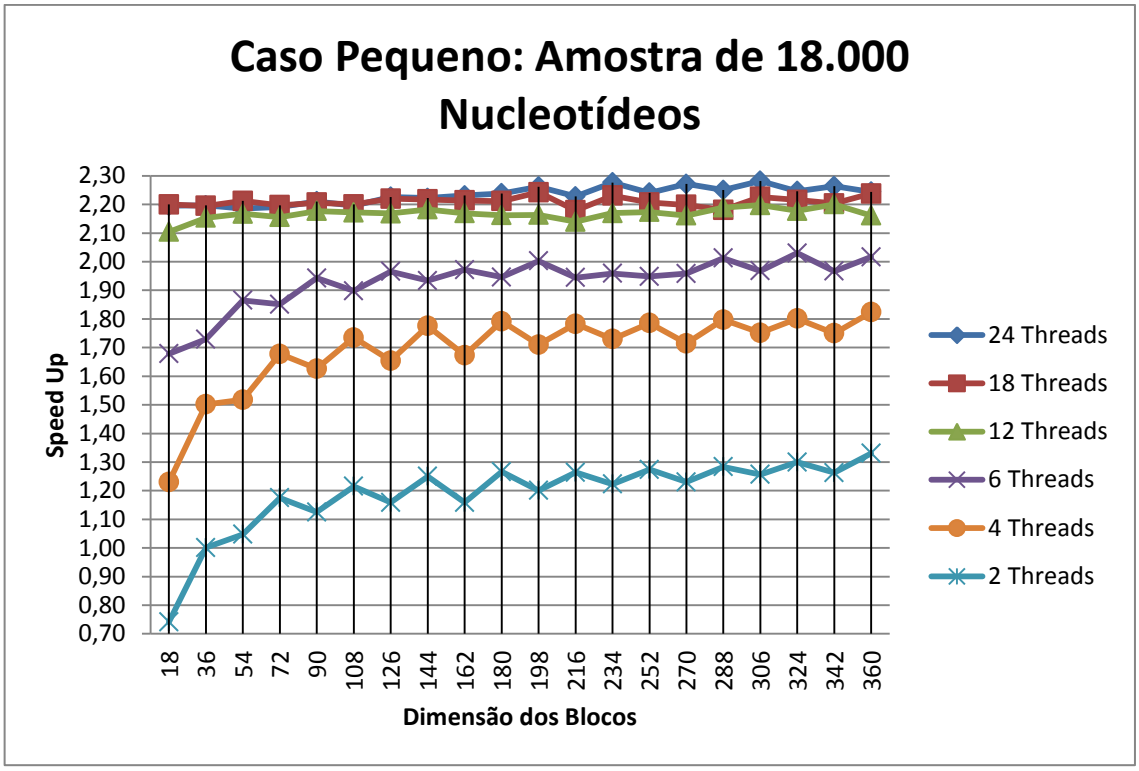


Figura 32: Gráfico demonstrando os resultados do ganho de desempenho para variação do tamanho de blocos por cenários de *threads*. Cada curva no gráfico representa um cenário de *threads.* O eixo y representa o *speedup* alcançado para determinada execução, enquanto o eixo x representa o tamanho do bloco para alcançar o *speedup*.

Pode-se observar no gráfico apresentado na figura 32 uma oscilação no *speedup* atingido para cada cenário diferente de *threads.* Este comportamento ocorre porque para blocos não múltiplos das dimensões da matriz é necessário completar a mesma com linhas e colunas auxiliares para permitir uma execução correta da paralelização do algoritmo proposta nesta seção, aumentando desta forma o trabalho a ser resolvido em paralelo.

É possível notar para este caso de teste um grande salto relativo à variação de *speedup* apresentada entre o cenário de execução com 2 *threads* e 4 *threads.* Entretanto para os cenários com 12, 18 e 24 *threads* os *speedups* atingidos foram relativamente próximos. Isto ocorre porque neste caso de teste a variação do tamanho de blocos é pequena, o que gera desbalanceamento com relação ao número de blocos em que a matriz era decomposta e a quantidade de tempo para se instanciar novas *threads*, e também devido ao *overhead* gerado pela comunicação de processadores quando usado mais de 6 *threads*.

Para os cenários de 2, 4 e 6 *threads* pôde-se observar uma grande queda de desempenho na execução do algoritmo com blocos de dimensão $[18 \times 18]$ em relação às demais execuções. Esse comportamento ocorre pelo fato de ter muitos blocos e poucas *threads* sendo utilizadas, o que gera certa aproximação com a versão implementada utilizando granularidade fina. Ao aumentar o tamanho dos blocos pôde-se observar uma melhora no desempenho do algoritmo, pois a granularidade dos subproblemas aumentou, melhorando assim a utilização das *threads* disponíveis.

A partir da análise dos resultados coletados para este experimento foi escolhido o bloco com dimensões $[198 \times 198]$ para testes efetivos de comparação entre a versão sequencial e a em paralelo. Foram executadas cinco vezes para cada cenário de *threads* as versões paralela e sequencial do algoritmo com o intuito de se obter uma média aritmética do tempo gasto por elas. Os tempos mensurados durante a execução do algoritmo foram os tempos referentes ao preenchimento da matriz de comparações e ao total, o qual contempla a própria etapa de preenchimento, assim como a etapa de *backtrack*, alocação de recursos, entre outras.

Posteriormente, com os resultados coletados a partir das execuções da versão em paralelo utilizando granularidade fina e a versão sequencial foram efetuadas análises sobre os ganhos de desempenho alcançados.

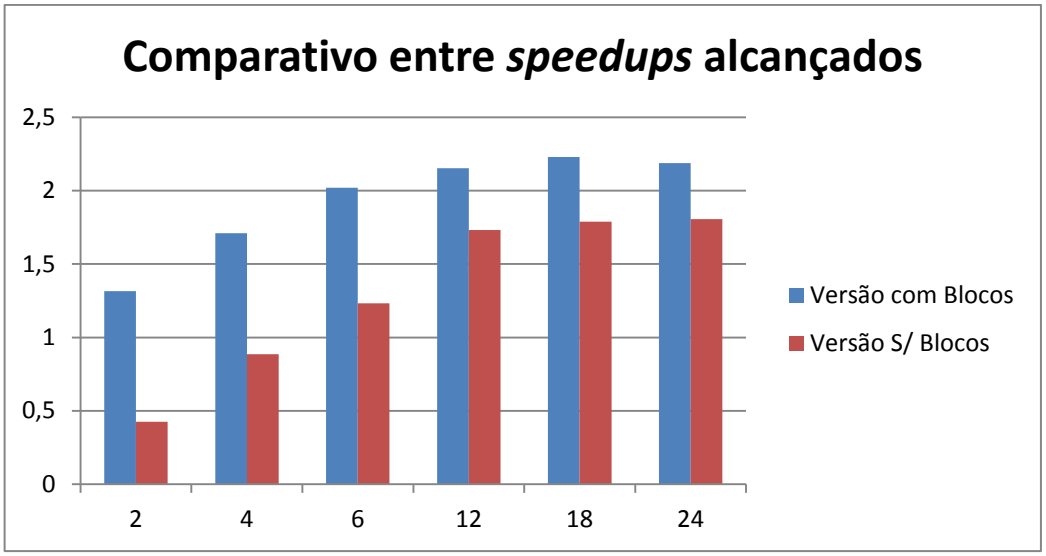


Figura 33: Gráfico em barras demonstrando o *speedup* alcançado entre versões paralelizadas implementadas em OpenMP e a versão tradicional do algoritmo NW. O eixo y representa o *speedup* alcançado e o eixo x representa cada cenário de *threads*. As barras por sua vez representam a versão utilizando particionamento da matriz de comparações em blocos (em azul) e a versão utilizando granularidade fina (em vermelho).

Pôde-se então observar que para o caso onde as sequências alinhadas são pequenas, a implementação proposta nesta seção quando comparada a versão de granularidade fina descrita na seção anterior não escalava bem para os cenários de 18 e 24 *threads*.

O ganho de desempenho para estes casos não variou muito. O aumento para 24 *threads* gerou aproximadamente o mesmo *speedup* em relação à execução utilizando 18 *threads*. Isso acontece porque como as amostras de sequências comparadas para esse caso de teste geram uma matriz relativamente pequena, ao se decompor esta em blocos de $[198 \times 198]$, a quantidade de diagonais que poderiam ter seus blocos processados por todas as *thread s*em paralelo diminui, gerando um desperdício de recursos, já que haveriam poucos blocos para as *threads* processarem.

Por último foi analisado a distribuição temporal do algoritmo paralelizado, isto é, quanto tempo a etapa de preenchimento da matriz de comparações contribui para o tempo total de execução.

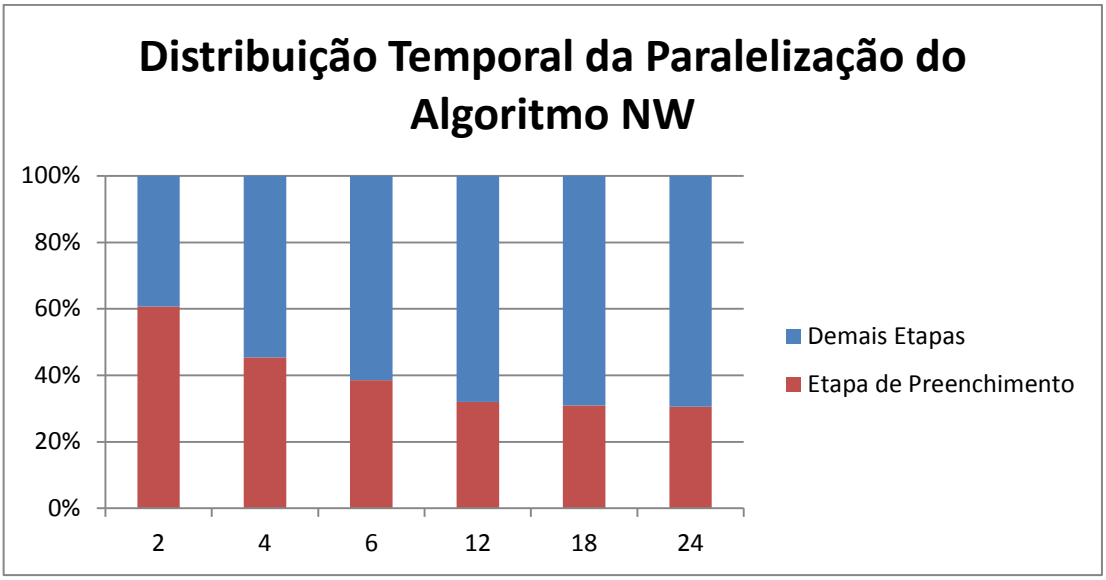


Figura 34: Gráfico demonstrando a distribuição temporal da versão paralelizada do algoritmo NW nesta seção. O eixo y representa a porcentagem do tempo total consumida por cada fase do algoritmo e o eixo x representa os diferentes cenários de *thread.* A parte vermelha da barra representa o consumo de tempo realizado pela etapa de preenchimento das matrizes e a parte azul, as demais etapas (alocação de recursos, etapa de *backtrack*, entre outras).

Com base nestes gráficos pôde-se analisar que para o cenário de 18 e 24 *threads* o tempo gasto pelo algoritmo na etapa de preenchimento da matriz foi reduzido em aproximadamente 30% em relação ao tempo consumido por esta mesma etapa na versão sequencial. Isto ocorre porque somente esta etapa foi paralelizada, as demais mantêm o seu padrão de tempo. Em alguns casos passam a consumir mais tempo do que a etapa de preenchimento da matriz.

Com base neste caso de teste pode-se concluir que para testes onde a quantidade de dados não é massiva, isto é, onde as sequências a serem comparadas são pequenas, o *speedup* em média alcançado não é muito superior ao tempo de execução do algoritmo original. Os resultados mostram que em todos os cenários de *threads* o desempenho do algoritmo é melhorado se utilizado subproblemas de granularidade mais grossa.

5.3.2 Caso de teste médio: 45.000 nucleotídeos

Para este caso de testes, de forma análoga à primeira bateria de testes, foi escolhida pelos autores uma configuração de blocos na qual a matriz de comparações é particionada. No caso médio, para a primeira execução do algoritmo a dimensão do bloco é de $[45 \times 45]$. Na última execução os blocos teriam dimensão de $[900 \times 900]$. Os valores destas dimensões são acrescidos em 45 unidades de forma constante a cada execução do algoritmo, e de forma análoga ao primeiro teste, totalizava em 20 execuções da versão paralelizada do algoritmo descrito nesta seção, cada qual com configurações de blocos distintas.

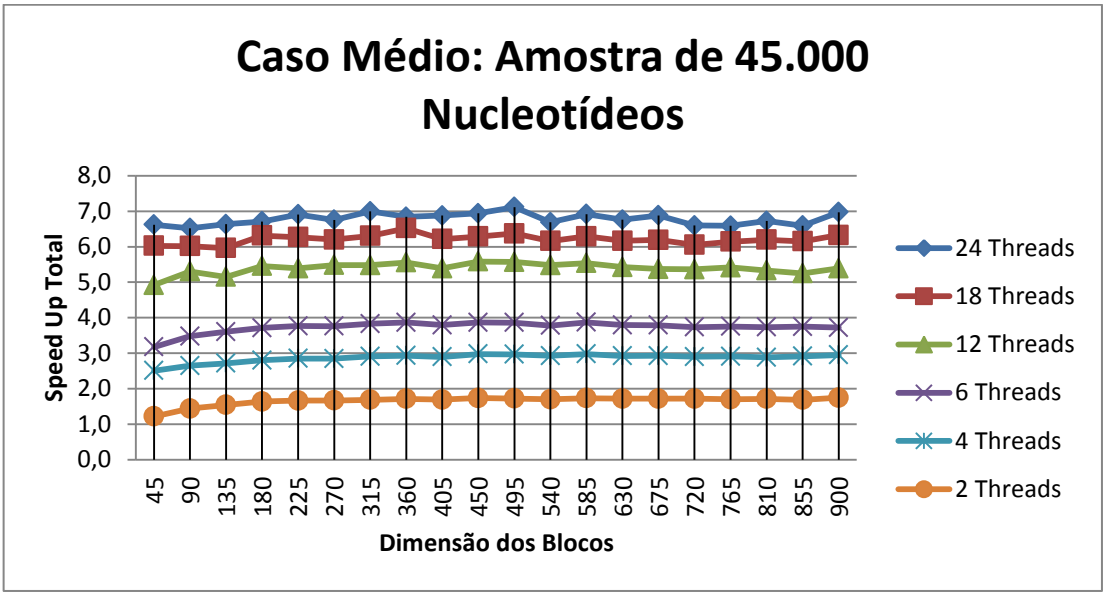


Figura 35: Gráfico demonstrando os resultados do ganho de desempenho para variação do tamanho de blocos por cenários de *threads*. Cada curva no gráfico representa um cenário de *threads.* O eixo y representa o *speedup* alcançado para determinada execução, enquanto o eixo x representa o tamanho do bloco para alcançar o *speedup*.

Neste caso de teste, com base nos resultados coletados é possível notar que para todos os cenários de *threads* existe certo ganho de desempenho, ao contrário do que ocorria no experimento anterior. Isto ocorre porque a matriz é dividida em muitos blocos de dimensões maiores do que a utilizada no experimento anterior, gerando assim um trabalho mais balanceado para as *threads*.

Baseando-se nos resultados coletados, pôde-se reparar que o comportamento das curvas, as quais representam os cenários de execução com 2, 4, 6 e 12 *threads*, possuem um comportamento menos oscilatório em relação aos resultados obtidos no experimento descrito na seção anterior. Para esses números de *threads,* ainda que o trabalho a ser processado tenha aumentado devido à adaptação da matriz para comportar os blocos não múltiplos, não houve variação no *speedup.* Isso acontece porque o trabalho acrescido é bem comportado para essa quantidade de *threads*.

O comportamento não oscilatório também pode ser possivelmente explicado pelo fato do *speedup* atingido para este caso de testes ser muito superior ao alcançado no caso de testes anteriores. Sendo assim, conforme a matriz de comparações aumente para comportar uma determinada configuração de blocos, o trabalho extra necessário para preenchê-la se mantêm razoavelmente no mesmo nível de ganho de performance em relação à versão serial do algoritmo.

Para os cenários com 18 e 24 *threads* deduz-se que houve picos para determinada configuração de blocos porque estes, por não serem múltiplos das dimensões da matriz, aumentam o tamanho do problema a ser resolvido em paralelo (o preenchimento da matriz). Pelo fato da quantidade de elementos a serem computados tenha aumentado, o *overhead* gerado para haver mais *threads* foi compensado pela melhor distribuição de trabalho.

A partir deste caso de testes foi escolhido pelos autores utilizar a configuração de blocos de dimensão $[495 \times 495]$ porque esta a princípio apresenta um maior *speedup*. Essa configuração foi utilizada para efetuar os testes comparativos com a versão sequencial do algoritmo. Após cinco execuções da versão paralelizada do algoritmo para essa mesma configuração de blocos e usando diferentes cenários de *threads*, uma média aritmética do tempo consumido pelo algoritmo foi calculada, este procedimento também foi adotado para se obter o tempo médio consumido pelo algoritmo sequencial.

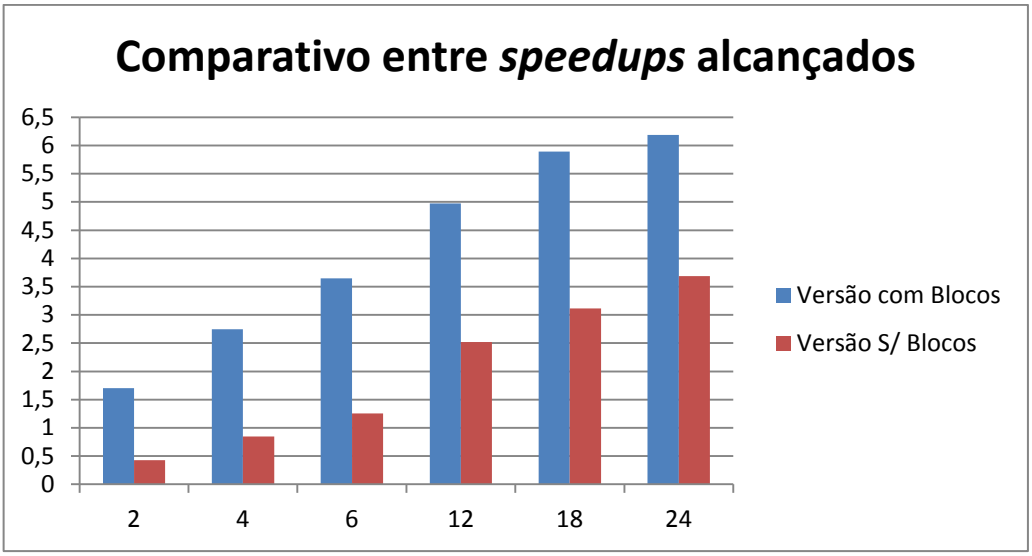


Figura 36: Gráfico em barras demonstrando o comportamento de *speedup* atingido em relação às duas implementações paralelas em OpenMP do algoritmo NW. O eixo y representa o *speedup* alcançado por cada cenário de *thread* representado no eixo x. As barras representam a versão paralela do algoritmo, matriz particionada em blocos (em azul), e implementação usando granularidade fina (em vermelho).

Com base nos resultados apresentados no gráfico da figura 36 é possível validar que a implementação usando estratégia de blocos ainda possui desempenho superior à outra. Para casos de testes maiores há uma grande diferença entre os ganhos de desempenho alcançados por cada versão. O *speedup* atingido para a versão em blocos do algoritmo é próxima ao dobro do *speedup* atingido pela versão que usa granularidade fina para esse mesmo caso de testes.

Pode-se reparar também que a taxa de crescimento do *speedup* cai quando são utilizadas 18 e 24 *threads.* É possível que, para este caso, um número maior de *threads* não gere um balanceamento entre blocos a ser processados por cada *thread*, comunicação entre os processadores e gerenciamento de *threads*, capazes de gerar um *speedup* linear. Também poderia ocorrer diminuição de trabalho para cada *thread*, pelo fato de haver mais *threads.*

Para este experimento também foi analisada a distribuição temporal das etapas que compõe o algoritmo NW.

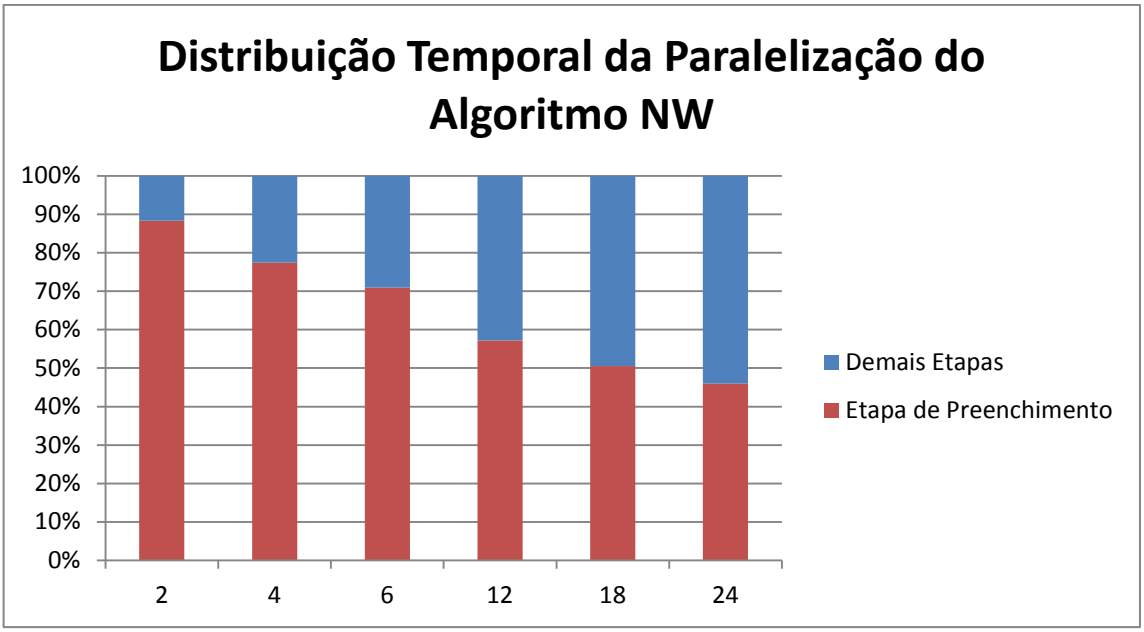


Figura 37: Gráfico em barras representando a distribuição temporal do algoritmo NW para os vários cenários de *threads.* O eixo y representa a porcentagem do tempo total consumida por cada fase do algoritmo e o eixo x representa os diferentes cenários de *thread.* A parte vermelha da barra representa o consumo de tempo realizado pela etapa de preenchimento das matrizes e a azul as demais etapas (alocação de recursos, etapa de *backtrack*, entre outras).

Com base neste gráfico é possível observar que no caso com menos *threads* a maior parte de tempo gasto pela execução do algoritmo se dá na fase de preenchimento, semelhantemente a versão sequencial. Ao aumentar o número de *threads*, mais blocos da matriz são capazes de serem preenchidos em paralelo, diminuindo a contribuição do tempo da etapa de preenchimento consideravelmente na execução do algoritmo, alcançando um *speedup* razoável.

É possível reparar também que o tempo utilizado pela etapa de preenchimento da matriz de comparações, o qual na versão sequencial para este caso de teste ocupava aproximadamente 95,31% do tempo total consumido pelo algoritmo, na versão paralela passou a compor aproximadamente 45,96% do tempo total consumido pela execução da mesma. No entanto, a economia de tempo durante a execução do algoritmo proveniente do *speedup* é limitada pela parte não paralelizada da aplicação (como enunciado pela Lei de Ambdal).

Para o caso de testes médio (sequências de 45.000 nucleotídeos) os *speedups* atingidos foram satisfatórios e muito superiores aos alcançados pela

versão sem utilizar blocos. Isto ocorre porque o número de vezes que *threads* são criadas e destruídas é menor, e ainda, o trabalho feito por uma *thread* para preencher um bloco de elementos é mais vantajoso do que desperdiçar o poder de processamento da mesma com o preenchimento de apenas um elemento da matriz.

5.3.3 Caso de teste grande: 100.000 nucleotídeos

Neste caso de testes o mesmo procedimento dos dois anteriores foi adotado. Primeiramente se determina uma configuração de blocos final e inicial, depois se varia o tamanho de blocos em razão constante e o cenário de *threads* em 2, 4, 6, 12, 18 e 24 *threads* executando em paralelo. Para cada execução do algoritmo paralelizado se determina o *speedup* atingido.

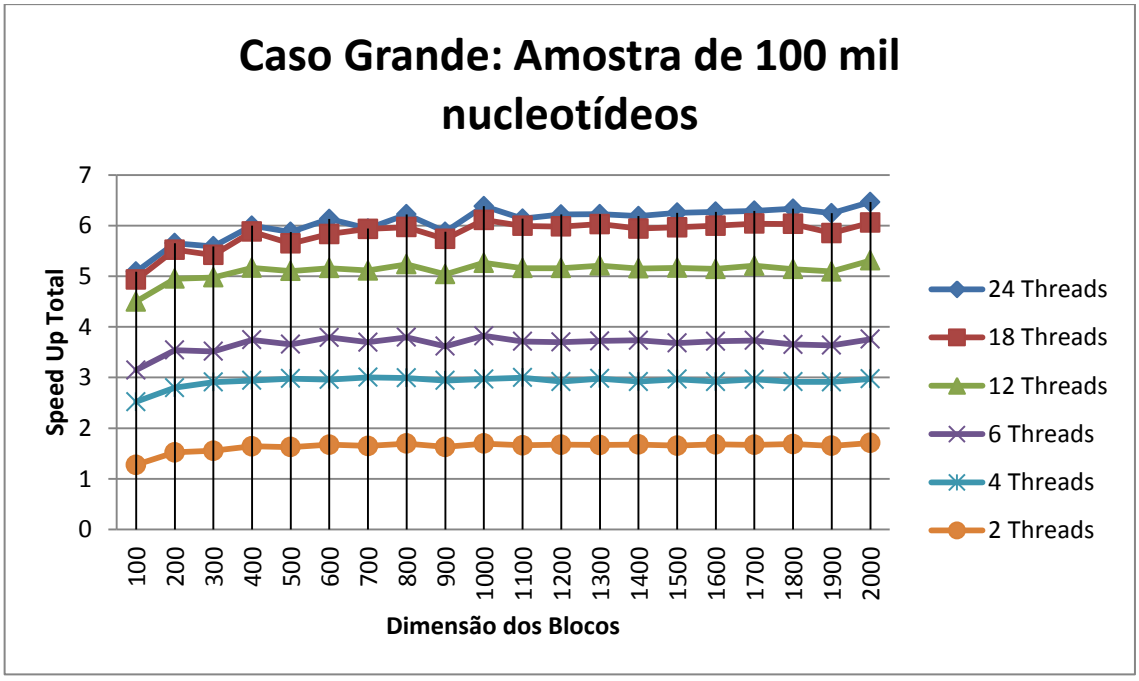


Figura 38: Gráfico demonstrando os resultados do ganho de desempenho para variação do tamanho de blocos por cenários de *threads*. Cada curva no gráfico representa um cenário de *threads.* O eixo y representa o *speedup* alcançado para determinada execução, enquanto o eixo x representa o tamanho do bloco para alcançar o *speedup*.

De acordo com os resultados do experimento, observa-se que para os cenários de 2 e 4 *threads* não haviam muitas oscilações ao longo da variação de blocos. Os cenários com 6 e 12 *threads* possuem alguns pontos oscilantes, porém ficou visível maiores oscilações para os cenários de 18 e 24 *threads*.

É provável que para alinhamentos de sequências dessa magnitude onde são utilizadas poucas *threads* para processamento paralelo, a possível perda de desempenho ocasionada pelo aumento da matriz de comparações em relação a blocos não múltiplos é suavizada pelo fato de não haver muita comunicação entre processadores.

Pode ser ressaltado também que menos tempo seria gasto com comunicação. Isto confirma o fato de que para mais números de *threads* há maiores oscilações, pois para estes cenários quando o tamanho da matriz aumentar, maior a necessidade de acessos à memória de longa latência proveniente da comunicação de diferentes processadores em uma arquitetura de memória compartilhada.

Com base nos resultados coletados pode ser validado que em todos os diferentes cenários de *threads*,quando executados com a configuração de blocos mínima $[100 \times 100]$, o *speedup* é bastante reduzido em relação às demais execuções para diferentes tamanhos de blocos. Este comportamento pode ser explicado possivelmente pelo fato da divisão da matriz em blocos de 100, diante de uma matriz de $[10^5 \times 10^5]$, apresentar uma granularidade razoavelmente fina. Isso geraria muito trabalho para pequenas quantidades de *threads* gerando uma queda no desempenho.

Para cenários onde existem mais *threads* executando em paralelo, há mais tempo gasto durante a execução para criar e destruir *threads*, assim como maior taxa de comunicações entre processadores.

É possível observar também que para os cenários de 18 e 24 *threads* o *speedup* alcançado não variou muito. Isto se dá porque essa implementação não apresenta um *speedup* linear, isto é, o aumento do número de *threads* não necessariamente gerará um ganho de desempenho. Essa indeterminação se dá porque o trabalho realizado por mais *threads* também implica em mais tempo gasto para comunicação, criação e manutenção entre elas.

Levando em consideração os resultados obtidos, foi determinado pelos autores usarem a configuração de blocos de dimensões $[1000 \times 1000]$. Essa configuração gera o melhor *speedup* e também permite que a matriz de

comparações possua 100 blocos que podem ser computados em paralelo em sua maior diagonal.

De forma análoga aos experimentos anteriores, para este também foi feito cinco execuções do algoritmo de NW comum e a versão paralelizada com as configurações de bloco escolhidas. Em seguida, foram calculadas as médias aritméticas dos tempos de execução para as duas versões do algoritmo.

Esta versão apresentou um *speedup* máximo de aproximadamente $6{,}3$ em relação ao algoritmo sequencial de NW. Também foi possível validar sua superioridade em relação à implementação utilizando granularidade fina.

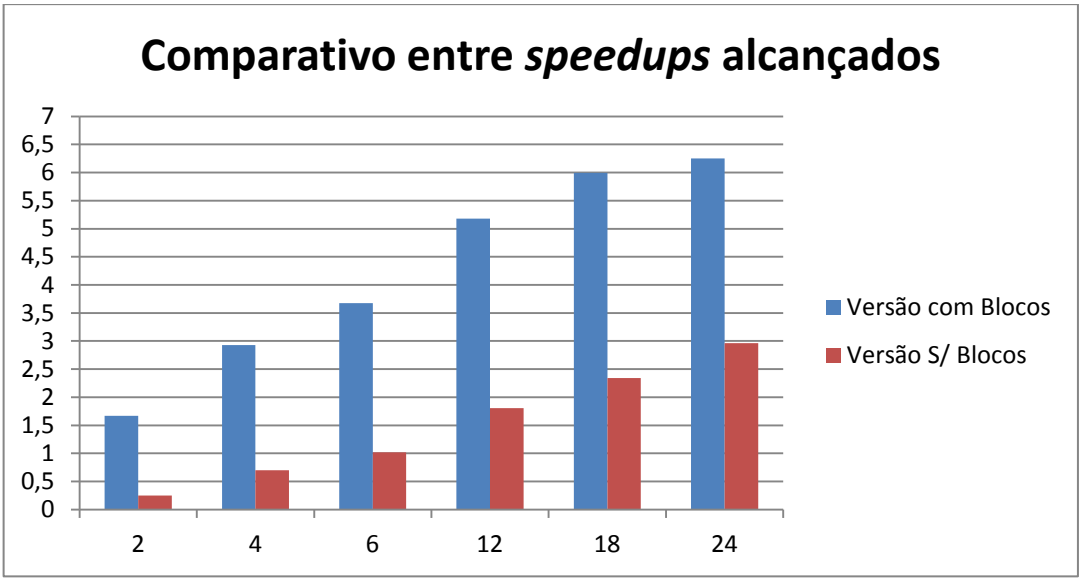


Figura 39: Gráfico em barras demonstrando o comportamento de *speedup* atingido em relação às duas implementações paralelas em OpenMP do algoritmo NW. O eixo y representa o *speedup* alcançado por cada cenário de *thread* representado no eixo x. As barras representam a versão paralela do algoritmo, matriz particionada em blocos (em azul), e implementação usando granularidade fina (em vermelho).

Para estas amostras os resultados de ganho de desempenho foram em média superiores ao dobro dos ganhos realizados pela versão utilizando granularidade fina. Semelhantemente ao caso de testes com amostra de 45.000 nucleotídeos, se os subproblemas a serem processados em paralelo forem muitos e houver necessidade de criação de *threads* para cada novo conjunto de subproblemas, isto é, a cada nova diagonal da matriz a ser processada, o ganho de performance será reduzido. Essa

diminuição de desempenho pode ser explicada pela grande parte do tempo dedicado à gastos do sistema para criar, destruir e promover comunicação entre *threads.*

De forma análoga ao experimento de carga média com 45.000 nucleotídeos, este caso de teste também manteve seu *speedup* máximo em torno de 6 vezes. Pode-se deduzir que o *speedup* máximo não pôde escalar muito, ainda que para uma carga de dados bastante superior, pois nestes dois casos foi encontrado o melhor balanceamento possível entre número de *threads* e carga de trabalho para esta paralelização do algoritmo NW.

Devido ao preenchimento da matriz de comparações ter sido paralelizado, e também ter seu consumo de tempo reduzido, pôde-se afirmar que essa etapa consume menos tempo para ser completada do que sua versão original e que a distribuição temporal do algoritmo NW é diferente da encontrada para sua versão serial graças a esse fato.

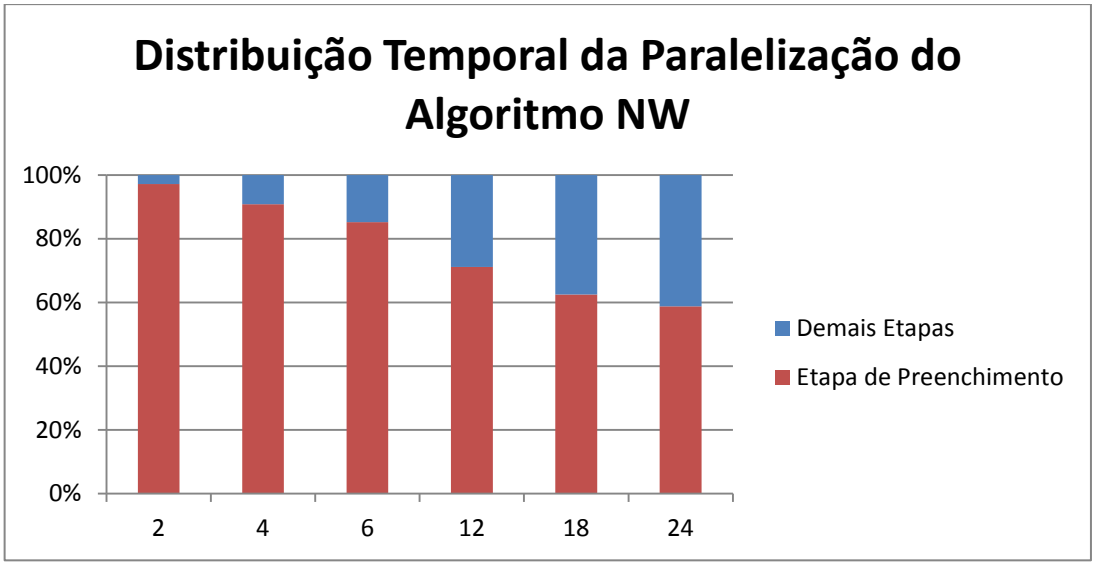


Figura 40: Gráfico em barras representando a distribuição temporal do algoritmo NW para os vários cenários de *threads.* O eixo y representa a porcentagem do tempo total consumida por cada fase do algoritmo e o eixo x representa os diferentes cenários de *thread.* A parte vermelha da barra representa o consumo de tempo realizado pela etapa de preenchimento das matrizes e a parte azul, as demais etapas (alocação de recursos, etapa de *backtrack*, entre outras).

Pôde-se observar com base nos resultados coletados que o *speedup* no melhor caso reduzia o consumo de tempo pela etapa de preenchimento em aproximadamente 40% quando comparado com a distribuição temporal proveniente do algoritmo tradicional de NW.

Pelo mesmo motivo do experimento com amostras de 45.000 nucleotídeos, é possível notar que como a etapa do algoritmo paralelizada obteve um *speedup* considerável, este deixou de contribuir fortemente para o alto consumo de tempo do algoritmo.

Para casos grandes onde a matriz gerada pelas sequências a serem alinhadas ultrapassam uma alocação em memória de 32GB, a paralelização do algoritmo oferece *speedups* satisfatórios. Entretanto, a taxa de ganho de desempenho não aumentou em relação ao caso de testes anterior, e nem mostrou um ganho muito expressivo de *speedup* entre execuções com 18 e 24 *threads.*

5.4 **Conclusões sobre paralelização utilizando OpenMP**

Utilizando uma arquitetura de memória compartilhada manipulada pela API OpenMP, foi possível verificar, para essa implementação, que *speedups* satisfatórios para alinhamentos de sequências de tamanho médio (45.000 nucleotídeos) e grande (100.000 nucleotídeos) podem ser alcançados se utilizado subproblemas de granularidade grossa.

Deve-se, no entanto, manter sempre um bom equilíbrio para o tamanho da granularidade dos subproblemas, como também utilizar uma boa proporção entre número de *threads* instanciadas e cargas de trabalho por *threads*. Caso contrário, poderá haver um *overhead* desnecessário com administração de *threads* e comunicação entre diferentes processadores através de acessos à memória principal do sistema.

5.5 **Implementação da versão paralela do algoritmo NW em CUDA**

Na implementação em CUDA utilizando arquitetura de GPUs proposta nesse trabalho, a estratégia utilizada para a paralelização em OpenMP foi mantida. Foram criadas basicamente duas versões paralelas do algoritmo de NW: uma decompondo a etapa de preenchimento da matriz de comparações em subproblemas independentes com granularidade fina, e outra utilizando subproblemas de granularidade mais grossa.

Em CUDA, as *threads* instanciadas durante a execução de um *kernel* são organizadas em blocos. Para que não haja ambiguidade em relação aos conceitos utilizados na versão paralela do algoritmo de granularidade grossa, já que está refere-se às partições da matriz em blocos, o termo bloco agora servirá para identificar blocos de *threads* organizados pelo *hardware* da GPU. Os blocos que representam partes da matriz de comparações serão denominados partições.

Para ambos os casos, o mesmo método de preenchimento em paralelo é utilizado: o *wavefront method*. Este método, com base em estudos de trabalhos anteriores, demonstrou ser a forma mais eficiente de se explorar a natureza paralela do algoritmo de Needleman-Wunsch.

Deve-se ressaltar também que algoritmos de alinhamento de sequências por programação dinâmica consomem bastante memória, e como placas gráficas disponíveis no mercado possuem pouca capacidade de memória, não foi possível testar as versões paralelas para cargas de dados massivas.

5.5.1 Múltiplas chamadas ao *kernel*

O algoritmo de Needleman-Wunsch, em sua etapa de preenchimento da matriz de comparações, possui uma grande dependência de dados. Para que cada elemento da matriz seja preenchido utilizando o *wavefront method* é necessário que se conheça as duas diagonais anteriores à diagonal correntemente processada.

No início de execução de um *kernel* a quantidade de *threads* informada por parâmetros específicos da função é criada. Neste momento, todas as *threads* simultaneamente executam o mesmo fluxo de instruções escritas no *kernel*. Considerando a dependência de dados existentes na primeira etapa do algoritmo, e

que cada *thread* estaria responsável por preencher um elemento, haveriam *threads* processando erroneamente elementos da matriz de comparações. Isto ocorre pelo fato de algumas *threads* não possuírem ainda os dados referentes às duas diagonais anteriores. Desta forma um preenchimento inválido da matriz de comparações seria efetuado, impossibilitando assim o retorno do alinhamento ótimo.

Por esse motivo, foi escolhido neste trabalho implementar várias chamadas ao *kernel.* Dessa forma, todas as *threads* instanciadas em uma chamada à GPU podem ser utilizadas em somente uma parte da matriz. Utilizando subproblemas de granularidade fina, cada chamada ao *kernel* é utilizada para preencher uma diagonal da matriz. Assim, todas as *threads* instanciadas durante a chamada são utilizadas para computar um elemento da diagonal corrente.

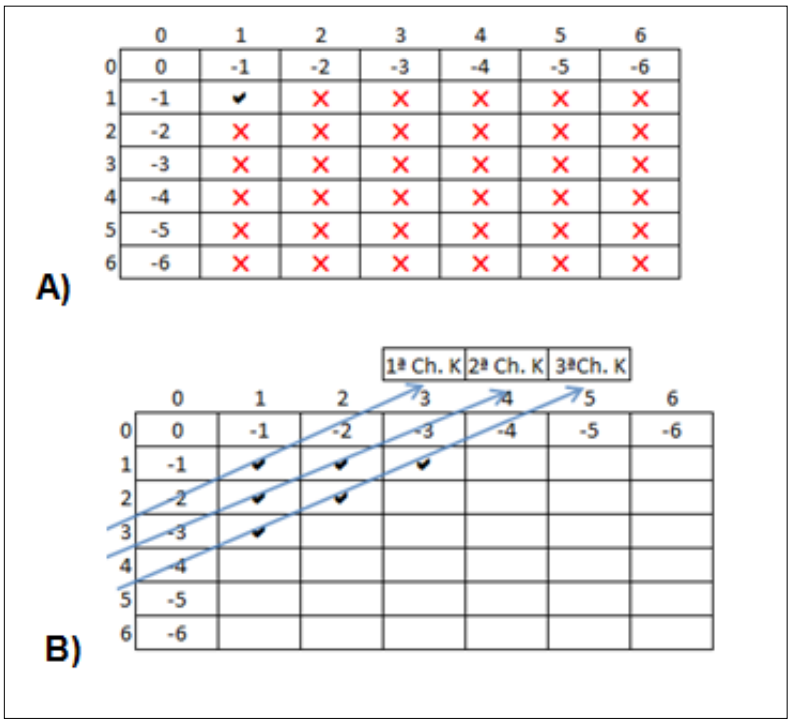
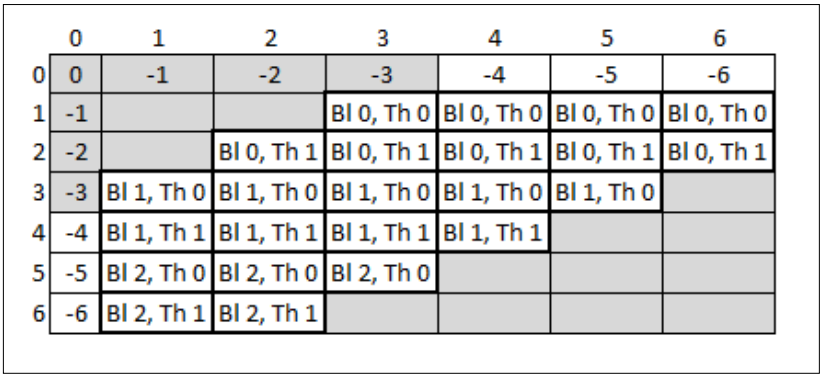


Figura 41: A) Exemplificação do preenchimento da matriz quando o *kernel* é invocado uma única vez. As células com um "X" em vermelho representam os elementos que são preenchidos invalidamente. A célula com o símbolo "✔" representa o único elemento que é preenchido de forma correta. B) Exemplificação do preenchimento da matriz com várias chamadas ao *kernel*. Cada seta em azul representa uma chamada e as legendas 1ª Ch. K, 2ª Ch. K, 3ª Ch. K representam 1ª, 2ª e 3ª chamadas ao *kernel* respectivamente.

Utilizando a estratégia descrita no parágrafo anterior não foi possível averiguar nenhum tipo de ganho de desempenho em relação à versão sequencial do algoritmo. Isto pode ser explicado pelo fato do grau de paralelismo da etapa de preenchimento da matriz de comparações crescer de forma gradual, e depois decrescer. Por esse motivo, o número total de *threads* instanciadas só seria utilizado no caso da maior diagonal, isto é, apenas uma vez durante a execução do algoritmo.

Nas primeiras e últimas diagonais o grau de paralelismo é muito inferior às demais diagonais ao ponto de justificar uma chamada ao *kernel* com muitas *threads*, pois geraria uma subutilização muito grande destas e também haveria desperdício de tempo com a criação de várias *threads* que não seriam utilizadas. Por este motivo, foi proposto preencher as primeiras diagonais no *host*, e quando o grau de paralelismo atingisse um nível aceitável, o processamento seria feito pela GPU. De forma semelhante, quando o grau de paralelismo decaísse ao ponto de não ser mais vantajoso o custo de utilização de varias *threads,* o processamento seria feito pela CPU novamente.

Com o intuito de melhorar o uso da GPU blocos de *threads* foram organizados de forma a preencher trechos da diagonal corrente, isto é, cada *thread* processava o elemento de um trecho da diagonal corrente de acordo com seu bloco de origem. Desta forma seria possível que mais SMPs fossem utilizados, e assim aumentaria o uso da placa gráfica.



	0	1	2	3	4	5	6
0	0	-1	-2	-3	-4	-5	-6
1	-1			Bl 0, Th 0	Bl 0, Th 0	Bl 0, Th 0	Bl 0, Th 0
2	-2		Bl 0, Th 1	Bl 0, Th 1	Bl 0, Th 1	Bl 0, Th 1	Bl 0, Th 1
3	-3	Bl 1, Th 0	Bl 1, Th 0	Bl 1, Th 0	Bl 1, Th 0	Bl 1, Th 0	
4	-4	Bl 1, Th 1	Bl 1, Th 1	Bl 1, Th 1	Bl 1, Th 1		
5	-5	Bl 2, Th 0	Bl 2, Th 0	Bl 2, Th 0			
6	-6	Bl 2, Th 1	Bl 2, Th 1				

Figura 42: Divisão da tarefa de preencher elementos da matriz de comparações entre blocos de *threads.* Os blocos são representados pela legenda "Bl Nº", onde Nº representa o número do bloco. Analogamente as *threads* em um bloco são representadas pela legenda "Th Nº". As células cinza representam os elementos que foram processados no host. Esta figura exemplifica o preenchimento da matriz utilizando 3 blocos com 2 *threads* cada.

Apesar dessas tentativas o tempo de execução do algoritmo na GPU demonstrou-se semelhante ao tempo de execução da versão sequencial. Este fator poderia ser explicado pela granularidade dos subproblemas ainda estar muito fina e haver necessidade de muitas chamadas ao *kernel*. Para diminuir a quantidade de chamadas foi proposta a utilização de subproblemas com granularidades maiores. De forma análoga à implementação feita em OpenMP, a matriz de comparações foi dividida em partições de submatrizes de dimensões iguais. Desta forma foi criada uma matriz virtual com menos elementos.

Estas partições deveriam possuir dimensões múltiplas da matriz de comparações, a fim de se evitar aumento da matriz para acomodar os blocos. Isso foi feito pois o espaço de memória em uma GPU é muito reduzido, e não foi considerado nesta implementação possuir matrizes maiores do que a original.

Semelhantemente às outras versões paralelizadas do algoritmo propostas neste trabalho, o método de preenchimento da matriz em paralelo é feito através do *wavefront method*, porém ao invés de cada diagonal da matriz de comparações ser processada em paralelo, nesta versão a diagonal preenchida em paralelo contemplava partições da matriz original.

Para este caso blocos de *threads* instanciados a cada chamada do *kernel* foram utilizados para preencher um conjunto de partições da matriz de comparações em paralelo. Desta forma o uso dos SMPs seria maximizado e as capacidades da placa gráfica seriam melhores utilizadas.

Semelhantemente a implementação em GPU utilizando granularidade fina, o grau de paralelismo no início e no fim da execução do algoritmo era muito pequeno. Desta forma foi preferível fazer o processamento das primeiras e últimas diagonais de partições da matriz no *host*, quando graus de paralelismos maiores fossem alcançados a matriz passaria a ser computada na GPU.

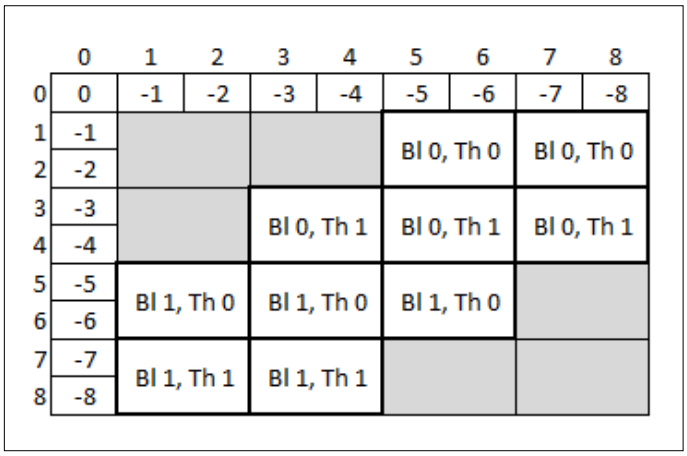


Figura 43: Divisão de tarefas de processamento das partições da matriz de comparações para blocos de *threads*. Mesma esquematização utilizada na figura 42. Este exemplo demonstra o preenchimento da matriz de comparações $[8 \times 8]$ quando esta é particionada em submatrizes de dimensões $[2 \times 2]$. Cada bloco de *thread* é responsável por preencher um conjunto de partições.

A carga de trabalho para cada *thread* foi aumentada, já que estas agora eram responsáveis por processar uma partição da matriz, ao invés de um elemento da diagonal. Com esta estratégia o número de chamadas ao *kernel* foi reduzido melhorando o *overhead* causado por uma chamada ao *kernel*. Porém, ainda sim, não foi alcançado um ganho de desempenho expressivo em relação às versões em CUDA anteriormente implementadas e também em relação à versão serial do algoritmo.

Esta falta de ganho, ainda que utilizando uma carga de trabalho maior para as *threads* e menor quantidade de chamadas ao *kernel,* pode ser possivelmente explicada pelo fato dos problemas submetidos à GPU não serem muito grandes, não compensando desta forma o custo de múltiplas chamadas ao *kernel*. Outro motivo se daria ao fato de haver muitos acessos à memória global da GPU, e também possíveis divergências de controle dentro do *kernel*.

5.5.2 Investigação de divergências de controle no *kernel*

Em funções executadas em GPU cláusulas condicionais, as quais causam divergência de controle, podem causar serialização das instruções em execução. (KIRK; HWU, 2010)

Na arquitetura *manycores* desenvolvida pela NVIDIA quando existem cláusulas condicionais dentro de um *kernel*, pode ocorrer serialização do código. Este comportamento ocorre porque quando há divergências de fluxo de execução, como no caso de um *if-else*, todas as *threads* que satisfizerem uma condição seguirão um fluxo de instruções enquanto as outras que não satisfizeram essa condição ficarão em espera. Posteriormente as *threads* que estavam em espera executarão suas instruções. Isso gera uma grave perda de paralelismo na aplicação. (KIRK; HWU, 2010)

No algoritmo de Needleman-Wunsch para efetuar a etapa de preenchimento da matriz de comparações é necessário descobrir o máximo entre o valor das três células da matriz adjacentes somadas com bônus ou penalidades ao elemento sendo processado. Desta forma deve existir uma função condicional do tipo: Se $a > b$, retorna a, caso contrário retorne b.

Um agravante no ganho de desempenho poderia estar ocorrendo por conta desta função dentro do *kernel.* Para averiguar esse argumento foi investigado em nível de *assembly* a chamada da instrução "max", a qual era responsável por encontrar o valor máximo necessário para preencher um elemento da matriz de comparações.

Primeiramente foi gerado o código em *assembly* do programa escrito em CUDA. Feito isso, foi investigado no arquivo ptx (tipo de arquivo referente ao código

em *assembly* de CUDA) o trecho que descrevia o comportamento da função “max”. Segue abaixo trecho do código em *assembly* referente à função “max”:

```
max.s32  %r38, %r35, %r37;
max.s32  %r39, %r32, %r38;
st.global.s32  [%rd17+8], %r39;
```

No código em *assembly* pôde-se notar que o trecho referente à chamada da função “max” dentro do *kernel* era representado por uma função *assembly* também chamada “max”. Esta função poderia estar gerando divergência de controle durante a execução do algoritmo na GPU, resultando assim em perda de desempenho.

Para melhorar a utilização da arquitetura foram retiradas as cláusulas condicionais relacionadas à seleção do valor máximo de cada célula adjacente ao elemento computado. Através dos operadores comparativos de ">", "<", "≤" e "≥", os quais retornam 1 se a condição for satisfeita, ou 0 se as condições não forem satisfeitas, foi utilizada as seguintes operações para encontrar o valor máximo:

$$max = \Big(\big((op2 \geq op3) \times op2\big) + \big((op3 > op2) \times op3\big)\Big) \quad (10)$$

$$max = (\big((max \geq op1) \times max\big) + \ ((op1 > max) \times op1)) \quad (11)$$

Nas fórmulas acima, $op1$, $op2$ e $op3$ são os valores das células adjacentes ao elemento computado somado às suas penalidades ou bonificações, e max representa o valor máximo entre dois elementos. Pode-se avaliar que se $op2$ for maior que $op3$, o valor retornado será 1, enquanto o valor retornado pela segunda parcela da expressão será 0. Desta forma a expressão ficará como $max = (1 \times op2 + 0)$ que é igual ao próprio $op2$, este é o valor máximo entre $op2$ e $op3$. Posteriormente basta utilizar o resultado da expressão (10) com o valor da terceira célula adjacente, aplicar a mesma fórmula, e será descoberto o valor máximo dos três elementos.

Quando gerado um novo *assembly* a partir do código com as modificações acima, o trecho no *assembly* referente à função “max” foi substituído por operações aritméticas referentes à fórmula (11).

Utilizando este método foi revelado um pequeno ganho em relação as outras versões que utilizavam cláusulas condicionais dentro do *kernel*. Apesar de ter havido

um ganho, este não foi muito superior à versão sequencial do algoritmo, os tempos ficaram quase iguais.

5.5.3 Utilização de várias configurações de memórias da placa gráfica

Placas gráficas da NVIDIA possuem em sua arquitetura sistemas de memória que podem ser utilizados pelo programador para melhorar o desempenho de seu programa. As memórias da placa gráfica utilizadas nesse trabalho foram:

1) Memória Global: Esta memória DRAM possui uma grande latência, porém possui uma grande capacidade de armazenamento. Quando *threads* fazem acesso de leitura e escrita a esta memória, *schedulers* conhecidos como *warp schedulers* escalonam as *threads* que fizeram esse tipo de acesso, e permitem que novas *threads* utilizem os CUDA *cores* de um SMP.
2) Memória compartilhada: Esta é uma memória cache que reside dentro de um SMP e possuí visibilidade para todas as *threads* dentro de um bloco de *threads*. Todas as *threads* de um mesmo bloco podem ler e escrever nessa memória. A memória compartilhada possui um tamanho reduzido, mas por se tratar de uma cache, sua latência é muito baixa se comparada às memórias globais das placas gráficas.
3) Memória Constante: Memórias do tipo constante possuem uma latência e capacidade menores que as memórias globais das placas gráficas. O *host* é o único que pode escrever na memória constante, enquanto *threads* só possuem acesso de leitura a esta. A memória constante é indicada para dados do problema computado na GPU que não sofreram mudanças e serão acessados simultaneamente por todas as *threads*.
4) Registradores ou memória local: Essas memórias são visíveis para todas as *threads*, e cada *thread* possui uma cópia dos dados presentes nelas. Memória local é a memória que possui acesso mais rápido em uma GPU, consequentemente possui a capacidade de armazenamento mais reduzida também.

Neste trabalho foi proposta a utilização de várias configurações de memórias a fim de se avaliar algum possível ganho de desempenho, o qual poderia estar sendo impedido de ser alcançado por conta de uma má organização de acesso às memórias pelas *threads*.

Em uma primeira implementação todos os dados utilizados durante a execução do algoritmo pelas *threads* (matriz de comparações, tabela de pontuação, o par de sequências de nucleotídeos) estavam armazenados na memória global. Para que possivelmente pudesse-se alcançar um melhor *speedup*, dados de natureza constante e razoavelmente pequenos foram alocados em outras memórias de menores latências.

Em uma segunda implementação a tabela de pontuação foi alocada na memória compartilhada dos SMPs, desta forma, ela estaria dentro do chip de processamento e possuiria um tempo de acesso menor pelas *threads* de um mesmo bloco. Ainda assim não houve ganho significativo para justificar que essa modificação pudesse efetuar algum *speedup* na aplicação.

Como a tabela de pontuação foi modelada propositalmente para possuir o menor tamanho possível e ser acessada de forma rápida pelas *threads,* esta pode ainda sim ser copiada para cada *thread* instanciada e alocada nos registradores das *threads.* Essa seria a forma de acesso mais rápida que uma *thread* poderia ter a tabela de pontuação. Nesta versão o ganho foi um pouco superior à descrita no parágrafo anterior, porém nada significativo ao ponto de justificar essa nova organização como solução para *speedups* baixos.

Na seguinte estratégia as sequências de nucleotídeos, as quais nunca seriam modificadas no decorrer da execução do algoritmo, foram alocadas na memória constante da placa gráfica. Desta forma as *threads* somente necessitariam fazer uma menor quantidade de leituras e escritas na memória global. Ainda sim o desempenho obtido não foi superior aos obtidos anteriormente.

Foram feitas então combinações dessas várias formas de organização de dados em diferentes memórias, mas em nenhum caso houve um *speedup* expressivo capaz de identificar qual estratégia de alocação de dados seria a mais vantajosa para o problema.

5.5.4 Utilizando a técnica *Zero Copy* para executar problemas maiores na GPU

A técnica de *Zero Copy*, como o nome sugere, consiste em resolver problemas na GPU, porém sem fazer cópia de dados para a memória global. Desta forma o modelo tradicional de programação em GPUs, no qual o *host* copia dados da memória principal para a memória global do *device*, é modificado. Com *Zero Copy* é possível ponteiros passados como parâmetros para o *kernel* referenciar estrutura de dados presentes na memória principal de uma máquina.

Para permitir o acesso do processador gráfico diretamente a memória RAM, primeiramente deve-se mudar a forma de alocação de dados nessa mesma memória. Tradicionalmente, na linguagem C, a forma como dados são alocados na memória principal é feito via função *malloc()*, e posteriormente a liberação desse espaço de memória é feito via outra função chamada *free()*. No caso de acesso direto à memória do *host* pela GPU NVIDIA, a função *cudaHostMalloc()* deve ser utilizada. Esta aloca um espaço na memória principal do sistema, porém a página alocada na memória do *host* se mantém fixa e não poderá ser realocada para outro endereço por ação do sistema operacional.

Esse método de alocação mantém uma região da memória do *host* sincronizada com a memória da GPU sem requerer intervenção do programador ou funções explicitas para copiar dados entre memórias do *host* e do *device*. Por este motivo o método de *Zero Copy* pode vir a gerar ganho de desempenho em determinadas aplicações pois a cópia de dados é feita implicitamente de forma assíncrona.

Foram testadas as versões de granularidade fina e grossa com o modelo *Zero Copy* utilizando a mesma máquina e amostras usadas para os testes das outras versões paralelas implementadas em CUDA neste trabalho.

Apesar de ser possível com este método executar problemas maiores, o tempo de execução do algoritmo paralelo excedeu o tempo de execução da versão sequencial. Essa perda de desempenho pode ser facilmente explicada pela alta latência de um acesso direto à memória principal do sistema pela GPU. O fato de que um processador gráfico necessitar de transferências de dados através de um barramento PCI-EXPRESS para acessar a memória principal do sistema gera um grande consumo de tempo para obter os dados alocados na memória do *host.*

A estratégia de *Zero Copy* pode ser possivelmente melhor utilizada em máquinas onde a GPU e a CPU são integradas e compartilham a mesma memória, como no caso de placas-mãe com vídeo *onboard*.

5.5.5 Possíveis problemas para esta implementação em arquiteturas *manycores*

Neste trabalho foi efetuado um *profiling* da aplicação - técnica para visualizar comportamento de uma aplicação durante sua execução - para descobrir o quanto de recursos da placa gráfica a versão paralelizada do algoritmo consumia durante sua execução.

Foi descoberto que para todos os casos ainda que se variasse a configuração de *threads* e blocos de *threads* o uso dos SMPs era de aproximadamente 10%. Este resultado do *profiling* confirmava que alguns SMPs eram poucos utilizados, ou até mesmo durante toda execução do algoritmo, alguns SMPs ficava em *idle*, isto é, sem serem utilizados.

Este comportamento pode ser explicado pela modelagem utilizada na implementação para dividir as tarefas de preenchimento da matriz de comparações em blocos de *threads.*

Na implementação proposta neste trabalho a divisão das tarefas em blocos de *threads* era efetuada da seguinte forma: um bloco de *threads* possui um número de *threads*. Cada *thread* é responsável por preencher um elemento da diagonal corrente ou, no caso da versão utilizando granularidade grossa, uma diagonal contendo partições da matriz de preenchimento. Conforme a execução do algoritmo avançasse na GPU o número de elementos em uma diagonal aumenta.

Para este caso, todas as *threads* de um bloco computaria um elemento, no momento em que houvesse mais elementos para serem computados do que o número de *threads* por blocos, então um segundo bloco é utilizado. Desta forma, somente na diagonal máxima, a qual ocorre uma única vez na execução, todos os SMPs seriam utilizados. Essa modelagem de divisão de tarefas gerou uma grande subutilização dos SMPs na placa gráfica.

Outro problema presente nesta implementação poderia possivelmente ser atribuído à forma como o acesso à memória era feito, não em relação às latências

dos diferentes tipos de memória em uma placa gráfica NVIDIA, mas sim à forma como os dados ficavam distribuídos na memória global.

Nesta implementação, a matriz de comparações foi linearizada. Desta forma, os elementos de uma diagonal estavam espaçados pelo vetor que representava essa matriz.

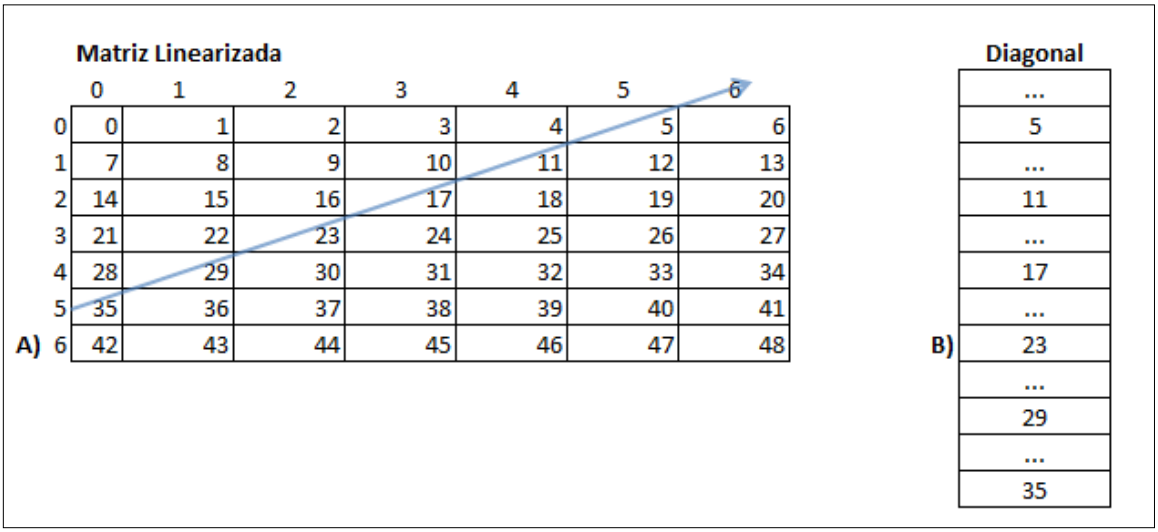


Figura 44: A) Exemplificação da matriz de comparações linearizada. O valor de cada célula da matriz representa sua posição em um vetor unidimensional. B) Representação da disposição dos elementos da diagonal destacada em (A) na memória global da GPU.

Durante o processamento dos elementos em paralelo, varias *threads* estariam acessando elementos da matriz, os quais estariam distribuídos pela memória global. Este fator geraria uma subutilização da *cache*, pois a cada acesso de alta latência, elementos vizinhos dos vetores seriam copiados para esta memória. Pelo fato dos elementos de uma diagonal estarem espaçados na matriz linearizada, a cada processamento de elementos um acesso de alta latência deverá ser feito para copiar esses dados para memória *cache*. Além disso, um conjunto de *threads* administradas por um *warp* serão sempre escalonadas quando ocorrerem acessos de alta latência, desta forma haveriam constantemente escalonamentos de *threads* em um SMP, resultando possivelmente em um desempenho não satisfatório da aplicação.

5.5.6 Soluções plausíveis para uma solução do algoritmo NW em CUDA

Um dos grandes fatores que dificultam a implementação e avaliação do algoritmo de NW está relacionada ao tamanho da matriz gerada pela comparação entre duas sequências de nucleotídeos. Uma possível solução para esse problema seria utilizar chamadas assíncronas ao *kernel* e utilizar *streams* para manter uma maior largura de banda nas transferências entre a memória global e memória principal do *host*.

Utilizando este método haveriam sempre dados disponíveis na memória global a serem processados pela GPU, sem haver necessidade de alocar nesta memória a matriz de comparações completa. Desta forma problemas maiores poderiam ser processados na GPU. Ainda sim com chamadas assíncronas ao *kernel* não haveria necessidade de deixar os SMPs ociosos enquanto estes esperam por dados serem organizados pelo *host* e posteriormente copiados para memória global.

Para implementação deste método seria necessário a utilização de páginas fixadas na memória do *host*, e por estas serem copiadas de forma mais eficiente para a memória global da GPU, possivelmente geraria um melhor *throughput* de dados, melhorando assim o desempenho da aplicação.

Visando melhorar também a forma como é distribuído os elementos das diagonais na memória, estas seriam projetadas em um espaço linear antes de serem copiadas para memória global. Desta forma o acesso a esses elementos poderiam ser feitos de forma mais eficiente pelo processador gráfico.

Abstraindo o problema de preenchimento de diagonais da matriz de comparações em um espaço linear, a forma de subdividir tarefas entre *threads* e blocos de *threads* se torna um pouco mais trivial. Neste método cada bloco de *thread* seria responsável por preencher quantidades relativamente iguais de elementos nas diagonais.

Uma possível implementação utilizando *streams* e chamadas assíncronas ao *kernel* é ilustrada abaixo (Figura 45).

Inicialmente, a primeira linha e coluna da matriz de comparações é preenchida. Posteriormente as primeiras diagonais que possuem grau de paralelismo baixo são processadas pelo *host.*

Figura 45: Matriz de preenchimento representada com suas coordenadas. As diagonais em cinza são as que serão processadas no *host*.

Conforme exemplificado na figura 45, é possível reparar que as diagonais 4, 5, 6, 7 e 8 que serão processadas na GPU.

Devido a utilização do método *wavefront* no algoritmo NW é necessário que para processar os elementos de uma diagonal, as duas diagonais imediatamente anteriores devem ser conhecidas. Por esse motivo seria alocada na memória global da GPU uma matriz com três linhas e $m + 2$ colunas. Nessa matriz estariam projetadas linearmente a diagonal corrente e as duas anteriores para possibilitar o processamento desta.

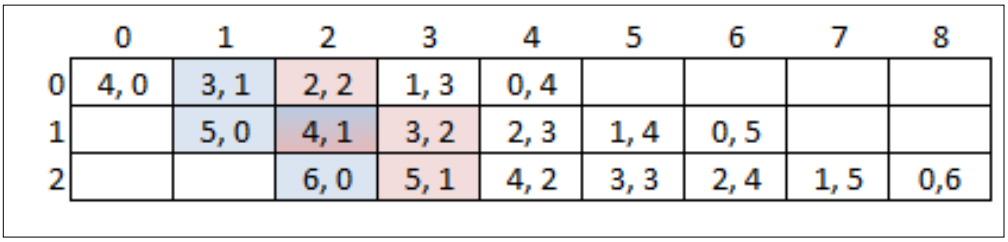


Figura 46: Exemplificação da matriz alocada na GPU com as diagonais linearizadas. Os elementos azuis e rosas na linha de número 2 destacam, como exemplo, dois elementos que serão preenchidos em paralelo. Os elementos em rosa e azul nas linhas 0 e 1 representam os valores que terão de ser conhecidos pelos elementos da mesma cor na linha 2. A célula que possui coloração mista de rosa e azul indica que este elemento será necessário para o preenchimento dos dois elementos em azul ou rosa na linha 2.

Conforme exemplificado na figura 46 a linha de número 2 seria preenchida em paralelo pelas diferentes *threads*. A quantidade de elementos nesta última linha seria dividia de forma igualitária entre os diferentes blocos de *threads* instanciados.

No decorrer do preenchimento desta matriz alocada na GPU haverá elementos que não mais necessitarão estar presentes na mesma. Através de chamadas assíncronas às funções responsáveis por transferirem dados do *device* para o *host* seria efetuada transferência de elementos desnecessária da GPU para o *host*. Paralelamente à execução do *kernel*, o *host* estaria executando instruções para capturar esses elementos e os posicionarem em suas localizações originais na matriz de comparações.

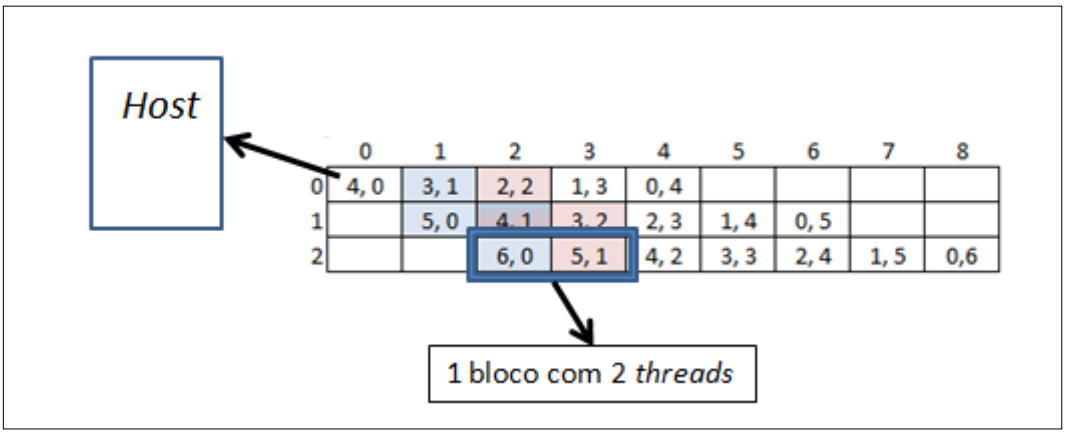


Figura 47: Exemplificação da forma de divisão das tarefas em blocos de *threads*. Exemplificação da transferência de dados da primeira linha da matriz alocada na GPU para o *host.* Esta transferência ocorrerá quando não houver mais necessidade desses elementos estarem na matriz alocada na memória global.

Desta forma haverá uma alimentação contínua de dados para o *host* organizar, assim como, espaços vazios para a GPU computar novas diagonais.

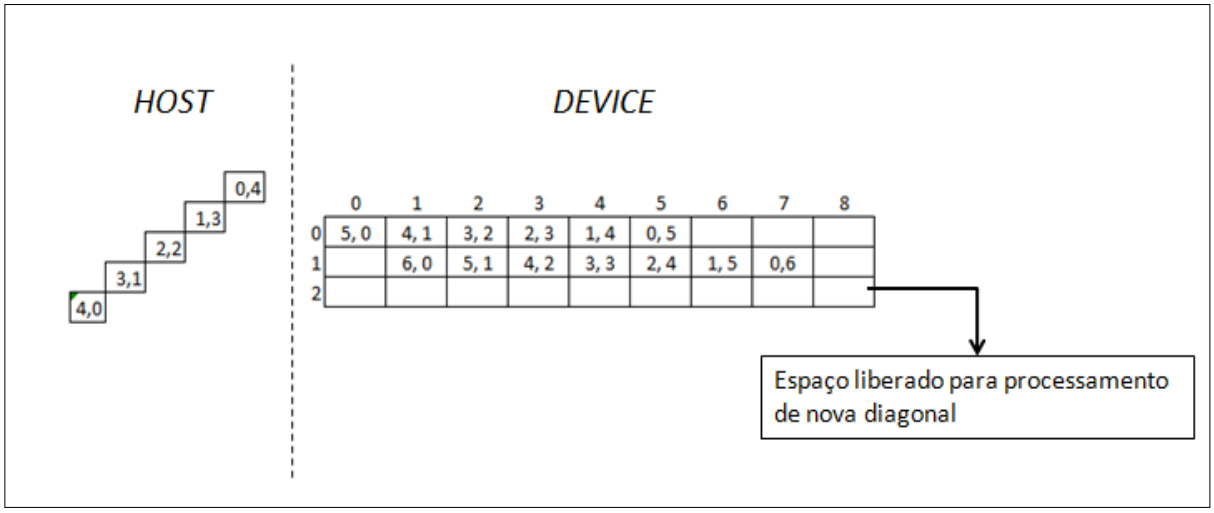


Figura 48: Exemplificação da transferência de dados entre o *host* e o *device.* Quando não houver necessidade dos dados de uma diagonal estarem presentes na matriz alocada na GPU, esta diagonal será copiada para o *host*. O *host* por sua vez reorganizará esses dados de forma a transformá-los novamente em uma diagonal da matriz de comparações. O espaço liberado pela cópia da diagonal mais antiga para o *host* será utilizado para computar uma nova diagonal, esta etapa será feita simultaneamente com o procedimento de reorganização efetuado pelo *host.*

6 **CONCLUSÕES E TRABALHOS FUTUROS**

Neste trabalho foi proposta uma análise de desempenho do algoritmo de alinhamento global de sequências Needleman-Wunsch utilizando conceitos do paradigma de programação paralela e implementações em arquiteturas *multicores* e *manycores.*

O presente capítulo disserta sobre conclusões efetuadas no decorrer deste estudo e possíveis trabalhos futuros a serem desenvolvidos.

6.1 **Conclusões**

O algoritmo de Needleman-Wunsch foi o primeiro algoritmo a utilizar técnicas de programação dinâmica para descobrir um alinhamento ótimo entre um par de sequências.

O algoritmo de Needleman-Wunsch pode ser decomposto basicamente em duas etapas. A primeira etapa responsável pelo preenchimento da matriz de comparações é a responsável por um maior consumo de tempo durante a execução. Neste trabalho a abordagem utilizada foi paralelizar somente esta etapa.

Ganhos foram obtidos utilizando a implementação em arquiteturas *multicores,* possivelmente por estas possuírem maior espaço de memória, que possibilita execução de problemas maiores, e melhor tolerância a divergências de controle. Entretanto não foi possível uma implementação viável na arquitetura *manycore* NVIDIA, pois esta ao contrário da anterior não possui espaço de memória suficiente que possibilite a execução de problemas grandes para esse algoritmo. Outro motivo está relacionado ao fato dessas arquiteturas *manycores* serem criadas a partir do modelo SIMD, onde existe um único fluxo de instruções para múltiplos fluxos de dados, restringindo assim tolerâncias à divergência de controle e dependência de dados.

Os ganhos obtidos não puderam ser melhorados possivelmente por conta da grande dependência de dados que esse algoritmo possui. Isto demonstra um dos grandes desafios do paradigma de programação paralela, onde não é somente necessário se preocupar com carga de trabalho para as unidades processadoras e

custos de comunicação. A dependência de dados proveniente de um algoritmo é um fator importante na paralelização de problemas computacionais e pode gerar grandes dificuldades na conversão de um problema implementado sequencialmente para execução em paralelo.

O problema gerado pelo alinhamento de grandes sequências biológicas ainda é um grande desafio no campo da Bioinformática. Algoritmos exatos com complexidade espacial e temporal menores devem ser desenvolvidos, assim como arquiteturas mais potentes que permitam uma melhor execução desses algoritmos. Uma segunda forma de abordar esse problema é através da melhoria dos algoritmos já existentes.

Em todas as abordagens existentes para solução dos problemas de desempenho do algoritmo estudado neste trabalho, arquiteturas e paradigmas de programação paralela desempenharão um importante papel no auxílio à melhoria do problema de alinhamento global de sequências.

6.2 **Trabalhos Futuros**

Um possível trabalho futuro a partir deste estudo seria explorar outras arquiteturas paralelas para solução dos problemas impostos na paralelização do algoritmo de Needleman-Wunsch. Também poderiam ser efetuadas maiores investigações na implementação utilizando arquiteturas *manycores* NVIDIA. Com uma investigação mais aprofundada, uma possível solução viável com um bom ganho de desempenho poderia ser desenvolvida ou, em um cenário pessimista, seja demonstrado definitivamente que em arquiteturas *manycores* NVIDIA o algoritmo NW não possui uma boa forma de ser paralelizado.

Outra proposta poderia ser a de estudos mais aprofundados do problema de alinhamento global de sequências por programação dinâmica de forma que seja possível minimizar a complexidade espacial do algoritmo e sua dependência de dados, tornando-o assim, mais trivial de ser paralelizado.

REFERÊNCIAS

Watson JD, Berry A. DNA: The secret of life. Knopf AA, New York: Random House; 2003.

Mount DW. Bioinformatics: Sequence and genome analysis. 2ª ed. Tucson: Cold Spring Harbor Laboratory Press; 2004.

Sharma KR. Bioinformatics: Sequence alignment and Markov models. New York: McGraw-Hill Professional; 2009.

Zomaya AY. Parallel computing for bioinformatics and computational biology. Hoboken: Wiley-Interscience; 2006.

Kirk DB, Hwu WW. Programming Massively Parallel Processors: A Hands-on Approach. Burlington: Morgan Kaufmann; 2010.

Alberts B, Johnson A, Lewis J, Raff M, Roberts K, Walter P. Molecular biology of the cell. 5ª ed. New York: Garland Science; 2008.

Zaha A, Schrank A, Loreto EL, Bunselmeyer H, Schrank IS, Rodrigues JJS, et al. Biologia Molecular Básica. 3ª ed. Porto Alegre: Editora Mercado Aberto; 2003.

Dasgupta S, Papadimitriou C, Vazirani U. Algorithms. Berkeley: McGraw-Hill Science/Engineering/Math; 2006.

Li WH, Graur D. Fundamentals of molecular evolution. 2ª ed. Massachusetts: Sinauer Associates; 2000.

Shavit N, Herlihy M. The Art of Multiprocessor Programming. Burlington: Morgan Kaufmann; 2008.

Tanenbaum AS. Modern operating systems. 2ª ed. Upper Saddle River: Prentice Hall; 2007.

Pacheco P. An introduction to parallel programming. Burlington: Morgan Kaufmann; 2011.

Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R. Parallel programming in OpenMP. Burlington: Morgan Kaufmann; 2001.

Sanders J, Kandrot E. CUDA by example: An introduction to general-purpose GPU programming. Michigan: Addison-Wesley Professional; 2010.

Henikoff S, Greene EA, Pietrokovski S, Bork P, Attwood TK, Hood L. Gene families: The taxonomy of protein paralogs and chimeras. Science. 1997;278:609–614.

Tatusov RL, Koonin EV, Lipman DJ. A genomic perspective on protein families. Science. 1997; 278:631-7

Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. 1970;48:443-453.

Smith T, Waterman M. Identification of common molecular subsequences. J.Mol.Biomol. 1981;147:195-197.

Dayhoff MO, Schwartz RM, Orcutt BC. Atlas of protein sequence and structure. Natl. Biomed. Res. Found. 1978;5:345-358.

Henikoff S, Henikoff JG. A model for evolutionary change in proteins. Proc. Natl. Acad. Sci. 1992;89:10915-10919.

Flynn MJ. Some computer organizations and their effectiveness. IEEE Trans. Comput. 1972;21:948-960.

Rajko S, Aluru S. Space and time optimal parallel sequence alignments. IEEE Trans. Parallel Distrib. Syst. 2004;15:1070-1081.

Batista RB, Magalhaes AC. An Exact and Parallel Strategy for Local Biological Sequence Alignment in User-Restricted Memory Space. ln: 2006 IEEE INTERNATIONAL CONFERENCE ON, 2006, Barcelona. Cluster Computing. Barcelona: 2006. p. 1-10.

Hirschberg DS. A linear space algorithm for computing maximal common subsequences. Commun. ACM. 1975;18:341–343.
Fickett J. Fast optimal alignments. Nucleic Acids Res. 1984;12:175–179.

CHEN, C.; SCHMIDT, B. Computing large-scale alignments on a multi-cluster. ln: IEEE International Conference on Cluster Computing, 2003, Hong Kong. Proceedings. USA: IEEE, 2003. p. 38-45.

CHEN, Y.; YU, S.; LENG, M. Parallel sequence alignment algorithm for clustering system. ln: PROLAMAT, 2006, Shangai. Proceedings. USA: Springer US, 2006. p. 311-321.

SEGUEL, J.; TORRES C. Parallelization of Needleman-Wunsch String Alignment Method. ln: BIOCOMP, 2011, Las Vegas. Proceedings. USA: CSREA Press, 2011. p. 239-244.

NAVEED T.; SIDDIQUI S.; AHMED S. Parallel Needleman-Wunsch Algorithm for Grid. ln: PAK-US International Symposium on High Capacity Optical Networks and Enabling Technologies, 2005, Islamabad. Proceedings. Pakistan: PAK-US, 2005. p. 19-21.

ALVES, C. E. R.; CÁCERES, E. N.; DEHNE F.; SONG S. W. A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison. ln: International Conference on Computational Science and its Applications, 2003, Montreal. Lecture Notes in Computer Science. Canada: ICCSA, 2003. p. 249-258.

MARTINS, W. S; Del Cuvillo, J. B.; Useche, F. J.; Theobald, K. B.; Gao G. R. A Multithreaded Parallel Implementation Of A Dynamic Programming Algorithm For Sequence Comparison. ln: Pacific Symposium on Biocomputing, 2001, Standford. Papers. USA: SBAC-PAD, 2001. p. 1-8.

LI, J; RANKA, S.; SAHNI, S. Pairwise sequence alignment for very long sequences on GPUs. ln: IEEE 2nd International Conference on Computational Advances in Bio and Medical Sciences, 2012, Las Vegas. Proceedings. USA: IEEE, 2012. p. 1-6.

STEINFADT, U.; SCHERGER, M.; BAKER, J. W. A Local Sequence Alignment Algorithm Using an Associative Model of Parallel Computation, Proc. of IASTED Computational and Systems Biology , Dallas, p. 38-43, 2006.

Steem AJ, Dongarra JJ. Netlib.org [homepage na internet].Memphis: The University of Tennessee Health Science Center.; c1995-06 [atualizada em 2004 October 7; acesso em 2013 Abril 18]. Disponível em: http://www.netlib.org/utk/papers/advanced-computers/