Evandro Augusto Marucci

Paralelização da ferramenta de alinhamento de seqüências MUSCLE para um ambiente distribuído

São José do Rio Preto – SP

Fevereiro / 2009

Evandro Augusto Marucci

Paralelização da ferramenta de alinhamento de seqüências MUSCLE para um ambiente distribuído

Dissertação apresentada à Coordenação do Mestrado em Ciência da Computação da UNESP/IBILCE para a obtenção do título de Mestre em Ciência da Computação

Orientador:
Prof. Dr. José Márcio Machado

Mestrado em Ciência da Computação
Departamento de Ciências da Computação e Estatística
Instituto de Biociências, Letras e Ciências Exatas
Universidade Estadual Paulista

São José do Rio Preto – SP

Fevereiro / 2009

Dedico esta dissertação à memória de meu amigo Angelo 'Bozo' Morato (1985 - 2008)

Agradecimentos

Devo este trabalho a muitas pessoas e instituições, sem aos quais sua realização não seria possível.

Antes de tudo, porém, dou graças à Deus pela felicidade que ele proporciona constantemente em minha vida. Por me fazer acreditar no amor e nas pessoas e por me proporcionar uma paz integral, em momentos de grandes incertezas.

Aos meus pais, Luis e Zezé, pelo constante amor e suporte em todas as minhas realizações. Ao meu irmão Gustavo pelo seu amor e cuidado, e também por estar sempre presente. Aos meus primos, tios e minhas avós, que sempre me passam um carinho muito grande. Todo este amor e carinho esteja eu aqui ou há milhares de quilômetros de distância.

Ao meu orientador, Prof. Dr. José Márcio Machado, por compartilhar comigo recursos e pontos de vista essenciais para o meu crescimento pessoal e profissional. Toda a estrutura provida por ele foi, sem dúvida, de suma importância para o desenvolvimento deste trabalho. Mais do que isto agradeço a sua disposição em me oferecer estímulos e inspiração, motivando-me a percorrer novos caminhos. Agradeço a nossa amizade desenvolvida, principalmente.

Ao meu grande amigo Geraldo (Bochecha), por além de amigo ser um grande parceiro profissional. Agradeço a sua presença constante na solução de problemas, dentro e fora do laboratório, e por tudo o que compartilhamos em nossa vida pessoal. Por ter sido a pessoa mais presente enquanto universitário.

Ao Prof. Dr. Yang Shiyou pelo período maravilhoso que passei na China. Pelo seu cuidado e sua generosidade em me oferecer até mais do que precisava. Pela confiança em minhas atitudes e pela liberdade me proporcionada.

Ao Prof. Dr. Aleardo Manacero Jr., por ter me iniciado à pesquisa e ter me introduzido

à computação paralela. A base de pesquisa que obtive com ele foi fundamental para o tranqüilo andamento deste trabalho.

Aos meus amigos do Brasil, Ivan, Sérgio, Alex, Luizinho, Francês, aos amigos do Chuck Norris, e a todos que, de alguma forma, se divertiram comigo durante este período.

Aos meus amigos na China, Forrest, Mayur, Ma Rui, Alisa, Pardo, Michael, Tracey, meu mais que brother Thiago Lins e meu quase tio Américo. Em especial à minha namorada Sissi por todo o seu amor.

Ao pessoal do laboratório genôma pela estrutura física, em especial à Helen e ao Gerrard. À Helen pela disposição em prover o cluster para a execução dos meus primeiros testes e pelo prazer de trabalhar com uma pessoa simpática como ela. Ao Gerrard pela amizade, pelas garrafas de vinho e por constantemente compartilhar suas histórias de vida.

À Aline do IFT por me dar acesso ao cluster em São Paulo, através do qual também pude testar e medir os resultados de meu trabalho.

À FAPESP, que durante 24 meses financiou minha pesquisa.

Agradeço à todos profundamente.

I can't be as confident about computer science as I can about biology. Biology easily has 500 years of exciting problems to work on. It's at that level.

Donald Knuth

Resumo

Devido a crescente quantidade de dados genômicos para comparação, a computação paralela está se tornando cada vez mais necessária para realizar uma das operações mais importantes da bioinformática, o alinhamento múltiplo de seqüências. Atualmente, muitas ferramentas computacionais são utilizadas para resolver alinhamentos e o uso da computação paralela está se tornando cada vez mais generalizado. Entretanto, embora diferentes algoritmos paralelos tenham sido desenvolvidos para suportar as pesquisas genômicas, muitos deles não consideram aspectos fundamentais da computação paralela.

O MUSCLE [1] é uma ferramenta que realiza o alinhamento múltiplo de seqüências com um bom desempenho computacional e resultados biológicos significativamente precisos [2]. Embora os métodos utilizados por ele apresentem diferentes versões paralelas propostas na literatura, apenas uma versão paralela do MUSCLE foi proposta [3]. Essa versão, entretanto, foi desenvolvida para sistemas de memória compartilhada.

O desenvolvimento de uma versão paralela do MUSCLE para sistemas distribuídos é importante dado o grande uso desses sistemas em laboratórios de pesquisa genômica. Esta paralelização é o foco deste trabalho e ela foi realizada utilizando-se abordagens paralelas existentes e criando-se novas abordagens. Como resultado, diferentes estratégias paralelas foram propostas. Estas estratégias podem ser incorporadas a outras ferramentas de alinhamento que utilizam, em determinadas etapas, a mesma abordagem seqüencial.

Em cada método paralelizado, considerou-se principalmente a eficiência, a escalabilidade e a capacidade de atender problemas reais da biologia. Os testes realizados mostram que, para cada etapa paralela, ao menos uma estratégia definida atende bem todos esses critérios. Além deste trabalho realizar um paralelismo inédito, ao viabilizar a execução da ferramenta MUSCLE em sistemas distribuídos, os resultados obtidos mostram que as novas estratégias definidas apresentam um desempenho melhor do que as estratégias existentes.

Abstract

Due to increasing amount of genetic data for comparison, parallel computing is becoming increasingly necessary to perform one of the most important operations in bioinformatics, the multiple sequence alignments. Nowadays, many software tools are used to solve sequence alignments and the use of parallel computing is becoming more and more widespread. However, although different parallel algorithms were developed to support genetic researches, many of them do not consider fundamental aspects of parallel computing.

The MUSCLE [1] is a tool that performs multiple sequence alignments with good computational performance and biological results significantly precise [2]. Although the methods used by them have different parallel versions proposed in the literature, only one parallel version of the MUSCLE tool was proposed [3]. This version, however, was developed for shared memory systems.

The development of a parallel MUSCLE tool for distributed systems is important given the wide use of such systems in laboratories of genomic researches. This parallelization is the aim of this work and it was done using existing parallel approaches and creating new approaches. Consequently, different parallel strategies have been proposed. These strategies can be incorporated into other alignment tools that use, in a given stage, the same sequential approach.

In each parallel method, we considered mainly the efficiency, scalability and ability to meet real biological problems. The tests show that, for each parallel step, at least one defined strategy meets all these criteria. In addition to the new MUSCLE parallelization, enabling it execute in a distributed systems, the results show that the defined strategies have a better performance than the existing strategies.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução p. 17

1.1 Organização da dissertação . . . . p. 19

2 Fundamentação teórica do projeto p. 20

2.1 Genética e bioinformática . . . . p. 20

2.1.1 Código Genético: ácidos nucléicos e proteínas . . . . p. 21

2.1.2 Comparação de seqüências . . . . p. 22

2.2 Alinhamento de seqüências . . . . p. 23

2.2.1 Alinhamento entre pares de perfis . . . . p. 25

2.2.2 Algoritmo progressivo . . . . p. 27

2.2.3 Algoritmo iterativo . . . . p. 27

2.3 A metodologia da ferramenta MUSCLE . . . . p. 28

2.3.1 Funcionamento básico . . . . p. 28

2.4 Descrição das etapas e métodos do MUSCLE . . . . p. 29

2.4.1 Medidas de similaridades e estimativas de distância . . . . p. 29

2.4.2 Construção da árvore . . . . p. 33

2.4.3 Comparação de árvores . . . . p. 34
2.4.4 Alinhamento entre perfis . . . . p. 34
2.4.5 Pontuação objetiva . . . . p. 36
2.5 Aspectos da programação paralela e sistemas distribuídos . . . . p. 37
2.5.1 O modelo tarefa/canal . . . . p. 38
2.5.2 Metodologia de projeto de programas paralelos . . . . p. 39
2.5.3 MPI - Message Passage Interface . . . . p. 44
2.5.4 Medidas de desempenho . . . . p. 46
2.6 Abordagens paralelas de alinhamento . . . . p. 48
2.6.1 CLUSTALW-MPI . . . . p. 50
2.6.2 MUSCLE-SMP . . . . p. 50
2.6.3 Técnicas paralelas do alinhamento progressivo . . . . p. 51
2.6.4 Técnicas paralelas do alinhamento par-a-par . . . . p. 54

3 Detalhamento e desenvolvimento do projeto p. 56
3.1 Paralelização do método de contagem de *k-mers* . . . . p. 57
3.2 Paralelização do método da identidade fracional . . . . p. 59
3.3 Paralelização do alinhamento progressivo . . . . p. 60
3.3.1 Abordagem com gargalo e soluções . . . . p. 61
3.3.2 O problema da abordagem existente . . . . p. 61
3.3.3 Estratégia baseada na abordagem com gargalo . . . . p. 63
3.3.4 Novas abordagens paralelas . . . . p. 63
3.3.5 Soluções 1 e 2: Escalonar apenas tarefas com dependências em processos ociosos . . . . p. 65

3.3.6 Solução 3: Fazer cópia de todos os dados no processo mestre . . . p. 67

3.3.7 Solução 4: Criar *threads* exclusivos para a troca de dados . . . . . p. 70

3.3.8 Considerações sobre as implementações no segundo estágio . . . . p. 71

3.4 Paralelização do alinhamento par-a-par . . . . p. 73

3.4.1 Estratégias implementadas sobre ambas as soluções . . . . p. 76

3.4.2 O tamanho dos blocos da matriz . . . . p. 77

3.5 Paralelização do cálculo da pontuação objetiva . . . . p. 77

4 Testes e Resultados p. 79

4.1 Contagem de *k-mers* . . . . p. 79

4.2 Identidade fracional . . . . p. 84

4.3 Alinhamento progressivo . . . . p. 87

4.3.1 Comparação entre as estratégias . . . . p. 89

4.3.2 O nível de paralelismo . . . . p. 98

4.3.3 A árvore filogenética e a escalabilidade do algoritmo . . . . p. 99

4.4 Alinhamento par-a-par . . . . p. 102

4.4.1 Comparação entre as estratégias . . . . p. 103

5 Conclusões p. 110

5.1 Trabalhos futuros . . . . p. 111

Referências Bibliográficas p. 113

Lista de Figuras

2.1 *Exemplo de um alinhamento múltiplo de seqüências* . . . . p. 23

2.2 *Armazenamento do perfil em uma matriz* . . . . p. 26

2.3 *Diagrama de fluxo do algoritmo do MUSCLE* . . . . p. 30

2.4 *Cálculo da identidade fracional entre duas seqüências* . . . . p. 31

2.5 *O modelo tarefa/canal* . . . . p. 39

2.6 *O modelo de passagem de mensagens* . . . . p. 45

2.7 *Fluxograma do algoritmo do processo mestre do alinhamento progressivo paralelo com escalonamento dinâmico* . . . . p. 52

2.8 *Fluxograma do algoritmo do processo escravo do alinhamento progressivo paralelo com escalonamento dinâmico* . . . . p. 53

2.9 *Mapeamento da árvore filogenética para a árvore de tarefas* . . . . p. 53

2.10 *Particionamento da matriz de programação dinâmica em três regiões* . . p. 54

2.11 *Estratégia* block-based wavefront . . . . p. 55

3.1 *Fluxograma do algoritmo paralelo do método de contagem de* k-mers . . . p. 58

3.2 *Exemplo de como o cálculo da matriz de similaridades é distribuído entre os processos* . . . . p. 59

3.3 *Fluxograma do algoritmo paralelo do método da identidade fracional* . . . p. 60

3.4 *Exemplo de caso da espera pela execução de processo de escravo vizinho para envio de dados dependentes* . . . . p. 62

3.5 *Fluxograma do algoritmo do processo mestre da primeira estratégia* . . . p. 64

3.6 *Fluxograma do processo mestre das estratégias* waitall *e* waitany . . . . . p. 67

3.7 *Fluxograma do processo escravo das estratégias* waitall *e* waitany . . . . p. 68

3.8 *Fluxograma do algoritmo do processo mestre da estratégia* sendmaster . . p. 70

3.9 *Fluxograma do algoritmo do processo escravo da estratégia* sendmaster . p. 71

3.10 *Fluxograma do algoritmo do processo mestre da estratégia com* threads . p. 72

3.11 *Fluxograma do algoritmo do processo escravo da estratégia com* threads . p. 73

4.1 *Gráfico de tempo de execução do algoritmo paralelo de contagem de* k-mers *para entradas com seqüências de aproximadamente 1000 resíduos* . . . . . p. 80

4.2 *Gráfico de* speedup *real do algoritmo paralelo de contagem de* k-mers *para entradas com seqüências de aproximadamente 1000 resíduos* . . . . p. 81

4.3 *Gráfico de tempo de execução do algoritmo paralelo de contagem de* k-mers *para entradas com seqüências de aproximadamente 50 resíduos* . . . . . . p. 81

4.4 *Gráfico de* speedup *real do algoritmo paralelo de contagem de* k-mers *para entradas com seqüências de aproximadamente 50 resíduos* . . . . p. 82

4.5 *Gráfico de comparação do* speedup *real do algoritmo paralelo de contagem de* k-mers *para entradas com 4000 seqüências de aproximadamente 50 e 1000 resíduos* . . . . p. 83

4.6 *Percentual de tempo gasto com comunicação e sincronismo do algoritmo de contagem de* k-mers *para a entrada com 500 seqüências de aproximadamente 1000 resíduos* . . . . p. 83

4.7 *Gráfico de tempo de execução do algoritmo paralelo da identidade fracional para entradas com seqüências de aproximadamente 1000 resíduos* . . . . . p. 85

4.8 *Gráfico de tempo de execução do algoritmo paralelo da identidade fracional para entradas com seqüências de aproximadamente 50 resíduos* . . . . . . p. 86

4.9 *Gráfico de* speedup *real do algoritmo paralelo da identidade fracional para entradas com seqüências de aproximadamente 50 resíduos* . . . . p. 86

4.10 *Gráfico de ganho de desempenho do algoritmo paralelo da identidade fracional para entradas com 4000 seqüências de aproximadamente 50 e 1000 resíduos* . . . . p. 87

4.11 *Percentual de tempo gasto com comunicação e sincronismo do algoritmo paralelo da identidade fracional para a entrada com 500 seqüências de aproximadamente 1000 resíduos* . . . . p. 88

4.12 *Gráfico de tempo de execução da estratégia com gargalo do alinhamento progressivo* . . . . p. 89

4.13 *Percentual de tempo gasto com comunicação e sincronismo da estratégia do alinhamento progressivo com gargalo para a entrada com 500 seqüências* p. 90

4.14 *Gráfico de tempo de execução da estratégia* sendmaster . . . . p. 90

4.15 *Percentual de tempo gasto com comunicação e sincronismo da estratégia* sendmaster *para a entrada com 500 seqüências* . . . . p. 91

4.16 *Gráfico de tempo de execução da estratégia* waitall . . . . p. 92

4.17 *Gráfico de tempo de execução da estratégia* waitany . . . . p. 92

4.18 *Percentual de tempo gasto com comunicação e sincronismo da estratégia* waitall *para a entrada com 500 seqüências* . . . . p. 93

4.19 *Percentual de tempo gasto com comunicação e sincronismo da estratégia* waitany *para a entrada com 500 seqüências* . . . . p. 93

4.20 *Gráfico de tempo de execução da estratégia com* threads . . . . p. 94

4.21 *Percentual de tempo gasto com comunicação e sincronismo da estratégia com* threads *para a entrada com 500 seqüências* . . . . p. 94

4.22 *Comparação do tempo de execução das estratégias paralelas do alinhamento progressivo para a entrada com 500 seqüências* . . . . p. 95

4.23 *Comparação dos* speedups *reais das estratégias paralelas do alinhamento progressivo para a entrada com 500 seqüências* . . . . p. 95

4.24 *Comparação do ganho de desempenho das estratégias paralelas do alinhamento progressivo para a entrada com 2000 seqüências* . . . . p. 96

4.25 *Comparação do ganho de desempenho da estratégia com* threads *para as entradas com 500, 1000, 2000 e 4000 seqüências* . . . . p. 97

4.26 *Comparação do tempo de execução da estratégia com* threads *com a árvore balanceada e a árvore normal para a entrada com 1000 seqüências* . . . . p. 100

4.27 *Comparação do* speedup *real da estratégia com* threads *com a árvore balanceada e a árvore normal para a entradas com 1000 seqüências* . . . . . p. 101

4.28 *Perfil da execução em três nós da estratégia que envia os dados após todos serem computados para uma entrada de duas seqüências de aproximadamente 1000 resíduos* . . . . p. 104

4.29 *Percentual de tempo gasto com comunicação e sincronismo da estratégia que envia dados após todos serem computados para uma entrada de cinco seqüências de aproximadamente 3000 resíduos* . . . . p. 104

4.30 *Perfil da execução em três nós da estratégia que envia dados em partes para uma entrada de duas seqüências de aproximadamente 1000 resíduos* p. 105

4.31 *Percentual de tempo gasto com comunicação e sincronismo da estratégia que envia dados em pedaços para uma entrada de cinco seqüências de aproximadamente 3000 resíduos* . . . . p. 105

4.32 *Perfil da execução em três nós da estratégia que paraleliza o método de construção do caminho de alinhamento para uma entrada de duas seqüências de aproximadamente 1000 resíduos* . . . . p. 106

4.33 *Percentual de tempo gasto com comunicação e sincronismo da estratégia que paraleliza o método de construção do caminho de alinhamento para uma entrada de cinco seqüências de aproximadamente 3000 resíduos* . . . p. 107

4.34 *Comparação do tempo de execução das três estratégias paralelas do alinhamento par-a-par para entradas com sequencias de aproximadamente 1000 resíduos* . . . . p. 107

4.35 *Comparação do tempo de execução das três estratégias paralelas do alinhamento par-a-par para entradas com seqüências de aproximadamente 5000 resíduos* . . . . p. 108

4.36 *Comparação do* speedup *real da estratégia que paraleliza o método de construção do caminho de alinhamento para entradas com seqüências de aproximadamente 1000, 2000, 3000, 4000 e 5000 resíduos* . . . . p. 109

Lista de Tabelas

2.1 *Tabela de aminoácidos* . . . . p. 22

4.1 *Nível de paralelismo com o uso da árvore normal produzida pelo MUSCLE para as entradas com 500, 1000, 2000 e 4000 seqüências* . . . . p. 99

4.2 *Comparação do nível de paralelismo com o uso da árvore normal produzida pelo MUSCLE e da árvore balanceada para a entrada com 1000 seqüências* p. 101

1 Introdução

O alinhamento múltiplo de seqüências possui uma diversidade de aplicações na bioinformática, sendo considerado uma das operações mais importantes desta área. Esta operação é realizada através de ferramentas computacionais, que incorporam diferentes métodos e apresentam diferentes metodologias.

Uma das abordagens mais utilizadas para o alinhamento múltiplo de seqüências, é a abordagem progressiva. Essa abordagem foi incorporada primeiramente em 1994, na ferramenta CLUSTALW [4]. Esta ferramenta utiliza uma versão pura do algoritmo progressivo e é muito utilizada até hoje devido a sua popularidade. Entretanto, devido ao intenso avanço das pesquisas genômicas, abordagens mais complexas foram desenvolvidas e aplicadas em novas ferramentas.

O MUSCLE [1] é uma ferramenta que realiza o alinhamento múltiplo de seqüências. Segundo um artigo de revisão publicado em 2006 [2], a estratégia do MUSCLE apresenta, em relação a outras ferramentas de alinhamento existentes, um bom desempenho computacional e resultados biológicos significativamente precisos. Adicionalmente, vale destacar que desde a sua primeira versão, anunciada em 2004, até a data de publicação dessa dissertação, várias otimizações foram feitas em seus algoritmos. Isto o vêm mantendo com uma boa aceitação pela comunidade científica, visto que uma gama de trabalhos genômicos recentes referenciam o MUSCLE como ferramenta de alinhamento utilizada [5–7].

A estratégia do MUSCLE divide o alinhamento em três estágios. O primeiro consiste basicamente em efetuar um alinhamento múltiplo progressivo com métodos computacionais relativamente rápidos. O segundo estágio aprimora este alinhamento também com uma abordagem progressiva, porém com técnicas de maior precisão biológica. O terceiro, por

fim, faz um refinamento no alinhamento através de um processo iterativo. Como o resultado de um estágio é a entrada para o estágio seguinte, a ordem de execução do programa deve ser mantida.

Embora exista essa dependência no algoritmo completo da ferramenta, cada estágio é composto de métodos que possuem trechos independentes. Essa independência possibilita a paralelização em nível de estágio. A ferramenta MUSCLE atual, desenvolvida por Robert C. Edgar, executa cada estágio de forma seqüencial.

O algoritmo seqüencial, original do MUSCLE, não permite que o alinhamento de seqüências, que contenham uma quantidade muito grande de dados, seja executado em um sistema de arquitetura paralela. E o desempenho é um fator cada vez mais importante em problemas de alinhamento devido ao uso crescente de seqüências para alinhamento. Com o alinhamento, por exemplo, classificamos novas seqüências e essas seqüências são utilizadas em futuros alinhamentos. Isto o classifica como um problema de ordem crescente, e o emprego de técnicas computacionais cada vez mais eficientes é de grande relevância. A paralelização de algoritmos ultrapassa barreiras de desempenho impostas pelo algoritmo seqüêncial e por sistemas seqüências, sendo uma boa abordagem para problemas de alinhamento múltiplo de seqüências, que, em geral, lidam com uma quantidade grande de dados.

O objetivo deste trabalho é desenvolver uma versão paralelizada da ferramenta MUSCLE para sistemas distribuídos, visto que este tipo de sistema é o mais comum em laboratórios de pesquisa genômica. No decorrer do mesmo, os algoritmos de alinhamento foram primeiramente compreendidos, através de um estudo teórico de cada método implementado pelo MUSCLE e de uma investigação em seu código fonte. Em seguida, foi feito um levantamento das abordagens paralelas de alinhamento adequadas a metodologia do MUSCLE. Apenas com este conhecimento, foi possível definir abordagens eficientes de paralelização, seguindo a metodologia de Foster [8], inovadoras para esta classe de problemas. Como o sistema alvo da aplicação paralela é de memória distribuída, a implementação do paralelismo foi feita através da biblioteca MPI (*message passage interface*).

Após o estudo inicial, cada estágio da ferramenta MUSCLE foi paralelizado, contemplando a execução das seguintes fases:

Investigação do código: Busca por trechos que caracterizam possíveis pontos de paralelização. Estes trechos são métodos que utilizam conjuntos de dados independentes e que podem ser distribuídos entre os processadores para que sejam executados simultaneamente;

Verificação da viabilidade do paralelismo: Verificação da viabilidade de paralelização dos trechos de cada estágio através de diversas estratégias. Nesta fase, levantam-se as vantagens e desvantagens de cada estratégia, considerando, principalmente, o balanceamento de carga possível entre os processadores envolvidos e a vazão de dados na rede;

Implementação da estratégia definida: Implementação das estratégias de paralelismo adotadas, em que concretiza-se a inserção das estratégias de paralelização definidas anteriormente;

Realização de testes: Testes e verificação da eficiência, em que verifica-se, através de vários tipos de entrada, o aumento de desempenho obtido com a paralelização em cada estágio. Para cada entrada, os desempenhos dos algoritmos paralelos são testados, variando-se o número de nós de execução, comparando-os com seus respectivos algoritmos seqüenciais.

Ajustes no algoritmo do MUSCLE também foram feitos para estabelecer uma coordenação na execução dos três estágios paralelos.

1.1 Organização da dissertação

Além da introdução, esta dissertação está organizada em outros quatro capítulos. No capítulo dois apresenta-se toda a revisão bibliográfica e alguns conceitos necessários para a contextualização e entendimento do projeto. No capítulo três apresenta-se como decorreu o processo de seleção dos algoritmos para a paralelização, o novo algoritmo em cada etapa paralelizada e uma análise teórica das estratégias adotadas. No capítulo quatro apresenta-se os resultados obtidos. No capítulo cinco apresenta-se as considerações finais e possíveis projetos futuros.

2 *Fundamentação teórica do projeto*

Esta revisão bibliográfica abrange todo o conhecimento prévio necessário à paralelização da ferramenta MUSCLE [1] para um ambiente distribuído. Inicialmente, apresenta-se toda a metodologia da ferramenta MUSCLE, juntamente com alguns conceitos de bioinformática. Em seguida, apresenta-se todos os aspectos referentes à paralelização da ferramenta.

2.1 Genética e bioinformática

A ciência responsável pelo estudo da transmissão das características biológicas de geração para geração é a genética. Essas características são representadas através de seqüências de nucleotídeos e aminoácidos.

A informação que essas seqüências carregam representa o código genético de cada indivíduo. No processo de propagação da informação genética para seus descendentes, essas seqüências são propensas a erros.

Esses erros nem sempre são visíveis ao olho humano. Apenas uma análise nas biossequências de cada indivíduo permite-nos ver quão diferente um indivíduo está de outro indivíduo. As diferenças em suas biossequências possibilitam o surgimento de diferentes indivíduos de uma determinada espécie.

Este mecanismo da vida conduz ao estudo de métodos computacionais que procuram determinar, com maior precisão possível, similaridades e diferenças entre biossequências, através de técnicas computacionais de alinhamento de *strings*. Colocando seqüências de diferentes indivíduos alinhadas, podemos, por exemplo, identificar mutações ou regiões

conservadas. Essa identificação habilita o estudo e a classificação de novas doenças ou indivíduos. Como essas novas seqüências também são utilizadas para comparação em futuros alinhamentos, estes métodos genômicos lidam com problemas de ordem crescente e que, continuamente, necessitam de melhores recursos computacionais.

2.1.1 Código Genético: ácidos nucléicos e proteínas

Os ácidos nucléicos são as biomoléculas de maior importância do controle celular. Eles são divididos em dois tipos principais, responsáveis por carregar o código genético de um indivíduo. Esses tipos são: ácidos desoxirribonucléicos (DNA) e ácidos ribonucléicos (RNA).

O código genético corresponde a um conjunto de símbolos (tabela do código genético) e a uma gramática que contém as propriedades de cada símbolo. É o código genético que configura as estruturas de DNA, RNA e proteínas. Este processo é realizado através de duas etapas, a etapa de transcrição e a etapa de tradução.

Na etapa de transcrição, a informação do RNA é sintetizada a partir do DNA. A etapa de tradução, por sua vez, transforma a mensagem codificada do RNA em proteínas. Apesar de resultar em uma mesma informação, elas diferem entre si na linguagem utilizada. Os ácidos nucléicos utilizam a linguagem dos nucleotídeos enquanto que as proteínas utilizam a linguagem dos aminoácidos.

Os nucleotídeos são compostos por uma base nitrogenada, uma pentose e um grupo fosfato. As bases nitrogenadas definem o nucleotídeo e são divididas em duas classes: as pirimidinas e as purinas. Tanto o DNA como o RNA tem duas bases púricas: a adenina e a guanina. Eles possuem também uma pirimidina principal: a citosina. Porém, existe uma diferença entre as bases de DNA e RNA: a segunda base pirimídica é a timina no DNA e a uracila no RNA.

A representação dessas bases é normalmente feita através dos símbolos A (adenina), G (guanina), C (citosina), T (timina) e U (uracila). Conseqüentemente, os nucleotídeos que compõem a estrutura de um DNA são representados por A, G, C e T e os nucleotídeos que compõem a estrutura de um RNA são representados por A, G, C e U.

Os aminoácidos, por sua vez, são obtidos a partir da conversão de uma seqüência de nu-

cleotídeos. Na cadeia polipeptídica, uma conjunto de 3 nucleotídeos (códons) corresponde a um aminoácido. Entretanto, sabemos de antemão que são 20 os tipos de aminoácidos no total. No código genético existem também códons de finalização (UAA,UGA e UAG) que indicam à célula que a sequência de aminoácidos destinada àquela proteína acaba ali.

Como a combinação de todas as trincas de nucleotídeos resulta em 64 tipos distintos, mais de uma combinação acaba por representar um mesmo aminoácido. A tabela 2.1 mostra os aminoácidos resultantes das 64 combinações de nucleotídeos.

Primeira posição	**Segunda Posição**				**Terceira posição**
	U	**C**	**A**	**G**	
U	Phe (F)	Ser (S)	Tyr (Y)	Cys (C)	**U**
	Phe (F)	Ser (S)	Tyr (Y)	Cys (C)	**C**
	Leu (L)	Ser (S)	finalização	finalização	**A**
	Leu (L)	Ser (S)	finalização	Trp (W)	**G**
C	Leu (L)	Pro (P)	His (H)	Arg (R)	**U**
	Leu (L)	Pro (P)	His (H)	Arg (R)	**C**
	Leu (L)	Pro (P)	Gln (Q)	Arg (R)	**A**
	Leu (L)	Pro (P)	Gln (Q)	Arg (R)	**G**
A	Ile (I)	Thr (T)	Asn (N)	Ser (S)	**U**
	Ile (I)	Thr (T)	Asn (N)	Ser (S)	**C**
	Ile (I)	Thr (T)	Lys (K)	Arg (R)	**A**
	Met (M)	Thr (T)	Lys (K)	Arg (R)	**G**
G	Val (V)	Ala (A)	Asp (D)	Gly (G)	**U**
	Val (V)	Ala (A)	Asp (D)	Gly (G)	**C**
	Val (V)	Ala (A)	Glu (E)	Gly (G)	**A**
	Val (V)	Ala (A)	Glu (E)	Gly (G)	**G**

Tabela 2.1: *Tabela de aminoácidos*

2.1.2 Comparação de seqüências

Através da comparação de seqüências é possível ver os mecanismos da evolução que as características morfológicas não nos permitem. Quando olhamos para a imagem de dois animais podemos dizer aparentemente o quanto eles se parecem e quais são suas diferenças, porém não temos como saber exatamente quais as mudanças que foram feitas. Ao olhar para um grupo de seqüências alinhadas podemos dizer que eles são diferentes devido a um conjunto de aminoácidos diferentes em determinadas regiões.

A figura 2.1 mostra um conjunto de seqüências alinhadas. A partir deste alinhamento, pode-se ver as mutações ocorridas em indivíduos que supostamente vieram de ancestrais comuns. Essas mutações são vistas onde ocorrem substituições ou remoções/inserções de resíduos (caracteres). No caso das remoções/inserções, estas são representadas com lacunas (*gaps*) no alinhamento. Com o alinhamento pode-se ver também as regiões conservadas e calcular o grau de similaridade existente entre as seqüências.

```
33|1i21A|gi|28261215     -------VNVRGLEVTDLG-QLCQLLS--------------QLSTVGDVSH-----ESLM
1i21A                    --SLPDGFYIRRXEEGDLE-QVTETLKVLT-----------TVGTITPESFCKLIKYWNE
24|1i21A|gi|4115735      ---LPQGYTFRKLKLTDYDNQYLETLKVLT-----------TVGEISKEDF------TEL
1b87A                    -------MIISEFDRNN---PVLKD----------------QLSDLLRLTWPEEYGDSSA
4|1b87A|gi|78231         -------ANILTEAFNDLG----------------------------NNSWPDM--TSAT
6|1b87A|gi|1743004       ----LKK------SFLDAG----------------------------NESWGDI--KNAI
36|1i21A|gi|23473444     ---LQEGFVIRPVRPADNA-AVAEIIRSVS-----------QEHGLTAEAGYAVGDAAVD
9|1i21A|gi|6458376       -------MNIRLATSADAE-TIAQQRD--------------AMFVDMGEAAEKLARVHDS
25|1b87A|gi|23027249     ------------IEVDDLSRPAIAE----------------LLSDHMREMWEVSNPESCH
2|1b87A|gi|27376228      -------MQIRPGDTFDPR--VVAL-----------------LDHHVTAARAQTAPGSAH
```

Figura 2.1: *Exemplo de um alinhamento múltiplo de seqüências*

2.2 Alinhamento de seqüências

O alinhamento de seqüências é um procedimento fundamental na biologia computacional. É ele que nos mostra quais as regiões que variam e quais são conservadas em um conjunto de seqüências.

Quando alinhamos um conjunto de seqüências, fazemo-no, normalmente, por elas serem homólogas. Na verdade, acreditamos que elas evoluíram de um ancestral comum. Neste processo, as espécies envolvidas sofreram mutações e suas biossequências foram alteradas. Estas alterações consistem de inserções, remoções e alterações de seus resíduos, como visto na seção 2.1. Todas essas ocorrências são demonstradas no alinhamento. No caso das inserções e remoções, são inseridas lacunas no alinhamento. Estas lacunas, mais conhecidas como *gaps*, são representadas por um conjunto de *indels*, termo utilizado para representar uma lacuna em uma única coluna.

O alinhamento é feito aos pares. Cada elemento do par pode ser uma seqüência ou um grupo de seqüências alinhadas. O alinhamento par a par pode ser, portanto, entre duas seqüências, entre dois grupos ou entre um grupo e uma seqüência. O princípio do alinhamento par a par é considerar todas as formas possíveis de alinhar esses pares. Entretanto, sempre buscamos encontrar o melhor alinhamento. Este alinhamento nos mostra

as maiores similaridades e as menores diferenças e também é conhecido por alinhamento ótimo. É este o alinhamento que nos mostra as mudanças que achamos mais prováveis de terem ocorrido durante a evolução.

Para encontrarmos o alinhamento ótimo é necessário adotarmos um sistema de pontuação, a partir do qual, obtém-se, para cada alinhamento, uma certa pontuação. Aquele que no final do processo tiver a maior pontuação é o alinhamento ótimo.

O primeiro passo para definir um sistema de pontuação é associar uma pontuação para cada par de letras pertencente ao alfabeto das seqüências envolvidas. Se estivermos alinhando seqüências de DNA, o alfabeto é apenas A, C, G e T. Para proteínas temos um alfabeto de 20 letras, correspondente a todos os aminoácidos. Essas pontuações encontram-se nas matrizes de substituição e são baseadas em propriedades matemáticas e modelos estatísticos. Uma matriz de substituição é uma tabela que descreve a probabilidade de um par de resíduos (aminoácidos ou nucleotídeos) ocorrer em um alinhamento [9].

O segundo passo para definirmos um sistema de pontuação é a penalidade de *gaps*. A penalidade de um *gap* pode ser medida a partir de uma função $W(s)$, que considera s como sendo o tamanho do *gap*. Em sua forma mais simples, $W(s)$ é uma função linear da forma $W(s) = gs$, onde g é a penalidade de ocorrer um único *indel*. Entretanto, como é mais provável que um único evento crie um *gap* de vários *indels*, uma função linear não atribui uma penalidade muito confiável.

Considerar uma função que atribua pesos diferentes de acordo com o tamanho do *gap* é uma melhor solução. Uma maneira é fazer com que o *gap* inicial apresente uma penalidade maior. Neste modelo, três *indels* isolados - três *gaps* -, por exemplo, apresentam uma penalidade maior do que três *indels* consecutivos - um único *gap*. Entretanto, não existe um consenso sobre qual penalidade de *gap* utilizar. Várias análises da distribuição empírica do tamanho dos *gaps* foram feitas, oferecendo diversas estimativas para diferentes sistemas de pontuação.

Além de depender do sistema de pontuação utilizado, o alinhamento final também depende de seu algoritmo, podendo este ser ótimo ou não (sub-ótimo). Para o alinhamento de duas seqüências, utiliza-se, normalmente, métodos de programação dinâmica, como os algoritmos de Needleman e Wunsch [10] e Smith e Waterman [11]. Esses algoritmos testam

todas as soluções possíveis e sempre encontram o alinhamento ótimo. Apesar de ser um problema de elevada complexidade computacional, o uso de um conjunto pequeno de dados (apenas duas seqüências, por exemplo) viabiliza sua execução.

Para obter o alinhamento múltiplo ótimo, uma possível solução é generalizar o algoritmo de Needleman e Wunsch para o espaço multidimensional. Neste caso, uma matriz N-dimensional é computada, onde cada dimensão representa uma seqüência. A complexidade computacional deste algoritmo é $O(2^N L^N)$, onde L é o tamanho médio das seqüências e N é o número de seqüências [12].

Devido às limitações práticas de tempo e memória, a generalização do algoritmo de Needleman e Wunsch é ineficiente para grandes quantidades de dados, limitando-se a problemas pequenos. Por esta razão, uma variedade de abordagens heurísticas foram propostas na literatura. As principais delas se dividem em duas classes: a dos algoritmos progressivos e dos algoritmos iterativos.

O algoritmo progressivo, explicado na seção 2.2.2, foi inicialmente proposto em [11] e posteriormente melhorado em [13] e [14]. Ele é utilizado pela maioria das ferramentas de alinhamento múltiplo de seqüências atualmente, como o CLUSTALW, o MAFFT e o MUSCLE. A abordagem iterativa, por sua vez, inicia com uma solução sub-ótima, que pode ser obtida através de um rápido alinhamento múltiplo progressivo. Este alinhamento sub-ótimo é melhorado a cada iteração. A ferramenta MUSCLE implementa uma etapa de refinamento iterativo em sua última versão.

Por fim, têm-se também as abordagens paralelas. Como o universo de seqüências para comparação cresceu imensamente ao longo dos últimos anos, paralelizações dessas abordagens também foram propostas na literatura. Esses algoritmos dividem o alinhamento em sub-tarefas, permitindo que várias máquinas realizem operações independentes simultaneamente.

2.2.1 Alinhamento entre pares de perfis

Para o cálculo de alinhamento múltiplo de seqüências, os métodos heurísticos são mais utilizados. Esses métodos normalmente realizam várias operações de alinhamento aos pa-

res, seguindo uma ordem específica. A realização de vários alinhamentos aos pares, seja progressivamente ou iterativamente, reduz a complexidade do algoritmo, uma vez que cada operação de alinhamento utiliza uma matriz bi-dimensional. Por outro lado, os métodos exatos, em geral, realizam apenas uma única operação de alinhamento, porém envolvendo uma matriz N-dimensional. Esta única operação, embora seja ótima, ou seja, teste todos os casos e forneça com garantia a melhor solução, apresenta uma ordem de complexidade muito elevada, inviabilizando sua execução para alinhamentos com muitas seqüências.

Os alinhamentos das abordagens heurísticas podem ser feitos entre duas seqüências, dois grupos de seqüências ou entre um grupo e uma seqüência. Para representar estatisticamente um elemento qualquer do par do alinhamento, seja ele grupo ou seqüência, utiliza-se o termo perfil. No MUSCLE, o perfil é armazenado computacionalmente em uma matriz que informa a freqüência relativa com que cada resíduo ou *indel* aparece em cada coluna do alinhamento múltiplo (figura 2.2).

Colunas do Perfil

Seqüências do Perfil	1	2	3	4	5	6	7
1	C	A	A	C	T	T	T
2	C	G	A	-	T	T	-
3	C	G	-	C	A	T	T
4	C	T	A	C	T	C	T

	1	2	3	4	5	6	7
A	0	0.25	0.75	0	0.25	0	0
C	1	0	0	0.75	0	0.25	0
G	0	0.5	0	0	0	0	0
T	0	0.25	0	0	0.75	0.75	0.75
-	0	0	0.25	0.25	0	0	0.25

Figura 2.2: *Armazenamento do perfil em uma matriz*

O alinhamento múltiplo no MUSCLE é feito sempre entre dois perfis, tanto nas etapas de alinhamento progressivo, em cada nó da árvore filogenética, quanto em cada iteração do estágio iterativo. Vários métodos de alinhamento perfil/perfil foram propostos na literatura. Melhorias na qualidade do resultado biológico foram reportadas com o uso desses métodos em relação aos métodos de alinhamento seqüência/seqüência e grupo/seqüência (ver [15]).

2.2.2 Algoritmo progressivo

O método de alinhamento progressivo consiste em alinhar famílias de seqüências que estão evolutivamente relacionadas, partindo do alinhamento das seqüências mais próximas. O princípio deste método é construir uma árvore com características biológicas e ir construindo o alinhamento progressivamente de acordo com a ordem especificada por esta árvore. Essa árvore, conhecida biologicamente por árvore filogenética, é percorrida visitando sempre os nós filhos antes dos nós pais. Assim, os primeiros alinhamentos são feitos entre os nós folhas e o alinhamento múltiplo resultante é obtido na raiz da árvore.

A qualidade deste método depende de dois fatores principais: o sistema de pontuação e a árvore filogenética utilizada. O sistema de pontuação é utilizado para definir qual o melhor alinhamento entre dois perfis durante cada etapa. A árvore, por sua vez, define a ordem dos alinhamentos. Esta abordagem, apesar de não produzir com garantia o melhor alinhamento, produz ótimos resultados, principalmente levando em consideração sua baixa complexidade em relação aos métodos exatos de programação dinâmica.

2.2.3 Algoritmo iterativo

Os métodos iterativos são muito utilizados como métodos de otimização para produção de alinhamento múltiplo de seqüências. Eles são utilizados sozinhos ou combinados com outros métodos. Esses algoritmos apresentam a vantagem de serem muito simples, tanto na codificação quanto na complexidade temporal e espacial. Eles recebem este nome pois trabalham repetidamente, realinhando ou adicionando novas seqüências ao alinhamento múltiplo existente.

Embora os algoritmos de alinhamento mais utilizados sejam os progressivos, estes produzem uma série de erros inerentes. Esses erros são reduzidos através de um estágio de refinamento, feito através de algoritmos iterativos, por exemplo. Diversas abordagens de refinamento foram propostas na literatura [16]. As ferramentas de alinhamento mais precisas atualmente incluem um estágio iterativo de refinamento.

Durante o refinamento, calcula-se a pontuação objetiva do novo alinhamento. Caso obtenha-se uma pontuação maior, o novo alinhamento é mantido. Caso contrário, ele é

descartado. Este processo é feito até que uma condição previamente estabelecida seja atingida. Esta condição define o número de iterações do algoritmo.

2.3 A metodologia da ferramenta MUSCLE

Esta seção explica a metodologia da ferramenta MUSCLE, apresentando toda a relação existente entre as várias etapas de sua execução. Uma descrição dessas etapas e dos métodos aplicados é feita em detalhes na seção 2.4.

2.3.1 Funcionamento básico

A ferramenta MUSCLE é dividida em três estágios principais, que utilizam tanto a abordagem progressiva quanto a abordagem iterativa. O alinhamento progressivo é utilizada pelo MUSCLE em seus dois primeiro estágios, e é dividido basicamente em três passos: calcular a distância entre cada par de seqüências, construir uma árvore filogenética e realizar o alinhamento entre dois nós irmãos, armazenando o resultado obtido no nó pai. Este último passo é repetido progressivamente até atingir o nó raiz, finalizando com o alinhamento de todas as seqüências.

Os algoritmos no primeiro e no segundo estágio aplicam métodos diferentes, que variam o nível de complexidade computacional e a precisão do resultado. O primeiro estágio é responsável pela obtenção de um alinhamento bruto, utilizando-se métodos de baixo custo computacional. O segundo estágio aprimora o resultado do alinhamento do primeiro estágio, através de outros métodos.

As principais modificações ocorrem na primeira e na terceira etapa. No primeiro estágio, a primeira etapa é calculada utilizando o algoritmo de contagem de *k-mer*, explicado na seção 2.4.1. No segundo estágio utiliza-se um algoritmo que calcula a identidade fracional entre todos os pares alinhados. A terceira etapa, por sua vez, é otimizada no segundo estágio, realizando o alinhamento apenas nos nós que sofreram alterações, auxiliado por um método de comparação de árvores. Na segunda etapa, de construção de árvore, ambos estágios utilizam o algoritmo UPGMA, explicado na seção 2.4.2.

O terceiro estágio do MUSCLE apresenta uma abordagem iterativa e é conhecido como estágio de refinamento. Este estágio recebe como entrada o resultado do segundo estágio e a árvore guia. Para cada iteração, o algoritmo primeiro elimina uma aresta da árvore, dividindo as seqüências em dois subconjuntos. Em seguida, um perfil é extraído de cada subconjunto e as colunas que não contém elementos (apenas *gaps*) são eliminadas. Por fim, é feito um novo alinhamento e sua pontuação objetiva é calculada. Se esta pontuação aumentar, o novo alinhamento é mantido; caso contrário, ele é descartado. O processo iterativo continua até que todas as arestas da árvore sejam visitadas sem que nenhuma mudança seja mantida, ou até que um número máximo de iterações seja atingido. As arestas da árvore são visitadas em ordem decrescente de distância até a raiz, realinhando primeiro seqüências individuais, portanto grupos fortemente relacionados.

A figura 2.3 apresenta a interação entre esses três estágios e os métodos utilizados, por padrão, pela ferramenta.

2.4 Descrição das etapas e métodos do MUSCLE

Nesta seção descreve-se as principais etapas do MUSCLE, bem como os métodos utilizados em cada uma delas. Em algumas etapas é possível utilizar métodos distintos, que são selecionados através de passagem de parâmetros durante a chamada de execução do programa. Apesar de todos os métodos serem apresentados, são enfatizados os métodos adotados como padrão pela ferramenta. Estes métodos apresentam uma melhor eficiência, do ponto de vista de precisão, velocidade e uso de memória (ver [17]).

2.4.1 Medidas de similaridades e estimativas de distância

Sempre que nos deparamos com duas seqüências, é natural questionarmos quão similares elas são. Para isso, adotamos algumas medidas que permitem identificar quantas substituições ocorreram nessas seqüências desde que elas divergiram. O termo similaridade é usado, portanto, como uma medida do grau de divergência evolucionária entre duas seqüências. Obviamente, essas medidas apenas fazem sentido em seqüências que possuem um certo grau de relação.

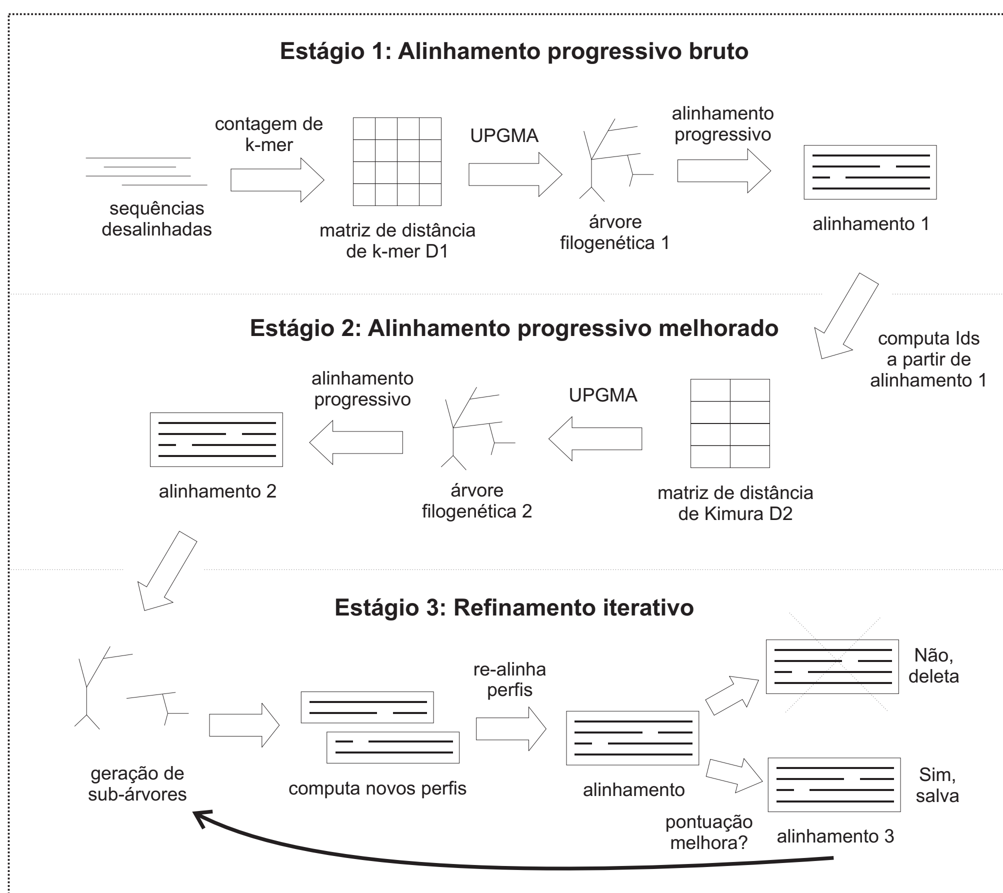

Figura 2.3: *Diagrama de fluxo do algoritmo do MUSCLE*

Este valor, entretanto, não deve ser usado diretamente como uma forma de medir a distância entre duas seqüências. Para encontrar uma estimativa boa de distância é preciso saber mais que sua similaridade. Saber apenas a taxa de similaridade entre duas seqüências não nos dá o conhecimento de todas as mudanças ocorridas durante o processo evolucionário. Sem este conhecimento, não é possível estimar, com precisão, a quantidade média de substituições ocorridas de uma seqüência para outra, a partir do ponto em que houve a divergência.

Quando realizamos o alinhamento não temos essas informações. Tudo o que sabemos é mostrado apenas nas seqüências a serem alinhadas. O uso de boas medidas de similaridade, aliadas as boas estimativas de distância, nos fornece ótimos valores aproximados. E são com esses valores aproximados que criamos as árvores filogenéticas, estruturas utilizadas no alinhamento múltiplo progressivo.

No MUSCLE, dois tipos de medidas de similaridade são utilizadas, cada um apresentando suas vantagens e desvantagens: a identidade fracional e a contagem de *k-mers*. Ambas as medidas, juntamente com suas respectivas estimativas de distância, são mostradas a seguir.

Identidade Fracional

A forma mais simples de medirmos a similaridade entre duas seqüências é colocando-as alinhadas uma sobre a outra e contando o número de posições idênticas. Este número em relação ao total de posições das seqüências é um valor fracional.

A obtenção deste valor é feita da seguinte maneira: dado o alinhamento, ignora-se todas as posições com *gaps* e, nas demais, calcula-se a quantidade de posições com resíduos iguais em relação a quantidade total de resíduos. A figura 2.4 apresenta um exemplo do cálculo da identidade fracional.

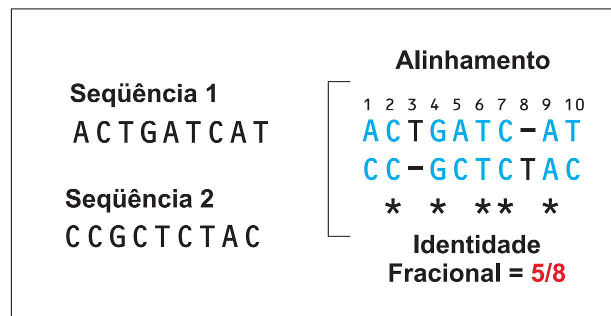

Figura 2.4: *Cálculo da identidade fracional entre duas seqüências*

Nesta figura, o par de seqüências alinhadas possui 10 posições. Dessas, apenas as posições 1, 2, 4, 5, 6, 7, 9 e 10 não possuem *gaps*. As posições 2, 4, 6, 7 e 9, por sua vez, são as únicas que possuem resíduos iguais em ambas as seqüências. Ou seja, 8 posições sem *gaps* e 5 posições com resíduos iguais. A identidade fracional entre essas seqüências é, portanto, 5/8.

A partir da identidade fracional D, obtém-se uma medida aproximada de distância através da correção de *Kimura* [17]:

$$d_{Kimura} = -\log_e(1 - D - D^2/5)$$

A árvore filogenética é então construída a partir da matriz de distância obtida.

Contagem de *k-mers*

O método de contagem de *k-mers* é um método de comparação de seqüências livres de alinhamento que calcula a similaridade entre pares de seqüências através de contagem de palavras de tamanho k.

Neste método, utiliza-se o termo *k-mer* para representar as palavras, ou k-tuplas. Adotado como opção padrão pela ferramenta MUSCLE, no seu primeiro estágio de execução, este método apresenta uma velocidade consideravelmente maior em relação aos métodos convencionais, que requerem alinhamento [18]. Seu algoritmo é de ordem $O(L)$, para seqüências de tamanho L, diferente dos algoritmos que requerem alinhamento e que apresentam ordem de complexidade $O(L^2)$.

Este algoritmo utiliza, em geral, um alfabeto um pouco diferente. Na maioria dos casos o alfabeto utilizado é uma variação do alfabeto padrão. Esses alfabetos contém símbolos que denotam classes que correspondem a duas ou mais letras diferentes (tipos de resíduos). Em [18] mostra-se como a escolha do alfabeto e do valor de k tem forte impacto no número de identidades conservadas. No MUSCLE essa escolha é feita com base em estatísticas.

O MUSCLE implementa a contagem de *k-mers* contando exatamente quantos *k-mers* apareceram em cada uma das seqüências. A fórmula para o cálculo da similaridade, entre as seqüências X e Y, é:

$$F = \sum_{\tau} \min[n_X(\tau), n_Y(\tau)] / [\min(L_X, L_Y) - k + 1]$$

Aqui τ é um *k-mer*, L_X e L_Y são os comprimentos das seqüências, e $n_X(\tau)$ e $n_Y(\tau)$ são o número de vezes que τ aparece em X e Y, respectivamente.

Segundo [17], uma boa estimativa de distância, empiricamente encontrada, é simplesmente $1-F$. No entanto, cada alfabeto apresenta estimativas específicas de acordo com o valor de k utilizado.

2.4.2 Construção da árvore

Árvores filogenéticas são árvores binárias que representam o caminho evolucionário de um conjunto de seqüências e a relação existente entre elas. Estas árvores são construídas a partir de uma matriz que mostra a distância entre todos os pares possíveis de seqüências, que é previamente calculada.

A busca da melhor árvore é um processo bastante exaustivo. A maioria dos métodos exige uma verificação de todos os arranjos possíveis para identificar a melhor solução. Há, porém, métodos exatos que não exigem a verificação de todos os arranjos, como o algoritmo *Branch-and-Bound*, primeiramente proposto por Land e Doig, em 1960 [19]. O algoritmo *Branch-and-Bound* apresenta uma abordagem paralela proposta recentemente na literatura [20] e é uma boa solução para árvores não muito grandes. Entretanto, para grandes quantidades de dados, o tempo computacional, qualquer que seja o método exato utilizado, acaba sendo impraticável e portanto longe de ser uma boa solução. Neste caso, utiliza-se métodos heurísticos para buscar a melhor hipótese para os dados através de um tempo mais curto para sua análise.

O MUSCLE implementa, como padrão, o método UPGMA. Este método é heurístico e segue um procedimento iterativo. Seus desempenhos estão fortemente relacionados com a matriz de distância utilizada. A idéia desses algoritmos é construir árvores agrupando perfis de seqüências similares. Este agrupamento ocorre entre sub-árvores e parte da junção de duas seqüências. Esta junção resulta em pequenas sub-árvores que, por sua vez, são interligadas formando sub-árvores maiores. Este processo é finalizado com a construção de uma árvore que interliga todas as seqüências do conjunto.

As junções são feitas da seguinte forma. Considere dois grupos (sub-árvores) $\boldsymbol{E}$ e $\boldsymbol{D}$ que serão unidas formando um novo grupo $\boldsymbol{P}$. Este grupo $\boldsymbol{P}$ se torna o pai de $\boldsymbol{E}$, que é filho a esquerda, e $\boldsymbol{D}$, que é filho a direita, em uma árvore binária. Em seguida, obtém-se a distância entre $\boldsymbol{P}$ e $\boldsymbol{C}$, tal que $\boldsymbol{C}$ pertence ao conjunto de seqüências. A ordem de conexão dos grupos é definida pela matriz de distância, partindo da conexão das duas seqüências mais próximas.

A distância entre $\boldsymbol{P}$ e $\boldsymbol{C}$ é utilizada para que o processo continue. Como os demais

grupos são interligados por meio de $\boldsymbol{P}$, e não mais por $\boldsymbol{E}$ e $\boldsymbol{D}$, a ordem de junção dos grupos é determinado com base na distância de $\boldsymbol{P}$ com os demais grupos.

2.4.3 Comparação de árvores

No segundo estágio da ferramenta MUSCLE, realiza-se um novo alinhamento progressivo visando-se um aprimoramento na qualidade do alinhamento final. Entretanto, a entrada para este segundo estágio é o alinhamento obtido no estágio anterior juntamente com a árvore guia.

O que acontece no segundo estágio é uma identificação dos pares alinhados que possuem uma maior identidade fracional, seguida da construção de uma nova árvore pela qual o novo alinhamento será guiado. Esta nova árvore, entretanto, pode ser muito similar à primeira (alguns ramos iguais). Conseqüentemente, o resultado do alinhamento de algumas sub-árvores acaba não sendo modificado. Para evitar desperdício de processamento durante o alinhamento neste novo estágio, indica-se na nova árvore o que deve ser refeito e o que não é necessário. Este procedimento é feito através do algoritmo de comparação de árvores.

A idéia deste algoritmo é associar progressivamente identificadores aos nós que possuem os mesmos filhos, em ambas as árvores. Isto é feito através da avaliação de pares de nós (um de cada árvore), para todos os nós das árvores, visitando sempre os nós filhos antes de seus pais. Como este algoritmo não foi selecionado para a paralelização devido a seu baixo consumo de processamento (ver capítulo 3), o mesmo não será descrito aqui. Maiores detalhes, entretanto, são encontrados no artigo do MUSCLE [17].

2.4.4 Alinhamento entre perfis

O alinhamento de perfis é feito de modo similar ao alinhamento de duas seqüências. A diferença é que cada perfil pode representar mais do que uma única seqüência. O que se faz neste caso é extrair informações estatísticas do conjunto de seqüências, reduzindo-o a valores como freqüência relativa de resíduos e grau de ocupação, para cada uma de suas colunas. Com esses valores calculados, inicia-se a busca do alinhamento ótimo entre dois perfis.

Programação dinâmica

A literatura sugere alguns algoritmos de programação dinâmica para realizar este procedimento [21–23]. Com esses alinhamentos constrói-se uma matriz, tal que cada uma de suas coordenadas corresponda a cada um dos perfis, e encontra-se, no final, a solução ótima, que corresponde ao alinhamento de maior pontuação. Este alinhamento é encontrado através da técnica do *traceback*.

A idéia desses algoritmos é encontrar o melhor alinhamento conforme os perfis vão sendo percorridos. Alinha-se os primeiros i caracteres do primeiro perfil (X^i) com os primeiros j caracteres do segundo perfil (Y^j), por meio de dois laços aninhados. Para cada alinhamento obtém-se a pontuação correspondente, através de um sistema de pontuação.

Apenas três casos podem ocorrer no final do alinhamento parcial: ou X_i e Y_j estarão alinhados um com o outro, ou X_i estará alinhado com um *gap* (remoção) ou Y_j estará alinhado com um *gap* (inserção). Para isto considere X_i e Y_j como sendo o último caractere de X^i e Y^j. Para demonstrar como a pontuação máxima é encontrada, considere também: S_{ij} a função de pontuação para alinhar X_i com Y_j, b_i^X a pontuação para uma abertura de *gap* em Y que está alinhado com X_i, t_i^X a pontuação para um *gap* de fechamento alinhado com X_i, U_{ij} o conjunto de todos os alinhamentos de X_i com Y_j, M_{ij} a pontuação do melhor alinhamento em U_{ij} terminando com um casamento (X_i e Y_j alinhados), R_{ij} a pontuação do melhor alinhamento terminando em uma remoção (X_i alinhado com um *gap*) e I_{ij} a pontuação do melhor alinhamento terminando em uma inserção (Y_j alinhado com um *gap*).

Procuramos então a máxima pontuação do alinhamento de X^i com Y^j, correspondentes as três possibilidades: casamento, remoção ou inserção. No MUSCLE, a equação que encontra a pontuação para cada uma das possibilidade é dada a seguir:

$$M_{ij} = S_{ij} + max\{M_{i-1j-1}, R_{i-1j-1} + t_{i-1}^X, I_{i-1j-1} + t_{j-1}^Y\}$$

$$R_{ij} = max\{R_{i-1j}, M_{i-1j} + b_i^X\}$$

$$I_{ij} = max\{I_{ij-1}, M_{ij-1} + b_j^Y\}$$

Durante o cálculo das pontuações, o laço externo itera sobre i e o laço interno itera sobre j. A função de pontuação é computada no laço interno e não será descrita aqui uma vez que

o paralelismo não ocorre neste nível (ver capítulo 3). Quando a pontuação máxima é obtida com um casamento, faz-se um movimento diagonal na matriz. Movimentos verticais e horizontais são feitos quando a melhor pontuação é obtida com uma inserção ou remoção. A marcação desses movimentos possibilita, no final, encontrar o melhor alinhamento, através da técnica do *traceback*. Com esta técnica, o melhor alinhamento final é designado pelo percurso na matriz de acordo com esses movimentos.

2.4.5 Pontuação objetiva

A qualidade do alinhamento final é medida através de uma função de pontuação objetiva. Esta função recebe como entrada um alinhamento e retorna sua pontuação.

O método utilizado pela ferramenta MUSCLE é a pontuação de soma de pares (SP). Este método calcula a pontuação objetiva a partir de um somatório das pontuações de todas as substituições, inserções ou remoções ocorridas entre todos os pares possíveis de seqüências alinhadas. Essas pontuações são computadas a partir de uma matriz de substituição mais a penalidade de *gaps*.

A matriz de substituição é utilizada para calcular a pontuação para cada par alinhado de resíduos. Como exemplo, considere que a posição um, em uma primeira seqüência, contém o aminoácido M e a mesma posição, em uma segunda seqüência, contém o aminoácido L. A matriz de substituição retorna a pontuação desta substituição específica e esta é utilizada no somatório que retorna a pontuação final do alinhamento.

A penalidade de *gap*, por sua vez, é computada descartando todas as colunas em que ambas as seqüências possuam um *indel*. Aplica-se, então, a penalidade $g + \lambda e$ para cada *gap*, onde g é a penalidade por *gap*, λ é o comprimento do *gap* (número de *indels*) e e é a penalidade de extensão.

No MUSCLE, utiliza-se a pontuação objetiva no estágio de refinamento. Sempre que um alinhamento é feito, este é comparado com o alinhamento anterior, prevalecendo apenas aquele com uma maior pontuação.

2.5 Aspectos da programação paralela e sistemas distribuídos

O uso da computação vem tornando-se cada vez mais intenso. Com a evolução das diversas áreas da ciência, o poder computacional exigido para a solução de diversas classes de problemas é crescente. A computação, por sua vez, sofreu um estrondoso avanço nos últimos anos. Hoje as estações de trabalho chegam a ser até cem vezes mais rápidas do que aquelas de uma década atrás. Entretanto, muitos problemas atuais são tão complexos de serem resolvidos que sua simulação numérica requer um poder computacional extraordinário, muitas vezes inviáveis de serem tratados com a tecnologia atual em uma simples estação de trabalho. Uma alternativa seria esperar um certo tempo - calculável segundo a Lei de Moore - para que o avanço tecnológico viabilizasse a solução desses problemas. No entanto, é possível trazer a solução desses problemas para os dias atuais, através do uso da computação paralela.

A computação paralela é a forma padrão utilizada pelos cientistas e engenheiros para resolver problemas da ciência que demandam alto desempenho computacional. Para que este tipo de computação seja feito é necessário o uso de máquinas envolvendo múltiplas unidades de processamento, através da computação paralela. Os computadores paralelos, entretanto, são divididos em duas classes principais: memória compartilhada e memória distribuída.

Os sistemas de memória compartilhada são sistemas altamente integrados, no qual as unidades de processamento compartilham o acesso a uma única memória global. Os sistemas de memória distribuída, por sua vez, são construídos com múltiplos computadores interconectados via rede. A interação entre os processadores dos diferentes computadores é feito através de passagem de mensagens.

Um sistema de memória distribuída poderoso e economicamente viável é o cluster *Beowulf*. Este projeto foi criado pela NASA em 1994 e é vastamente utilizado em laboratórios de pesquisa em todo o mundo. A construção de um cluster *Beowulf* é feito a partir de computadores pessoais, não especializados, portanto mais baratos. Eles são amplamente utilizados na ciência para atuarem, por exemplo, em projetos de desdobramento de proteí-

nas, análise genética, dinâmica de fluídos, dentre outros.

Para o desenvolvimento de algoritmos capazes de executarem e usufruírem ao máximo os recursos de tais máquinas paralelas, são necessários ambientes de desenvolvimento específicos. Na maioria dos casos, esses ambientes são construídos sobre uma linguagem seqüencial existente, como C e Fortran, por exemplo, adicionada de recursos para a comunicação entre processos. Essa comunicação é feita através da troca de mensagens e suas rotinas encontram-se em bibliotecas específicas.

Atualmente, a biblioteca de passagem de mensagens mais utilizada é o MPI. Além de sua distribuição estar disponível livremente pela internet, muitas são as vantagens de utilizá-la. A maior delas é a portabilidade. Escrever programas paralelos utilizando MPI permite porta-los para diferentes computadores paralelos, variando seu desempenho de acordo com o hardware utilizado. Este é o padrão utilizado para implementação e execução de programas paralelos em clusters *Beowulf* e, portanto, foi a biblioteca padrão escolhida para a implementação deste projeto.

2.5.1 O modelo tarefa/canal

A metodologia de programação de algoritmos paralelos utilizada pelo MPI é baseada no modelo tarefa/canal. Neste modelo, a computação paralela é representada como um conjunto de tarefas que interagem entre si através de canais de comunicação (figura 2.5). Por estes canais são enviadas mensagens, possibilitando as tarefas trocarem dados locais por meio de suas portas de entrada e saída. Um canal é uma fila de mensagens que conecta a porta de saída de uma tarefa com a porta de entrada de outra tarefa.

Neste modelo há uma certa distinção entre os dados que cada tarefa pode acessar. Existem os dados privados de cada tarefa, contidos inicialmente em suas memórias locais, e os dados compartilhados, cujo acesso ocorre através dos canais de comunicação. Apesar deste modelo permitir acesso a todos os dados do sistema, deve-se levar em consideração que o tempo de acesso aos dados locais é muito mais rápido quando comparado ao acesso feito pelo canal.

O tempo de execução de um programa paralelo é medido enquanto pelo menos uma

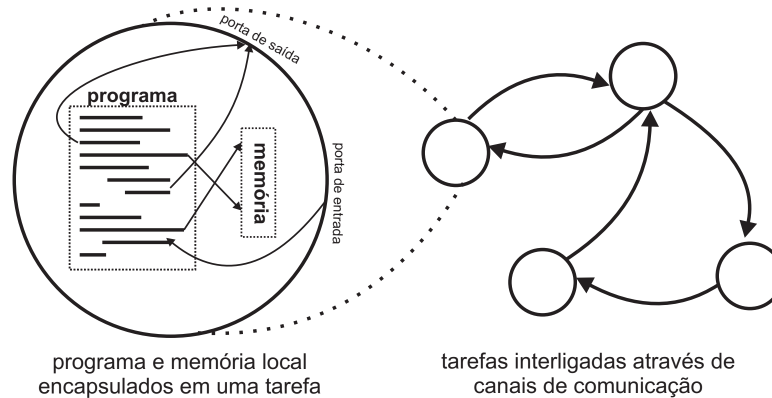

Figura 2.5: *O modelo tarefa/canal*

tarefa está ativa. Este tempo se inicia com a execução das várias tarefas simultâneas e finaliza quando a última tarefa deixa de executar.

2.5.2 Metodologia de projeto de programas paralelos

Segundo Ian Foster [8], quatro passos são importantes para o desenvolvimento de algoritmos paralelos. Esses passos são: particionamento, comunicação, aglomeração e escalonamento. Com esta metodologia é possível desenvolver algoritmos paralelos para um vasto conjunto de aplicações. Além disto, tal metodologia preza pela portabilidade dos algoritmos, uma vez que não são consideradas características dependentes de arquitetura. Estas são, quando necessárias, consideradas apenas em passos posteriores.

Particionamento

O particionamento consiste em dividir os dados e a computação em vários pedaços. Um bom particionamento efetua essa divisão em pequenos pedaços. Essa divisão ocorre através de uma abordagem centrada nos dados ou de uma abordagem centrada na computação. Independente da decomposição escolhida, esses pedaços recebem o nome de tarefas primitivas.

A abordagem centrada nos dados consiste em dividir os dados em partes e, em seguida, determinar como associar o processamento com os dados. Ela também é conhecida como decomposição do domínio. A abordagem centrada na computação coloca a ênfase na divisão

da computação envolvida no problema, dividindo o processamento em várias partes e, em seguida, determinando a associação dos dados com o processamento. Esta abordagem é também conhecida como decomposição funcional.

Como um guia na decisão de qual particionamento escolher, Foster propôs uma lista de verificação capaz de medir a qualidade do particionamento. Essa lista é composta de quatro atributos [8]:

- O número de tarefas primitivas é pelo menos uma ordem maior que o número de processadores;
- Processamento e armazenamento de estrutura de dados redundantes são minimizados;
- Tarefas primitivas são do mesmo tamanho;
- O número de tarefas aumenta conforme aumenta o problema.

Comunicação

Identificadas as partições, o passo seguinte preocupa-se em definir a forma com a qual essas partes trocam informações. Foster define quatro categorias de comunicações possíveis de serem adotadas na implementação de um algoritmo paralelo. A comunicação pode ser local ou global, dependendo do número de tarefas envolvidas na comunicação, estruturada ou não-estruturada, dependendo da forma como elas interagem entre si, estática ou dinâmica, dependendo se a definição das tarefas envolvidas é feita ou não em tempo de execução e, por fim, síncrona ou assíncrona, dependendo da existência ou não de coordenação entre as tarefas comunicantes.

Na comunicação local, cria-se um canal quando uma tarefa necessita de dados de tarefas vizinhas, pertencentes a um determinado grupo. A comunicação global, porém, não possui limites na troca de dados, podendo cada tarefa se comunicar arbitrariamente.

A estruturação, por sua vez, depende de uma ordem na interação entre as tarefas. Caso as tarefas formem uma estrutura regular, como uma árvore, por exemplo, e a comunicação seja estabelecida respeitando tal estrutura, dizemos que a comunicação é estruturada. Por outro lado, o modelo não-estruturado permite que as tarefas formem grafos arbitrários.

Na comunicação estática, a identidade dos parceiros comunicantes é fixa, ou seja, a comunicação ocorre sempre entre as mesmas tarefas. Já na comunicação dinâmica, os parceiros comunicantes não são previamente definidos, dependendo dos valores calculados em tempo de execução.

Por fim, a comunicação considera o sincronismo entre as tarefas comunicantes. Caso a comunicação entre as tarefas seja feita coordenadamente a comunicação é síncrona. Em contrapartida, a comunicação é assíncrona quando não existe nenhum tipo de coordenação.

Em projeto de algoritmos paralelos, define-se abordagens de comunicação de modo a manter o menor *overhead* possível. Para isto, Foster sugere a seguinte lista de verificação [8]:

- As operações de comunicação estão balanceadas entre as tarefas;
- Cada tarefa se comunica apenas com um pequeno número de vizinhos;
- Tarefas podem realizar sua comunicação concorrentemente;
- Tarefas podem realizar seu processamento concorrentemente.

Aglomeração

Nesta etapa definimos a forma pela qual as tarefas primitivas encontradas devem ser agrupadas. Para isto, três pontos conflituosos devem ser considerados.

O primeiro deles é o aumento da granularidade. Quanto maior a granularidade obtida, menor o custo com a comunicação. Conseqüentemente, menor o *overhead* gerado com a troca de mensagens. Obviamente, uma forma de diminuir a comunicação, é agrupando ao máximo as tarefas primitivas que se comunicam. Isto pode ser feito através da combinação de grupos de tarefas emissoras e receptoras. Outra forma, entretanto, considera o tamanho da mensagem. O envio de menos mensagens, porém mais longas, demanda menos tempo do que o envio de mais mensagens, porém mais curtas, com o mesmo comprimento total.

O segundo ponto é manter a escalabilidade do algoritmo. Aqui, considera-se que o programa possa ser portado para um sistema que possua um número maior ou menor de

processadores. Se a aglomeração combinar muitas tarefas primitivas, o número de tarefas distribuídas entre os processadores pode não atingir uma boa eficiência quando o programa for executado em um sistema com um conjunto muito grande de processadores. Portanto, a aglomeração deve ser feita de tal forma que a granularidade seja controlada por um parâmetro em tempo de execução ou compilação. Dessa forma, o número de tarefas pode ser adaptado ao número de processadores disponíveis.

Por fim, busca-se obter uma redução de custos de engenharia de software. Enquanto um número baixo de aglomerados pode ser ineficiente em sistemas com um grande número de processadores, o tempo e os gastos envolvidos com o desenvolvimento do programa paralelo são minimizados. Um número baixo de aglomerados utiliza muito código seqüencial. Em contrapartida, muitos aglomerados implicam em grandes alterações no código, aumentando o tempo e os custos para sua confecção.

A qualidade de uma aglomeração pode ser medida através de uma lista de verificação proposta por Foster. Esta lista constitui-se de sete itens [8]:

- A aglomeração aumentou a localidade da comunicação;
- Processamento replicado toma menos tempo do que a comunicação que substituiu;
- A quantidade de dados replicados é pequeno o suficiente para permitir o aumento de escala do algoritmo;
- Tarefas aglomeradas têm custos de processamento e comunicação similares;
- O número de tarefas é uma função do tamanho do problema;
- O número de tarefas é o menor possível mas pelo menos tão grande quanto o número de processadores;
- O custo representado pela escolha de uma certa aglomeração para economizar com o uso de um algoritmo seqüencial existente é compensador.

Escalonamento

O escalonamento é o processo de atribuir tarefas aos processadores. Para isto, dois são os objetivos a serem alcançados: maximizar a utilização dos processadores e minimizar a comunicação entre eles.

Por utilização de processadores entende-se o tempo que estes se mantém ativos em relação ao tempo total de execução do programa. Ela é maximizada quando a computação é igualmente balanceada, ou seja, quando as tarefas em todos os processadores iniciam e terminam ao mesmo tempo. Portanto, ocorre uma queda no desempenho sempre que alguns processadores estiverem ociosos enquanto outros estiverem ocupados.

A comunicação entre os processadores é minimizada quando duas tarefas conectadas por um canal são escalonadas para o mesmo processador. Quando isto ocorre o acesso aos dados é mais rápido, pois estes se encontram na memória local do sistema.

Ambos os objetivos, entretanto, conflitam entre si. Como exemplo, suponha que p processadores encontram-se disponíveis. Ao mapear todas as tarefas para um único processador, o custo de comunicação entre os processadores é zero. A utilização dos processadores, porém, é reduzida a $1/p$. Neste caso, recomenda-se encontrar um ponto de equilíbrio.

A lista de verificação a seguir, proposta por Foster, ajuda na obtenção de um bom escalonamento [8]:

- Projetos baseados em uma tarefa por processador e múltiplas tarefas por processador foram considerados;
- Alocações estática e dinâmica das tarefas aos processadores foram avaliadas;
- Se a alocação dinâmica de tarefas for escolhida, o controlador não é um gargalo de desempenho;
- Se a alocação estática de tarefas for escolhida, o número de tarefas é pelo menos 10 vezes o número de processadores.

2.5.3 MPI - Message Passage Interface

Muitas linguagens paralelas foram propostas nos últimos 40 anos. Entre elas encontramos muitas de alto-nível, simplificando a forma de controlar o paralelismo. Nenhuma delas, porém, foi escolhida como de uso padrão para um caso geral. Conseqüentemente, muitas linguagens de alto-nível continuam sendo desenvolvidas, com funções que realizam passagem de mensagens entre processos.

O MPI é a especificação de passagem de mensagem mais popular, que suporta programação paralela, e que executa em sistemas de memória distribuída. Ele contém uma série de rotinas que definem a forma como os computadores devem se comunicar para que o paralelismo seja feito. Essas rotinas podem ser utilizadas em Fortran, C, C++ ou qualquer outra linguagem que seja capaz de fazer uma interface com a biblioteca MPI.

Nesta seção é apresentada o modelo de passagem de mensagem, mostrando de forma breve como os processos são tratados e como a comunicação é estabelecida. Também é apresentada a implementação do MPI utilizada neste projeto.

O modelo de passagem de mensagens

Este modelo é similar ao modelo tarefa/canal. Neste modelo, o hardware assume a forma ilustrada na figura 2.6. Como pode ser visto, cada estação possui seu próprio processador e memória, sendo que cada processador tem acesso direto apenas as instruções e dados de sua própria memória. A interconexão de rede é utilizada para que seja possível a troca de mensagens entre os processadores. Assim, o processador A pode enviar uma mensagem contendo alguns de seus valores de dados locais para um processador B, dando a ele acesso indireto a esses valores.

Esta comunicação na verdade é feita entre processos. O que é uma tarefa no modelo tarefa/canal é um processo neste modelo. Um processador pode ter mais de um processo e a comunicação é estabelecida com o processo com o qual ele deseja se comunicar. A informação é então passada ao processo do processador correspondente, em um ambiente transparente ao programador. Dessa forma, cada processo pode se comunicar com todos os outros processos. Estes processos possuem um identificador único. Eles realizam diferentes

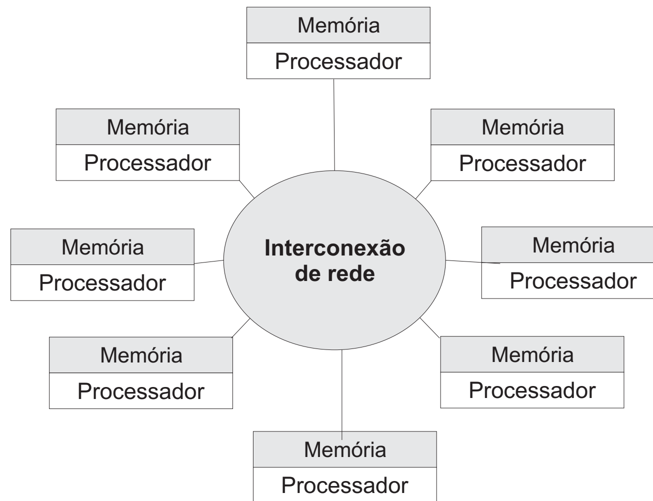

Figura 2.6: *O modelo de passagem de mensagens*

operações, de acordo com as especificações, ao longo da execução do programa.

A comunicação, por sua vez, é feita através de um canal virtual. É nele que todas as mensagens são enviadas, de um processo para outro. Essas mensagens podem ser anônimas, podendo ser recebidas por qualquer processo, ou nomeadas, sendo enviadas apenas ao processo que possua um determinado identificador. Elas, entretanto, não são apenas utilizadas para troca de dados. Mecanismos de sincronização também são implementados neste modelo. Durante a transferência síncrona, nem o processo emissor nem o receptor continuam seu processamento até que a mensagem seja totalmente transferida. Por este motivo, até mesmo mensagens sem conteúdo possuem um significado.

Open MPI

O OpenMPI [25] é uma implementação do MPI que implementa as especificações do MPI-1.2 [26] e MPI-2 [27] e suporta aplicações multi*threads*. Outras versões do MPI como o LAM/MPI não oferecem este suporte.

O MPICH2 é uma outra implementação do MPI-2 e que também suporta aplicações multi*threads*. Entretanto, o Open MPI apresenta uma menor latência e utiliza melhor a largura de banda disponível [28].

Uma vez que utilizamos *threads* em algumas estratégias definidas e devido às vantagens

de desempenho apresentadas, optamos por adotar o Open MPI como a implementação utilizada para testes. Outras implementações, entretanto, podem ser utilizadas, desde que satisfaçam os requisitos de execução do algoritmo quanto ao uso ou não de *threads*.

2.5.4 Medidas de desempenho

A medição de desempenho utilizada nesse trabalho pretende quantificar o trabalho que um computador consegue realizar por unidade de tempo. Essa tarefa exige um planejamento que considere não apenas a técnica em si, mas também outros fatores, como:

- qual estágio do ciclo de vida do desenvolvimento da aplicação deve se aplicar a medição e qual técnica deve ser utilizada;
- qual a capacidade da técnica em prover as métricas desejadas;
- qual a validade e a confiabilidade dos resultados obtidos com a técnica;
- qual o custo e o esforço investido para cada estratégia no contexto de recursos computacionais e humanos.

De antemão, sabemos que a computação paralela apresenta duas vantagens principais. Primeiro, com mais memórias e recursos de armazenamento do que os disponíveis em uma única máquina, um computador paralelo pode resolver instâncias maiores de um mesmo problema. Segundo, com mais processadores agregados a sub-conjuntos de memórias, um computador paralelo pode resolver problemas mais rapidamente.

Em geral, as métricas aplicadas em algoritmos paralelos visam medir os ganhos de desempenho sob o ponto de vista de tempo de execução. Entretanto, tais medições, de uma forma indireta, acabam por mostrar o impacto do paralelismo na capacidade de resolver instâncias maiores de um problema. Ao medir-se a escalabilidade de um algoritmo, por exemplo, utiliza-se entradas de dados de diferentes tamanhos. Dependendo da estratégia adotada para verificar-se a escalabilidade, pode-se ver claramente como o aumento do número de nós habilita a execução de instâncias maiores.

Na computação paralela, portanto, o foco não está apenas no tempo de execução. Também devem ser considerados a forma como recursos adicionais (tipicamente processadores) afetam o tempo de execução e a capacidade de solução de problemas maiores. Dessa forma, perguntas freqüentes como 'utilizar duas vezes mais processadores reduz na metade o tempo de execução?' ou 'qual o número de processadores máximo para que a computação use os recursos eficientemente?' podem ser facilmente respondidas. Para tal, utiliza-se em geral as medidas de *speedup*.

Speedup

Geralmente o *speedup* é definido como o tempo necessário para resolver o problema em um único processador sobre o tempo requerido para resolver o mesmo problema em um sistema paralelo, com p processadores. Dependendo da forma que este tempo seqüencial é medido, podemos distinguir *speedups* absoluto, real e relativo [29]. No *speedup* real, o tempo seqüencial é obtido pela execução do programa seqüencial mais eficiente em um único processador do computador paralelo. No *speedup* absoluto, o tempo seqüencial é obtido pela execução do programa seqüencial mais eficiente no computador seqüencial mais rápido existente. No *speedup* relativo, o tempo seqüencial é obtido pela execução do programa paralelo em um único processador do computador paralelo.

O *speedup* absoluto é uma medida pouco utilizada devido a dificuldade de utilizar o computador seqüencial mais rápido existente. A evolução rápida de novos processadores dificulta também esta comparação. O *speedup* real e o relativo, por outro lado, apresentam o problema de limitação de memória. Sem o uso de computadores extremamente potentes, pode-se tornar inviável a obtenção do tempo de execução em um único processador.

Nos algoritmos implementados, foram feitas análises de *benchmarks* (não analíticas) através do *speedup* real e relativo. A medida utilizada é indicada em cada teste. Nos casos em que instâncias muito grandes são utilizadas e que não podem ser executadas em um único processador, uma alternativa foi calcular o ganho de desempenho do algoritmo paralelo a partir do menor número de nós que habilita a execução do problema. Dessa forma mantém-se a característica do *speedup* relativo, provendo informações sobre a eficiência do algoritmo paralelo em solucionar o problema [30] e possibilitando que certas degradações

e variações do desempenho do algoritmo sejam reveladas.

Degradação do paralelismo

Dois importantes motivos para a degradação do paralelismo durante a sua execução são o desbalanceamento de carga e os custos com a comunicação. Enquanto o desbalanceamento de carga depende apenas da aplicação, o custo de comunicação depende do processo de comunicação, da latência, da aplicação e do hardware que está sendo utilizado. Entende-se pela aplicação uma função que considera o algoritmo, a instância do problema e o número de processadores.

A fim de obter uma medição precisa sobre a degradação do paralelismo, ambos os parâmetros foram medidos durante a fase de testes. Para medir os custos com a comunicação, realizou-se uma instrumentação no código através do MPE2, um pacote de software do MPI para programadores que contém uma API para análise de desempenho [31] e uma poderosa ferramenta de visualização gráfica, o Jumpshot [32–35]. Com o MPE2 é possível realizar medições de tempo em todas as funções do MPI, sejam funções de comunicação ou sincronismo. Dessa forma, obteve-se claramente o tempo gasto com comunicação e a latência em cada nó utilizado.

Para medir o desbalanceamento de carga, definiu-se um parâmetro adotado como nível de paralelismo e que é explicado na seção 4.3.2. Este parâmetro, junto com os gráficos de comunicação e sincronismo em cada nó, possibilita ver o que, no desbalanceamento da carga, é inerente ao problema, e o que é devido à estratégia de escalonamento adotada. O nível de paralelismo foi obtido através de uma instrumentação manual implementada durante a fase de testes do projeto.

2.6 Abordagens paralelas de alinhamento

Com o crescimento da área de tecnologia da informação, métodos digitais, cada vez mais eficientes, vêm se tornando indispensáveis para a solução de diversas classes de problemas. São vários os campos da ciência que lidam com problemas de gerenciamento e armazenamento de grandes quantidades de dados e os recursos de informática viabilizam a

extração de informações úteis desses dados. O campo da genômica é um deles, onde os dados tomam a forma de biossequências, estruturas tridimensionais, *motifs*, etc. O problema é que enquanto a quantidade de dados de projetos genômicos cresce de forma constante e exponencial, nossa habilidade de absorver e processar essas informações permanece praticamente constante [36].

Existem várias formas de ultrapassar esses limites da computação. Uma maneira é a criação de algoritmos heurísticos, onde busca-se uma solução aproximada ao invés da melhor solução. Entretanto, a forma mais promissora é a computação paralela. Na computação paralela vários processadores são utilizados no processamento de uma tarefa que é inviável de ser resolvida em um simples processador.

Os algoritmos paralelos habilitam a computação massiva de dados dividindo a tarefa em vários processos que podem ser executados concorrentemente. Levando-se em conta as características apresentadas pela metodologia de Ian Foster [8], várias estratégias de paralelismo foram definidas para os vários problemas de bioinformática.

Dentre os problemas da bioinformática, encontram-se os problemas de alinhamento múltiplo de seqüências. Um algoritmo de alinhamento múltiplo pode ser feito seguindo várias abordagens, produzindo resultados ótimos ou não. Uma ferramenta de alinhamento, por sua vez, pode misturar as abordagens existentes, como é o caso da ferramenta MUSCLE.

As abordagens paralelas propostas são inúmeras. Alguns trabalhos propõem ferramentas inteiras paralelizadas, como a versão do CLUSTALW para sistemas distribuídos, utilizando o MPI [37]. Outros trabalhos propõem o paralelismo apenas de etapas específicas.

Muitas pesquisas envolvendo o paralelismo de técnicas de alinhamentos de seqüências vêm sendo feitas, tanto no Brasil [38, 39] quanto em outras partes do mundo [40, 41]. O levantamento bibliográfico aqui apresentado, entretanto, tem seu foco apenas no que há de mais relevante nas pesquisas que envolvem a paralelização de métodos similares aos da ferramenta MUSCLE. Uma análise aprofundada é feita nesses métodos, visando auxiliar no desenvolvimento de novas estratégias paralelas.

2.6.1 CLUSTALW-MPI

O CLUSTALW realiza o alinhamento progressivo em três etapas: cálculo das distâncias, construção da árvore filogenética e o alinhamento de perfis, seguindo a ordem especificada pela árvore. No CLUSTALW-MPI [37], todas essas etapas foram paralelizadas, reduzindo o tempo de execução da ferramenta. A ferramenta utiliza a biblioteca MPI e executa em sistemas de memória distribuída.

No CLUSTALW, o cálculo da matriz de distância é feito utilizando-se a identidade fracional. Realiza-se nesta etapa o alinhamento par a par entre todas as seqüências. Uma vez que esses alinhamentos são independentes entre si, a implementação paralela da identidade fracional é bastante simples. O algoritmo proposto considera cada alinhamento como uma única tarefa. As tarefas então são aglomeradas em blocos maiores, formando p processos distintos, cada um executando em uma máquina do cluster.

Com a matriz de distância calculada, constrói-se a árvore filogenética. O método utilizado no CLUSTALW é *neighbor-joining*. No artigo publicado da ferramenta [4], nada está descrito sobre sua implementação paralela. Entretanto, uma publicação mais recente [42] sugere uma implementação paralela mais eficiente deste método.

Por fim, a etapa de alinhamento mistura uma abordagem de paralelismo que, segundo Li [37], está em duas granularidades. O paralelismo em maior granularidade é feito em todos os nós folhas da árvore. A eficiência obviamente depende da topologia da árvore. O paralelismo do alinhamento nos nós intermediários é um paralelismo de menor granularidade uma vez que esses nós dependem dos resultados de seus nós filhos.

2.6.2 MUSCLE-SMP

O MUSCLE-SMP [3] é a primeira implementação paralela da ferramenta MUSCLE e foi desenvolvida para executar em um sistema multiprocessado de memória compartilhada. Essa paralelização foi feita através da biblioteca OpenMP e considera apenas os dois estágios progressivos da ferramenta. O estágio iterativo não é paralelizado.

O primeiro estágio progressivo é relativamente rápido de executar. Como visto na seção 2.3.1, ele utiliza um método de baixa complexidade para construir a matriz de distância. O

alinhamento é então feito de baixo para cima, seguindo uma árvore previamente construída. Este alinhamento implica na dependência entre um nó pai e seus nós filhos, de tal forma que um nó só possa ser alinhado quando seus dois nós filhos estiverem alinhados. Esta dependência impede que seja feito um paralelismo em grão grosso no código fonte. No MUSCLE-SMP utiliza-se um modelo que considera uma fila de tarefas. Essas tarefas são os alinhamentos e a tarefa do alinhamento de um nó pai só é habilitada quando a tarefa de alinhamento de seus nós filhos for finalizada. Dessa forma, as tarefas habilitadas não têm nenhuma dependência entre si, podendo executar paralelamente.

O segundo estágio, por sua vez, concentra quase toda a sua computação na construção de uma nova matriz de distância. O método utilizado neste estágio é o da identidade fracional. O alinhamento, por sua vez, é menos intenso. Ele é feito apenas nos nós da árvore que foram modificados. Uma vez que quase toda a computação está na construção da matriz de distância, optou-se em paralelizar, neste estágio, apenas este método. Este método realiza um conjunto de operações em todos os pares de seqüências, através de dois laços aninhados, como apresentado no pseudo-código seguinte:

```
for ( i = 1 ; i < num_seq ; ++i)
   for ( j = 0 ; j < i ; ++i)
      Calcula_elemento_da_Matriz(i,j)
```

Neste código, as variáveis do laço interno dependem das variáveis do laço externo. Esta dependência acarretaria em um paralelismo com forte desbalanceamento de carga, caso um algoritmo de balanceamento de carga estático fosse utilizado. O que o MUSCLE-SMP faz, neste caso, é atribuir dinamicamente cada par de laços a seu processador correspondente. Deng [3] omite detalhes dessa paralelização, dizendo apenas que a mesma é feita através da diretiva `task` do OpenMP.

2.6.3 Técnicas paralelas do alinhamento progressivo

As estratégias de alinhamento paralelo mais adotadas utilizam uma abordagem que considera o alinhamento de cada nó da árvore como uma tarefa primitiva, utiliza o modelo de comunicação mestre-escravo e utiliza uma estratégia de escalonamento dinâmico. Nesta

abordagem conforme as tarefas vão se tornando prontas para o processamento - ou seja, quando seus dados dependentes estão disponíveis - e conforme os processadores se tornam disponíveis, novos escalonamentos são realizados. O fluxograma da figura 2.7 mostra, de forma geral, como é feito este escalonamento no processador mestre.

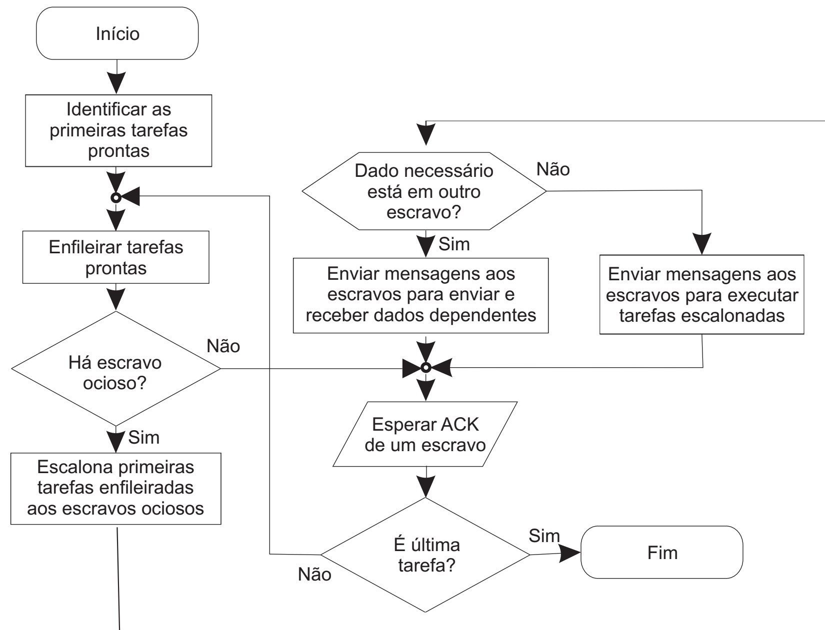

Figura 2.7: *Fluxograma do algoritmo do processo mestre do alinhamento progressivo paralelo com escalonamento dinâmico*

Cada processador escravo, por sua vez, executa a tarefa a ele associada e armazena o alinhamento parcial em sua memória local. Caso uma tarefa requisite como entrada alinhamentos que residem em outros escravos, o mestre envia comandos para estes escravos solicitando a transferência de seus alinhamentos para o escravo que está necessitando. O fluxograma do algoritmo que é executado nos processadores escravos, por sua vez, é ilustrado na figura 2.8.

Algumas estratégias propõem, sobre esta abordagem, a manutenção das tarefas ativas em listas que são mapeadas de acordo com sua prioridade. A estratégia proposta em [12] é um exemplo. Nela, utiliza-se uma lista com todas as tarefas ativas em um certo instante e cujas prioridades são definidas a partir das informações de todas as tarefas com as quais uma certa tarefa está relacionada.

O relacionamento das tarefas é obtido no processo mestre através de uma árvore de

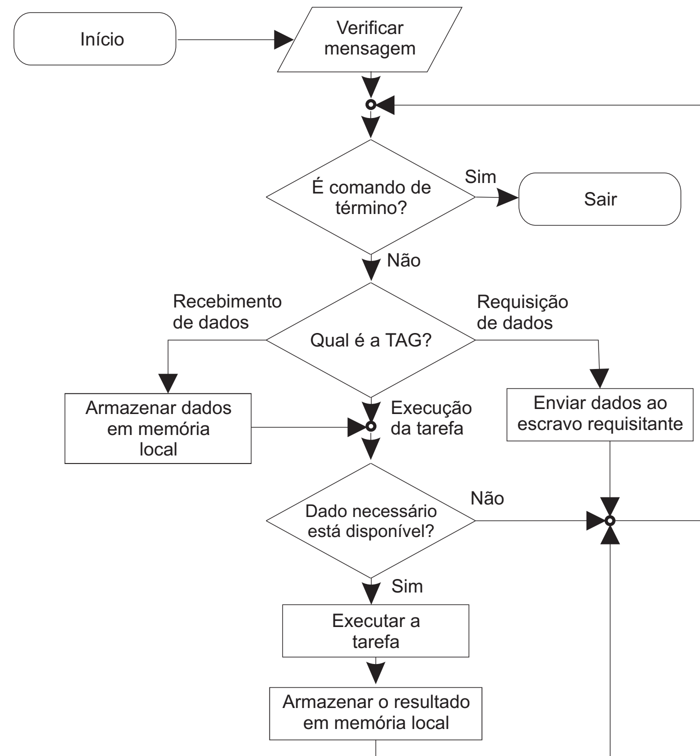

Figura 2.8: *Fluxograma do algoritmo do processo escravo do alinhamento progressivo paralelo com escalonamento dinâmico*

tarefas. Esta árvore é baseada na árvore filogenética, onde cada nó interno dessa árvore é uma tarefa na árvore de tarefas. A relação de dependência entre as tarefas corresponde aos galhos da árvore. Nós que residem em diferentes galhos são independentes. A figura 2.9 ilustra uma árvore filogenética e a árvore de tarefas obtida.

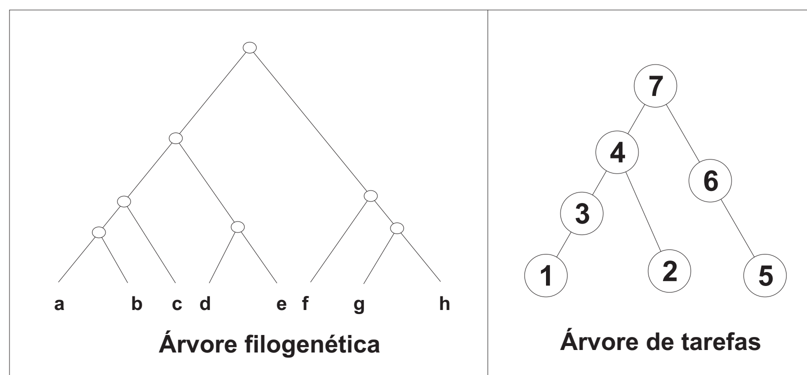

Figura 2.9: *Mapeamento da árvore filogenética para a árvore de tarefas*

A partir da árvore de tarefas e dos índices das tarefas finalizadas, mantém-se uma

lista de tarefas prontas (L_R) no nó mestre. Uma tarefa pronta é uma tarefa folha ou uma tarefa com todas as tarefas filhas finalizadas. A estratégia proposta em [12] define as prioridades dessas tarefas segundo uma equação que considera o seu custo de execução e de comunicação e de todas suas tarefas ascendentes. Estes custos são calculados segundo uma fórmula que considera o comprimento das seqüências e o número de seqüências da tarefa.

Nesta estratégia, quando uma tarefa é finalizada, o processador mestre atualiza a lista de tarefas prontas com as prioridades. A tarefa de maior prioridade é escalonada a um escravo ocioso.

2.6.4 Técnicas paralelas do alinhamento par-a-par

As técnicas paralelas do alinhamento par-a-par realizam um particionamento em nível de matriz de programação dinâmica. A abordagem de particionamento utilizada nessas técnicas é a de decomposição do domínio. A decomposição funcional não é utilizada uma vez que para cada elemento da matriz o mesmo processamento é realizado e este processamento é muito rápido, portanto inviável de ser paralelizado.

A construção de uma matriz de programação dinâmica é feita através de um processo recursivo, não sendo possível paraleliza-la a grosso modo. Entretanto, em uma menor granularidade, a decomposição dos dados pode ser feita. Essa decomposição deve levar em conta como os elementos da matriz se relacionam.

Lopes e Moritz [43] investigam essa relação e mostram que uma matriz pode ser particionada em três regiões principais. Essas regiões, entretanto, são quase totalmente independentes. Uma delas deve ser parcialmente computada para que as demais sejam computadas. A figura 2.10 mostra esse particionamento.

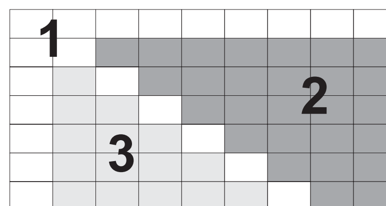

Figura 2.10: *Particionamento da matriz de programação dinâmica em três regiões*

Seguindo esta estratégia é possível dividir o processo de construção dessa matriz em até três processos. A região 1 (células em branco) é a que deve ser parcialmente computada antes das demais. Primeiro computa-se todas as células desta região que são bordas da matriz. Em seguida, computa-se a célula M(2,2). Neste ponto é possível iniciar dois processos paralelos para computar as regiões 2 e 3 ao mesmo tempo. O primeiro processo, então, toma o controle da divisão das tarefas, atribui a dois outros processos a computação das células das regiões 2 e 3 e termina de computar as células da região 1.

Uma técnica mais recente é a *block-based wavefront* [44]. Essa técnica foi utilizada como base na definição da estratégia de paralelismo do alinhamento par-a-par do MUSCLE, aplicável tanto no estágio progressivo quanto no estágio iterativo.

O princípio desta técnica é dividir a matriz de programação dinâmica verticalmente em p grupos, onde p é o número de processadores, e associar cada processador à cada grupo. Cada grupo contém em média o mesmo número de colunas da matriz. As colunas em cada processador são então agrupadas em blocos de altura a. Este valor deve ser ajustado de acordo com o número de linhas da matriz. Portanto, a computação de um dado bloco requer apenas a última coluna do bloco imediatamente à esquerda e o elemento da diagonal principal - o último elemento da última coluna do bloco da diagonal superior esquerda -, totalizando $a+1$ elementos. O alinhamento paralelo então é feito calculando-se os blocos na ordem anti-diagonal, partindo-se do bloco superior esquerdo até o bloco inferior direito. A figura 2.11 mostra um exemplo para uma matriz 16x16 distribuída em quatro processadores.

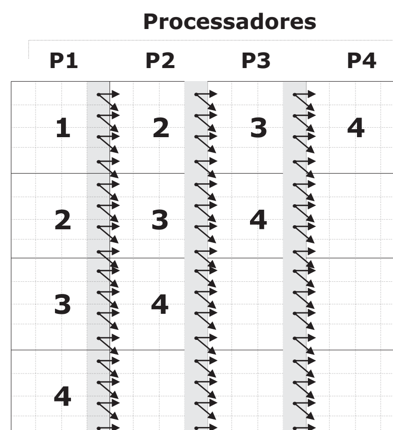

Figura 2.11: *Estratégia* block-based wavefront

3 Detalhamento e desenvolvimento do projeto

A paralelização da ferramenta MUSCLE consiste na paralelização de um conjunto de métodos que a compõem e na interligação desses métodos. Cada método é empregado em uma etapa específica do MUSCLE.

Em estágios distintos do MUSCLE, uma mesma etapa pode se repetir. Entretanto, métodos alternativos podem ser empregados. É o que ocorre na primeira etapa do primeiro e do segundo estágio do MUSCLE, responsável pelo cálculo da matriz de distância. Para a construção da matriz de distância, utiliza-se o método de contagem de *k-mers* no primeiro estágio e o método da identidade fracional no segundo estágio. As demais etapas, apesar de contemplarem pequenas alterações em suas implementações, adaptando o algoritmo ao tipo de entrada e saída em cada estágio, utilizam basicamente o mesmo método.

Nem todos os métodos do MUSCLE, entretanto, foram paralelizados. Para decidir em quais o paralelismo é vantajoso, aplicou-se, inicialmente, testes para verificar a viabilidade de paralelizar cada um deles. O método UPGMA, utilizado para a construção de árvores, foi um dos não paralelizados. Mesmo existindo referências sobre possíveis formas de paraleliza-lo, este método apresenta uma complexidade espacial e temporal muito reduzida, sendo, portanto, extremamente rápido. Para entradas com milhares de seqüências, o algoritmo seqüencial do UPGMA levou apenas alguns milisegundos para executar. Pelo mesmo motivo, o algoritmo de comparação de árvores não foi paralelizado.

Embora todos os métodos apresentem uma abordagem mestre-escravo, que contempla a característica de reunir no mestre os dados gerados em todas as máquinas, procurou-se maximizar o uso da memória distribuída, descentralizando os dados na memória do

sistema. Dessa forma, a menor quantidade possível de dados foi coletada, mantendo uma menor quantidade de estruturas no mestre e possibilitando a execução de problemas muito grandes sem a exigência de *front-ends* extremamente potentes em recursos de memória. Para isto, as entradas e saídas são interligadas através de uma estrutura que armazena a posição de todos os dados no sistema distribuído.

Essa interligação proporciona além de um bom uso da memória disponível um melhor tempo de processamento, uma vez que reduz-se o custo com a comunicação. Adicionalmente, todos os métodos adotam técnicas dinâmicas de escalonamento, tomando decisões em tempo de execução que consideram medidas como custos de comunicação, balanceamento de carga, dependência dos dados e ocupação de processadores. Várias abordagens são propostas e implementadas, e as vantagens e desvantagens, assim como uma descrição em alto nível da implementação de cada uma delas, são apresentadas nas seções seguintes.

3.1 Paralelização do método de contagem de *k-mers*

O algoritmo de contagem de *k-mers* é dividido em duas etapas pelo MUSCLE. A primeira consiste em encontrar a quantidade de tuplas comuns entre todos os pares de seqüências, através da qual mede-se a similaridade entre as seqüências. A segunda consiste em encontrar a distância entre essas seqüências através de transformadas específicas. As distâncias entre as seqüências são encontradas na segunda etapa a partir das medidas de similaridades encontradas na primeira etapa.

A quantidade de cálculo realizado na segunda etapa, entretanto, é mínima quando comparada ao cálculo da primeira etapa. Enquanto que na primeira etapa, verifica-se todos os caracteres de todos os pares de seqüências para encontrar as medidas de similaridades entre as seqüências, na segunda etapa estima-se a distância apenas com poucas operações sobre o valor calculado na primeira etapa. No entanto, uma paralelização apenas da primeira etapa exige que quantidades enormes de dados sejam coletados e armazenados no nó *front-end*, prejudicando a escalabilidade do algoritmo. Por este motivo paralelizou-se ambas as etapas. A figura 3.1 mostra o fluxograma do algoritmo paralelo. Este algoritmo utiliza o modelo mestre-escravo de distribuição de tarefas e o balanceamento de carga é feito dividindo-se dinamicamente a computação entre os processos existentes.

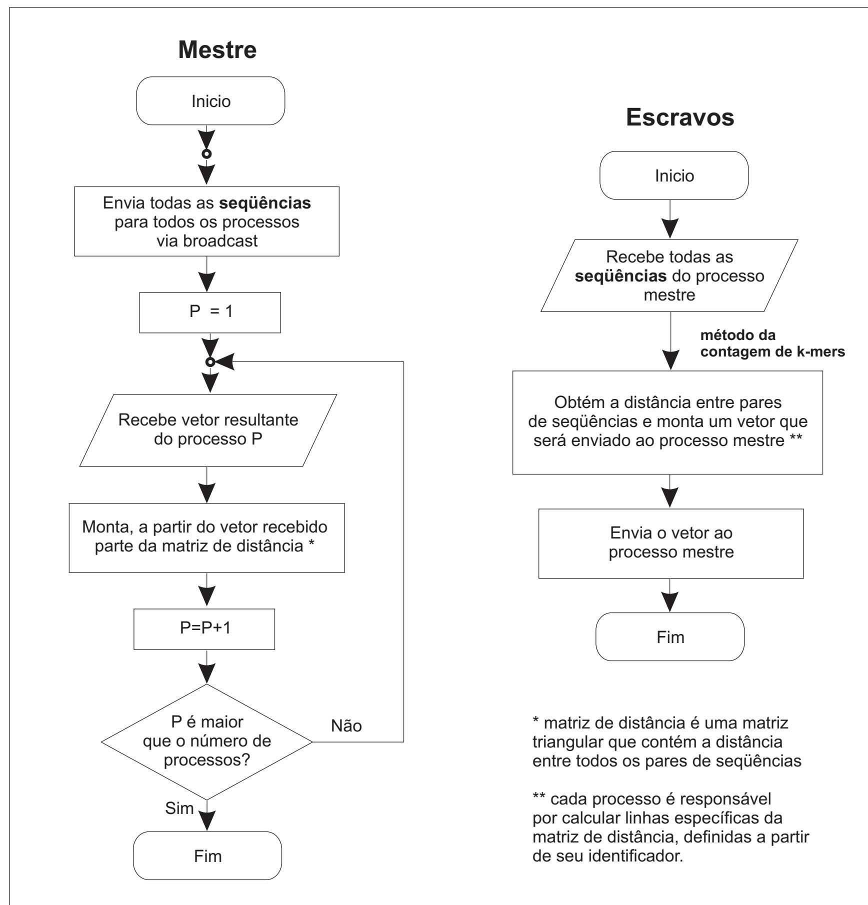

Figura 3.1: *Fluxograma do algoritmo paralelo do método de contagem de* k-mers

Inicialmente, todas as seqüências são enviadas por *broadcast* para todos os processos escravos. O envio por *broadcast* reduz o *overhead* com a troca de mensagens, diminuindo o tempo de comunicação entre os processos. Durante os testes verificou-se que o tempo gasto com este envio é insignificante em relação ao tempo gasto com a execução das tarefas.

A distribuição das tarefas é feita com base no identificador do processo, atribuindo a cada processo o cálculo de linhas específicas da matriz de distância. Essa matriz é triangular e a forma como ela é obtida está exemplificada na figura 3.2. Cada processo calcula inicialmente a linha correspondente ao seu identificador, em um laço de passo p, onde p é o número de processos. No exemplo, tem-se sete seqüências, totalizando sete linhas na matriz. O primeiro processo é responsável pelo cálculo das linhas 1, 4 e 7. O segundo pelas linhas 2 e 5 e o terceiro pelas linhas 3 e 6. Os valores em cinza são os

resultados obtidos nos outros escravos e que se juntarão apenas no processo mestre para a criação da matriz final de similaridades. Nota-se, neste exemplo, que este escalonamento tende a uma boa distribuição, para qualquer quantidade de seqüências envolvidas.

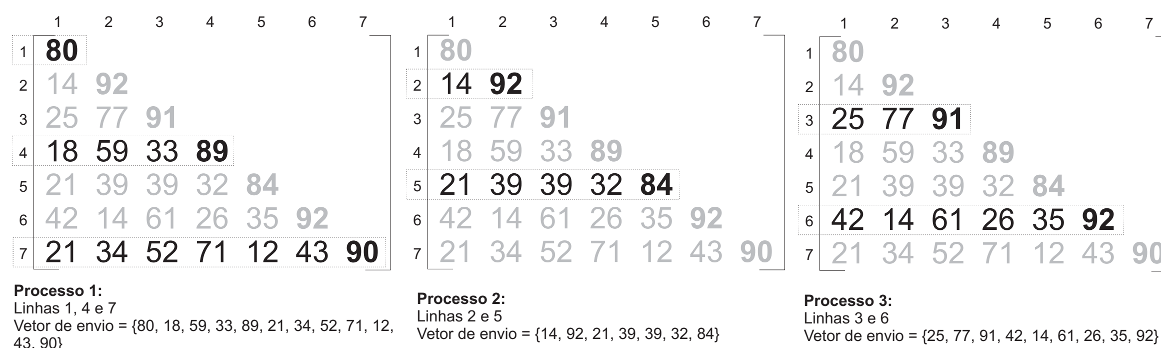

Figura 3.2: *Exemplo de como o cálculo da matriz de similaridades é distribuído entre os processos*

Durante o cálculo das linhas da matriz de distância, cada processo escravo armazena os resultados de todas as linhas em um único vetor. Este vetor armazena as distâncias calculadas em cada processo e, ao final de todos os cálculos, ele é enviado ao processo mestre. O processo mestre recebe os vetores de todos os processos e, a partir do vetor e do identificador do processo remetente, constrói a matriz de distância.

3.2 Paralelização do método da identidade fracional

O método da identidade fracional é empregado pelo MUSCLE em seu segundo estágio, utilizando como entrada apenas o alinhamento resultante do primeiro estágio. A execução do método consiste basicamente de dois laços aninhados, que calculam a distância entre todos os pares de seqüências.

A versão paralela deste método é muito similar à versão paralela do método de contagem de *k-mers*. Todos os processos recebem inicialmente o alinhamento resultante do primeiro estágio via *broadcast* e iniciam a execução da tarefa. Cada processo identifica as linhas por quais ele é responsável e calcula a distância entre os pares. Os resultados são armazenados em um único vetor que é enviado ao processo mestre. O processo mestre recebe este vetor e, a partir do identificador do processo emissor, armazena os valores recebidos nas posições

corretas da matriz. O fluxograma deste algoritmo é bastante similar ao fluxograma do algoritmo paralelo da contagem de *k-mers* e é mostrado na figura 3.3.

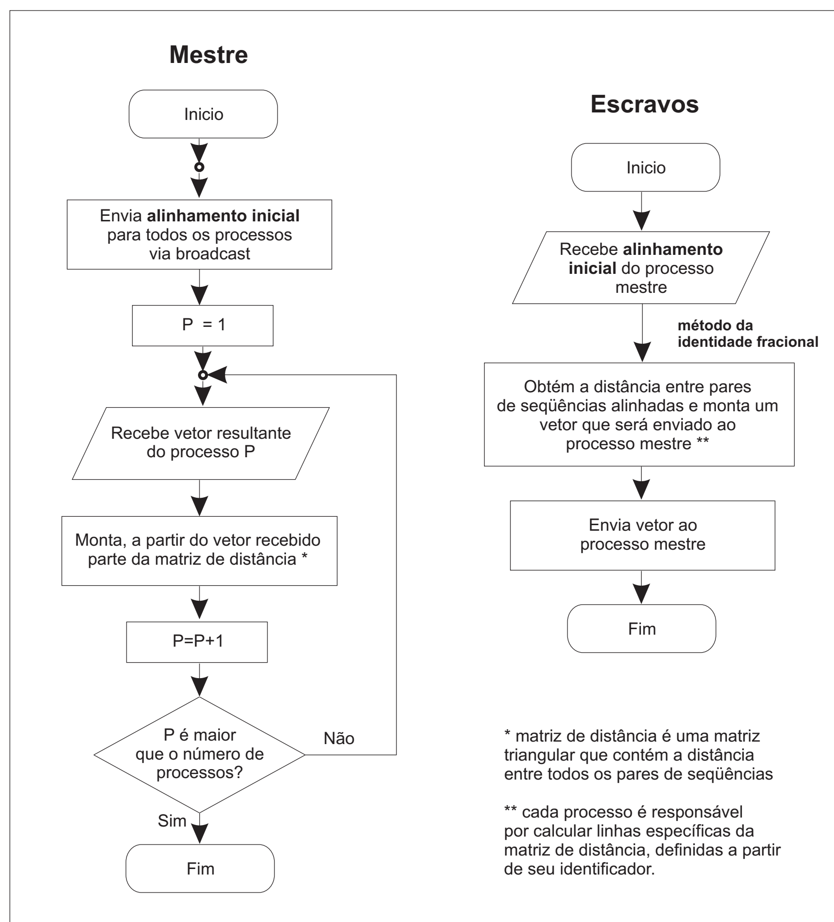

Figura 3.3: *Fluxograma do algoritmo paralelo do método da identidade fracional*

3.3 Paralelização do alinhamento progressivo

Como explicado no capítulo 2, o alinhamento progressivo consiste em alinhar pares de perfis, progressivamente, através de uma árvore previamente construída. O paralelismo de um problema com essas características pode ocorrer, fundamentalmente, em dois níveis, variando a granularidade do paralelismo. O primeiro nível considera cada nó da árvore como um único problema, onde cada nó da árvore tem seus dados decompostos e distribuídos. O segundo nível escalona nós inteiros de uma árvore.

Ambas as abordagens apresentam suas vantagens e desvantagens. Estratégias que atuam no primeiro nível, por exemplo, possuem um maior *overhead* de comunicação, devido a sua maior granularidade. Entretanto, apresentam um menor *overhead* de sincronismo, devido a uma menor dependência dos dados.

Estratégias em ambos os níveis foram desenvolvidas neste projeto. Como o paralelismo no primeiro nível também é utilizado no estágio iterativo, este é apresentado na seção 3.4. Esta seção, portanto, exibe apenas as estratégias desenvolvidas sobre o segundo nível, que considera todo o alinhamento progressivo como um único problema.

3.3.1 Abordagem com gargalo e soluções

Inicialmente são apresentados os problemas de eficiência encontrados em alguns algoritmos de paralelização do alinhamento progressivo. Esses algoritmos, em geral, utilizam uma abordagem que é apresentada em detalhes no artigo [12].

Para diminuir ou eliminar este problema, quatro soluções foram desenvolvidas e implementadas. Essas soluções apresentam vantagens e desvantagens em diferentes aspectos, e são mostradas através de uma análise conceitual.

Os testes de desempenho são feitos em seguida e são apresentados no capítulo 4. Como é feita uma comparação entre as quatro soluções e a abordagem com gargalo, implementou-se ao todo cinco estratégias. Todas essas estratégias realizam o escalonamento dinamicamente e empregam o modelo mestre-escravo de distribuição de tarefas.

3.3.2 O problema da abordagem existente

O mecanismo de escalonamento dinâmico utilizado por várias estratégias paralelas do alinhamento progressivo apresenta um gargalo responsável por bloquear a execução imediata de tarefas prontas. Este gargalo torna processos disponíveis desnecessariamente ociosos, e está, especificamente, na abordagem de troca de dados adotada. Como pode ser visto no fluxograma da figura 2.7, sempre que um processo escravo $\boldsymbol{A}$ precisa de dados que estão armazenados em um processo escravo $\boldsymbol{B}$, o processo mestre solicita a $\boldsymbol{B}$ o envio desses dados. Entretanto, $\boldsymbol{B}$ pode estar executando uma tarefa de alinhamento. Neste caso, $\boldsymbol{A}$

fica ocioso esperando receber de ***B*** os dados de entrada necessários para processar a nova tarefa.

Esta espera pode ocorrer freqüentemente, independente do número de tarefas de alinhamento e do número de processos. A figura 3.4 mostra uma situação em que pode ocorrer esta espera. Neste exemplo, apenas o processo ***D*** está processando uma tarefa de alinhamento. Os processos ***A***, ***B*** e ***C*** estão ociosos, esperando dados que estão em ***D*** e que apenas serão enviados quando o mesmo terminar de processar a tarefa em andamento.

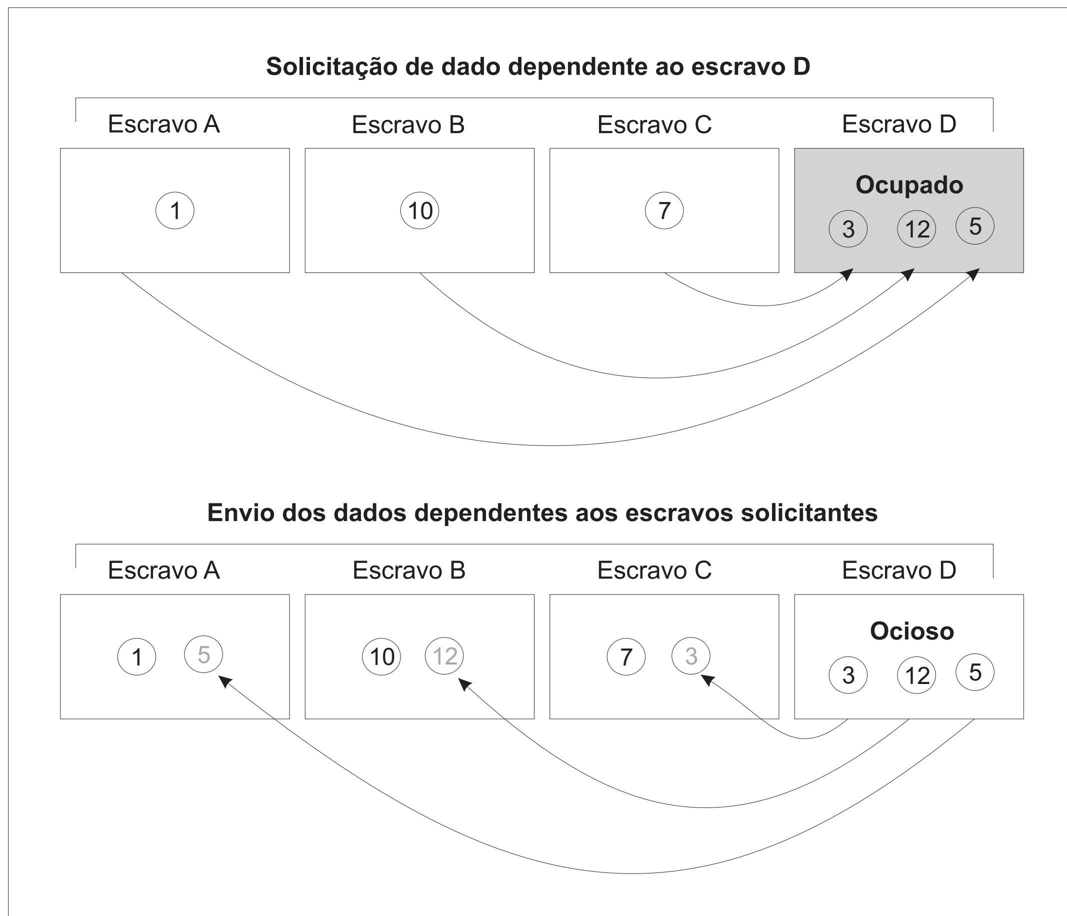

Figura 3.4: *Exemplo de caso da espera pela execução de processo de escravo vizinho para envio de dados dependentes*

3.3.3 Estratégia baseada na abordagem com gargalo

A primeira estratégia implementa a abordagem de troca de dados com gargalo. Sua implementação foi feita com o intuito de comparar a abordagem existente com as novas abordagens propostas. Embora Luo [12] proponha o uso de uma lista de prioridades para reduzir custos de comunicação, seu modelo não foi implementado. Este modelo define a ordem de execução das tarefas com base nos seus custos de execução e comunicação e, portanto, não considera a dependência entre as tarefas. Um modelo de prioridade com o foco na dependência entre as tarefas considera, por exemplo, a localização distribuída dos dados, administrada através de uma estrutura no nó *front-end*. Este modelo pode diminuir a latência e os custos com a comunicação e foi utilizado nas novas abordagens definidas.

Dessa forma, a estratégia baseada na abordagem com gargalo apenas realiza um escalonamento dinâmico através de um modelo de filas sem prioridades.

Implementação de um modelo de filas sem prioridades

O modelo de filas sem prioridades é implementado da seguinte forma. O processo mestre, inicialmente, distribui todas as tarefas dos nós folhas entre os processos escravos disponíveis, através de um algoritmo de balanceamento de carga. Em seguida, ele espera pela confirmação de término de alguma tarefa, seja ela tarefa de nó folha ou tarefa de nó intermediário. Obviamente, as primeiras tarefas finalizadas são as tarefas de nó folha. Ao recebe-las, ele verifica se sua tarefa irmã também está finalizada. Caso positivo, a tarefa de alinhamento do nó pai do nó da tarefa finalizada se torna ativa e é imediatamente escalonada. O fluxograma da figura 3.5 mostra o algoritmo do processo mestre. O algoritmo do processo escravo é o mesmo mostrado na figura 2.8.

3.3.4 Novas abordagens paralelas

Visando eliminar ou diminuir o gargalo de troca de dados existente na primeira estratégia, quatro possíveis soluções foram definidas. As duas primeiras soluções apresentam uma abordagem semelhante e são apresentadas juntas em 3.3.5. As demais soluções apresentam abordagens próprias e, portanto, são apresentadas separadamente em 3.3.6 e 3.3.7.

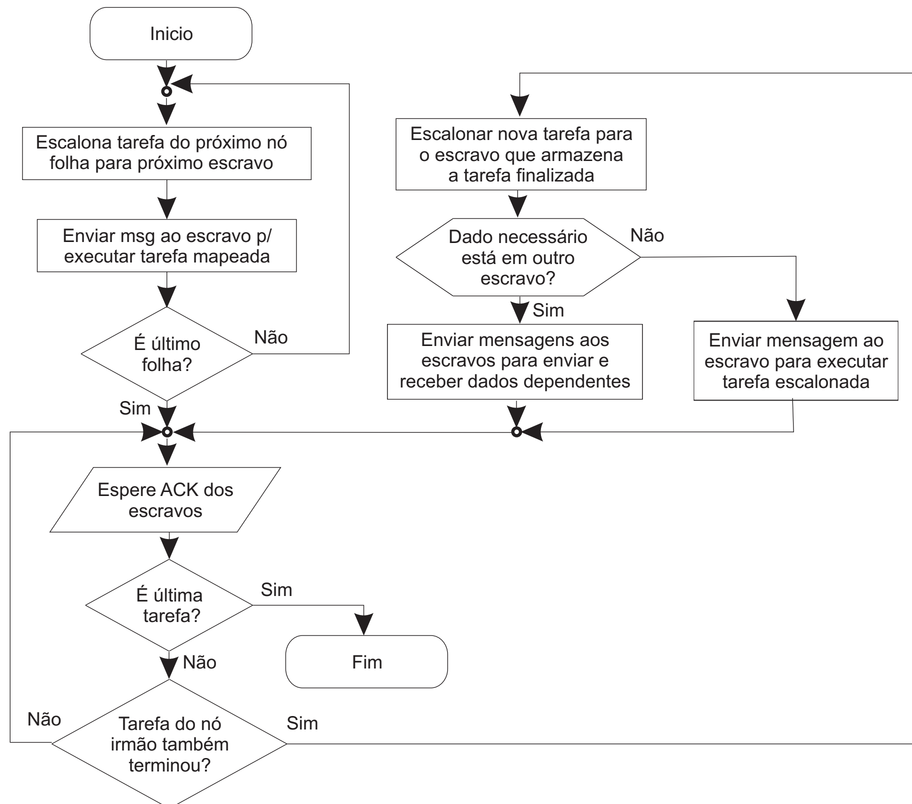

Figura 3.5: *Fluxograma do algoritmo do processo mestre da primeira estratégia*

Todas essas abordagens, entretanto, utilizam uma lista de tarefas ativas e identificam, dentre essas tarefas, quais podem ser distribuídas para o cálculo em um determinado instante, levando-se em conta a localização dos dados dependentes e a ociosidade dos processos. Durante essa identificação, as tarefas ativas recebem níveis distintos de prioridades, a partir das quais definem-se os escalonamentos. No entanto, todas as trocas de dados necessárias são feitas entre os processos escravos antes das tarefas serem processadas.

Da mesma forma que na estratégia com gargalo, dispensou-se no desenvolvimento das demais a criação de uma árvore de tarefas. As tarefas ativas são definidas seguindo a ordem de término de suas tarefas filhas, como explicado em 3.3.3. A diferença é que na estratégia com gargalo uma tarefa ativa é imediatamente escalonada. Por este motivo, nenhuma lista é mantida em memória. Aqui, as tarefas ativas não são alocadas ao primeiro processo ocioso e, portanto, são armazenadas em uma lista.

3.3.5 Soluções 1 e 2: Escalonar apenas tarefas com dependências em processos ociosos

Para diminuir a ociosidade com a espera da execução do processo dependente, identificam-se nessa abordagem as tarefas da lista de tarefas ativas que não dependem de dados de processos ocupados. Ou seja, apenas tarefas em que os resultados das tarefas de ambos os nós filhos estejam em processos ociosos. Sobre essas tarefas define-se uma prioridade de dois níveis: alto e baixo.

Uma tarefa de prioridade baixa é uma tarefa que possui dados dependentes em processos distintos. Neste caso, é necessário, no mínimo, uma troca de dados. Uma tarefa de prioridade alta, por sua vez, é uma tarefa que possui dados dependentes em um mesmo processo. Sua prioridade é alta pois o número de troca de dados pode ser nulo. Neste caso, a tarefa pai deve ser escalonada ao mesmo processo de seus dados dependentes.

Quem define o escalonamento com base nessas prioridades é a próxima etapa do algoritmo. O ponto aqui é encontrar os melhores mapeamentos tarefa/processo tais que o número de troca de dados entre os processos seja minimizado. Este procedimento é feito pelo processo mestre, mapeando primeiramente as tarefas mais prioritárias e finalizando quando todos os processos ociosos tiverem sido mapeados ou não existirem mais tarefas ativas.

Há casos, entretanto, em que duas tarefas de prioridade alta contém dados dependentes no mesmo processo. Neste caso, uma tarefa é escolhida para o mapeamento e a outra tem sua prioridade modificada para baixa. Sua prioridade é baixa pois, caso esta tarefa seja mapeada em seguida, todos os seus dados dependentes, que pertencem ao processo mapeado à outra tarefa, precisarão ser enviados.

Já entre duas tarefas de prioridade baixa, o critério que decide o mapeamento é a posição de um de seus dados dependentes. Se existirem processos que ainda não foram mapeados e que contém um dos dados dependentes de uma tarefa de prioridade baixa, um mapeamento é feito entre a tarefa e o processo em que encontra-se um de seus dados dependentes. Dessa forma, apenas uma troca de dados precisa ser feita. Caso isto não ocorra para nenhuma das tarefas restantes, o mapeamento é escolhido aleatoriamente.

Identificados os melhores mapeamentos, as tarefas são escalonadas. Entretanto, antes que o processo mestre envie mensagens de execução das tarefas aos processos escravos, todas as trocas de dados são feitas. Isto impede que ocorra o gargalo existente na abordagem anterior. Com todas as trocas de dados feitas inicialmente, evita-se que um processo espere um dado que se encontra em outro processo e que apenas será enviado quando este terminar de fazer o que está fazendo.

Após os dados serem trocados e armazenados localmente, o processo mestre envia as mensagens de execução e os processos escravos iniciam o processamento. Em seguida, o processo mestre espera por mensagens de término dessas tarefas.

As duas primeiras soluções seguem esta abordagem, porém diferem na forma em que esperam as mensagens de término dessas tarefas. Enquanto a primeira espera por todas as tarefas que estão em processamento, a segunda espera que apenas uma delas seja finalizada. Essas soluções levaram ao desenvolvimento das estratégias *waitall* (primeira solução) e *waitany* (segunda solução).

A vantagem da estratégia *waitall* é que no momento de fazer os próximos escalonamentos, todos os processos escravos estão disponíveis. Assim, é provável que uma maior quantidade de tarefas seja escalonada simultaneamente, uma vez que não existe a barreira de obter um dado necessário em um processo ocupado.

A vantagem da estratégia *waitany*, por outro lado, é que após uma tarefa ser finalizada, o processo mestre tenta imediatamente escalonar uma tarefa pronta, sem que seja necessário esperar pelo término das tarefas dos demais processos escravos. Enquanto uma abordagem apresenta a vantagem de ter uma quantidade maior de escalonamentos simultâneos, a outra apresenta a vantagem de eliminar a espera do processamento das demais tarefas.

O fluxograma do processo mestre é mostrado através da figura 3.6 e o fluxograma do processo escravo através da figura 3.7. A parte do algoritmo em que as estratégias *waitall* e *waitany* diferem entre si é mostrada através da condicional em negrito no fluxograma do processo mestre.

Essa abordagem não elimina completamente o gargalo pois considera apenas tarefas cujos dados dependentes estão em processos ociosos. Caso, em um instante de escalonamento,

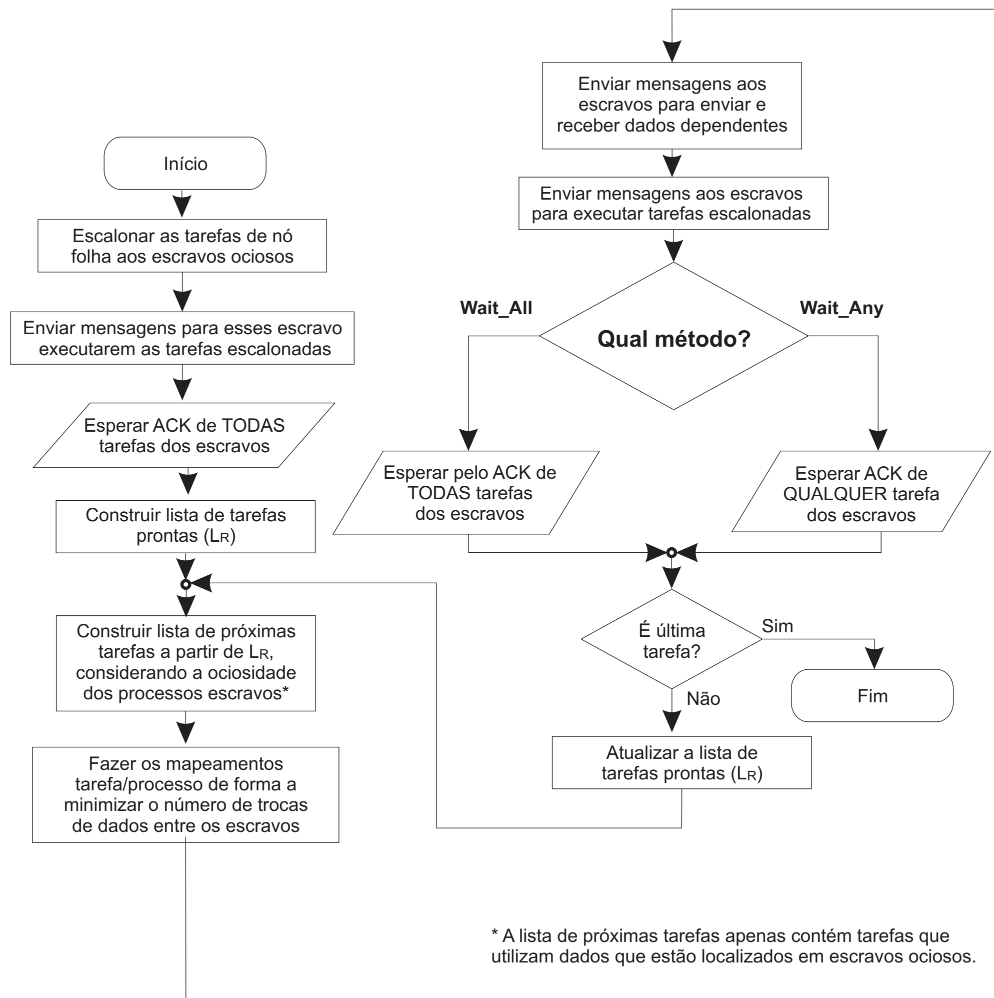

Figura 3.6: *Fluxograma do processo mestre das estratégias* waitall *e* waitany

existam apenas tarefas ativas que contém dados que estão em processos ocupados, os processos ociosos continuarão ociosos e nenhuma tarefa será escalonada. Ou seja, o gargalo nesta abordagem existe, porém de forma reduzida em relação à abordagem existente.

3.3.6 Solução 3: Fazer cópia de todos os dados no processo mestre

A terceira solução elimina o gargalo na troca de dados. A idéia desta solução é copiar o resultado de um alinhamento ao processo mestre sempre que um alinhamento é finalizado. Assim, sempre que um processo se torna ocioso, qualquer tarefa da lista de tarefas

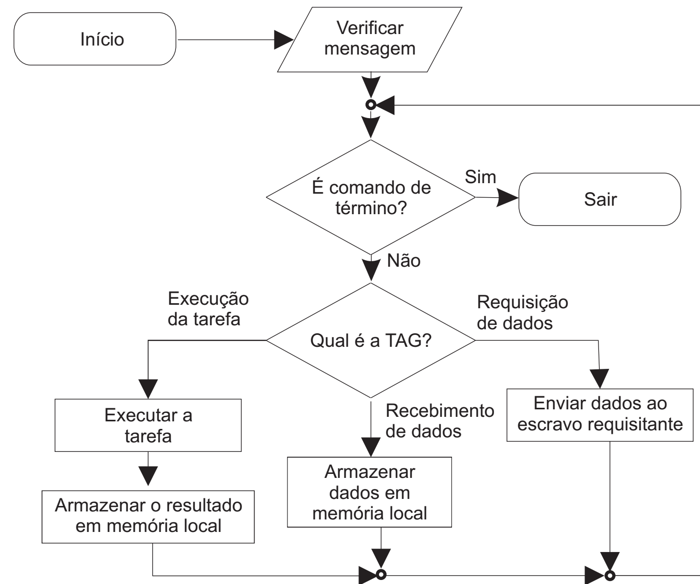

Figura 3.7: *Fluxograma do processo escravo das estratégias* waitall *e* waitany

ativas pode ser escalonada, independente da posição de seus dados dependentes. Caso esta tarefa necessite de um dado, uma mensagem é enviada ao processo mestre que o envia imediatamente.

O processo mestre, portanto, além de coordenar a execução das tarefas, também é responsável por manter cópias e enviar dependências, quando necessário. A operação de cópia de dados é feita sempre, pois não há como saber no futuro se aquele dado necessitará ou não ser enviado. Já a operação de envio de dados é feita apenas quando uma tarefa é executada em um processo que não contém todos os seus dados dependentes.

Caso uma tarefa seja escalonada a um processo que contém em memória local todas as suas dependências, nenhuma troca de dados é feita. Caso este processo contenha apenas uma dependência, apenas uma troca de dados é feita. Se ambas as dependências não estão em memória local, é necessário duas trocas de dados.

Para minimizar o número de troca de dados em um certo instante de escalonamento, três níveis de prioridades são utilizados: alto, quando ambos os dados dependentes estão no mesmo processo ocioso; médio, quando ambos os dados dependentes estão em processos distintos, porém um deles está em um processo ocioso; e baixo, quando ambos os dados dependentes estão em processos ocupados.

Após a prioridade ser atribuída às tarefas, definem-se os mapeamentos tarefa/processo que minimizam o número de trocas de dados. Inicialmente mapeiam-se as tarefas de prioridade alta, em seguida as tarefas de prioridade média e, por fim, as tarefas de prioridade baixa. Este procedimento finaliza quando todos os processos ociosos tiverem sido mapeados ou não existirem mais tarefas ativas.

Quando duas tarefas de prioridade alta contém dados dependentes no mesmo processo, uma tarefa é escolhida para o escalonamento e a outra tem sua prioridade modificada para baixa. Sua prioridade é baixa pois, caso esta tarefa seja escalonada a outro processo, todos os seus dados dependentes precisarão ser enviados.

Uma tarefa de prioridade média, por sua vez, pode conter dados em processos que já foram mapeados. Neste caso, a prioridade desta tarefa é modificada para baixa. Caso contrário, é feito um mapeamento tarefa/processo tal que o processo contenha um dos dados dependentes da tarefa. Por fim, são feitos os mapeamentos das tarefas de prioridade baixa. Como todos os processos restantes não contém dados das tarefas restantes, sendo sempre necessário duas trocas de dados, esses mapeamentos são feitos aleatóriamente.

Da mesma forma que na solução 1 e 2, todas as trocas de dados são feitas inicialmente. Em seguida, mensagens de execução são enviadas às tarefas mapeadas. Após a execução, todos os dados dependentes são dinamicamente eliminados da memória.

A estratégia *sendmaster* foi criada contemplando as características dessa solução. O algoritmo do processo mestre desta estratégia é mostrada pelo fluxograma da figura 3.8. O fluxograma do algoritmo do processo escravo, por sua vez, é visto na figura 3.9.

Esta estratégia elimina completamente o gargalo de dependência na troca de dados pois permite que qualquer tarefa ativa seja escalonada no momento em que um processo se torna ocioso. Entretanto, o custo com a comunicação é muito elevado, piorando, por outro lado, o desempenho do algoritmo. Adicionalmente, a memória do nó *front-end* deve ser capaz de suportar todas as cópias de dados, caso contrário ela é um grande gargalo.

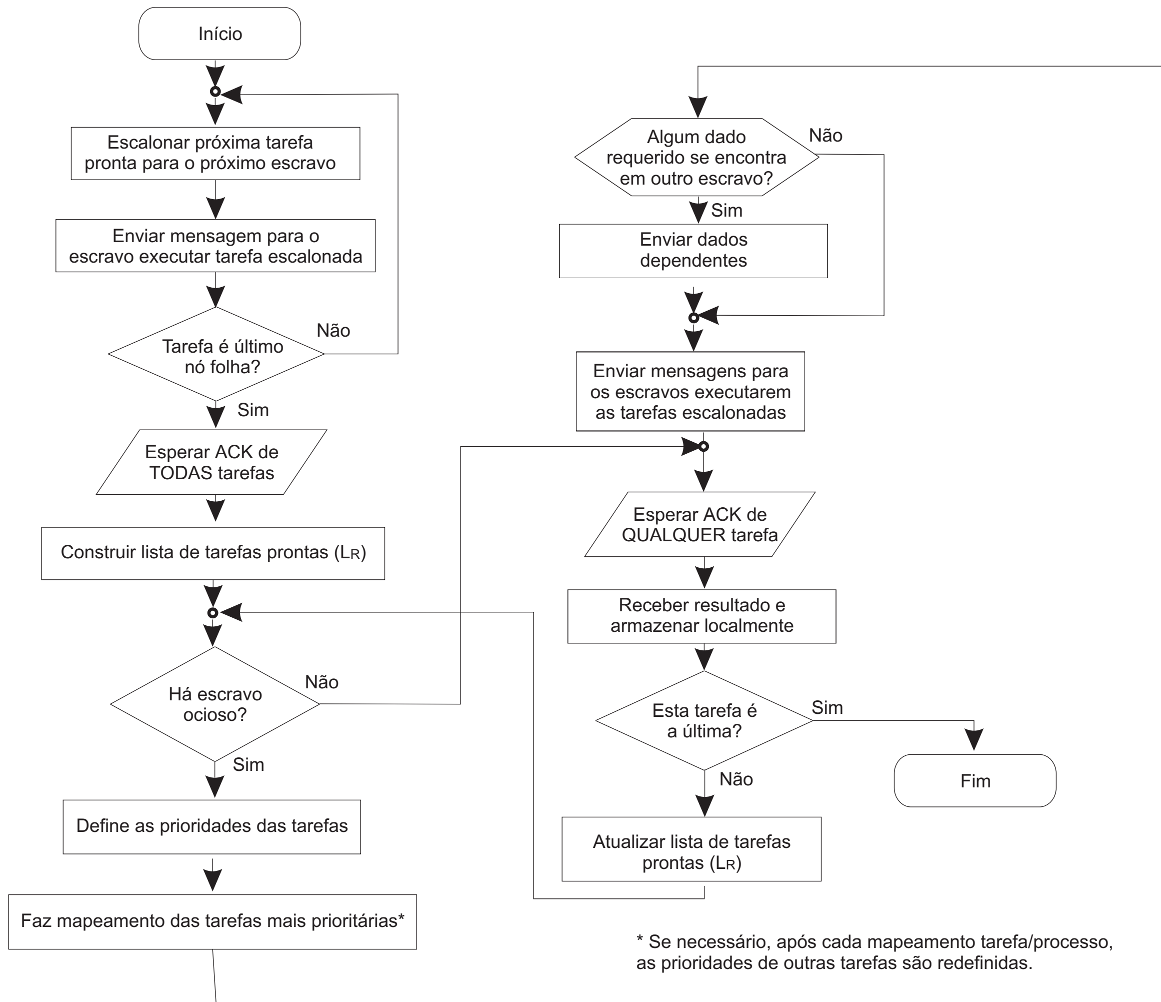

Figura 3.8: *Fluxograma do algoritmo do processo mestre da estratégia* sendmaster

3.3.7 Solução 4: Criar *threads* exclusivos para a troca de dados

A última solução também elimina o gargalo na troca de dados. Sempre que um processo se torna ocioso e existem tarefas ativas, um escalonamento é realizado. Este escalonamento é feito independente da posição dos dados dependentes e é obtido diretamente do processo escravo que o contém, reduzindo o custo de comunicação em relação à solução anterior e a provável ocorrência de um gargalo de memória. Para isso, um *thread* exclusivo para a troca de dados é disparado em cada processo escravo.

Com o uso de *threads*, processos ocupados compartilham seus recursos para que o dado seja enviado imediatamente, evitando a espera com o término de execução do processo. A

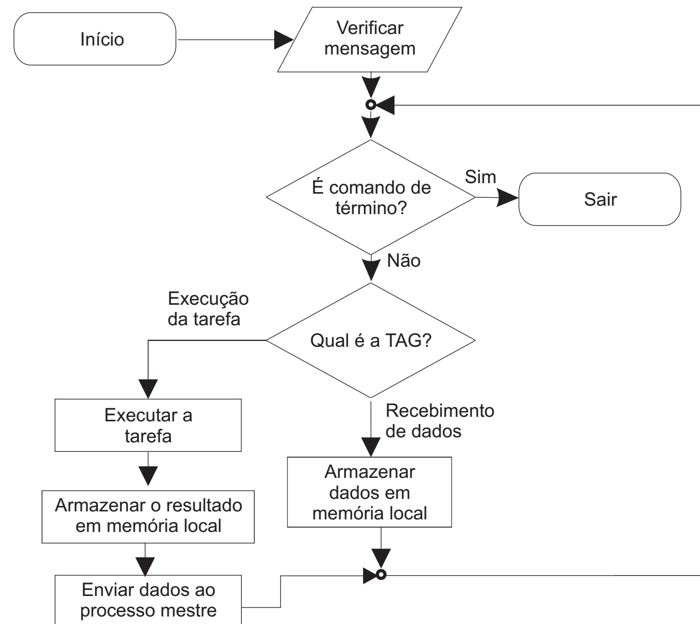

Figura 3.9: *Fluxograma do algoritmo do processo escravo da estratégia* sendmaster

desvantagem é que nem todas as implementações do MPI contemplam o uso de *threads*, impossibilitando a execução de tal estratégia.

As prioridades e o mecanismo de escalonamento desta abordagem são semelhantes aos da solução anterior. Três níveis de prioridades são definidos e os mapeamentos são feitos de modo a minimizar o número de troca de dados.

Definidos todos os mapeamentos tarefa/processo, as trocas de dados são feitas e, em seguida, todas as mensagens de execução são enviadas. Este procedimento continua até que todas as tarefas sejam finalizadas e os escalonamentos são feitos após cada processador se tornar ocioso. As figuras 3.10 e 3.11 mostram os fluxogramas no processo mestre e escravo, respectivamente, da estratégia implementada sobre esta abordagem.

3.3.8 Considerações sobre as implementações no segundo estágio

No segundo estágio de execução do MUSCLE, o alinhamento progressivo é feito seguindo uma nova árvore guia. Para evitar redundância de cálculo, faz-se inicialmente uma comparação da nova árvore com a árvore construída no primeiro estágio e realizam-se novos alinhamentos par-a-par apenas nos nós que sofreram modificações.

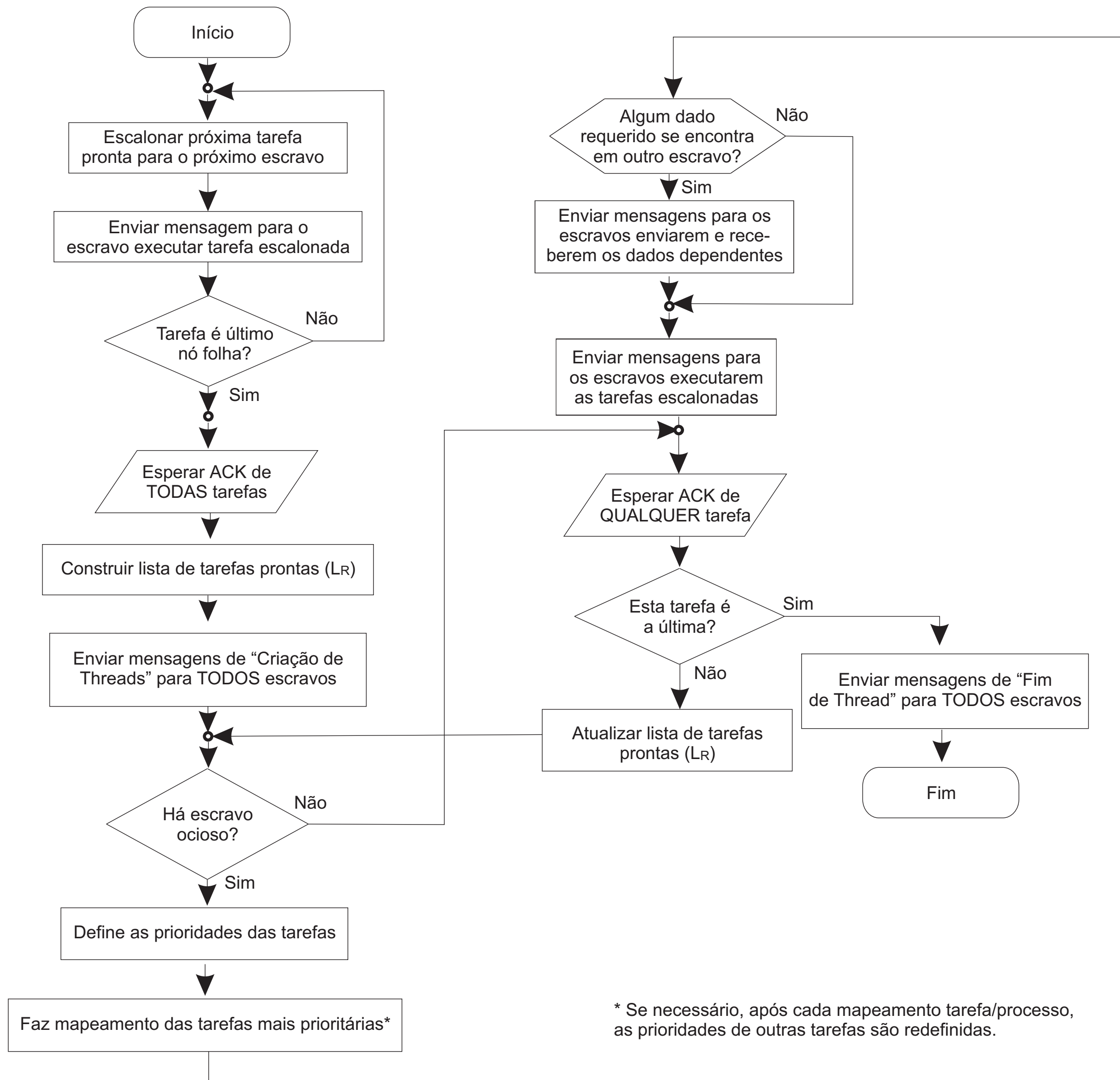

Figura 3.10: *Fluxograma do algoritmo do processo mestre da estratégia com* threads

As estratégias utilizadas são as mesmas descritas anteriormente, a diferença ocorre apenas na implementação, que contempla características específicas deste estágio. Um exemplo é o tipo de dado de entrada utilizado e a disposição desses dados no sistema. Enquanto no primeiro estágio, os dados de entrada encontram-se no processo mestre e estes são enviados aos processos escravos, no segundo estágio, os dados de entrada estão distribuídos entre os vários processos escravos. No segundo estágio, utiliza-se também uma estrutura adicional para evitar os cálculos redundantes na etapa do alinhamento progressivo.

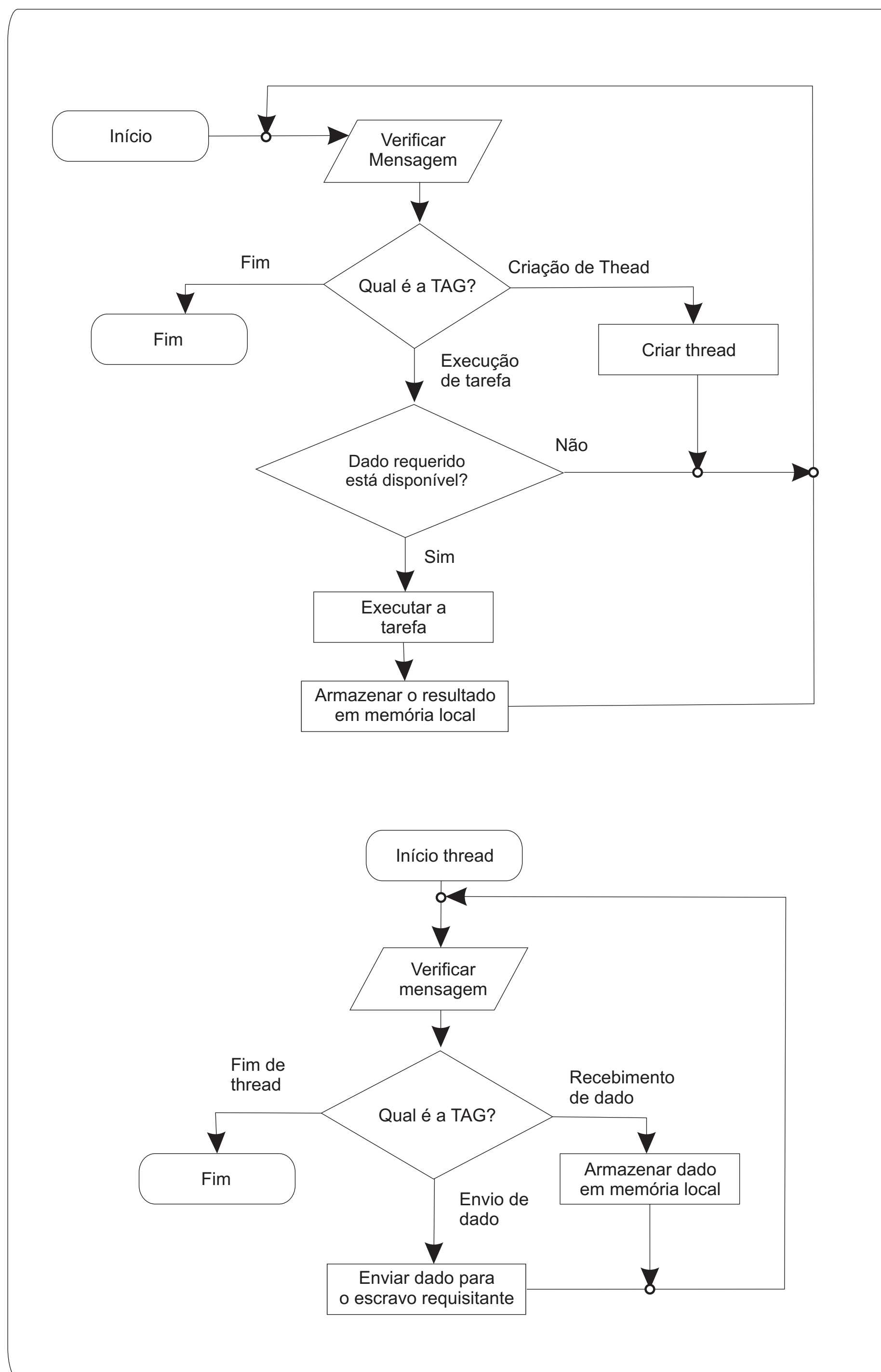

Figura 3.11: *Fluxograma do algoritmo do processo escravo da estratégia com* threads

3.4 Paralelização do alinhamento par-a-par

O paralelismo do alinhamento par-a-par é utilizado no estágio iterativo e progressivo do MUSCLE. No estágio iterativo, este paralelismo ocorre na função que demanda o maior

custo computacional dentro de cada iteração. Nos estágios progressivos, este paralelismo substitui o paralelismo feito em função da árvore filogenética. Com esta abordagem, cada nó da árvore é paralelizado.

As implementações das estratégias de alinhamento par-a-par sofrem pequenas variações de acordo com o estágio. No estágio iterativo, o alinhamento é feito sobre um conjunto de seqüências alinhadas enquanto que no estágio progressivo o alinhamento é feito sobre dois perfis de alinhamento.

No estágio iterativo as seqüências são dividas em dois sub-conjuntos. Esses sub-conjuntos são alinhados através da matriz de programação dinâmica. Inicialmente, obtém-se um caminho a ser percorrido na matriz para a construção do alinhamento. Em seguida, utilizando uma técnica conhecida por *tracebak*, obtém-se o alinhamento resultante.

No estágio progressivo, o alinhamento de dois perfis também é feito através de uma matriz de programação dinâmica obtendo-se inicialmente um caminho. Este caminho, entretanto, é utilizado na construção de um novo perfil. Apenas no nó raiz da árvore o perfil resultante é convertido em um alinhamento.

Para decidir qual estratégia paralela adotar em cada caso, estudou-se, primeiramente, o custo com a comunicação. Dependendo da quantidade dos dados trocados, o *overhead* com a comunicação pode inviabilizar o uso de estratégias paralelas. Esta análise respeita as características especificas da implementação do alinhamento par-a-par em cada estágio.

No estágio progressivo, os dados de entrada e saída são perfis de alinhamento. No estágio iterativo, esses dados são os próprios alinhamentos, representados através de conjuntos alinhados de seqüências. Uma solução consiste em obter o resultado no *front-end* (um novo perfil ou um novo alinhamento) e enviar esses dados para todos os processos escravos para que o processamento paralelo da próxima etapa seja feito. Uma análise teórica da transmissão foi feita para ambos os estágios, através de uma investigação no código-fonte do MUSCLE. No estágio iterativo, para alinhamentos de n seqüências de tamanho L, o custo com a comunicação é $O(nL)$. No estágio progressivo, considerando que cada coluna de um perfil possui um conjunto de variáveis que totalizam 467 bytes, identificado através de uma inspeção em seu código fonte, a transmissão de um único perfil de tamanho L tem um custo de comunicação de $O(467L)$, independente da quantidade de seqüências que cada

perfil representa.

Para reduzir o *overhead* com a comunicação, uma solução foi enviar os dados necessários para o cálculo de cada perfil/alinhamento e replicar o cálculo em todas as máquinas. Esta solução é possível desde que todos os nós contenham os perfis dos nós filhos (no estágio progressivo) ou o alinhamento imediatamente anterior (no estágio iterativo). A função que calcula o perfil/alinhamento resultante utiliza os perfis filhos ou o alinhamento anterior junto com o caminho. Esta solução é viável pois este cálculo é relativamente mais rápido que a comunicação e é feito simultaneamente em todos os processos. Os dados de um caminho de alinhamento são sempre 9 bytes para cada coluna do perfil, acrescentados de 8 bytes de variáveis de controle. Neste caso, o custo de comunicação é $8+9L$ para perfis de tamanho L, reduzindo em aproximadamente 52 vezes a quantidade de dados enviados em relação ao envio do perfil, e $n/9$, desde que $n > 9$, em relação ao envio do alinhamento.

Enquanto por um lado esta última solução apresenta a vantagem de gerar um menor *overhead* de comunicação, o processamento do sistema é totalmente utilizado. A solução que envia o caminho e replica o processamento é mais vantajosa em sistemas dedicados ou em casos onde a rede do sistema é relativamente lenta. Porém, analisando apenas o tempo de execução do algoritmo, ela é preferível em relação a solução que envia o alinhamento já pronto. Ambas, portanto, foram implementadas no MUSCLE.

No estágio progressivo, as redundâncias dos perfis filhos são eliminadas conforme os perfis pais são calculados ou caso este seja o perfil do nó raiz. Caso o algoritmo seja executado no primeiro estágio, apenas um processo o mantém armazenado, através de um algoritmo que mantém uma boa distribuição dos dados entre os processos. Uma cópia é mantida pois ela é utilizada no segundo estágio progressivo. Caso o algoritmo seja executado no segundo estágio, todos as cópias são eliminadas, mantendo em todos os processos apenas os perfis recentemente calculados, que são utilizados para o cálculo de novos perfis.

Seguindo as características de cada estágio, três estratégias de paralelismo foram desenvolvidas. Todas utilizam a técnica *block-based wavefront* [44], explicada em 2.11, através de um modelo mestre/escravo. Entretanto, ao contrário do proposto em [44], implementou-se um algoritmo que possibilita o ajuste do tamanho dos blocos da matriz em ambas as

dimensões, como explicado em 3.4.2.

3.4.1 Estratégias implementadas sobre ambas as soluções

As duas primeiras estratégias enviam os dados da matriz de programação dinâmica para o processo mestre, calculando o caminho no processo mestre de forma seqüencial. A técnica *traceback* pode então ser executada de duas formas. Ou ela é executada no processo mestre, e o perfil/alinhamento resultante propagado em todos os processos, ou ela é executada em todos os processos, obtendo-se os resultados localmente. Uma análise teórica dessas abordagens é feita na seção 3.4.

Essas duas estratégias diferem, entretanto, na forma como os dados são enviados. Enquanto uma delas envia todos os dados de uma só vez após todo o processamento, a outra envia os dados em partes, em momentos distintos da execução do algoritmo.

A vantagem da primeira estratégia é que uma única mensagem é enviada por processo, e portanto há um menor custo de comunicação. A segunda estratégia, por outro lado, diminui o *overhead* de sincronismo, pois enquanto há processos se comunicando também há processos trabalhando. Em ambos os casos, o *overhead* com a comunicação é muito alto e são estratégias praticamente inviáveis.

Para minimizar o *overhead* exagerado de comunicação dessas estratégias, uma terceira foi definida. A terceira estratégia realiza o cálculo do caminho de forma distribuída. Esta estratégia, além de diminuir o *overhead* de comunicação, evita que uma quantidade enorme de dados, que estão distribuídos, se reunam em um único ponto (*front-end*), potencializando a ocorrência de um gargalo.

A distribuição do cálculo do caminho é feito da seguinte forma. Ao terminar o cálculo da matriz de programação dinâmica, o último processo escravo inicia a técnica *traceback*. Esta técnica parte do último elemento da matriz e vai até o primeiro, verificando em cada elemento qual o tipo de operação que está a ele associado, a partir do qual identifica-se o próximo elemento da matriz que será examinado. Quando um certo elemento deve ser verificado e este encontra-se em outro processo, uma mensagem é enviada solicitando a continuação da operação.

3.4.2 O tamanho dos blocos da matriz

A técnica *block-based wavefront* divide a matriz em p colunas, onde p é o número de processadores. As colunas, por sua vez, são dividas em b blocos, de acordo com o número de linhas da matriz. Ao invés de definir estaticamente a quantidade de blocos por coluna (b) e definir o número de colunas (p) apenas com base no número de processadores, esta técnica recebe como entrada os valores do tamanho horizontal e vertical mínimo de cada bloco. A partir desses valores define-se, então, o número de blocos.

Com esta estratégia, a quantidade de colunas pode variar de 2 a p blocos, enquanto que em linhas este valor é limitado pelo tamanho do perfil/alinhamento. Isto permite que uma quantidade variável de processos sejam utilizados para cada operação, de acordo com o volume de dados. Este ajuste automático no número de processadores impede que seqüências pequenas sejam distribuídas em muitos processadores, impedindo a geração de um *overhead* de comunicação que degrade o desempenho do algoritmo.

O tamanho mínimo ideal de um bloco depende de fatores como velocidade de processamento e comunicação de um sistema. Por este motivo, este valor não é fixo e deve ser ajustado de acordo com as características de cada sistema. Como um futuro trabalho esses ajustes podem ser feitos dinamicamente, definindo-se parâmetros de desempenho críticos e obtendo-os através de testes rápidos no início da execução do algoritmo. No algoritmo desenvolvido, entretanto, a opção automática atua como a estratégia proposta em [44]. Esta opção divide a matriz em p linhas e p colunas e é executada caso nenhuma medida de tamanho de bloco seja informada na chamada de execução. Esta opção foi utilizada nos testes comparativos das três estratégias, como mostrado na seção 4.4.1, e, apesar de não ser a melhor opção, apresenta, em geral, um bom comportamento.

3.5 Paralelização do cálculo da pontuação objetiva

A paralelização deste método é semelhante a do método da contagem de *k-mers* e do método da identidade fracional. A diferença é que aqui calcula-se pontuações de colunas específicas de um alinhamento.

A distribuição das colunas aqui também é feita com base no identificador do processo. Cada processo calcula inicialmente a coluna correspondente ao seu identificador, e percorre o alinhamento calculando pontuações de colunas específicas dentro de um laço de passo p, onde p é o número de processos.

A pontuação das colunas é somada em cada processo e o resultado final enviado ao processo mestre. O processo mestre recebe todos os valores e soma-os, obtendo a pontuação resultante.

4 Testes e Resultados

Muitos experimentos foram realizados para testar o desempenho dos algoritmos paralelos propostos. As seqüências utilizadas foram extraídas da base de dados do NCBI (www.ncbi.nlm.nih.gov) com o foco apenas no número de seqüências e no número de resíduos em cada seqüência. Para cada teste feito, descreve-se as informações específicas da instância do problema utilizada, como o número de seqüências e o comprimento médio das seqüências.

Os testes foram executados em um cluster *Beowulf* constituído de 16 máquinas, cada uma com um processador Intel(R) Pentium(R) 4 CPU 2.80GHz e 1GB de memória. Estas máquinas estão conectadas através de um *switch* dedicado de 100 Mb/s. Os testes foram executados no cluster com um número crescente de processadores, permitindo-nos analisar a escalabilidade do algoritmo. Os tempos de execução do algoritmo foram calculados executando-os em modo *stand-alone*, para garantir o uso exclusivo de comunicação, CPU e memória.

Cada tópico deste capítulo refere-se a um método do MUSCLE paralelizado, e cuja descrição encontra-se no capítulo 3. Todas as informações de cada teste, bem como seus respectivos resultados, são apresentadas a seguir.

4.1 Contagem de *k-mers*

Para verificar a eficiência deste método, realizou-se testes com várias entradas distintas. Para cada arquivo de entrada, verificou-se como o algoritmo se comporta quando executado em um número crescente de máquinas. Deve-se notar, porém, que um sistema mínimo para

a execução do algoritmo paralelo consiste de dois processos, pois tal algoritmo é do tipo mestre-escravo. Ou seja, um processo é responsável apenas pelo gerenciamento dos dados. Dessa forma, os resultados da execução em apenas um nó podem ser obtidos com a versão seqüencial do algoritmo ou com a execução de dois processos em uma única máquina. Como optou-se por calcular o *speedup* real a primeira alternativa foi adotada.

A primeira classe de entradas mantém constante o comprimento médio (número médio de resíduos) das seqüências, variando apenas o número de seqüências envolvidas. O comprimento de cada seqüência, nessas entradas, é de aproximadamente 1000. Ao todo, quatro entradas distintas foram utilizadas. Essas entradas contém 500, 1000, 2000 e 4000 seqüências.

As figuras 4.1 e 4.2 mostram o comportamento do algoritmo para essa primeira classe de entrada. Para cada uma das quatro entradas, essas figuras mostram o tempo de execução e o *speedup* real do algoritmo.

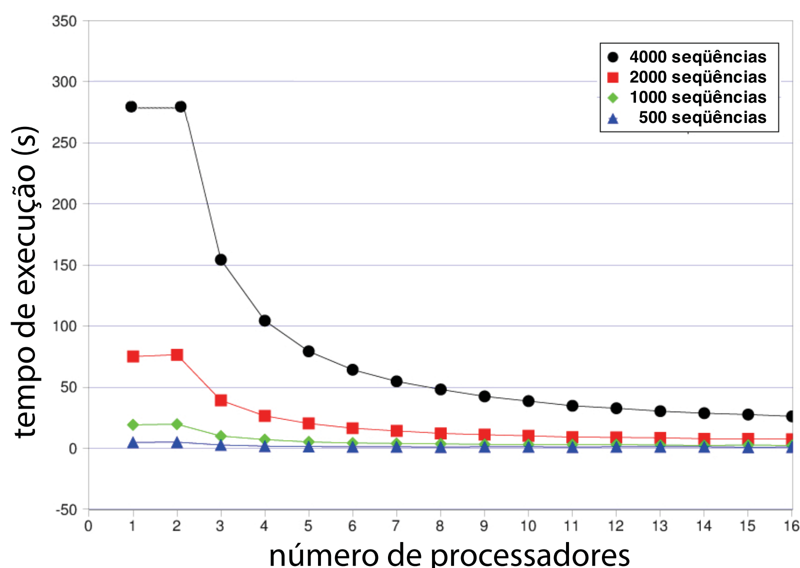

Figura 4.1: *Gráfico de tempo de execução do algoritmo paralelo de contagem de* k-mers *para entradas com seqüências de aproximadamente 1000 resíduos*

A segunda classe de testes também mantém constante o comprimento das seqüências, porém cada seqüência possui um tamanho consideravelmente menor em relação ao tamanho das seqüências da primeira classe de entradas. Aqui, cada seqüência possui em média 50 resíduos. O número de seqüências, por outro lado, é variado. Utilizou-se, ao todo, três

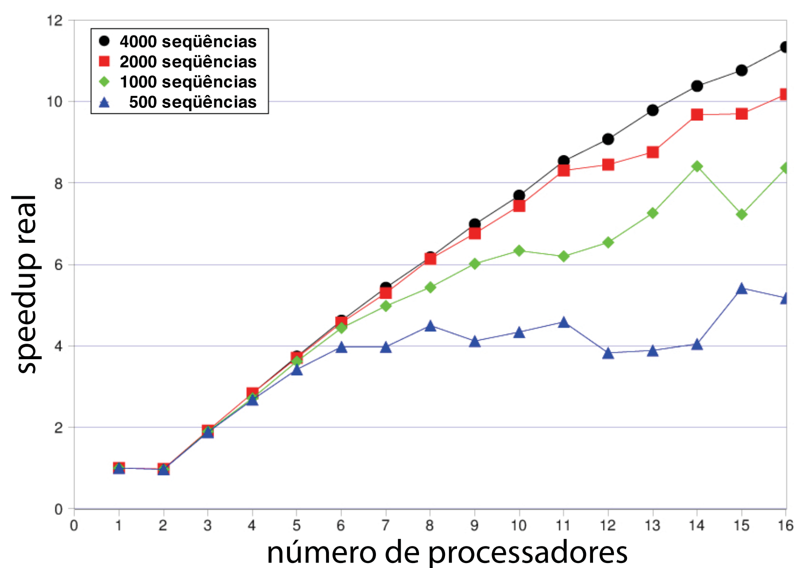

Figura 4.2: *Gráfico de* speedup *real do algoritmo paralelo de contagem de* k-mers *para entradas com seqüências de aproximadamente 1000 resíduos*

entradas, com 3000, 4000 e 5000 seqüências. Para cada entrada, as figuras 4.3 e 4.4 mostram, respectivamente, o tempo de execução e o *speedup* real do algoritmo para um número crescente de processadores.

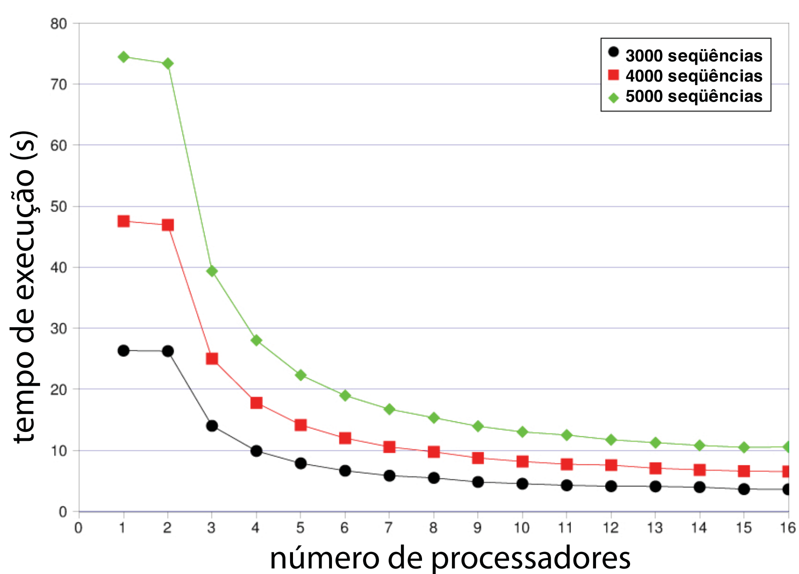

Figura 4.3: *Gráfico de tempo de execução do algoritmo paralelo de contagem de* k-mers *para entradas com seqüências de aproximadamente 50 resíduos*

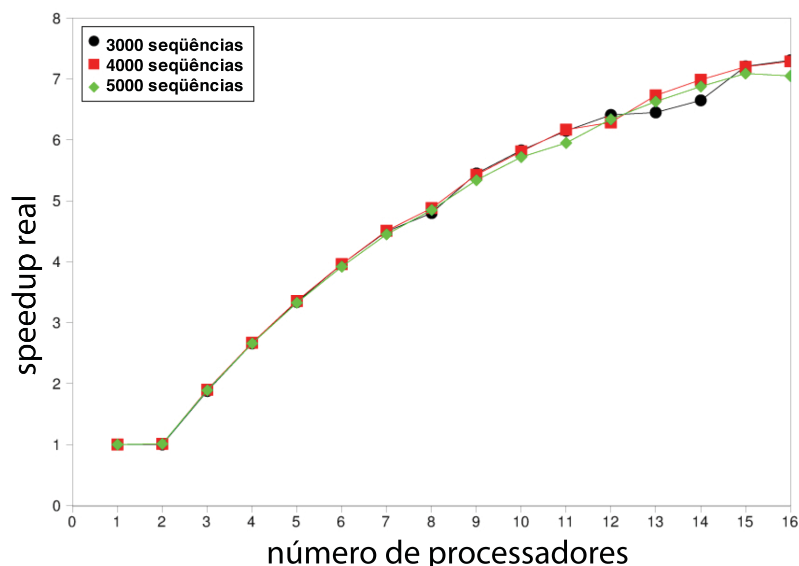

Figura 4.4: *Gráfico de* speedup *real do algoritmo paralelo de contagem de* k-mers *para entradas com seqüências de aproximadamente 50 resíduos*

A partir dos testes anteriores, é possível comparar o ganho de desempenho do algoritmo para entradas com o mesmo número de seqüências, porém de comprimentos diferentes. Para isso, obteve-se o comportamento das entradas com 4000 seqüências da primeira e da segunda classe. A figura 4.5 mostra uma comparação do *speedup* real nos dois casos. Note que a execução do programa com uma entrada com seqüências de maior comprimento apresenta um maior ganho de desempenho com um número maior de nós, confirmando a característica de escalabilidade do algoritmo.

Para visualizarmos a quantidade de *overhead* de comunicação e sincronismo neste algoritmo, extraiu-se, por fim, o perfil de execução de uma das entradas. A entrada escolhida foi a de 500 seqüências com aproximadamente 1000 resíduos cada. O algoritmo, por sua vez, foi executado utilizando-se apenas quatro nós do cluster. Essa escolha foi feita de forma a manter um *overhead* mínimo gerado pela instrumentação do código, ao mesmo tempo em que testou-se um caso com uma quantidade considerável de dados de entrada. A figura 4.6 mostra o tempo gasto com o *overhead* de comunicação e sincronismo obtido a partir do perfil dessa execução.

Nesta figura, as barras indicam o percentual de tempo gasto com as funções de comunicação e sincronismo do MPI. Essas funções podem estar trocando dados, esperando um

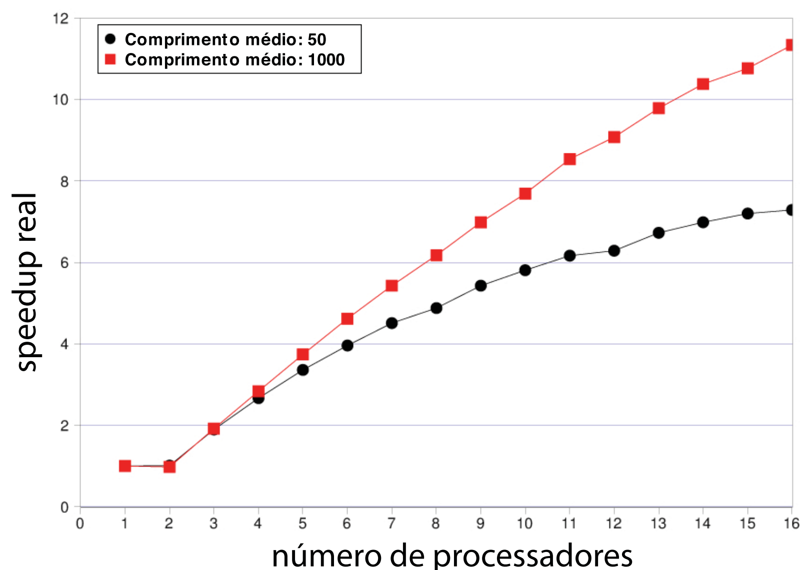

Figura 4.5: *Gráfico de comparação do* speedup *real do algoritmo paralelo de contagem de* k-mers *para entradas com 4000 seqüências de aproximadamente 50 e 1000 resíduos*

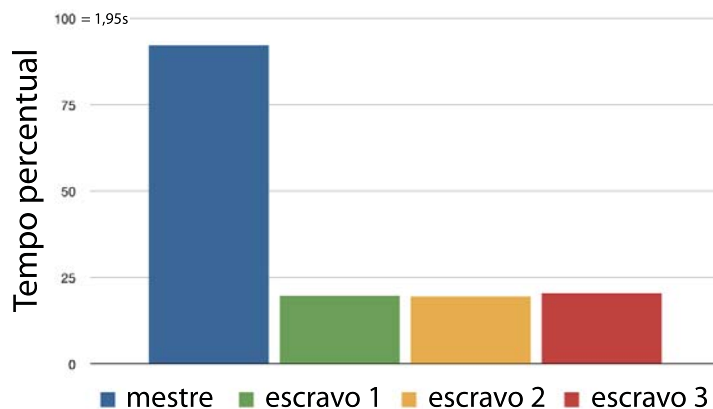

Figura 4.6: *Percentual de tempo gasto com comunicação e sincronismo do algoritmo de contagem de* k-mers *para a entrada com 500 seqüências de aproximadamente 1000 resíduos*

momento para o recebimento ou envio de dados (comunicação síncrona) ou simplesmente esperando outros processos através de funções de sincronismo. Em todos esses casos, o uso de processador é mínimo. O percentual de tempo em que o processador fica de fato ocupado é representado pelo espaço após as barras. Essa informação é mostrada para cada processo.

O processador com identificador zero é o processador mestre. Sua função é de apenas distribuir os dados, receber os resultados e junta-los para formar o resultado final. Neste processo, a maior parte do tempo é gasto com a espera dos dados, através de uma função de recebimento síncrono do MPI (função `MPI_Recv`). Os processos escravos, por outro lado, passam a maior parte do tempo trabalhando, mas também possuem um *overhead* de comunicação e sincronismo que ocorre no início e no final da execução do algoritmo. No exemplo da figura 4.6, uma parte considerável do tempo é gasta com o *overhead*. Porém, este tempo tende a diminuir com o aumento do problema, como é visto implicitamente nas figuras 4.1 e 4.2.

4.2 Identidade fracional

A abordagem deste método é semelhante a abordagem do algoritmo paralelo da contagem de *k-mers*, tanto na forma como os dados são divididos entre os processos quanto na abordagem de gerenciamento dos mesmos. Todos os testes realizados com o algoritmo paralelo partiram da execução do algoritmo com no mínimo dois processadores até o número máximo de processadores do cluster. A obtenção do ganho de desempenho seqüencial foi obtida com a execução do algoritmo seqüencial. Entretanto, devido às limitações de memória, não foi possível executar algumas entradas tanto com o algoritmo seqüencial quanto com o algoritmo paralelo em poucos nós. Neste caso, o tempo de execução não foi medido e o ganho de desempenho foi calculado em relação ao tempo de execução do algoritmo paralelo com o número mínimo de nós que habilita a execução.

Devido a semelhança com a abordagem paralela da contagem de *k-mers*, o algoritmo paralelo da identidade fracional apresentou características semelhantes de escalabilidade quando executado sobre os mesmos conjuntos de entrada. Os primeiros testes são mostrados na figura 4.7. Utilizou-se aqui as entradas com 500, 1000, 2000 e 4000 seqüências com aproximadamente 1000 resíduos cada e mediu-se o tempo de execução do algoritmo utilizando-se um número crescente de nós de execução. Entretanto, para as entradas com 1000, 2000 e 4000 seqüências não foi possível medir o tempo de execução do algoritmo seqüencial. Também não foi possível medir, para a entrada de 2000 e 4000 seqüências, o tempo de execução do algoritmo paralelo utilizando-se poucos nós.

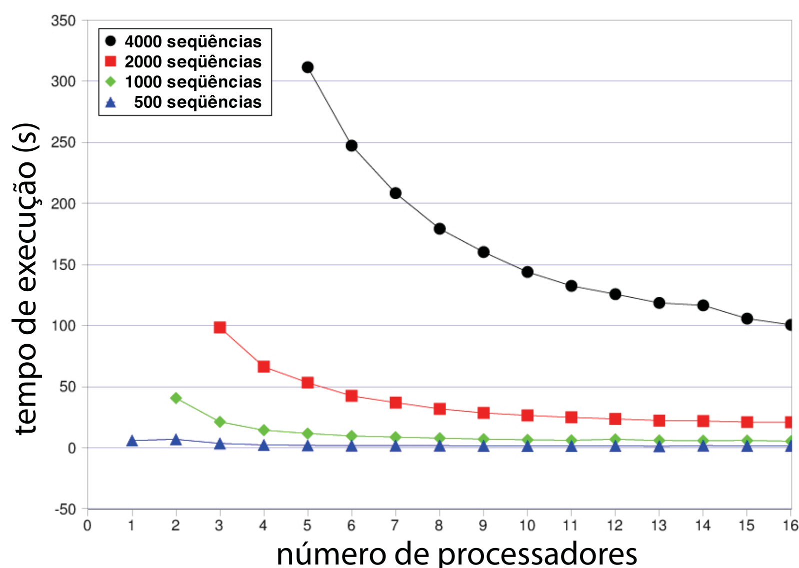

Figura 4.7: *Gráfico de tempo de execução do algoritmo paralelo da identidade fracional para entradas com seqüências de aproximadamente 1000 resíduos*

A inviabilidade da execução do algoritmo nesses casos se deve às limitações de memória do sistema. Conforme aumenta-se o problema, aumenta-se o uso de memória. A memória disponível, por sua vez, é expandida com o aumento do número de nós utilizados. Como o algoritmo da identidade fracional no MUSCLE é executado apenas no segundo estágio, utilizando como entrada o resultado do primeiro estágio, a memória disponível deve ser suficiente para que todos os métodos do primeiro estágio sejam executados, caso contrário o algoritmo da identidade fracional não é executado. Neste teste, houve insuficiência de memória para as entradas com 1000, 2000 e 4000 seqüências com o algoritmo seqüencial. Para a entrada com 2000 seqüências, a execução do algoritmo paralelo em dois nós também não pôde ser realizada. Para a entrada com 4000 seqüências, a execução em paralelo só foi possível a partir de cinco nós.

A segunda classe de testes realizada utiliza entradas menores, ocupando menos memória do sistema. Dessa forma, pôde se executar as entradas também em 1 único nó, através do algoritmo seqüencial, possibilitando a medição do *speedup* real. As entradas utilizadas possuem 3000, 4000 e 5000 seqüências com aproximadamente 50 resíduos cada. As figuras 4.8 e 4.9 mostram o tempo de execução e o *speedup* real respectivamente.

A figura 4.10 mostra, por sua vez, o ganho de desempenho obtido variando-se o com-

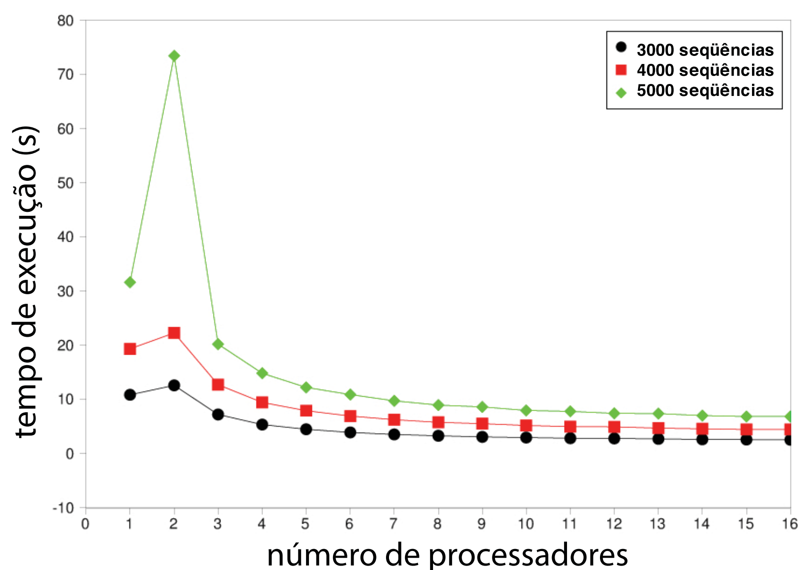

Figura 4.8: *Gráfico de tempo de execução do algoritmo paralelo da identidade fracional para entradas com seqüências de aproximadamente 50 resíduos*

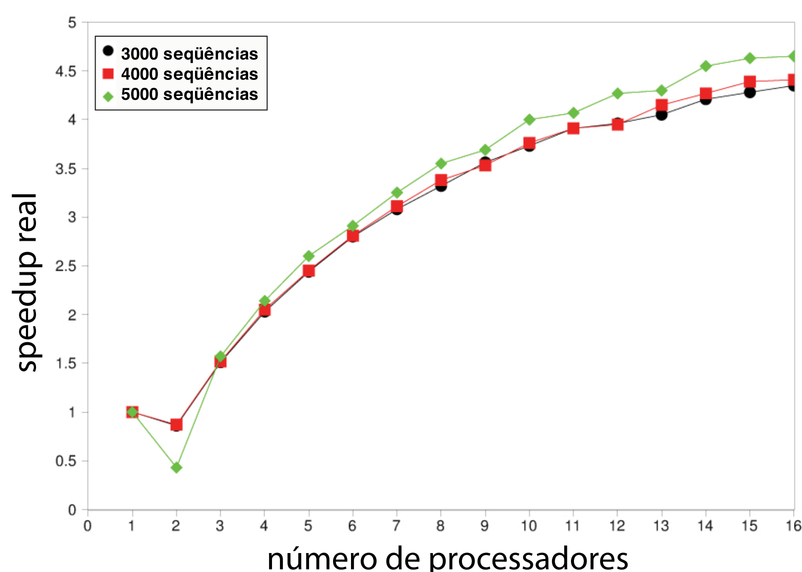

Figura 4.9: *Gráfico de* speedup *real do algoritmo paralelo da identidade fracional para entradas com seqüências de aproximadamente 50 resíduos*

primento médio das seqüências. Ambas entradas deste teste possuem 4000 seqüências, porém o número de resíduos médio por seqüência em cada entrada é 50 e 1000. Como a execução da entrada com comprimento médio de 1000 resíduos só pôde ser realizada com o algoritmo paralelo em cinco nós, o ganho de desempenho foi calculado a partir do tempo

desta execução. A partir dessa figura vemos que tal algoritmo paralelo apresenta uma boa escalabilidade.

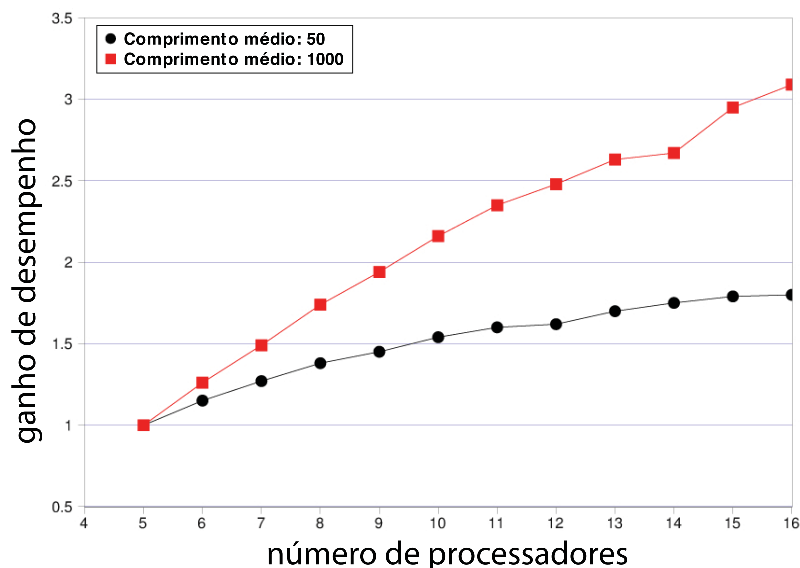

Figura 4.10: *Gráfico de ganho de desempenho do algoritmo paralelo da identidade fracional para entradas com 4000 seqüências de aproximadamente 50 e 1000 resíduos*

Para mostrar o impacto do *overhead* de comunicação e sincronismo neste algoritmo, extraiu-se também o perfil de execução para o mesmo caso de teste do algoritmo da contagem de *k-mers*: 500 seqüências de aproximadamente 1000 resíduos. Em ambos algoritmos, o tempo gasto com *overhead* e com a execução são bem semelhantes. A figura 4.11 mostra essa informação para o algoritmo da identidade fracional.

4.3 Alinhamento progressivo

Os testes a seguir mostram o tempo de execução das abordagens paralelas do alinhamento progressivo. Como a etapa do alinhamento progressivo é realizada no primeiro e no segundo estágio do MUSCLE, seu algoritmo possui algumas diferenças de implementação. Essas diferenças não afetam o desempenho do algoritmo pois a abordagem em ambos os estágios é a mesma. Como a intenção é avaliar o desempenho de cada estratégia, os testes aqui apresentados foram todos realizados no primeiro estágio.

Dentre essas estratégias incluem-se a estratégia com gargalo e as quatro soluções im-

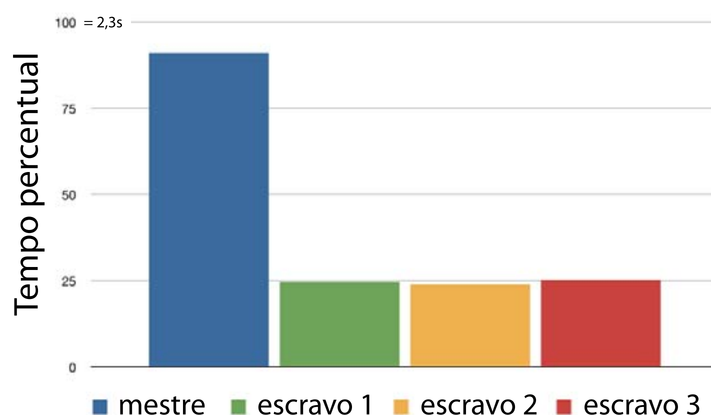

Figura 4.11: *Percentual de tempo gasto com comunicação e sincronismo do algoritmo paralelo da identidade fracional para a entrada com 500 seqüências de aproximadamente 1000 resíduos*

plementadas. Para cada algoritmo, os testes foram feitos com várias entradas distintas, variando-se o número de processadores a partir do número mínimo possível para a sua execução, que satisfaz as restrições de memória, até 16 processadores.

Para qualquer solução adotada, o desempenho do paralelismo é sempre dependente da estrutura da árvore filogenética, obtida a partir do conjunto de seqüências de entrada. Dessa forma, a escolha do arquivo de entrada influencia fortemente o desempenho do algoritmo. Para mostrar como o mesmo se comporta em vários tipos de casos, optou-se, primeiramente, por realizar testes com o uso de entradas normais, compostas por um conjunto de seqüências aleatórias e por uma árvore filogenética válida, construída, a partir das distâncias entre essas seqüências, por algum método válido de construção de árvores. Entretanto, o melhor caso ocorre quando a árvore filogenética está balanceada, o que é raro de acontecer com o uso de entradas normais. Para analisar este caso, implementou-se um algoritmo modificado do método de construção de árvore que, independente das seqüências utilizadas, sempre produz uma árvore balanceada. Este algoritmo é chamado através do uso da *flag* `balance`, na passagem de parâmetros durante a chamada de execução do MUSCLE. Por produzir um resultado de alinhamento inválido, ele é utilizado apenas para medir o desempenho do algoritmo no melhor caso e não como uma opção de uso normal da ferramenta.

Além do desempenho em casos aleatórios e no melhor caso, também é possível ver nos testes a escalabilidade do algoritmo, variando-se o número de processadores e o conjunto de seqüências utilizadas.

4.3.1 Comparação entre as estratégias

O primeiro teste realizado mostra o desempenho da estratégia com gargalo. Apesar de habilitar a execução de problemas maiores com o paralelismo, o ganho de desempenho obtido é relativamente pequeno quando o algoritmo paralelo é executado. A figura 4.12 mostra o tempo de execução deste algoritmo para quatro entradas distintas. Essas entradas contêm 500, 1000, 2000 e 4000 seqüências de aproximadamente 1000 resíduos cada.

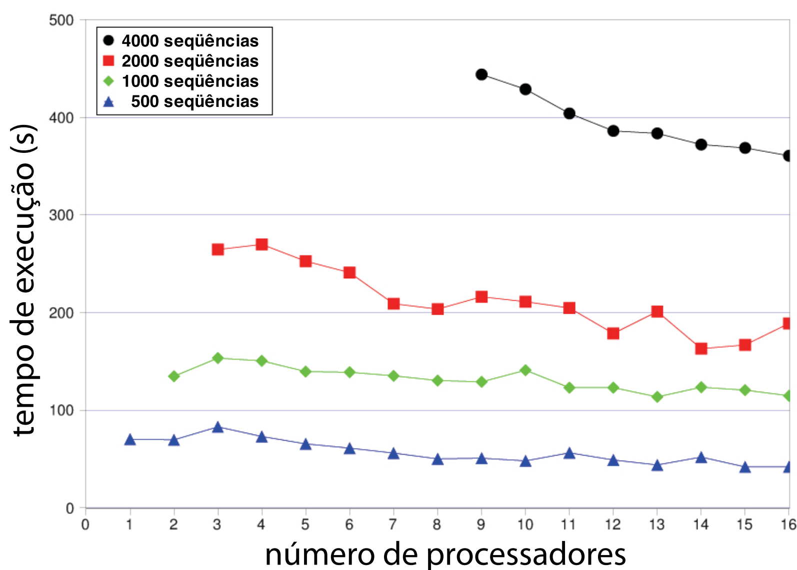

Figura 4.12: *Gráfico de tempo de execução da estratégia com gargalo do alinhamento progressivo*

Note que este algoritmo apenas consegue executar o teste com a maior entrada a partir de nove nós de execução. Como será mostrado adiante, este mesmo teste pode ser executado a partir de cinco nós de execução nas demais estratégias. Isto ocorre pois tal algoritmo não realiza uma boa distribuição dos dados entre os nós. A figura 4.13 mostra o percentual de tempo gasto com a comunicação e sincronismo, para o teste com 500 seqüências. Nesta figura vemos como o nó 2 é sobrecarregado enquanto os demais ficam ociosos a maior parte

do tempo. Este desbalanceamento, além de diminuir o ganho de desempenho também diminui a capacidade de memória do sistema.

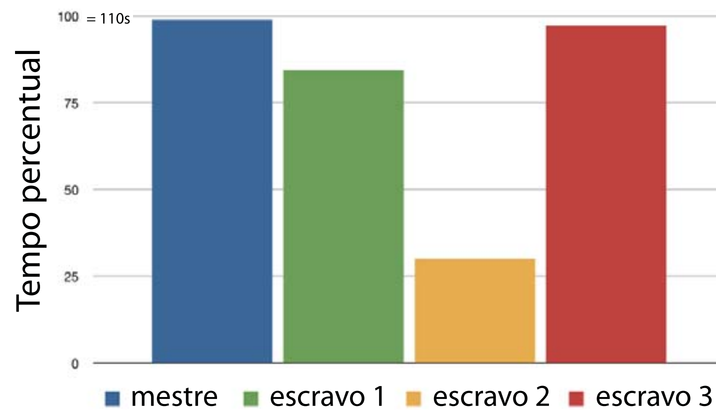

Figura 4.13: *Percentual de tempo gasto com comunicação e sincronismo da estratégia do alinhamento progressivo com gargalo para a entrada com 500 seqüências*

O segundo teste mostra que o paralelismo com a solução *sendmaster* também não é eficiente. A figura 4.14 mostra os resultados obtidos com a execução das mesmas entradas do teste anterior.

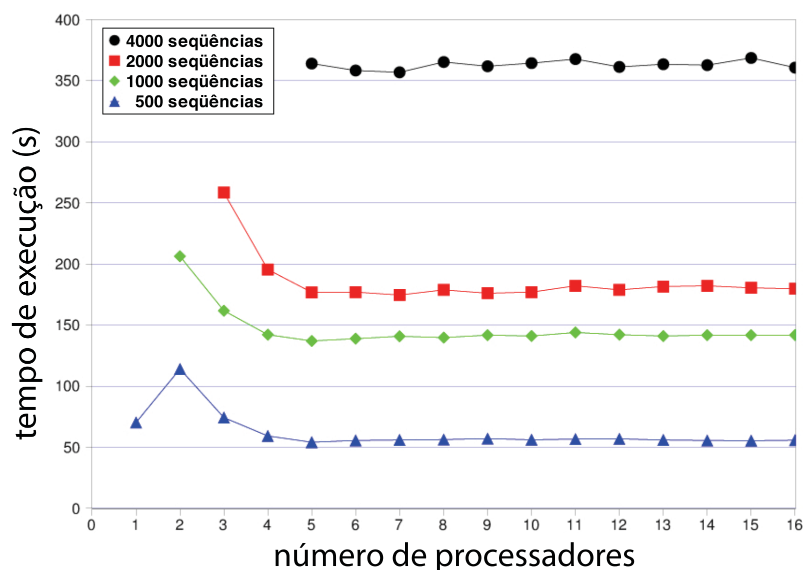

Figura 4.14: *Gráfico de tempo de execução da estratégia* sendmaster

Neste gráfico vemos que a estratégia *sendmaster* não é uma boa estratégia. O tempo

de execução cresce absurdamente quando o algoritmo é executado em poucos nós e o *overhead* com a comunicação sobrepõe o ganho de desempenho com a divisão das tarefas. Já com o aumento do número de nós, o ganho de desempenho mantém-se constante. Neste caso, o ganho de desempenho com a divisão das tarefas é anulado pelo *overhead* com a comunicação. A única vantagem nesta estratégia é o melhor balanceamento de carga, como mostrado na figura 4.15. Esta característica possibilita a execução de problemas grandes em clusters de tamanho menor. Por exemplo, a execução da entrada de 4000 seqüências em um cluster com mesma capacidade de memória, porém com apenas cinco nós de execução, ao invés dos nove exigidos na estratégia com gargalo.

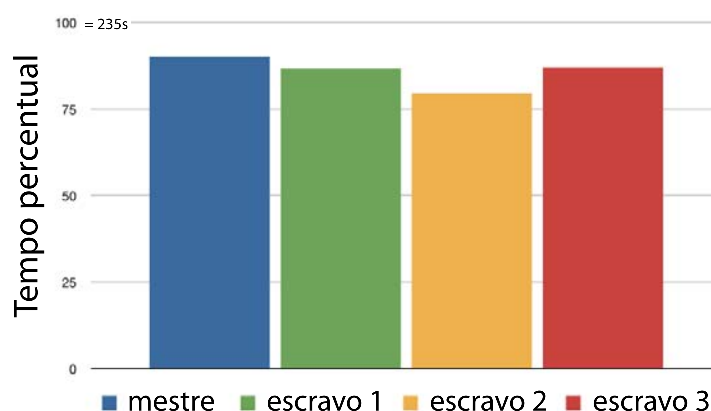

Figura 4.15: *Percentual de tempo gasto com comunicação e sincronismo da estratégia* sendmaster *para a entrada com 500 seqüências*

As estratégias *waitall* e *waitany* foram testadas em seguida. Apesar delas não eliminarem totalmente o gargalo com a latência, elas apresentam resultados melhores que as duas primeiras estratégias. Em testes realizados com os mesmos conjuntos de entrada dos testes anteriores, ambas as estratégias apresentaram um melhor desempenho, com destaque para a estratégia *waitany*. As figuras 4.16 (*waitall*) e 4.17 (*waitany*) mostram os tempos de execução dessas estratégias para todas as entradas.

Como pode ser visto, o ganho de desempenho obtido com a diminuição do gargalo é considerável. O *overhead* de comunicação é praticamente o mesmo da estratégia existente, porém com uma redução no gargalo e um melhor balanceamento de carga. As figuras 4.18 e 4.19 mostram o percentual de tempo gasto com *overhead* de comunicação e sincro-

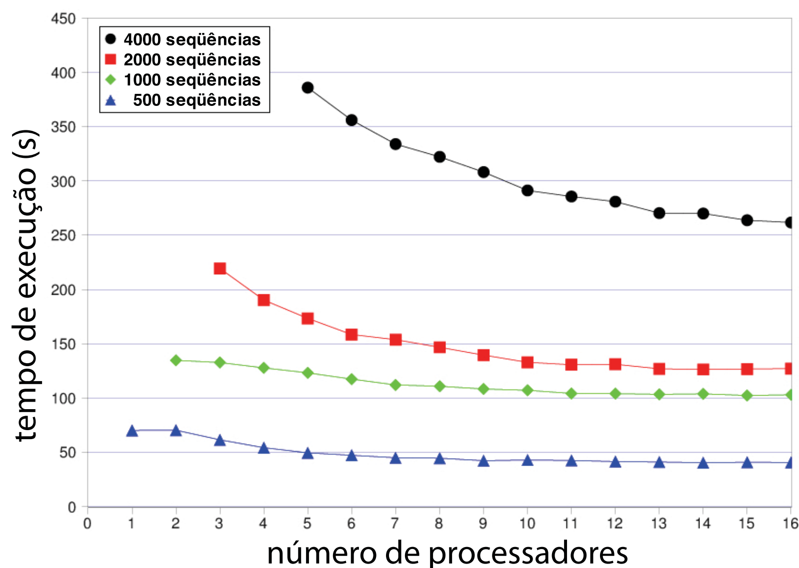

Figura 4.16: *Gráfico de tempo de execução da estratégia* waitall

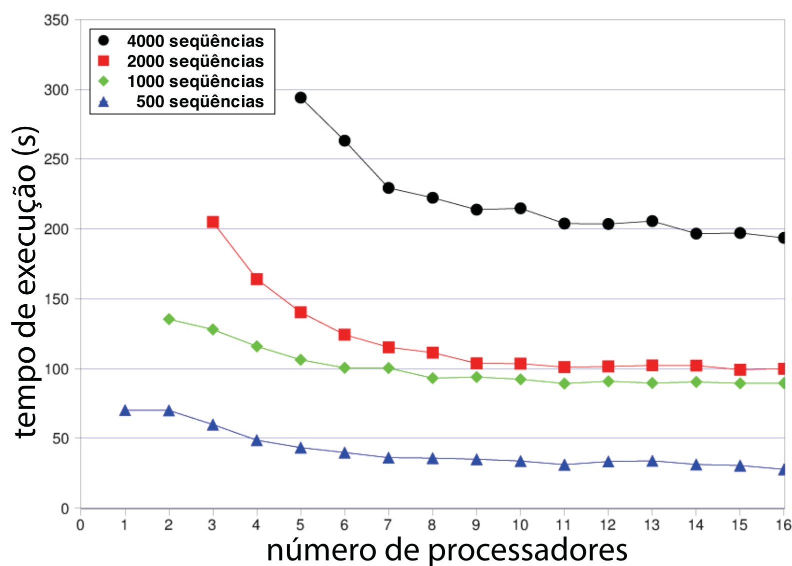

Figura 4.17: *Gráfico de tempo de execução da estratégia* waitany

nismo para o teste com 500 seqüências. Através desses gráficos também é possível ver o balanceamento de carga destes algoritmos.

A última solução implementada foi a estratégia com *threads*. Essa estratégia, apesar de estar restrita à execução apenas com algumas implementações do MPI, é a melhor das quatro estratégias. A figura 4.20 mostra os tempos de execução com as entradas dos testes

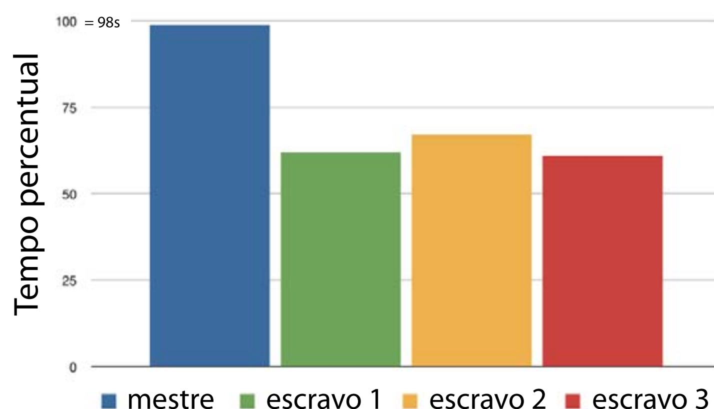

Figura 4.18: *Percentual de tempo gasto com comunicação e sincronismo da estratégia* waitall *para a entrada com 500 seqüências*

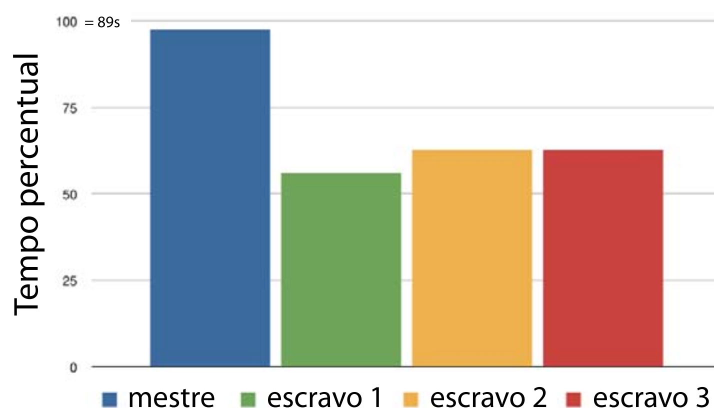

Figura 4.19: *Percentual de tempo gasto com comunicação e sincronismo da estratégia* waitany *para a entrada com 500 seqüências*

anteriores. Aqui o gargalo da latência é totalmente eliminado e a comunicação entre os processos é mínima.

Para esta estratégia também foi feita uma extração do perfil de execução. A figura 4.21 mostra o percentual de tempo gasto com o *overhead* de comunicação e sincronismo apenas para o processo principal. O processo principal é aquele que realiza o processamento de fato. O *thread*, por ficar a maior parte do tempo ocioso, esperando o recebimento de uma solicitação de envio de dados, não é mostrado aqui. A etapa em que o *thread* realmente

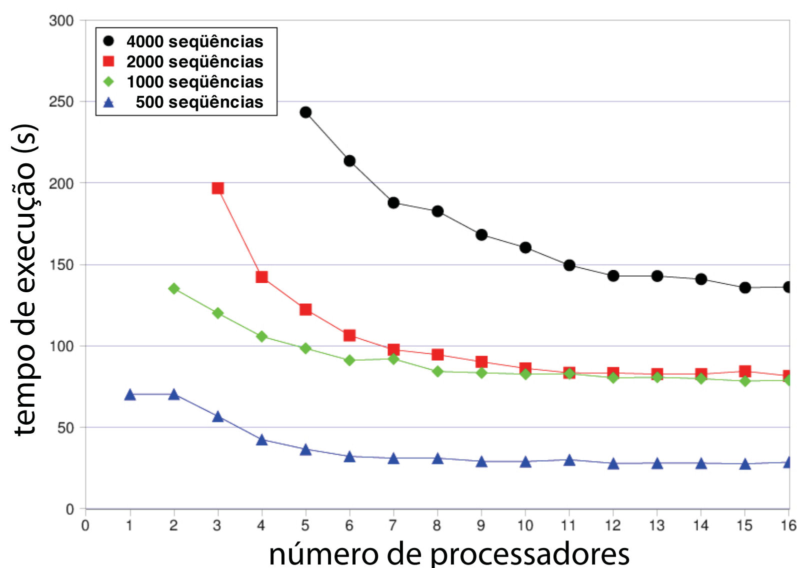

Figura 4.20: *Gráfico de tempo de execução da estratégia com* threads

trabalha é quando este envia os dados aos escravos dependentes. Para isto, o tempo gasto com processamento é mínimo, pois este é necessário apenas para o encapsulamento dos dados para envio.

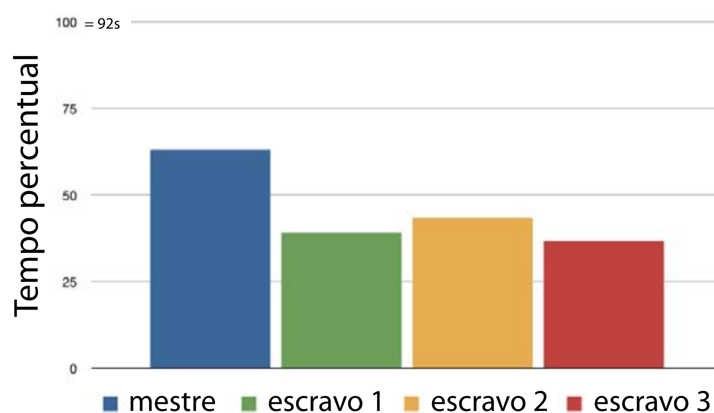

Figura 4.21: *Percentual de tempo gasto com comunicação e sincronismo da estratégia com* threads *para a entrada com 500 seqüências*

Para comparação, é mostrado na figura 4.22 o tempo de execução de todas as estratégias para a entrada com 500 seqüências. A figura 4.23, por sua vez, coloca lado a lado os *speedups* reais de todas as estratégias.

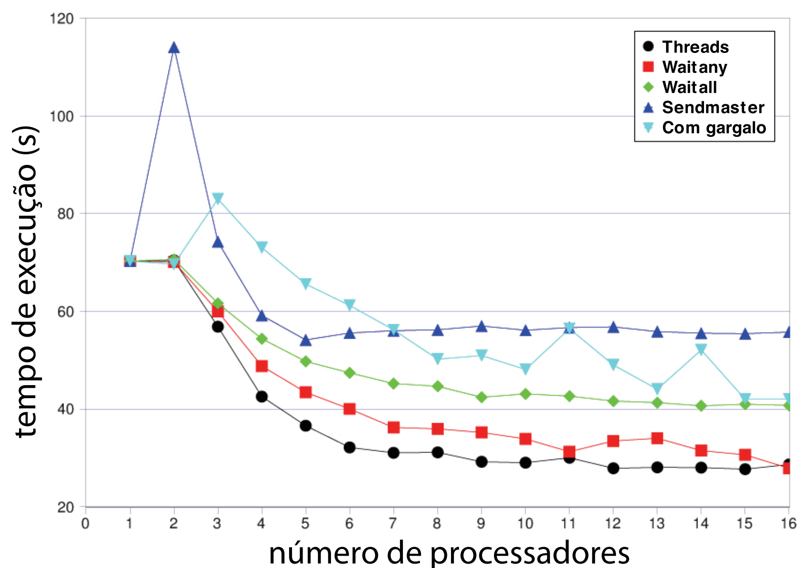

Figura 4.22: *Comparação do tempo de execução das estratégias paralelas do alinhamento progressivo para a entrada com 500 seqüências*

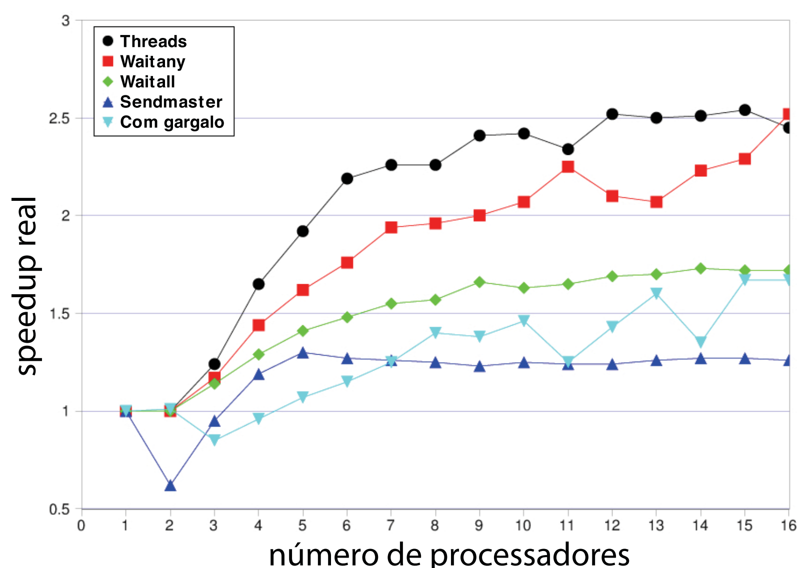

Figura 4.23: *Comparação dos* speedups *reais das estratégias paralelas do alinhamento progressivo para a entrada com 500 seqüências*

Esses algoritmos melhoram o ganho de desempenho conforme aumenta-se o número de seqüências. Com o aumento do número de seqüências, por outro lado, aumenta-se o uso de memória. Este aumento de memória faz com que o algoritmo torne-se executável apenas em múltiplos nós. Para visualizarmos a escalabilidade através de uma entrada maior do

que a entrada da figura anterior, calculou-se o ganho de desempenho em relação ao tempo de execução do algoritmo paralelo com o menor número de nós que habilita a execução. A figura 4.24 mostra o ganho de desempenho para uma entrada com 2000 seqüências, cujo sistema mínimo é de três nós.

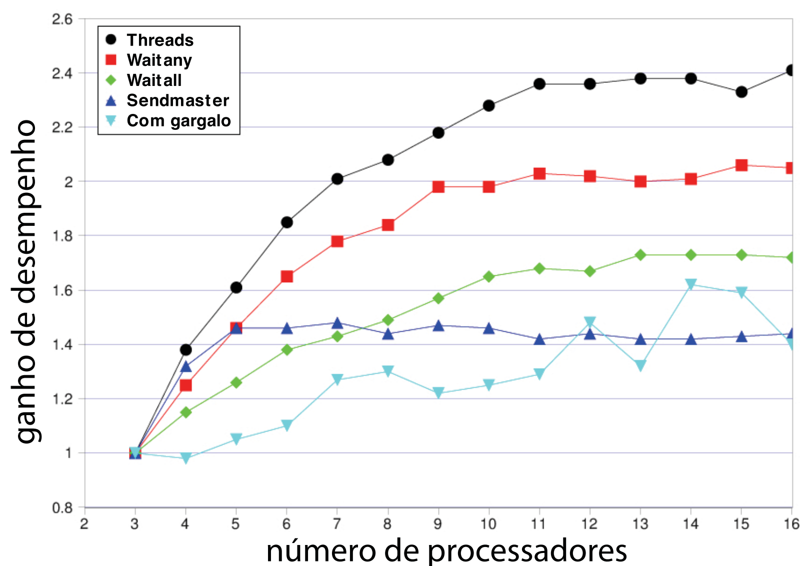

Figura 4.24: *Comparação do ganho de desempenho das estratégias paralelas do alinhamento progressivo para a entrada com 2000 seqüências*

Para mostrar também como o ganho de desempenho varia com a entrada utilizada, selecionamos a melhor estratégia (com *threads*) e calculamos o ganho de desempenho para as quatro entradas. Novamente, o ganho de desempenho é obtido em relação ao tempo de execução do algoritmo em um sistema maior que um nó. Como o maior sistema mínimo para este conjunto de entrada é de cinco nós - execução da entrada com 4000 seqüências -, o ganho de desempenho, variando-se a quantidade de nós e a entrada utilizada, é calculado em relação ao tempo de execução em cinco nós. O gráfico da figura 4.25 mostra os resultados obtidos.

Para mostrar que o algoritmo apresenta um certo grau de escalabilidade, o ganho de desempenho deve aumentar com o aumento da entrada utilizada, o que é visto na figura anterior para as entradas com 2000 e 4000 seqüências. Entretanto, o ganho de desempenho não está unicamente vinculado com o tamanho da entrada utilizada. A dependência entre as tarefas também afeta o paralelismo, e, dependendo do problema, a eficiência do paralelismo

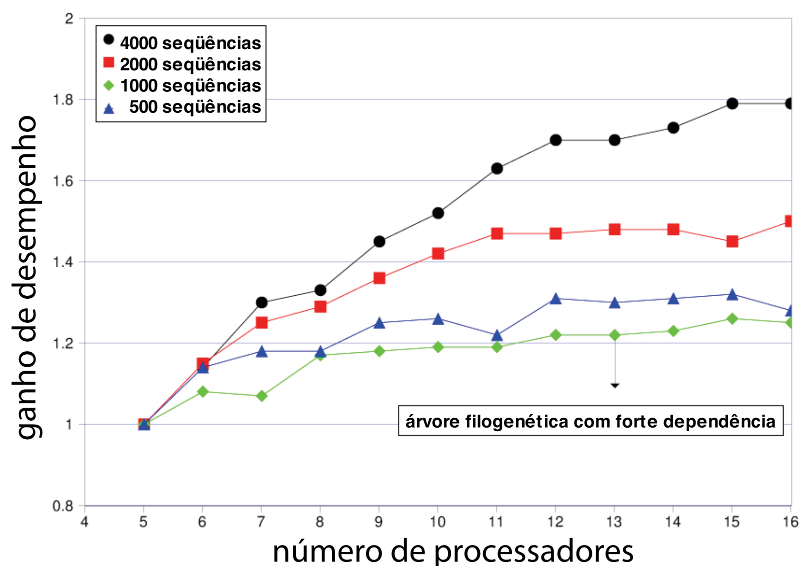

Figura 4.25: *Comparação do ganho de desempenho da estratégia com* threads *para as entradas com 500, 1000, 2000 e 4000 seqüências*

pode variar, aumentando ou diminuindo o ganho de desempenho do algoritmo. É o que acontece com as entradas com 500 e 1000 seqüências. Apesar da entrada aumentar, o ganho de desempenho diminui. Dessa forma, a escalabilidade com a entrada de 1000 seqüências, que deveria ser superior, é inferior.

Essa mesma análise pode ser feita através de uma outra perspectiva, utilizando-se o gráfico de tempo de execução da mesma estratégia (figura 4.20). Com ele é possível ver que a entrada com 500 seqüências apresenta um tempo de execução mais que duas vezes menor do que o tempo de execução com a entrada com 1000 seqüências. Por este mesmo gráfico também vemos que a entrada com 2000 seqüências apresenta um tempo de execução bem próximo do tempo de execução da entrada com 1000 seqüências. Neste caso, dobrou-se a quantidade de dados de entrada, porém manteve-se próximo o tempo de execução do algoritmo. Como será visto adiante, este comportamento é justificado pelo fato do nível de paralelismo da aplicação, em geral, ser aproximadamente o dobro com a entrada de 2000 seqüências. Entende-se pela aplicação uma função que considera o algoritmo, a entrada utilizada (instância do problema) e o número de processadores.

4.3.2 O nível de paralelismo

O nível de paralelismo depende do algoritmo, da entrada utilizada e da quantidade de processadores. Para o mesmo algoritmo e mesmo número de processadores, o nível de paralelismo pode mudar de acordo com a instância do problema. O motivo é que a definição de vários estágios de um algoritmo paralelo é, em geral, feita dinamicamente. Por exemplo, a maioria dos algoritmos paralelos de alinhamento progressivo definem o particionamento em nível de nó de árvore e, portanto, este estágio é estático. Entretanto, utilizando-se este particionamento, a comunicação e o escalonamento podem variar, uma vez que a dependência das tarefas primitivas é definida por uma estrutura intrínseca ao problema. No caso do alinhamento progressivo com particionamento em nível de nó de árvore, quem define a dependência das tarefas é a árvore filogenética, que é dependente das seqüências de entrada (ver seção 2.6.3). Essa dependência pode variar desde o caso em que todos os nós da árvore possuem apenas um único filho (exceto os nós folhas) até o caso em que a árvore se encontra balanceada.

Uma maior quantidade de processadores, por sua vez, possibilita uma menor aglomeração das tarefas que, em um certo instante, estão prontas para o processamento. Entretanto, dependendo da granularidade atingida, o aumento de processadores pode não proporcionar melhoras de desempenho. Em piores casos, este aumento pode degradar o desempenho. A definição de uma boa estratégia de aglomeração sempre é feita dinamicamente e associada com boas estratégias de comunicação e escalonamento.

Portanto, dos quatro estágios definido por Foster [8], apenas o particionamento é em geral definido como estático na maioria dos algoritmos paralelos. Todos os outros estágios são definidos dinamicamente e, portanto, variam de acordo com a instância do problema e com o número de processadores.

O nível de paralelismo então foi definido como uma medida que nos mostra quão perto de um paralelismo ótimo um algoritmo paralelo, que calcule estaticamente o particionamento, pode chegar, utilizando-se uma instância específica do problema e um número específico de processadores. Esta medida é obtida considerando um ambiente paralelo perfeito, em que não existam custos com a comunicação e a carga esteja sempre balanceada. Ou seja, um algoritmo com estratégias ótimas de comunicação, aglomeração e escalonamento.

Neste caso o desempenho é unicamente afetado por características intrínsecas ao problema e pela variação no número de processadores.

A tabela 4.1 mostra como o aumento do número de processadores aumenta o nível de paralelismo para o problema do alinhamento progressivo do MUSCLE com particionamento em nível de nó de árvore. Este número, porém, tende a se estabilizar em um certo ponto. É neste ponto que a entrada utilizada passa a ser o gargalo do paralelismo, impossibilitando que o aumento do número de processadores traga algum benefício no desempenho. Como explicado, este gargalo se deve à limitação na quantidade de tarefas que podem ser escalonadas simultaneamente, limitação esta inerente ao módulo do problema que define as dependências entre as tarefas. No alinhamento progressivo do MUSCLE, este módulo é o de construção da árvore filogenética.

Número	**Nível de paralelismo**			
de escravos	500 seq	1000 seq	2000 seq	4000 seq
1	1	1	x	x
2	1.92	1.75	1.99	x
3	2.71	2.35	2.95	x
4	3.28	2.78	3.87	3.79
5	3.75	3.15	4.66	4.61
6	4.09	3.42	5.33	5.41
7	4.34	3.61	6.04	6.12
8	4.5	3.78	6.55	6.8
9	4.58	3.89	7.04	7.46
10	4.66	4	7.43	8.08
11	4.75	4.08	7.84	8.62
12	4.8	4.13	8.13	9.15
13	4.84	4.18	8.33	9.64
14	4.89	4.23	8.58	10.05
15	4.89	4.27	8.77	10.47

Tabela 4.1: *Nível de paralelismo com o uso da árvore normal produzida pelo MUSCLE para as entradas com 500, 1000, 2000 e 4000 seqüências*

4.3.3 A árvore filogenética e a escalabilidade do algoritmo

A árvore de dependência de tarefas no alinhamento progressivo é baseada na árvore filogenética previamente construída. Dependendo de sua estrutura, uma maior quantidade

de tarefas pode ou não ser executada em paralelo. Uma árvore que apresenta a melhor estrutura para o paralelismo é a árvore balanceada. Esta árvore representa o melhor caso pois ela apresenta uma maior quantidade de nós independentes uns dos outros em relação às demais árvores. Entretanto, uma árvore filogenética tem um significado biológico, e é construída a partir de um método específico. Quanto mais próxima de uma árvore balanceada estiver a arvore gerada, maior será o nível de paralelismo no alinhamento progressivo.

Para mostrarmos como o desempenho de um algoritmo varia com a estrutura da árvore gerada, escolhemos primeiro uma instância do problema. Sobre esta instância, comparamos o nível de paralelismo obtido com o uso de uma árvore balanceada, obtida a partir da versão modificada do algoritmo, e com o uso de uma árvore normal. Neste caso, a árvore normal é a arvore gerada pelo método UPGMA do MUSCLE. Realizamos testes com estratégia com *threads* e uma entrada de 1000 seqüências, variando-se o numero de processadores. O tempo de execução de ambas as árvores é mostrado na figura 4.26. Calculamos também o *speedup* real, mostrado na figura 4.27.

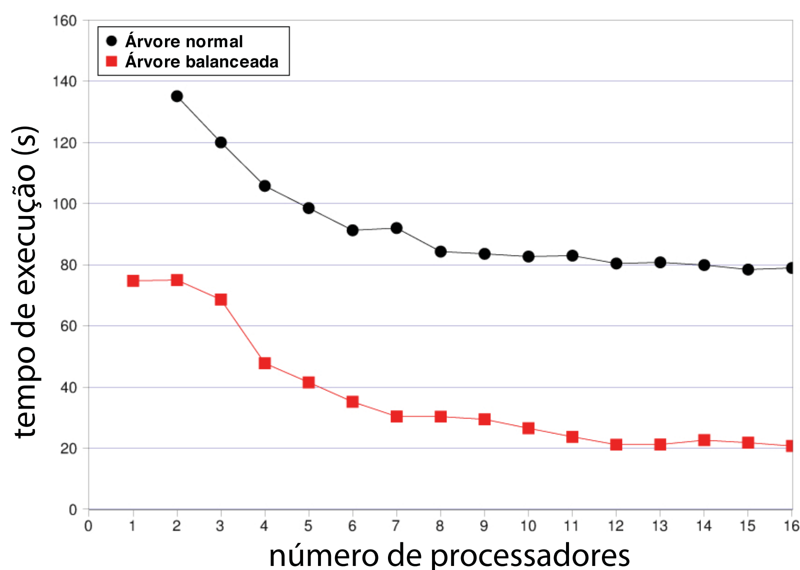

Figura 4.26: *Comparação do tempo de execução da estratégia com* threads *com a árvore balanceada e a árvore normal para a entrada com 1000 seqüências*

Através da figura 4.27 pode-se ver como o algoritmo torna-se mais escalável com o uso da árvore balanceada. Essa escalabilidade é maior devido ao maior número de tarefas executadas simultaneamente. A tabela 4.2 mostra uma comparação dos níveis de paralelismo

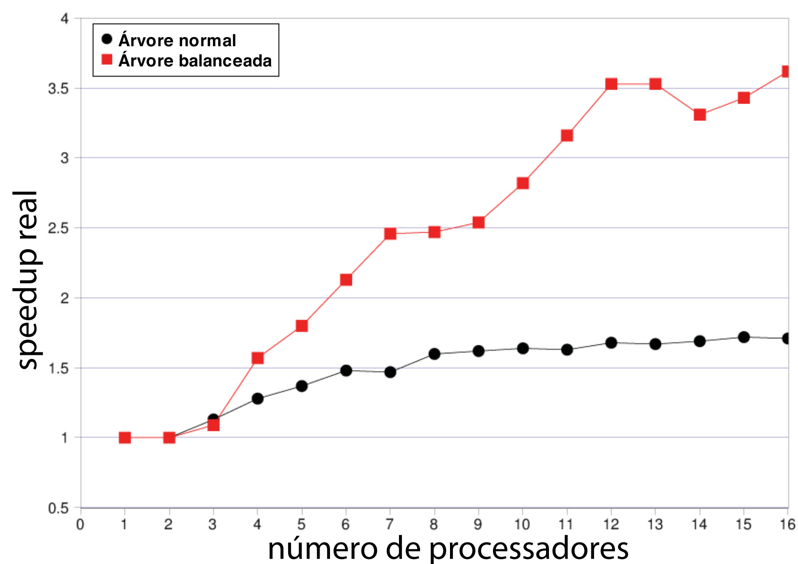

Figura 4.27: *Comparação do* speedup *real da estratégia com* threads *com a árvore balanceada e a árvore normal para a entradas com 1000 seqüências*

obtidos com o uso da árvore normal e com o uso da árvore balanceada.

Número de escravos	**Nível de paralelismo** Balanceada	Não balanceada
1	1	1
2	2	1.75
3	2.99	2.35
4	3.98	2.78
5	4.95	3.15
6	5.91	3.42
7	6.89	3.61
8	7.87	3.78
9	8.76	3.89
10	9.7	4
11	10.63	4.08
12	11.62	4.13
13	12.49	4.18
14	13.32	4.23
15	14.27	4.27

Tabela 4.2: *Comparação do nível de paralelismo com o uso da árvore normal produzida pelo MUSCLE e da árvore balanceada para a entrada com 1000 seqüências*

Além do aumento do nível de paralelismo, realizar o alinhamento em uma árvore ba-

lanceada é uma operação menos custosa. Isto é notado observando o gráfico de tempo de execução do alinhamento com apenas dois processadores (figura 4.26). Para dois processadores, a quantidade de tarefas independentes não influencia no tempo de execução, pois as tarefas são executadas no único processador escravo disponível. Além disso, a quantidade de tarefas de alinhamento é a mesma independentemente da árvore gerada. Realizar o alinhamento em uma árvore balanceada é uma operação menos custosa devido a existência de um maior número de tarefas de alinhamento envolvendo perfis menores. Esses perfis menores contém uma menor quantidade de dados a serem processados.

Consequentemente, o uso de memória é reduzido, possibilitando sua execução em clusters menores. Na figura 4.26 vemos que a entrada com 1000 seqüências só pode ser executada a partir de dois nós com uma árvore normal. Já o sistema mínimo para executar a mesma entrada, porém com uma árvore balanceada, é de apenas um nó, habilitando a execução do algoritmo seqüencial.

Portanto, a topologia da árvore filogenética afeta o desempenho do algoritmo em diferentes maneiras. Quanto mais perto a árvore está de uma condição balanceada, maior é o nível de paralelismo, menor é o esforço computacional e menores são os requisitos de memória.

4.4 Alinhamento par-a-par

Os testes a seguir mostram o tempo de execução das abordagens paralelas do alinhamento par-a-par. Como a etapa do alinhamento par-a-par é realizada nos dois estágios progressivos e no estágio iterativo do MUSCLE, seu algoritmo possui algumas diferenças de implementação. Essas diferenças não afetam o desempenho do algoritmo pois a abordagem em todos os estágios é a mesma. Como a intenção é avaliar o desempenho das estratégias paralelas construídas sobre a abordagem de alinhamento par-a-par da ferramenta MUSCLE, os testes aqui apresentados foram todos realizados no primeiro estágio progressivo. O comportamento da execução paralela no segundo estágio progressivo e no estágio iterativo é o mesmo e, portanto, não será mostrado.

Ao todo três estratégias foram definidas. Para cada algoritmo, os testes foram realiza-

dos com várias entradas distintas, variando-se o número de processadores.

4.4.1 Comparação entre as estratégias

O desenvolvimento das três estratégias implementadas foi baseado no algoritmo *block-based wavefront*, proposto em [44]. Este algoritmo divide a computação da matriz dinâmica, porém não apresenta uma técnica para coletar os resultados da execução e gerar o resultado final, que é o caminho do alinhamento. Neste projeto, três possíveis técnicas foram implementadas.

A primeira delas faz a coleta de todos os dados de uma só vez no final da execução do algoritmo *block-based wavefront*. Esses dados são todos os elementos da matriz de programação dinâmica. A figura 4.28 mostra o perfil de execução desta técnica para o alinhamento de duas seqüências de aproximadamente 1000 resíduos em três nós de execução. Essa extração de perfil foi feita através da ferramenta Jumpshot [35]. Através dessa figura é possível ver como é grande o *overhead* de comunicação e sincronismo. Como vários escravos tentam quase que ao mesmo tempo enviar dados ao processo mestre, estes escravos mantém-se esperando, em uma função de envio de dados, até que o processo mestre atenda todas as requisições de envio. A espera é relativamente longa pois a quantidade de dados enviados é $O(NxM)$, onde N e M são os tamanhos das seqüências.

A figura 4.29 mostra o percentual de tempo gasto com a comunicação e sincronismo para uma entrada com cinco seqüências de aproximadamente 3000 resíduos cada. Utilizou-se quatro nós de execução neste teste. Note que apenas uma pequena parte do tempo é destinado ao processamento. A maior parte corresponde ao *overhead* com o sincronismo e a comunicação.

Visando minimizar o tempo gasto com este sincronismo, implementou-se uma segunda estratégia em que os dados dos processos escravos são enviados em partes para o processo mestre. Esta técnica utiliza melhor a rede, porém aumenta-se o custo com a comunicação. A figura 4.30 mostra o perfil de execução desta técnica para o alinhamento de duas seqüências de aproximadamente 1000 resíduos em apenas três nós de execução. Em relação a primeira técnica, o tempo total da execução desta entrada foi ligeiramente menor.

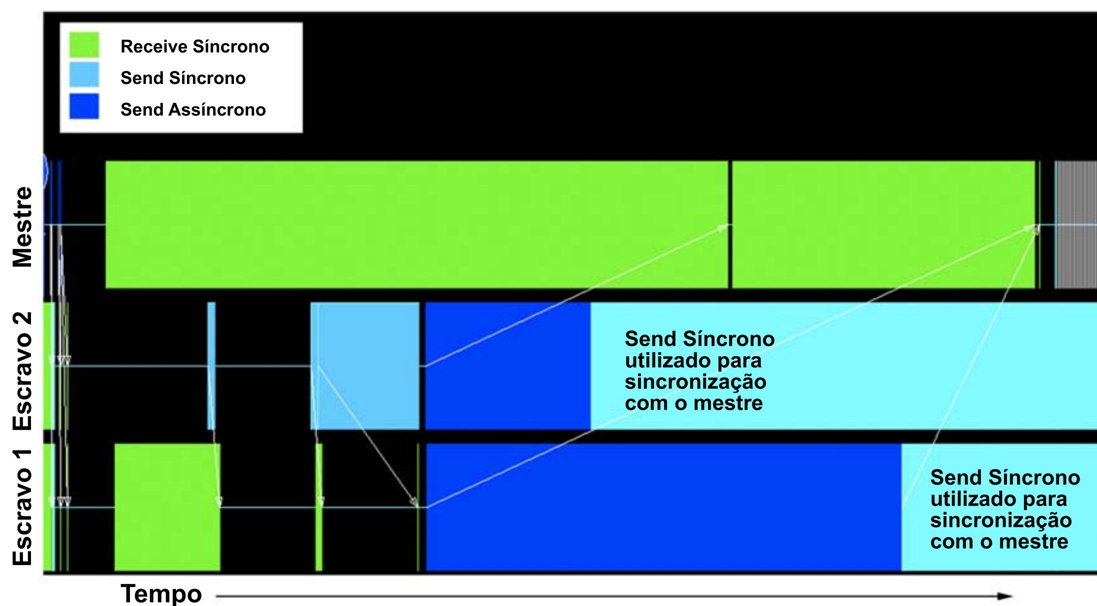

Figura 4.28: *Perfil da execução em três nós da estratégia que envia os dados após todos serem computados para uma entrada de duas seqüências de aproximadamente 1000 resíduos*

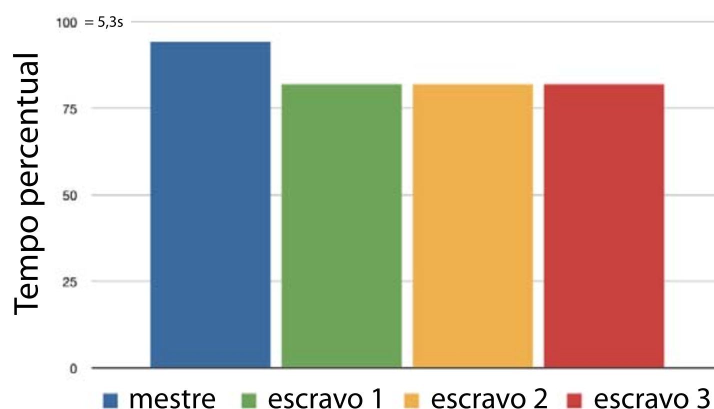

Figura 4.29: *Percentual de tempo gasto com comunicação e sincronismo da estratégia que envia dados após todos serem computados para uma entrada de cinco seqüências de aproximadamente 3000 resíduos*

A figura 4.31, por sua vez, mostra o percentual de tempo gasto com a comunicação e sincronismo. Utilizou-se neste teste quatro nós de execução e a entrada de cinco seqüências de aproximadamente 3000 resíduos. As características apresentadas são semelhantes às da técnica anterior (figura 4.29), não apresentando, portanto, uma melhora significante no

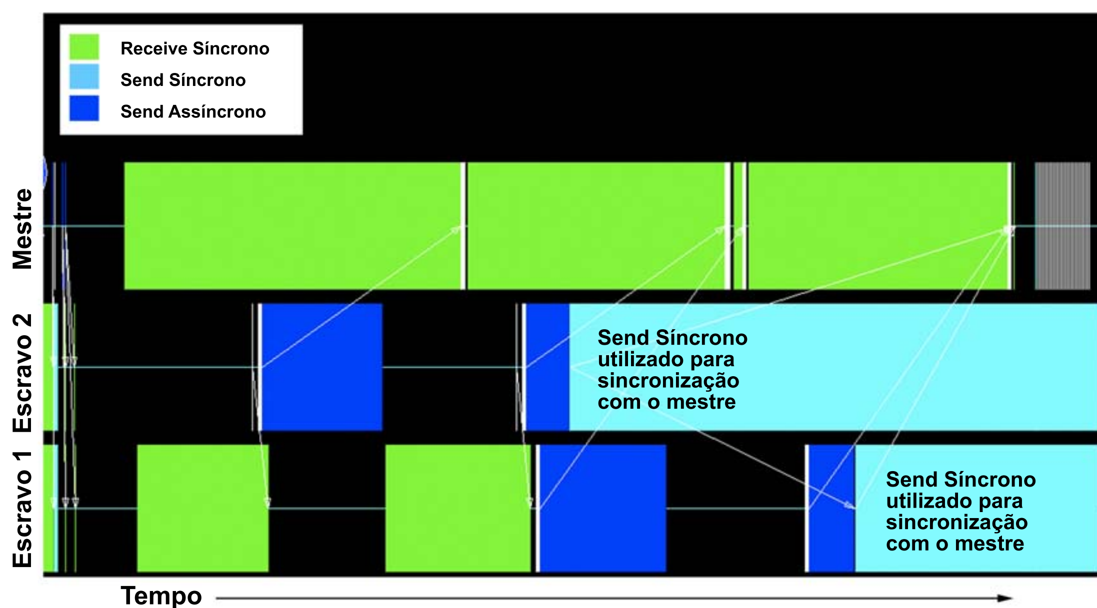

Figura 4.30: *Perfil da execução em três nós da estratégia que envia dados em partes para uma entrada de duas seqüências de aproximadamente 1000 resíduos*

tempo de execução total do alinhamento.

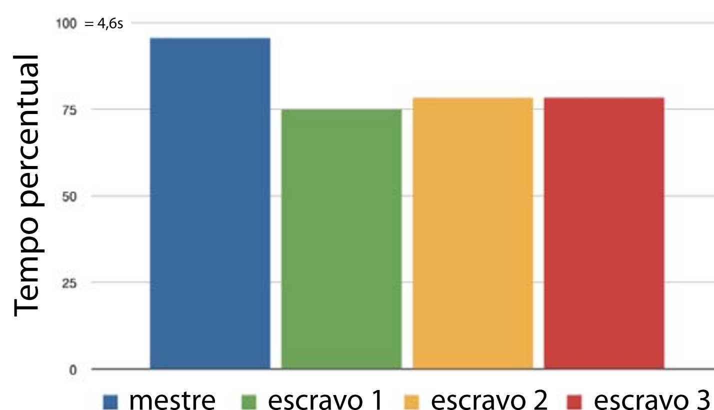

Figura 4.31: *Percentual de tempo gasto com comunicação e sincronismo da estratégia que envia dados em pedaços para uma entrada de cinco seqüências de aproximadamente 3000 resíduos*

A terceira e última técnica, ao invés de apresentar uma forma alternativa de reunir os dados da matriz, que estão distribuídos, calcula o caminho do alinhamento nessa matriz sobre os dados distribuídos, realizando uma parte do processamento em cada processo,

de acordo com a disponibilidade dos dados. Este caminho, então, é enviado ao processo mestre que o utiliza para o cálculo do novo alinhamento (estágio iterativo) ou perfil do alinhamento (estágio progressivo). A figura 4.32 mostra o perfil de execução desta técnica para o alinhamento de duas seqüências de 1000 resíduos em três nós de execução. Além de eliminar os custos com sincronismo e comunicação, o cálculo distribuído do caminho de alinhamento mostrou-se extremamente rápido.

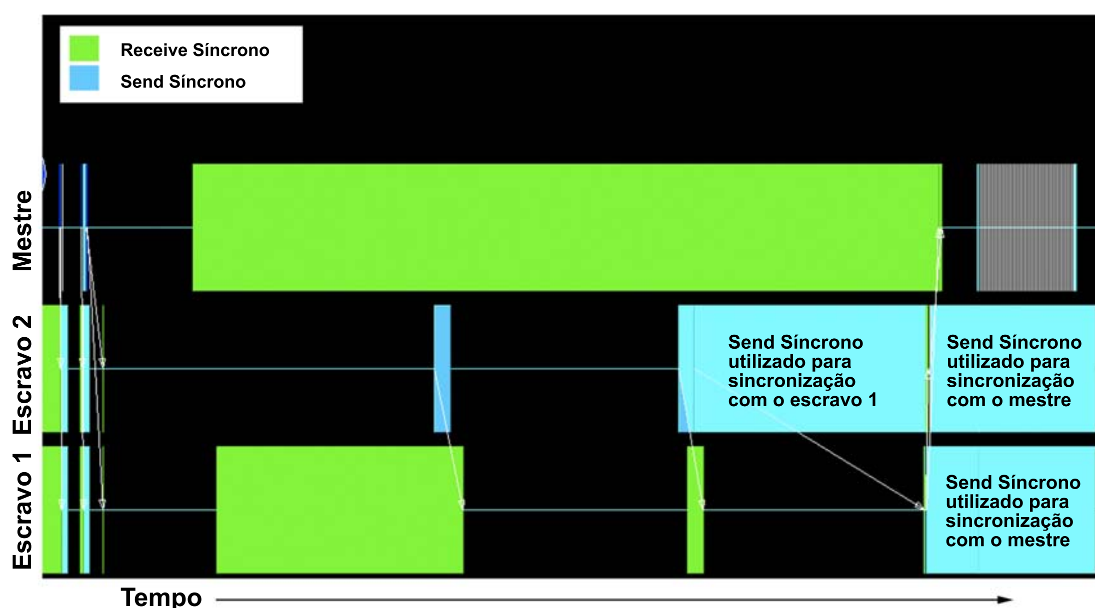

Figura 4.32: *Perfil da execução em três nós da estratégia que paraleliza o método de construção do caminho de alinhamento para uma entrada de duas seqüências de aproximadamente 1000 resíduos*

A figura 4.33 mostra o percentual de tempo gasto com a comunicação e o sincronismo, alinhando-se cinco seqüências de aproximadamente 3000 resíduos em quatro nós de execução. Como pode ser visto, aqui a maior parte do tempo é gasto com a execução e não com a espera por sincronismo/comunicação.

Em seguida, foram feitos testes com o intuito de comparar o tempo de execução dos três algoritmos. Para isso, utilizou-se duas entradas de 100 seqüências: uma de comprimento médio igual a 1000 e outra de comprimento médio igual a 5000. As figuras 4.34 e 4.35 mostram os resultados obtidos.

Percebe-se que a execução em paralelo das duas primeiras estratégias não é vantajosa

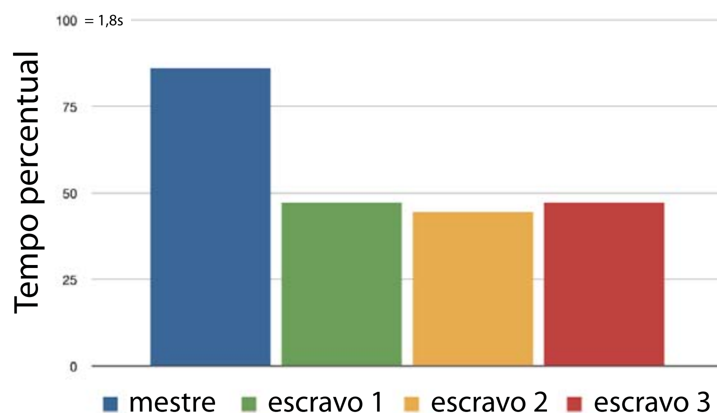

Figura 4.33: *Percentual de tempo gasto com comunicação e sincronismo da estratégia que paraleliza o método de construção do caminho de alinhamento para uma entrada de cinco seqüências de aproximadamente 3000 resíduos*

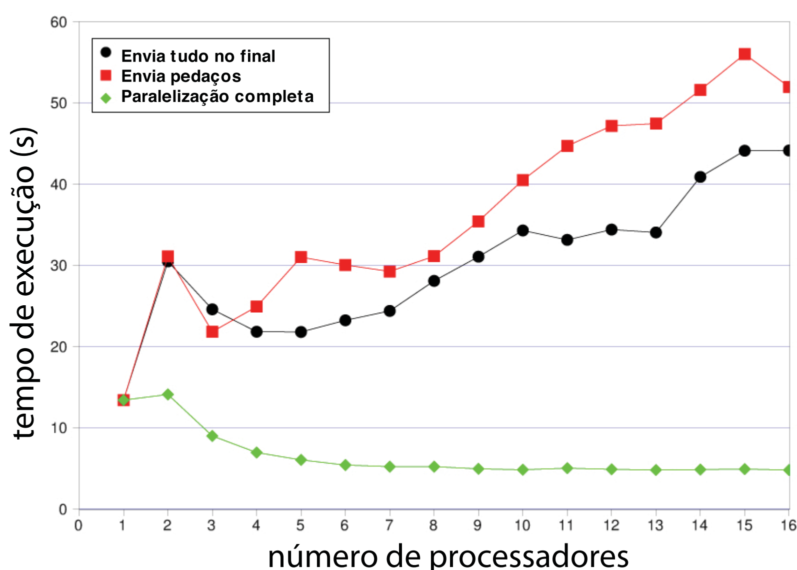

Figura 4.34: *Comparação do tempo de execução das três estratégias paralelas do alinhamento par-a-par para entradas com sequencias de aproximadamente 1000 resíduos*

com seqüências de poucos resíduos. O *overhead* gerado por elas sobrepõe o ganho com a divisão do processamento e este comportamento piora conforme aumenta-se o número de nós. Já para a terceira estratégia, o *overhead* é pequeno e, apesar do paralelismo ser melhor com seqüências de maior tamanho, utilizar o algoritmo paralelo para seqüências pequenas também pode trazer uma melhora significante. Entretanto, em qualquer um dos

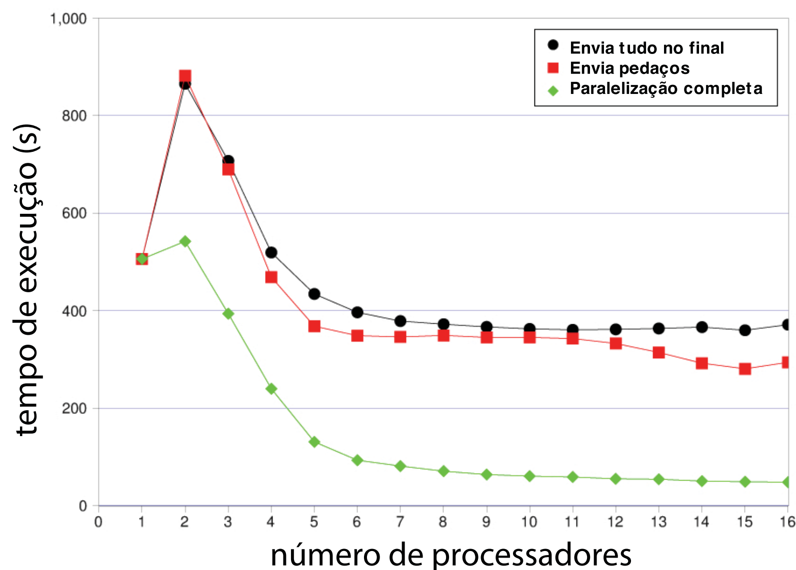

Figura 4.35: *Comparação do tempo de execução das três estratégias paralelas do alinhamento par-a-par para entradas com seqüências de aproximadamente 5000 resíduos*

casos, o tempo de execução sempre é menor com o uso da terceira estratégia.

Por fim, realizou-se um último teste para mostrar o impacto do paralelismo variando-se o tamanho das seqüências. Este teste utilizou a melhor estratégia e entradas com o mesmo número de seqüências. O comprimento médio das seqüências em cada entrada varia de 1000 à 5000. A figura 4.36 mostra o *speedup* real obtido com vários nós de execução.

Como mostra a figura este algoritmo apresenta sinais de escalabilidade. Com o aumento do tamanho do problema, aumenta-se o ganho de desempenho para uma mesma configuração de sistema.

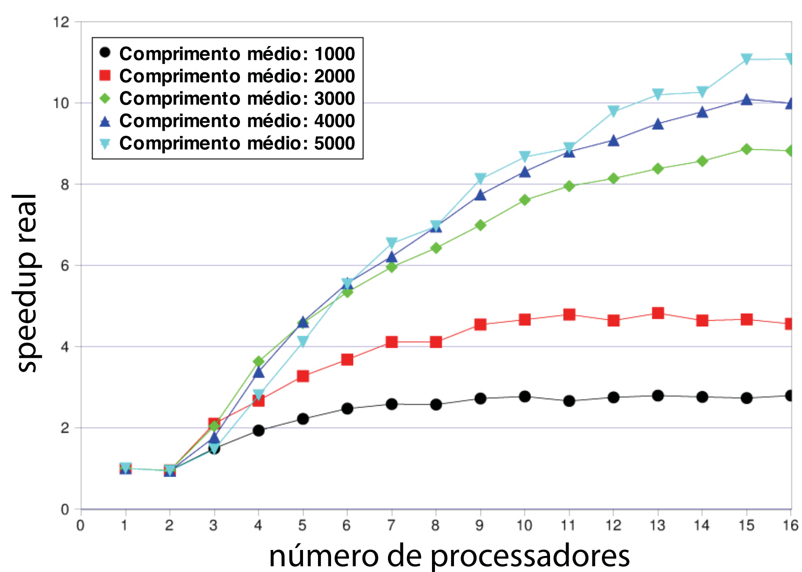

Figura 4.36: *Comparação do* speedup *real da estratégia que paraleliza o método de construção do caminho de alinhamento para entradas com seqüências de aproximadamente 1000, 2000, 3000, 4000 e 5000 resíduos*

5 Conclusões

O uso da computação paralela em ferramentas de alinhamento múltiplo de seqüências apresenta uma demanda crescente, visto o número cada vez maior de seqüências utilizadas para comparação. Muitos trabalhos propõem o paralelismo de técnicas isoladas, de uso impraticável pela maioria dos biólogos. Poucas são as ferramentas completas paralelizadas e que consideram aspectos fundamentais da computação paralela. Encontrou-se na literatura abordagens que, segundo a metodologia de Foster [8], não consideram aspectos como o balanceamento da carga e a latência. A paralelização proposta contorna este problema possibilitando uma fácil execução de problemas reais em sistemas reais com eficiência e escalabilidade, através da paralelização de uma ferramenta de alinhamento múltiplo muito bem aceita, o MUSCLE.

Identificamos, primeiramente, várias formas de paralelizarmos os estágios de execução do MUSCLE. Em seguida, definimos possíveis estratégias, baseadas ou não em abordagens existentes. Nas estratégias em que identificamos gargalos de execução, trabalhou-se sobre uma otimização em busca da diminuição ou eliminação completa desses gargalos. Com isso, novas estratégias foram criadas e comparadas com o que já foi proposto. A melhor estratégia foi incorporada como padrão na versão paralela da ferramenta MUSCLE. Ela, entretanto, não está restrita ao código do MUSCLE, podendo ser adaptada em ferramentas que abordam, de uma forma similar, um mesmo sub-problema de alinhamento.

Ao todo, dez estratégias paralelas foram desenvolvidas: duas na etapa de construção da matriz de distância, cinco na etapa de alinhamento progressivo e três na etapa de alinhamento par-a-par. A estratégia do alinhamento par-a-par, em particular, pode ser utilizada em algoritmos progressivos. O alinhamento par-a-par é um subproblema do alinhamento progressivo, o que possibilita a execução do problema maior em uma menor granularidade.

Todas as adaptações foram feitas no código do MUSCLE. A escolha das estratégias é feita através de chamadas de execução, utilizando a mesma interface com o usuário da ferramenta MUSCLE original. Os testes apresentam o tempo de execução e ganho de desempenho com cada estratégia para vários nós de execução. Também é feita uma análise dos tipos de entrada utilizada, mostrando como a árvore de dependência de tarefas primitivas afeta o desempenho do algoritmo.

Os resultados são muito satisfatórios. Ao menos uma estratégia definida e que pode ser incorporada em uma etapa específica do MUSCLE apresenta melhores resultados no tempo de execução e no uso de memória em relação a estratégias já existentes. O algoritmo do MUSCLE paralelo completo, entretanto, não foi totalmente mensurado e comparado com seu algoritmo seqüencial, uma vez que esta análise é muito restrita a problemas pequenos, devido às limitações de memória do sistema seqüencial. Adicionalmente, vários são os caminhos de execução, definidos pelo usuário na chamada de execução. Pelo mesmo motivo, e pela falta de um sistema de memória compartilhada, não foi feita uma comparação com o MUSCLE-SMP. Já com o CLUSLTAW-MPI, uma comparação não faz muito sentido, uma vez que o resultado obtido é muito diferente e o CLUSTALW-MPI não apresenta um estágio de refinamento. Fez-se uma apresentação de todas as paralelizações propostas por essas ferramentas, e que são incorporadas em algumas etapas do algoritmo, e comparou-se a abordagem que eles adotam com a abordagem proposta neste trabalho. Como já dito, melhores resultados foram obtidos com as paralelizações definidas neste trabalho.

5.1 Trabalhos futuros

Identificou-se alguns possíveis caminhos para a continuidade deste trabalho. O primeiro consiste na incorporação de um mecanismo automático de definição de tamanho dos blocos (grãos) da matriz de programação dinâmica para ser utilizado no algoritmo do alinhamento par-a-par baseado na estratégia *block-based wavefront*. Como o tamanho mínimo ideal de um bloco depende de fatores como velocidade de processamento e comunicação de um sistema, a obtenção de medidas de desempenho críticas através de testes rápidos no início da execução do algoritmo pode conduzir a uma melhor definição do tamanho da granularidade.

Um outro caminho é resolver o problema do alinhamento múltiplo progressivo através de uma abordagem mista, utilizando a melhor estratégia de alinhamento progressivo proposta (seção 3.3.7) e a melhor estratégia do alinhamento par-a-par proposta (seção 3.4.1). Essa abordagem consideraria no início de sua execução uma granularidade maior. Cada nó da árvore seria uma tarefa primitiva. Utiliza-se uma granularidade maior uma vez que, no início, mais tarefas podem ser executadas simultaneamente. Identificaria-se, então, o momento, na árvore de tarefas, em que tal granularidade habilita a ocorrência de ociosidade, levando à um desbalanceamento de carga. Essa ociosidade ocorre uma vez que a dependência das tarefas é maior quando elas estão mais próximas da tarefa raiz. A partir deste ponto, utilizaria-se uma granularidade menor, em nível de bloco da matriz.

Ambos os caminhos possuem uma série de aplicações, não restritas apenas ao código do MUSCLE. O primeiro caminho visa melhorar o paralelismo de técnicas de alinhamento par-a-par, enquanto que o segundo visa uma melhora no paralelismo do alinhamento progressivo. Como o alinhamento par-a-par é um subproblema do alinhamento progressivo que está sendo proposto, um terceiro caminho pode ser a unificação de ambos os caminhos.

Referências Bibliográficas

1 EDGAR, R. C. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res*, bob@drive5.com, v. 32, n. 5, p. 1792–1797, 2004. ISSN 1362-4962. Disponível em: <http://view.ncbi.nlm.nih.gov/pubmed/15034147>.

2 EDGAR, R. C.; BATZOGLOU, S. Multiple sequence alignment. *Current Opinion in Structural Biology*, v. 16, n. 3, p. 368–373, June 2006. Disponível em: <http://dx.doi.org/10.1016/j.sbi.2006.04.004>.

3 DENG, X. et al. Parallel implementation and performance characterization of muscle. *Parallel and Distributed Processing Symposium, International*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 359, 2006.

4 THOMPSON, J. D.; HIGGINS, D. G.; GIBSON, T. J. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, European Molecular Biology Laboratory, Heidelberg, Germany., v. 22, n. 22, p. 4673–4680, November 1994. ISSN 0305-1048. Disponível em: <http://dx.doi.org/10.1093/nar/22.22.4673>.

5 JUAN, D.; PAZOS, F.; VALENCIA, A. High-confidence prediction of global interactomes based on genome-wide coevolutionary networks. *Proceedings of the National Academy of Sciences*, p. 0709671105+, January 2008. Disponível em: <http://dx.doi.org/10.1073/pnas.0709671105>.

6 WONG, K. M.; SUCHARD, M. A.; HUELSENBECK, J. P. Alignment uncertainty and genomic analysis. *Science*, v. 319, n. 5862, p. 473–476, January 2008. Disponível em: <http://dx.doi.org/10.1126/science.1151532>.

7 FAVIA, A. D. et al. Molecular docking for substrate identification: the short-chain dehydrogenases/reductases. *J Mol Biol*, European Molecular Biology Laboratory-European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton, Cambridge CB10 1SD, UK. afavia@ebi.ac.uk, v. 375, n. 3, p. 855–874, January 2008. ISSN 1089-8638. Disponível em: <http://dx.doi.org/10.1016/j.jmb.2007.10.065>.

8 FOSTER, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201575949.

9 OGATA, S. O. *Alinhamento de seqüências biológicas com o uso de algoritmos genéticos.* Dissertação (Mestrado) — UFSCAR - Universidade Federal de São Carlos, 2005.

10 NEEDLEMAN, S.; WUNSCH, C. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, v. 48, n. 3, p. 443–53, 1970.

11 SMITH, T. F.; WATERMAN, M. S. Identification of common molecular subsequences. *Journal of Molecular Biology*, v. 147, p. 195–197, 1981. Disponível em: <citeseer.ist.psu.edu/smith81identification.html>.

12 LUO, J. et al. Parallel multiple sequence alignment with dynamic scheduling. In: *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I.* Washington, DC, USA: IEEE Computer Society, 2005. p. 8–13. ISBN 0-7695-2315-3.

13 FENG, D. F.; DOOLITTLE, R. F. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J Mol Evol*, Department of Chemistry, University of California-San Diego, La Jolla 92093., v. 25, n. 4, p. 351–360, 1987. ISSN 0022-2844. Disponível em: <http://view.ncbi.nlm.nih.gov/pubmed/3118049>.

14 TAYLOR, W. A flexible method to align large numbers of biological sequences. *J. Mol. Evol.*, v. 28, p. 161–169, 1988.

15 EDGAR, R. C.; SJöLANDER, K. A comparison of scoring functions for protein sequence profile alignment. *Bioinformatics*, Oxford University Press, Oxford, UK, v. 20, n. 8, p. 1301–1308, 2004. ISSN 1367-4803.

16 WALLACE, I. M.; ORLA, O.; HIGGINS, D. G. Evaluation of iterative alignment algorithms for multiple alignment. *Bioinformatics*, Oxford University Press, v. 21, n. 8, p. 1408–1414, April 2005. ISSN 1367-4803. Disponível em: <http://dx.doi.org/10.1093/bioinformatics/bti159>.

17 EDGAR, R. C. Muscle: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, Department of Plant and Microbial Biology, 461 Koshland Hall, University of California, Berkeley, CA 94720-3102, USA. bob@drive5.com, v. 5, n. 1, August 2004. ISSN 1471-2105. Disponível em: <http://dx.doi.org/10.1186/1471-2105-5-113>.

18 EDGAR, R. Local homology recognition and distance measures in linear time using compressed amino acid alphabets. *Nucleic Acids Res*, v. 32, p. 380–385, 2004.

19 LAWLER, E. L.; WOOD, D. E. Branch-and-bound methods: A survey. *Operations Research*, v. 14, n. 4, p. 699–719, 1966.

20 YU, K.-M. et al. Parallel branch-and-bound algorithm for constructing evolutionary trees from distance matrix. In: *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*. Washington, DC, USA: IEEE Computer Society, 2005. p. 66. ISBN 0-7695-2486-9.

21 HIGGS, P. G.; ATTWOOD, T. K. *Bioinformatics and molecular evolution*. [S.l.]: Blackwell Publishing, 2005. 384 p.

22 DURBIN, R. et al. *Biological Sequence Analysis : Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999. Paperback. ISBN 0521629713. Disponível em: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0521629713>.

23 SETUBAL, J.; MEIDANIS, J. *Introduction to Computational Molecular Biology*. [S.l.]: PWS Publishing, 1997.

24 QUINN, M. J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education (ISE Editions), 2003. Paperback. ISBN 0071232656. Disponível em: <http://www.amazon.fr/exec/obidos/redirect?tag=citeulike06-21&path=ASIN/0071232656>.

25 GABRIEL, E. et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary: [s.n.], 2004. p. 97–104.

26 FORUM, M. P. I. *MPI: A Message Passing Interface Standard*. Junho 1995. `http://www.mpi-forum.org/`.

27 FORUM, M. P. I. *MPI-2: Extensions to the Message Passing Interface*. Julho 1997. `http://www.mpi-forum.org/`.

28 GRAHAM, R. L.; WOODALL, T. S.; SQUYRES, J. M. Open MPI: A flexible high performance MPI. In: *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*. Poznan, Poland: [s.n.], 2005.

29 SUN, X.-H.; NI, L. M. Another view on parallel speedup. In: *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990. p. 324–333. ISBN 0-89791-412-0.

30 BERTSEKAS, D. et al. Parallel computing in network optimization. In: *In Handbooks in Operations Research*. [S.l.: s.n.], 1995. p. 330–399.

31 KARRELS, E.; LUSK, E. Performance analysis of MPI programs. In: DONGARRA, J.; TOURANCHEAU, B. (Ed.). *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing*. [S.l.]: SIAM Publications, 1994. p. 195–200.

32 CHAN, A.; GROPP, W.; LUSK, E. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 16, n. 2-3, p. 155–165, 2008. ISSN 1058-9244.

33 LUSK, E.; CHAN, A. Early experiments with the OpenMP/MPI hybrid programming model. In: EIGENMANN, R.; SUPINSKI, B. R. de (Ed.). *OpenMP in a New Era of Parallelism*. [S.l.]: Springer, 2008. (Lecture Notes in Computer Science, v. 5004), p. 36–47. IWOMP, 2008.

34 WU, C. E. et al. From trace generation to visualization: A performance framework for distributed parallel systems. *SC Conference*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 50, 2000. ISSN 1063-9535.

35 ZAKI, O. et al. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, v. 13, n. 2, p. 277–288, Fall 1999.

36 TRELLES, O. On the parallelization of bioinformatic applications. *Briefings in Bioinformatics*, v. 2, p. 181–194, 2001.

37 LI, K.-B. Clustalw-mpi: Clustalw analysis using distributed and parallel computing. *Bioinformatics*, v. 19, p. 1585–1586, 2003.

38 BOUKERCHE, A. et al. Parallel strategies for the local biological sequence alignment in a cluster of workstations. *J. Parallel Distrib. Comput.*, Academic Press, Inc., Orlando, FL, USA, v. 67, n. 2, p. 170–185, 2007. ISSN 0743-7315.

39 BOUKERCHE, A. et al. An exact parallel algorithm to compare very long biological sequences in clusters of workstations. *Cluster Computing*, Kluwer Academic Publishers, Hingham, MA, USA, v. 10, n. 2, p. 187–202, 2007. ISSN 1386-7857.

40 CATALYUREK, U. et al. A component-based implementation of multiple sequence alignment. In: *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2003. p. 122–126. ISBN 1-58113-624-2.

41 ZOLA, J. et al. Parallel multiple sequence alignment with local phylogeny search by simulated annealing. *Parallel and Distributed Processing Symposium, International*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 279, 2006.

42 DU, Z.; LIN, F. pnjtree: a parallel program for reconstruction of neighbor-joining tree and its application in clustalw. *Parallel Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 32, n. 5, p. 441–446, 2006. ISSN 0167-8191.

43 LOPES, H. S.; MORITZ, G. L. A distributed approach for a multiple sequence alignment algorithm using a parallel virtual machine. *Engineering in Medicine and*

Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the, p. 2843–2846, 2005.

44 DU, Z.; JI, Z.; LIN, F. Parallel computing for optimal genomic sequence alignment. In: *FSKD*. [S.l.: s.n.], 2006. p. 532–535.