Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Comparação Paralela Exata de Sequências Biológicas em Plataformas Híbridas de Alto Desempenho

Fernando Machado Mendonça

Dissertação apresentada como requisito parcial
para conclusão do Programa de Pós Graduação em Informática

Orientador
Prof. Dr. Alba Cristina Magalhães Alves de Melo

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de Pós Graduação em Informática

Coordenador: Prof. Dr. Mylene Christine Queiroz de Farias

Banca examinadora composta por:

Prof. Dr. Alba Cristina Magalhães Alves de Melo (Orientador) — CIC/UnB
Prof. Dr. Bruno Richard Schulze — LNCC
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB

CIP — Catalogação Internacional na Publicação

Mendonça, Fernando Machado.

Comparação Paralela Exata de Sequências Biológicas em Plataformas Híbridas de Alto Desempenho / Fernando Machado Mendonça. Brasília : UnB, 2013.
171 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2013.

1. Plataformas Híbridas, 2. Bioinformática, 3. GPU

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Comparação Paralela Exata de Sequências Biológicas em Plataformas Híbridas de Alto Desempenho

Fernando Machado Mendonça

Dissertação apresentada como requisito parcial
para conclusão do Programa de Pós Graduação em Informática

Prof. Dr. Alba Cristina Magalhães Alves de Melo (Orientador)
CIC/UnB

Prof. Dr. Bruno Richard Schulze
LNCC

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Mylene Christine Queiroz de Farias
Coordenador do Programa de Pós Graduação em Informática

Brasília, 31 de janeiro de 2013

Agradecimentos

Agradeço à minha família por estar sempre ao meu lado e ter me apoiado durante todos esses anos de estudo. À minha namorada, Sandra, pelo apoio e pela paciência, estando do meu lado sempre.

Agradeço também à minha orientadora, Professora Alba, por tudo que aprendi desde que nos conhecemos. Obrigado por todo os esforço e tempo aplicados e pela incrível boa vontade em ajudar. Nada disso seria possível sem você.

Obrigado aos membros da banca por se disporem a ler a minha dissertação e ajudar a melhorá-la. A sua experiência e opinião são muito bem vindas.

Obrigado aos professores do departamento, que me aceitaram de braços abertos em todas as disciplinas que me serviram de base e instrumento para que hoje eu terminasse o Mestrado.

Por último, obrigado aos amigos que fiz durante os anos, tanto dentro quanto fora da Universidade. Cada passo fica um pouco mais fácil e divertido tendo vocês por perto.

Resumo

Quando uma nova sequência biológica é descoberta, suas características funcionais e estruturais devem ser estabelecidas. Para isso, a sequência é comparada com outras sequências, procurando por similaridades. A comparação de sequências é, então, uma das operações básicas em Bioinformática. O algoritmo mais preciso para executar comparações é o proposto por Smith-Waterman (SW), que é baseado em programação dinâmica e possui complexidade quadrática de tempo e espaço. Essa complexidade pode facilmente levar a um alto tempo de execução e uso de memória. Técnicas de processamento paralelo podem ser utilizadas para produzir resultados em menos tempo. Existem muitas versões paralelas do algoritmo SW na literatura que se executam em multicores, GPUs, FPGAs e CellBEs. Mesmo que existam algumas abordagens que executem o algoritmo SW em plataformas híbridas compostas por GPUs e multicores, elas alocam trabalho de forma fixa, baseada no desempenho teórico das unidades de processamento ou nos resultados obtidos por *benchmarks*. Essa dissertação de Mestrado propõe e avalia uma estratégia otimizada e flexível para executar o algoritmo SW em plataformas híbridas compostas por GPUs e multicores com extensões SIMD. A nossa estratégia fornece múltiplas políticas de alocação de tarefas e o usuário pode escolher a que é mais apropriada para o seu problema. Propomos também um mecanismo de re-trabalho que trata situações que ocorrem quando nodos mais lentos recebem as últimas e maiores tarefas. Os resultados obtidos comparando sequências de busca com cinco diferentes bancos de dados genômicos em uma plataforma composta por 4 GPUs e 2 multicores mostram que a nossa abordagem é capaz de reduzir o tempo de execução em plataformas híbridas, quando comparada com soluções que utilizam apenas GPUs. Mostramos também que o nosso mecanismo de re-trabalho pode melhorar significativamente o desempenho na plataforma utilizada.

Palavras-chave: Plataformas Híbridas, Bioinformática, GPU

Abstract

Once a new biological sequence is discovered, its functional and structural characteristics must be established. In order to do that, the newly discovered sequence is compared against other sequences, looking for similarities. Sequence comparison is, therefore, one of the most basic operations in Bioinformatics. The most accurate algorithm to execute pairwise comparisons is the one proposed by Smith-Waterman (SW), which is based on dynamic programming, with quadratic time and space complexity. This can easily lead to very high execution times and huge memory requirements. Parallel processing can be used to produce results faster, reducing significantly the time needed to obtain results with the SW algorithm. There are many parallel versions of SW in the literature, which run in multicores, GPUs, *Field-Programmable Gate Array*s (FPGAs) and CellBEs. Even though there are some versions of SW that run on hybrid platforms composed of GPUs and multicores, they assign work in a fixed way, based on the theoretical performance of the processing units or in the results obtained by some benchmarks. This MsC Dissertation proposes and evaluates a flexible and optimized strategy to run Smith-Waterman applications in hybrid platforms composed of GPUs and multicores with SIMD extensions. Our strategy provides multiple task allocation policies and the user can choose the one which is more appropriate to his/her problem. We also propose a workload adjustment mechanism that tackles situations that arise when slow nodes receive the last tasks. The results obtained comparing query sequences to 5 public genomic databases in a platform composed of 4 GPUs and 2 multicores show that we are able to reduce the execution time with hybrid platforms, when compared to the GPU-only solution. We also show that our workload adjustment technique can provide significant performance gains in our target platform.

Keywords: Hybrid Platforms, Bioinformatics, GPU

Sumário

1 Introdução **1**
1.1 Motivação . . . 1
1.2 Objetivos . . . 2
1.3 Contribuições . . . 2
1.4 Organização do texto . . . 2

2 Plataformas de alto desempenho **4**
2.1 Extensões *Single-Instruction Multiple-Data* (SIMD) para processadores de uso geral . . . 5
2.1.1 Intel *Streaming SIMD Extensions* (SSE) e *Advanced Vector Extensions* (AVX) . . . 5
2.2 Unidades de Processamento Gráfico (*Graphics Processing Unit*s (GPUs)) . 6
2.2.1 Arquitetura SIMT . . . 7
2.2.2 *Compute Unified Device Architecture* (CUDA) . . . 8
2.2.3 Evolução das GPUs . . . 10

3 Comparação de sequências biológicas **12**
3.1 Conceitos básicos . . . 12
3.2 Bancos de dados genômicos . . . 13
3.2.1 Principais bancos . . . 13
3.2.2 Formato FASTA . . . 13
3.3 Matrizes de substituição e penalidades de gap . . . 14
3.3.1 Matrizes de substituição de aminoácidos . . . 14
3.3.2 Penalidades de GAP . . . 16
3.4 Algoritmos exatos de alinhamento . . . 16
3.4.1 AlgoritmoNeddleman-Wunsch (NW) . . . 16
3.4.2 Algoritmo Smith-Waterman (SW) . . . 17
3.4.3 Algoritmo Gotoh . . . 18
3.4.4 Paralelização dos Algoritmos Exatos . . . 19
3.5 Algoritmos heurísticos . . . 21
3.5.1 Algoritmo FASTA . . . 21
3.5.2 BLAST . . . 22

4 Alocação de tarefas em ambientes paralelos **23**
4.1 Definições gerais e terminologia . . . 23
4.2 Taxonomia de Casavant e Kuhl . . . 24
4.2.1 Classificação hierárquica . . . 24

4.2.2 Classificação linear 26
4.3 Taxonomia de Krauter *et. al* 27
4.3.1 Organização do sistema 28
4.3.2 Recursos 28
4.3.3 Escalonamento 28
4.3.4 Previsão de estado 29
4.3.5 Re-escalonamento 29
4.4 Classes de Aplicações Paralelas 30
4.4.1 Aplicações do tipo *Parameter Sweep* 30
4.4.2 Aplicações do tipo *Bag of Tasks* 30
4.4.3 Aplicações *Workflow* 30
4.5 Uso de Informações Dinâmicas 31
4.6 Alocação Mestre-Escravo 31
4.6.1 *Fixed* (F) 31
4.6.2 *Self-Scheduling* (SS) 32
4.6.3 *Guided Self-Scheduling* (GSS) 32
4.6.4 *Package Weighted Adaptive Self-Scheduling* (PSS) 33
4.6.5 *Work Stealing* 33

5 Estado da arte **34**
5.1 Abordagem de Bonny e Zidan (2011) [6] 34
5.2 Abordagem de Singh e Aruni (2011) [67] 35
5.3 Abordagem de Chen *et. al* (2010) [11] 36
5.4 Abordagem de Benkrid *et. al* (2012) [4] 37
5.5 Abordagem de Meng e Chadhary (2010) [45] 39
5.6 Abordagem de Rognes 2011 [61] 41
5.7 Abordagem de Ino *et. al* (2009) [32] 42
5.8 Abordagem de Jacek *et. al* (2012) [36] 44
5.9 Abordagem de Ino *et. al* (2012) [33] 44
5.10 Quadro comparativo 46

6 Projeto da Estratégia com Re-Trabalho para Plataformas Híbridas **48**
6.1 Considerações iniciais 48
6.2 Visão geral 49
6.3 Módulo de aquisição das sequências 50
6.4 Módulo de conversão de formato 51
6.5 Módulo de alocação de tarefas 52
6.5.1 Mecanismo de re-trabalho 52
6.6 Execução em GPU 53
6.6.1 Perfil de busca 54
6.6.2 Modificações no CUDASW++ 2.0 55
6.7 Execução em processadores Intel 55
6.7.1 *Bias* 55
6.7.2 Implementação do algoritmo Farrar 55
6.8 Adição de novas políticas de alocação de tarefas 56

7 Resultados experimentais **58**
7.1 Ambiente utilizado 58
7.2 Bancos de dados selecionados 58
7.3 Tempos de Execução e *Giga Cells Updates per Second* (GCUPS) 59
7.3.1 Execução com núcleos SSE 59
7.3.2 Execução com GPU 59
7.3.3 Execução com GPU e SSE 60
7.3.4 Avaliação do Mecanismo de re-trabalho 62
7.3.5 Execução com carga local 64

8 Conclusão **67**
8.1 Conclusão 67
8.2 Trabalhos futuros 68

Glossary **69**

Acronyms **70**

Referências **71**

Lista de Figuras

2.1 Paralelismo de dados em instruções SIMD [55]. 6
2.2 Tipos de dados suportados pelas extensões SSE e AVX [35]. 7
2.3 Área utilizada com processamento (verde) na *Central Processing Unit* (CPU) e na GPU [55]. 8
2.4 Blocos de *threads* sendo executados de forma independente por duas GPUs diferentes. [55]. 9

3.1 Diferença entre o alinhamento global e local de duas sequências 12
3.2 Exemplo de sequência no formato FASTA 14
3.3 Estratégias para paralelizar o algoritmo Smith-Waterman (SW). 21

4.1 Classificação hierárquica de Casavant e Kuhl [9]. 25

5.1 Técnica utilizada por Zidan para comparar sequências do banco de dados na GPU e na CPU simultaneamente [6]. 35
5.2 Paradigma de programação do CUDA e esquema de execução de tarefas utilizando uma fila [11]. 37
5.3 Plataforma heterogênea incluindo processadores legado, com SSE e FPGAs [45]. 39
5.4 Speedups obtidos com sequências de busca de diversos tamanhos [45]. . . . 40
5.5 Diferentes abordagens consideradas para paralelizar o algoritmo SW em núcleos SSE [61]. 41
5.6 Desempenho obtidos com BLAST, SWIPE e Striped para núcleos SSE [61]. 42
5.7 Comparação de desempenho entre o sistema implementado por Ino *et. al*, um sistema baseado em protetor de tela [32] e um *cluster* [33]. 46

6.1 Arquitetura do Sistema Proposto. 49
6.2 Organização dos processos mestre e escravos. 50
6.3 Comparação entre a alocação de tarefas com e sem o mecanismo de re-trabalho. 54

7.1 Speedup obtido na execução com SSE. 60
7.2 Speedup obtido na execução com GPU. 62
7.3 Speedup obtido na execução com SSE e GPU. 63
7.4 Ganho de desempenho ao utilizar o mecanismo de re-trabalho. 64
7.5 Execução normal com quatro núcleos SSE comparando o banco de dados Ensembl Dog. 65

7.6 Execução com quatro núcleos SSE comparando o banco de dados Ensembl Dog. O desempenho do núcleo 0 foi artificialmente reduzido para demonstrar o funcionamento da alocação de tarefas em ambiente não-dedicado. . . 66

Lista de Tabelas

2.1 Desenvolvimento da tecnologia utilizada nas GPUs da Nvidia [54, 55]. . . . 11
2.2 Desenvolvimento da tecnologia utilizada nas GPUs da ATI [3, 27]. 11

3.1 Matriz de substituição BLOSUM 62. Nessa matriz, cada bloco possui pelo menos 62% de similaridade com algum outro bloco. 15
3.2 Tabela de similaridade utilizando o algoritmo NW. 18
3.3 Tabela de similaridade utilizando o algoritmo SW. 19
3.4 Matriz de similaridade utilizando o algoritmo Gotoh. 20

5.1 Comparação entre as plataformas FPGA, GPU, CellBE e processador de uso geral [4]. 38
5.2 Desempenho das plataformas em GCUPS e os speedups obtidos em relação à implementação serial utilizando CPU [4]. 38
5.3 Equipamentos utilizados nos testes de Ino *et. al* (2009) [32]. 43
5.4 Valores médios em GCUPS para o algoritmo SW em núcleos SSE, 1 GPU e 4 GPUs [36]. 45
5.5 Quadro comparativo entre as 9 abordagens. 46

6.1 Campos que fazem parte do formato indexado. Cada linha contém quatro bytes e os campos dos nomes e sequências possuem tamanho variável. . . . 51

7.1 Informações sobre os bancos de dados selecionados para os testes. 58
7.2 Resultados obtidos na execução com SSE em tempo de execução e GCUPS. 59
7.3 Resultados obtidos na execução com GPU em tempo de execução e GCUPS. 61
7.4 Resultados obtidos na execução com SSE e GPU em tempo de execução e GCUPS. 61

Capítulo 1

Introdução

1.1 Motivação

Quando um organismo novo tem o seu código sequenciado, é importante que suas características funcionais sejam descobertas. Isso é feito a partir da comparação do seu código com o de outros organismos em busca de similaridades. A comparação de sequências biológicas é, então, uma das operações básicas em Bioinformática. Em cada comparação, um escore indica o grau de semelhança entre as sequências [46]. Além do escore, também é geralmente produzido o alinhamento, no qual uma sequência é colocada sobre a outra de modo a ressaltar as similaridades/diferenças [16]. No alinhamento, espaços podem ser inseridos para produzir melhores resultados.

O algoritmo exato mais utilizado para comparar sequências é o proposto por SW [68], baseado em técnicas de programação dinâmica, com complexidade quadrática de tempo e espaço. O algoritmo SW utiliza uma matriz de tamanho $m + 1 \times n + 1$ para comparar duas sequências de tamanho m e n. Embora ele seja bastante preciso, necessita de muitos recursos computacionais para ser executado em um tempo razoável. Para diminuir essa exigência por poder de processamento, foram criadas soluções heurísticas que, com o prejuízo de o resultado não ser ótimo, diminuem consideravelmente o tempo de execução [46].

High Performance Computing (HPC) pode ser utilizada para acelerar a execução dos algoritmos exatos de comparação de sequências. De fato, foram propostas na última década várias técnicas para executar o algoritmo SW em *cluster*s [7] e *grid*s [33]. Especificamente, aceleradores como as GPUs e FPGAs vêm sendo utilizados na execução do algoritmo SW [37, 42, 64]. Além disso, extensões SIMD dos processadores que compõem os multicores, como as extensões SSE da Intel, também têm sido utilizadas para acelerar a execução do SW [21, 22, 61, 62, 70].

Normalmente, aceleradores como GPUs e FPGAs são conectados à computadores multicore, o que permitiria que tanto os aceleradores quanto os núcleos ociosos do computador fossem utilizados na comparação das sequências. Embora existam na literatura abordagens que exploram essa idéia, elas: (a) assumem que os núcleos SSE e os aceleradores possuem poder de processamento iguais [67], (b) distribuem a carga de trabalho proporcionalmente, considerando o poder computacional teórico de cada unidade de processamento [45] ou (c) alocam uma tarefa em cada requisição [11, 33]. No nosso conhecimento, não há ainda na literatura trabalhos que executem o algoritmo SW em ambientes híbri-

dos, alocando tarefas proporcionalmente ao desempenho observado de cada unidade de processamento.

1.2 Objetivos

Essa dissertação tem como objetivo propor e avaliar uma estratégia de execução mestre-escravo do algoritmo SW em plataformas híbridas de alto desempenho que utiliza ajustes na carga de trabalho. Os ajustes são feitos de acordo com o desempenho observado para cada unidade de processamento ao longo da execução. Além disso, a estratégia é capaz de realocar tarefas na parte final da execução, a partir do momento em que pelo menos uma unidade de processamento está ociosa e há tarefas cujos resultados ainda não foram recebidos.

1.3 Contribuições

A presente dissertação de mestrado possui duas contribuições:

Mecanismo de re-trabalho Tendo em vista que as plataformas híbridas de alto desempenho possuem unidades de processamento com poder de computação muito díspares, propomos nessa dissertação o mecanismo de re-trabalho, que visa reduzir o desbalanceamento de carga causado pela atribuição das últimas tarefas às unidades de processamento muito lentas. Quando todas as tarefas restantes já estão sendo executadas e uma unidade de processamento se torna ociosa, uma das tarefas restantes é também entregue a essa unidade de processamento. A primeira unidade de processamento que terminar a execução da tarefa fornece o resultado para o mestre, que a considera concluída. Mostramos nessa dissertação que, em uma plataforma composta por 2 GPUs e 2 núcleos SSE, o mecanismo de re-trabalho é capaz de reduzir o tempo de execução em 57,2%.

Ferramenta flexível e extensível para comparação de sequências em plataformas híbridas Propomos também, nessa dissertação, uma ferramenta de comparação que possibilita a aplicação de diferentes políticas de alocação de tarefas e combinação de vários tipos de unidades de processamento. Além disso, a ferramenta pode ser estendida, adicionando-se novas políticas de alocação de tarefas e novos tipos de unidades de processamento.

1.4 Organização do texto

O restante da dissertação está organizado da seguinte forma. O Capítulo 2 trata das principais plataformas de alto desempenho utilizadas atualmente, descrevendo brevemente cada uma. O Capítulo 3 introduz os conceitos envolvendo a comparação de sequências biológicas, incluindo o algoritmo SW, o tratamento de gaps e as diferentes matrizes de substituição. O Capítulo 4 discorre sobre a alocação de tarefas em ambientes paralelos, apresenta as taxonomias mais comuns e as diferentes estratégias de alocação de tarefas. O Capítulo 5 discute o estado da arte na área de execução paralela do algoritmo SW,

apresentando, ao final, um quadro comparativo. O Capítulo 6 apresenta o projeto da estratégia proposta, com uma visão geral e detalhes sobre cada uma das técnicas utilizadas. O Capítulo 7 apresenta e discute os resultados experimentais coletados utilizando a estratégia proposta. Por último, o Capítulo 8 conclui a dissertação, comentando sobre os próximos passos a serem tomados.

Capítulo 2

Plataformas de alto desempenho

A computação de alto desempenho é uma sub-área da Ciência da Computação que teve seu início na década de 1960, quando pela primeira vez cogitou-se quebrar um problema em sub-problemas menores, com o objetivo de resolvê-los em diversos elementos de processamento simultaneamente [15].

Na década de 1970, surgiram os supercomputadores vetoriais, que eram projetados para calcular rapidamente os elementos de um vetor ou matriz. Essas arquiteturas eram exemplos da categoria SIMD de Flynn [23], na qual uma mesma instrução era executada ao mesmo tempo sobre diversos dados diferentes. Os supercomputadores vetoriais conseguiram ser uma ordem de magnitude mais rápidos do que as arquiteturas convencionais da época.

No início da década de 1980, tornou-se popular uma outra categoria de máquinas, denominadas *Symmetric multiprocessor* (SMP), que eram exemplos da categoria *Multiple-Instruction Multiple-Data* (MIMD) de Flynn, na qual códigos diferentes podiam ser executados ao mesmo tempo em processadores distintos. Essas máquinas geralmente utilizavam o paradigma de memória compartilhada para a troca de dados entre os processadores.

No final da década de 1980, ficou claro que as máquinas SMP não eram escaláveis, ou seja, não era possível conectar um grande número de processadores às mesmas, pois os meios de comunicação (barramentos) rapidamente se tornavam saturados [15].

Para solucionar esse problema, foram criados sistemas de computação distribuída, nos quais computadores relativamente completos eram conectados através de uma rede de interconexão de modo a criar um sistema integrado. Nos sistemas de computação distribuída, a troca de dados se dava através do paradigma de troca de mensagens.

Nessa categoria, inserem-se os *cluster*s de computadores, que foram definidos como sendo sistemas de computação distribuída que utilizam hardware e software disponíveis no mercado (*commodity components*) [59]. Os *cluster*s de computadores rapidamente se disseminaram por possuírem um grande apelo comercial, que combinava baixo custo e relativa facilidade de programação.

Paralelamente à evolução dos *cluster*s, arquiteturas específicas foram propostas. Essas arquiteturas são geralmente otimizadas para execução de um tipo de aplicação, sendo chamadas de aceleradores. Exemplos de aceleradores são GPUs e FPGAs.

De maneira a acelerar a execução em um único núcleo, vários fabricantes propuseram extensões SIMD, tais como SSE (arquiteturas Intel) e Altivec (arquitetura PowerPC). Ao

utilizar essas extensões, o programador consegue realizar, nas extensões mais novas, até 16 operações vetoriais ao mesmo tempo.

Atualmente, observa-se uma integração entre essas diversas categorias de arquiteturas. Por exemplo, é comum a existência de *cluster*s de multicores (SMPs) nos quais um ou mais multicores estão conectados a aceleradores (GPU, FPGAs ou outros).

A seguir, apresentamos detalhes sobre algumas dessas arquiteturas.

2.1 Extensões SIMD para processadores de uso geral

Uma arquitetura SIMD é capaz de, a partir de uma única instrução, processar múltiplos dados simultaneamente. Tais arquiteturas são particularmente úteis em aplicações de processamento digital de sinais, imagem, audio, vídeo e computação biológica. Originalmente, os supercomputadores conhecidos como processadores *array*, como o Cray-1 [63] tinham capacidade de processamento SIMD. Atualmente praticamente todos os processadores modernos implementam alguma forma de instruções SIMD. Os processadores da Intel iniciaram essa tendência implementando as instruções MMX, continuando depois com as extensões SSE, SSE2, SSE3 e recentemente o conjunto de instruções AVX. Paralelamente, processadores da arquitetura PowerPC implementaram o conjunto de instruções AltiVec para fornecer suporte à instruções SIMD.

Como pode ser visto na Figura 2.1, uma arquitetura SIMD baseia-se no paralelismo de dados, que é alcançado quando uma massa de dados de tipo conhecido é processada em paralelo por um conjunto de instruções. Um exemplo bem simples de paralelismo de dados é a inversão de uma figura para produzir o seu negativo. Nesse caso, necessita-se iterar sobre uma matriz de valores inteiros (pixels) e executar a mesma operação (subtração) em cada um.

A unidade básica utilizada em uma instrução SIMD é um vetor, motivo pelo qual as arquiteturas SIMD também são chamadas arquiteturas vetoriais. Enquanto um processador SISD (*Single Instruction Single Data*) opera sobre um escalar em cada ciclo de clock, um processador vetorial alinha um vetor de escalares e opera sobre todos em um mesmo ciclo de clock.

2.1.1 Intel SSE e AVX

A partir do Pentium II, a Intel introduziu as extensões MMX para operações SIMD. Desde então, diversas extensões foram introduzidas para ampliar o suporte à operações SIMD nos processadores, incluindo as extensões SSE, SSE2, SSE3, SSE4 e, mais recentemente, AVX e AVX2 [34]. Operações SIMD com inteiros podem ser realizadas com os registradores MMX de 64 ou 128 bits. Intruções MMX realizam operações SIMD sobre bytes, palavras e palavras duplas do tipo inteiro nos registradores XMM. Essas instruções são úteis em aplicações que operam sobre vetores do tipo inteiro.

As extensões SSE foram introduzidas com a família de processadores Pentium III e foram as primeiras a trabalharem com números ponto flutuante de precisão simples, além de suportar as instruções das extensões MMX sobre dados do tipo inteiro [34].

As extensões SSE2 foram apresentadas juntamente com a família de processadores Pentium 4 e são capazes de operar sobre valores ponto flutuante de precisão dupla contidos

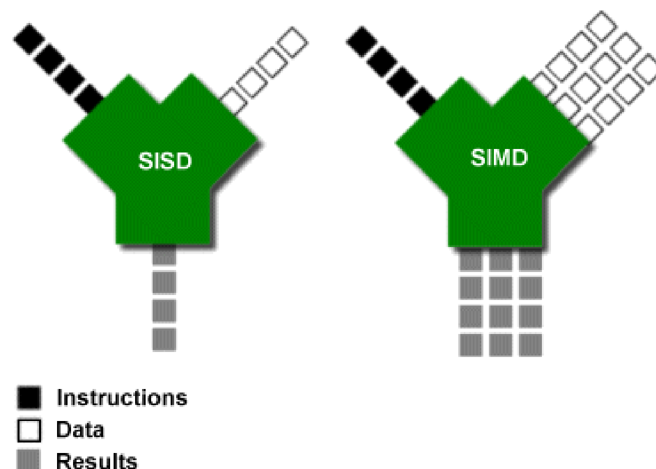


Figura 2.1: Paralelismo de dados em instruções SIMD [55].

nos registradores XMM. Foram adicionadas também as operações sobre valores do tipo inteiro utilizando os novos registradores XMM de 128 bits.

As extensões SSE3 foram introduzidas com processadores mais recentes da família Pentium 4 suportando a tecnologia *Hyper-Threading*. Foram adicionadas 13 instruções que melhoram o desempenho de instruções anteriores, das extensões SSE e SSE2.

As extensões SSE4 adicionaram 54 instruções ao conjunto já disponível aos programadores e foram introduzidas inicialmente com a família de processadores Core 2.

As extensões AVX foram introduzidas em conjunto com a família de processadores Core i7 e oferecem melhorias às gerações anteriores de extensões SIMD. Incluem suporte ao novo conjunto de registradores SIMD YMM de 256 bits e novas instruções sobre valores ponto flutuante que utilizam os novos registradores de 256 bits.

A Figura 2.2 ilustra os tipos de dados utilizados nas extensões SSE e AVX. De forma geral, qualquer múltiplo de inteiro que some 128 bits pode ser utilizado. No entanto, a geração atual não suporta operações sobre inteiros nos registradores YMM, apenas ponto flutuante. Recentemente a Intel anunciou que a próxima geração de processadores suportará a extensão AVX2, que inclui suporte a operações sobre números inteiros nos registradores de 256 bits [34].

A estratégia proposta nessa dissertação utiliza extensivamente operações sobre valores do tipo inteiro. Por esse motivo, foram utilizadas, na sua maior parte, instruções da versão 4 das extensões SSE.

2.2 Unidades de Processamento Gráfico (GPUs)

As unidades de processamento gráfico (GPUs) se tornaram uma parte importante dos sistemas de computação atuais. Os últimos anos foram marcados pelo aumento na performance e na capacidade de armazenamento das GPUs. As placas atuais são poderosas unidades de processamento gráfico para jogos e processamento de imagens mas também processadores programáveis altamente paralelizados.

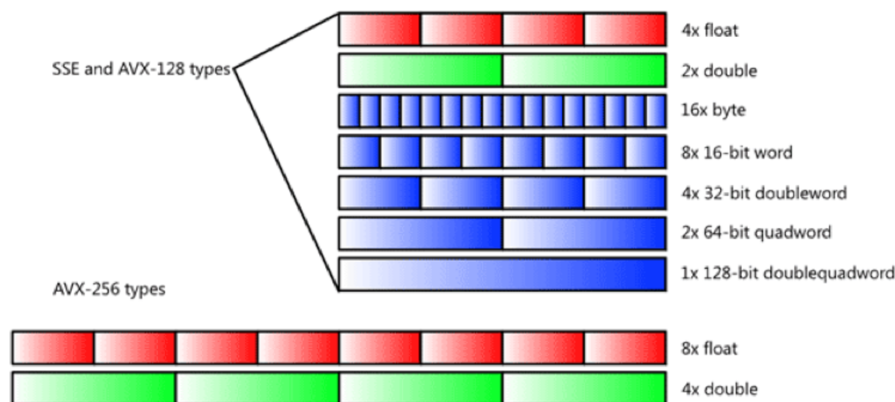


Figura 2.2: Tipos de dados suportados pelas extensões SSE e AVX [35].

As GPUs são otimizadas para uma classe de problemas com algumas características em comum [55]:

Requisitos computacionais altos. Aplicações de renderização, como por exemplo aplicações médicas de geração de exames, processam bilhões de pixels por segundo, e cada pixel requer um número grande de operações. Além disso, aplicações que manipulam matrizes extensas com alta taxa de atualização também se beneficiam do poder computacional das GPUs.

Alto paralelismo. Operações em vértices e pixels são adequados a unidades computacionais que suportam paralelismo fino, que podem ser usadas a vários outros problemas computacionais. Os problemas que melhor utilizam a arquitetura e as ferramentas disponíveis nas GPUs são aqueles que podem ser divididos em dois ou mais níveis de paralelismo. Nesse caso, o problema pode ser dividido em problemas menores, que por sua vez podem ser divididos em operações paralelas.

Alto *throughput*. Placas gráficas priorizam o *throughput* sobre a latência. Em muitos casos, acessos à memória (como a memória global) podem levar centenas de ciclos. Em contrapartida, se o número de *threads* disponíveis é alto, o nível de ocupação dos multiprocessadores pode ser mantido.

2.2.1 Arquitetura SIMT

Uma GPU é composta por vários multiprocessadores. Cada um dos multiprocessadores é responsável por criar, manter, escalonar e executar *threads* em grupos de 32 *threads* paralelas chamados *warps* [55]. Threads que fazem parte de um *warp* em particular começam a executar em um mesmo momento no mesmo endereço de programa.

Quando um multiprocessador recebe um ou mais blocos de *threads* para executar, ele divide esses blocos em *warps* que são escalonados para execução. Os *warps* são gerados sempre da mesma forma: cada *warp* é composto por *threads* consecutivas com identificação crescente, sendo que o primeiro contém a *thread* com identificação 0.

A arquitetura SIMT (*Single Instruction Multiple Threads*) é semelhante à arquitetura SIMD no que diz respeito à forma como uma única instrução controla múltiplos elementos de dados. Uma diferença chave é que um vetor SIMD expõe o seu tamanho ao programa sendo executado, enquanto instruções SIMT especificam que as *threads* executarão a mesma instrução, sobre dados diferentes. Em contraste com os processadores que suportam SIMD, a arquitetura SIMT permite ao programador utilizar tanto paralelismo a nível de *thread* quanto de dados.

O multiprocessador é responsável por executar centenas de *threads* em paralelo. Com o objetivo de maximizar o uso de suas unidades funcionais, é utilizado paralelismo a nível de *thread* além do paralelismo a nível de instruções dentro de uma mesma *thread*. Há um *pipeline* de instruções, mas diferentemente do processador principal, as instruções são executadas em ordem e não há predição de saltos ou execução especulativa.

2.2.2 CUDA

Nos últimos anos, os processadores gráficos deixaram de ser utilizados apenas nos campos da animação gráfica e modelagem e evoluíram, atingindo um enorme poder computacional, com múltiplos núcleos e memória com alta largura de banda. A maior razão para essa evolução é que as GPUs se especializaram em computação altamente paralela, exatamente do que trata renderização gráfica, mas que também pode ser utilizada para resolver um número de problemas presentes na literatura [6, 42, 64]. Por esse motivo, são utilizados na sua criação um número bem maior de transistores voltados ao processamento de dados do que cache de dados e controle de fluxo, como pode ser visto na Figura 2.3, que faz uma comparação entre a CPU e a GPU.

Mais especificamente, as GPUs são particularmente eficientes ao executar problemas com bastante paralelismo de dados, ou seja, o mesmo programa é executado sobre diferentes elementos de dados e com um grande volume de cálculos aritméticos. Pelo fato de o mesmo programa ser executado sobre um volume grande de dados, o controle de fluxo fornecido pelo hardware pode ser mais simples e a latência da memória, que normalmente é alta, pode ser escondida por uma grande quantidade de cálculos aritméticos.

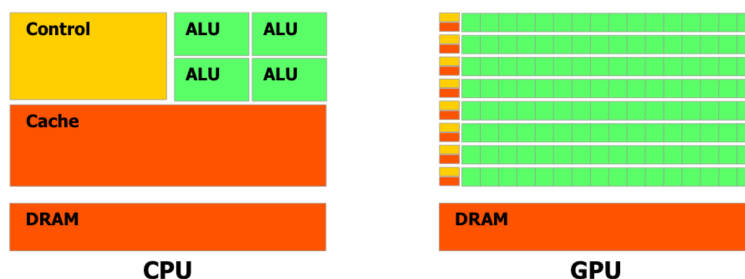


Figura 2.3: Área utilizada com processamento (verde) na CPU e na GPU [55].

CUDA é uma arquitetura de software e hardware utilizada para gerenciar processamento na GPU sem a necessidade de uma API gráfica [55]. O ambiente operacional é responsável por gerenciar o acesso à GPU por um ou mais programas CUDA e por executar as aplicações concorrentemente.

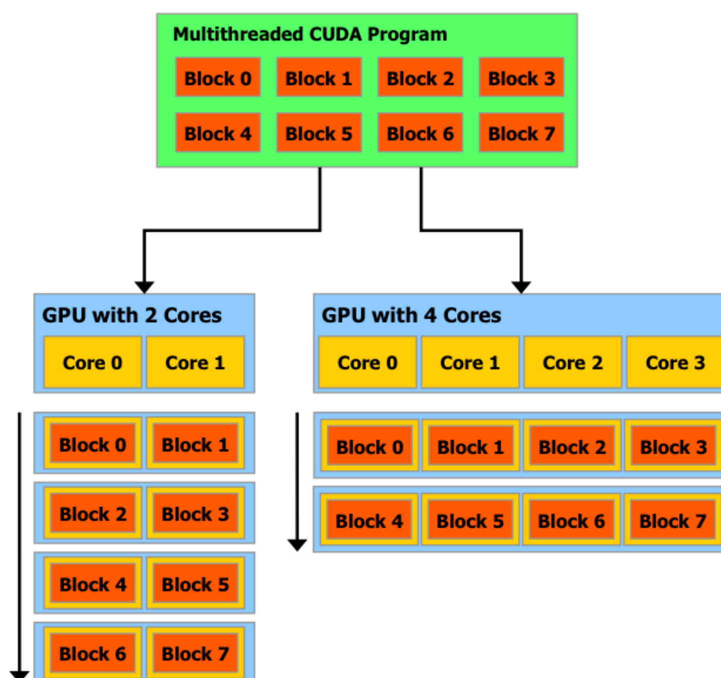


Figura 2.4: Blocos de *threads* sendo executados de forma independente por duas GPUs diferentes. [55].

Ao programar utilizando CUDA, a GPU é vista como um dispositivo capaz de executar um número grande de *threads* em paralelo, que opera como um coprocessador ao processador principal.

Mais precisamente, partes de um problema (ou todo o problema) que executam muitas vezes mas de forma independente e sobre dados diferentes podem ser isolados em uma função que é executada na GPU por muitas *threads* em paralelo. Para esse fim, a função é compilada para o conjunto de instruções conhecido pela GPU e o programa resultante, chamado *kernel*, pode ser carregado diretamente na GPU.

Tanto o processador principal quanto a GPU mantêm suas próprias áreas de memória, chamadas memória do *host* e memória do dispositivo, respectivamente. O programador pode copiar dados de uma área para a outra utilizando chamadas à API.

Um bloco de *threads* é um conjunto de *threads* que trocam dados por meio de memória compartilhada e sincronizam a sua execução para coordenar o acesso à memória. Para isso o programador pode definir pontos de sincronização dentro do *kernel*. Apenas quando todas as *threads* alcançarem o ponto de sincronização o programa continua.

Cada *thread* é identificada por um número, que é o número da *thread* dentro do bloco. Para facilitar a organização das *threads* em aplicações mais complexas, o bloco pode ser especificado como um vetor de duas ou três dimensões de tamanho arbitrário. Para um bloco de duas dimensões (D_x, D_y), a identificação da *thread* com índice (x, y) é $(x+y*D_x)$ e para o bloco de três dimensões (D_x, D_y, D_z) a *thread* com índice (x, y, z) possui o número $(x + y * D_x + z * D_x * D_y)$.

Como pode ser visto na Figura 2.4, um programa paralelo composto por várias *threads* pode ser dividido em blocos de *threads* que executam de forma independente entre si.

Desta forma, um processador gráfico com mais multiprocessadores disponíveis automaticamente executa o mesmo programa em menos tempo.

2.2.3 Evolução das GPUs

Em parte, a evolução das GPUs presenciada nos últimos anos se deve à natureza competitiva e dinâmica do segmento de criação de jogos eletrônicos. Uma das consequências desse rápido avanço tecnológico é o barateamento das peças de computador e console, especialmente das unidades de processamento gráfico (GPUs). Devido às suas arquiteturas altamente paralelas, as GPUs modernas são capazes de atingir picos de desempenho significativamente maiores que os de uma CPU. Além disso, a performance das GPUs vem aumentando um fator de 2 a 2,5 vezes por ano, bem mais rápido que o aumento presenciado nas CPUs atuais, como previsto pela lei de Moore [54].

Com o alto desempenho com baixo custo, a ampla disponibilidade de GPUs no mercado e o avanço significativo no desenvolvimento das arquiteturas CUDA e *Open Computing Language* (OpenCL) colocaram as GPUs numa posição privilegiada na computação de alto desempenho. Com esse desenvolvimento nas arquiteturas GPGPU (*General-Purpose GPUs*), a utilidade de uma placa gráfica foi expandida além do processamento gráfico utilizado nos jogos eletrônicos e programas de modelagem 3D, alcançando uma variedade de aplicação paralelas e tornando-se uma área de pesquisa intensa em computação de alto desempenho [14, 33, 36, 42, 66].

GPUs Nvidia

Embora a idéia de utilizar-se processadores gráficos para computação de propósito geral já existisse há algum tempo [26], principalmente na área médica, essa pesquisa estava limitada à programação da GPU utilizando *pipeline* gráfico. Isso mudou com a plataforma CUDA, da Nvidia, que permitiu que novas aplicações intensamente paralelas utilizassem programação em C e C++ para aproveitar o desempenho das GPUs, com milhares de núcleos e *threads* trabalhando em paralelo [55]. A plataforma CUDA é suportada na grande maioria das placas vendidas nos últimos cinco anos, inclusive as GPUs orientadas à computação de alto desempenho, chamadas Tesla. Com isso, mais de 50 milhões de placas com suporte à plataforma CUDA foram vendidas nos últimos anos [44].

A Tabela 2.1 mostra a evolução das GPUs construídas pela Nvidia desde 1997. A primeira GPU com suporte à CUDA foi colocada no mercado em 2006, com a GeForce 8800, com 681 milhões de transistores. Atualmente, são colocados nas placas 3,5 bilhões de transistores e 1536 núcleos CUDA, um crescimento de 413% e 1100%, respectivamente.

GPUs ATI/AMD

A Tabela 2.2 mostra a evolução das GPUs da ATI/AMD entre 1999 e 2013. Como pode ser visto, a primeira GPU com suporte à plataforma OpenCL foi colocada no mercado em 2008, quando a arquitetura foi certificada pela Apple e pelo Khronos Group [28]. Desde então, novas placas receberam suporte às versões atualizadas da arquitetura, OpenCL 1.1 em 2009 e 1.2, a partir de 2012. Assim como no caso das placas da Nvidia, as placas da ATI/AMD passaram por extensos avanços na quantidade de transistores e núcleos OpenCL, indo de 700 milhões em 2007 para 5,1 bilhões em 2013, um aumento de 628%.

Ano	Modelo	Transistores	Núcleos CUDA	Tecnologia
1997	Riva 128	3 milhões	-	Acelerador gráfico 3D
1999	GeForce 256	25 milhões	-	Primeira GPU, com suporte ao DX7 e OpenGL
2011	GeForce 3	60 milhões	-	Primeira GPU com *shader* programável. Suportava DX8 e OpenGL
2002	GeForce FX	125 milhões	-	GPU programável de 32 bits, com DX8 e OpenGL
2004	GeForce 6800	222 milhões	-	GPU programável escalável, programas GPGPU Cg, DX9 e OpenGL
2006	GeForce 8800	681 milhões	128	Primeira GPU unificada, programável em C com CUDA
2007	Tesla T8, C870	681 milhões	128	Primeiro sistema de computação programado em C com CUDA
2008	GeForce GTX 280	1,4 bilhões	240	GPU unificada, IEEE FP, CUDA C, OpenCL e DirectCompute
2008	Tesla T10, S1070	1,4 bilhões	240	*Cluster*s com GPU, IEEE FP de 64 bits, 4GB de memória, CUDA C e OpenCL
2009	GeForce GTX 480	3 bilhões	512	Arquitetura GPGPU, IEEE 754-2008 FP, endereçamento 64 bits, cache, memória ECC, CUDA C, C++, OpenCL e DirectCompute
2012	GeForce GTX 680	3,5 bilhões	1536	CUDA 5.0

Tabela 2.1: Desenvolvimento da tecnologia utilizada nas GPUs da Nvidia [54, 55].

O número de núcleos OpenCL aumentou de 800 na primeira revisão da arquitetura para 2560, nas GPUs lançadas recentemente, um aumento de 220%.

Ano	Modelo	Transistores	Núcleos	Tecnologia
1999	Rage Fury MAXX	8 milhões	Não disp.	Pipeline de pixels. OpenGL 1.0
2001	Radeon 7500	30 milhões	Não disp.	DirectX 7, OpenGL 1.4
2004	Radeon 9250	36 milhões	Não disp.	DirectX 8, OpenGL 1.4
2006	Radeon X1050	75 milhões	Não disp.	DirectX 9, OpenGL 2
2007	Radeon HD 2900 XT	700 milhões	320	DirectX 10, OpenGL 3.3
2008	Radeon HD 3870	666 milhões	320	DirectX 10.1 e OpenGL 3.3
2008	Radeon HD 4890	959 milhões	800	DirectX 10.1, OpenGL 3.3 e OpenCL 1.0
2009	Radeon HD 5870	2,1 bilhões	1440	DirectX 11, OpenGL 4.1 e OpenCL 1.1
2011	Radeon HD 6970	2,6 bilhões	1536	DirectX 11, OpenGL 4.1 e OpenCL 1.2
2012	Radeon HD 7970	4,3 bilhões	2048	DirectX 11.1, OpenGL 4.2 e OpenCL 1.2
2013	Radeon HD 8970	5,1 bilhões	2560	DirectX 11.1, OpenGL 4.2 e OpenCL 1.2

Tabela 2.2: Desenvolvimento da tecnologia utilizada nas GPUs da ATI [3, 27].

Capítulo 3

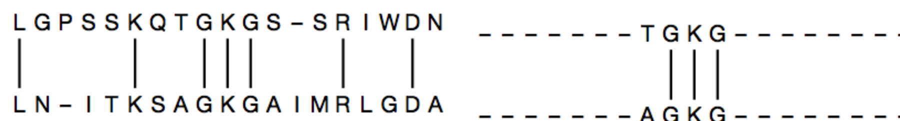
Comparação de sequências biológicas

3.1 Conceitos básicos

As sequências biológicas são cadeias ordenadas de resíduos (DNA, RNA ou proteínas) [46]. Sequências de DNA e RNA são compostas por nucleotídeos, representados pelo alfabeto $\Sigma = \{A, T, G, C\}$ e $\Sigma = \{A, T, G, U\}$, respectivamente. As proteínas são sequências de aminoácidos do alfabeto $\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$.

A comparação de sequências biológicas visa determinar o grau de similaridade entre as sequências biológicas e tornar explícitas as diferenças/semelhanças entre as mesmas. Como resultado de uma operação de comparação de sequências temos um escore, que é a medida de similaridade entre as sequências, e o alinhamento [46].

Há dois tipos básicos de alinhamento, global e local. No alinhamento global, as duas sequências são alinhadas por completo, tentando-se evidenciar o maior número possível de resíduos idênticos. Sequências de tamanho e estrutura semelhantes são boas candidatas para o alinhamento global. No alinhamento local, pedaços das sequências com o maior número possível de resíduos idênticos ou semelhantes são alinhados. Alinhamentos locais são adequados no alinhamento de sequências que são semelhantes em algumas partes mas diferentes nas demais ou sequências que possuem partes semelhantes mas tamanhos diferentes [16].



(a) Alinhamento global (b) Alinhamento local

Figura 3.1: Diferença entre o alinhamento global e local de duas sequências

Para as duas sequências mostradas na Figura 3.1, o alinhamento global contém as duas sequências de forma integral e o alinhamento local só inclui quatro resíduos. Barras verticais entre as sequências indicam a presença de resíduos idênticos e barras horizontais indicam lacunas (gaps) em uma das sequências.

No alinhamento local, o alinhamento termina no final das regiões em que os resíduos são idênticos ou semelhantes, e a prioridade do algoritmo é encontrar estas regiões e não

estendê-las para incluir outras regiões de semelhança. Este tipo de alinhamento favorece a busca por padrões nos resíduos.

3.2 Bancos de dados genômicos

3.2.1 Principais bancos

Ensembl

O projeto Ensembl [18] foi iniciado em 1999, alguns anos antes de o genoma humano ser completamente sequenciado. Seu objetivo principal é oferecer de forma pública e fácil um banco de dados de sequências com suporte à anotações automáticas, realizadas com comparações de padrões entre as proteínas conhecidas e o DNA. O Ensembl é um projeto conjunto entre o instituto Europeu de Bioinformática (EBI), o Laboratório Europeu de Biologia Molecular (EMBL) e o Instituto *Sanger Trust Wellcome* (WTSI).

NCBI RefSeq

A coleção de sequências de referência RefSeq [51] tem como principal objetivo oferecer um conjunto de sequências de forma compreensiva, integrada e bem anotada, incluindo DNA genômico e proteínas. Seu escopo inclui diversos organismos, como células eucariontes, bactérias e vírus. Atualmente, a coleção RefSeq inclui 17 milhões de proteínas compreendendo 18 mil organismos.

UniProt

A missão da organização UniProt [17] é fornecer à comunidade científica uma fonte de sequências protéicas de alta qualidade, fácil acesso e de foram compreensiva. O banco de dados Swiss-Prot é anotado de forma manual e revisado. Já o banco de dados TrEMBL é anotado automaticamente e não é revisado. Além disso, é disponibilizada a coleção de referência UniRef, utilizada normalmente para acelerar buscas por similaridades em sequências. A última versão do banco de dados UniProtKB/Swiss-Prot, disponível no dia 9 de janeiro de 2013, contém 538 mil sequências formadas a partir de 191 milhões de aminoácidos.

3.2.2 Formato FASTA

O formato de sequência FASTA é um dos formatos mais usados em bancos de dados genômicos. Inclui três partes principais, como pode ser visto na Figura 3.2: uma linha de comentário com o nome e a origem da sequência, a sequência em caracteres e um caractere opcional que indica o fim da sequência e que pode ou não estar presente.

O formato FASTA é um dos mais usados pelos programas de alinhamento. Ele provê uma forma conveniente de transferir apenas a parte da sequência para outro arquivo porque essa parte não mantém outros caracteres que não fazem parte da sequência.

```
>YCZ2_YEAST protein in HMR 3' region
MKAVVIEDGKAVVKEGVPIPELEEGFV
GNPTDWAHIDYKVGPQGSILGCDAAGQ
IVKLGPAVDPKDFSIGDYIYGFIHGSS
VRFPSNGAFAEYSAISTVVAYKSPNEL
KFLGEDVLPAGPVRSLEGAATIPVSLT*
```

Figura 3.2: Exemplo de sequência no formato FASTA

3.3 Matrizes de substituição e penalidades de gap

3.3.1 Matrizes de substituição de aminoácidos

Os biólogos descobriram que certas substituições de aminoácidos ocorrem comumente em proteínas relacionadas em diferentes espécies[31]. Os aminoácidos substituídos assim são compatíveis com a estrutura e as funções das proteínas porque estas funcionam mesmo com as substituições. Frequentemente, as substituições ocorrem entre aminoácidos quimicamente semelhantes, embora outras substituições também possam ocorrer, menos frequentemente. Saber os tipos de modificações que são mais ou menos comuns em um número grande de proteínas pode auxiliar no alinhamento de famílias de proteínas. Se proteínas relacionadas são semelhantes, elas podem ser facilmente alinhadas. Se relações ancestrais sobre um grupo de proteínas são analisadas, as modificações em aminoácidos que ocorrem com mais frequência durante o processo evolutivo podem ser previstas. Esse tipo de análise foi feita de forma pioneira por Dayhoff e Schwartz [13].

Matrizes de substituição de aminoácidos são utilizadas para representar as pontuações das substituições. Os aminoácidos são listados tanto nas linhas quanto nas colunas, e cada posição da matriz é preenchida com um escore que reflete a frequência com a qual dois aminoácidos são encontrados juntos em sequências de proteínas relacionadas. A probabilidade de um aminoácido ser substituído por outro é sempre considerada simétrica. Essa hipótese é feita porque, para quaisquer duas sequências, o aminoácido ancestral na árvore filogenética é frequentemente desconhecido. Adicionalmente, a probabilidade de substituição depende do produto da frequência de ocorrência dos dois aminoácidos além de suas similaridades químicas e estruturais. A predição deste modelo é que a frequência de um aminoácido não se modifica durante o processo evolutivo [13].

As famílias de substituição de aminoácidos Dayhoff (*Percent Accepted Mutation* ou PAM) fornecem a probabilidade de mudança de um aminoácido para outro em sequências de proteínas homólogas durante a evolução. Cada matriz fornece as modificações esperadas para um dado período do processo evolutivo, evidenciado por uma diminuição na similaridade entre as sequências conforme a codificação dos genes nas mesmas proteínas diverge com o passar o tempo.

No processo de derivação das tabelas PAM, assume-se que cada modificação no aminoácido de uma determinada área da proteína é independente dos eventos de mutação ocorridos anteriormente naquela mesma área [13]. Logo, a probabilidade de substituição de um aminoácido A por um aminoácido B é a mesma, independentemente das modificações anteriores naquela área da proteína e da posição do aminoácido A na proteína. Substituições de aminoácidos em uma sequência de proteína são então vistas como um

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	9	-1	-1	-3	0	-3	-3	-3	-4	-3	-3	-3	-3	-1	-1	-1	-1	-2	-2	-2
S	-1	4	1	-1	1	0	1	0	0	0	-1	-1	0	-1	-2	-2	-2	-2	-2	-3
T	-1	1	4	1	-1	1	0	1	0	0	0	-1	0	-1	-2	-2	-2	-2	-2	-3
P	-3	-1	1	7	-1	-2	-1	-1	-1	-1	-2	-2	-1	-2	-3	-3	-2	-4	-3	-4
A	0	1	-1	-1	4	0	-1	-2	-1	-1	-2	-1	-1	-1	-1	-1	-2	-2	-2	-3
G	-3	0	1	-2	0	6	-2	-1	-2	-2	-2	-2	-2	-3	-4	-4	0	-3	-3	-2
N	-3	1	0	-2	-2	0	6	1	0	0	-1	0	0	-2	-3	-3	-3	-3	-2	-4
D	-3	0	1	-1	-2	-1	1	6	2	0	-1	-2	-1	-3	-3	-4	-3	-3	-3	-4
E	-4	0	0	-1	-1	-2	0	2	5	2	0	0	1	-2	-3	-3	-3	-3	-2	-3
Q	-3	0	0	-1	-1	-2	0	0	2	5	0	1	1	0	-3	-2	-2	-3	-1	-2
H	-3	-1	0	-2	-2	-2	1	1	0	0	8	0	-1	-2	-3	-3	-2	-1	2	-2
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5	2	-1	-3	-2	-3	-3	-2	-3
K	-3	0	0	-1	-1	-2	0	-1	1	1	-1	2	5	-1	-3	-2	-3	-3	-2	-3
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5	1	2	-2	0	-1	-1
I	-1	-2	-2	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4	2	1	0	-1	-3
L	-1	-2	-2	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4	3	0	-1	-2
V	-1	-2	-2	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4	-1	-1	-3
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6	3	1
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7	2
W	-2	-3	-3	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11

Tabela 3.1: Matriz de substituição BLOSUM 62. Nessa matriz, cada bloco possui pelo menos 62% de similaridade com algum outro bloco.

modelo de Markov, caracterizado pelo conjunto de mudanças de estado em um sistema tal que a modificação de um estado não depende do histórico do estado.

Há várias matrizes PAM, cada uma com um sufixo numérico diferente. A matriz PAM1 foi construída com um conjunto de proteínas 85% (ou mais) idênticas entre si. As demais matrizes da família PAM foram construídas multiplicando a matriz PAM1 por ela mesma: 100 vezes para PAM100, 160 vezes para PAM160 e assim sucessivamente.

A premissa que suporta o método de construção da matriz de escore Dayhoff foi bastante questionada [24]. Primeiro, porque assume-se que cada posição contendo um aminoácido possui igual probabilidade de mutação, quando na verdade partes da sequências variam consideravelmente quanto à probabilidade de mutação. Além disso, as matrizes PAM são criticadas por serem baseadas em um conjunto pequeno de proteínas relacionadas.

As matrizes de substituição de blocos de aminoácidos (*Blocks Amino Acid Substitution Matrix* ou BLOSUM) são baseadas nos escores de substituição encontrados em vários períodos do processo evolutivo e revelam substituições que não são evidenciadas pelo modelo PAM. Os valores são baseados em substituições de aminoácidos observadas em um conjunto de aproximadamente 2000 padrões de aminoácidos, chamados blocos. Estes blocos são encontrados em bancos de dados de sequências de proteínas representando mais de 500 famílias de proteínas relacionadas [31]. As matrizes BLOSUM são então baseadas em um tipo diferente de análise de sequências e um conjunto maior de sequências do que as matrizes PAM. A Tabela 3.1 mostra a matriz BLOSUM62.

3.3.2 Penalidades de GAP

A inclusão de gaps e penalidades de gap no processo de comparação é necessária para melhorar a qualidade do alinhamento [16]. As penalidades utilizadas podem ser calculadas a partir de um custo linear (Equação 3.1):

$$w_x = gd, \tag{3.1}$$

em que g é o custo de cada gap e d é a quantidade de gaps inseridos ou *affine-gap*, no qual existe um custo inicial g e uma penalidade para estender o gap r:

$$w_x = g + rx, \tag{3.2}$$

em que x é a extensão do gap (Equação 3.2).

Se a penalidade utilizada para o gap for muito alta em relação aos valores de escore presentes na matriz de substituição, gaps nunca aparecerão no alinhamento. De forma análoga, se a penalidade for muito pequena em relação aos valores presentes na matriz de substituição, gaps serão extensivamente utilizados para alinhar o maior número possível de resíduos. Ao decidir os valores para as penalidades de gap utilizados em um alinhamento local, é importante considerar que as penalidades devem ser grandes o suficiente para permitir o alinhamento local das sequências. Valores pequenos podem causar a extensão do alinhamento local ótimo a partir de gaps.

3.4 Algoritmos exatos de alinhamento

Dadas duas sequências a serem comparadas, os algoritmos de alinhamento visam determinar um ou mais alinhamentos e seus respectivos escores. Idealmente, os algoritmos buscam o alinhamento ótimo, ou seja, o alinhamento que possui o maior escore.

Alinhamentos melhores possuirão escores maiores, então o que se quer é maximizar o escore para encontrar o alinhamento ótimo [46]. Em alguns casos, escores recebem outro significado e são interpretados como distâncias ou custos e, nesse caso, o que se quer é minimizar o custo do alinhamento. Algoritmos de programação dinâmica se aplicam aos dois casos com poucas alterações.

3.4.1 AlgoritmoNeddleman-Wunsch (NW)

O algoritmo de programação dinâmica, utilizado para encontrar o alinhamento global ótimo entre duas sequências, permitindo-se o uso de gaps, é o algoritmo Needleman-Wunsch (NW) [52]. A idéia é construir um alinhamento ótimo utilizando soluções parciais anteriores para o alinhamento ótimo de subsequências. O algoritmo NW possui duas fases: Criação da Matriz de Similaridade e Obtenção do Alinhamento.

Criação da matriz de similaridade Esta fase recebe como entrada as sequências s e t, com $|s| = m$ e $|t| = n$, em que m e n são os tamanhos das sequências s e t, respectivamente. A notação utilizada para representar o n-ésimo resíduo de uma sequência u é $u[n]$ e, para representar um prefixo com n caracteres, usa-se $u[1 \ldots n]$. A matriz de similaridade é

denotada por $D_{m+1 \times n+1}$, em que $D_{i,j}$ contem o escore de similaridade entre os prefixos $s[1 \dots i]$ e $t[1 \dots j]$.

No início do processamento, a primeira linha e a primeira coluna são preenchidas com a penalidade de gap gi, sendo que i é o tamanho da sequência não vazia terminando no i-ésimo elemento e g é a penalidade de gap. Os elementos restantes são obtidos através da Equação 3.3. Nessa equação, $p(i, j)$ é igual ao escore de um *match* se $s[i] = t[j]$ ou de um *mismatch*, caso contrário. Em ambos os casos g é a penalidade de gap. Para a comparação de aminoácidos $p(i, j)$ é obtido através da matriz de substituição (Seção 3.3.1). Cada célula da matriz $D_{i,j}$ armazena também a célula que foi utilizada para gerar o valor $D_{i,j}$ ($D_{i,j-1}$, $D_{i-1,j}$ ou $D_{i-1,j-1}$). O valor de similaridade entre as sequências s e t é o valor presente na posicão $D[m, n]$. Sua posição marca o final do alinhamento global ótimo.

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + p(i,j) \\ D_{i,j-1} + g \\ D_{i-1,j} + g \end{cases} \tag{3.3}$$

Nesta equação, $D_{i,j}$ contém o escore ótimo entre os elementos os prefixos $s[0 \dots i]$ e $t[0 \dots j]$, ou seja, $D_{m,n}$ contém o resultado do alinhamento global terminando nos elementos s_m e t_n.

Obtenção do alinhamento O caminho com melhor alinhamento pode ser obtido partindo do canto inferior direito da matriz (resíduo $D_{m,n}$) e procedendo em direção à origem segundo a informação de *backtrack* armazenada na fase anterior. A Tabela 3.2 apresenta a matriz $D_{i,j}$ calculada com o algoritmo Needleman-Wunsh para comparação de duas sequências de proteínas, utilizando a matriz de substituição BLOSUM62.

3.4.2 Algoritmo Smith-Waterman (SW)

Para encontrar a similaridade entre partes das sequências, deve ser encontrado o alinhamento ótimo local. O algoritmo exato normalmente utilizado nesse caso é o proposto por Smith e Waterman (SW) [68]. Assim como o algoritmo NW, é baseado em programação dinâmica e possui tempo e espaço quadrático. No entanto, há três diferenças básicas entre os dois.

A primeira diferença está na inicialização da primeira coluna e da primeira linha, que são preenchidas com zeros no algoritmo SW. Desde modo, os gaps não recebem penalidade se estiverem no começo do alinhamento. A segunda diferença envolve a equação utilizada no cálculo das células remanescentes. No algoritmo SW, valores negativos não são considerados. Para que isto ocorra, o valor 0 é incluído na Equação 3.3, gerando a Equação 3.4.

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + p(i,j) \\ D_{i,j-1} + g \\ D_{i-1,j} + g \\ 0 \end{cases} \tag{3.4}$$

	L	N	I	T	K	S	A	G	K	G	A	I	M	R	L	G	D	A
L	**4**	1	2	-1	-2	-2	-1	-4	-2	-4	-1	2	2	-1	4	1	-2	-1
G	1	**4**	1	0	-3	-2	-2	5	2	4	1	-1	-1	0	1	10	7	4
P	-2	**1**	1	0	-1	-4	-3	2	4	1	3	0	-3	-3	-2	7	9	6
S	-2	-1	**-1**	2	0	3	0	-1	2	4	2	1	-1	-4	-5	4	7	10
S	-2	-1	-3	**0**	2	4	4	1	-1	2	5	2	0	-2	-5	1	4	8
K	-2	-2	-4	-3	**5**	2	3	2	6	3	2	2	1	2	-1	-2	1	5
Q	-2	-2	-5	-5	2	**5**	2	1	3	4	2	-1	2	2	0	-3	-2	2
T	-1	-2	-3	0	-1	3	**5**	2	0	1	4	1	-1	1	1	-2	-4	-1
G	-4	-1	-4	-3	-2	0	3	**11**	8	6	3	0	-2	-2	-2	7	4	1
K	-2	-4	-4	-5	2	-1	0	8	**16**	13	10	7	4	1	-2	4	6	3
G	-4	-2	-5	-6	-1	2	-1	6	13	**22**	19	16	13	10	7	4	3	6
S	-2	-3	-4	-4	-4	3	3	3	10	19	**23**	**20**	17	14	11	8	5	4
S	-2	-1	-4	-3	-4	0	4	3	7	16	20	21	**19**	16	13	11	8	6
R	-2	-2	-4	-5	-1	-3	1	2	5	13	17	18	20	**24**	21	18	15	12
I	2	-1	2	-1	-4	-3	-2	-1	2	10	14	21	19	21	**26**	23	20	17
W	-1	-2	-1	0	-3	-6	-5	-4	-1	7	11	18	20	18	23	**24**	21	18
D	-4	0	-3	-2	-1	-3	-6	-6	-4	4	8	15	17	18	20	22	**30**	27
N	-3	2	-1	-3	-2	0	-3	-6	-6	1	5	12	14	17	17	20	27	**28**

Tabela 3.2: Tabela de similaridade utilizando o algoritmo NW.

A terceira diferença está na célula utilizada para começar o processo de *traceback*. Para obter o melhor alinhamento local, o algoritmo SW começa da célula com o maior escore, seguindo as posições armazenadas até que o valor zero seja encontrado.

A Tabela 3.3 apresenta a matriz Di, j calculada com o algoritmo Smith-Waterman para a comparação das mesmas duas sequências de proteínas da Tabela 3.2, com a matriz de substituição BLOSUM62.

3.4.3 Algoritmo Gotoh

O algoritmo Gotoh [25] utiliza programação dinâmica para o alinhamento global exato de pares de sequências (Seção 3.4.1) com o modelo *affine-gap*. Nesse modelo, a penalidade para se iniciar uma sequência de gaps é maior do que a penalidade para estendê-la. Para o autor, esse modelo de gaps de tamanho variável é interessante porque muitas vezes um gap longo pode ser produzido por um único evento de mutação.

Sejam $s = s_1, s_2, \ldots, s_m$ e $t = t_1, t_2, \ldots, t_n$ duas sequências. Um gap de tamanho k é da forma $w_k = uk + v$, com $u \leq 0$ e $v \leq 0$. O peso $p(i, j)$ é dado ao alinhamento do par s_i e t_j e normalmente $p(i, j) \geq 0$ se $s_i = t_j$ e $p(i, j) < 0$ se $s_i \neq t_j$.

Além da matriz D, necessita-se de duas outras matrizes (P e Q) para tratar sequências de gaps em cada uma das sequências. Sendo assim, as novas equações de recorrência são dadas pelas Equações 3.5, 3.6, 3.7.

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + p(i, j) \\ P_{i,j} \\ Q_{i,j} \end{cases} \tag{3.5}$$

	L	N	I	T	K	S	A	G	K	G	A	I	M	R	L	G	D	A
L	4	1	2	0	0	0	0	0	0	0	0	2	2	0	4	1	0	0
G	1	4	1	0	0	0	0	6	3	6	3	0	0	0	1	10	7	4
P	0	1	1	0	0	0	0	3	5	3	5	2	0	0	0	7	9	6
S	0	1	0	2	0	4	1	0	3	5	4	3	1	0	0	4	7	10
S	0	1	0	**1**	2	4	5	2	0	3	6	3	2	0	0	1	4	8
K	0	0	0	0	**6**	3	3	3	7	4	3	3	2	4	1	0	1	5
Q	0	0	0	0	3	**6**	3	1	4	5	3	0	3	3	2	0	0	2
T	0	0	0	5	2	4	**6**	3	1	2	5	2	0	2	2	0	0	0
G	0	0	0	2	3	2	4	**12**	9	7	4	1	0	0	0	8	5	2
K	0	0	0	0	7	4	1	9	**17**	14	11	8	5	2	0	5	7	4
G	0	0	0	0	4	7	4	7	14	**23**	20	17	14	11	8	6	4	7
S	0	1	0	1	1	8	8	5	11	20	**24**	**21**	18	15	12	9	6	5
S	0	1	0	1	1	5	9	8	8	17	21	22	**20**	17	14	12	9	7
R	0	0	0	0	3	2	6	7	10	14	18	19	21	**25**	22	19	16	13
I	2	0	4	1	0	1	3	4	7	11	15	22	20	22	**27**	24	21	18
W	0	0	1	2	0	0	0	1	4	8	12	19	21	19	24	**25**	22	19
D	0	1	0	0	1	0	0	0	1	5	9	16	18	19	21	23	**31**	28
N	0	6	3	0	0	2	0	0	0	2	6	13	15	18	18	21	28	29

Tabela 3.3: Tabela de similaridade utilizando o algoritmo SW.

$$P_{i,j} = \max \begin{cases} D_{i-1,j} + w_1 \\ P_{i-1,j} + u \end{cases} \tag{3.6}$$

$$Q_{i,j} = \max \begin{cases} D_{i,j-1} + w_1 \\ Q_{i,j-1} + u \end{cases} \tag{3.7}$$

A Tabela 3.4 apresenta a matriz D calculada com o algoritmo Gotoh.

3.4.4 Paralelização dos Algoritmos Exatos

Existem diversas maneiras de se paralelizar os algoritmos apresentados nas Seções 3.4.1, 3.4.2 e 3.4.3. Nos parágrafos a seguir, assumimos que um conjunto de x sequências de busca $(q_1, q_2, \ldots, q_x)$ será comparado com um conjunto de y sequências do banco de dados $(d_1, d_2, \ldots, d_y)$ e que $x << y$.

Na abordagem de granularidade fina, a comparação entre uma sequência de busca e uma sequência de banco de dados é paralelizada. O padrão de dependência de dados nesse cálculo é não-uniforme e os cálculos que podem ser feitos em paralelo correspondem às anti-diagonais da matriz (Equações 3.3 a 3.7). Esse padrão de paralelização é conhecido como wavefront [59].

A Figura 3.3(a) ilustra uma comparação em granularidade fina entre uma sequência de busca e uma sequência de banco de dados para quatro unidades de processamento. No início do cálculo, somente $P0$ calcula. Quando $P0$ termina de calcular o primeiro bloco, ele envia a coluna de borda para $P1$, que pode iniciar a computação. Nesse momento,

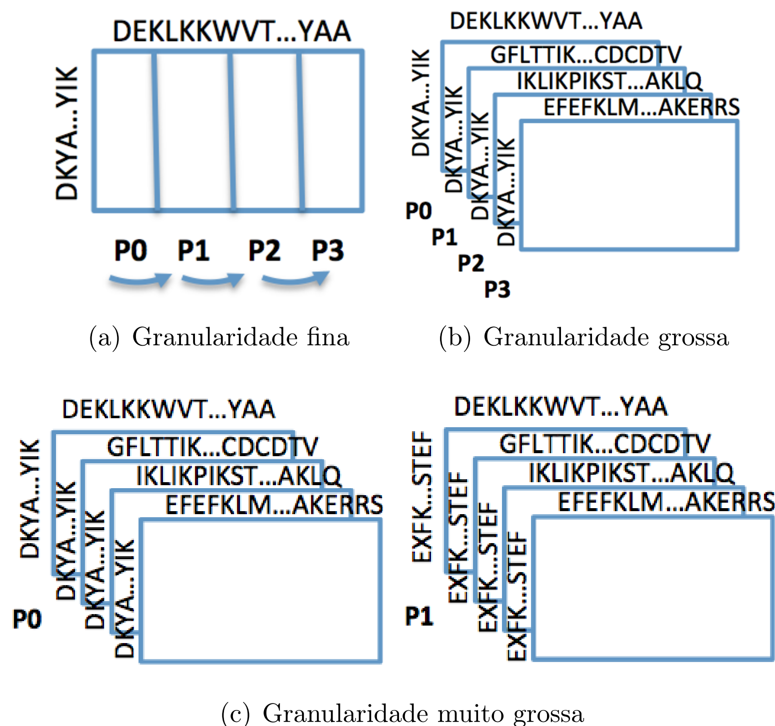
	L	N	I	T	K	S	A	G	K	G	A	I	M	R	L	G	D	A
L	4	0	2	0	0	0	0	0	0	0	0	2	2	0	4	0	0	0
G	0	4	0	0	0	0	0	6	1	6	1	0	0	0	0	10	5	3
P	0	0	1	0	0	0	0	1	5	1	5	0	0	0	0	5	9	4
S	0	1	0	2	0	4	1	0	1	5	2	3	0	0	0	3	5	10
S	0	1	0	**1**	2	4	5	1	0	1	6	1	2	0	0	1	3	6
K	0	0	0	0	**6**	2	3	3	6	1	1	3	0	4	0	0	0	3
Q	0	0	0	0	1	**6**	1	1	4	4	0	0	3	1	2	0	0	1
T	0	0	0	5	0	2	**6**	1	0	2	4	0	0	2	0	0	0	0
G	0	0	0	0	3	0	2	**12**	7	6	3	1	0	0	0	6	1	0
K	0	0	0	0	5	3	0	7	**17**	12	10	8	6	4	2	1	5	0
G	0	0	0	0	0	5	3	6	12	**23**	18	16	14	12	10	8	6	5
S	0	1	0	1	0	4	6	3	10	18	**24**	**19**	17	15	13	11	9	7
S	0	1	0	1	1	4	5	6	8	16	19	22	**18**	16	13	13	11	10
R	0	0	0	0	3	0	3	3	8	14	17	17	21	**23**	18	16	14	12
I	2	0	4	0	0	1	0	0	4	12	15	21	18	18	**25**	20	18	16
W	0	0	0	2	0	0	0	0	2	10	13	16	20	16	20	**23**	18	16
D	0	1	0	0	1	0	0	0	0	8	11	14	15	18	18	19	**29**	24
N	0	6	1	0	0	2	0	0	0	6	9	12	13	15	16	18	24	27

Tabela 3.4: Matriz de similaridade utilizando o algoritmo Gotoh.

$P0$ e $P1$ estão calculando simultaneamente. Quando $P1$ termina o cálculo de um bloco, ele envia a coluna de borda para $P2$, que pode iniciar sua computação. $P1$ continua a computação quando receber a segunda coluna de borda de $P0$. Nesse momento, $P0$, $P1$ e $P2$ estão calculando simultaneamente. Note que, próximo ao final do cálculo da matriz, somente $P3$ está calculando. Quando as unidades de processamento terminam o cálculo $q_1 \times d_1$, elas iniciam o cálculo de $q_1 \times d_2$ e assim consecutivamente, até que $q_x \times d_y$ seja calculado.

Na paralelização em granularidade grossa, cada unidade de processamento recebe as sequências de busca e um subconjunto das sequências de banco de dados. As unidades de processamento calculam então as matrizes de programação dinâmica entre q_1 e um subconjunto de d, sem comunicação entre as mesmas, conforme mostrado na Figura 3.3(b). Depois disso, as unidades de processamento fazem o cálculo de $q_2 \times d$, $q_3 \times d$ e assim consecutivamente, até que a comparação $q_x \times d$ seja concluída.

Na paralelização em granularidade muito grossa, cada unidade de processamento compara uma sequência de busca diferente com todas as sequências de banco de dados (conjunto d), conforme ilustrado na Figura 3.3(c). Por exemplo, $P0$ compara q_1 a d, $P1$ compara q_2 a d e assim por diante. Note que, nesse caso, o número de cálculos de matrizes realizados em cada etapa de comparação é bem grande. Por essa razão, essa abordagem pode levar facilmente a desbalanceamento de carga, caso as unidades de processamento possuam poder de computação muito distinto.



(a) Granularidade fina (b) Granularidade grossa

(c) Granularidade muito grossa

Figura 3.3: Estratégias para paralelizar o algoritmo SW.

3.5 Algoritmos heurísticos

3.5.1 Algoritmo FASTA

O algoritmo heurístico FASTA é executado em quatro passos para determinar o escore de similaridade entre duas sequências [58].

Grande parte da velocidade e seletividade do algoritmo FASTA são alcançados no primeiro passo, que utiliza uma tabela de busca para localizar todas as identidades ou grupos de identidades entre duas sequências de DNA ou de proteína. O parâmetro *ktup* determina quantas identidades consecutivas são exigidas em um *match*. No primeiro passo, as 10 melhores regiões diagonais são encontradas usando uma fórmula baseada no valor do parâmetro *ktup* e na distância entre os *matches* [58].

No segundo passo, as 10 regiões diagonais são recomparadas usando uma das matrizes de substituição que permitem trocas de resíduos de forma conservativa com o objetivo de aumentar o escore obtido. Para cada uma destas 10 regiões diagonais uma sub-região com escore máximo é identificada.

No terceiro passo, o algoritmo utiliza a melhor sub-região encontrada para caracterizar a similaridade entre as duas sequências. Além disso, ele verifica se mais de uma dessas sub-regiões podem ser unidas. Dadas as localizações das sub-regiões, seus escores e uma penalidade de união (análoga à penalidade de gap), o algoritmo FASTA calcula um alinhamento ótimo de sub-regiões.

No último passo, os melhores conjuntos de sub-regiões são alinhados utilizando uma versão modificada do algoritmo SW (Seção 3.4.2). Essa última comparação considera

todos os possíveis alinhamentos que são aproximações do alinhamento da sub-região com maior escore encontrada anteriormente.

3.5.2 BLAST

BLAST (*Basic Local Alignment Search Tool*) [1] é uma heurística que tenta otimizar a medida de similaridade. Permite a criação de um *tradeoff* entre velocidade do algoritmo e sensibilidade por meio de um parâmetro T. Um valor mais alto desse parâmetro aumenta a velocidade mas também aumenta a probabilidade de regiões de baixa similaridades serem ignoradas.

A idéia central por trás do algoritmo BLAST é que alinhamentos estatisticamente relevantes provavelmente possuem um par de palavras idênticas alinhadas com escore alto. O BLAST primeiro processa o banco de dados por palavras (tipicamente de tamanho 3 três para proteínas) que possuem escore igual ou maior que T quando alinhadas com uma palavra da sequência de busca. Qualquer par de palavras alinhadas que satisfaz esta condição é chamado hit.

O segundo passo do algoritmo avalia se cada um dos hits estão contidos em alinhamentos com escore alto o suficiente para serem reportados. Isso é feito estendendo-se os hits nas duas direções até que o escore decaia a um valor X menor que o maior escore encontrado até então. Esse passo de extensão dos hits é computacionalmente caro. Com valores de T e X necessários para obter uma sensibilidade razoável na comparação o passo de extensão é responsável por pelo menos 90% do tempo de execução do BLAST. Por esse motivo é recomendável diminuir o número de extensões utilizadas.

No último passo os alinhamentos obtidos são avaliados quanto à sua relevância estatística. Os alinhamentos considerados significantes são chamados *High-scoring Segment Pairs* (HSP). A avaliação utiliza um parâmetro S para ordenar os alinhamentos encontrados.

Capítulo 4

Alocação de tarefas em ambientes paralelos

Em sistemas paralelos, um dos problemas importantes é o escalonamento de tarefas, que consiste em determinar em que unidades de processamento as tarefas serão executadas [71]. No problema de escalonamento de tarefas, existe uma relação de dependência temporal entre as tarefas, que deve ser respeitada. As políticas de escalonamento de tarefas visam otimizar certos critérios, como minimizar o tempo de resposta ou maximizar o *throughput*. Um dos principais fatores no desempenho do sistema paralelo é, portanto, o escalonador, responsável por garantir que os recursos disponíveis sejam utilizados por tarefas visando otimizar o critério escolhido.

A alocação de tarefas é um problema muito parecido com o escalonamento de tarefas. A diferença entre os dois consiste no fato de não existir uma ordem de precedência temporal na alocação de tarefas, sendo permitido que todas as tarefas comecem a execução ao mesmo tempo. Ambos os problemas são NP-Completos [57] e foram extensivamente estudados na literatura.

4.1 Definições gerais e terminologia

A terminologia abaixo, adaptada de Xhafa e Abraham [75], foi originalmente proposta para sistemas em *grid*, porém pode ser utilizada também em sistemas distribuídos de maneira geral. A seguir são definidos os conceitos que serão utilizados ao longo desse capítulo.

Tarefa Representa uma unidade computacional (tipicamente um programa e possivelmente dados associados) que se executa em um nó do sistema. Embora não exista na literatura uma definição exata de tarefa, usualmente é considerada uma unidade escalonável indivisível. Tarefas podem ser independentes ou pode haver dependências entre elas.

Job Um *job* é uma atividade computacional formada por várias tarefas que pode requisitar diferentes unidades de processamento, recursos (processamento, quantidade de nós, memória, bibliotecas) e pode possuir restrições, usualmente expressas por meio da descrição do *job*. Nos casos mais simples, um *job* pode possuir apenas uma tarefa.

Recurso É basicamente uma entidade computacional (nó ou processador) no qual as tarefas são escalonadas, alocadas e processadas de acordo com as regras adotadas no sistema. Recursos podem possuir as suas próprias características como capacidade de processamento, memória e software instalados. Alguns parâmetros são normalmente atribuídos aos recursos, entre eles a velocidade de processamento e a carga de trabalho, que podem variar com o tempo. Além disso, os recursos podem pertencer a diferentes domínios administrativos, levando a diferentes políticas de uso e acesso.

Escalonador Componente de software encarregado de fazer o mapeamento de tarefas aos recursos, utilizando múltiplos critérios e configurações do ambiente. Níveis diferentes em escalonadores foram identificados na literatura, incluindo super-escalonadores, meta-escalonadores e escalonadores locais/distribuídos. Como componente importante de qualquer sistema distribuído, o escalonador interage com os outros componentes: sistema de informação, gerenciamento local de recursos e de rede. Em ambientes distribuídos mais complexos, como por exemplo sistemas em *grid*, todos esses tipos de escalonadores devem coexistir, e podem possuir diferentes critérios conflitantes. Por isso, existe a necessidade de interação e coordenação entre os diferentes escalonadores para executar as tarefas.

4.2 Taxonomia de Casavant e Kuhl

A taxonomia de Casavant e Kuhl [9] foi proposta em 1988 para a classificação de escalonadores de tarefas para sistemas paralelos e distribuídos. É composta de uma parte hierárquica e de uma parte linear, detalhadas a seguir.

4.2.1 Classificação hierárquica

A classificação hierárquica apresenta uma distinção clara entre os diversos níveis da hierarquia, permitindo a diferenciação e comparação entre diferentes escalonadores. A Figura 4.1 apresenta a classificação hierárquica da taxonomia de Casavant e Kuhl:

Local vs. Global No nível mais alto, Casavant e Kuhl distinguem os escalonadores locais dos globais. Os escalonadores locais são responsáveis por alocar tarefas em um único processador, enquanto os escalonadores globais alocam tarefas à múltiplos processadores com o objetivo de otimizar um ou mais objetivos de desempenho relevantes à todo o sistema. Consequentemente, os escalonadores utilizados em sistemas paralelos e distribuídos são considerados globais.

Estático vs. Dinâmico No próximo nível da hierarquia são comparados os escalonamentos estático e dinâmico. Essa escolha indica em que momento as decisões acerca do escalonamento são tomadas. No caso do escalonamento estático, assume-se que as informações relevantes à execução das tarefas estão disponíveis no momento em que o *job* é submetido ao escalonador. Em contra-partida, no escalonamento dinâmico parte-se da idéia de que as tarefas são alocadas à medida que são submetidas ao escalonador e

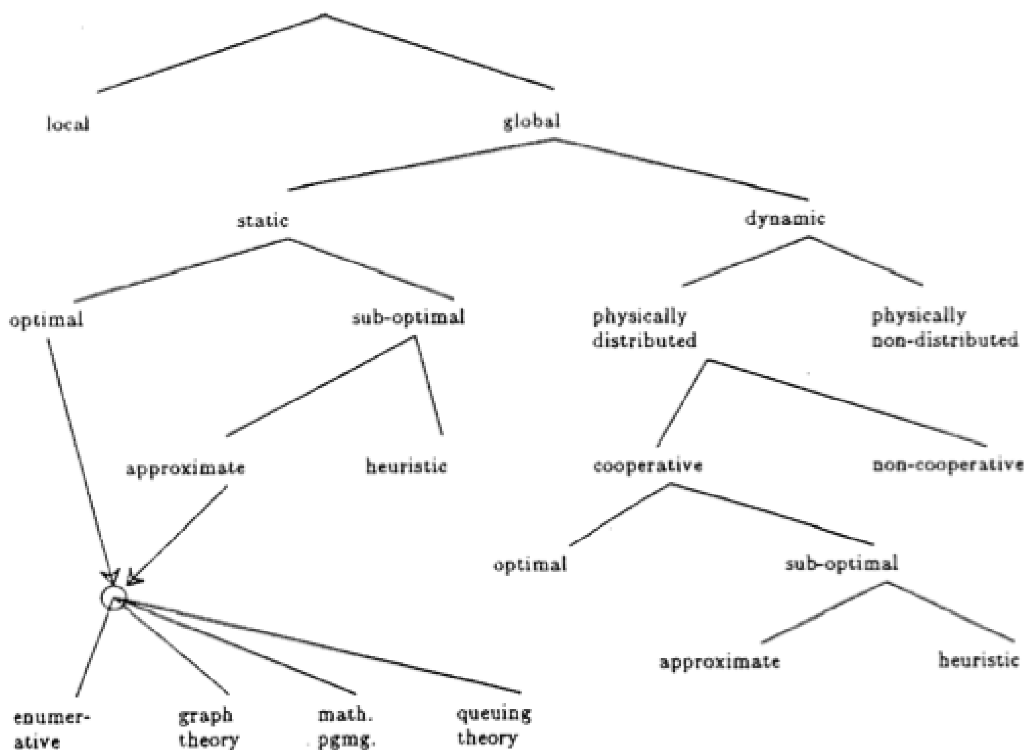


Figura 4.1: Classificação hierárquica de Casavant e Kuhl [9].

simultaneamente à execução do *job*. Essa prática é vantajosa nas vezes em que é impossível determinar o tempo de execução, direção dos saltos e extensão dos laços das tarefas submetidas bem como prever o padrão de submissão das tarefas.

Ótimo vs. Sub-ótimo Nos casos em que todas as informações referentes ao estado dos recursos e às tarefas são conhecidas, uma alocação ótima é possível em casos muito específicos. No entanto, devido à natureza NP-Completa do problema de escalonamento genérico [57], estudos tentam encontrar soluções sub-ótimas, que podem ser subsequentemente divididas nas duas seguintes categorias:

Aproximado vs. Heurístico Os algoritmos ditos aproximados utilizam modelos computacionais formais para, ao invés de procurar por uma solução ótima, procurar por uma solução considerada suficientemente boa. No caso em que uma métrica é utilizada para avaliar a solução, essa técnica pode ser utilizada para tentar diminuir o tempo que o algoritmo leva para encontrar um escalonamento aceitável, segundo esses padrões.

A outra tendência na categoria sub-ótima é chamada heurística, que representa a classe de algoritmos que utilizam premissas consideradas realistas sobre o que é conhecido, a priori, acerca das tarefas e as características de utilização dos recursos no sistema. Essa tendência também inclui os algoritmos que não conseguem encontrar uma solução ótima para o problema de escalonamento mas exigem apenas uma quantidade razoável de recursos do sistema para realizar a sua função. A análise desse tipo de solução é usualmente baseada em experimentos com aplicações reais ou em simulações.

Distribuído vs. Centralizado Nos cenários de escalonamento dinâmico, a responsabilidade das decisões globais de escalonamento pode ficar com um único nó de processamento, ou ser compartilhada por múltiplos nós. Nesse cenário, vários *jobs* podem ser submetidos ou requerer recursos simultaneamente. A estratégia centralizada pode ser implementada mais facilmente, mas sofre com a falta de escalabilidade, tolerância à falhas e a possibilidade de se tornar um gargalo de desempenho.

Cooperativo vs. Não cooperativo Se um algoritmo de escalonamento distribuído é utilizado em um determinado cenário, a próxima decisão a ser tomada deve ser se a decisão de escalonamento é tomada de forma cooperativa ou independente. No caso não cooperativo, escalonadores individuais atuam sozinhos como entidades autônomas, e tomam as decisões de escalonamento sem levar em consideração as decisões tomadas no restante do sistema.

No caso cooperativo, cada escalonador possui a responsabilidade de executar a sua porção de tarefas escalonadas, mas todos os escalonadores envolvidos estão trabalhando visando o mesmo objetivo em nível de sistema. Cada escalonador possui as suas próprias políticas locais mas as tomadas de decisão são feitas em consonância com os outros escalonadores do sistema para atingir o mesmo objetivo, ao invés de tomar decisões que consideram somete o desempenho local ou o desempenho de uma tarefa particular.

4.2.2 Classificação linear

A classificação linear define características que permeiam os diversos níveis da hierarquia, descritas a seguir.

Adaptativo vs. Não adaptativo Uma solução adaptativa tem como característica possuir algoritmos e parâmetros que se modificam dinamicamente de acordo com comportamentos anteriores e atuais do sistema. Um exemplo de escalonador adaptativo seria um escalonador que usa um algoritmo quando o sistema está no estado não saturado e outro quando o sistema está no estado saturado.

Sendo assim, em resposta ao comportamento do sistema, o escalonador pode passar a ignorar um determinado parâmetro ou reduzir a sua importância se acreditar que este está fornecendo informações inconsistentes com a entrada ou não está fornecendo qualquer informação relativa às modificações no sistema com relação aos outros parâmetros observados.

Balanceamento de carga Essa categoria de políticas aborda o problema com a filosofia de que o uso igualitário dos recursos de hardware contribui para que as tarefas como um todo terminem a execução em tempo reduzido. A idéia básica é tentar fazer com que todos os processadores possuam a mesma carga, permitindo o progresso de todas as tarefas em todos os nós participantes aproximadamente na mesma taxa. O balanceamento de carga é mais simples quando os nós do sistema são homogêneos já que isso permite que todos os nós saibam mais sobre a estrutura geral dos outros nós. Além disso, informações são geralmente transmitidas entre os nós periodicamente ou sob demanda para permitir que todos os nós obtenham informações sobre o estado global do sistema. Os nós então podem atuar de forma cooperativa com o objetivo de retirar carga de trabalho de nós mais

ocupados e transferi-la a nós com menos carga. Essa é uma classe de soluções que apóia-se na premissa de que as informações disponíveis em cada nó são relativamente precisas, o que é necessário para que as tarefas não fiquem circulando entre os nós e deixem de avançar na sua execução.

Bidding Nessa classe de políticas, um protocolo básico existe que descreve a forma com a qual as tarefas são designadas a processadores. O escalonador resultante é normalmente cooperativo no sentido que informação suficiente é trocada entre os nós com tarefas para executar e os nós disponíveis para execução para que as tarefas sejam alocadas de tal forma que todos os nós se beneficiam.

Para ilustrar o mecanismo básico envolvendo *bidding*, Casavant e Kuhl utilizam a terminologia encontrada em [73]. Cada nó na rede é responsável por duas funções com relação ao processo de *bidding*: gerente e contratante. O gerente representa a tarefa que precisa de um local para executar, e o contratante representa um nó capaz de realizar trabalho para outros nós. Note que um único nó realiza as duas funções, e que não há nós estritamente gerentes ou contratantes. O gerente anuncia a existência de uma tarefa a ser executada por meio de um anúncio de tarefa, que então recebe lances dos outros nós (contratantes). A quantidade e o tipo de informação trocada são fatores essenciais ao determinar a efetividade e o desempenho de um escalonador utilizando a noção de *bidding*. Uma característica importante dessa classe de escalonadores é que todos os nós geralmente possuem autonomia no sentido de que o gerente possui o poder de decidir para onde enviar uma tarefa, dentre os nós que efetuaram lances. Além disso, os contratantes também são autônomos já que eles nunca são forçados a aceitar trabalho se eles não quiserem.

Alocação Única vs. Alocação Dinâmica Na alocação única, somente uma decisão de escalonamento é tomada e essa decisão não pode ser revista ao longo da execução da tarefa. Em outras palavras, caso a tarefa t seja atribuída ao nó de processamento p, toda a execução de t será feita por p.

Em contraste com a abordagem de alocação única, soluções na classe de realocação dinâmica tentam melhorar decisões tomadas anteriormente utilizando informações sobre as unidades de processamento e as tarefas em execução. Esses sistemas utilizam informações criadas dinamicamente para se adaptar às demandas dos processos dos usuários. Essa adaptação possui a forma de migração de tarefas. Existe claramente um preço a ser pago em termos de overhead, e esse preço deve ser cuidadosamente considerado em relação aos possíveis benefícios.

4.3 Taxonomia de Krauter *et. al*

A taxonomia de Krauter *et. al* [38] foi proposta para sistemas de *grid* e consiste em uma classificação em cinco categorias (organização, recursos, escalonamento, previsão de estado e re-escalonamento) que podem ser combinadas para descrever um determinado escalonador.

4.3.1 Organização do sistema

A organização das máquinas do sistema descreve como estas tomam as decisões de escalonamento, a estrutura da comunicação entre elas e os diferentes papéis desempenhados nas decisões de escalonamento.

Organização Linear Todas as máquinas podem se comunicar diretamente sem ter que passar por um intermediário.

Organização em Células As máquinas da mesma célula comunicam-se utilizando a organização linear. Algumas máquinas das células atuam como elementos de fronteira que são responsáveis por toda a comunicação com elementos de fora da célula. A estrutura interna da célula não é visível de outra célula, apenas as máquinas da fronteira o são.

Organização Hierárquica As máquinas são organizadas de maneira hierárquica, de modo que máquinas que estão no mesmo nível podem se comunicar diretamente. Além disso, a comunicação direta também é possível com máquinas imediatamente acima ou abaixo na hierarquia.

4.3.2 Recursos

Organização em Esquemas Os dados que descrevem um recurso são descritos por meio de uma linguagem pré-definida. Em alguns sistemas, uma linguagem de comandos é integrada à linguagem dos esquemas. Em um esquema extensível, novos tipos de esquema para descrever os recursos podem ser adicionados.

Modelo de Objeto As operações sobre os recursos são definidas como parte do modelo de recursos. Assim como com os esquemas, o modelo de objeto pode ser predeterminado e fixo como parte da definição do sistema. No modelo de objeto extensível, os recursos provêem um mecanismo para estender a definição do modelo de objeto gerenciado pelo sistema.

4.3.3 Escalonamento

Centralizado Existe apenas um controlador de escalonamento que é responsável pelas tomadas de decisão. Tal organização possui várias vantagens, incluindo facilidade de gerenciamento, implementação e a habilidade de co-alocar recursos. As desvantagens dessa organização são a falta de escalabilidade, tolerância a falhas e a dificuldade em acomodar múltiplas políticas, o que pode ser mais significativo que as vantagens.

Hierárquico Os controladores são organizados em hierarquia. Uma forma simples de organizar os controladores seria permitir que os escalonadores de alto nível controlem um grande número de recursos e os escalonadores de baixo nível controlem um número menor de recursos. Se comparada com a abordagem centralizada, esta resolve os problemas de escalabilidade e tolerância a falhas. Além disso essa abordagem retêm algumas das vantagens da centralizada, como co-alocação dos recursos.

Descentralizado No escalonamento descentralizado, todos os nós do sistema estão potencialmente envolvidos nas decisões de escalonamento. Essa abordagem resolve naturalmente os problemas de tolerância a falhas e pouca escalabilidade mas introduz outros problemas que incluem gerenciamento, rastreamento e co-alocação. Espera-se que essa organização funcione bem em redes de larga escala mas é necessário que os escalonadores coordenem-se por meio de uma forma de descoberta de recursos ou protocolos de troca de recursos. O overhead na operação desses protocolos será o fator determinante para a escalabilidade do sistema como um todo.

4.3.4 Previsão de estado

No caso dos sistemas distribuídos e paralelos, a previsão de estado é sempre feita utilizando informação parcial devido à propagação da informação possuir um certo atraso. O foco da taxonomia de Krauter *et. al* está no mecanismo utilizado para prever o estado, que afeta a implementação da coleta e manutenção de informação nos nós de processamento.

Não preditiva Utiliza apenas a execução atual do *job*, sem levar em conta informação histórica sobre o estado dos recursos. Utiliza heurísticas baseadas nas características do *job* e dos recursos ou um modelo de probabilidade baseado em uma análise estatística baseada nas características esperadas do *job*.

Preditiva Utiliza informações atuais e históricas, como por exemplo execuções anteriores do *job*, para prever o estado. Modelos como esse utilizam abordagens heurísticas, modelos de mercado ou de aprendizagem. Na abordagem heurística, regras pré-definidas são utilizadas para orientar a previsão de estado baseando-se em comportamentos esperados das aplicações de *grid*. Em um modelo de mercado, recursos são comprados e vendidos utilizando dinâmicas de mercado que levam em consideração disponibilidade e demanda dos recursos. No modelo de aprendizado, esquemas de aprendizado são utilizados para prever o estado baseados em distribuições potencialmente desconhecidas.

4.3.5 Re-escalonamento

As características de re-escalonamento de um sistema paralelo ou distribuído determinam quando o escalonamento atual é re-examinado e as execuções das tarefas reordenadas. A execução das tarefas pode ser reordenada para maximizar a utilização dos recursos, *throughput* ou outra métrica dependendo da política de escalonamento. O re-escalonamento pode ser realizado em lotes ou ser orientado a evento. O re-escalonamento periódico ou em lotes aborda pedidos de recursos de grupos de tarefas que são então processados em intervalos. Re-escalonamento orientado à eventos é feito assim que o sistema recebe o pedido de recurso ou evento.

O re-escalonamento em lotes permite potencialmente uma utilização mais eficiente dos recursos já que mais pedidos podem ser considerados simultaneamente. Esquemas orientados a eventos podem reagir mais rápido e fornecer latências menores.

4.4 Classes de Aplicações Paralelas

Além de conhecer o ambiente e coletar informações ao longo da execução, o escalonador, responsável por alocar tarefas no ambiente paralelo, precisa conhecer a aplicação sendo executada. Intuitivamente, podemos afirmar que quanto mais o escalonador conhecer a aplicação mais fácil será para ele ser eficiente e obter desempenho satisfatório ao alocar as tarefas da aplicação no ambiente paralelo. Existem escalonadores criados com o objetivo de serem genéricos, podendo gerenciar a execução no ambiente paralelo de qualquer aplicação que siga um conjunto de regras, como por exemplo utilizar uma API para enviar e receber mensagens, iniciar e terminar a execução.

Também existem escalonadores que objetivam executar classes de aplicações que possuem uma estrutura em comum. Algumas classes já foram estudadas na literatura, como as aplicações *parameter-sweep*, *bag of tasks* e *workflow*. Uma das vantagens desse tipo de escalonador é que ele naturalmente possui mais informações sobre as aplicações, facilitando a implementações de políticas de escalonamento que são sabidamente eficientes para aquele tipo de aplicação.

4.4.1 Aplicações do tipo *Parameter Sweep*

Uma aplicação que se encaixa nessa classe pode ser vista como sendo composta por n tarefas sequênciais independentes, ou seja, não existe nenhum tipo de comunicação entre tarefas ou relação de precedência entre elas. Além disso, todas as tarefas realizam exatamente o mesmo tipo de processamento. A única diferença entre as tarefas é o conjunto de parâmetros passados a cada uma [8].

As aplicações do tipo *Parameter Sweep* são, em geral, aplicações desenvolvidas para explorar diferentes soluções de um problema. São criadas várias tarefas, cada uma resolvendo o mesmo problema com um conjunto de parâmetros. Por serem tarefas independentes entre si, muitos consideram esse tipo de aplicação ideal para ser executado em sistemas distribuídos cuja distribuição geográfica pode implicar em altos custos de comunicação entre os nodos.

4.4.2 Aplicações do tipo *Bag of Tasks*

Aplicações da classe *Bag of Tasks* [12] podem ser vistas como generalizações das aplicações do tipo *Parameter Sweep*. Assim como aquelas, são compostas por n tarefas totalmente independentes entre si. No entanto, não há nada que indique que as tarefas compõem uma aplicação que executa o mesmo tipo de processamento. Ou seja, duas tarefas quaisquer desse conjunto de n tarefas podem estar realizando processamento distintos uma da outra.

4.4.3 Aplicações *Workflow*

Uma aplicação *Workflow* é uma aplicação composta por uma coleção de componentes que devem ser executados pelo escalonador de acordo com uma ordem parcial determinada por dependências de controle e de dados. Essa classe de aplicações representa todas as aplicações que podem ser descritas por meio de um grafo acíclico dirigido (DAG) $G = (V, E)$ em que os vértices V são as tarefas e as arestas E são as relações temporais

entre as tarefas [43]. Caso exista uma aresta direcionada da tarefa T_1 para a tarefa T_2, T_2 não pode iniciar sua execução antes que T_1 termine.

Essas relações temporais tem de ser levadas em conta pelo escalonador, tornando-o geralmente mais complexo do que os escalonadores de tarefas *parameter-sweep* ou *bag-of-tasks*.

4.5 Uso de Informações Dinâmicas

Os escalonadores estáticos dependem de conhecimento prévio de características relevantes tanto da aplicação quanto do sistema. Isso não ocorre frequentemente, uma vez que sistemas paralelos tendem a ser utilizados por diversas aplicações simultaneamente e, mesmo se esse não for o caso, estão sujeitos a variações no ambiente que podem afetar o desempenho do sistema [9].

Dentre as informações consideradas importantes para o escalonador, algumas variam pouco ou não variam ao longo do tempo, como capacidade total de memória, processamento e armazenamento e comunicação. Por outro lado, certas informações variam ao longo do tempo, podendo sofrer flutuações severas ou até indisponibilidade. As principais informações desse tipo são o nível de utilização do processador, da memória e da largura de banda disponível, mas fatores elétricos e outros fatores ambientais também podem ser importantes.

Por esse motivo, um sistema desenhado tendo em mente robustez, ou seja, a capacidade de se adaptar tendo em vista situações adversas, precisa levar em consideração características dinâmicas coletadas antes e durante a execução da aplicação. A título de exemplo, podemos considerar o escalonador de um sistema operacional como um escalonador que utiliza informações dinâmicas para modificar os seus parâmetros de alocação de processos. Normalmente, um processo consegue ser alocado para execução de forma quase integral enquanto não houver outros processos suspensos aguardando execução. A partir do momento que um ou mais processos competem pelo processador, este é obrigado a rever a frequência com a qual o processador é alocado ao primeiro processo, passando então a dividir melhor o tempo.

4.6 Alocação Mestre-Escravo

Em um modelo de aplicação composto por um nó mestre e vários nós escravos, o mestre é responsável por controlar a distribuição de tarefas e coletar os resultados. Nesse modelo, o número de escravos controlados por um mestre depende da carga de trabalho do mestre e da disponibilidade de recursos. O número de tarefas dentre as N tarefas distribuídas a um escravo S_i é determinado pela função $A(N, S_i)$. Nesse caso, A indica uma política de alocação de tarefas específica. Nas seções 4.6.1 a 4.6.4 serão detalhadas algumas políticas de alocação.

4.6.1 *Fixed* (F)

A política de alocação *Fixed* [65] é adequada aos sistemas homogêneos com recursos dedicados e aplicações que não apresentam variações no seu comportamento ou no tempo

de execução. Nesses casos, uma abordagem estática pode ser empregada, dividindo uniformemente as N tarefas a serem executadas pelo número de nós escravos (S). Com isso, a política de alocação é definida segundo a Equação 4.1.

$$F(N, S_i) = \frac{N}{S} \tag{4.1}$$

Uma desvantagem dessa estratégia de alocação é que nem sempre é possível determinar de forma precisa a quantidade de trabalho necessário para um determinado problema. Nesses casos, erros na previsão dos parâmetros podem resultar em erros de superestimação da carga de trabalho em alguns dos nós e subestimação em outros, causando um aumento no tempo total de execução, uma vez que algumas tarefas levarão mais tempo que o estimado para executar enquanto outros nós ficarão ociosos.

4.6.2 *Self-Scheduling* (SS)

A estratégia de alocação *Self-Scheduling* [72] distribui as tarefas à medida que são solicitadas pelos nós escravos, por estarem ociosos. Quando o nó termina o processamento de uma tarefa, uma nova é solicitada ao nó mestre, até que todas as tarefas sejam executadas. A Equação 4.2 descreve essa estratégia.

$$SS(N, S_i) = 1, \tag{4.2}$$

enquanto existirem tarefas a serem alocadas.

Nessa estratégia, cada nó recebe apenas uma tarefa. O máximo de tempo que um conjunto de nós pode ficar ocioso é determinado pelo tempo de processamento de uma tarefa pelo nó escravo mais lento. No entanto, uma desvantagem da estratégia *Self-Scheduling* é o grande número de mensagens trocadas entre os nós escravos e o nó mestre.

4.6.3 *Guided Self-Scheduling* (GSS)

A estratégia *Guided Self-Scheduling* [60] aloca tarefas em grupos que diminuem de tamanho exponencialmente ao longo do tempo. Nessa estratégia, o tamanho dos blocos de tarefas em uma iteração é $\frac{1}{S}$ do total de tarefas restantes. A Equação 4.3 representa essa política.

$$GSS(s, N, S_i) = \max(\frac{N(1 - \frac{1}{S})^{s-1}}{S}, 1), \tag{4.3}$$

sendo que s representa o estágio de alocação das tarefas e $s > 0$.

Na estratégia GSS o tamanho unidades de alocação é relativamente grande no começo e vai diminuindo exponencialmente a cada nova alocação. Uma das suas desvantagens é que, em um ambiente heterogêneo, é possível que blocos grandes de processamento sejam atribuídos a nós com pouco poder computacional, causando um desbalanceamento de carga e, consequentemente, aumentando o tempo de processamento.

4.6.4 *Package Weighted Adaptive Self-Scheduling* (PSS)

O *Package Weighted Adaptive Self-Scheduling* (PSS) [69] é uma estratégia utilizada para adaptar o tamanho dos blocos de alocação durante a execução, levando em consideração a heterogeneidade dos escravos e as características dinâmicas do ambiente. Além disso, essa estratégia define a atribuição de pesos aos escravos de forma genérica, facilitando a utilização de qualquer função de alocação, $A(N, P)$.

A Equação 4.4 mostra a estratégia PSS. Nessa equação, $A(N, S)$ é uma função de alocação para um sistema com N tarefas, um mestre m e S escravos e $\Phi(m, S_i, S)$ é a média ponderada do desempenho das últimas Ω execuções de cada um dos escravos S_i.

$$A(m, S_i, N, S) = A(N, S) * \Phi(m, S_i, S) \quad (4.4)$$

Na estratégia PSS, o número de tarefas a serem alocadas a cada escravo é calculado com base em um histórico de notificações recentes de desempenho. Sendo assim, essa estratégia de alocação é capaz de se adaptar às características de ambientes heterogêneos e não-dedicados.

4.6.5 *Work Stealing*

Work Stealing [5] é uma estratégia bastante popular de alocação de tarefas, proposta inicialmente para multiprocessadores com memória compartilhada no contexto de aplicações *multi-threaded*. Cada processador possui uma fila duplamente encadeada e, quando um processador fica ocioso, ele "rouba" uma tarefa de um dos outros processadores, escolhido aleatoriamente, desde que a fila não esteja vazia.

Para essa política, alguns resultados teóricos importantes relacionados com o tempo de execução e o número de tentativas de roubo foram obtidos Um deses resultados nos mostra que a estratégia *Work Stealing* é estável, no sentido de que a comunicação é iniciada apenas quando os nós envolvidos estão ociosos. No caso dos ambientes distribuídos, algumas tentativas de uso da estratégia *Work Stealing* foram realizadas. Kumar et al. [39] propuseram uma framework para avaliar o balanceamento de carga de alguns algoritmos em arquiteturas paralelas com memória distribuída e com processadores idênticos.

Capítulo 5

Estado da arte

Neste capítulo, apresentamos diversas estratégias existentes na literatura para a execução do Smith-Waterman em plataformas compostas por diversos processadores de uso geral e/ou aceleradores. Ao final do capítulo, apresentamos e discutimos um quadro comparativo das diversas abordagens.

5.1 Abordagem de Bonny e Zidan (2011) [6]

Bonny e Zidan [6] implementam o algoritmo Smith-Waterman (Seção 3.4.2) em GPU com o objetivo de aumentar a eficiência da execução nos casos em que há sequências pequenas no banco de dados. Nesses casos, a eficiência é geralmente menor porque a GPU fica relativamente menos tempo processando as células e mais tempo é gasto com o escalonamento do *kernel* e com as operações de acesso à memória e com a comunicação com o *host*.

Usualmente, um banco de dados contém sequências de tamanhos diferentes. Os autores argumentam que a comparação de sequências pequenas não é tão eficiente quanto a comparação de sequências maiores. No entanto, quando as sequências maiores são comparadas em uma GPU e as sequências menores são comparadas na CPU simultaneamente, o tempo necessário para que todas as comparações sejam realizadas diminui consideravelmente.

Para verificar essa afirmação, os autores utilizaram o algoritmo SW desenvolvido por Farrar [21], comparando sequências grandes em GPUs e sequências pequenas na CPU. Foi atingida uma melhora no desempenho do algoritmo da ordem de 2,2 vezes quando comparado com o algoritmo de Farrar executando somente na CPU.

Para implementar esta técnica, os autores primeiro ordenam as sequências do banco de dados em ordem decrescente de tamanho. Esta ordenação é realizada antes do início do processamento e apenas uma vez. Depois de ordenar as sequências, um algoritmo é utilizado para separá-las entre a GPU e a CPU de tal forma que as sequências longas são enviadas à GPU e as sequências curtas à CPU (Figura 5.1). A sequência de busca é enviada a ambos. A GPU e a CPU comparam então as sequências e determinam os maiores escores de alinhamento separadamente.

Para determinar quais sequências serão executadas na GPU e quais serão executadas na CPU é usado o seguinte algoritmo: são escolhidos, inicialmente, dois pontos de divisão, um no começo do banco de dados e outro no final. Depois de cada divisão, o desempenho

em GCUPS é computado. A métrica CUPS determina o número de células da matriz de programação dinâmica calculadas por segundo. Se a diferença entre as duas medidas for maior que a precisão exigida, então uma nova divisão é feita entre as duas divisões anteriores. O desempenho dessa nova divisão é computado e se for maior que o apresentado pela primeira divisão então esta se torna a nova divisão. Caso contrário, esta se torna a segunda divisão e o desempenho é computado até que a diferença entre as duas divisões seja menor que a precisão exigida. A Figura 5.1 ilustra essa técnica.

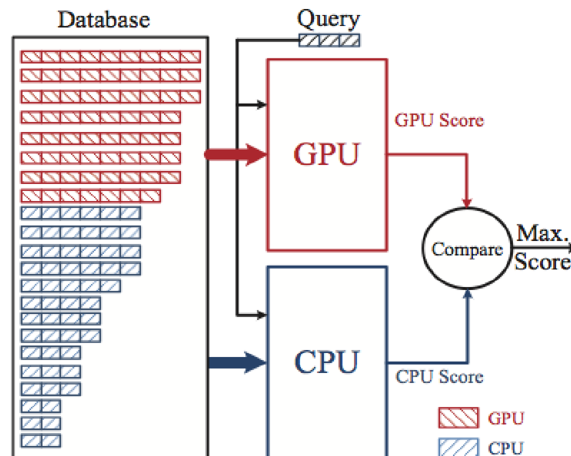


Figura 5.1: Técnica utilizada por Zidan para comparar sequências do banco de dados na GPU e na CPU simultaneamente [6].

A técnica proposta pelos autores foi implementada e executada em um processador Intel Xeon X5550 (2,676 GHz) e uma GPU Nvidia Quadro FX 4800 (602 MHz). Os testes foram realizados utilizando as sequências de proteínas do banco de dados Swiss-Prot versão 15.12 de 15 de dezembro de 2009.

Foi analisado o desempenho obtido na implementação do algoritmo SW considerando dois casos: quando as sequências do banco de dados são distribuídas corretamente (as sequências longas são comparadas pela GPU e as sequências curtas pela CPU) e quando essa ordem é invertida. A melhoria no desempenho obtida ao distribuir corretamente as sequências foi de 1,35 vezes. Também foi mostrado o desempenho obtido em três casos: apenas a CPU é utilizada, apenas a GPU é utilizada e ambos são utilizadas. Nesse caso, os desempenhos obtidos utilizando apenas a CPU, apenas a GPU e ambos foram de 4,2 GCUPS, 4,7 GCUPS e 9,6 GCUPS, respectivamente.

5.2 Abordagem de Singh e Aruni (2011) [67]

Singh e Aruni [67] têm como foco no seu trabalho a extensão e integração das versões otimizadas do algoritmo SW já presentes na literatura tanto para a arquitetura Intel quanto para GPUs Nvidia. Para a GPU, foi escolhida a implementação CUDASW++ 2.0 [42], enquanto que, para a CPU, Singh e Aruni escolheram a implementação SWPS3 [70]. É proposta uma implementação híbrida que utiliza tanto a CPU quanto a GPU.

A implementação SWPS3 utiliza operações de *fork* e *join* para distribuir a carga de trabalho entre os núcleos disponíveis. Singh e Aruni adaptaram essa abordagem com uma

implementação que utiliza Pthread para diminuir o overhead e aumentar o desempenho do SWPS3. Além disso, estenderam a implementação CUDASW++ 2.0 para integrá-la à implementação SWPS3 utilizando Pthread.

Inicialmente, a GPU e a CPU foram testadas com relação ao desempenho e, de acordo com seus speedups relativos, as sequências do banco de dados foram divididas e distribuídas entre os núcleos do processador e as GPUs disponíveis. Essa abordagem permitiu aos autores escalar de forma transparente o desempenho em um sistema heterogêneo.

A implementação SWPS3 foi testada em um ambiente multi-core para avaliar o desempenho. Em um processador Intel Xeon E5420 executando a 2,5 GHz foi obtido um pico de desempenho de 16 GCUPS para sequências de tamanhos diferentes presentes no Swiss-Prot 56.6 de dezembro de 2008.

A implementação CUDASW++ 2.0 foi testada em uma única Tesla C1060, que possui 240 núcleos executando a 1,3 GHz e 4GB de memória global. O pico de desempenho obtido foi 15 GCUPS.

A execução simultânea na GPU e na CPU utilizando o mesmo banco de dados mostrou um pico de desempenho de 27 GCUPS, que está próximo da adição aritmética dos dois desempenhos, 31 GCUPS.

5.3 Abordagem de Chen *et. al* (2010) [11]

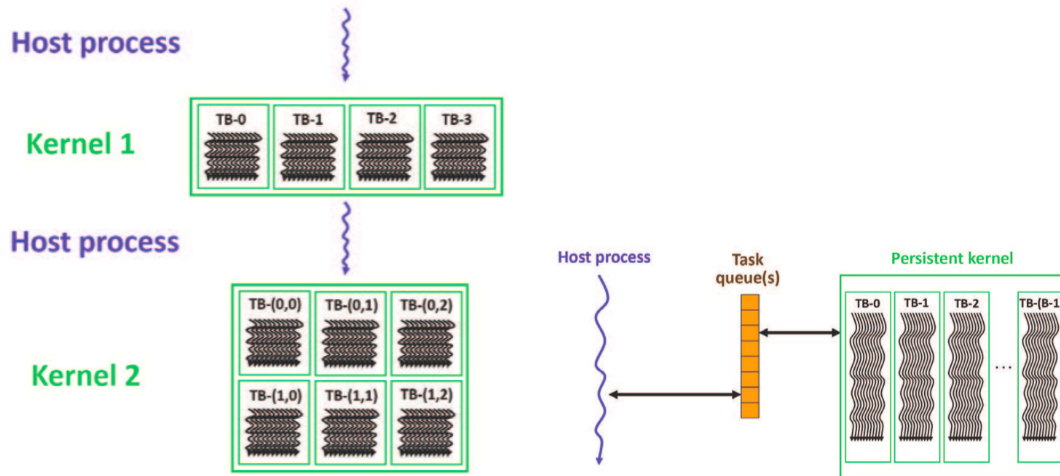
Chen *et. al* propuseram um esquema de execução baseado em tarefas capaz de balancear dinamicamente a carga de trabalho em GPUs de forma individual ou em múltiplas placas. O objetivo é utilizar o hardware disponível de forma eficiente.

Embora muitas aplicações sejam adequadas ao processamento paralelo de tarefas, um número grande de aplicações apresentam mais paralelismo de dados do que de tarefas [10]. Chen *et. al* afirmam, então, que a utilização de um esquema de programação paralela de tarefas facilitará o desenvolvimento desse tipo de aplicação em GPUs. Em segundo lugar, é afirmado que ao explorar de forma mais eficiente o paralelismo de tarefas é possível aproveitar melhor os recursos do hardware. Em terceiro lugar, é possível que algumas tarefas sejam incapazes de utilizar paralelismo de dados suficiente para ocupar eficientemente a GPU. A execução então de múltiplas tarefas de forma concorrente pode melhorar a utilização dos recursos computacionais da GPU e consequentemente aumentar o desempenho da aplicação. Foram feitas as seguintes contribuições:

- Identificação de um mecanismo de comunicação entre o *host* e a GPU baseado na tecnologia CUDA que funcione enquanto a GPU está processando.
- Introdução de um esquema de fila de tarefas utilizado para balancear dinamicamente a carga com uma granularidade mais fina do que o que é suportado atualmente pelos paradigmas de programação da arquitetura CUDA.
- Implementação de uma aplicação que trata de dinâmica molecular com o objetivo de aplicar o esquema de fila de tarefas.

Resultados obtidos com uma única GPU mostraram que o esquema implementado por eles é capaz de utilizar o hardware mais eficientemente do que apenas os mecanismos fornecidos pela arquitetura CUDA.

O paradigma atualmente empregado pela arquitetura CUDA exige, para executar diferentes tarefas, que a CPU do *host* execute as tarefas de forma sequencial (Figura 5.2(a)).



(a) Paradigma de programação fornecido pelo CUDA.

(b) Esquema de execução de tarefas utilizando uma fila.

Figura 5.2: Paradigma de programação do CUDA e esquema de execução de tarefas utilizando uma fila [11].

Na implementação proposta por Chen *et. al*, ao invés de executar diferentes *kernels* de forma sequencial, um único *kernel* persistente é executado. O número de blocos de *threads* depende do número máximo de tarefas exigido pela aplicação sendo executada. Como os blocos de *threads* não serão interrompidos até que eles finalizem a execução, eles ficarão ativos até que todas as tarefas distribuídas pela fila de tarefas sejam executadas.

Quando o *kernel* está executando na GPU, o processo do *host* é responsável por enfileirar tarefas a serem executadas na GPU. Já o *kernel* é responsável por retirar tarefas da fila e executá-las de acordo com as informações definidas anteriormente. A Figura 5.2(b) mostra a execução do processo do *host*, um *kernel* e vários blocos de *threads* executando tarefas.

Os resultados experimentais obtidos com apenas uma GPU mostram que a solução proposta é capaz de utilizar o hardware mais eficientemente do que a execução sequencial de *kernels*. Em sistemas com múltiplas GPUs, a solução proposta por Chen é capaz de atingir speedup próximo do linear, balanceamento de carga e uma melhoria na performance quando comparado com uma implementação puramente baseada em CUDA.

5.4 Abordagem de Benkrid *et. al* (2012) [4]

Benkrid *et. al* [4] apresentam um estudo comparativo entre três unidades de processamento diferentes: placa programável (FPGA), GPU e o processador Cell BE. O estudo é feito utilizando o algoritmo SW (Seção 3.4.2) e os critérios de comparação são velocidade da implementação, consumo de energia, custo do equipamento e custo de desenvolvimento. Ao contrário de outros trabalhos presentes na literatura, esse não utiliza as implementa-

ções já disponíveis e reúne um grupo de alunos com o objetivo de implementar o algoritmo SW para as diferentes plataformas.

O estudo mostra que o FPGA possui desempenho amplamente superior às outras plataformas no critério desempenho por watt, e também possui desempenho superior no critério desempenho por dólar, embora mais próximo das outras plataformas.

As comparações são realizadas utilizando os seguintes equipamentos: Virtex-4 da Xilinx , GeForce 8800GTX da Nvidia, Cell BE da IBM e o Pentium 4 Prescott, da Intel. Foi utilizado o banco de dados Swiss-Prot de agosto de 2008, quando continha 392768 sequências.

Seq. de busca	Tamanho	FPGA (s)	GPU (s)	Cell BE (s)	CPU (s)
P36515	4	1,5	4,1	0,5	24
P81780	8	1,6	4,1	1	30
P83511	16	1,6	4,3	1,3	43
O19927	32	1,6	4,7	1,4	62
A4T9V0	64	1,6	6,7	2,5	115
Q2IJ63	128	1,6	12,8	5,1	210
P28484	256	1,9	30	9,4	424
Q1JLB7	512	4,5	76	17,2	779
A2Q8L1	768	6,7	136,2	22,2	1356
P08715	1024	8,9	172,8	31,8	1817

Tabela 5.1: Comparação entre as plataformas FPGA, GPU, CellBE e processador de uso geral [4].

Como pode ser visto na Tabela 5.1, para sequências pequenas, o FPGA pode facilmente armazenar um número maior de elementos de processamento no *chip* e por isso, se houver banda suficiente para transmitir os dados em paralelo, o tempo de execução pode ser reduzido consideravelmente. A Tabela 5.2 apresenta o desempenho das plataformas em GCUPS e os speedups obtidos em comparação com a implementação serial utilizando a CPU.

Plataforma	GCUPS	Speedup
FPGA	19,4	228
GPU	1,2	14
Cell BE	3,84	45
CPU	0,085	1

Tabela 5.2: Desempenho das plataformas em GCUPS e os speedups obtidos em relação à implementação serial utilizando CPU [4].

Nesse caso, a implementação no FPGA atinge 19,4 GCUPS e executa mais de duzentas vezes mais rápido que a implementação na CPU, enquanto as implementações em GPU e utilizando o processador Cell BE atingem desempenho dezenas de vezes mais rápido que a implementação na CPU.

Os autores admitem que as implementações utilizadas nas outras plataformas (GPU, Cell BE e processador Intel) não são tão otimizadas quando as implementações disponíveis

na literatura. Por exemplo, a implementação utilizada por Benkrid *et. al* não utiliza as extensões SSE da Intel ou sequer executam o código em todos os núcleos disponíveis no processador.

5.5 Abordagem de Meng e Chadhary (2010) [45]

Meng e Chadhary [45] implementaram uma plataforma de computação heterogênea utilizando MPI (*Message Passing Interface*) para análise de sequências biológicas em ambientes não dedicados. A plataforma integra três plataformas heterogêneas: processadores convencionais com instruções SSE, coprocessadores FPGA e processadores legados, ou seja, que não utilizam as extensões SSE. Cada uma dessas unidades de processamento foi utilizada para executar o algoritmo SW com *affine-gap*, em alguns casos no mesmo *host*. A Figura 5.3 mostra como a plataforma heterogênea configura-se com as diferentes unidades de processamento.

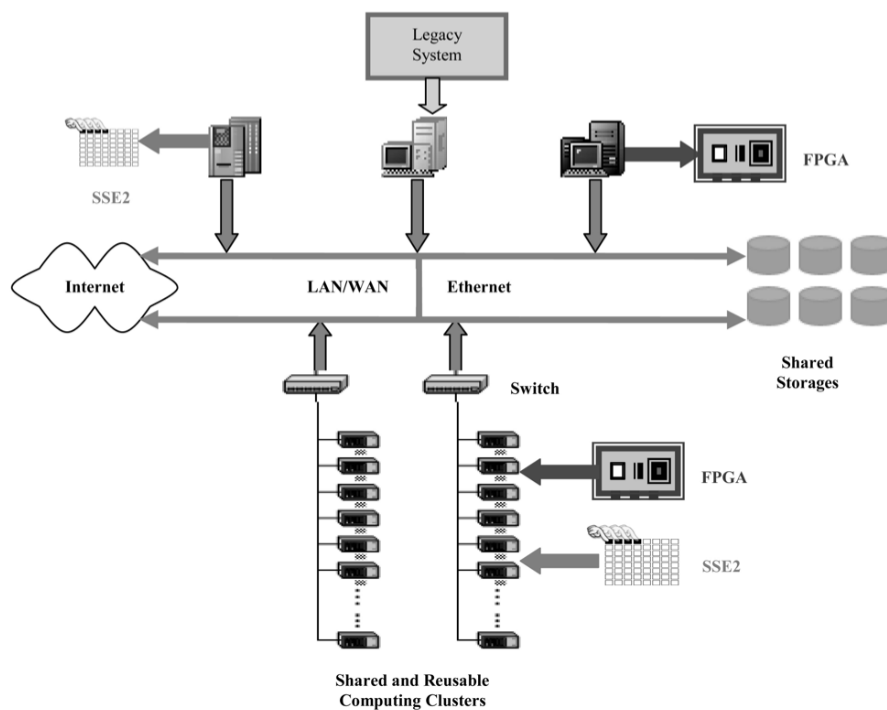


Figura 5.3: Plataforma heterogênea incluindo processadores legado, com SSE e FPGAs [45].

Para lidar com a diferença considerável de desempenho entre as diferentes unidades de processamento, os autores utilizaram paralelismo em vários níveis, balanceamento flexível e diversas otimizações.

A plataforma implementada incorpora paralelismo de granularidade fina e grossa nas diferentes unidades de processamento, quando aplicável. Num nível mais alto, uma tarefa de comparação de uma sequência de busca com um banco de dados genômico é feita de maneira paralela, dividindo-se a mesma em diversas tarefas que comparam a sequência de busca com um subconjunto do banco de dados genômico (granularidade grossa - Seção 3.4.4).

Nesse nível, um banco de dados extenso é dividido em fragmentos de tamanhos idênticos. Cada unidade de processamento recebe, durante o processamento, um número de fragmentos dependendo da sua capacidade de processamento. No nível mais baixo, técnicas de paralelismo foram utilizadas obter ganhos de desempenho em cada unidade de processamento (granularidade fina - Seção 3.4.4). Nesse nível, instruções SSE e coprocessadores FPGA são capazes de obter paralelismo a nível de instruções.

Para tratar os diversos níveis de heterogeneidade que podem ser encontrados em uma plataforma como essa, os autores utilizaram uma estratégia de escalonamento e um mecanismo de busca de sequências feito paralelamente ao processamento.

Devido ao extenso desbalanceamento entre as unidades de processamento, a busca no banco de dados foi conduzida de forma a utilizar eficientemente os recursos disponíveis. Para obter essa eficiência, os autores implementaram um mecanismo de balanceamento de carga baseado no desempenho de cada unidade de processamento. O mestre mantém um arquivo de configuração, responsável por descrever as configurações de cada escravo e seus recursos disponíveis. Dessa forma, o escalonador no mestre realiza o balanceamento baseado no conjunto de escravos escolhido pelo usuário e distribui os fragmentos de banco de dados a todos os escravos disponíveis.

A plataforma utilizada nos testes foi um *cluster* Sunfire x2100 executando o sistema operacional GNU/Linux. O *cluster* consiste em um nó mestre com um AMD Opteron 275 com dois núcleos e 10 escravos com conexão Gigabit Ethernet entre os *hosts*. Cada escravo possui um AMD Opteron 175 com dois núcleos e 2GB de memória RAM. O dispositivo FPGA utilizada foi a ADP-WRC-II baseada no barramento PCI com a placa Xilinx Virtex II XC2V6000 e 1GB de memória SDRAM.

Comparando sequências de busca de diversos tamanhos com o banco de dados month.aa [50], o desempenho máximo, com speedup de aproximadamente 60×, foi obtido com a sequência de 1223 resíduos utilizando um dispositivo FPGA, em comparação com um processador Intel sem as instruções SSE. Nos testes, é possível perceber que há uma queda de desempenho no FPGA quando a sequência de busca é maior que 1420 resíduos. Nesse caso, o código precisa segmentá-la e com isso há um aumento no overhead de transmissão e uma diminuição no desempenho. No entanto, o speedup ainda é de aproximadamente 54× sobre a mesma comparação utilizando o processador sem instruções SSE.

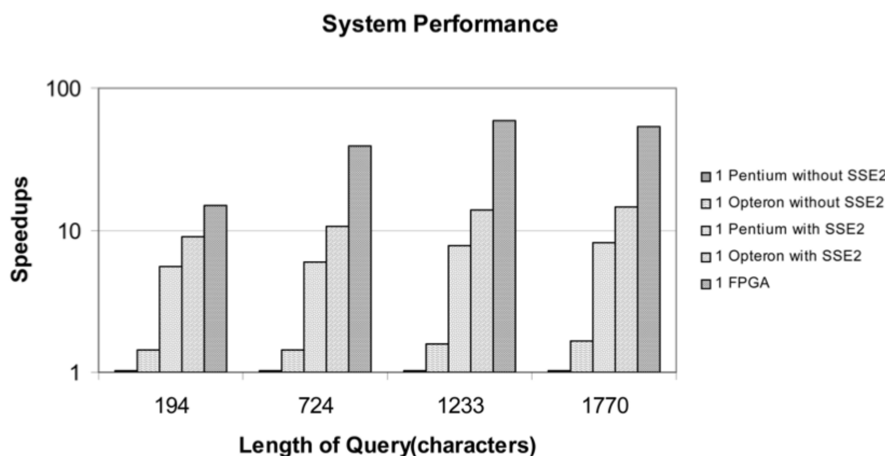


Figura 5.4: Speedups obtidos com sequências de busca de diversos tamanhos [45].

5.6 Abordagem de Rognes 2011 [61]

Rognes propõe a ferramenta SWIPE [61], que implementa uma versão do algoritmo SW (Seção 3.4.2) que compara dezesseis diferentes sequências do banco de dados com a mesma sequência de busca, a cada vez.

O algoritmo foi implementado para processadores Intel utilizando as instruções SSE3 com paralelização sobre múltiplas sequências do banco de dados, como pode ser visto na Figura 5.5, que mostra as diferentes abordagens para paralelizar o algoritmo SW. Para simplificar a figura, vetores de quatro elementos foram mostrados, quando o normal seria utilizar 8 ou 16 elementos: vetores posicionados na anti-diagonal, como descrito por Wozniak *et. al* [74] (Figura 5.5(a)), na sequência de busca, como utilizado por Rognes e Seeberg [62] (Figura 5.5(b)), na sequência de busca mas particionados, como implementado por Farrar [21] (Figura 5.5(c)) e na sequência do banco de dados comparando sequências em paralelo, como feito por Rognes nesse artigo (Figura 5.5(d)).

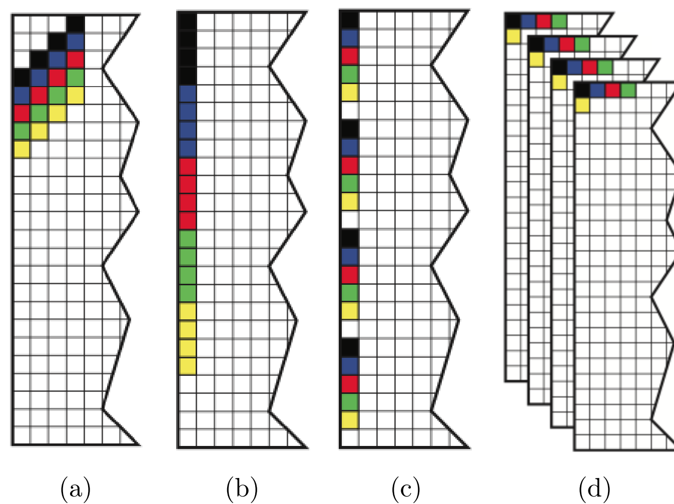


Figura 5.5: Diferentes abordagens consideradas para paralelizar o algoritmo SW em núcleos SSE [61].

A principal vantagem da paralelização intra-tarefa, na qual múltiplas sequências do banco de dados são comparados em paralelo com uma sequência de busca, é que ela simplesmente evita qualquer dependência de dados entre os elementos da matriz. Essa abordagem computa quatro células consecutivas ao longo da sequência do banco de dados antes de passar para a próxima célula da sequência de busca para reduzir o número de acessos à memória.

Resíduos de dezesseis diferentes sequências do banco de dados são comparadas em paralelo. As operações são realizadas utilizando vetores SSE de dezesseis elementos, cada um com 1 byte de tamanho. Quando a primeira dessas dezesseis sequências termina, o primeiro resíduo da próxima sequência do banco de dados é carregado no vetor. As sequências do banco de dados são lidas e carregadas na ordem em que aparecem no banco de dados.

Para tornar o processamento mais rápido, Rognes implementou a criação do perfil de escore a partir da matriz de substituição e das sequências do banco de dados. Como o

perfil é criado para os próximos quatro resíduos de cada uma das dezesseis sequências sendo comparadas, ele deve ser reconstruído a cada quatro execuções do laço interno.

O desempenho da implementação foi testado com diferentes matrizes de substituição, penalidades de gap, sequências de busca e número de *threads*. A velocidade obtida manteve-se acima de 100 GCUPS na plataforma utilizada, que inclui o processador Intel Xeon X5650 com seis núcleos. Foram testadas também as ferramentas SWPS3 [70], Striped [21] e duas versões do BLAST [2]. Foi utilizado o banco de dados UniProt-KB 11 [17], que inclui os bancos de dados Swiss-Prot 53 e TrEMBL 36, de maio de 2007. As matrizes de substituição utilizadas foram as BLOSUM45, BLOSUM50, BLOSUM62, BLOSUM80, BLOSUM90, PAM30, PAM70 e PAM250, todas obtidas da página do NCBI [49].

A Figura 5.6 indica o desempenho obtido pelas ferramentas utilizadas nos testes com diferentes números de *threads*. Para esse teste, foi utilizada a sequência de busca com 375 resíduos e a matriz de substituição BLOSUM62. Foram utilizadas as penalidades de abertura e extensão de gap de 11 e 1, respectivamente. A ferramenta SWIPE executa a 9,1 GCUPS em uma única *thread* e chega a 106,2 GCUPS executando em 19 *threads*, embora não exista praticamente nenhum ganho a partir de 12 *threads*.

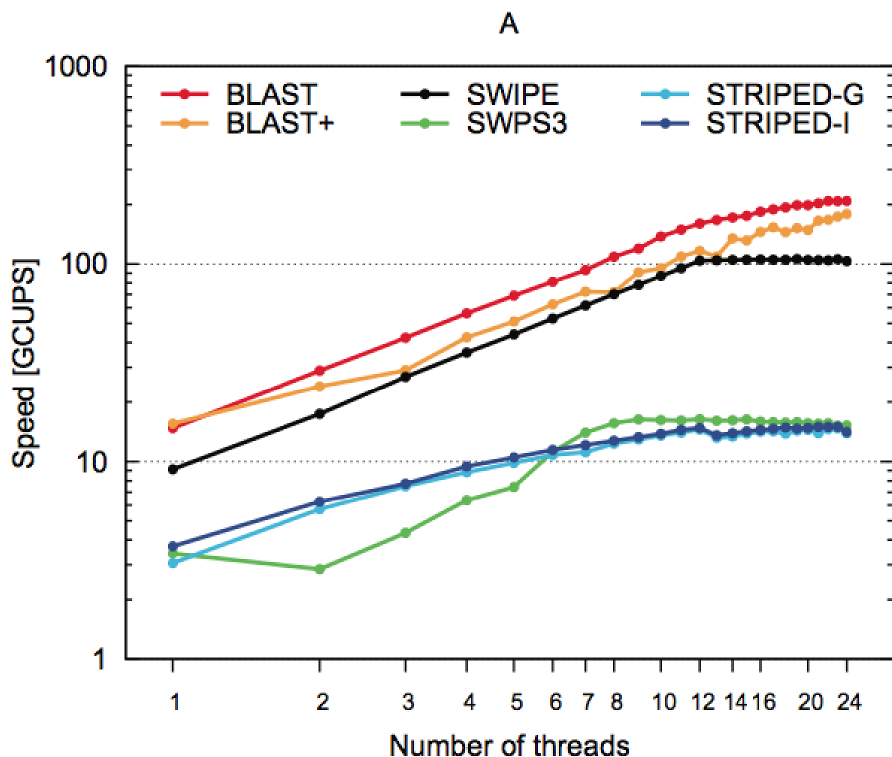


Figura 5.6: Desempenho obtidos com BLAST, SWIPE e Striped para núcleos SSE [61].

5.7 Abordagem de Ino *et. al* (2009) [32]

Ino *et. al* [32] implementaram uma versão modificada do algoritmo criado por Liu [42] e baseado em OpenGL [41], utilizando o paradigma mestre-escravo para acelerá-lo. Essa implementação foi integrada a um sistema de *grid* baseado em descansos de tela

que detectam recursos ociosos sobre os quais o alinhamento pode ser executado. O *grid* utilizado, baseado em GPUs, possui basicamente a mesma estrutura que sistemas de *grid* existentes, com a exceção de ser constituído essencialmente por GPUs.

O sistema possui três principais componentes: os recursos disponíveis no *grid*, um servidor de gerenciamento de recursos e os clientes. Os recursos do *grid* são os *hosts* disponíveis conectados à rede. Esses recursos não são utilizados exclusivamente pelo *grid*, ou seja, são utilizados pelos usuários e, quando detectado que estão ociosos, pelo servidor do *grid*.

O sistema baseado em um protetor de tela é executado em todos os recursos registrados para detectar o estado ocioso. Foi utilizado, inicialmente, um período de cinco minutos de ociosidade para ativar o processamento no *grid*. A partir da ativação, o sistema verifica o estado da GPU e da memória para confirmar que ambos (GPU e CPU) estão ociosos.

A implementação em GPU é paralelizada utilizando o paradigma mestre-escravo. Dado um *job*, que consiste de N sequências de busca e M sequências de banco de dados, o job é decomposto em N tarefas, sendo que cada uma corresponde à comparação de uma sequência de busca com todo o banco de dados (granularidade muito grossa - Seção 3.4.4). Como não há dependência de dados entre as tarefas, cada uma é independentemente designada a um recurso ocioso disponível. Portanto, essa implementação pode ser considerada uma aplicação do tipo *parameter-sweep* (Seção 4.4.1), que executa em sistemas de *grid*.

As comparações foram feitas utilizando o banco de dados Swiss-Prot 51 [17]. Para comparar a implementação, foi utilizada a implementação SSEARCH [48], que executa em CPU. A Tabela 5.3 mostra os equipamentos utilizados nos testes.

Recurso	1	2	3	4
CPU	Pentium 4	Xeon	Pentium 4	Pentium 4
Clock (GHz)	3,4	2,8	2,8	2,8
RAM (MB)	2	4	2	1
GPU	8800 GTX	8800 GTX	8800 GTX	8800 GTX
VRAM (MB)	768	768	768	768
GFLOPS	345	345	345	345
Recurso	5	6	7	8
CPU	Xeon	Core 2 Duo	Pendium D	Core Duo
Clock (GHz)	3,8	2,4	3	2
Ram (MB)	4	2	2	2
GPU	8800 GTS	7950 GX2	7900 GTX	7900 GTX
VRAM (MB)	640	512x2	512	512
GFLOPS	230	64x2	124	96

Tabela 5.3: Equipamentos utilizados nos testes de Ino *et. al* (2009) [32].

Para executar as comparações no ambiente não compartilhado foram necessárias cinco horas, aproximadamente, para comparar o banco de dados em um único processador. O tempo de execução foi reduzido para 11 minutos utilizando as 8 GPUs disponíveis. Por outro lado, uma única GPU dedicada leva aproximadamente uma hora para executar a mesma comparação.

Para o teste com ambiente compartilhado, foram utilizados os recursos 1, 5, 6 e 7 por cinco dias consecutivos. Esses recursos foram utilizados por alunos do laboratório durante aproximadamente oito horas por dia durante o período. Utilizando esses quatro recursos, foram registradas, em média, 202 comparações por dia. O mesmo número de comparações pode ser feito em aproximadamente 16 horas de execução em uma CPU dedicada, ou seja, uma única GPU não dedicada nessas condições equivale a dois processadores dedicados.

5.8 Abordagem de Jacek *et. al* (2012) [36]

A abordagem de Jacek *et. al* [36] utiliza o algoritmo SW (Seção 3.4.2) para resolver o problema de alinhamento múltiplo de sequências (MSA). Para isso, são comparadas todas as sequências do conjunto de entrada entre si. Além disso, a solução proposta utiliza uma variação do algoritmo Smith-Waterman que usa Myers-Miller [47] para recuperar o alinhamento.

Para que múltiplas GPUs fossem utilizadas de uma forma eficiente, levando em consideração o desempenho de cada placa individualmente, os autores implementaram um gerenciador de tarefas. O objetivo do gerenciador é balancear o trabalho entre as GPUs disponíveis. Primeiro, as tarefas são ordenadas em ordem descendente de tamanho. Depois, as tarefas são designadas de forma consecutiva a qualquer GPU que esteja ociosa, usando a estratégia *Self-Scheduling* (Seção 4.6.2).

Para testar a abordagem, os autores escolheram o banco de dados Ensembl 55 [18]. Os testes foram divididos entre seis grupos com diferentes tamanhos médios: 51, 154, 257, 459, 608 e 1103 resíduos. Foi utilizada a matriz de substituição BLOSUM50 [31], além de penalidades de gap de abertura 10 e extensão 2. A plataforma utilizada incluía o processador Intel Xeon E5405 de 2,0 GHz e duas GPUs Nvidia Tesla S1070 com 16GB de memória RAM.

A Tabela 5.4 apresenta os resultados dos testes, o número de sequências e o tamanho médio das sequências em cada caso. Os tempos de execução são os tempos médios para o algoritmo SW aplicado aos diferentes conjuntos de sequências. A implementação do Farrar [22] apenas calcula os escores, enquanto a abordagem baseada em GPU de Blazewicz *et. al* calcula os escores e os alinhamentos.

A abordagem foi comparada com o algoritmo de Farrar e executada em 1 e 4 GPUs, conforme mostrado na Tabela 5.4. Como pode ser visto, para um número grande de sequências, de tamanho médio maior que 459 resíduos, o algoritmo de Farrar (núcleos SSE) apresenta melhor desempenho. Comparando o desempenho de 1 GPU com 4 GPUs, observa-se que um speedup próximo do linear é obtido para todos os casos.

5.9 Abordagem de Ino *et. al* (2012) [33]

Em seu trabalho, Ino *et. al* [33] propuseram um sistema de compartilhamento de granularidade fina capaz de explorar o poder de processamento ocioso de GPUs em ambientes de rede local. O sistema aproveita pequenos períodos de ociosidade nas GPUs executando comparações com duração de aproximadamente um segundo cada. Para detectar esses períodos de ociosidade, são utilizados os eventos de mouse e teclado, diferentemente do trabalho anterior (Seção 5.7), no qual comparações são executadas quando um mecanismo

Tamanho médio das sequências	Número de sequências	CPU, Farrar (GCUPS)	1 GPU (GCUPS)	4 GPUs (GCUPS)
51	4000	0,863	2,581	10,055
	8000	0,875	2,676	10,597
	12000	0,888	2,711	10,740
154	2000	1,677	2,647	10,296
	4000	1,693	2,801	10,980
	6000	2,177	2,835	11,173
459	2000	2,824	2,709	10,679
	4000	2,834	2,837	11,274
	6000	2,831	2,846	11,370
608	800	2,842	2,358	9,224
	1200	2,871	2,493	9,720
	1600	2,885	2,469	9,770
1103	800	3,154	2,366	9,293
	1200	3,151	2,442	9,752
	1600	3,144	2,478	9,819

Tabela 5.4: Valores médios em GCUPS para o algoritmo SW em núcleos SSE, 1 GPU e 4 GPUs [36].

de proteção de tela é acionado. Além disso, o sistema implementado por Ino *et. al* divide as tarefas a serem realizadas em pequenos pedaços de acordo com o desempenho necessário para realizar a comparação de cada pedaço.

A implementação baseada em CUDA inclui o esquema de paralelização utilizado por Liu [40], no qual é utilizada a biblioteca gráfica OpenGL. Esse esquema de paralelização executa o código com um *pipeline* típico de aplicações gráficas.

O mestre é responsável por gerenciar todos os recursos registrados. Ele conhece detalhes acerca de cada recurso, como desempenho teórico, memória disponível e níveis de utilização. Além disso, o mestre recebe os jobs dos usuários e os agrupa em uma fila. O escalonador é responsável por decompor os jobs em tarefas independentes e alocar as tarefas aos recursos ociosos de forma sequencial.

O sistema busca por períodos de ociosidade de acordo com a seguinte definição: um escravo é considerado ocioso se a CPU e a GPU estão ambos ociosos. Embora o sistema seja executado em GPU, é importante que a CPU também esteja ociosa para que consiga realizar as operações de entrada e saída adequadamente. A CPU é considerado ociosa quando seu nível de utilização encontra-se abaixo de uma porcentagem σ, definida pelo dono do recurso. A GPU é considerada ocupada se uma de duas situações ocorrerem: um *kernel* do usuário está sendo executado ou o buffer gráfico está sendo atualizado como consequência de eventos de teclado ou mouse.

Para os testes, todos os escravos possuíam uma GPU Nvidia GTX 280. O sistema que baseia-se num protetor de tela é ativado após 5 minutos de tempo ocioso. Já o sistema de *cluster* é considerado um sistema dedicado. A Figura 5.7 mostra o número de tarefas bem sucedidas e com falha em cada sistema, bem como o desempenho observado. O sistema implementado nesse trabalho contou 1011 horas de tempo ocioso, o que é aproximadamente 2,1 vezes maior que o número de horas ociosas observadas pelo sistema

antigo, 479 horas. Essa melhoria na detecção do tempo ocioso levou a um desempenho de cerca de duas vezes melhor, indo de 31,7 GCUPS para 64 GCUPS utilizando o mesmo equipamento.

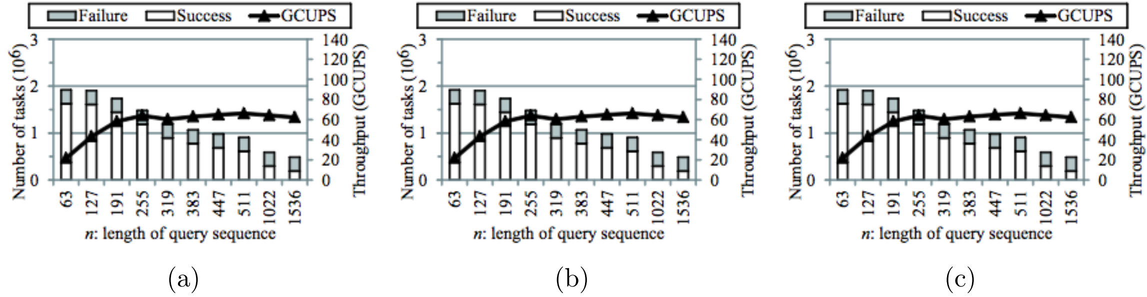


Figura 5.7: Comparação de desempenho entre o sistema implementado por Ino *et. al*, um sistema baseado em protetor de tela [32] e um *cluster* [33].

5.10 Quadro comparativo

Ref.	Plataforma	Alocação	Dedicado	Reatribuição	GCUPS
[6]	CPU e GPU	Fixa	Sim	Não	9,6 1 GPU, 1 SSE
[67]	CPU e GPU	Fixa	Sim	Não	27 1 GPU, 4 SSE
[11]	GPU	SS	Sim	Não	N/D N/D
[45]	CPU e FPGA	Fixa	Sim	Não	11,3 1 FPGA, 20 SSE
[66]	CPU e GPU	SS	Não	Não	4,4 10 GPUs
[61]	CPU	SS	Sim	Não	106 12 SSE
[32]	GPU	SS	Não	Não	3,09 8 GPUs
[36]	GPU	SS	Sim	Não	11 4 GPUs
[33]	GPU	SS	Não	Sim	64 8 GPUs

Tabela 5.5: Quadro comparativo entre as 9 abordagens.

A Tabela 5.5 apresenta uma comparação entre as abordagens discutidas nessa seção. As abordagens apresentadas nessa tabela são capazes de explorar múltiplos núcleos SSE [61], múltiplas GPUs [11, 32, 33, 36], núcleos SSE e GPUs [6, 67] ou núcleos SSE e FPGA [45].

Com relação à alocação de tarefas, a maior parte das abordagens utiliza uma política baseada na SS (*Self-Scheduling*) [11, 32, 33, 36, 61, 66], na qual as unidades de processamento ociosas recebem apenas uma tarefa. Algumas abordagens aplicam pesos fixos à política de alocação, baseando-se no desempenho teórico das UPs [6, 45] ou em testes de desempenho realizados antes da execução [67].

A maioria das estratégias analisadas executa-se em ambiente dedicado, com exceção de [36, 61, 66]. Nessas três estratégias para ambientes não-dedicados, a execução do Smith-Waterman possui prioridade baixa. Por essa razão, somente são realizadas comparações quando nenhuma tarefa local está sendo executada.

Das abordagens analisadas nesse capítulo, apenas uma permite a reatribuição de tarefas [33]. No entanto, essa atribuição ocorre quando um dos escravos é impedido de terminar a comparação. Nesse caso, a tarefa é retornada para a fila e é alocada a um dos escravos que estiver ocioso.

Dentre as abordagens analisadas, a que obtém o melhor desempenho atinge 106 GCUPS com 12 núcleos SSE. Os dados apresentados na sexta coluna da Tabela 5.5 não podem ser utilizados para comparação direta pelas seguintes razões. Primeiramente, algumas abordagens calculam somente o escore enquanto outras calculam também o alinhamento. Em segundo lugar, diferentes placas e processadores são usados nas abordagens, bem como diferentes bancos de dados genômicos e diferentes sequências de busca. No entanto, os valores apresentados nessa coluna permitem que se forme uma idéia do potencial de cada abordagem.

A nosso conhecimento, não há abordagens para ambiente não dedicado que não use a política *Self-Scheduling*. Além disso, não encontramos na literatura abordagens que usem a redistribuição de tarefas para reduzir o tempo de execução da aplicação como um todo.

Capítulo 6

Projeto da Estratégia com Re-Trabalho para Plataformas Híbridas

6.1 Considerações iniciais

Na literatura, podemos encontrar diversos artigos que descrevem métodos e ferramentas que implementam comparação de bancos de dados de sequências biológicas (Capítulo 5). No entanto, essas ferramentas foram projetadas para utilizar um número limitado de unidades de processamento [67] e, na maior parte dos casos, de apenas um tipo [22, 42].

O objetivo dessa dissertação é propor e implementar uma ferramenta capaz de comparar bancos de dados de sequências biológicas utilizando diversas unidades de processamento, de diferentes arquiteturas, integradas utilizando políticas de alocação de tarefas e balanceamento de carga para diminuir o tempo de processamento. A nossa ferramenta inclui suporte à alocação dinâmica, o que é importante dada a natureza heterogênea das unidades de processamento suportadas. Alocando dinamicamente as tarefas, unidades de processamento mais rápidas recebem mais tarefas, na proporção da sua velocidade, o que contribui para um menor tempo de processamento total. Além disso, propomos um mecanismo de re-trabalho, que permite a reatribuição de tarefas a nodos ociosos para tratar situações onde nodos muito lentos recebem algumas das últimas tarefas. Finalmente, a ferramenta foi projetada tendo em mente novos tipos de unidades de processamento, políticas de alocação e tipos de tarefas que podem vir a ser suportadas no futuro.

O algoritmo utilizado nas nossas comparações é o Smith-Waterman com *affine-gap* (Seção 3.4.2). Foram escolhidas para a implementação a plataforma da Intel com suporte às extensões SSE (Seção 6.1) e a arquitetura CUDA, da Nvidia (Seção 2.2.2). Além disso, utilizamos o modelo mestre-escravo para que a execução possa ser realizada por um número de diferentes unidades de processamento e controlada por um processo mestre. Alguns dos requisitos definidos são:

- **Sempre que possível, usar programas já existentes na literatura para comparação de sequências em plataformas específicas:** quando existir um programa já desenvolvido que oferece o melhor desempenho para determinada arquitetura, iremos adaptá-lo para execução na nossa ferramenta, ao invés de criar um novo programa.

- **Tempo baixo de execução:** a implementação da nossa ferramenta deve processar na ordem de bilhões de células atualizadas por segundo. A medição do desempenho dos escravos em GCUPS facilita a comparação com outras implementações, disponíveis na literatura.

- **Limite alto no tamanho das sequências:** esse limite existe devido à capacidade de memória global das GPUs. Em alguns casos, as memórias mais rápidas das GPUs podem ser usadas, o que impõe um limite ainda mais restrito ao tamanho das sequências. Por exemplo, no caso da ferramenta CUDASW++ 2.0 [42], o tamanho das sequências é limitado à quantidade de memória global disponível na GPU.

- **Valor máximo do escore de similaridade em $64k$:** dependendo da arquitetura utilizada para processamento vetorial, o valor máximo do escore é limitado ao tamanho dos elementos utilizados nos vetores. No caso da arquitetura Intel SSE , se utilizamos elementos de 8 ou 16 bits com *bias*, o valor máximo do escore fica restrito a 256 ou 65534, respectivamente. No caso da execução em GPUs, a ferramenta CUDASW++ utiliza elementos de 32 bits no formato *packed data* para o escore.

6.2 Visão geral

A organização do sistema proposto segue o modelo mestre-escravo, tendo o mestre a função de receber requisições do usuário, controlar a execução das tarefas e apresentar-lhe os resultados. Os escravos recebem do mestre tarefas a serem executadas, executam-nas e retornam os resultados. A Figura 6.1 apresenta a arquitetura do sistema.

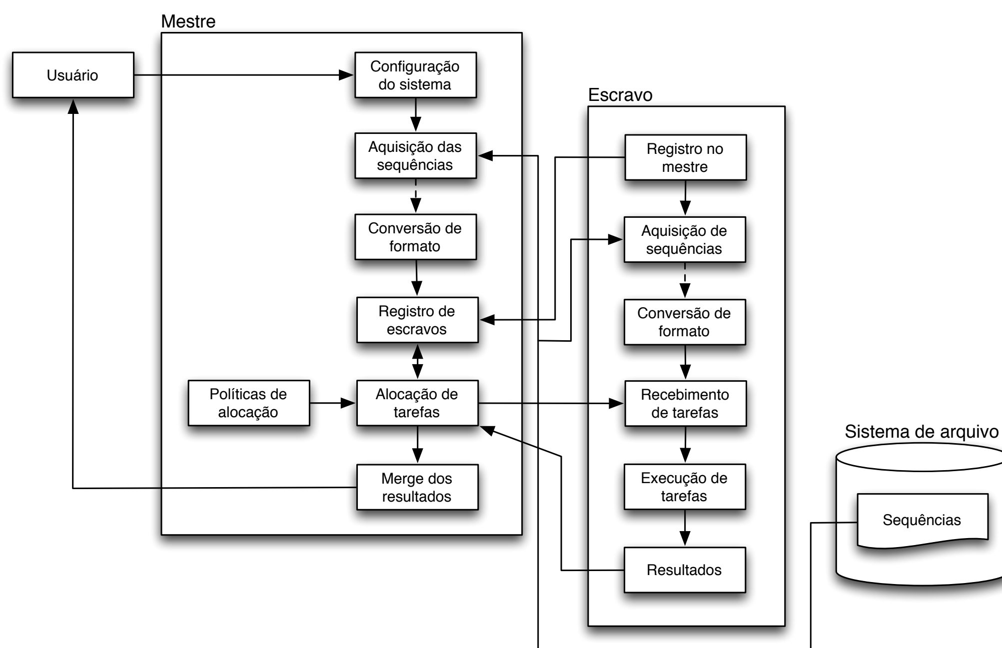


Figura 6.1: Arquitetura do Sistema Proposto.

Primeiro, é iniciado, em um determinado nó, o processo mestre e, para cada unidade de processamento disponível (incluindo as presentes no nó do processo mestre), um pro-

cesso escravo é criado. Baseando-se nos parâmetros de entrada fornecidos pelo usuário, o processo mestre lê as sequências de busca e o banco de dados do sistema de arquivos e, se necessário, as converte para o formato indexado. Depois, o processo mestre aguarda as conexões dos processos escravos. Quando um processo escravo se conecta ao processo mestre, ele é identificado e recebe a primeira alocação de tarefas. Essa primeira alocação compreende apenas uma tarefa, para que o processo mestre teste o desempenho do processo escravo sem descartar o tempo de processamento. À medida que os processos escravos terminam de executar as tarefas designadas, novas alocações são feitas, levando em consideração o desempenho atual de cada um, utilizando uma média ponderada móvel. Ao terminar o processamento de todas as tarefas, o processo mestre desconecta-se dos processos escravos e mostra ao usuário os resultados, na forma de uma lista de comparações e escores.

A Figura 6.2 mostra como os processos mestre e escravos se organizam. Para aproveitar as unidades de processamento disponíveis no nó mestre, processos escravos podem ser iniciados para utilizar as unidades de processamento ociosas. Além disso, cada um dos nós escravos pode iniciar qualquer número de processos escravos, um para cada unidade de processamento.

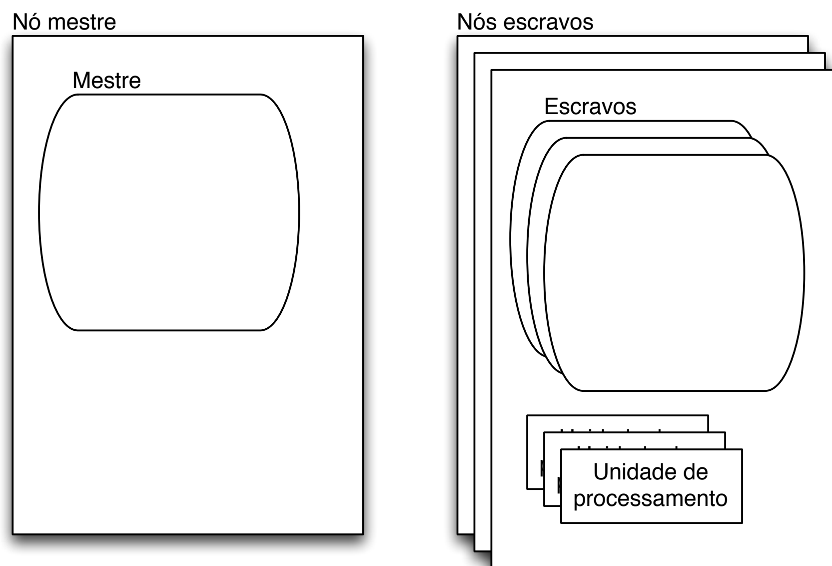


Figura 6.2: Organização dos processos mestre e escravos.

A seguir, serão detalhados os principais módulos da arquitetura.

6.3 Módulo de aquisição das sequências

Para que o processo mestre consiga ler as sequências de busca e banco de dados, é necessário que elas estejam disponíveis localmente. Para isso, podem ser feitas cópias manuais dos arquivos ou pode ser utilizado um sistema de arquivos distribuído, como NFS [53].

Tanto o processo mestre como os processos escravos recebem, no início da execução, parâmetros de entrada. Dentre os parâmetros de entrada estão os arquivos contendo as sequências de busca e banco de dados. Dessa forma, cada um dos nós participantes pode decidir como vai disponibilizar aos escravos os arquivos de sequências.

6.4 Módulo de conversão de formato

Os bancos de dados genômicos são na realidade arquivos "flat" no formato FASTA (Seção 3.2.2), nos quais o final de uma sequência é colado no início da próxima. Sendo assim, o acesso às sequências que se encontram no meio do arquivo é demorado.

No nosso modelo mestre-escravo, é comum um determinado escravo receber do mestre uma alocação na qual apenas um subconjunto das sequências de busca deve ser comparada com as sequências do banco de dados. No pior caso, a última sequência de busca deve ser comparada e o escravo pode, para isso, ter que ler todas as sequências anteriores até chegar na última. Com o objetivo de acelerar o acesso direto às sequências, propusemos o formato indexado [30] para manipulação das mesmas. Tanto o mestre quanto os escravos podem utilizar esse módulo. A ferramenta utiliza o formato indexado, que permite otimizações na leitura. Caso os arquivos de sequências não estejam disponíveis no formato indexado, ou seja, são passados nos parâmetros arquivos de sequências no formato FASTA, o processo, seja ele mestre ou escravo, converte o arquivo para o formato indexado e salva no disco. Caso o mestre receba como parâmetro um arquivo no formato FASTA e o sistema de arquivos utilizado seja distribuído, essa conversão pode ser aproveitada pelos outros escravos que por acaso tenham acesso ao mesmo sistema de arquivos. Caso os arquivos sejam distribuídos manualmente, cada um dos escravos precisará converter os arquivos individualmente. A Tabela 6.1 apresenta o formato indexado.

Para melhorar esse processo de leitura, criamos um formato simples com alguns campos adicionais. Utilizando esse formato, os escravos conseguem ler sequências em qualquer posição do arquivo com apenas duas operações de leitura. Além disso, a alocação de memória é simplificada pelo fato de o arquivo manter o tamanho das sequências armazenado.

Número de sequências
Tamanho da maior sequência
Posições iniciais das sequências 0 a $n-1$
Tamanho do nome da sequência 0
Nome da sequência 0
Tamanho da sequência 0
Sequência 0
⋮

Tabela 6.1: Campos que fazem parte do formato indexado. Cada linha contém quatro bytes e os campos dos nomes e sequências possuem tamanho variável.

No formato indexado, o primeiro campo é o número de sequências contidas no arquivo (4 bytes). O segundo campo contém o tamanho da maior sequências (4 bytes). Esse campo é necessário porque a alocação de espaço para as sequências baseia-se nesse valor.

Após isso, existem n campos, que indicam as posições iniciais das n sequências que compõem o arquivo (4 bytes cada). Esses campos implementam a indexação.

Para cada sequência, existem quatro campos: tamanho do nome da sequência (4 bytes), nome da sequência (tamanho variável), tamanho da sequência (4 bytes) e a sequência propriamente dita (tamanho variável).

Junto com a ferramenta, foi implementado um programa cuja única função é converter um arquivo no formato FASTA para o formato indexado. Dessa forma, os arquivos de busca e banco de dados podem ser preparados antes da execução e, talvez, distribuídos já no formato indexado entre os escravos.

6.5 Módulo de alocação de tarefas

O módulo de alocação de tarefas é responsável por determinar quais comparações cada escravo deve executar. As tarefas são agrupadas em blocos de tarefas de tamanho variável, proporcional ao poder atual de processamento de cada unidade de processamento (UP). Sendo assim, por exemplo, a UP1 (mais rápida) pode receber quatro tarefas em uma alocação, enquanto a UP2 (mais lenta) pode receber somente uma tarefa.

Na nossa abordagem, as comparações possuem granularidade muito grossa (Seção 3.4.4) no nível mais alto e granularidade fina na execução em cada UP. O desempenho de cada escravo, em GCUPS, é enviado juntamente com os resultados ao mestre e utilizado para manter uma média ponderada móvel calculada com a política PSS (Seção 4.6.4). Dessa forma, o módulo de alocação de tarefas pode se adaptar às flutuações de desempenho dos escravos.

Mesmo que o desempenho dos escravos não flutue com o tempo (o que normalmente ocorre), a média ponderada móvel é útil porque a quantidade de comparações alocadas aos escravos tenderá ao desempenho desses escravos durante a execução.

Além do desempenho de cada um dos escravos em GCUPS, o módulo de alocação de tarefas utiliza uma quantidade de tempo que cada escravo deve levar para executar todas as comparações alocadas. Atualmente esse valor é determinado arbitrariamente.

6.5.1 Mecanismo de re-trabalho

Quando executamos os primeiros testes em plataformas híbridas compostas por núcleos SSE e GPUs disponíveis no laboratório LAICO (Laboratório de sistemas Integrados e Concorrentes) da Universidade de Brasília, percebemos a existência de um gargalo no desempenho dos núcleos SSE. O gargalo é causado pela grande diferença entre o desempenho das CPUs e das GPUs. Nos casos em que as unidades de processamento mais rápidas terminam as tarefas designadas a elas e as unidades mais lentas ainda estão processando, aquelas ficam ociosas, até que as unidades de processamento mais lentas terminem e o processo mestre finalize a execução.

Para melhorar o desempenho em casos como esse, criamos um mecanismo de re-trabalho, utilizado apenas nos blocos de tarefas finais. Mais especificamente, o mecanismo

é utilizado a partir do momento que uma ou mais unidades de processamento terminaram de executar as tarefas designadas mas nem todos os resultados foram recebidos pelo mestre. Ou seja, todos os blocos de tarefas foram enviados às unidades de processamento mas alguns ainda não tiveram seus resultados recebidos. Nesses casos, o mestre passa a reenviar blocos de tarefas às unidades de processamento ociosas. Vale salientar que apenas os blocos de tarefas ainda sem resultados recebidos pelo mestre podem ser reenviados. Quando esses blocos de tarefas tiverem seus resultados recebidos a execução é finalizada.

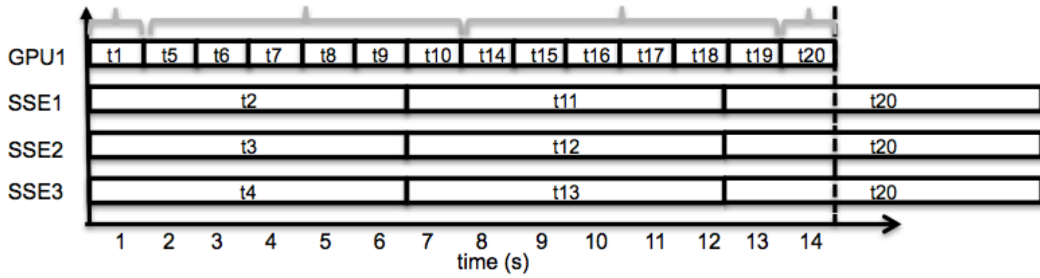
O ganho dessa abordagem pode ser ilustrado pelo exemplo a seguir, nas Figuras 6.3(a) e 6.3(b). Para esse exemplo, considere 1 GPU e 3 núcleos SSE. Na Figura 6.3(a), o mecanismo de re-trabalho é utilizado. No início da execução, cada uma das unidades de processamento recebe uma tarefa, que corresponde à comparação de uma sequência com o banco de dados. Quando a UP GPU1 termina a sua primeira tarefa, ela entra em contato com o mestre e entrega o resultado. Nesse momento, o mestre atualiza, em memória, o seu desempenho médio e envia à GPU1 seis tarefas ($t5$ a $t10$). As demais UPs SSE1, SSE2 e SSE3 terminam de processar suas tarefas no tempo $6s$ e também enviam os resultados ao mestre, que atualiza seus desempenhos e atribui uma tarefa para cada ($t11$, $t12$ e $t13$, respectivamente). No tempo 7 a UP GPU1 termina de processar as tarefas $t5$ a $t10$ e recebe do mestre as tarefas $t14$ a $t19$. No tempo 12, as UPS SSE1, SSE2 e SSE3 terminam de executar suas tarefas e, como resta apenas uma tarefa não atribuída, a UP SSE1 a recebe ($t20$). No tempo 13, a UP GPU1 termina de processar a sua tarefa e todas as demais já foram atribuídas, embora nem todas tenham resultados produzidos. Nesse momento, o mecanismo de re-trabalho determina que a tarefa $t20$ é a primeira que foi atribuída mas cujos resultados ainda não foram recebidos e, por isso, a atribuí também à UP GPU1. No tempo 14, a UP GPU1 termina de processar a tarefa $t20$ e a execução é finalizada.

Caso o mecanismo de re-trabalho não seja utilizado, como pode ser visto na Figura 6.3(b), os primeiros 13 segundos da execução são idênticos. Quando a UP GPU1 termina de processar as tarefas $t14$ a $t19$ e a UP SSE1 está processando a tarefa $t20$, o mestre diz à UP GPU1 que não há mais tarefas disponíveis. Nesse caso, a UP GPU1 fica ociosa até que a UP SSE1 termine de processar a tarefa $t20$ e a execução seja finalizada, no tempo 18.

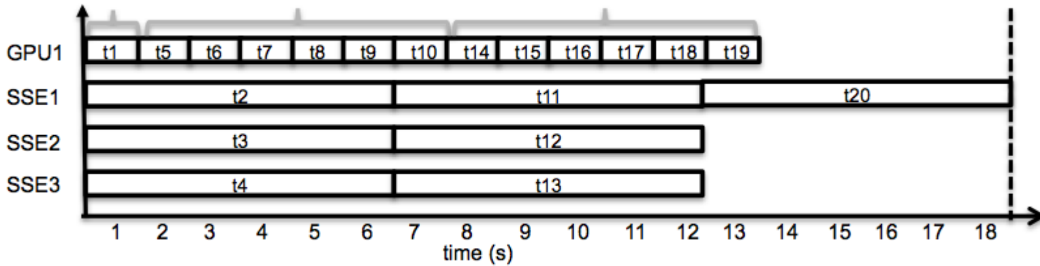
O mecanismo de re-trabalho é implementado com um vetor de tarefas e dois ponteiros, indicando a próxima tarefa a ser alocada (*start*) e a última tarefa terminada (*done*). No início, todas as tarefas estão no vetor com o valor 0 e os dois ponteiros (*start* e *done*) apontam para o início do vetor. À medida que as tarefas são alocadas, *start* é avançado. Da mesma maneira, *done* é avançado sempre que uma tarefa é terminada, ou seja, quando seu o resultado é recebido pelo mestre. O mecanismo de re-trabalho entra em atividade quando o ponteiro *start* está apontando para a última tarefa do vetor mais um e o ponteiro *done* está em alguma posição anterior. Quando os dois ponteiros estão na posição da última tarefa mais um, a execução é finalizada.

6.6 Execução em GPU

Para a execução em GPU, decidimos utilizar a ferramenta de código aberto CUDASW++ 2.0 [42], que é considerada a melhor ferramenta CUDA para comparação de proteínas com Smith-Waterman. Nessa versão, os autores melhoraram o algoritmo utili-



(a) Alocação de tarefas com o mecanismo de re-trabalho. O tempo total de execução é 14 segundos.



(b) Alocação de tarefas sem o mecanismo de re-trabalho. O tempo total de execução é 18 segundos.

Figura 6.3: Comparação entre a alocação de tarefas com e sem o mecanismo de re-trabalho.

zado com a adição do perfil de busca sequencial (Seção 6.6.1) da ferramenta CUDASW++ 2.0.

6.6.1 Perfil de busca

Para calcular os valores $D_{i,j}$ com o algoritmo SW (Seção 3.4.2), o escore $p(i, j)$ deve ser adicionado ao valor armazenado na posição $D_{i-1,j-1}$. Devido ao grande número de iterações que o algoritmo SW deve executar em uma busca de banco de dados, reduzir a quantidade de processamento necessário para calcular uma posição da tabela tem um impacto significativo no desempenho do algoritmo. Pensando nisso, Rognes [61] e, posteriormente, Farrar [22], implementaram o perfil de busca paralelo às sequências de busca para cada possível resíduo da tabela de substituição. O perfil de busca só precisa ser calculado uma vez, antes do início do cálculo da matriz de programação dinâmica, e o seu formato depende das estruturas de dados utilizadas.

Dada uma sequência de busca s de tamanho $|s|$ e um alfabeto Σ, o perfil de busca é definido como uma tabela $P_{|s|\times|\Sigma|}$. Cada linha da tabela, $P_r, r \in \Sigma$, contém os escores resultantes da comparação entre o resíduo r e todos os resíduos da sequência de busca:

$$P_r(i) = p(r, s[i]), 0 \leq i < |s| \quad (6.1)$$

No caso da ferramenta CUDASW++ 2.0 [42], o perfil de busca é armazenado na memória de textura da GPU. Como as operações sobre a memória de textura têm como saída elementos empacotados de cor com quatro componentes (RGBA), o perfil de busca

foi reorganizado para se adequar a esse formato. Dessa maneira, quatro escores são buscados em uma única instrução.

Como a nossa ferramenta utilizará o programa CUDASW++ 2.0, o perfil de busca também será utilizado.

6.6.2 Modificações no CUDASW++ 2.0

Foram necessárias apenas pequenas modificações no código, sendo a mais importante um controle sobre o laço que varia sobre as sequências de busca. Como a ferramenta CUDASW++ 2.0 é executada para comparar todas as sequências de busca com todas as sequências de banco de dados e a nossa ferramenta distribui as comparações entre os escravos, esse laço foi modificado para iterar apenas sobre as sequências de busca alocadas pelo mestre ao escravo em questão. Modificações menores foram feitas na forma como os parâmetros de entrada são recebidos e na leitura da matriz de substituição.

6.7 Execução em processadores Intel

Para a execução em processadores Intel utilizando as instruções SSE, fizemos a nossa própria implementação baseada no trabalho do Farrar [22], com algumas melhorias e modificações. Na nossa implementação, como na de Farrar, usamos o perfil de busca (Seção 6.7.1) e também o *bias*, explicado a seguir.

6.7.1 *Bias*

Em sua implementação, Farrar [21, 22] utiliza um valor subtraído do escore em algumas partes do código. Como resultado, os elementos de 8 e 16 bits, em alguns casos, podem armazenar valores maiores do que se o *bias* não fosse utilizado. Para os elementos de 8 bits com sinal, já que o escore utilizado pelo algoritmo SW é sempre maior ou igual a zero, um *bias* de -128 permite que os escores utilizados sejam de 0 a 255. O mesmo pode ser feito com os elementos de 16 bits.

Farrar utiliza esse mecanismo porque algumas das instruções SSE utilizadas por ele exigem que sejam utilizados inteiros sem sinal. Como os elementos do perfil de busca e da matriz de substituição são armazenados com sinal, Farrar converte esses elementos para o tipo sem sinal utilizando o *bias* e o utiliza no restante do código.

No nosso caso, o tipo inteiro com sinal é utilizado em todas as operações. O valor do *bias* é inserido como se fosse o zero, e as operações aritméticas e de comparação são feitas com base nisso. No final da comparação, o *bias* é retirado e o valor final do escore é obtido.

6.7.2 Implementação do algoritmo Farrar

O algoritmo Farrar [21] foi implementado conforme o descrito em [22] com algumas modificações. A modificação mais importante foi a utilização de instruções SSE mais novas, eliminando algumas limitações enfrentadas pela implementação do Farrar. Por exemplo, isso permitiu que o *bias* fosse utilizado de forma integral para aumentar o tamanho máximo do escore tanto com 8 quanto com 16 bits, o que diminui a frequência

com que a execução com 8 bits deve ser interrompida para que os elementos de 16 bits sejam utilizados.

6.8 Adição de novas políticas de alocação de tarefas

Para adicionar uma nova política de alocação de tarefas em nossa ferramenta, é necessário apenas que o programador crie uma nova classe que herde as propriedades da classe *sw_task*, como demonstrado no trecho de código abaixo.

```
class sw_task_ss : public sw_task {
public:
    sw_task_ss(sw_core_protein *query, sw_core_protein *database);
    virtual ~sw_task_ss();

    virtual int allocate(uint64_t cells_target, sw_net_event_u *event);
};
```

O construtor já é executado pela classe herdada (*sw_task*), mas pode ser estendido pela classe sendo implementada. Ele é responsável por inicializar os valores das variáveis, o que é normalmente necessário para todas as políticas de alocação de tarefas. O mesmo vale para o destrutor. Ele é responsável por liberar o espaço em memória utilizado pela classe e pode ser estendido, se for necessário.

O método *allocate* inclui a funcionalidade de selecionar um número de comparações que o escravo deve executar. Como parâmetros de entrada, ele recebe o número de células que o escravo deve computar e um ponteiro para o evento de rede que deve ser preenchido e depois transmitido para o escravo. Esse método não é implementado pela classe pai (*sw_task*) e deve obrigatoriamente ser implementado pela classe sendo implementada.

Como exemplo, o trecho de código abaixo ilustra a implementação da política de alocação mais simples, SS (Seção 4.6.2), por meio da classe *sw_task_ss*.

```
int sw_task_ss::allocate(uint64_t cells_target, sw_net_event_u *event) {
  uint64_t cells_temp = 0;

  event->job.start = event->job.end = _start;

  cells_temp += (uint64_t)_query->length(_start) * _database->total_size();
  event->job.cells = cells_temp;

  return EXIT_SUCCESS;
}

sw_task_ss::sw_task_ss(sw_core_protein *query, sw_core_protein *database) :
     sw_task(query, database) {
}

sw_task_ss::~sw_task_ss() {
}
```

Nessa implementação, faz-se $job.start = job.end$. Dessa maneira, cada escravo recebe somente uma tarefa por vez. Após isso, o número de céulas a serem alocadas ($job.cells$) é atualizado.

Capítulo 7

Resultados experimentais

7.1 Ambiente utilizado

Foi utilizado apenas um ambiente para compilação e execução dos testes, com dois *hosts* interconectados por uma rede Gigabit Ethernet. Cada *host* possuía duas GPUs Nvidia GTX 580 com 512 núcleos CUDA, clock 772MHz e 1,5GB de memória e um processador Intel Core i7 2600K, com clock 3,4GHz e 8GB de memória RAM. O código foi compilado com o ambiente de desenvolvimento CUDA 4.2.9 e o compilador GNU 4.5.2. O sistema operacional utilizado foi o Linux 3.0.0-15 Ubuntu 64 bits.

A matriz de substituições utilizada nas comparações foi a BLOSUM62 (Figura 3.1) e as penalidades de abertura e extensão de gaps foram, respectivamente, -5 e -2. Em todos os testes da Seção 7.3, o mecanismo de re-trabalho estava ativado.

7.2 Bancos de dados selecionados

Para realizar os testes, foram selecionados os mesmos bancos de dados genômicos utilizados por Hains *et. al* [29]: Ensembl Dog [19] e Rat [20], RefSeq Human e Mouse [51] e UniProtDB/Swiss-Prot [17]. A Tabela 7.1 mostra os bancos de dados, o número de sequências contidas em cada um, o número de sequências de busca e o tamanho da menor e da maior sequência de busca em cada caso.

Banco de dados	Sequências	Sequências de busca	Menor sequência de busca	Maior sequência de busca
Ensembl Dog Proteins	25160	40	100	4996
Ensembl Rat Proteins	32971	40	100	4992
RefSeq Human Proteins	34705	40	100	4981
RefSeq Mouse Proteins	29437	40	100	5000
UniProtDB/Swiss-Prot	537505	40	100	4998

Tabela 7.1: Informações sobre os bancos de dados selecionados para os testes.

Para testar a nossa ferramenta com cada um dos bancos de dados selecionados, foram utilizadas 40 sequências de busca. Assim como foi feito por Liu *et. al* [42], foi gerada,

para cada banco de dados, uma busca composta por sequências de tamanho indo de 100 a 5000, com tamanhos igualmente espaçados.

7.3 Tempos de Execução e GCUPS

7.3.1 Execução com núcleos SSE

Na execução dos testes utilizando apenas núcleos SSE, coletamos resultados para 1, 2, 4 e 8 núcleos. Para 1, 2 e 4 núcleos, somente uma máquina foi utilizada. Para 8 núcleos, duas máquinas foram utilizadas.

A Tabela 7.2 mostra o desempenho obtido para os bancos de dados selecionados. Observamos que cada núcleo SSE é capaz de atualizar aproximadamente 2,8 bilhões de células por segundo (GCUPS). Além disso, esse desempenho se mantém nos diferentes bancos de dados selecionados, com variados tamanhos. Pode ser visto, também, que o mesmo ocorre ao se utilizar mais de uma unidade de processamento. Mesmo com a diferença de tamanho entre os bancos de dados, o desempenho das unidades de processamento se manteve próximo do ideal.

Banco de dados	1 SSE Tempo (s) GCUPS	2 SSE Tempo (s) GCUPS	4 SSE Tempo (s) GCUPS	8 SSE Tempo (s) GCUPS
Ensembl Dog	553,03 2,68	283,27 5,23	157,13 9,43	76,74 19,31
Ensembl Rat	629,08 2,77	337,27 5,17	181,31 9,61	88,89 19,6
NCBI RefSeq Human	673,55 2,92	353,79 5,56	197,87 9,95	96,02 20,49
NCBI RefSeq Mouse	572,47 2,8	305,92 5,24	169,54 9,46	82,01 19,55
UniProtDB/Swiss-Prot	7190,6 2,8	3615,38 5,38	2020,68 9,63	1027,28 18,94

Tabela 7.2: Resultados obtidos na execução com SSE em tempo de execução e GCUPS.

O Gráfico 7.1 mostra o speedup das unidades de processamento do tipo SSE. Como pode ser visto, as unidades de processamento do tipo SSE possuem speedup próximo do linear. Além disso, observa-se que essa evolução se manteve nos diferentes bancos de dados que fizeram parte dos testes.

7.3.2 Execução com GPU

Na execução dos testes com unidades de processamento apenas do tipo GPU, foram utilizadas 1, 2 e 4 GPUs. Para 4 GPUs, duas máquinas foram utilizadas.

A Tabela 7.3 mostra os resultados obtidos. Com exceção do banco de dados UniProtDB/SwissProt, cada GPU obteve desempenho aproximado de 20 GCUPS. Pode ser visto que, no caso do banco de dados UniProtDB/Swiss-Prot, o desempenho foi bem superior, próximo de 44 GCUPS. Acreditamos que a diferença no desempenho se deva

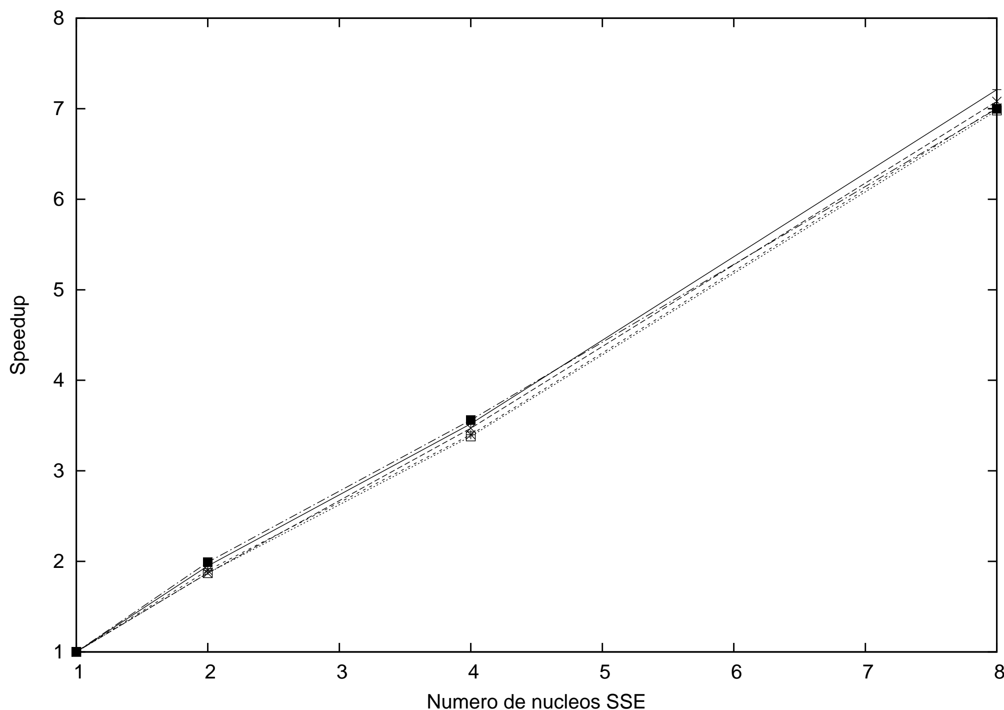


Figura 7.1: Speedup obtido na execução com SSE.

ao tamanho do banco de dados, muito maior que os outros. Com um banco de dados grande, cada comparação demora mais tempo na GPU, com menos interação com a CPU, podendo assim se beneficiar melhor do poder computacional da GPU.

O melhor desempenho foi obtido ao se comparar o banco de dados UniProtDB/Swiss-Prot com 4 GPUs (163,33 GCUPS). Nesse caso, o tempo de execução foi reduzido de 7 minutos e 18 segundos (1 GPU) para 1 minuto e 58 segundos (4 GPUs).

O Gráfico 7.2 mostra o speedup obtido para as GPUs. Como pode ser visto, as unidades de processamento do tipo GPU, na maior parte dos casos, mantêm speedup próximo do linear. Para os bancos de dados menores (RefSeq Mouse e Ensembl Dog) o speedup ficou próximo de 3×, para 4 GPUs. Pode ser visto também que o desempenho das GPUs flutua um pouco mais que os processadores SSE.

7.3.3 Execução com GPU e SSE

Foram executados testes, também, utilizando simultaneamente unidades de processamento do tipo SSE e GPU. O objetivo era observar se, embora os cores SSE sejam bem mais lentos que as GPUs, a utilização destes aumenta o desempenho total, ou seja, diminui o tempo necessário para realizar toda a comparação.

A Tabela 7.4 mostra os resultados obtidos, em tempo de execução e GCUPS. De fato, como pode ser observado, mesmo com a presença de uma unidade de processamento bem

Banco de dados	1 GPU Tempo (s) GCUPS	2 GPU Tempo (s) GCUPS	4 GPU Tempo (s) GCUPS
Ensembl Dog	72,54 20,43	39,44 37,58	23,75 62,41
Ensembl Rat	93,34 18,62	52,85 32,96	25,9 67,25
NCBI RefSeq Human	89,16 22,07	46,52 42,3	24,75 79,52
NCBI RefSeq Mouse	69,99 22,91	36,82 43,54	22,95 69,81
UniProtDB/Swiss-Prot	439,15 44,3	222,85 87,3	118,67 163,93

Tabela 7.3: Resultados obtidos na execução com GPU em tempo de execução e GCUPS.

mais rápida (GPU), a adição de unidades de processamento do tipo SSE é benéfica para o desempenho total da ferramenta. Isso ocorre porque o mecanismo de re-trabalho (Seção 6.5.1) garante que em momento algum as unidades de processamento mais rápidas vão ficar ociosas, mesmo que não existam mais tarefas a serem alocadas.

Como pode ser visto ao se comparar a Tabela 7.3 com a Tabela 7.4, a adição de núcleos SSE às GPUs é sempre capaz de reduzir o tempo de execução e, consequentemente, aumentar o GCUPS.

Nesse caso, o melhor desempenho (172,82 GCUPS) foi obtido com 4 GPUs e 4 núcleos SSE. Note que esse desempenho é maior do que os desempenhos reportados pelos artigos analisados no Capítulo 5 (Tabela 7.4).

Banco de dados	1SSE+ 1GPU Tempo (s) GCUPS	2SSE+ 1GPU Tempo (s) GCUPS	4SSE+ 1GPU Tempo (s) GCUPS	4SSE+ 2GPU Tempo (s) GCUPS	4SSE+ 4GPU Tempo (s) GCUPS
Ensembl Dog	65,02 22,79	60,87 24,35	53,82 27,53	33,47 44,28	24,93 59,46
Ensembl Rat	84,64 20,58	79,79 21,83	68,33 25,49	47,99 36,3	28,83 60,41
NCBI RefSeq Human	81,09 24,27	78,67 25,01	65,47 30,06	46,87 41,98	25,16 71,77
NCBI RefSeq Mouse	65,44 24,5	61,01 26,28	50,59 31,69	34,38 46,63	25,16 63,67
UniProtDB/Swiss-Prot	425,19 45,76	400,64 48,56	393,17 49,48	211,85 91,83	112,43 172,82

Tabela 7.4: Resultados obtidos na execução com SSE e GPU em tempo de execução e GCUPS.

O Gráfico 7.3 mostra os speedups obtidos nesses testes. É possível observar que o melhor speedup é obtido ao se comparar um banco genômico pequeno (Ensembl Dog).

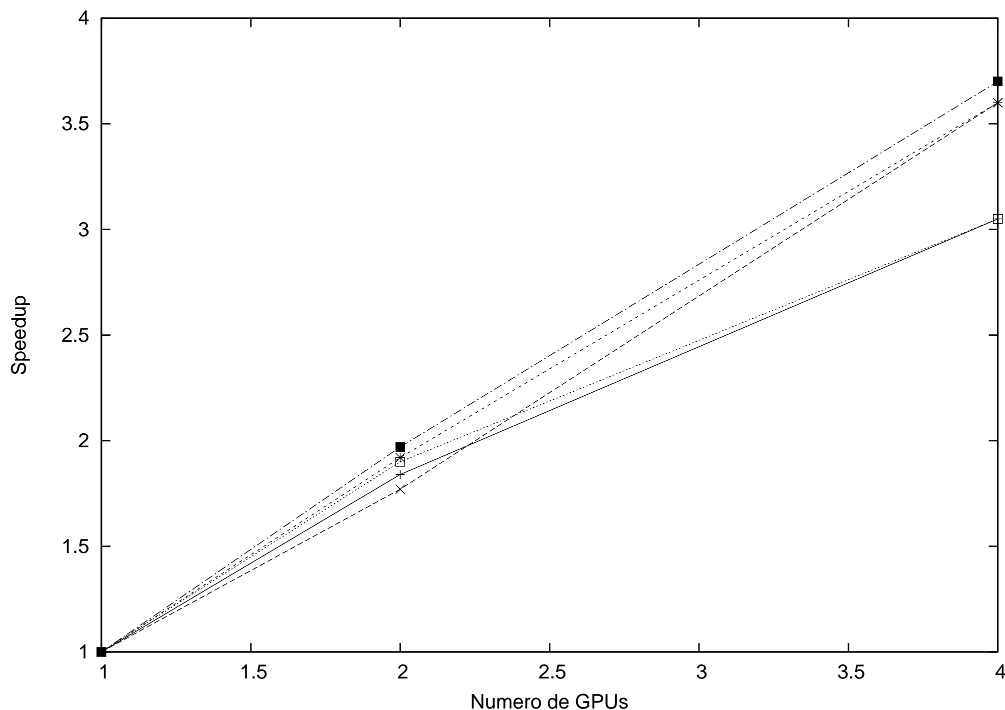


Figura 7.2: Speedup obtido na execução com GPU.

No entanto, ao se passar de 1 GPU para 2 GPUs, a comparação do UniProtKB/Swiss-Prot obtém um ganho substancial de desempenho, passando a ter o segundo melhor speedup.

7.3.4 Avaliação do Mecanismo de re-trabalho

Durante a execução dos testes sem o mecanismo de re-trabalho, percebemos uma diminuição na velocidade de processamento nas últimas comparações. Essa diminuição ocorria quando adicionávamos uma unidade de processamento mais lenta, em comparação com a mesma execução utilizando apenas as unidades de processamento rápidas. A diminuição ocorre porque, nas últimas tarefas, pode ser que a unidade de processamento mais lenta receba a tarefa e fique um longo tempo processando, enquanto a unidade de processamento mais rápida já terminou a sua parte da execução e está ociosa (Seção 6.4). Por exemplo, sem o mecanismo de re-trabalho, o desempenho da comparação caiu de 78,38 GCUPS com 2 GPUs para 35,62 utilizando 2 GPUs e 1 núcleo SSE.

O mecanismo de re-trabalho foi proposto justamente para resolver esse problema. Ele evita que unidades de processamento fiquem ociosas, mesmo depois que todas as comparações já foram alocadas. Além disso, ele apenas começa a atuar quando a última sequência do conjunto de sequências de busca é alocada. A partir desse momento, se uma das unidades de processamento ficar ociosa e uma ou mais unidades de processamento ainda estão comparando sequências anteriores, são alocadas a essa unidade de processamento.

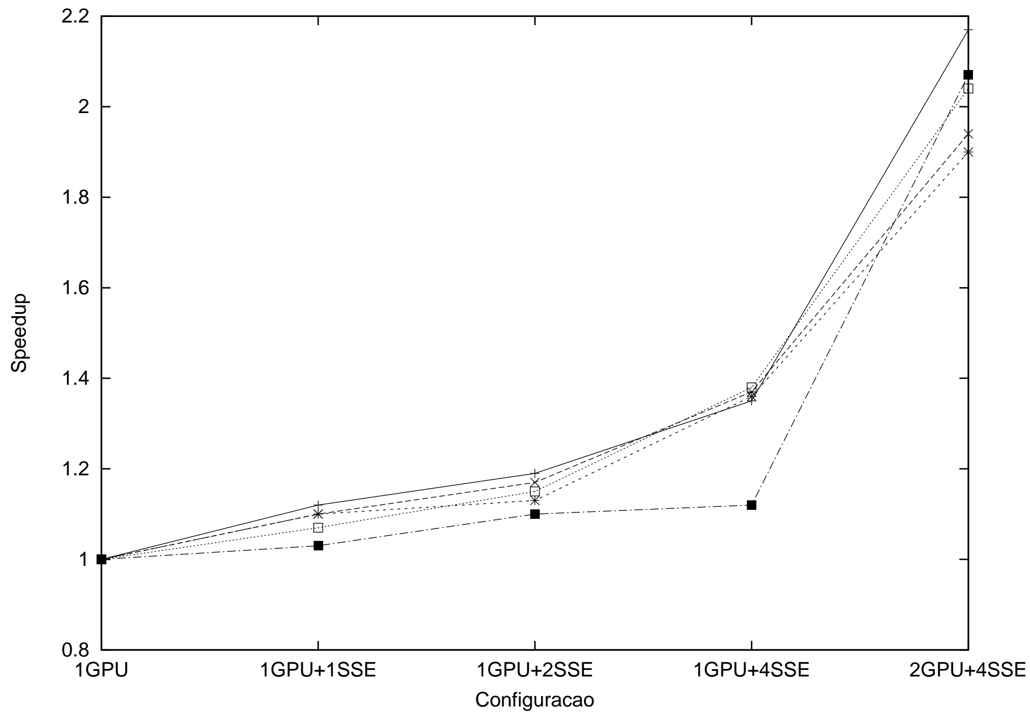


Figura 7.3: Speedup obtido na execução com SSE e GPU.

Para evitar que ocorra re-trabalho desnecessariamente, apenas sequências que ainda não tiveram seus resultados recebidos podem ser realocadas. Além disso, apenas blocos contíguos de sequências do conjunto de sequências de busca podem ser alocados e, para manter o código coerente e o mais simples possível, a mesma característica se aplica ao mecanismo de re-trabalho.

A Figura 7.4 apresenta os GCUPS para 2 GPUs, 2 GPUs + 1 núcleo SSE, 2 GPUs + 2 núcleos SSE e 4 GPUs + 4 núcleos SSE, com e sem o mecanismo de re-trabalho. Como pode ser visto nessa figura, o mecanismo de re-trabalho não incorre em perda de desempenho quando este não é necessário. Na configuração uniforme (2 GPUs), há uma variação mínima no tempo de execução ao incluir o mecanismo. Isso mostra que esse mecanismo possui um overhead pequeno.

Já na segunda configuração (2 GPUs e 1 núcleo SSE), a utilização do mecanismo é essencial para que ocorra uma melhoria, mesmo que pequena, no tempo de execução, quando comparada à execução somente com GPUs. O desempenho total, nesse caso, foi de 35,62 GCUPS sem o mecanismo e 83,41 GCUPS com o mecanismo, um aumento de 134%.

O mesmo comportamento ocorre para 2 GPUs + 2 núcleos SSE e 4 GPUs + 4 núcleos SSE. Na terceira configuração (2 GPU + 2 núcleos SSE), o desempenho foi de 40,14 GCUPS sem o mecanismo e 81,21 GCUPS utilizando-o, com um ganho de 102%. Na última configuração, que inclui 4 GPUs e 4 núcleos SSE, o desempenho sem o mecanismo

foi de 120,38 GCUPS e 176,14 GCUPS utilizando o mecanismo, um aumento de 46,31%.

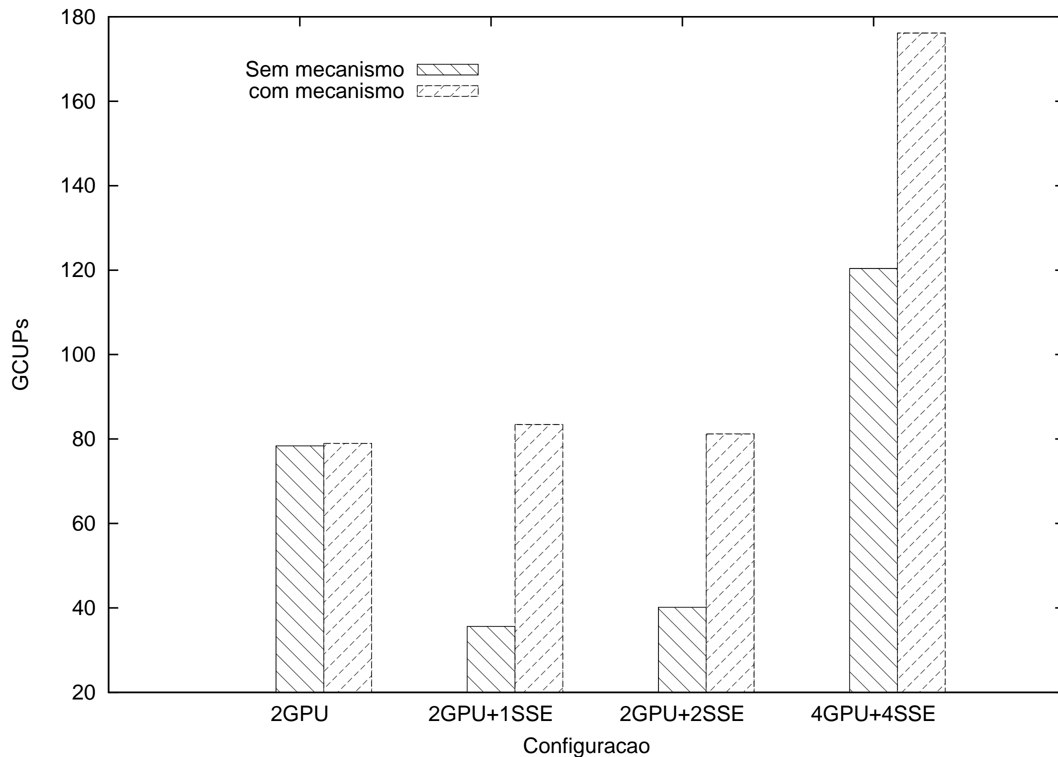


Figura 7.4: Ganho de desempenho ao utilizar o mecanismo de re-trabalho.

7.3.5 Execução com carga local

Para verificar como a política de alocação de tarefas se comporta em casos onde há variações bruscas de carga, criamos as Figuras 7.5 e 7.6, utilizando a política de alocação *Self-Scheduling*. Em ambos, cada ponto indica a entrega do resultado de uma tarefa ao mestre e a coordenada y indica a velocidade de processamento desta tarefa, em GCUPS. Na Figura 7.5 foram utilizados quatro núcleos SSE, todos no mesmo *host*. Não foi inserida qualquer redução artificial no desempenho. As sequências utilizadas foram as mesmas das seções anteriores e o banco de dados comparado foi o Ensembl Dog.

Já na Figura 7.6, foi inserida uma redução artificial de desempenho no núcleo 0 a partir da sexta tarefa recebida. Utilizamos uma aplicação de *benchmarking* chamada SuperPI [56] que calcula o número π com um número grande de casas com uso intenso do processador.

Como pode ser visto na Figura 7.5, a queda de desempenho do núcleo 0 é de aproximadamente 50%, o que causa um impacto também no número de tarefas que esse núcleo

consegue entregar. A partir da redução na velocidade, os outros núcleos receberam entre 5 e 6 tarefas, cada um. Já o núcleo que teve a sua velocidade reduzida recebeu apenas 2 tarefas. É importante notar também que a queda no desempenho de um núcleo não prejudica a execução nos outros núcleos, como pode ser visto ao comparar o gráfico da Figura 7.6 com o gráfico da Figura 7.5, sem a queda de desempenho.

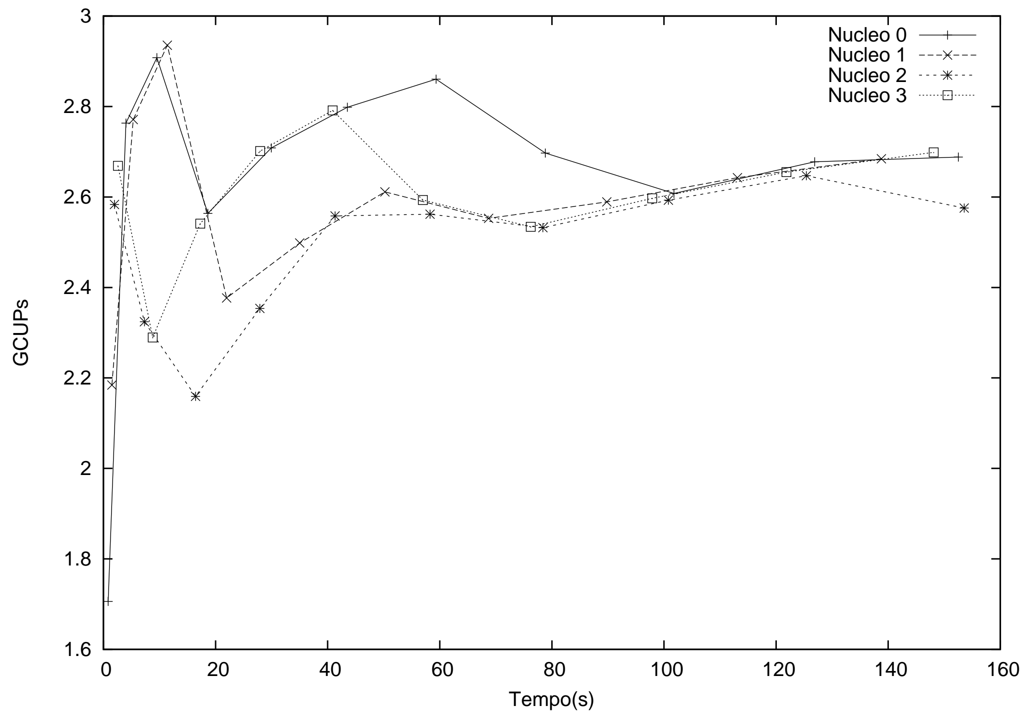


Figura 7.5: Execução normal com quatro núcleos SSE comparando o banco de dados Ensembl Dog.

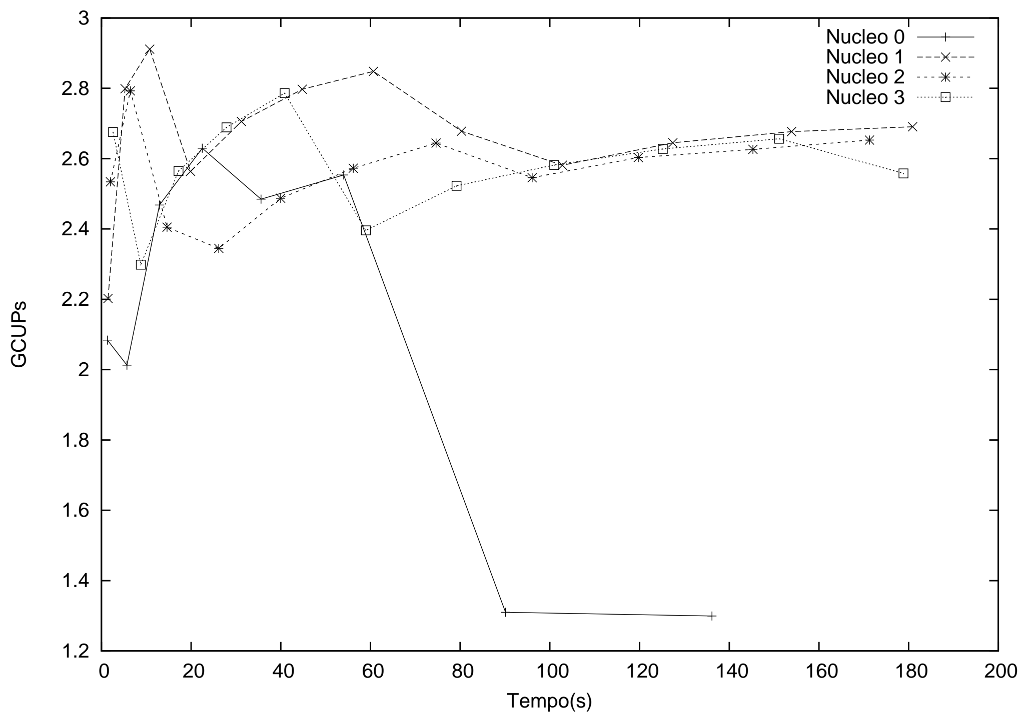


Figura 7.6: Execução com quatro núcleos SSE comparando o banco de dados Ensembl Dog. O desempenho do núcleo 0 foi artificialmente reduzido para demonstrar o funcionamento da alocação de tarefas em ambiente não-dedicado.

Capítulo 8

Conclusão

8.1 Conclusão

Nessa dissertação, propusemos e avaliamos uma abordagem de comparação paralela exata de sequências utilizando plataformas híbridas de alto desempenho. A nossa abordagem baseou-se na arquitetura mestre-escravo para implementar uma ferramenta capaz de comparar conjuntos de sequências biológicas com bancos de dados.

Uma das principais metas da nossa abordagem é ser flexível. Por esse motivo, a ferramenta foi implementada com a capacidade de agregar novas unidades de processamento, sejam do mesmo tipo ou de tipos diferentes. Ela é capaz também de utilizar novas técnicas e ferramentas de comparação que venham a ser disponibilizadas no futuro, sendo necessário apenas que estas sejam encapsuladas no protocolo de rede utilizado pela nossa ferramenta. Adicionalmente, a nossa ferramenta permite que novas políticas de alocação de tarefas sejam utilizadas, bastando que a classe de alocação de tarefas tenha suas propriedades herdadas e seja estendida numa nova classe.

No estudo realizado sobre o estado da arte, percebemos que as abordagens limitaram-se a duas políticas de alocação de tarefas: *Self-Scheduling* (SS) e fixa. Além disso, os pesos utilizados nas abordagens que aplicaram a política de alocação fixa concentram-se no desempenho teórico das unidades de processamento utilizadas ou em um único teste de desempenho, realizado antes da execução principal. A nossa abordagem, além de permitir a utilização de diversas políticas de alocação, baseou os testes na política de alocação *Package-Weighted Self-Scheduling* (PSS), que varia o número de tarefas alocadas às UPs durante a execução das comparações, baseando-se no desempenho demonstrado por cada UP separadamente.

Nas partes finais da comparação de sequências com um banco de dados, é possível que todas as tarefas já tenham sido alocadas e algumas UPs estejam ociosas. Isso ocorre porque, na alocação de tarefas, uma das UPs mais lentas recebeu a tarefa e ainda não terminou a comparação. Em casos como esse, nenhuma das abordagens analisadas permite que as tarefas sejam redistribuídas às UPs que estão ociosas. A nossa ferramenta implementou essa técnica e mostramos que, ao utilizá-la, um ganho de 102% foi obtido, em termos de GCUPS. Com essa técnica, conseguimos atingir 172,82 GCUPS, utilizando 4 GPUs e 4 núcleos SSE. Esse desempenho é superior aos resultados apresentados pelos artigos analisados no Capítulo 5 (estado da arte).

8.2 Trabalhos futuros

Uma das limitações da nossa ferramenta é evidenciada quando uma das unidades de processamento falha durante a execução. Por esse motivo, deseja-se propor uma extensão do mecanismo de re-trabalho, que seja capaz de atribuir as tarefas que estavam alocadas à unidade de processamento que falhou para outras unidades de processamento.

Por fim, desejamos aplicar a ferramenta proposta na presente dissertação na execução de outras aplicações de Bioinformática, tais como alinhamento múltiplo de sequências e em plataformas com um número maior de UPs.

Além disso, diferentemente de algumas das abordagens analisadas no estudo do estado da arte, a nossa abordagem não divide o banco de dados em segmentos menores para facilitar a distribuição do trabalho entre as UPs (granularidade grossa). Isso poderia ser feito em casos em que o banco de dados é muito extenso e, por isso, leva um longo tempo para ser comparado pelas UPs mais lentas. Nesse caso, as UPs mais lentas comparariam segmentos menores do banco de dados, enquanto as UPs mais rápidas comparam segmentos maiores ou, mais provavelmente, o banco de dados todo.

Glossário

cluster conjunto de computadores conectados trabalhando juntos de tal forma que em muitos aspectos são considerados um único sistema. vii, 1, 4, 5, 11, 40, 45, 46

grid conjunto de computadores de múltiplas localidades conectados para atingir um mesmo objetivo. 1, 23, 24, 27, 29, 42, 43

Lista de Abreviaturas e Siglas

AVX *Advanced Vector Extensions*. iv, vii, 5–7

CPU *Central Processing Unit*. vii, ix, 8, 10, 34–38, 43–46, 52, 60

CUDA *Compute Unified Device Architecture*. iv, vii, 8–11, 36, 37, 45, 48, 53, 58

FPGA *Field-Programmable Gate Array*. iii, vii, ix, 1, 4, 5, 37–40, 46

GCUPS *Giga Cells Updates per Second*. vi, ix, 35, 36, 38, 42, 45–47, 49, 52, 59–64, 67

GPU *Graphics Processing Unit*. iv–vii, ix, 1, 2, 4–11, 34–38, 43–46, 49, 52–54, 58–63, 67

HPC *High Performance Computing*. 1

MIMD *Multiple-Instruction Multiple-Data*. 4

OpenCL *Open Computing Language*. 10, 11

SIMD *Single-Instruction Multiple-Data*. iv, vii, 1, 4–6, 8

SMP *Symmetric multiprocessor*. 4

SSE *Streaming SIMD Extensions*. iv, vi–ix, 1, 2, 4–7, 39–42, 44–49, 52, 53, 55, 59–67

SW Smith-Waterman. vii, ix, 1, 2, 17–19, 21, 34, 35, 37–39, 41, 44, 45, 54, 55

Referências

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990. 22

[2] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389, 1997. 42

[3] AMD. All graphics cards, 2012. URL `http://www.amd.com/us/products/Pages/graphics.aspx`. ix, 11

[4] Khaled Benkrid, Ali Akoglu, Cheng Ling, Yang Song, Ying Liu, and Xiang Tian. High performance biological pairwise sequence alignment: Fpga vs. gpu vs. cell be vs. gpp. 2012. v, ix, 37, 38

[5] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356–368. IEEE, 1994. 33

[6] T. Bonny and K.N.S.M.A. Zidan. High performance technique for database applications using a hybrid gpu/cpu platform. In *Proceedings of the 21st edition of the great lakes symposium, New York*, pages 879–899, 2011. v, vii, 8, 34, 35, 46, 47

[7] A. Boukerche, A.C.M.A. De Melo, E.F.O. Sandes, and M. Ayala-Rincon. An exact parallel algorithm to compare very long biological sequences in clusters of workstations. *Cluster Computing*, 10(2):187–202, 2007. ISSN 1386-7857. 1

[8] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*, pages 349–363. IEEE, 2000. 30

[9] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, 14(2): 141–154, 1988. vii, 24, 25, 31

[10] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 74–83. ACM, 1995. 36

[11] L. Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single-and multi-gpu systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010. v, vii, 1, 36, 37, 46, 47

[12] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F.A.B. Silva, C.O. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The mygrid approach. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 407–416. IEEE, 2003. 30

[13] M.O. Dayhoff and R.M. Schwartz. A model of evolutionary change in proteins. In *In Atlas of protein sequence and structure*. Citeseer, 1978. 14

[14] Edans Flavius de O. Sandes and Alba Cristina M. A. de Melo. Smith-waterman alignment of huge sequences with gpu in linear space. *IEEE International Parallel & IEEE International Parallel & Distributed Processing Symposium*, 2011. 10

[15] J. Dongarra. Trends in high performance computing. *The Computer Journal*, 47(4): 399–403, 2004. 4

[16] R. Durbin, S.R. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998. 1, 12, 16

[17] EBI. Uniprotkb/swiss-prot homepage, 2012. URL `http://www.ebi.ac.uk/uniprot`. 13, 42, 43, 58

[18] Ensembl. Ensembl, 2012. URL `http://www.ensembl.org/index.html`. 13, 44

[19] Ensembl. Ensembl dog, 2012. URL `http://www.ensembl.org/Canis_familiaris`. 58

[20] Ensembl. Ensembl rat, 2012. URL `http://www.ensembl.org/Rattus_norvegicus`. 58

[21] M. Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007. 1, 34, 41, 42, 55

[22] M.S. Farrar. Optimizing smith-waterman for the cell broadband engine, 2008. 1, 44, 48, 54, 55

[23] M.J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972. 4

[24] D.G. George, W.C. Barker, and L.T. Hunt. Mutation data matrix and its uses. *Methods in enzymology*, 183:333–351, 1990. 15

[25] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982. 18

[26] gpgpu.org. General-purpose computation on graphics hardware, 2012. URL `http://gpgpu.org/`. 10

[27] GPUReview. Gpureview, 2012. URL `http://www.gpureview.com/`. ix, 11

[28] Khronos Group. Opencl, 2012. URL `http://www.khronos.org/opencl`. 10

[29] D. Hains, Z. Cashero, M. Ottenberg, W. Bohm, and S. Rajopadhye. Improving cudasw++, a parallelization of smith-waterman for cuda enabled devices. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 490–501. IEEE, 2011. 58

[30] Khaled Hamidouche, FernandoMachado Mendonca, Joel Falcou, AlbaCristinaMagalhaesAlves Melo, and Daniel Etiemble. Parallel smith-waterman comparison on multicore and manycore computing platforms with bsp++. *International Journal of Parallel Programming*, 41:111–136, 2013. ISSN 0885-7458. doi: 10.1007/s10766-012-0209-6. URL `http://dx.doi.org/10.1007/s10766-012-0209-6`. 51

[31] S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915, 1992. 14, 15, 44

[32] F. Ino, Y. Kotani, Y. MUNEKAWA, and K. HAGIHARA. Harnessing the power of idle gpus for acceleration of biological sequence alignment. *Parallel Processing Letters*, 19(04):513–533, 2009. v, vii, ix, 42, 43, 46, 47

[33] F. Ino, Y. Munekawa, and K. Hagihara. Sequence homology search using fine grained cycle sharing of idle gpus. *Parallel and Distributed Systems, IEEE Transactions on*, 23(4):751–759, 2012. v, vii, 1, 10, 44, 46, 47

[34] Intel. Intel press release, 2012. URL `http://blogs.intel.com/technology/2012/01/intels-innovative-year-sandy-bridge-wins-best-processor/`. 5, 6

[35] SSE Intel. Programming reference. *Intel's software network, sofwareprojects. intel. com/avx*, 2007. vii, 7

[36] B. Jacek, F. Wojciech, K. Michal, P. Erwin, and W. Pawel. Protein alignment algorithms with an efficient backtracking routine on multiple gpus. *BMC Bioinformatics*, 12, 2012. v, ix, 10, 44, 45, 46, 47

[37] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun. A reconfigurable accelerator for smith–waterman algorithm. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 54(12):1077–1081, 2007. 1

[38] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, 2002. 27

[39] V. Kumar, A.Y. Grama, and N.R. Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994. 33

[40] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a gpu. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006. ISBN 1424400546. 45

[41] Y. Liu, W. Huang, J. Johnson, and S. Vaidya. Gpu accelerated smith-waterman. *Computational Science–ICCS 2006*, pages 188–195, 2006. 42

[42] Y. Liu, B. Schmidt, and D. Maskell. Cudasw++ 2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions. *BMC Research Notes*, 3(1):93, 2010. 1, 8, 10, 35, 42, 48, 49, 53, 54, 58

[43] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. 31

[44] D. Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838. IEEE, 2008. 10

[45] X. Meng and V. Chaudhary. A high-performance heterogeneous computing platform for biological sequence analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 21(9):1267–1280, 2010. v, vii, 1, 39, 40, 46, 47

[46] David W. Mount. *Bioinformatics: sequence and genome analysis*. CSHL press, 2004. 1, 12, 16

[47] E.W. Myers and W. Miller. Optimal alignments in linear space. *Computer applications in the biosciences: CABIOS*, 4(1):11–17, 1988. 44

[48] H. Nash, D. Blair, and J. Grefenstette. Comparing algorithms for large-scale sequence analysis. In *Bioinformatics and Bioengineering Conference, 2001. Proceedings of the IEEE 2nd International Symposium on*, pages 89–96. IEEE, 2001. 43

[49] NCBI. National center for biotechnology information, 2012. URL `http://www.ncbi.nlm.nih.gov`. 42

[50] Ncbi. Month.aa database, 2012. URL `http://www.ncbi.nlm.nih.gov`. 40

[51] NCBI. Refseq, 2012. URL `http://www.ncbi.nlm.nih.gov/RefSeq/`. 13, 58

[52] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970. 16

[53] NFS. Network file system, 2012. URL `http://nfs.sourceforge.net/`. 50

[54] J. Nickolls and W.J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, 2010. ix, 10, 11

[55] C. Nvidia. Compute unified device architecture programming guide. *NVIDIA: Santa Clara, CA*, 83:129, 2007. vii, ix, 6, 7, 8, 9, 10, 11

[56] Takuya Ooura. Super pi, 2012. URL `http://myownlittleworld.com/miscellaneous/computers/piprogram.html`. 64

[57] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Dover publications, 1998. 23, 25

[58] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444, 1988. 21

[59] G.F. Pfister. *In search of clusters*. Prentice-Hall, Inc., 1998. 4, 19

[60] C.D. Polychronopoulos and D.J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *Computers, IEEE Transactions on*, 100(12): 1425–1439, 1987. 32

[61] T. Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC bioinformatics*, 12(1):221, 2011. v, vii, 1, 41, 42, 46, 47, 54

[62] T. Rognes and E. Seeberg. Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16 (8):699, 2000. 1, 41

[63] R.M. Russell. The cray-1 computer system. *Communications of the ACM*, 21(1): 63–72, 1978. 5

[64] E.F.O. Sandes and A.C. de Melo. Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pages 137–146. ACM, 2010. 1, 8

[65] G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, Citeseer, 2001. 31

[66] A. Singh, C. Chen, W. Liu, W. Mitchell, and B. Schmidt. A hybrid computational grid architecture for comparative genomics. *Information Technology in Biomedicine, IEEE Transactions on*, 12(2):218–225, 2008. 10, 46, 47

[67] J. Singh and I. Aruni. Accelerating smith-waterman on heterogeneous cpu-gpu systems. In *Bioinformatics and Biomedical Engineering,(iCBBE) 2011 5th International Conference on*, pages 1–4. IEEE, 2011. v, 1, 35, 46, 47, 48

[68] TF Smith and MS Waterman. Identification of common molecular subsequences. *J. Mol. Bwl*, 147:195–197, 1981. 1, 17

[69] M.S. Sousa, A.C.M.A. Melo, and A. Boukerche. An adaptive multi-policy grid service for biological sequence comparison. *Journal of parallel and distributed computing*, 70 (2):160–172, 2010. 33

[70] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz. Swps3 - fast multithreaded vectorized smith-waterman for ibm cell/be and sse2. *BMC Research Notes*, 1(1):107, 2008. 1, 35, 42

[71] A.S. Tanenbaum and R. Van Renesse. Distributed operating systems. *ACM Computing Surveys (CSUR)*, 17(4):419–470, 1985. 23

[72] P. Tang and P.C. Yew. Processor self-scheduling for multiple-nested parallel loops. In *Proceedings of the 1986 international conference on parallel processing*, pages 528–535, 1986. 32

[73] Y.T. Wang and R.J.T. Morris. Load sharing in distributed systems. *Computers, IEEE Transactions on*, 100(3):204–217, 1985. 27

[74] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer applications in the biosciences: CABIOS*, 13(2):145, 1997. 41

[75] F. Xhafa and A. Abraham. Computational models and heuristic methods for grid scheduling problems. *Future Generation Computer Systems*, 26(4):608–621, 2010. 23