**Daniela Saccol Peranconi**

# Alinhamento de Seqüências Biológicas em Arquiteturas com Memória Distribuída

São Leopoldo

2005

**Daniela Saccol Peranconi**

# Alinhamento de Seqüências Biológicas em Arquiteturas com Memória Distribuída

Dissertação submetida a avaliação como requisito parcial para a obtenção do grau de Mestre em Computação Aplicada

Orientador:
Gerson Geraldo Homrich Cavalheiro

UNIVERSIDADE DO VALE DO RIO DOS SINOS
CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

São Leopoldo

2005

# AGRADECIMENTOS

Agradeço a Deus pelas oportunidades que me foram concedidas e por ter me dado forças nos momentos mais difíceis durante esses dois últimos anos. Só Ele sabe o quanto foi difícil continuar em certos momentos...

Aos meus pais, só tenho a dizer MUITO OBRIGADA! Não existem palavras capazes de expressar o quanto são importantes para mim nem o quanto me auxiliaram durante este trabalho. Sempre tiveram uma palavra de apoio e motivação quando tudo parecia ir mal. Souberam compreender minha ausência em horas importantes e aceitaram minhas escolhas mesmo nem sempre concordando com elas. Ainda bem que eles existem!!

Ao Professor Gerson Cavalheiro, meu orientador, meus sinceros agradecimentos. Obrigada pela orientação, pelos conselhos, pela disponibilidade e pelos puxões de orelha (apesar de sutis, como é do costume dele, existiram). E esse também é o momento para pedir desculpas por eventuais falhas e por coisas que deixaram a desejar...

Ao meu amor, Du, também devo muitos agradecimentos. Foi no ombro dele que chorei muitas vezes, quando tudo parecia dar errado e quando a saudade de casa apertava... Foi ele quem ouviu minhas lamentações sem nunca reclamar... E só não voltei para casa depois do primeiro mês de Mestrado, pois ele esteve a meu lado sempre. Ah, sem falar na crucial ajuda para colocar a dissertação nas normas ABNT.

Não posso esquecer dos meus colegas do grupo de pesquisa: Evandro Dall'Agnol, Otávio Cordeiro e Lucas Villa Real... meninos, vocês foram extremamente importantes para a conclusão deste trabalho. Obrigada por toda ajuda durante estes dois anos.

Também não posso deixar de fora a Rejane, secretária do Mestrado... agradeço pelo auxílio acadêmico e por emprestar seu tempo quando eu precisava conversar com alguém...

Agradeço, também, à Capes que financiou meu segundo ano de Mestrado e à HP Brasil R&D que me proporcionou cinco meses de estágio em pesquisa.

Aos meus colegas de Mestrado, obrigada pelos ensinamentos que pude absorver de cada um de vocês... cada um a seu jeito, me ensinou alguma coisa... exemplos que devo seguir e outros que jamais devo pensar em fazer igual.

Por fim, agradeço a todos aqueles que de alguma forma contribuíram para a conclusão deste trabalho... se esqueci o nome de alguém, me desculpem...

# RESUMO

A utilização de aglomerados de computadores na solução de problemas que demandam grande quantidade de recursos computacionais vem se mostrando uma alternativa interessante. Aglomerados são economicamente viáveis e de fácil manutenção, oferecendo poder computacional equivalente ao de supercomputadores. No entanto, o desenvolvimento de aplicações para este tipo de arquitetura é complexo, uma vez que envolve questões não presentes na programação seqüencial, como a comunicação de dados e a sincronização de tarefas concorrentes, problemas estes que, em geral, são tratados em supercomputadores por pacotes de software especializados. Neste contexto, este trabalho apresenta o desenvolvimento de um mecanismo de suporte à comunicação sobre aglomerados de computadores, focado na exploração desta plataforma de hardware para o processamento de alto desempenho. O mecanismo criado e disponibilizado sob a forma de uma biblioteca de funções em C, é baseado no modelo de Mensagens Ativas. Sua implementação é realizada na camada aplicativa, empregando técnicas de multiprogramação leve como recurso de programação. Para atestar a necessidade de tal mecanismo de comunicação e aferir o desempenho da biblioteca implementada, uma aplicação apresentando reais necessidades de processamento em aglomerados é apresentada e discutida: o alinhamento de seqüências biológicas. No caso específico, as necessidades do alinhamento de seqüências refletem seu custo computacional e, em particular, seu consumo de memória. Além da aplicação real, o projeto e a implementação do mecanismo de comunicação também consideraram sua introdução no ambiente Anahy para processamento de alto desempenho sobre aglomerados de computadores. A avaliação da biblioteca de comunicação desenvolvida é realizada pela implementação de um protótipo de uma aplicação para alinhamento de seqüências biológicas utilizando métodos de programação dinâmica e de uma aplicação para cálculo do Número de Fibonacci.

Palavras-chave: Alinhamento de seqüências. Processamento de Alto Desempenho. Memória distribuída. Mensagens Ativas. Multiprogramação leve.

# ABSTRACT

The use of cluster of computers for solving problems that require a great quantity of computational resources is becoming an interesting alternative. Clusters are economically feasible and of easy maintenance, offering a computational power equivalent to that of supercomputers. However developing applications for this kind of architecture is complex because it involves issues that are not present in the sequential programming such as data communication and concurrent tasks synchronization, problems that usually are handled by specialized software packages in supercomputers. Considering this context, this work presents the development of a mechanism for supporting communication on clusters of computers focused on exploring this hardware platform for high performance processing. The mechanism was created as a library of functions written in C and it is based on the Active Messages model. Its implementation was performed on the applicative level, using light multiprogramming techniques as programming resource. In order to prove the need of such communication mechanism and to measure the performance of the implemented library, an application with real processing needs in clusters is presented and discussed: the alignment of biological sequences. In this specific case, the needs of the sequences alignment reflect its computational cost and particularly its memory consumption. Besides the real application, the project and implementation of the communication mechanism considered also its introduction in the Anahy environment for high performance processing on clusters of computers. The evaluation of the communication library developed was achieved by implementing a prototype application for alignment of biological sequences using dynamic programming methods and an application for calculating the Fibonacci Number.

Key-words: Sequence alignment. High Performance Processing. Distributed memory. Active Messages. Multithreading.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS

FASTA - *Fast-All*
BLAST - *Basic Local Alignment Search Tool*
PVM - *Parallel Virtual Machine*
MPI - *Message Passing Interface*
SPMD - *Single Program Multiple Data*
BSP - *Bulk-Synchronous Parallel*
RSR - *Remote Service Request*
RPC - *Remote Procedure Call*
SMP - *Symetric Multiprocessor*
DNA - Ácido Desoxirribonucléico
RNA - Ácido Ribonucléico
bp - Pares de bases
PAM -*Point Accepted Mutations* ou *Percent Accepted Mutations*
BLOSUM - *Blocks Amino Acid Substitution Matrices*
MSP - *Maximum Segment Pair*
ATGC - *Another Tool for Genome Comparison*
CGM - *Coarse Grained Multicomputers*
CPU - *Central Processing Unit*
PV - Processador Virtual

# SUMÁRIO

# 1 INTRODUÇÃO

A cada ano, diferentes grupos de pesquisa disponibilizam novos genomas completamente seqüenciados, fazendo com que o número de informações nos bancos de dados públicos cresça, exponencialmente, em um fator entre 1.5 e 2 todo ano (SCHMIDT; SCHRÖDER; SCHIMMLER, 2002). Para manipular eficientemente esta enorme quantidade de dados gerada em laboratório, a Bioinformática surgiu como auxílio aos pesquisadores, unindo profissionais de diferentes áreas. Dentre as tarefas da Bioinformática, o alinhamento de seqüências é considerada a operação básica mais importante, servindo como base para operações mais complexas (SETUBAL; MEIDANIS, 1997).

Needleman e Wunsch (NEEDLEMAN; WUNSCH, 1970), Sankoff (SANKOFF, 1975) e Smith e Waterman (SMITH; WATERMAN, 1981), deram início, na década de setenta, ao estudo do problema do alinhamento de seqüências, baseando-se no método de programação dinâmica. Em estudos mais recentes, técnicas de aprendizado de máquina, como *Simulated Annealing* ((ISHIKAWA et al., 1993), (KIM; COLE, 1993)), Algoritmos Genéticos ((NOTREDAME; HIGGINS, 1996), (ZHANG; WONG, 1997), (ZALIZ, 2001)), dentre outras, estão sendo propostas para casos de alinhamentos mais complexos.

Considerando as diversas técnicas utilizadas para alinhar seqüências ((NEEDLEMAN; WUNSCH, 1970), (SANKOFF, 1975), (SMITH; WATERMAN, 1981), (ISHIKAWA et al., 1993), (KIM; COLE, 1993), (NOTREDAME; HIGGINS, 1996), (ZHANG; WONG, 1997), (ZALIZ, 2001)), é provado matematicamente que um alinhamento ótimo é obtido como resultado quando se utiliza o método de programação dinâmica (GUSFIELD, 1997). Esta precisão facilita o trabalho dos biólogos, pois quanto mais precisos forem os resultados,

mais significativas serão as informações fornecidas, podendo-se, por exemplo, prever informações estruturais, funcionais e evolucionárias das seqüências (GUSFIELD, 1997). No entanto, esta é uma operação que requer muito cálculo (YAP; FRIEDER; MARTINO, 1998), envolvendo muito tempo de execução e requerendo computadores com grande quantidade de memória disponível.

Duas alternativas surgem neste caso: ou se reduz a sensitividade do alinhamento ou se desenvolve hardware especializado. A primeira solução faz uso de métodos heurísticos como FASTA (PEARSON; LIPMAN, 1988) e BLAST (ALTSCHUL et al., 1990) para realizar o alinhamento. Tais algoritmos são cerca de 40 vezes mais rápidos que as melhores implementações de Smith-Waterman desenvolvidas até o momento, porém, por serem menos sensitivos, podem perder informações biologicamente relevantes (LAU, 2000). Por outro lado, a segunda solução trata do desenvolvimento de hardware com propósito especial, como Paracel's GeneMatcher e Compugen's Bioaccelerator. Tais máquinas são capazes de processar milhões de comparações por segundo, podendo ser expandidas e atingir velocidades muito superiores (YANG, 2002).

A segunda solução aparece como sendo a mais indicada quando se buscam resultados ótimos para o alinhamento, sem precisar preocupar-se com questões de tempo e memória. Porém, é praticamente inviável para a maioria dos usuários adquirir máquinas como as citadas anteriormente, uma vez que este tipo de hardware requer altos investimentos financeiros. Como alternativa de solução de hardware, surgiram os aglomerados de computadores[1], que podem ser definidos como sendo um sistema de computação paralela composto por um conjunto de computadores independentes trabalhando de forma integrada como se fosse um recurso computacional único (BUYYA, 1999). Os aglomerados são mecanismos eficientes na obtenção de alto desempenho e disponibilidade de memória e, por serem construídos com peças comuns, encontradas com facilidade no mercado, a relação custo-benefício de um aglomerado é mais baixa que a encontrada em supercomputadores (PASIN; KREUTZ, 2003).

[1] *Clusters of workstations.*

Apesar de mostrarem-se como excelente alternativa, a utilização de aglomerados de computadores esbarra em um problema ainda sem solução eficiente: além de serem poucas, as interfaces de programação para este tipo de arquitetura oferecem pouco conforto ao programador. Para que possa obter um programa que tire proveito dos recursos oferecidos por aglomerados, o programador necessita, primeiramente, decompor sua aplicação em diversas atividades concorrentes e estabelecer os critérios de sincronização entre estas atividades, para, em seguida, poder mapeá-las sobre os recursos disponíveis (CAVALHEIRO, 2004). Em alguns casos, o programador necessita introduzir alguma estratégia de distribuição da carga gerada pelo programa em execução entre os recursos de processamento; em outros, o próprio ambiente possui recursos de escalonamento capazes de distribuir a carga computacional gerada por um programa. Anahy (CAVALHEIRO; DALL'AGNOL; VILLA REAL, 2003), Athapascan-1 (GALLILÉE et al., 1998), Cilk (BLUMOFE et al., 1995) e Jade (RINARD; LAM, 1998) são exemplos de ambientes para o processamento paralelo, com esta última característica, que podem ser encontrados na literatura.

## 1.1 MOTIVAÇÃO

A crescente disponibilidade de dados biológicos para serem tratados e a necessidade cada vez maior de métodos eficientes para preparar e analisar tais dados, torna necessário o desenvolvimento de soluções eficientes, capazes de fornecer resultados biologicamente relevantes e em tempo hábil. Considerando-se a falta de recursos financeiros em muitas instituições, a utilização de aglomerados de computadores mostra-se como excelente solução.

Apesar de os aglomerados apresentarem-se como fortes candidatos à resolução do problema, ainda é trabalhoso desenvolver aplicações para alto desempenho capazes de tirar proveito das vantagens oferecidas por esta classe de arquitetura. As ferramentas existentes exigem intervenção direta do programador no momento de preparar sua aplicação

de forma a utilizar melhor os recursos de hardware disponíveis. Em particular, em se tratando de uma arquitetura com memória distribuída, uma das questões de maior impacto no desenvolvimento de programas são os mecanismos de comunicação empregados. Diversas ferramentas de comunicação encontram-se disponíveis para arquiteturas do tipo aglomerados, embora poucas tenham sido desenvolvidas tendo como finalidade o processamento de alto desempenho.

## 1.2 CONTEXTO DO TRABALHO

O presente trabalho foi desenvolvido no contexto do projeto Anahy. Anahy é uma ferramenta para exploração de processamento de alto desempenho em aglomerados de computadores. Ela se caracteriza por explorar um diagrama de fluxo de dados descrevendo a aplicação em termos de dependências de dados entre tarefas.

No contexto do projeto, o presente trabalho visa a implementação de uma aplicação real sobre aglomerados de computadores, em seus termos gerais. Para tanto, foi desenvolvida uma biblioteca de comunicação sobre a qual a referida aplicação foi implementada observando os modelos de programação e execução de Anahy. O trabalho é completado com uma análise do uso desta biblioteca como componente do suporte executivo de Anahy. Trabalhos futuros deverão levar em consideração questões aprofundadas de balanceamento de carga e demais especificidades dos algoritmos de alinhamento e seus usos, bem como a introdução definitiva de suporte à comunicação em Anahy.

## 1.3 OBJETIVOS

Neste trabalho é proposta uma ferramenta de comunicação sobre aglomerados que seja de fácil utilização e que, ao mesmo tempo, proporcione uma diminuição de perdas de desempenho ocasionadas por latências nas comunicações. Para atender ao primeiro requisito, busca-se um conjunto reduzido de funcionalidades, capazes de atender a todas as necessidades de programação em aglomerados. O segundo objetivo é alcançado com o

emprego de duas importantes estratégias: a de multiprogramação leve[2] (VALIANT, 1990) e a de Mensagens Ativas (EICKEN et al., 1992). Para validar a ferramenta proposta, serão desenvolvidas duas aplicações, uma para o cálculo do Número de Fibonacci e outra para o alinhamento de seqüências biológicas.

### 1.3.1 Objetivo Geral

Desenvolvimento de um mecanismo de suporte à execução de aplicações em arquiteturas com memória distribuída, utilizando Mensagens Ativas e multiprogramação leve (PERANCONI; CAVALHEIRO, 2004, 2005).

### 1.3.2 Objetivos Específicos

Atingir o objetivo geral deste trabalho implica em atender uma série de objetivos específicos associados tanto a modelagem e construção do mecanismo de comunicação como a sua avaliação através de seu emprego em uma aplicação real.

Estes objetivos específicos são listados a seguir, identificando onde encontram-se abordados no texto e publicações onde estes aspectos são contemplados.

- Modelagem e implementação do mecanismo de comunicação entre os nós de um aglomerado de computadores, utilizando Mensagens Ativas (Capítulo 6);
- Integração do mecanismo com o ambiente Anahy (Seções 5.5, 6.3, 6.4 e 7.3, (DALL'AGNOL et al., 2004), (DALL'AGNOL et al., 2005));
- Implementação de uma aplicação sintética para avaliação (cálculo do Número de Fibonacci) (Seção 7.1);
- Implementação de uma aplicação para alinhamento de seqüências biológicas (Seção 7.2, (LERMEN; PERANCONI; CAVALHEIRO, 2004b, 2004c, 2004a));

[2] *Multithreading*

- Validação do mecanismo de comunicação através das aplicações implementadas (Seções 7.1.3, 7.2.5, 7.2.6 e 7.4, (CORDEIRO et al., 2005)).

## 1.4 ORGANIZAÇÃO DO TEXTO

Os demais capítulos da presente dissertação de Mestrado encontram-se organizados como segue.

O Capítulo 2 apresenta alguns conceitos básicos de Biologia Molecular, concentrando-se no problema do alinhamento de seqüências. Relacionado a este tema, são destacados dois tipos de alinhamento (global e local) e as principais técnicas utilizadas para alinhar seqüências. Dentre estas técnicas, destaca-se a de programação dinâmica, empregada pelo algoritmo de alinhamento local utilizado como estudo de caso neste trabalho. São destacados, ainda, alguns trabalhos referenciados na literatura que introduzem paralelização ao método de programação dinâmica, de maneira a diminuir o tempo de processamento e a possibilitar o alinhamento de seqüências maiores.

No Capítulo 3 são destacados alguns conceitos relacionados à programação concorrente. Abordam-se os níveis de concorrência que podem ser explorados em uma arquitetura de suporte para execução de aplicações e as principais ferramentas que podem ser empregadas para explorar tais níveis.

O Capítulo 4 apresenta o mecanismo de Mensagens Ativas como ferramenta para troca de mensagens entre os nós de um aglomerado, com sua conceituação e diferentes implementações. Descreve-se o funcionamento da implementação original e das implementações alternativas, com destaque à implementação desenvolvida dentro do contexto do projeto Anahy. São, ainda, apresentados alguns ambientes que implementam as Mensagens Ativas.

No Capítulo 5 é apresentado o ambiente de programação paralela Anahy. Abordam-se: (*i*) as primitivas de programação disponíveis ao usuário, (*ii*) o núcleo exe-

cutivo do ambiente e (*iii*) os serviços de escalonamento. Destaca-se, por fim, a principal característica de Anahy: realizar toda a sincronização de tarefas para o programador segundo um modelo de fluxo de dados.

O Capítulo 6 apresenta o mecanismo de suporte à execução de aplicações em aglomerados desenvolvido neste trabalho. Destacam-se as principais características deste mecanismo e seu enquadramento no projeto Anahy. São apresentadas e descritas as primitivas de programação disponíveis ao usuário do mecanismo. Por fim, são descritos os serviços de comunicação a serem implementados para introdução do mecanismo no ambiente Anahy, o que tornará tal ambiente operacional para aglomerados de computadores.

O Capítulo 7 destina-se à análise de desempenho do mecanismo desenvolvido. Neste capítulo são descritas as duas aplicações implementadas para a realização dos experimentos: cálculo do Número de Fibonacci e alinhamento de seqüências. Para ambas aplicações são apresentados o grafo de dependências entre as tarefas, o algoritmo implementado e os resultados de desempenho obtidos.

No último capítulo desta dissertação, Capítulo 8, são apresentadas as conclusões obtidas com o desenvolvimento do trabalho e descritos os trabalhos futuros que podem ser desenvolvidos como extensão da presente dissertação.

# 2 ALINHAMENTO DE SEQÜÊNCIAS BIOLÓGICAS

A cada ano, disponibilizam-se novas seqüências biológicas (DNA ou proteínas) para serem analisadas. Pouca utilidade teriam, não fosse a possibilidade de se extrair informações a respeito de suas funcionalidades, estruturas e evolução. Uma das formas de se extrair tais informações, consiste no alinhamento de seqüências. No entanto, alinhar seqüências trata-se de um problema que demanda alto poder computacional para ser solucionado. Dada a importância deste assunto, o objetivo deste capítulo é apresentar tópicos relacionados a ele.

Na Seção 2.1 são abordados alguns conceitos básicos de Biologia Molecular, necessários ao entendimento do restante do trabalho. A Seção 2.2 trata mais especificamente do problema de alinhamento de seqüências, destacando os tipos de alinhamento e os esquemas de pontuação utilizados no processo do alinhamento. Na Seção 2.3 são apresentadas as técnicas mais utilizadas para o alinhamento de seqüências, com ênfase nos algoritmos que empregam a técnica de programação dinâmica. A Seção 2.4 destaca alguns trabalhos que utilizam o método de programação dinâmica para realizar o alinhamento de seqüências biológicas com o diferencial de paralelizarem tal método para acelerar o processo do alinhamento. Por fim, na Seção 2.5 são feitos alguns comentários sintetizando os principais pontos do capítulo.

## 2.1 CONCEITOS BÁSICOS DE BIOLOGIA MOLECULAR

Nesta seção são apresentados alguns conceitos básicos de Biologia Molecular, necessários ao melhor entendimento deste trabalho. As definições aqui apresentadas estão baseadas, principalmente, nos livros Introduction to Computational Molecular Biology (SETUBAL; MEIDANIS, 1997) e Biologia Molecular Básica (ZAHA; FERREIRA; PASSAGLIA, 2003).

### 2.1.1 Vida

A natureza é formada de objetos vivos e inanimados. Enquanto os primeiros podem mover-se, reproduzir-se, crescer e comer, participando ativamente do ambiente em que vivem, os outros não agem dessa forma. Se pesquisas mostraram que ambos são compostos pelos mesmos átomos e pelas mesmas regras físicas e químicas, o que, na realidade, os difere? A resposta para esta pergunta está no complexo arranjo de reações químicas que ocorrem dentro dos seres vivos. O que dá vida a um organismo é justamente o fato dessas reações nunca cessarem, pois o produto de uma reação é consumido por outra.

As primeiras formas de vida, que surgiram há cerca de 3,5 bilhões de anos, eram muito simples, mas, com o passar dos anos, estas formas evoluíram para organismos mais complexos, chegando até os que se conhece atualmente. Porém, tanto as formas simples de bilhões de anos quanto as complexas de hoje, possuem uma bioquímica semelhante. Na química da vida, os principais atores são moléculas chamadas *proteínas* e *ácidos nucléicos*.

### 2.1.2 Proteínas

As proteínas são um dos constituintes básicos das células e muitas das substâncias do nosso organismo são proteínas. Elas desempenham inúmeras funções biológicas, como, por exemplo, catalisar (acelerar) reações químicas e transportar oxigênio, e são conhecidas como as moléculas que realizam o trabalho celular, sendo constituídas de moléculas mais

simples denominadas *aminoácidos*.

### 2.1.3 Ácidos Nucléicos

Os ácidos nucléicos são macromoléculas de extrema importância biológica em todos os organismos vivos. Além de transmitirem às células instruções sobre quais proteínas sintetizar e em que quantidade, são responsáveis por estocar e transmitir a informação genética na célula. Existem dois tipos de ácidos nucléicos: *ácido desoxirribonucléico* (DNA) e *ácido ribonucléico* (RNA).

#### 2.1.3.1 DNA

Assim como uma proteína, uma molécula de DNA é uma cadeia (fita) formada por moléculas mais simples. No caso do DNA, porém, esta cadeia é dupla e cada cadeia possui uma espinha dorsal que consiste de repetições da mesma unidade básica. Esta unidade é formada por uma molécula de açúcar, chamada 2´-desoxirribose, ligada a um resíduo fosfato. A molécula de açúcar contém cinco átomos de carbono numerados de 1´a 5´, como apresentado na Figura 1. A ligação que cria a espinha dorsal se dá entre o carbono 3´ de uma unidade, o resíduo fosfato e o carbono 5´ da próxima unidade. Devido a isso, as moléculas de DNA possuem uma orientação que, por convenção, inicia no 5´ e termina no 3´ (5´ $\longrightarrow$ 3´). Mesmo estando ligadas, cada fita preserva sua própria orientação e as duas orientações são opostas (5´ $\longrightarrow$ 3´ e 3´ $\longrightarrow$ 5´), sendo fitas antiparalelas.

Na Figura 1 pode-se observar que, a cada carbono 1´ está ligada uma molécula chamada *base*. Para o caso do DNA, existem quatro tipos de bases: adenina (A), guanina (G), citosina (C) e timina (T). Uma unidade básica de uma molécula de DNA consistirá, então, de um açúcar, um resíduo fosfato e uma base, formando um *nucleotídeo*. Um conjunto de nucleotídeos forma, assim, uma molécula de DNA.

Em 1953 James Watson e Francis Crick descobriram que, na verdade, as duas fitas das moléculas de DNA estão ligadas de maneira que formam uma dupla hélice. O que

mantém a estrutura da molécula é o pareamento das bases. Cada base de uma fita está pareada com uma base da outra fita. A base A está sempre pareada com a base T e C está sempre pareada com G, como mostrado na Figura 2. As bases A e T, assim como C e G são ditas *bases complementares*. Estes pares de bases (bp) são a unidade de medida mais utilizada quando se trata de moléculas de DNA. Assim, diz-se que um certo pedaço de DNA tem tamanho, por exemplo, de 10.000 bp.

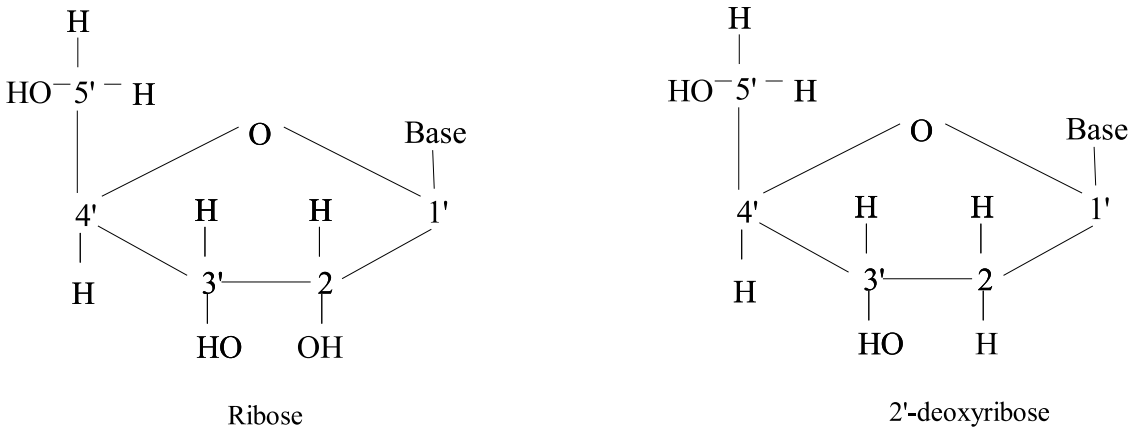


Figura 1: Açúcares presentes nos ácidos nucléicos (SETUBAL; MEIDANIS, 1997).

#### 2.1.3.2 RNA

O RNA é uma molécula formada, geralmente, por uma só fita. A seqüência de bases é semelhante a do DNA, diferindo pela substituição da timina pela uracila, que, neste caso, será pareada com adenina. A outra diferença composicional entre DNA e RNA é a substituição da desoxirribose, presente no DNA, pela ribose, encontrada no RNA, como mostrado na Figura 1.

Além disso, enquanto o DNA realiza, essencialmente, a função de decodificar informação, o RNA é responsável por diferentes funções, tais como conter a informação genética para a seqüência de aminoácidos, identificar e transportar as moléculas de aminoácidos até o ribossomo

### 2.1.4 Genes e Código Genético

O DNA é responsável por codificar a informação necessária para construir cada proteína ou molécula de RNA em um organismo. Porém, somente alguns trechos contíguos em uma molécula de DNA é que codificam informação para construir uma proteína ou molécula de RNA. Além disso, a cada tipo diferente de proteína corresponde um e somente um desses trechos contíguos, chamados *genes*. O tamanho de cada gene varia dependendo do tipo de organismo e, no caso do Homem, um gene tem cerca de 10.000 bp. Como cada gene é responsável por codificar uma proteína específica, as células desenvolveram mecanismos capazes de identificar os pontos precisos que marcam o início e o fim de um gene.

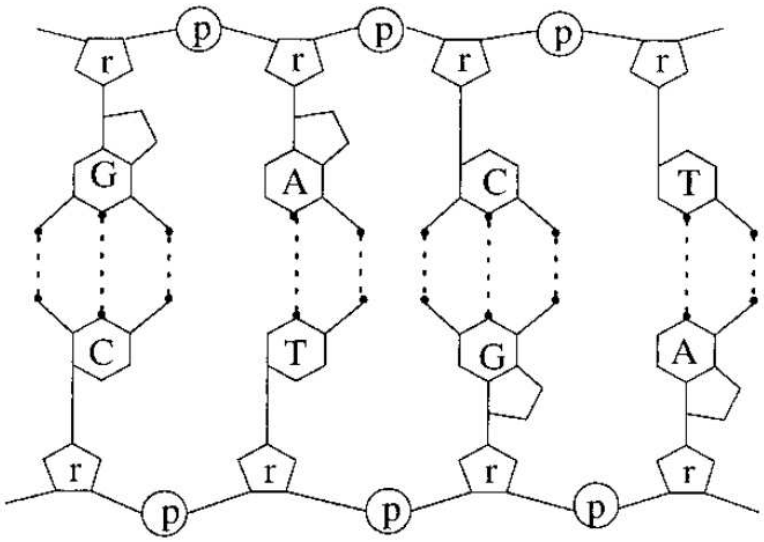


Figura 2: Visualização esquemática da estrutura de dupla fita do DNA (SETUBAL; MEIDANIS, 1997).

Para que seja possível especificar uma proteína, é preciso determinar cada aminoácido que ela contém. O gene faz exatamente isso, utilizando, para tanto, trios de nucleotídeos chamados *códons*. A correspondência entre cada códon e cada aminoácido é denominada *código genético*, apresentado na Tab. 1. Pode-se notar que os trios de nucleotídeos são dados com bases de RNA no lugar das bases de DNA. A explicação para isso é o fato das moléculas de RNA serem o vínculo entre o DNA e as proteínas.

Tabela 1: O código genético mapeando códons para aminoácidos.

| Primeira Posição | Segunda Posição | | | | Terceira Posição |
|---|---|---|---|---|---|
| | G | A | C | U | |
| G | Gly | Glu | Ala | Val | G |
| | Gly | Glu | Ala | Val | A |
| | Gly | Asp | Ala | Val | C |
| | Gly | Asp | Ala | Val | U |
| A | Arg | Lys | Thr | Met | G |
| | Arg | Lys | Thr | Ile | A |
| | Ser | Asn | Thr | Ile | C |
| | Ser | Asn | Thr | Ile | U |
| C | Arg | Gln | Pro | Leu | G |
| | Arg | Gln | Pro | Leu | A |
| | Arg | His | Pro | Leu | C |
| | Arg | His | Pro | Leu | U |
| U | Trp | STOP | Ser | Leu | G |
| | STOP | STOP | Ser | Leu | A |
| | Cys | Tyr | Ser | Phe | C |
| | Cys | Tyr | Ser | Phe | U |

Outro ponto a ser observado na Tab. 1 é a existência de 64 códons, mas somente 20 aminoácidos a serem especificados. A conseqüência disso é a existência de diferentes trios de nucleotídeos correspondendo ao mesmo aminoácido. Por outro lado, existem códons que não especificam nenhum aminoácido, mas somente delimitam o ponto final do gene. Essa terminação especial é identificada na Tab. 1 pela palavra STOP. Por fim, o código genético apresentado na Tab. 1 é utilizado pela maioria dos organismos vivos, com algumas exceções de organismos que utilizam um código ligeiramente modificado.

As células de organismos vivos possuem algumas moléculas de DNA muito longas. Cada uma dessas moléculas é chamada de *cromossomo*. O conjunto completo de cromossomos dentro de uma célula forma o que denominamos *genoma*. Os genomas contêm informação referente ao organismo que ele codifica e deseja-se saber que informação está contida neles. Uma das formas de se interpretar tal informação é a realização de uma operação denominada *comparação de seqüências*. O restante deste capítulo destina-se a detalhar melhor tal operação.

## 2.2 COMPARAÇÃO DE SEQÜÊNCIAS

A ocorrência de mutação em seqüências de DNA é um processo evolucionário natural: erros na replicação do DNA causam substituições, inserções e deleções de nucleotídeos, o que leva a um DNA modificado. A similaridade entre seqüências de DNA pode ser um indício de uma origem evolucionária comum (como a similaridade entre alguns genes nos homens e chimpanzés) ou o indício de uma funcionalidade comum (PEVZNER, 2001).

Setubal e Meidanis (SETUBAL; MEIDANIS, 1997), consideram a comparação de seqüências a operação primitiva mais importante na Biologia Computacional, servindo como base para operações mais complexas, tais como a busca de similaridades entre biosseqüências (SELLERS, 1980), o emparelhamento de strings (HALL; DOWLING, 1980), a comparação de arquivos (HUNT; SZYMANSKY, 1977) e a pesquisa em textos com erros (WU; MANBER, 1992).

Lecompte e colaboradores (LECOMPTE et al., 2001) destacam algumas informações que podem ser obtidas a partir da comparação de seqüências:

- Predição da função biológica mediante homologia de seqüências;
- Revelação de padrões conservados ao longo da evolução, o que permite uma detecção direta de estruturas conservadas, sinais de localização ou resíduos funcionais que caracterizam uma família ou sub-família de proteínas;
- Estudos para definir os relacionamentos filogenéticos entre organismos;
- Determinação da organização do domínio de uma família de proteínas.

Uma das formas de identificar similaridades entre seqüências é alinhando-as, com a inserção de espaços em ambas seqüências, de forma que fiquem com o mesmo tamanho. Existem, basicamente, dois tipos de alinhamento: *alinhamento global* e *alinhamento local*, os quais diferenciam-se pelas características das seqüências a serem alinhadas.

### 2.2.1 Alinhamento de Seqüências

Mount (MOUNT, 2001) considera o alinhamento de seqüências uma operação útil para a descoberta de informações funcionais, estruturais e evolucionárias nas seqüências biológicas, sendo, por isso, importante encontrar o melhor alinhamento possível ou alinhamento ótimo entre seqüências.

Segundo Setubal e Meidanis (SETUBAL; MEIDANIS, 1997) o alinhamento de seqüências é definido como a inserção de espaços em posições arbitrárias das seqüências, de forma que ambas fiquem com o mesmo tamanho. A partir daí, as seqüências podem ser colocadas uma sobre a outra, criando uma correspondência entre caracteres ou espaços na primeira seqüência com caracteres ou espaços na segunda seqüência, sem que nenhum espaço em uma seqüência seja alinhado com um espaço na outra.

A formalização da definição anterior é dada pela Definição 1 (TICONA, 2003), a seguir.

**Definição 1** *Dado um alfabeto finito A, sejam s e t duas cadeias sobre A de comprimento m e n, respectivamente. Seja* $A^* = A \cup \{ \text{ - } \}$ *um alfabeto finito onde '-' representa o caractere de buraco (gap). A tupla* $(s^*, t^*)$ *representa um alinhamento entre s e t se as seguintes propriedades são satisfeitas:*

- *As cadeias* $s^*$ *e* $t^*$ *têm o mesmo comprimento,* $| s^* | = | t^* |$*;*
- *Não existem dois buracos na mesma posição em* $s^*$ *e* $t^*$*;*
- *Eliminando os buracos,* $s^*$ *é reduzido para s;*
- *Eliminando os buracos,* $t^*$ *é reduzido para t.*

Quando se trata de seqüências de DNA, o alfabeto é $A^* = \{ A, T, C, G, \text{-} \}$, ou seja, as quatro bases e o caractere de espaço. Já, quando se trata de proteínas, o alfabeto $A^*$ possui vinte e um elementos (os vinte aminoácidos encontrados na natureza e o caractere de espaço).

### 2.2.2 Tipos de Alinhamento

Como mencionado anteriormente, existem, basicamente, dois tipos de alinhamento: *alinhamento global*, que leva em consideração seqüências inteiras e *alinhamento local*, que considera, apenas, subseqüências.

#### 2.2.2.1 Alinhamento Global

O alinhamento global se destina a encontrar o melhor alinhamento entre duas seqüências inteiras, utilizando tantos caracteres quanto possível. As melhores seqüências a serem alinhadas neste caso são aquelas com alta similaridade e com tamanho aproximado. O emparelhamento de dois caracteres distintos indica a ocorrência de uma mutação em uma das seqüências. O emparelhamento entre um caractere de uma seqüência e um espaço da outra, indica a inserção ou remoção de caracteres em uma das seqüências.

Como exemplo de um alinhamento global, considere as seqüências hipotéticas X = GAAGGATTAG e Y = GATCGAAG. O alinhamento global resultante destas seqüências é apresentado na Figura 3.

```
G A A - G G A T T A G
G A T C G G A - - A G
```

Figura 3: Exemplo de alinhamento global entre as seqüências X = GAAGGATTAG e Y = GATCGGAAG.

#### 2.2.2.2 Alinhamento Local

Um alinhamento local entre duas seqüências $s$ e $t$ ocorre entre uma substring de $s$ e uma substring de $t$. Em geral, quando um alinhamento local é realizado, busca-se não somente alinhamentos ótimos, mas também alinhamentos com pontuação superior a um valor pré-definido.

Dadas, por exemplo, as seqüências hipotéticas X = AAGACGG e Y = GATC-

GAAG, um possível alinhamento local entre elas é mostrado na Figura 4.

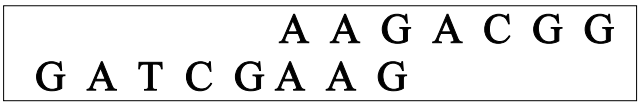


Figura 4: Exemplo de alinhamento local entre as seqüências X = AAGACGG e Y = GATCGAAG.

Este tipo de alinhamento é utilizado para comparar grandes seqüências, com tamanhos diferentes ou que compartilham uma região ou domínio conservados. É mais sensível quando se deseja comparar duas seqüências pouco relacionadas e favorece que se encontrem padrões de nucleotídeos conservados, seqüências de DNA ou padrões de aminoácidos em seqüências de proteínas.

### 2.2.3 Esquemas de Pontuação

Na seção anterior foram citados os dois principais tipos de alinhamento de seqüências. A cada alinhamento, independente do tipo, é associada uma pontuação para que seja possível detectar, por exemplo, o alinhamento ótimo entre duas seqüências. A pontuação resultante para um alinhamento é produto do esquema de pontuação adotado durante o processo do alinhamento e exerce grande influência no resultado do mesmo, uma vez que diferentes sistemas de pontuação levam a diferentes alinhamentos (ROCHA, 2004).

A similaridade entre duas seqüências é definida como a maior pontuação de um alinhamento. Um esquema de pontuação é dado pelo par ($p$, $g$), onde a função $p$ é utilizada para pontuar cada par de caracteres alinhados e $g$ é utilizado para penalizar espaços inseridos, sendo, usualmente, menor que zero (g $<$ 0). O esquema de pontuação fornece um valor numérico para cada alinhamento possível.

Dado um par de seqüências $s$ e $t$ e o alinhamento ($s^*$, $t^*$), se adiciona *p(a, b)* cada vez que um caractere $a$ de $s^*$ é alinhado com um caractere $b$ de $t^*$ e, cada vez que um caractere $a$ de $s^*$ ou $b$ de $t^*$ é alinhado com um caractere de espaço, é adicionado $g$ à

pontuação. A soma de todas as pontuações em todas as posições do alinhamento ($s^*$, $t^*$) é denotada por *score(*$s^*$*,* $t^*$*)*. Por fim, a similaridade entre duas seqüências $s$ e $t$ é dada por:

$$sim(s,t) = max_{(s^*,t^*)\in\beta} score(s^*,t^*)$$

onde $\beta$ é o conjunto de todos os alinhamentos entre $s$ e $t$.

Um esquema de pontuação muito utilizado para seqüências de DNA consiste em:

- Somar 1 na pontuação quando ocorre um emparelhamento (*match*) entre caracteres de $s$ e $t$, ou seja, quando os caracteres de $s$ e $t$, na mesma posição, são iguais;
- Somar -1 na pontuação quando os caracteres de $s$ e $t$, na mesma posição, são diferentes. Isto também é chamado de *mismatch*;
- Somar -2 na pontuação quando um caractere de $s$ ou $t$ é alinhado com um caractere de espaço.

Tomando como exemplo o alinhamento da Figura 3 e assumindo os valores anteriormente mencionados, a pontuação resultante para este caso é apresentado na Figura 5.

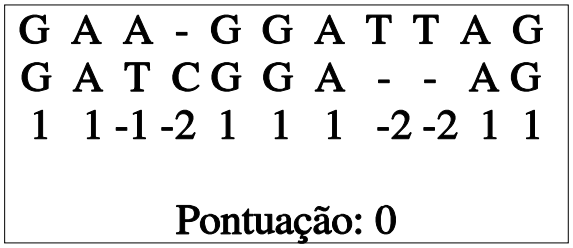


Figura 5: Exemplo do cálculo da pontuação entre duas seqüências utilizando 1 para um *match*, -1 para um *mismatch* e -2 para o alinhamento de um caractere com um espaço.

Porém, o esquema de pontuação simples apresentado anteriormente é inadequado para alguns casos. Um exemplo é o alinhamento de proteínas, onde os aminoácidos que compõem as proteínas possuem propriedades bioquímicas, as quais determinam como eles serão substituídos durante o processo evolutivo. Assim, uma vez que a comparação

de proteínas é realizada levando em conta critérios evolutivos, é importante utilizar um esquema de pontuação que reflita estas probabilidades da forma mais realista possível.

Um esquema de pontuação bastante utilizado para atender a este propósito envolve as chamadas *matrizes de substituição de aminoácidos*, sendo que as duas famílias de matrizes de substituição mais utilizadas são PAM (DAYHOFF; SCHWARTZ; ORCUTT, 1979) e BLOSUM (HENIKOFF; HENIKOFF, 1991).

## 2.3 ALGORITMOS DE ALINHAMENTO DE SEQÜÊNCIAS

A importância do alinhamento de seqüências está, principalmente, na utilidade de se descobrir informações funcionais, estruturais e evolucionárias presentes nas seqüências biológicas sendo analisadas. Diante disto, torna-se interessante apresentar métodos capazes de realizar o alinhamento de seqüências, sempre priorizando técnicas capazes de fornecer resultados com a maior relevância biológica possível. Salienta-se que, técnicas que produzem resultados ótimos, em geral exigem maiores recursos computacionais e tempo de processamento, para serem concluídas. Esta seção objetiva, portanto, apresentar algumas técnicas utilizadas para o alinhamento de seqüências, sejam elas ótimas ou não.

### 2.3.1 Algoritmos que Utilizam Programação Dinâmica

A técnica de programação dinâmica consiste, basicamente, em resolver uma instância de um problema a partir de instâncias menores, do mesmo problema, calculadas anteriormente (SETUBAL; MEIDANIS, 1997). Apesar de ser uma técnica que necessita de grande quantidade de recursos computacionais, tanto em consumo de processamento quanto de memória, a grande vantagem da utilização deste método por algoritmos para alinhamento de seqüências está na possibilidade de obtenção de alinhamentos ótimos entre duas seqüências (GUSFIELD, 1997). É de conhecimento que resultados ótimos fornecem

dados úteis para biólogos a respeito das seqüências que estão sendo analisadas, podendo-se prever informações estruturais, funcionais e evolucionárias a respeito de tais seqüências (GUSFIELD, 1997).

#### 2.3.1.1 Alinhamento Global

Um dos algoritmos para alinhamento de seqüências que emprega o método de programação dinâmica é o de Needleman-Wunsch (NEEDLEMAN; WUNSCH, 1970), o qual encontra o alinhamento global ótimo entre duas seqüências sendo analisadas.

Dadas duas seqüências $s$ e $t$, de comprimento $m$ e $n$, respectivamente, constrói-se uma matriz $A$ de tamanho $(m + 1)$ x $(n + 1)$, onde cada entrada $(i, j)$ desta matriz contém a similaridade entre $s[1..i]$ e $t[1..j]$. As seqüências $s$ e $t$ são colocadas na margem esquerda e no topo da matriz, respectivamente. O primeiro valor a ser preenchido é $A[0, 0] = 0$, pois a pontuação obtida ao alinhar duas seqüências vazias é zero. Seguindo esta mesma idéia, preenchem-se a primeira linha e a primeira coluna, as quais são inicializadas com múltiplos da penalidade por espaço $(g)$. Assim, os valores calculados para a primeira linha são $A[0, j] = gj$, para $1 \leq j \leq n$ e para a primeira coluna são $A[i, 0] = gi$, para $1 \leq i \leq m$. Isso significa que uma das seqüências ($s$ ou $t$) é vazia, o que faz com que a pontuação do alinhamento de uma das seqüências com a seqüência vazia seja $-2k$, onde $k$ é o tamanho da seqüência não vazia.

O preenchimento do restante da matriz ocorre de cima para baixo e da esquerda para a direita. Salienta-se que, para obter-se o valor $(i, j)$ da matriz, necessita-se, apenas de três valores previamente calculados: $(i - 1, j)$, $(i - 1, j - 1)$ e $(i, j - 1)$. Existem, então, três possibilidades para obtenção do alinhamento entre $s$ e $t$, sendo elas:

- Alinhar $s[1..i]$ com $t[1..j - 1]$ e um espaço com $t[j]$;
- Alinhar $s[1..i - 1]$ com $t[1..j - 1]$ e $s[i]$ com $t[j]$;
- Alinhar $s[1..i - 1]$ com $t[1..j]$ e $s[i]$ com um espaço.

A cada uma destas possibilidades está associada uma determinada pontuação e, fazendo $p(i, j)$ ser o valor associado a um *match* ou *mismatch* e $g$ o valor atribuído a um *gap*, estas possibilidades podem ser representadas pela Equação 2.1.

$$A[i,j] = max \left\{ \begin{array}{c} A[i,j-1] + g \\ A[i-1,j-1] + p(i,j) \\ A[i-1,j] + g \end{array} \right\} \tag{2.1}$$

Cada elemento calculado na matriz, mantém um apontador para o elemento do qual foi derivado ([$i$, $j$ - 1], [$i$ - 1, $j$ - 1] ou [$i$ - 1, $j$]). Assim que toda a matriz estiver preenchida, dá-se por encerrada a primeira parte do algoritmo. A segunda parte consiste em encontrar o alinhamento propriamente dito entre as duas seqüências. Partindo-se do elemento ($i$, $j$) da matriz, percorrem-se os apontadores até o elemento (0, 0) ser encontrado. Nesta operação de *traceback*, uma das três alternativas seguintes é possível:

- Se o apontador indicar que o elemento ($i$, $j$) derivou do elemento da esquerda ([$i$, $j$ - 1]), então um *gap* é inserido na seqüência $s$;
- Se o apontador indicar que o elemento ($i$, $j$) derivou do elemento superior ([$i$ - 1, $j$]), então um *gap* é inserido na seqüência $t$;
- Se o apontador indicar que o elemento ($i$, $j$) derivou do elemento da diagonal superior esquerda ([$i$ - 1, $j$ - 1]), então nenhum *gap* é inserido, indicando que os elementos de $s$ e $t$ que estão sendo analisados, são iguais.

Como mencionado, o algoritmo para alinhamento global de duas seqüências consiste em dois passos básicos: (1) cálculo da matriz de similaridades e (2) operação de *traceback*. O primeiro passo consiste da inicialização da primeira linha e da primeira coluna, consumindo tempos de *O(m)* e *O(n)*, respectivamente, e do preenchimento do restante da matriz, com tempo de *O(mn)*. O segundo passo consome o tempo de *O(m + n)*, sendo claramente inferior ao tempo de computação do primeiro passo.

#### 2.3.1.2 Alinhamento Local

Em seções anteriores, definiu-se um alinhamento local entre duas seqüências $s$ e $t$ como sendo um alinhamento entre uma substring de $s$ e uma substring de $t$. O algoritmo utilizado para encontrar alinhamentos locais entre duas seqüências é conhecido como algoritmo de Smith-Waterman (SMITH; WATERMAN, 1981), sendo uma variação do algoritmo para alinhamento global descrito anteriormente.

Como antes, a estrutura de dados principal continua sendo uma matriz de $(m + 1)$ x $(n + 1)$ elementos. No entanto, agora cada elemento $(i, j)$ contém a pontuação mais alta de um alinhamento entre um sufixo de $s[1..i]$ e um sufixo de $t[1..j]$. Outra diferença é que a primeira linha e a primeira coluna são inicializadas com zeros, não somente o elemento (0,0) da matriz.

Após a fase de inicialização, a matriz é preenchida como antes, com o elemento $(i, j)$ dependendo de três valores previamente calculados. A recorrência resultante é a Equação 2.2, utilizada para o preenchimento da matriz de similaridades no caso de um alinhamento local.

$$A[i,j] = max \left\{ \begin{array}{c} A[i,j-1] + g \\ A[i-1,j-1] + p(i,j) \\ A[i-1,j] + g \\ 0 \end{array} \right\} \tag{2.2}$$

Esta equação é a mesma utilizada pelo algoritmo para alinhamento global, exceto pela presença de uma quarta possibilidade para preenchimento do elemento $(i, j)$. Neste caso aceita-se um alinhamento vazio, o qual é pontuado com valor zero.

Encerrada esta primeira parte do algoritmo, o segundo passo consiste em encontrar os possíveis alinhamentos locais resultantes. Para isto, encontra-se o elemento de maior valor da matriz (qualquer elemento contendo este valor máximo poderá ser utilizado como ponto de partida) e, a partir dele, realiza-se a operação de *traceback* descrita para

o alinhamento global. No entanto, diferentemente do que ocorre com o alinhamento global, no qual a operação de *traceback* é encerrada quando o elemento [0, 0] da matriz é encontrado, para o caso do alinhamento local, esta operação é terminada no momento em que um valor zero é atingido.

### 2.3.2 Ferramentas para Alinhamento de Seqüências

Apesar de os algoritmos que utilizam programação dinâmica para alinhamento de seqüências fornecerem resultados mais precisos, estes demandam grande quantidade de recursos computacionais. Como alternativa, foram desenvolvidas algumas ferramentas que não necessitam de grandes recursos computacionais, mas, em contrapartida, fornecem resultados apenas próximos ao ótimo, com menor relevância biológica, portanto. Duas destas ferramentas são apresentadas na seqüência.

#### 2.3.2.1 BLAST

Os programas *Basic Local Alignment Search Tool* (BLAST) (ALTSCHUL et al., 1990) estão entre os mais utilizados para pesquisa de seqüências em bases de dados. O resultado destes programas são os alinhamentos locais de maior pontuação entre uma seqüência de consulta e uma base de dados de seqüências. Dentre os algoritmos BLAST desenvolvidos, destacam-se BLASTP, utilizado para alinhamento de proteínas e BLASTN, para alinhar seqüências de DNA.

BLAST retorna uma lista de *segmentos pareados de alta pontuação* entre a seqüência de consulta e seqüências na base de dados. Um *segmento* é uma substring de uma seqüência. Dadas duas seqüências, um *segmento pareado* entre elas é um par de segmentos de mesmo comprimento, um de cada seqüência. Como as substrings possuem o mesmo tamanho, o alinhamento entre elas ocorre de forma que não sejam incluídos *gaps*, principal razão da eficiência de BLAST. Tal alinhamento pode ser pontuado utilizando uma matriz de substituição.

Um *segmento pareado máximo* (*Maximum Segment Pair* - MSP) entre duas seqüências é um segmento pareado com pontuação máxima. Esta pontuação máxima corresponde à medida de similaridade entre as seqüências e pode ser precisamente calculada pelo método de programação dinâmica, como descrito para algoritmos apresentados em seções anteriores. No entanto, BLAST utiliza outro método para estimar tal pontuação, o que o torna muito mais rápido que ferramentas que utilizam programação dinâmica.

Os programas BLAST podem ser pensados como um procedimento algorítmico composto de três passos:

1. Criação de uma lista de palavras (sementes) com alta pontuação: Dada uma quantidade $w$ de caracteres para uma palavra e uma matriz de pontuação, cria-se uma lista de todas as palavras de tamanho $w$ com pontuação acima de um limite $T$ (*threshold*) quando são alinhadas com as palavras da seqüência de consulta. Geralmente, utiliza-se $w = 3$ para seqüências de proteínas e $w = 11$ para seqüências de DNA;

2. Pesquisa de cada palavra nas seqüências da base de dados;

3. Extensão dos alinhamentos: Caso a semente seja encontrada em uma seqüência da base de dados, tenta-se estender o alinhamento, em ambas as direções, sem utilizar *gaps*, enquanto a pontuação do alinhamento não seja menor que um limite $S$. Com isto, os melhores segmentos pareados são mantidos, mas ainda existe a probabilidade de perda de algumas extensões importantes (SETUBAL; MEIDANIS, 1997).

Após estes três passos, BLAST retorna como resultado todos os alinhamentos contendo os segmentos pareados de alta pontuação entre a seqüência de consulta e a base de dados.

#### 2.3.2.2 FAST

Outra família de programas bastante utilizada para pesquisa em base de dados de seqüências é FAST (PEARSON; LIPMAN, 1988). Dentre os programas disponíveis nesta

família, um dos mais populares é FASTA, para alinhar pares de seqüências de proteínas ou DNA. A idéia básica do algoritmo utilizado pela família FAST é comparar cada seqüência da base de dados com a seqüência procurada e retornar, apenas, aquelas com semelhança significativa.

Dadas duas seqüências $s$ e $t$, com tamanhos denotados por $m = |s|$ e $n = |t|$. O algoritmo inicia determinando as *k-tuplas* comuns entre as seqüências, onde uma *k-tupla* corresponde a uma palavra de tamanho $k$ (*ktup*), com $k = 1$ ou 2. Além disso, é necessário determinar um valor de deslocamento (*offset*) de uma tupla comum às seqüências. O deslocamento é um valor entre -$n$ + 1 e $m$ - 1.

A isso segue-se a construção de uma tabela listando todas as posições de uma $k$-tupla na seqüência $s$. Assumindo-se duas seqüências hipotéticas, $s$ = HARFYAAQIVL e $t$ = VDMAAQIA, e $k = 1$, a Tab. 2 apresenta todas as $k$-tuplas de $s$ e suas devidas posições.

Tabela 2: Listagem das $k$-tuplas da seqüência $s$ = HARFYAAQIVL.

| $k$-tupla | Posições |
|---|---|
| A | 2, 6, 7 |
| F | 4 |
| H | 1 |
| I | 9 |
| L | 11 |
| Q | 8 |
| R | 3 |
| V | 10 |
| Y | 5 |

Em seguida, é construída uma outra tabela com as $k$-tuplas de $t$ e os deslocamentos de cada $k$-tupla em relação à $k$-tupla correspondente em $s$, como apresentado na Tab. 3.

Um vetor de deslocamentos, inicializado com zeros, é, então, construído e a cada $k$-tupla comum entre ambas as seqüências, a entrada do vetor, correspondente ao deslocamento da $k$-tupla de $t$ em relação à $k$-tupla de $s$, é incrementada de uma unidade. Para o caso dado como exemplo, o vetor de deslocamento é apresentado na Tab. 4.

Observa-se na Tab. 4 que o deslocamento +2 tem a maior freqüência (4), signif-

Tabela 3: Listagem das $k$-tuplas da seqüência $t$ = VDMAAQIA e seus deslocamentos em relação às $k$-tuplas da seqüência $s$.

| $k$-tupla | Deslocamentos |
|---|---|
| V | +9 |
| D | |
| M | |
| A | -2, +2, +3 |
| A | -3, +1, +2 |
| Q | +2 |
| I | +2 |
| A | -6, -2, -1 |

icando que a maioria dos *matches* são encontrados neste deslocamento. Este método é conhecido como *método da diagonal*, uma vez que o deslocamento pode ser visto como uma diagonal em uma matriz de programação dinâmica (SETUBAL; MEIDANIS, 1997).

Tabela 4: Vetor de deslocamento.

| Valor do deslocamento | Freqüência |
|---|---|
| -7 | |
| -6 | 1 |
| -5 | |
| -4 | |
| -3 | 1 |
| -2 | 2 |
| -1 | 1 |
| 0 | |
| +1 | 1 |
| +2 | 4 |
| +3 | 1 |
| +4 | |
| +5 | |
| +6 | |
| +7 | |
| +8 | |
| +9 | 1 |
| +10 | |

O passo seguinte do algoritmo consiste em agrupar duas ou mais $k$-tuplas que estão na mesma diagonal, formando uma *região*, a qual pode ser vista como um alinhamento local sem *gaps*. A isso segue-se a fase de pontuação das regiões anteriormente estabelecidas, utilizando uma matriz de pontuação, como as citadas em seções anteriores. Este processo é repetido para cada seqüência da base de dados e, para as regiões que apresen-

taram maior pontuação, uma pontuação ótima é calculada utilizando um algoritmo de programação dinâmica restrito a uma banda ao redor das regiões.

Por fim, é importante salientar que o valor atribuído ao parâmetro *ktup* afeta o desempenho do algoritmo aqui apresentado no que se refere à sensibilidade e seletividade do mesmo. *Sensibilidade* diz respeito a habilidade de um algoritmo em reconhecer seqüências relacionadas, mesmo que distantes. Já *seletividade* refere-se à capacidade do algoritmo em descartar falsos positivos, ou seja, *matches* entre seqüências não relacionadas. Embora ambos objetivos sejam de grande importância, tratam-se de metas opostas - um valor baixo para *ktup* aumenta a sensibilidade, enquanto um valor alto para *ktup* favorece a seletividade (SETUBAL; MEIDANIS, 1997).

## 2.4 TRABALHOS QUE INTRODUZEM CONCORRÊNCIA AO CÁLCULO DA MATRIZ DE SIMILARIDADES

O emprego de programação dinâmica para o alinhamento de seqüências biológicas produz, como resultado, alinhamentos ótimos, com informações mais relevantes aos biólogos. Em contrapartida, o tempo de processamento e consumo de memória exigidos por este método inviabilizam sua utilização para grandes seqüências em arquiteturas seqüenciais convencionais. Uma das alternativas encontradas para viabilizar o uso de programação dinâmica no alinhamento é paralelizar tal método e, por conseqüência, distribuir a carga de processamento entre diversos processadores, acelerando o processo e aumentando a quantidade de memória disponível para o alinhamento.

Martins e colaboradores (MARTINS et al., 2001b) destacam a possibilidade de paralelizar o método de programação dinâmica, mais especificamente o cálculo da matriz de similaridades, naqueles algoritmos que utilizam tal método para o alinhamento. Dadas as dependências de dados existentes entre os elementos da matriz de similaridades, são propostas três formas de preenchimento da matriz: linha por linha, coluna por coluna ou

pelas antidiagonais. Como cada elemento de uma linha (ou coluna) depende dos elementos da mesma linha (ou coluna) para ser calculado, não é possível calcular uma linha (ou coluna) em paralelo. São calculadas, então, as antidiagonais em paralelo. Além disso, para diminuir a quantidade de comunicações necessárias para o cálculo dos elementos da matriz (cada elemento necessita dos valores de seus vizinhos para ser calculado), a matriz de similaridades é dividida em blocos de $q$ linhas e $r$ colunas (Figura 6). Como o tamanho das antidiagonais varia durante o cálculo, toda uma linha de blocos é atribuída ao mesmo processador (Figura 7), para amenizar o problema do balanceamento de carga entre os processadores e mantê-los ocupados o maior tempo possível durante a execução do alinhamento. Os experimentos realizados foram baseados na implementação EARTH (THEOBALD, 1999) para a plataforma MANNA e variaram o número de nós de processamento de 4 a 120 e o tamanho das seqüências a serem alinhadas de 512 a 10K elementos (k = 1024). Os resultados mostraram que a estratégia de paralelização foi satisfatória, apresentando bom desempenho. O objetivo de balanceamento de carga também foi atingido: com 8 nós, os processadores permaneceram utilizados 99% do tempo; com 120 nós e um problema maior, o tempo de utilização foi de 75%.

| | | | | |
|---|---|---|---|---|
| P1 | 1 | 2 | 3 | 4 |
| P2 | 5 | 6 | 7 | 8 |
| P3 | 9 | 10 | 11 | 12 |
| P4 | 13 | 14 | 15 | 16 |
| P1 | 17 | 18 | 19 | 20 |
| P2 | 21 | 22 | 23 | 24 |
| P3 | 25 | 26 | 27 | 28 |
| P4 | 29 | 30 | 31 | 32 |

Figura 6: Divisão em blocos da matriz de similaridades.

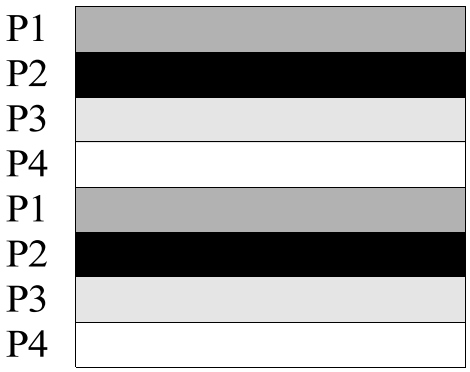


Figura 7: Distribuição de blocos entre processadores.

Em outro trabalho, Martins e colaboradores (MARTINS et al., 2001a) utilizaram a mesma estratégia de paralelização descrita no parágrafo anterior, porém substituíram a implementação MANNA pela implementação Beowulf de EARTH. Além disso, as

seqüências a serem alinhadas variaram de 30K a 900K elementos e os resultados foram comparados aos de outras ferramentas para alinhamento de seqüências. Os experimentos foram executados em uma máquina Beowulf (um aglomerado de computadores padrão executando Linux) composta de 64 nós biprocessados (128 processadores no total). A versão seqüencial da aplicação desenvolvida executou em 53 horas. Com a versão paralela executando nos 64 nós do aglomerado, o tempo foi reduzido para 1,3 horas. Para validar a ferramenta proposta no trabalho (*Another Tool for Genome Comparison* - ATGC), os resultados dos alinhamentos foram comparados aos de outras ferramentas (MUMmer[1] e Blast2[2]) e mostraram que, em alguns casos, ATGC foi mais sensitivo que MUMmer.

O trabalho de Alves e colaboradores (ALVES et al., 2003) propõe um algoritmo paralelo para comparação de seqüências baseado no modelo *Coarse Grained Multicomputers* (CGM) (DEHNE; FABRI; RAU-CHAPLIN, 1996), similar ao modelo BSP (VALIANT, 1990). No modelo CGM, o termo *coarse granularity* vem do fato que o tamanho do problema em cada processador $n/p$ é consideravelmente maior que o número de processadores ($p$). Um algoritmo CGM consiste de uma seqüência de ciclos que alternam entre computação local e comunicação global. O objetivo do modelo CGM é minimizar o número de *super-steps* e a quantidade de computação local. O algoritmo encontra a similaridade entre duas seqüências $A$ e $C$, com $|\mathrm{A}| = m$ e $|\mathrm{C}| = n$, utilizando, para tanto, $p$ processadores. Inicialmente, a seqüência $A$ é distribuída para todos os processadores e a seqüência $C$ é dividida em $p$ partes de tamanho $n/p$ e cada processador $P_i$, $1 \leq i \leq p$, recebe o $i$-ésimo pedaço de $C$ (Figura 8).

Desta forma, cada processador, em um ciclo $k$, irá calcular uma parte da matriz de similaridades, sendo responsável pelo cálculo de toda uma coluna da matriz. Ao concluir o cálculo de um bloco da matriz, o processador $P_i$ envia o resultado ao processador vizinho $P_{i+1}$, responsável por calcular a próxima coluna de blocos. O processador $P_{i+1}$ inicia sua computação e o processador $P_i$ passa a calcular o próximo bloco a partir do

[1]http://www.tigr.org/ tigr-scripts/CMR2/webmum/mumplot
[2]http://www.ncbi.nlm.nih.gov/ blast/bl2seq/bl2.html

bloco anteriormente calculado. O algoritmo segue neste processo de "inundação" até que o processador $P_p$ finalize a computação do último bloco da matriz. Para melhor balancear a carga de trabalho dos processadores, o parâmetro $\alpha$ é inserido e as mensagens enviadas do processador $P_i$ ao processador $P_{i+1}$ deixam de ter tamanho $m/p$ e passam a ter tamanho $\alpha m/p$ (Figura 9). Para verificar a eficiência do algoritmo proposto, foram conduzidos experimentos em um aglomerado Beowulf com 64 nós, seqüências com tamanhos $m = 8192$ e $n = 16384$ e diferentes valores de $\alpha$. Utilizando todos os nós, os menores tempos de processamento foram obtidos quando $\alpha$ assumiu os valores 1/4 e 1/8.

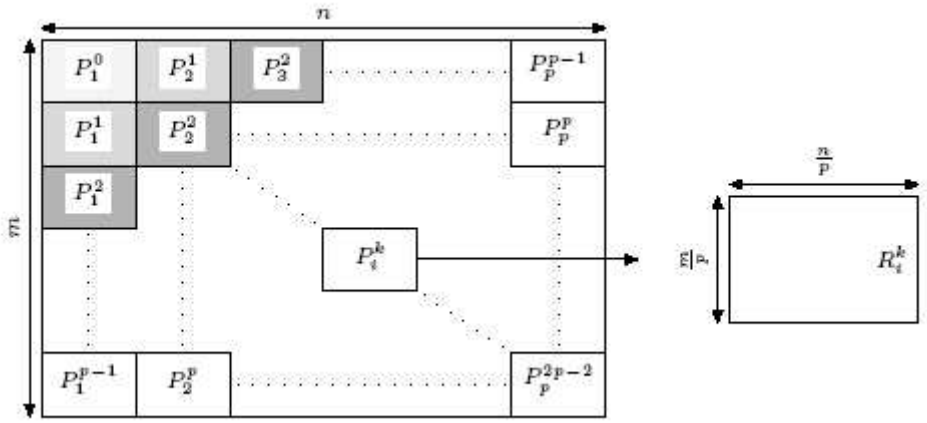


Figura 8: Distribuição de seqüência entre processadores.

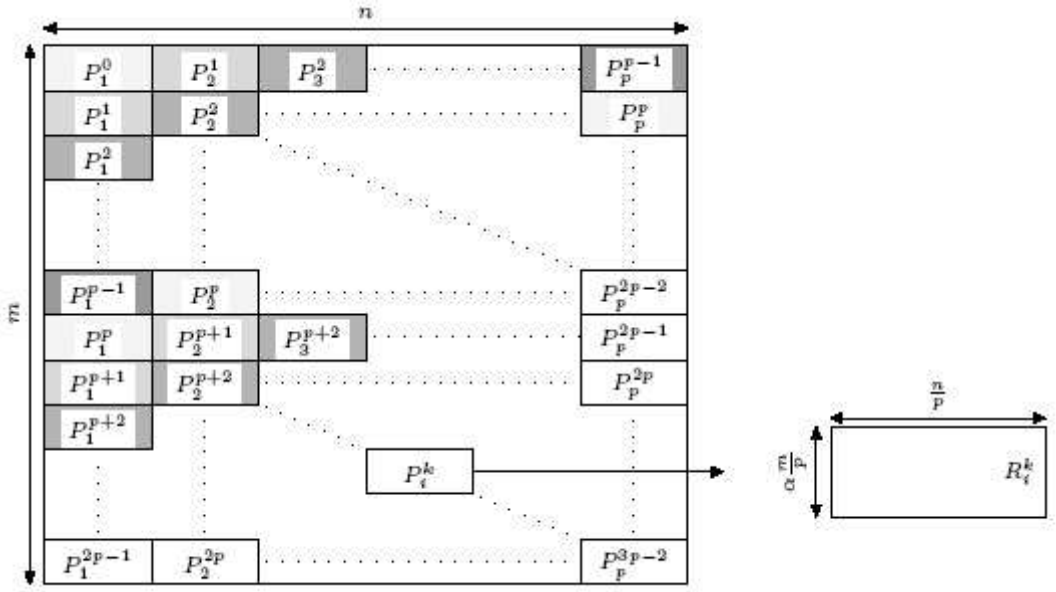


Figura 9: Distribuição de seqüência entre processadores com parâmetro $\alpha$.

O mesmo algoritmo descrito acima é utilizado por Almeida e colaboradores (ALMEIDA JR. et al., 2003) para comparar os genomas completos dos organismos *Xanthomonas axonopodis* pv. *citri*, com 5.175.554 pares de bases, e *Xanthomonas campestris* pv. *campestris*, com 5.076.187 pares de bases. Os resultados experimentais estão divididos em três fases, sendo que a que consome maior tempo destina-se a encontrar os pares de genes homólogos entre os dois genomas. Dois genes são ditos *homólogos* se são descendentes de um mesmo gene ancestral. Para encontrar os pares homólogos, todos os genes de um genoma são comparados a todos os genes do outro genoma, totalizando mais de 18 milhões de comparações no caso dos organismos estudados. Executando a solução paralela para encontrar os genes homólogos em um aglomerado Beowulf de 64 nós formado por computadores de baixo custo, o tempo de processamento foi de 1h15min, incluindo as fases de computação e comunicação. Em um trabalho similar (SILVA et al., 2002) o cálculo levou cerca de 3 horas para ser concluído, cabendo salientar que o método utilizado neste caso foram as heurísticas Blast e EGG (ALMEIDA JR., 2003), em oposição ao trabalho de Almeida e colegas, onde o método empregado foi o de programação dinâmica, o que favoreceu a obtenção de resultados com melhor qualidade do ponto de vista biológico.

Galper e Brutlag (GALPER; BRUTLAG, 1990) apresentam uma adaptação do método de programação dinâmica para execução em uma arquitetura com memória compartilhada, desenvolvendo um algoritmo que utilize todos os processadores disponíveis para prover o maior ganho de desempenho possível, sem, no entanto, sacrificar a simplicidade do método de programação dinâmica e a qualidade dos resultados obtidos pelo mesmo. Na primeira estratégia de paralelização apresentada, cada processador $p$ recebe uma linha de blocos a ser calculada, podendo, tal estratégia, ser dividida em síncrona (Figura 10) e assíncrona (Figura 11). No caso síncrono, o cálculo da matriz de similaridades avança $p$ linhas por vez, já que, quando um processador termina o cálculo da linha destinada a ele, deve aguardar até que todos os processadores tenham terminado suas linhas, para, então, todos receberem novas linhas a serem calculadas ao mesmo tempo. Já no caso assíncrono, o cálculo avança continuamente, pois quando um processador completa uma linha, ele

avança para o cálculo da próxima linha disponível, sem necessitar esperar pelos demais. Em ambos os casos, ao detectar a indisponibilidade de um valor necessário à continuidade do cálculo da linha, o processador bloqueia, aguardando que o dado seja computado.

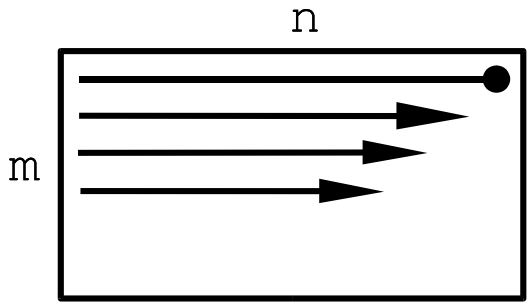


Figura 10: Cálculo síncrono por linha.

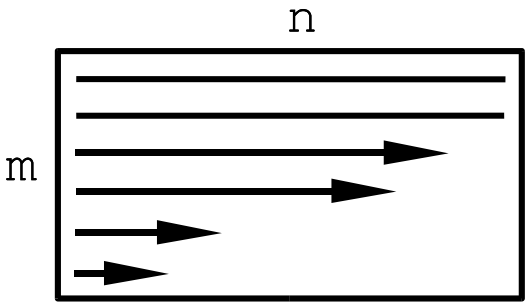


Figura 11: Cálculo assíncrono por linha.

A segunda estratégia apresentada, realiza os cálculos ao longo das diagonais da matriz de similaridades. No caso síncrono desta estratégia (Figura 12), os elementos de uma diagonal são divididos entre todos os processadores; ao terminar a computação do elemento destinado a ele, um processador pode analisar qual dos elementos da diagonal ainda não foi calculado e tomar o cálculo para si; quando todos terminarem, a próxima diagonal é dividida e calculada. O caso assíncrono (Figura 13), por sua vez, determina que cada processador receba uma diagonal inteira para calcular. Como antes, em ambos os casos, a indisponibilidade de um dado bloqueia um processador. Os experimentos realizados em um multiprocessador Encore Multimax com 16 nós, para vários tamanhos de seqüência, mostraram que a primeira estratégia (versões síncrona e assíncrona) apresentaram os maiores ganhos de tempo em relação a versão seqüencial.

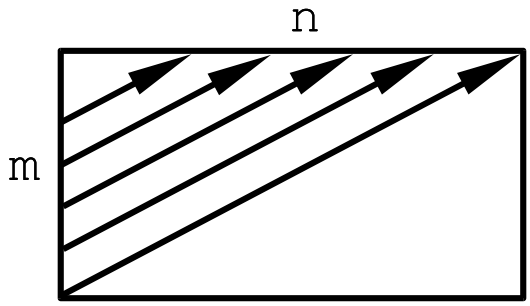


Figura 12: Cálculo síncrono por diagonais.

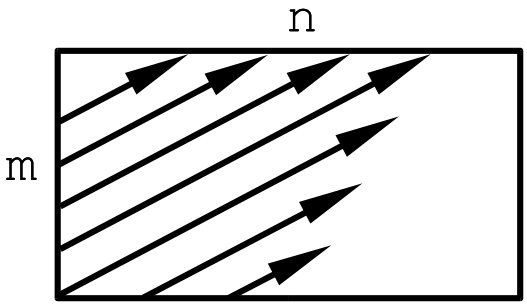


Figura 13: Cálculo assíncrono por diagonais.

## 2.5 COMENTÁRIOS FINAIS

A grande quantidade de seqüências biológicas disponíveis atualmente para serem analisadas, torna necessário o emprego de técnicas computacionais capazes de auxiliarem os biólogos no processo de análise, de forma a se obter mais rapidamente os resultados desejados.

Neste capítulo, abordou-se o problema do alinhamento de seqüências, considerado o processo primitivo mais importante na Biologia Computacional. Para automatizar tal processo, foram destacados dois conjuntos de técnicas: as que empregam o método de programação dinâmica e as que não o empregam (técnicas heurísticas). As técnicas que empregam programação dinâmica, fornecem resultados ótimos, sem perda de informações, mas demandam grande quantidade de recursos computacionais no que se refere a tempo de processamento e consumo de memória. Por outro lado, as técnicas heurísticas produzem resultados até 40 vezes mais rápido que as que empregam programação dinâmica e consomem menor quantidade de memória. No entanto, os resultados obtidos são apenas próximos ao ótimo, podendo haver perda de informações relevantes aos biólogos.

Uma alternativa capaz de viabilizar o uso do método de programação dinâmica consiste em paralelizar tal método. Com isso, a carga computacional é distribuída entre diversos processadores, diminuindo o tempo de processamento e possibilitando o alinhamento de seqüências maiores. Foram destacados alguns trabalhos encontrados na literatura que empregam estratégias de paralelização ao método de programação dinâmica. Diferentes estratégias foram encontradas, mas, em todos os casos, foram consideradas as questões relativas às dependências de dados existentes no cálculo da matriz de similaridades. Pode-se observar, ainda, que em todos os casos obteve-se ganho das execuções paralelas em relação as execuções seqüenciais.

Dados os resultados ótimos fornecidos pelo método de programação dinâmica e a possibilidade de introdução de concorrência no mesmo, optou-se pela implementação de

uma aplicação para alinhamento de seqüências que emprega este método. Além disso, o alinhamento com programação dinâmica pode ser descrito em termos de tarefas e de suas dependências de dados. Uma execução eficiente em aglomerado (arquitetura que temos disponível) pode ser obtida pela manipulação de um grafo que descreve estas tarefas e dependências.

# 3 PROGRAMAÇÃO PARALELA E DISTRIBUÍDA

Historicamente, a demanda por poder computacional vem, principalmente, de áreas como Ciência, Engenharia, Defesa, Economia, Ambiente e Sociedade. Os problemas mais complexos nessas e outras áreas, chamados de "grandes desafios" para a ciência, foram classificados por Levin (LEVIN, 1989) dentro de diversas categorias:

- Química quântica, mecânica estatística e física relativística;
- Cosmologia e astrofísica;
- Biologia, farmacologia, seqüenciamento de genomas, engenharia genética e modelagem de células;
- Medicina e modelagem de órgãos e ossos humanos;
- Clima global e modelagem ambiental;
- Dinâmica de fluidos e turbulência;
- Design de materiais e supercondutividade.

Quando se fala na busca de soluções para estes grandes e complexos problemas, vem à mente a utilização, se possível, de supercomputadores com hardware e software especializados a cada problema. Idealmente, esta seria a solução buscada pela maioria dos envolvidos no estudo destes problemas. No entanto, adquirir computadores de grande porte a baixos custos, ou pelo menos razoáveis, não é uma realidade com as tecnologias

atuais. Dadas as condições financeiras da maioria das instituições onde se desenvolve pesquisa nas áreas citadas e os grandes investimentos necessários à aquisição e manutenção de supercomputadores, esta solução torna-se, praticamente, inacessível.

Como alternativa ao exposto anteriormente, surgem os aglomerados de computadores. Aglomerados são sistemas de computação paralela, compostos por um conjunto de computadores independentes (estações de trabalho ou computadores pessoais, por exemplo) que trabalham de forma integrada como se fossem um recurso computacional único (BUYYA, 1999). O crescimento no uso de aglomerados de computadores pode ser observado através das estatísticas apresentadas no site www.top500.org, que realiza, semestralmente, uma classificação das 500 máquinas mais poderosas do mundo.

A partir do momento que se tem disponível uma arquitetura dotada de múltiplos recursos de processamento, como, por exemplo, os aglomerados de computadores, o grande problema que se observa é a necessidade de desenvolvimento de uma forma eficiente de explorar este tipo de arquitetura, utilizando, para tanto, o paradigma de programação concorrente. Nesta forma de programação, também chamada de programação paralela ou distribuída, uma aplicação pode ser descrita por programas capazes de serem executados por diversos fluxos de execução independentes.

O objetivo deste capítulo é apresentar alguns tópicos relacionados à programação paralela e distribuída. A Seção 3.1 aborda alguns conceitos referentes à programação concorrente, com destaque aos níveis de concorrência que podem ser explorados em uma arquitetura para execução de aplicações. Na Seção 3.2 é apresentado o conceito de multiprogramação leve, utilizada para exploração da concorrência intra-nó. A Seção 3.3 destaca algumas ferramentas de comunicação empregadas na troca de dados e tarefas entre os nós de um aglomerado de computadores. Por fim, na Seção 3.4 são feitos alguns comentários sobre o que foi apresentado no capítulo, destacando os pontos mais importantes do mesmo.

## 3.1 A PROGRAMAÇÃO CONCORRENTE

Em geral, utiliza-se o termo *concorrência* para definir sistemas nos quais há a possibilidade de compartilhamento dos recursos de um computador, tais como processador, memória, dispositivos de entrada/saída (E/S), etc., a fim de suportar o fluxo de execução de dois ou mais programas independentes ((TANENBAUM, 1992) (SILBERSCHATZ; GALVIN, 1997)). A idéia por trás deste compartilhamento é prover as duas principais características de um ambiente concorrente: alto desempenho e eficiência. O alto desempenho é obtido sobrepondo-se computação e comunicação, de maneira que, ao ser constatada a inatividade do processador, durante a realização de operações de E/S por um programa, um novo programa é selecionado para utilizar o processador, evitando o desperdício de tempo da unidade de processamento. Já a segunda característica refere-se ao fato de uma aplicação poder utilizar todos os recursos disponibilizados por uma arquitetura, da forma mais otimizada possível, buscando, com isso, atingir alto desempenho e aumentar a quantidade de tarefas executadas em um intervalo de tempo (NAVAUX, 2001).

Apesar das vantagens apresentadas pela execução concorrente de programas, não é tarefa trivial utilizar este modelo de programação. Um dos fatores que contribuem para isto é a inexistência de um modelo universal para computação paralela similar ao modelo de von Neumann para a computação seqüencial (GOLDMAN, 2003). De acordo com o modelo de von Neumann, um computador consiste de um processador, uma unidade aritmética e uma área de memória. Neste modelo, as instruções são executadas seqüencialmente, na ordem especificada pelo programa fonte dentro de um fluxo de execução único, existindo um controle implícito que não permite que uma instrução execute antes que a anterior termine, garantindo que dados produzidos pela execução de instruções estejam disponíveis para leitura na memória. Um programa concorrente, por sua vez, possui diversos fluxos de execução, cada um com sua própria seqüência de instruções, sem um controle implícito das relações de ordem de execução entre os diferentes fluxos, portanto sem mecanismos implícitos de controle a dados em memória. Além disso, os fluxos divi-

dem o acesso aos dispositivos de E/S e competem pelo tempo de utilização da unidade de processamento.

Neste caso, passa a ser associado à concorrência um novo significado: o de competição. Os programas em execução, inteiramente independentes e sem qualquer forma de colaboração entre si, disputam o uso dos recursos de arquitetura para garantirem que suas necessidades de execução serão atendidas. Esta disputa reflete o compartilhamento tanto do tempo de utilização do processador quanto do espaço de armazenamento na memória como de qualquer outro recurso da arquitetura. Assim, é preciso haver mecanismos capazes de garantir que a execução de um programa não interfira na execução de outro, uma vez que não existe um controle implícito das instruções.

Existe, ainda, de acordo com (WILKINSON; ALLEN, 1999), uma segunda abordagem para concorrência: colaboração. Neste caso, uma única aplicação é decomposta em diversos fluxos de execução. Embora cada fluxo seja responsável pela execução de uma parte do problema, eles ainda necessitam trocar informações entre si para comporem a solução final, o que não os torna completamente independentes. Assim, programar de forma concorrente implica em detectar na aplicação tanto as atividades que não possuem restrições temporais de execução, as quais irão concorrer por recursos de processamento, quanto as que necessitam compartilhar dados, estas concorrendo por informação em uma determinada posição de memória. Desta forma, diferentes fluxos de execução colaboram entre si, utilizando mecanismos de sincronização (SEBESTA, 2000), a fim de chegarem ao resultado esperado.

Um programa concorrente, portanto, é composto por um conjunto de fluxos de execução que, coordenadamente, trocam informações. Os fluxos de execução dão suporte à execução das tarefas definidas na aplicação e as sincronizações determinam como ocorrem as trocas de dados entre estas tarefas, as quais devem ser executadas respeitando as sincronizações existentes entre elas, para que o resultado final da aplicação seja o esperado (CAVALHEIRO, 2004).

No contexto de um programa concorrente, quando se fala em **tarefa** está se tratando de um conjunto de instruções capazes de resolver uma parte de um problema maior, no caso, a aplicação como um todo. As instruções que formam uma tarefa devem ser executadas de forma seqüencial sob um fluxo de execução. Para que uma tarefa possa ter sua execução iniciada, ela necessita ter disponível um conjunto de parâmetros de entrada. Ao final do processamento sobre estes parâmetros iniciais, será produzido um conjunto de dados de saída. Eventualmente, durante a execução de um programa, os dados de saída de uma tarefa podem servir como parâmetros de entrada para outra(s). Neste caso, as duas tarefas devem ser executadas de forma síncrona, uma vez que há restrições temporais entre elas, sendo necessário respeitar a ordem de execução das tarefas. Caso não haja nenhuma relação entre os dados de saída de uma tarefa e os parâmetros de entrada de outra, estas são consideradas tarefas independentes, não havendo restrições temporais para suas execuções.

No caso em que há uma dependência entre as tarefas, surge a necessidade de sincronização entre elas. Sincronização é um mecanismo que controla a ordem de execução das tarefas (SEBESTA, 2000), determinando a partir de qual momento da execução uma tarefa pode passar a acessar os dados gerados por outra. Esta sincronização entre produção e consumo de dados entre as tarefas é que coordena a forma como elas comunicam-se (trocam dados). Durante a execução de um programa, os mecanismos de sincronização utilizados são responsáveis por garantir que as tarefas serão executadas segundo a ordem definida pelo programa.

Durante a execução de uma aplicação, as tarefas que a compõem trocam de estado, podendo assumir um dos quatro estados seguintes (GALLILÉE et al., 1998):

- **Pronta**: a tarefa pode ser executada a qualquer momento, pois os dados necessários a sua execução já estão disponíveis;
- **Aguardando**: a tarefa poderia passar ao estado de pronta não fosse a indisponibilidade dos dados necessários ao início de sua execução;

- **Executando**: as instruções de uma tarefa estão sendo executadas seqüencialmente;
- **Concluída**: a tarefa produziu seus dados de saída e liberou o fluxo de execução ao qual estava associada.

Outro aspecto a ser considerado na programação concorrente refere-se aos níveis de concorrência que podem ser explorados durante a execução de uma aplicação. Este aspecto depende da arquitetura utilizada e, ao considerar-se um aglomerado de computadores, a concorrência se apresenta intra-nó e entre-nós. A principal diferença entre as duas está na forma como os fluxos de execução interagem entre si. No primeiro caso, os fluxos compartilham um mesmo espaço de endereçamento, trocando dados através desta área de memória comum. Já, no segundo caso, os fluxos necessitam fazer uso de algum mecanismo de comunicação, como, por exemplo, troca de mensagens, para trocarem informações entre si.

### 3.1.1 Concorrência Intra-nó

A concorrência intra-nó é aquela que ocorre entre atividades concorrentes executadas em um mesmo nó de um aglomerado (ou dentro de um computador independente). Este tipo de concorrência pode ser classificada como real ou temporal, dependendo do número de fluxos de execução ativos em um dado momento e do número de processadores disponíveis no nó. Se o número de processadores for igual ou maior que o número de fluxos de execução ativos em um determinado instante, esta concorrência é dita real (o chamado paralelismo). Se, no entanto, existirem mais fluxos de execução do que processadores, tem-se uma concorrência temporal (clássica), onde há compartilhamento de recursos.

Para que fluxos de execução compartilhem informações onde há este tipo de concorrência, o mecanismo de comunicação empregado é o que faz uso de operações de leitura e escrita no espaço de endereçamento comum a estes fluxos. Assim, um dado escrito na memória por uma instrução em um fluxo de execução, pode ser lido por qualquer instrução em outro fluxo. Embora este mecanismo de comunicação funcione de forma idêntica ao

de programas seqüenciais, com operações de leitura e escrita, no caso de uma execução concorrente devem ser tomados cuidados adicionais para garantir sincronia na produção e consumo dos dados compartilhados.

Diferentemente do que ocorre em uma execução seqüencial, onde a sincronização entre duas instruções é realizada implicitamente, pela garantia que uma instrução somente será executada após o término da instrução que a precede, em uma execução concorrente não existe o sincronismo implícito entre instruções de diferentes fluxos de execução. Neste caso, somente a utilização correta da sincronização é capaz de garantir que a comunicação entre as tarefas será realizada como esperado. Para tanto, um mecanismo externo deve realizar a sincronização entre a instrução que produz o dado e a instrução que necessita deste dado.

Os mecanismos de sincronização mais utilizados são aqueles que garantem exclusão mútua no acesso a memória (*mutexes*) e os que realizam controle no avanço de execução, tais como criação de novos fluxos de execução (*create*) e bloqueio aguardando o término da execução de um fluxo (*join*). As ferramentas que possibilitam a exploração da concorrência intra-nó mais populares baseiam-se no padrão POSIX (AMERICAN NATIONAL STANDARDS INSTITUTE, 1994) para processos leves (ou *threads*).

### 3.1.2 Concorrência Entre-nós

Como mencionado anteriormente, está sendo considerada, como arquitetura de suporte à execução de uma aplicação, um aglomerado de computadores. Neste tipo de arquitetura, cada nó possui seus próprios recursos de processamento. Considerando que diferentes fluxos executam em diferentes nós, não existe entre eles uma competição pelos recursos de processamento de um nó. Existe, sim, a possibilidade de explorar o paralelismo real da arquitetura caso as tarefas a serem executadas sejam independentes e possam executar ao mesmo tempo. Embora estejam sendo consideradas tarefas independentes, ainda podem ser necessárias algumas sincronizações e trocas de dados entre elas. No

entanto, ao contrário do que foi considerado quando se tratou da concorrência intra-nó, não é possível realizar trocas de dados entre as tarefas através de um espaço de endereçamento comum, uma vez que, neste caso, cada nó possui seu próprio espaço de memória. Para a cooperação entre as tarefas, utiliza-se, então, a rede de interconexão entre os nós, fato, este, que caracteriza uma *aplicação distribuída*, onde o termo *distribuída* faz alusão à memória, a qual encontra-se distribuída entre os nós.

A partir do momento que não se tem mais a possibilidade de trocar dados através da memória comum entre as tarefas, utilizando primitivas de *leitura* e *escrita*, e passa-se a utilizar a rede que conecta os nós, as primitivas de comunicação também mudam e passam a ser do tipo *envia* e *recebe*. Neste caso, as tarefas interagem por meio da troca de mensagens; uma forma de interação mais complexa e onerosa do que quando a memória é comum às tarefas, pois implica na necessidade do emprego de algum mecanismo de endereçamento. Surge, aqui, um quinto estado para uma tarefa, a ser somado àqueles apresentados na Seção 3.1: **local**. Tal estado corresponde a uma tarefa que pode ser executada e os dados necessários à sua execução estão presentes no módulo de memória local (CAVALHEIRO, 2001).

Durante a execução de um programa, a chamada tanto a uma primitiva *envia* quanto a uma *recebe* pode ser vista como o fim de uma tarefa e o início de outra. Quando uma tarefa faz uma chamada à primitiva *envia*, é considerada terminada e os dados produzidos como resultado, são enviados à tarefa *receptora*. Por outro lado, se a chamada é realizada à primitiva *recebe*, a tarefa chamadora é considerada terminada e as instruções após o *recebe* definem a próxima tarefa, que passará ao estado de *pronta* assim que receber os dados necessários ao início de sua execução.

Entre as ferramentas encontradas na literatura que possibilitam o compartilhamento de dados por meio da troca de mensagens, podem ser citadas *Parallel Virtual Machine* (PVM) (GEIST et al., 1994) e *Message Passing Interface* (MPI) (FORUM, 1994), além do padrão *sockets* (STEVENS, 1998).

### 3.1.3 Concorrência Intra e Entre-nós

Como visto, tanto a concorrência intra-nó quanto a entre-nós pode ser explorada em um aglomerado de computadores. Dessa forma, para uma melhor exploração dos recursos de processamento de um aglomerado, o ideal é empregar as duas formas de concorrência. No entanto, a utilização simultânea destes dois mecanismos não é uma tarefa simples (CARISSIMI, 1999). O emprego dos dois níveis de concorrência se justifica pelo ganho que pode ser obtido pelo recobrimento de parte do tempo gasto em comunicações pela execução de cálculo útil (VALIANT, 1990); mesma idéia presente quando um processo que necessita realizar uma operação de E/S perde o processador para outro que está na fila de espera para execução.

Na bibliografia podem ser encontradas ferramentas que disponibilizam recursos para exploração dos dois níveis de concorrência. Dentre elas, podem ser citadas as bibliotecas Athapascan (CARISSIMI, 1999), Nexus (FOSTER; KESSELMAN; TUECKE, 1996) e $PM^2$ (DENNEULIN; NAMYST; MÉHAUT, 1997) e a linguagem de programação Java (DEITEL; DEITEL, 2000).

## 3.2 MULTIPROGRAMAÇÃO LEVE

A multiprogramação leve (*multithreading*) é a ferramenta comumente utilizada para explorar a concorrência intra-nó. Esta estratégia permite que vários fluxos de execução sejam criados no interior de um processo. Cada um destes fluxos recebe o nome de processo leve ou *thread*. O termo *leve* refere-se ao compartilhamento de recursos, alocados a um processo, por todas suas *threads* ativas (VAHALIA, 1976), não sendo necessário que cada *thread* possua sua própria descrição de recursos. Isto torna a manipulação de *threads* menos onerosa ao sistema operacional.

Um dos recursos compartilhados pelas *threads* é a memória do processo, que serve como base de comunicação e troca de dados, sendo acessada através de primiti-

vas de *leitura* e *escrita*. O problema, neste caso, está em garantir um acesso correto a esta memória compartilhada, sendo necessário o emprego de algum mecanismo de sincronização para este fim.

Cabe destacar que uma *thread* não é necessariamente uma tarefa, mas sim um suporte à execução do conjunto de instruções pertencentes a uma tarefa. Com esta distinção é possível diferenciar a concorrência existente entre as atividades de uma aplicação da concorrência real que pode ser obtida dependendo da arquitetura (BLACK, 1990). Sendo assim, o programador é incumbido de definir as atividades concorrentes da sua aplicação e codificá-las em tarefas em um programa. As *threads* serão associadas a estas tarefas no momento da execução e dependerão dos recursos de processamento disponíveis na arquitetura de suporte à execução da aplicação. Enquanto aguardam o momento para serem mapeadas sobre uma *thread* disponível, as tarefas podem permanecer em uma fila de tarefas prontas (SEBESTA, 2000; CAVALHEIRO; DENNEULIN; ROCH, 1998).

## 3.3 COMUNICAÇÃO EM AGLOMERADOS

Considerando-se a execução de uma aplicação em um aglomerado de computadores, um dado produzido em um nó do aglomerado por uma tarefa poderá ser consumido por outra tarefa em um nó diferente. Esta comunicação entre tarefas executando em nós distintos, é uma das operações mais custosas em se tratando de processamento concorrente em arquiteturas com memória distribuída. Como existe a necessidade de processamento de alto desempenho, busca-se reduzir o tempo entre a produção de um dado e seu consumo.

Dessa forma, um bom desempenho da aplicação está diretamente ligado ao sobrecusto introduzido pela troca de dados entre os nós. Com base nessa situação, protocolos de comunicação mais eficientes e hardwares para comunicação com melhor desempenho têm sido desenvolvidos, buscando explorar com maior eficiência os recursos oferecidos pelas redes de comunicação e reduzir o sobrecusto de comunicações.

O restante desta seção dedica-se a apresentar algumas ferramentas de comunicação

que podem ser utilizadas para a troca de dados entre os nós de um aglomerado.

### 3.3.1 Troca de Mensagens

A troca de mensagens é a estratégia tradicionalmente empregada para o compartilhamento de dados entre os nós de um aglomerado. Para a comunicação são utilizadas operações do tipo *send()*, para enviar dados, e *receive()*, para receber dados. Variantes desta estratégia permitem comunicações síncronas ou assíncronas.

No caso síncrono, *send()* e *receive()* são operações bloqueantes, ou seja, uma chamada *send()* permanece bloqueada até que o *receive()* correspondente seja executado. Neste caso, o transmissor somente enviará os dados após receber do receptor uma confirmação de que este encontra-se pronto para a recepção. Nitidamente, utilizando-se a troca de mensagens síncrona, não é possível realizar processamento e comunicação ao mesmo tempo (EICKEN et al., 1992).

Por outro lado, a variante assíncrona permite que parte do tempo gasto com comunicações seja sobreposto por cálculo efetivo. Isto é possível, pois o transmissor de uma mensagem não necessita esperar nem pelo requerimento da mensagem por parte do receptor nem pela confirmação de recebimento desta mensagem. Com isso, após enviar uma mensagem, o transmissor está livre para novas transmissões ou para a realização de cálculo efetivo. O custo do uso desta técnica reflete-se em uma maior complexidade de controle da evolução do programa e da respectiva troca de dados.

Duas das mais populares bibliotecas de comunicação baseadas em troca de mensagens para processamento de alto desempenho são PVM e MPI. Cita-se, também, o padrão *sockets*, bastante popular em ambientes UNIX.

#### 3.3.1.1 PVM

A biblioteca PVM (GEIST et al., 1994) permite que uma rede de computadores heterogêneos seja utilizada como uma única máquina virtual para execução de aplicações

paralelas e distribuídas. A máquina virtual pode ser composta por um número praticamente ilimitado de *hosts* heterogêneos (COSTA; STRINGHINI; CAVALHEIRO, 2002).

O sistema PVM é composto de duas partes: uma biblioteca de funções paralelas e um *daemon*, chamado *pvmd3*, que reside em todos os computadores que fazem parte da máquina virtual. A biblioteca fornece primitivas de comunicação, inicialização e término de processos PVM, adição e remoção de *hosts* da máquina virtual, sincronização e envio de sinais entre processos PVM, dentre outras. O *daemon* consiste em um processo que executa em segundo plano, sendo responsável pelo gerenciamento das tarefas paralelas oferecidas pelo PVM. O conjunto de *daemons* executando em diferentes máquinas, forma a máquina virtual do PVM (COSTA; STRINGHINI; CAVALHEIRO, 2002).

Com PVM, qualquer processo PVM pode enviar uma mensagem para qualquer outro processo PVM, não havendo limite para o tamanho nem para o número destas mensagens. Esta comunicação entre processos é baseada em operações *send* e *receive*, com envio e recebimento assíncronos e, ainda, recepção síncrona.

#### 3.3.1.2 MPI

MPI é uma biblioteca de comunicação para programação paralela baseada em troca de mensagens, que surgiu durante o MPI Fórum organizado por laboratórios, indústrias e universidades, onde foi definido um vasto conjunto de primitivas relacionadas à sincronização e comunicação entre tarefas.

Assim como PVM, MPI também cria uma máquina virtual sobre uma arquitetura real. Cada nó virtual consiste em um processo executando sobre um nó real da arquitetura, sendo que mais de um nó virtual poderá residir no mesmo nó real e nós virtuais poderão estabelecer comunicação para envio e recebimento de dados quando necessário.

As primitivas para troca de dados entre processos oferecidas por MPI, vão desde as tradicionais *send* e *receive* até mecanismos de comunicação em grupo. Além disso, um processo pode tanto enviar dados para todos os processos quanto receber dados de todos

os processos. Para tanto, o nó deve ser indicado como raíz, sendo responsável por enviar, receber e tratar todos os dados.

#### 3.3.1.3 Bibliotecas *Sockets*

Bibliotecas *sockets* são populares em ambientes UNIX, tal GNU Linux, e sua implementação segue o padrão *sockets* definido em (STEVENS, 1998). É o mecanismo mais básico utilizado para comunicação interprocesso em um computador ou entre aplicações em diferentes computadores conectados por redes locais ou redes de longa distância. Possui um conjunto pequeno de primitivas e pode ser empregado tanto com o padrão TCP (orientado a conexão) quanto com UDP (pacotes), sendo muito eficientes em redes padrão Ethernet.

### 3.3.2 Chamada Remota de Procedimento

Outra estratégia que pode ser empregada para comunicação em aglomerados de computadores consiste em chamadas remotas de procedimento (RPC) (BIRRELL; NELSON, 1984), que oferecem um maior nível de abstração ao programador. RPC oferece ao programador uma estrutura de programação bastante próxima à chamada de um procedimento ordinário para invocar um serviço a ser executado em outro nó (CAVALHEIRO, 2004), passando, como parâmetro, um conjunto de dados.

Um ambiente que implementa RPC é composto por um servidor, responsável pelo cálculo, um cliente, que invoca os serviços do servidor, e um *stub*, que é executado junto ao cliente e consiste em um procedimento responsável por capturar as invocações ao serviço remoto.

A popularidade de RPC, originalmente síncrona, deve-se ao emprego das conhecidas chamadas de procedimento, onde serviços remotos são vistos como procedimentos locais. Além disso, este modelo onde o cliente espera sincronamente pelo servidor, é bem próximo ao esquema seqüencial conhecido pelos programadores (BARCELLOS; GASPARY,

2003).

### 3.3.3 Mensagens Ativas

Mensagens Ativas (EICKEN et al., 1992) são soluções clássicas ao problema de comunicação em ambientes paralelos (LUMETTA; MAINWARING; CULLER, 1997), permitindo realizar comunicações sem introduzir grande quantidade de sobrecusto de execução (WALLACH et al., 1995). O mecanismo de comunicação com Mensagens Ativas é assíncrono e tenta explorar toda a flexibilidade e desempenho das modernas redes de computadores (EICKEN et al., 1992). Neste caso, cada mensagem enviada passa a conter, em seu cabeçalho, o endereço de um serviço a ser executado no receptor e, em seu corpo, os dados que compõem os argumentos (EICKEN et al., 1992). No momento da recepção de uma mensagem ativa, um procedimento é acionado para recuperar a mensagem da rede e inserí-la no processamento em andamento no nó, de forma que o serviço requisitado execute rapidamente e por completo.

A estratégia empregada por este mecanismo para minimizar a sobrecarga de comunicação, está em não *bufferizar* as Mensagens Ativas, a não ser o necessário exigido pelo sistema operacional.

Apesar das semelhanças entre Mensagens Ativas e RPC, cabe destacar que as primeiras foram desenvolvidas com o objetivo de obterem alto desempenho em arquiteturas distribuídas, enquanto as outras preocupam-se apenas com a comunicação em arquiteturas deste tipo.

## 3.4 COMENTÁRIOS FINAIS

É consenso entre pesquisadores as vantagens apresentadas pela utilização da programação concorrente para solucionar problemas que demandam grande quantidade de recursos, seja em tempo de processamento, seja em consumo de memória. No entanto, não é tarefa trivial empregar este modelo de programação. Na programação seqüencial, por

exemplo, as instruções são executadas ordenadamente em um fluxo de execução único, havendo um controle implícito entre elas que não permite que uma instrução execute antes que a anterior termine, o que garante a correta evolução do programa. Em contrapartida, existem diversos fluxos de execução em um programa concorrente, cada um com sua própria seqüência de instruções e que necessitam de mecanismos externos de controle para garantir o correto acesso a memória e a correta evolução do programa.

Eventualmente, estes fluxos necessitam trocar dados entre si. Existindo uma memória compartilhada entre eles, estas trocas ocorrem através de operações de *leitura* e *escrita* diretamente em memória, com algum controle para acesso aos dados compartilhados. Por outro lado, se os fluxos não estiverem executando no mesmo nó, serão necessárias primitivas do tipo *envia* e *recebe* para estabelecer comunicação entre os nós. Assim como deve haver um controle de acesso à memória compartilhada, as comunicações entre os nós também devem ser sincronizadas. Dessa maneira, busca-se garantir que instruções relacionadas entre si e executando em nós diferentes, tenham à sua disposição os dados necessários quando forem executar.

Considerando-se a execução de uma aplicação em um aglomerado, a situação encontrada é a de troca de dados entre os nós do mesmo. Foram apresentadas algumas ferramentas que podem ser empregadas para estabelecimento desta comunicação entre fluxos executando em nós diferentes. Dentre elas, as Mensagens Ativas mostraram-se mais adequadas quando se busca alto desempenho. A restrição é que esta solução permite somente comunicações assíncronas, sendo necessária a introdução de algum mecanismo externo de controle de execução, para garantir a correta evolução do programa.

# 4 MENSAGENS ATIVAS

O desenvolvimento de software que tire o máximo proveito de arquiteturas como os aglomerados de computadores, apresenta três desafios: (1) minimizar o sobrecusto gasto em comunicações, (2) permitir que o tempo gasto em comunicações seja sobreposto por computação (cálculo efetivo) e (3) coordenar os dois anteriores sem sacrificar o desempenho do processador (EICKEN et al., 1992).

Uma aplicação implementada para executar em arquiteturas como aglomerados e que vise alto desempenho, deve buscar minimizar o tempo entre a produção de um dado e seu consumo. Este tempo, que pode ser visto como a reatividade da aplicação, refere-se à capacidade da aplicação em disparar novos serviços à medida que novos dados são produzidos. Considerando-se a execução em um aglomerado de computadores, um dado produzido em um nó do aglomerado por uma tarefa poderá ser consumido por outra tarefa em um nó diferente. Para tanto, será necessária uma troca de mensagens entre os dois nós, a qual deverá ocorrer de maneira rápida, já que o dado produzido deve ser consumido no menor tempo possível.

Uma solução eficiente para implementação de troca de mensagens entre os nós de um aglomerado, consiste na utilização de Mensagens Ativas (EICKEN et al., 1992), que, normalmente, está entre os mecanismos de comunicação mais rápidos disponíveis (LUMETTA; MAINWARING; CULLER, 1997). Este mecanismo é assíncrono e tenta explorar toda a flexibilidade e desempenho das modernas redes de computadores (EICKEN et al., 1992), permitindo realizar comunicações sem introduzir grande quantidade de sobrecustos de execução (WALLACH et al., 1995).

Neste mecanismo, cada mensagem enviada contém, em seu cabeçalho, o endereço de uma função a ser executada no receptor e, em seu corpo, os dados que compõem os argumentos (LUMETTA; MAINWARING; CULLER, 1997). A idéia é que o transmissor envie uma mensagem para a rede e continue seu processamento. A recepção de uma mensagem ativa provoca o acionamento de um procedimento para recuperar a mensagem da rede e a invocação do serviço desejado. Para minimizar o sobrecusto de comunicação, as Mensagens Ativas são *bufferizadas* apenas o necessário para o transporte na rede.

A diferença entre Mensagens Ativas e RPC, está no fato de que o mecanismo de Mensagens Ativas busca extrair os dados da rede o mais rápido possível e executar o processamento requisitado pela mensagem, de forma a minimizar o tempo entre a produção e o consumo dos dados. A grande vantagem deste mecanismo está na forma como a mensagem recebida é tratada: ela é inserida dentro do fluxo de execução corrente, o que evita o sobrecusto de criação de um novo fluxo (CARISSIMI, 1999).

Este capítulo apresenta um estudo sobre o mecanismo de Mensagens Ativas. A Seção 4.1 apresenta a implementação original, em hardware, do mecanismo. Na Seção 4.2 são apresentadas duas implementações alternativas que buscam minimizar as limitações do modelo original. Alguns ambientes de programação que empregam o mecanismo de Mensagens Ativas são apresentados na Seção 4.3. A Seção 4.4 descreve outra implementação alternativa, que foi desenvolvida para inclusão no ambiente Anahy. Alguns comentários finais sobre Mensagens Ativas são feitos na Seção 4.5.

## 4.1 IMPLEMENTAÇÃO ORIGINAL

O modelo original de Mensagens Ativas, implementado em hardware, foi apresentado em (EICKEN et al., 1992), podendo funcionar de duas formas: por interrupção ou por *polling*, como apresenta a Figura 14, onde o lado *(a)* refere-se a estratégia de interrupção e o lado *(b)*, a estratégia de polling.

Se estiver sendo utilizado o modelo de interrupções, a interface de rede interrompe

a *thread* em execução, avisando da chegada de uma mensagem (1). Esta interrupção é então tratada (2) e, em seguida, o descritor da mensagem é extraído da interface de rede (3). Logo após é iniciado o tratamento da mensagem (4) e os dados são retirados da rede (5) e armazenados em uma área de dados.

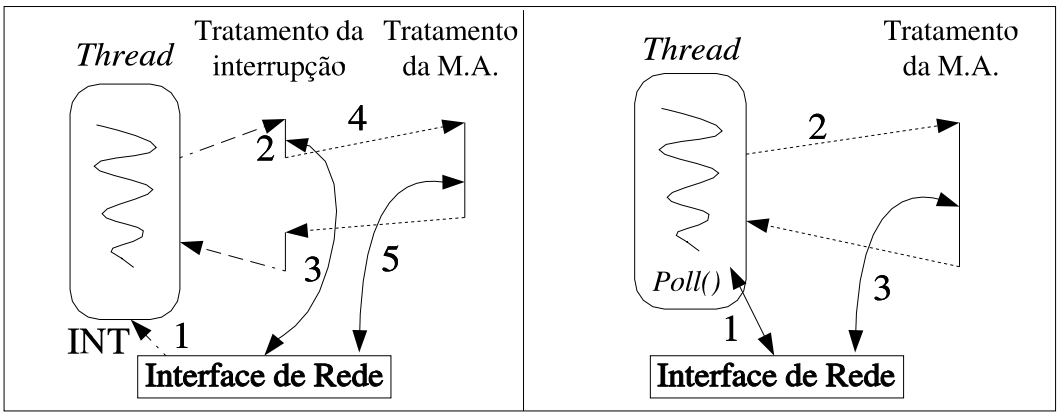


Figura 14: Esquema de funcionamento do mecanismo de Mensagens Ativas. (a) Interrupção. (b) *Polling*.

Quando se utiliza o modelo de *polling*, a *thread* em execução consulta, periodicamente, a interface de rede para verificar se chegaram novas mensagens. Se, nesse processo de *polling*, ocorrer a chegada de uma mensagem, seu descritor será retirado da rede pela *thread* (1) e será disparado o tratamento dessa mensagem ativa (2). Por fim, os dados necessários à execução da tarefa são retirados da rede (3).

No entanto, este modelo apresenta um problema ocasionado pela presença de uma única *thread* no receptor. Como esta *thread* é responsável por receber e tratar as Mensagens Ativas, o serviço executado pela Mensagem Ativa não poderá realizar operações de sincronização, o que poderia ocasionar uma situação de *deadlock*, uma vez que não há nenhum outro fluxo de execução ativo realizando atividades da aplicação.

## 4.2 IMPLEMENTAÇÕES ALTERNATIVAS

O problema do modelo original de Mensagens Ativas mencionado anteriormente, pode não ser interessante quando se deseja obter programas concorrentes de cunho genérico pela restrição imposta ao uso de sincronizações.

Buscando solucionar este problema, foram desenvolvidas as seguintes variações do modelo original: *UpCall*, que apresenta uma *thread* especializada para as mensagens, e *PopUp*, que trabalha com várias *threads*.

### 4.2.1 UpCall

Nesta variação existe uma *thread* especialista responsável pelo tratamento de todas as mensagens recebidas. Esta *thread* é chamada de *daemon* de comunicação. A existência do *daemon* evita que a *thread* que está executando, pare seu processamento para tratar a mensagem que chegou (CARISSIMI, 1999). A Figura 15 apresenta o esquema de funcionamento do mecanismo de *UpCall* tanto para a política de interrupção (a) quanto para a de *polling* (b).

Caso esteja sendo utilizada a política de interrupção, a chegada de uma mensagem gera uma interrupção (1), que provoca o acionamento do escalonador. A *thread* em execução é preemptada (2) e o escalonador transfere a execução para o *daemon* (3). A mensagem é, então, extraída da interface de rede (4) e tem sua execução iniciada (5).

Se, por outro lado, estiver sendo utilizada a política de *polling*, o *daemon* será o responsável por verificar periodicamente a interface de rede a fim de detectar a chegada de mensagens. Para que possa realizar tal verificação, o *daemon* deve receber do escalonador o direito de executar (1). Assim que detectar a chegada de uma mensagem na interface de rede, o *daemon* a retira da interface (2) e o devido tratamento é dado à mensagem (3).

Ao contrário da implementação original, nesta variação as *threads* podem ser sincronizadas, já que são independentes. No entanto, a presença de uma única *thread* res-

ponsável por todas as mensagens do nó, fará com que o sistema perca reatividade no atendimento às mensagens recebidas.

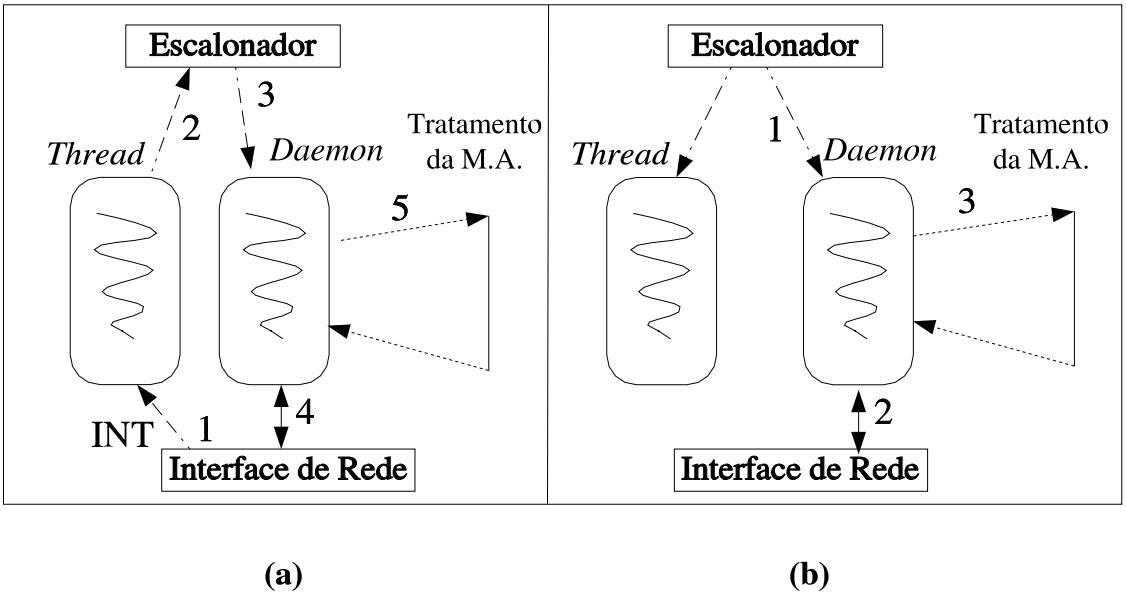


Figura 15: Esquema de funcionamento do mecanismo de *UpCall*, por interrupção (a) e *polling* (b).

### 4.2.2 PopUp

Nesta segunda variação de Mensagens Ativas, ao invés de existir um *daemon* de comunicação como no caso de *UpCall*, ocorre a criação de uma nova *thread* para tratar cada mensagem recebida. Isto evita que a *thread* em execução necessite interromper seu trabalho para retirar os dados da mensagem da interface de rede (CARISSIMI, 1999) e aumenta a reatividade da aplicação, já que os dados são consumidos assim que são entregues. Como nos demais casos, este mecanismo também pode funcionar por interrupção (lado *a* da Figura 16) ou por *polling* (lado *b* da Figura 16).

Quando a estratégia de interrupção está sendo utilizada, a chegada de uma nova mensagem gera uma interrupção (1), que passará a ser tratada pela *thread* em execução (2). O cabeçalho da mensagem é retirado da interface de rede (3) e uma nova *thread* é criada para tratá-lo (4), com a responsabilidade de retirar, da interface de rede, os dados que chegaram como parâmetros da mensagem (5).

Ao se utilizar a estratégia de *polling*, a *thread* em execução verifica, periodicamente, a interface de rede para detectar a presença de novas mensagens (1). Com a chegada de uma nova mensagem, a *thread* em execução cria uma nova *thread* para tratamento da mensagem (2). Esta nova *thread* será responsável por retirar a mensagem da interface de rede (3) e tratá-la da forma necessária.

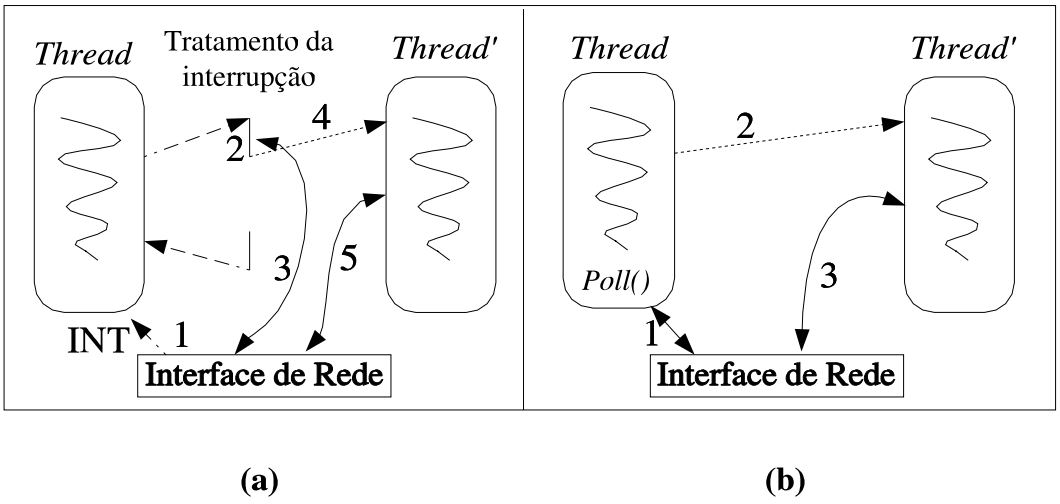


Figura 16: Esquema de funcionamento do mecanismo de *PopUp*. (a) Interrupção. (b) *Polling*.

Esta variação de Mensagens Ativas soluciona os problemas das duas implementações anteriores. Diferentemente do modelo original, permite a realização de sincronização pelas *threads* sem o risco de *deadlock*, pois todas as *threads* são independentes. Em oposição a *UpCall*, evita que o sistema perca reatividade, pois cada mensagem entregue é tratada por uma nova *thread*, o que diminui o tempo entre a produção e o consumo dos dados. Porém, a cada criação de *thread* custos adicionais são introduzidos no total da aplicação.

## 4.3 TRABALHOS RELACIONADOS

Nesta seção são apresentados alguns ambientes de programação paralela e distribuída que empregam o mecanismo de Mensagens Ativas para comunicação.

### 4.3.1 Split-C

Split-C (CULLER et al., 1993) é uma extensão paralela da linguagem de programação C, que suporta o acesso a um espaço de endereçamento global em multiprocessadores com memória distribuída. O modelo de programação seguido é o *Single Program Multiple Data* (SPMD), onde todos os processadores iniciam a execução em um mesmo ponto de um código comum e no qual cada processador, identificado por um número único, pode seguir um fluxo de controle distinto, sincronizando sua execução com os demais processadores em pontos pré-definidos, com a utilização de barreiras.

Uma das extensões da linguagem C implementadas em Split-C, permite que qualquer processador acesse qualquer localização em um espaço de endereçamento global e, ao mesmo tempo, cada processador possua uma região específica do espaço de endereçamento global. A região local a cada processador pode ser acessada através de ponteiros locais padrão da linguagem C, enquanto o espaço global pode ser acessado por ponteiros globais. Sendo que o custo de acesso a ponteiros globais é maior do que a ponteiros locais.

Em Split-C as Mensagens Ativas são um recurso de implementação do próprio ambiente, não sendo oferecido acesso a estes recursos ao programador. A camada de Mensagens Ativas utilizada tem seus serviços disponibilizados pela interface SPMA (CHANG et al., 1996), construída para manipular diretamente um hardware de comunicação.

### 4.3.2 Chant

Chant (HAINES; CRONK; MEHROTRA, 1994) é um ambiente para programação paralela desenvolvido para sistemas com memória distribuída, baseado em *threads* comunicantes (*talking threads*). O termo *talking threads* foi introduzido para representar a noção de duas *threads* em comunicação direta, através de primitivas *send()* e *receive()*, independente de estarem ou não no mesmo espaço de endereçamento.

Chant estende a interface e as funcionalidades do padrão POSIX para *threads* (Pthreads) (IEEE, 1994) para uma categoria de objetos computacionais chamados

*chanters*, que correspondem às *threads* com identificadores globais únicos dentro da máquina paralela e que podem comunicar-se remotamente com outras *threads*. Para criar uma *thread chanter*, local ou remota, a primitiva *pthread_create()* de Pthreads foi estendida para *pthread_chanter_create()*, que retorna o identificador global único da *thread* criada.

Duas *threads chanters* comunicam-se com base no padrão MPI, utilizando primitivas do tipo *send* e *receive*. Tais primitivas são utilizadas para passagem de parâmetros e retorno de resultados, possibilitando, também, o envio síncrono e o recebimento síncrono e assíncrono de mensagens. Para situações onde a comunicação entre *chanters* não pode ser realizada através da troca de mensagens ponto-a-ponto, Chant provê um mecanismo de *Remote Service Requests* (RSR), que permite que funções sejam executadas dentro de um contexto remoto.

O suporte à comunicação em Chant oferece a possibilidade de realizar troca de dados ponto-a-ponto entre *threads* e a invocação remota de *threads*. Este segundo serviço é viabilizado pela introdução no ambiente de execução de uma *thread* especializada (*server thread*), que aguarda o recebimento de mensagens solicitando a execução de um serviço. Ao receber uma mensagem, a *server thread* passa a executar o serviço solicitado, possuindo prioridade de execução mais alta em relação às demais *threads*. Em Chant, os serviços de Mensagens Ativas estão disponíveis ao programador, mas deve-se ter cuidado para que os serviços solicitados não sejam bloqueantes, de forma a não ocorrerem situações de *deadlock*.

### 4.3.3 Nexus

Nexus (FOSTER; KESSELMAN; TUECKE, 1994, 1996) consiste em uma camada de suporte à execução de aplicações paralelas irregulares, que busca possibilitar que aglomerados de computadores geograficamente distribuídos e heterogêneos sejam utilizados como um único aglomerado global (conceito de *Grid Computing*).

A interface do Nexus está organizada sobre cinco abstrações: nós, contextos, *threads*, ponteiros globais e requisições de serviços remotos, conforme ilustrado pela Figura 17.

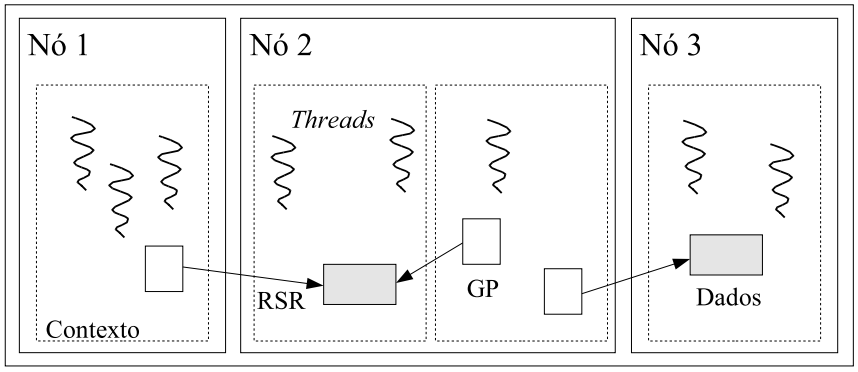


Figura 17: Abstrações do Nexus.

Assim, uma aplicação consiste em um conjunto de *threads* executando dentro de espaços de endereçamento denominados *contextos*, que são mapeados para um conjunto de *nós*, que representam os recursos físicos de processamento. Dentro de um mesmo contexto, as *threads* comunicam-se por meio da memória compartilhada; estando em contextos diferentes, as *threads* podem disparar *requisições de serviços remotos* através de *ponteiros globais*, que provocam a execução de funções dentro de outros contextos.

Um espaço de endereçamento global é provido pelo Nexus através dos ponteiros globais. Um ponteiro global é uma estrutura de dados que contém uma referência direta a um objeto dentro de um contexto. Cada ponteiro global é formado por um par (*startpoint, endpoint*) para comunicação ponto-a-ponto, porém pode ser estendido para suportar comunicação *multicast* através da associação de um *startpoint* a vários *endpoints*.

Para que *threads* pertencentes a contextos diferentes possam comunicar-se, são utilizadas as requisições de serviços remotos (RSR). Uma RSR resulta na execução de uma função especial, chamada *handler*, no contexto referenciado pelo ponteiro global. Esta função é invocada assincronamente dentro do contexto; nenhuma ação, tal como a execução de um *receive*, precisa ser tomada para que a função seja executada. Esta execução

pode ocorrer de dois modos: em uma nova *thread* ou em uma *thread* pré-alocada, caso a função não seja bloqueante. Cabe salientar que para uma RSR não existe confirmação ou retorno de resultados e a *thread* chamadora não permanece bloqueada.

### 4.3.4 Athapascan-0

O núcleo executivo de Athapascan-0 (GINZBURG, 1997; BRIAT; GINZBURG; PASIN, 1998) combina multiprogramação e comunicação, fornecendo um conjunto de primitivas para a realização de troca de mensagens. Athapascan-0 possibilita a criação de *threads* local e remotamente. No caso de serem criadas localmente, as *threads* comunicam-se por meio de memória compartilhada e o núcleo oferece mecanismos de sincronização (como *mutexes*, semáforos e variáveis de condição) para garantir o acesso concorrente aos dados compartilhados. No caso de *threads* remotas, sua criação ocorre através de uma chamada a um procedimento remoto assíncrono, correspondendo à execução de um serviço.

Além de permitir a criação remota de *threads*, as quais não possuem nenhuma restrição no que diz respeito ao uso de primitivas de sincronização, Athapascan-0 introduz a possibilidade de uso de "mensagens urgentes". Mensagens urgentes são tratadas por um *daemon* especializado, o qual reage ao recebimento de uma mensagem pela ativação imediata do serviço solicitado - estes serviços não devem realizar nenhuma operação de sincronização.

## 4.4 FILA DE EXECUÇÃO

Os três modelos de Mensagens Ativas apresentados nas duas primeiras seções deste capítulo não são satisfatórios quando se trata de aplicações altamente paralelas. No modelo original, uma operação de sincronização realizada pela mensagem que está sendo processada poderá ocasionar uma situação de *deadlock*, uma vez que o fluxo de execução será bloqueado. A variação *UpCall* soluciona este problema incluindo uma *thread* es-

pecializada em tratar mensagens. No entanto, caso a mensagem em execução bloqueie, o sistema perderá reatividade ao atendimento de novas solicitações. Os problemas de reatividade e *deadlock* são resolvidos na variação *PopUp* pela criação de uma nova *thread* para tratar cada mensagem entregue. Porém, cada criação de *thread* adiciona custos no total da aplicação.

Uma terceira variação do mecanismo de Mensagens Ativas foi proposto em (ROLOFF; CARISSIMI; CAVALHEIRO, 2004). Este modelo pode ser visto como uma estratégia híbrida, que reúne características de *UpCall* e *PopUp*. Com isso, é possível executar vários serviços simultaneamente, sem a necessidade de criação de novas *threads* e podendo existir um *daemon* especializado em retirar as mensagens da interface de rede. Esta variação foi concebida para ser inserida no modelo de execução do ambiente Anahy (CAVALHEIRO; DALL'AGNOL; VILLA REAL, 2003), utilizando-se de processadores virtuais e listas de tarefas.

Na Figura 18 pode ser observada uma visão esquemática deste mecanismo. Neste caso, existe um número $n$ de *threads* em cada nó, executando as tarefas definidas pelo programa em execução. Ao ser criada, uma tarefa é armazenada em uma lista de tarefas (fila de execução). Quando uma *thread* encerra a execução de uma tarefa, ela verifica se existe alguma outra na lista de tarefas à espera de ser executada. Caso exista, a *thread* toma para si a tarefa e passa a executá-la. Caso a fila esteja vazia, a *thread* permanece bloqueada até que ocorra a inclusão de uma nova tarefa na lista. Toda mensagem, ao ser recebida, é armazenada na lista de tarefas para futuro tratamento.

Nesta variação, o *daemon* de comunicação consulta periodicamente a rede para verificar a chegada de uma nova mensagem (1). Se nesse processo for detectada a entrega de uma mensagem, o *daemon* a retira da interface de rede, a insere na lista de tarefas do nó (2) e volta a repetir o processo. Quando uma das *threads* do nó encerrar o processamento de uma tarefa, ela consulta a lista de tarefas prontas para serem calculadas (3). Caso a lista não esteja vazia, a *thread* retira uma tarefa da lista e a processa. Caso contrário,

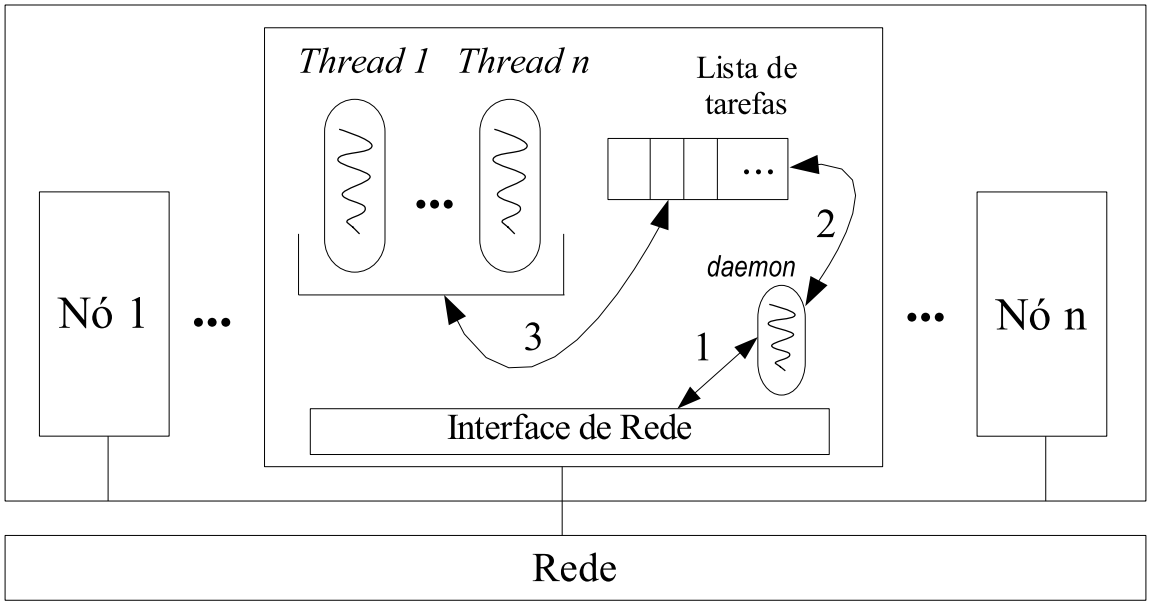


Figura 18: Esquema de funcionamento do mecanismo de *fila de execução*.

bloqueia até que uma tarefa pronta para ser calculada seja inserida na lista.

## 4.5 COMENTÁRIOS FINAIS

O mecanismo de Mensagens Ativas apresentado neste capítulo mostra-se como uma boa alternativa para a comunicação em aglomerados, permitindo que sejam atingidos bons índices de desempenho. No entanto, os modelos existentes possuem características que influenciam tanto no desenvolvimento das aplicações quanto no desempenho final. A Tab. 5 apresenta um resumo com as principais características de cada modelo de Mensagens Ativas.

Tabela 5: Principais características das diferentes variações de Mensagens Ativas.

| | **Modelo Original** | **PopUp** | **UpCall** | **Fila de Execução** |
|---|---|---|---|---|
| **Serviços Bloqueantes** | Não | Sim | Sim | Sim |
| **Reatividade** | Tamanho do serviço | Velocidade de criação da *thread* | Tamanho do serviço | Número de *threads* de serviço |
| **Implementação** | HW | HW/SW | HW/SW | SW |

No modelo original, pode ocorrer uma situação de *deadlock* caso a mensagem que

está sendo processada realize uma operação de sincronização. Na variação *UpCall* é introduzida uma *thread* especializada em tratar mensagens para solucionar o problema de *deadlock*, mas caso a mensagem em execução bloqueie, o sistema perderá reatividade. A estratégia de *PopUp* cria uma nova *thread* para tratar cada mensagem entregue, solucionando os problemas de perda de reatividade e *deadlock*. Porém, a cada nova *thread* criada, um custo é adicionado no total da aplicação.

Buscando solucionar os problemas das variações existentes, o modelo de fila de execução proposto para ser inserido em Anahy, introduz um *daemon* de comunicação para tratamento de mensagens e um número $n$ de *threads* (processadores virtuais) para execução de tarefas definidas pelo programa em execução. Com isso, é possível que várias tarefas sejam executadas simultaneamente, mantendo os processadores ocupados com cálculo útil o maior tempo possível.

Apesar de mostrarem-se eficientes para a comunicação em aglomerados, as Mensagens Ativas apresentam-se unicamente assíncronas. Isso faz com que exista a necessidade de mecanismos de controle externos que permitam uma correta evolução do programa em execução. Em geral, este controle fica a cargo do programador da aplicação e não do mecanismo de Mensagens Ativas.

Tabela 6: Principais características dos diferentes ambientes que empregam Mensagens Ativas.

| | **Athapascan-0** | **Nexus** | **Split-C** | **Chant** |
|---|---|---|---|---|
| **Implementação** | SW | SW | HW | SW |
| **Suporte de Programação** | MPI | Proprietário | Próprio | MPI |
| **Interface** | *Threads* remotas | Serviços remotos | Recurso do ambiente | *Threads* remotas |

Neste capítulo também foram apresentados alguns ambientes de programação paralela e distribuída, encontrados na literatura, que empregam o mecanismo de Mensagens Ativas. As principais características de cada ambiente estão destacadas na Tab. 6. Observa-se que, à exceção de Split-C, a implementação de Mensagens Ativas se dá em software, sendo empregadas técnicas de multiprogramação leve para possibilitar a con-

corrência entre operações de cálculo e comunicação. Também cabe citar que Athapascan-0 conta com uma interface de programação aplicativa (GALLILÉE et al., 1998) que incorpora ao ambiente um mecanismo de escalonamento capaz de contornar os problemas advindos do assincronismo das Mensagens Ativas.

# 5 ANAHY

Com a popularização dos aglomerados de computadores nos últimos anos[1], um novo conjunto de ferramentas vem sendo desenvolvido para exploração de processamento de alto desempenho neste tipo de arquitetura. No entanto, aglomerados são constituídos de nós com capacidades distintas de processamento, característica, em geral, não abordada por ferramentas clássicas de programação, como chamada de procedimentos remotos (RPC), *threads* e comunicação entre processos. Tais ferramentas, embora permitam a exploração de forma especializada de um determinado recurso de processamento, não tendo sido desenvolvidas especialmente para aglomerados, não possibilitam obter o máximo de desempenho do conjunto de recursos disponíveis.

Nexte contexto, Anahy é um ambiente de programação e execução de aplicações paralelas em aglomerados de computadores, onde cada nó pode vir a ser um multiprocessador com memória compartilhada. O objetivo de Anahy é oferecer recursos para a exploração do processamento de alto desempenho, provendo tanto uma interface de programação concorrente de alto nível, como um núcleo executivo. Com isso, o programador é capaz de descrever a concorrência de sua aplicação independentemente dos recursos computacionais disponíveis na arquitetura.

Neste capítulo, que está baseado em (CAVALHEIRO; DALL'AGNOL; VILLA REAL, 2003), é descrito o ambiente Anahy. Na Seção 5.1 é apresentada a arquitetura virtual de Anahy. A Seção 5.2 define os conceitos de tarefa e grafo no ambiente Anahy. A Seção 5.3 apresenta a interface de programação do ambiente, destacando os serviços oferecidos e a

[1]Dados obtidos em http://www.top500.org

sintaxe utilizada. Na Seção 5.4 é apresentado o núcleo executivo de Anahy, com destaque para a descrição do algoritmo de escalonamento empregado pelo núcleo. A Seção 5.5 identifica os serviços de escalonamento existentes no ambiente. Para encerrar o capítulo, são feitos alguns comentários finais na Seção 5.6.

## 5.1 ARQUITETURA VIRTUAL

Embora a arquitetura real considerada seja um aglomerado de computadores, a visão que o programador tem é de uma arquitetura *virtual* multiprocessada com memória compartilhada, conforme ilustrado na Figura 19.

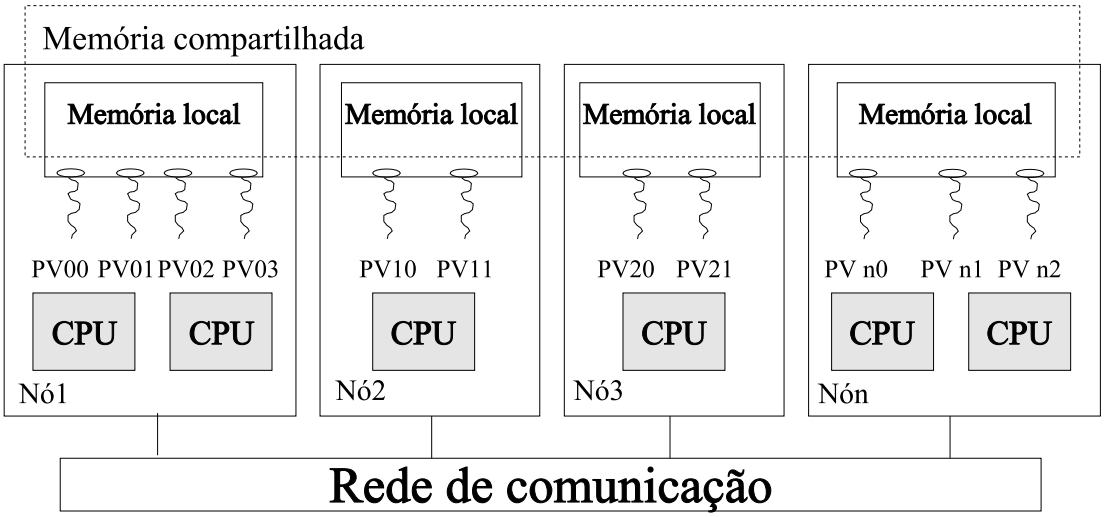


Figura 19: Modelos real e virtual da arquitetura de suporte à Anahy.

Nesta figura pode-se observar que a arquitetura real é formada por um conjunto de nós de processamento, dotados de memória local e de unidades de processamento (CPUs). Sobre esta arquitetura real existe uma arquitetura virtual composta por um conjunto de processadores virtuais (PVs) alocados sobre os nós e por uma memória compartilhada pelos PVs. Cada PV executa seqüencialmente as atividades atribuídas a ele, sem interrupções durante a execução. Quando ocioso, o PV pode ser despertado para executar uma atividade apta para execução. Existindo a necessidade de comunicação entre os PVs, esta será realizada através da memória compartilhada, que é acessada pelas instruções de sin-

cronização oferecidas pela arquitetura virtual. Estas instruções de sincronização permitem o controle ao acesso aos dados compartilhados e, portanto, o controle de concorrência das atividades.

## 5.2 TAREFA E GRAFO

Embora a comunicação de dados entre as instruções esteja resolvida através dos recursos da arquitetura virtual, não há garantia de que o programa executará corretamente. Para isso, deve existir alguma forma de sincronização entre as instruções para evitar que uma instrução comece a executar antes de ter disponível, na memória compartilhada, os dados de que necessita como entrada. No paradigma de execução imperativo seqüencial, onde as instruções são executadas seqüencialmente, a sincronização está implicitamente garantida, não sendo iniciada a execução de uma instrução $\mathcal{S}_i$ antes que a instrução $\mathcal{S}_{i-1}$ tenha sido completada. Essa dependência é representada por $\mathcal{S}_{i-1} \prec \mathcal{S}_i$.

Assim, existe, na execução de um programa seqüencial imperativo $\mathcal{P}$, uma ordem específica para ativação das instruções dado um conjunto $\mathcal{X}$ de dados de entrada:

$$\mathcal{E}(\mathcal{P}, \mathcal{X}) = \mathcal{S}_1 \prec \mathcal{S}_2 \prec \mathcal{S}_3 \prec ... \prec \mathcal{S}_s$$

De maneira semelhante, programas concorrentes necessitam transformar dados recebidos como entrada para produzirem resultados. Porém, neste tipo de programação, o trabalho total da aplicação é dividido em atividades concorrentes, neste trabalho sendo denominadas *tarefas*. Estas tarefas necessitam trocar dados entre si para garantir a evolução do programa. São necessários, portanto, mecanismos de comunicação de dados e sincronização de tarefas para garantir a correta execução do programa. Os mecanismos de comunicação permitem que dados produzidos por uma tarefa sejam disponibilizados para outra tarefa. Com os mecanismos de sincronização, uma tarefa pode verificar a disponibilidade de um dado ou avisar a outra que um dado já está disponível.

A execução $\mathcal{E}(\mathcal{P}_c\ , \mathcal{X})$ de um programa concorrente pode, então, ser repre-

sentada por uma coleção de tarefas $\mathcal{T} = (\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_n)$ e um conjunto de dados $\mathcal{X} = (x_1, x_2, ..., x_n)$, descritos por um grafo de dependências $\mathcal{G} = (\mathcal{V},E)$, onde os nós $\mathcal{V} = \mathcal{T} \cup \mathcal{X}$ são representados pelo conjunto de tarefas e de dados manipulados e os acessos são representados pelas arestas $E = (\mathcal{T}$ x $\mathcal{X}) \cup (\mathcal{X}$ x $\mathcal{T})$. Neste caso, $(\mathcal{T}_i, x_j)$ indica que a tarefa $\mathcal{T}_i$ produz o dado $x_j$ e $(x_i, \mathcal{T}_j)$ indica que o dado $x_i$ é necessário para executar a tarefa $\mathcal{T}_j$.

A partir destas considerações, pode-se definir um programa em execução em Anahy como um conjunto de tarefas, cada uma delimitando uma seqüência de instruções elementares e definindo dois conjuntos de dados: (*i*) os dados necessários ao início de sua execução e (*ii*) os dados produzidos como resultado de sua execução. Dessa forma, a ordem de execução das tarefas é definida pela disponibilidade de seus dados de entrada e, ao terminar, uma tarefa produz um resultado que poderá desencadear a execução de uma outra tarefa.

## 5.3 INTERFACE DE PROGRAMAÇÃO

As dificuldades enfrentadas pelo programador para o desenvolvimento de um programa concorrente encontram-se na decomposição de sua aplicação em atividades concorrentes, no controle da ordem de execução destas atividades e no mapeamento dessas atividades sobre os recursos de processamento disponíveis. Além disso, há uma grande variedade de arquiteturas com características distintas e inúmeras ferramentas de programação. Dentre os diversos critérios considerados úteis em um modelo de programação concorrente (SKILLICORN, 1994) adotados por Anahy, está a capacidade de minimizar as dificuldades de gerenciamento de um grande número de fluxos de execução concorrente e das comunicações entre eles.

### 5.3.1 Serviços Oferecidos

A interface de programação de Anahy disponibiliza ao programador serviços para explorar o paralelismo de uma arquitetura multiprocessada dotada de uma área de memória compartilhada. Neste caso, as tarefas concorrentes da aplicação são sincronizadas através de trocas implícitas de dados entre elas. Os serviços oferecidos podem ser representados por operações *fork/join*, semelhante ao modelo de multiprogramação leve no que se refere a criação e sincronização de fluxos de execução.

A invocação da operação *fork* cria logicamente um novo fluxo de execução, retornando um identificador ao fluxo criado. Este novo fluxo é responsável pela execução de uma função F, que deve ser identificada e ter passados os parâmetros necessários a sua execução no momento em que o *fork* é invocado. Embora o programador não tenha noção do momento em que o fluxo criado passará, efetivamente, a executar, sabe-se que, ao seu término, um resultado será produzido.

A operação *join*, por sua vez, é responsável pela sincronização com o término de um fluxo de execução. O identificador do fluxo retornado pela operação *fork* é, agora, utilizado para indicar qual fluxo deseja-se sincronizar. Quando um fluxo necessita de resultados produzidos por outro, a operação *join* permite que ele permaneça bloqueado, aguardando o término do outro fluxo e a disponibilização dos resultados.

### 5.3.2 Sintaxe Utilizada

O desenvolvimento de Anahy está sendo realizado de forma a permitir compatibilidade com o padrão POSIX para *threads*, sendo as primitivas e estruturas oferecidas, um subconjunto dos serviços oferecidos por este padrão. Atualmente, a interface de serviços está disponível para programas implementados nas linguagens C/C++.

Um fluxo de execução tem seu corpo definido como uma função C convencional, conforme apresentado no exemplo abaixo, no qual a função *func* pode ser instanciada em um fluxo de execução próprio.

```
void *func (void *in){
        /*código da função*/
        return out;
}
```

A localização dos dados de entrada da função é indicada pelo argumento *in* que corresponde ao endereço na memória compartilhada de onde estão os dados. Já, *return out* serve para indicar que *func* deve retornar, ao seu término, o endereço de um dado na memória compartilhada, referente ao armazenamento do resultado produzido pela tarefa.

As operações *fork* e *join* correspondem, respectivamente, as operações *pthread_create* e *pthread_join* do padrão POSIX; a primeira para criar uma *thread* e a segunda para aguardar pelo término de uma *thread*. Abaixo são exemplificadas as sintaxes de *athread_create* e *athread_join*, correspondentes, respectivamente, de *pthread_create* e *pthread_join*.

```
int athread_create (athread_t *th, athread_attr_t *atrib,
                    void*(*func)(void *), void *in);
int athread_join(athread_t th, void **res);
```

De forma semelhante a *pthread_create* do padrão POSIX, em Anahy a função *athread_create* cria um novo fluxo de execução para a função *func*, que terá como entrada o argumento *in*. Este novo fluxo poderá, quando necessário, ser referenciado pelo identificador único indicado em *th*. O outro argumento passado para *athread_create*, *atrib*, é utilizado para definir atributos através dos quais o programador informa características referentes à execução do novo fluxo de execução (como, por exemplo, espaço de memória necessário para execução).

## 5.4 NÚCLEO EXECUTIVO

Com Anahy um programa concorrente pode ser executado tanto em aglomerados de computadores como sobre arquiteturas multiprocessadas. Ao utilizar Anahy como ambiente de programação/execução, o programador necessita, apenas, codificar sua aplicação, sem especificar o mapeamento das tarefas nos processadores. O núcleo

executivo de Anahy foi concebido para suportar a introdução de mecanismos de balanceamento de carga. É importante destacar que, atualmente, Anahy encontra-se operacional somente sobre arquiteturas SMP [2].

### 5.4.1 Algoritmo de Escalonamento

O algoritmo de escalonamento manipula tarefas, que consistem em unidades de trabalho, definidas pelo programa em execução, compostas por uma seqüência de instruções, capazes de serem executadas em tempo finito. Dentre as instruções que podem ser executadas por uma tarefa, estão as operações de criação de novas tarefas. O término de uma tarefa é marcado pela execução de uma operação de sincronização com uma outra tarefa.

Quatro listas de tarefas são gerenciadas pelo algoritmo de escalonamento:

- A primeira lista contendo as tarefas **prontas**, ou seja, aquelas que encontram-se aptas a serem lançadas;
- A segunda, com as tarefas **terminadas** cujos resultados ainda não foram solicitados, isto é, aquelas tarefas sobre as quais ainda não foi realizada a operação de *join*;
- A terceira contém as tarefas **bloqueadas**;
- Por fim, a quarta lista contém as tarefas que encontram-se **executando**.

Considerando-se uma arquitetura monoprocessada, o algoritmo de escalonamento trabalha da seguinte maneira: o processador busca a primeira tarefa, $\mathcal{T}_1$, da lista de tarefas prontas e inicia sua execução. As instruções elementares de $\mathcal{T}_1$ são executadas seqüencialmente e, ao realizar uma operação de sincronização, $\mathcal{T}_1$ termina sua execução. Caso tenha sido executada uma operação de *fork*, duas tarefas são criadas, $\mathcal{T}_2$ e $\mathcal{T}_3$, onde $\mathcal{T}_2$ é a tarefa criada pelo *fork* para execução de uma nova função e $\mathcal{T}_3$, a continuação de $\mathcal{T}_1$. Ambas tarefas estão prontas para executar, no entanto, $\mathcal{T}_2$ é colocada na lista de

[2] A versão protótipo de Anahy para aglomerados está em curso de desenvolvimento no momento da redação do presente trabalho. Parte dos resultados desta dissertação estão sendo utilizados na obtenção desta versão.

prontas e $\mathcal{T}_3$ é lançada, privilegiando uma execução em profundidade do grafo. Caso a operação realizada seja um *join*, por exemplo de $\mathcal{T}_2$ com $\mathcal{T}_3$, uma nova tarefa $\mathcal{T}_4$ é criada, tendo como entrada os dados produzidos por $\mathcal{T}_2$ e $\mathcal{T}_3$. Como $\mathcal{T}_3$ não foi, ainda, executada, $\mathcal{T}_4$ é colocada na lista de tarefas bloqueadas e $\mathcal{T}_3$ é lançada para execução. Com o término de $\mathcal{T}_3$, $\mathcal{T}_4$ é colocada na lista de tarefas prontas, vindo a receber o processador.

Em uma arquitetura paralela, a presença de dois ou mais processadores executando o mesmo algoritmo descrito no parágrafo anterior, possibilita a execução simultânea de duas ou mais tarefas. Então, quando uma tarefa $\mathcal{T}_i$ executar um *join* com uma tarefa $\mathcal{T}_j$, pode ocorrer ou de $\mathcal{T}_j$ já ter terminado ou ainda estar executando. Caso $\mathcal{T}_j$ já tenha terminado, ela é retirada da lista de tarefas terminadas e os dados por ela produzidos permitem que o processador prossiga com a execução de $\mathcal{T}_{i+1}$. Caso $\mathcal{T}_j$ esteja em execução, $\mathcal{T}_{i+1}$ permanece na lista de tarefas bloqueadas e uma nova tarefa é buscada pelo processador na lista de tarefas prontas para execução. Quando $\mathcal{T}_j$ terminar e produzir os dados necessários à execução de $\mathcal{T}_{i+1}$, esta é desbloqueada e passa à lista de tarefas prontas.

A descrição do algoritmo de escalonamento deixa evidente que uma tarefa não pode ser iniciada antes que todas as tarefas que produzam dados necessários a sua execução tenham sido concluídas. Em um programa existem, então, relações entre suas tarefas, representadas por $\mathcal{T}_j \prec \mathcal{T}_i$, de maneira que se pode identificar caminhos de dependência de dados entre as tarefas. O maior caminho de transformação de dados no programa $\mathcal{P}_c$ tendo como entrada $\mathcal{X}$, é definido pela seqüência:

$$\mathcal{E}(\mathcal{P}_c, \mathcal{X}) = \mathcal{T}_1 \ ... \ \prec \mathcal{T}_{k-2} \prec \mathcal{T}_{k-1} \prec \mathcal{T}_k$$

Essa seqüência é denominada caminho crítico (GRAHAM; KNUTH; PATASHNIK, 1989) e representa a carga computacional do problema que não pode ser paralelizada, devido às dependências existentes entre as tarefas. No escalonador de Anahy busca-se, então, evitar todo o custo adicional à execução de tarefas e, durante toda a execução do

programa, ao menos um dos processadores deve estar ativo executando uma tarefa do caminho crítico.

### 5.4.2 Escalonamento Multinível

O núcleo executivo de Anahy tem seu escalonamento organizado em três níveis:

- O primeiro é realizado pelo sistema operacional e consiste no mapeamento dos fluxos de execução associados aos PVs aos recursos físicos de processamento;
- O segundo nível de escalonamento refere-se a alocação das tarefas aos PVs, sendo considerada a ordem de execução das tarefas e gerenciadas as listas de tarefas de forma global aos PVs. As tarefas são encapsuladas em *threads* Anahy. Uma *thread* Anahy não realiza nenhuma operação de sincronização, exceto criação de novas *threads* e obtenção de resultados de retorno de outras *threads*;
- O terceiro nível é responsável pela distribuição da carga computacional gerada entre os nós que compõem a arquitetura utilizada, podendo ser considerados fatores como o custo computacional das tarefas e a localidade física dos dados.

## 5.5 SERVIÇOS DE ESCALONAMENTO

Os serviços de escalonamento de Anahy têm a função de acessar a lista de *threads* e a memória compartilhada. O controle destes acessos pelo ambiente é uma das principais vantagens de Anahy, já que a responsabilidade de executar tarefas e sincronizar as trocas de dados deixa de ser do programador e passa a ser do ambiente. Com isso, a ordem de execução das tarefas é garantida pelo próprio ambiente, que realiza a sincronização entre tarefas controlando o acesso aos dados.

São quatro os serviços de escalonamento do ambiente Anahy: (1) criar uma *thread*, (2) manipulação de listas, (3) sincronização e (4) controlar o acesso à memória global.

### Criação de uma thread

Um novo fluxo de execução é criado quando uma operação *fork* é realizada. No momento da invocação desta operação, devem ser indicados a função a ser executada pelo novo fluxo e os parâmetros necessários a sua execução. Embora a *thread* criada esteja pronta para executar, o programador não tem garantias do momento em que ela será lançada para execução. O resultado da invocação de um serviço de criação de *thread* é a criação do descritor de *thread*.

### Manipulação de listas

As listas de tarefas de Anahy, citadas na Seção 5.4, são gerenciadas pelo algoritmo de escalonamento do ambiente, o qual realiza a manipulação de descritores de tarefas em função das relações de dependências de dados. Quando uma *thread* tem disponível os dados necessários a sua execução, ela é armazenada na lista de tarefas prontas. Ao ser criada, a *thread* já encontra-se pronta para execução. Uma decisão do escalonador passa a *thread* do estado de pronta para o de executando. Caso uma *thread* que esteja executando realize uma operação de sincronização com outra que esteja pronta, bloqueada ou executando, ela passa ao estado de bloqueada aguardando pelo término da *thread* sincronizada, passando novamente ao estado de pronta somente quando a condição de sincronização for satisfeita. Ao terminar sua execução, a *thread* passa ao estado de terminada.

### Sincronização

Três situações podem ocorrer quando uma operação *join* é realizada: a *thread* sincronizada está terminada, pronta ou executando. Supondo-se duas *threads* **A** e **B**, onde **A** está no estado executando. Se **A** realiza um *join* com **B** e **B** encontra-se na lista de tarefas terminadas, os dados produzidos por **B** são passados para **A** e **A** segue sua execução. Caso **B** esteja no estado de pronta, o código de **B** passa a ser executado sobre o mesmo fluxo de execução de **A**, que passa ao estado de bloqueada. Quando **B** termina,

os dados produzidos por ela são passados para **A**, que segue sua execução de onde havia parado. Na terceira situação, **A** realiza um *join* com **B**, que está executando. Neste caso, a execução de **A** é interrompida até que **B** termine.

**Acesso à memória global (passagem de parâmetros/leitura de resultados)**

O acesso propriamente dito é realizado pelas operações de acesso à memória local executadas pelas *threads*. A garantia do correto acesso é dada pela gestão dos estados das *threads*. Uma *thread* somente é executada quando estiver no estado de pronta, indicando que seus dados estão disponíveis na memória.

## 5.6 COMENTÁRIOS FINAIS

O ambiente de programação Anahy apresentado neste capítulo foi desenvolvido com o objetivo de oferecer recursos para a exploração do processamento de alto desempenho em arquiteturas com memória distribuída. Anahy provê uma interface de programação concorrente de alto nível e um núcleo executivo, permitindo que o programador descreva a concorrência de sua aplicação independentemente dos recursos computacionais disponíveis na arquitetura.

Outra facilidade introduzida por Anahy, consiste na garantia, pelo próprio ambiente, da ordem de execução das tarefas da aplicação. Com isso, a atribuição das tarefas aos nós reais de processamento, a criação de tarefas e sincronização entre elas e o controle de acesso aos dados compartilhados tornam-se atribuições de Anahy, não mais do programador.

A utilização de Anahy em aglomerados de computadores requer que estes serviços sejam também aplicáveis a um contexto distribuído. Isso, porém, deve ser realizado de maneira transparente ao usuário. A introdução de mecanismos de comunicação deve, assim, contemplar a implementação dos serviços de escalonamento definidos pelo ambiente.

# 6 MECANISMO DE SUPORTE À EXECUÇÃO DE APLICAÇÕES EM AGLOMERADOS

A disponibilização de Anahy em aglomerados de computadores exige que sejam introduzidos no ambiente serviços que possibilitem a troca de dados e tarefas entre os nós, de forma a viabilizar a implementação de seu suporte executivo em arquiteturas com memória distribuída. Além disso, deve-se buscar que toda tarefa seja executada o mais rápido possível, de forma a diminuir o impacto das comunicações no tempo total de execução da aplicação.

Neste contexto, este capítulo apresenta uma camada desenvolvida para suporte de Anahy em aglomerados de computadores, descrevendo os serviços detectados como necessários para integração desta camada com a parte operacional de Anahy. A Seção 6.1 descreve mais detalhadamente a camada desenvolvida. Na Seção 6.2 são descritas as primitivas implementadas e disponíveis ao usuário desta biblioteca. Na Seção 6.3 são analisados os serviços detectados como necessários para integração da camada desenvolvida com o ambiente Anahy. Para finalizar o capítulo, são feitos alguns comentários sobre o mecanismo implementado no contexto do projeto Anahy.

## 6.1 MECANISMO IMPLEMENTADO

No Capítulo 3 foram apresentados os níveis de concorrência a serem explorados em uma arquitetura para execução de aplicações paralelas. Um melhor aproveitamento dos recursos computacionais de um aglomerado de computadores é obtido quando se consegue

explorar tanto a concorrência intra-nó quanto a entre-nós. Com isso, é possível sobrepor parcialmente tempos gastos em comunicações com cálculo útil, de maneira a diminuir a ociosidade do processador.

O mecanismo implementado neste trabalho teve seu desenvolvimento baseado no princípio de atender a solicitação de execução de uma tarefa o mais rápido possível, procurando minimizar a adição de sobrecusto ao tempo total da aplicação. Para tanto, empregou-se *multithreading* para exploração da concorrência intra-nó (CAVALHEIRO; DALL'AGNOL; VILLA REAL, 2003; DALL'AGNOL et al., 2003) e Mensagens Ativas (EICKEN et al., 1992) para exploração da concorrência entre-nós. A ferramenta de *multithreading* é empregada para permitir que diversos fluxos de execução compartilhem os recursos de processamento internos a um nó. Enquanto que as Mensagens Ativas permitem que fluxos executando em diferentes nós do aglomerado troquem tarefas e dados.

Optou-se por Mensagens Ativas, pois elas introduzem pouca sobrecarga de comunicação no custo total da aplicação, por terem sido projetadas com o objetivo de permitirem alto desempenho em arquiteturas distribuídas. Também foram consideradas restrições ao uso de mecanismos baseados em trocas de mensagens associados à multiprogramação leve (conforme (FOSTER et al., 1993)).

No entanto, não foi identificada na literatura uma biblioteca de Mensagens Ativas que atendesse as necessidades do projeto Anahy. A biblioteca Split-C, primeira a empregar Mensagens Ativas, possui seu suporte implementado em hardware. Chant e Athapascan-0 realizam as comunicações com base no padrão MPI, o qual não apresenta resultados satisfatórios quando empregado em redes padrão Ethernet com a qual optamos por trabalhar. Por fim, no ambiente Nexus, uma RSR implica na ativação de um serviço dentro de uma *thread*, não permitindo a composição das tarefas Anahy. Por isso, decidiu-se construir o mecanismo apresentado neste trabalho.

Para atender ao princípio de execução rápida em aglomerados, duas estruturas foram introduzidas para oferecer suporte ao mecanismo de escalonamento: os proces-

sadores virtuais e o *daemon* de comunicação. Os processadores virtuais são *threads* especializadas na execução de uma tarefa específica e não realizam comunicações nem tratamento de mensagens. Toda troca de mensagens entre os nós do aglomerado e tratamento das mensagens enviadas ou recebidas por um nó é de responsabilidade do *daemon* de comunicação. Com isso, os processadores virtuais dedicam-se exclusivamente à realização de cálculo efetivo, possibilitando que o tempo gasto em comunicações seja sobreposto por computação (VALIANT, 1990).

Os esforços de implementação do mecanismo proposto neste trabalho, concentraram-se no desenvolvimento das funcionalidades deste módulo e na sua avaliação como parte operacional do suporte executivo de Anahy. Tal módulo fez uso do serviço de *sockets*, do sistema operacional GNU/Linux, para possibilitar a comunicação entre os nós do aglomerado. Optou-se pela utilização de *sockets* por este mecanismo ser implementado sobre o protocolo TCP, diretamente suportado pelo hardware disponível (rede padrão Ethernet), apresentando bons resultados de desempenho (BENITEZ; CAVALHEIRO, 2004). Além do critério de facilidade de uso, também foi considerado na elaboração das funcionalidades do módulo, o padrão POSIX para *threads*, por buscar-se a portabilidade de código, a exemplo da construção dos demais módulos de Anahy (CAVALHEIRO; GARZÃO; VILLA REAL, 2001).

Uma visão esquemática do mecanismo implementado pode ser observada na Figura 20. Toda mensagem entregue a um nó é retirada da interface de rede pelo *daemon* de comunicação (1). Se esta mensagem corresponde a uma tarefa que já pode ser executada, ela é colocada em uma lista de tarefas prontas (2); caso contrário, é colocada em uma lista de tarefas que aguardam para serem liberadas para execução. Quando um dos processadores virtuais do nó encerrar o processamento que estava realizando, ele consulta a lista de tarefas prontas para verificar a disponibilidade de tarefas para execução (3). Caso a lista não esteja vazia, ele retira a primeira tarefa e passa a executá-la; caso contrário, ele permanece bloqueado até que a inclusão de uma tarefa na lista o desbloqueie e ele passe a executá-la. Como as tarefas somente são atribuídas aos processadores virtuais quando

estão prontas para execução, existe garantia de que a evolução da aplicação será correta.

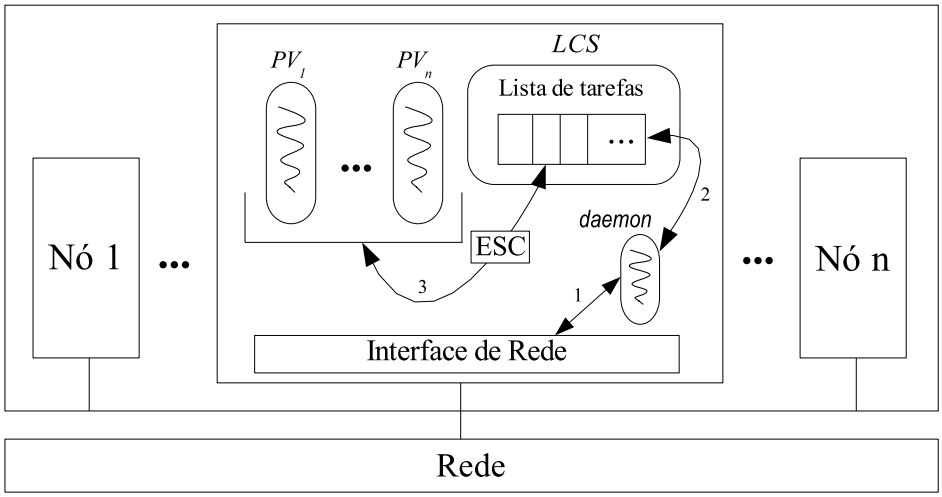


Figura 20: Visão esquemática do mecanismo de Mensagens Ativas implementado em Anahy.

No entanto, o mecanismo apenas garante a distribuição de tarefas e dados entre os nós do aglomerado. O controle de acesso aos dados compartilhados, como é o caso das listas de tarefas, deve ser realizado pelo programador (representado na Figura 20 pelo módulo LCS). Também é de responsabilidade do programador controlar a atribuição de tarefas aos processadores virtuais (na Figura 20, representado por ESC).

Na Figura 21, os dois blocos em destaque mostram a localização do presente trabalho no contexto do projeto Anahy. O bloco indicado por API engloba as aplicações desenvolvidas para verificação do mecanismo de Mensagens Ativas implementado e apresentadas no próximo capítulo. Já o bloco indicado por Mensagens Ativas refere-se ao mecanismo desenvolvido. Ao integrar este módulo com o ambiente Anahy, os serviços do mesmo passarão a ser funcionalidades internas ao Anahy, não havendo conhecimento do usuário em relação a sua utilização. A Lógica de Controle Semântico e os serviços de Escalonamento, atualmente disponíveis no contexto local a um nó, também serão disponibilizados no contexto distribuído.

## 6.2 PRIMITIVAS

O mecanismo de Mensagens Ativas implementado neste trabalho pode ser utilizado pelo usuário de forma independente do ambiente Anahy, por se tratar de uma biblioteca a parte. No entanto, as facilidades de escalonamento e controle semântico ficam ao encargo do próprio programador. A integração do mecanismo com Anahy passa estas responsabilidades ao ambiente.

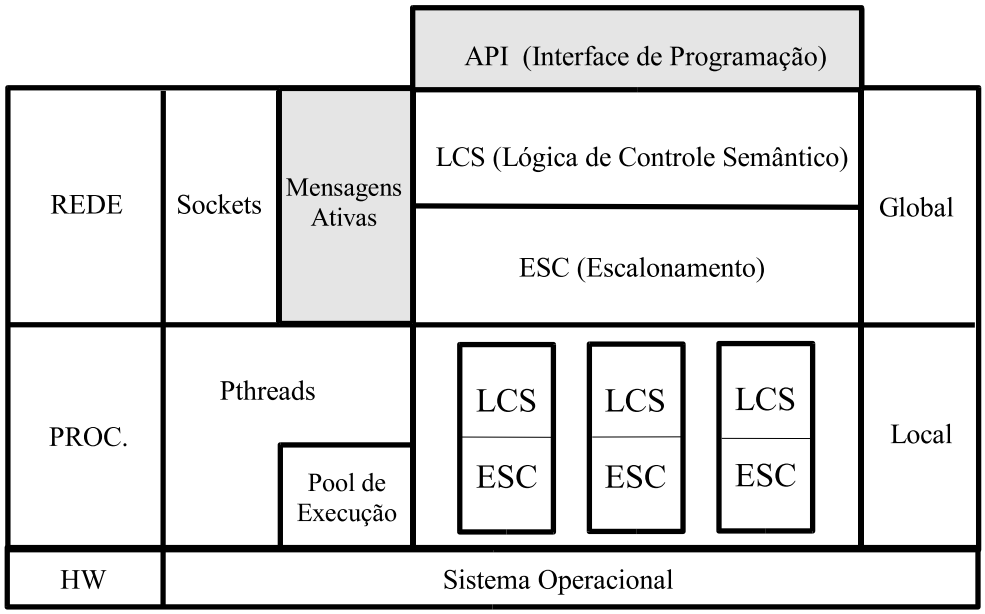


Figura 21: Composição modular de Anahy.

Neste trabalho, o mecanismo de Mensagens Ativas está sendo avaliado como componente final da versão de Anahy para aglomerados de computadores. Portanto, nos experimentos realizados foi necessário montar a aplicação com o suporte à distribuição de tarefas bem como de controle ao acesso aos dados compartilhados. Sua efetiva introdução no ambiente Anahy é uma tarefa desenvolvida no contexto do projeto Anahy ((CORDEIRO et al., 2005)), não do presente trabalho.

A biblioteca possui as primitivas apresentadas na Tab. 7, sendo descritas na seqüência.

Tabela 7: Primitivas da biblioteca.

| Nome | Descrição |
|---|---|
| act_msg_regserv | Recebe um nome de função e a registra na tabela de serviços do *daemon* |
| act_msg_create | Aloca espaço em memória, fazendo o *buffer* da mensagem apontar para o novo espaço em memória |
| act_msg_init | Inicializa a mensagem com o serviço previamente retornado pela função **act_msg_regserv** |
| act_msg_pack | Copia `n` bytes da área de dados para dentro do *buffer* da mensagem |
| act_msg_unpack | Copia `n` bytes do *buffer* da mensagem para uma área de memória |
| act_msg_send | Envia uma mensagem para o *host* destino |
| act_msg_reset | Desaloca o espaço em memória utilizado pelo *buffer* da mensagem e atribui `-1` ao identificador de serviço |
| av_init | Abre as conexões entre todos os nós e inicializa os *daemons* em todos eles |
| av_terminate | Finaliza os *daemons* em todos os nós do aglomerado e fecha todas as conexões abertas |

**unsigned int act_msg_regserv( void * (*function_name)(void *))**

Esta primitiva é responsável por registrar os serviços que serão invocados pelo usuário durante a execução de sua aplicação. É importante destacar que todos os serviços devem ser registrados antes do *daemon* de comunicação ser lançado, para que todos os nós do aglomerado tenham conhecimento dos serviços que o usuário utilizará.

O parâmetro passado à função corresponde ao nome do serviço que se deseja registrar. Tal nome refere-se ao nome de uma função que será executada quando o serviço registrado for solicitado.

O retorno da função deve ser atribuído a uma variável, que será utilizada no momento da invocação do serviço, e corresponde ao número do serviço registrado.

Observe-se, ainda, que o registro de diferentes serviços em diferentes nós de um aglomerado deve ser feito na mesma ordem em todos os nós, garantindo a correspondência de registros.

**int act_msg_create(act_msg_t *m, unsigned long size)**

Esta função corresponde à alocação de espaço para uma mensagem e para os respectivos dados. A mensagem (do tipo **act_msg_t**) armazenará, basicamente, informações sobre o espaço (de tamanho **size**) reservado aos dados e sobre o identificador do serviço a ser executado. Deve ser a primeira função chamada quando uma mensagem começa a ser criada para posterior envio. O retorno da função será zero em caso de sucesso e -1 em caso de falha.

**int act_msg_init(act_msg_t *m, int am_function)**

A função **act_msg_init** faz a inicialização do campo da mensagem **m** que identifica o serviço a ser executado, com o valor de **am_function** passado. Tal valor refere-se a variável à qual foi atribuído o retorno de **act_msg_regserv**. Em caso de sucesso, o retorno da função será zero. Caso contrário, será -1.

**int act_msg_pack(act_msg_t *m, const void *data, unsigned long size)**

Tal função agrega os dados (**data**) de tamanho **size** à mensagem **m** a ser enviada. Retorna a quantidade em bytes de espaço ainda livre no *buffer* da mensagem.

**int act_msg_send(const char *host, const int port, act_msg_t *m)**

A função **act_msg_send** é utilizada para enviar uma mensagem a um nó. O nó ao qual se deseja enviar a mensagem **m** é indicado por **host** e **port** indica a porta que será utilizada para comunicação. O retorno da função corresponde ao número de bytes que foram enviados.

**int act_msg_unpack(act_msg_t *m, void *data, size_t datasize, char *host)**

Esta função retira de um pacote de dados recebidos os dados relativos a uma

Mensagem Ativa. Os dados indicados por **data** devem ser atribuídos a uma variável do mesmo tipo dos dados empacotados em **act_msg_pack**. O parâmetro **host** indica o nó que enviou a mensagem para possível resposta. Retorna a quantidade de bytes restantes no *buffer* da mensagem que não foram copiados.

**int act_msg_reset(act_msg_t *m)**

Para liberar o espaço de memória anteriormente alocado para a mensagem **m**, utiliza-se a função **act_msg_reset**. O retorno será zero em caso de sucesso. Caso contrário, nenhum valor será retornado.

**int av_init(int argc, char **argv)**

Logo após serem registrados todos os serviços, a próxima função a ser invocada é **av_init**. Tal função é responsável por abrir as conexões entre todos os nós do aglomerado e inicializar os *daemons* de comunicação em todos eles, deixando-os preparados para começar a trocar mensagens. Cabe salientar que após a invocação de **av_init** somente o nó 0 (zero) voltará para a função *main* e executará o código que segue a função; os demais permanecem bloqueados esperando por mensagens a serem tratadas.

Os parâmetros passados para **av_init** são os mesmos da função **main**, pois é em **av_init** que os argumentos da linha de comando são utilizados. Os principais argumentos correspondem ao número de nós participantes da execução, ao nome do arquivo que contém a configuração (nome e IP) dos nós e a porta que será utilizada para comunicação entre os nós. O retorno da função será zero em caso de sucesso. Em caso de falha, nenhum valor é retornado.

**int av_terminate()**

Quando a aplicação tiver encerrado sua execução, deve ser invocada a função

**av_terminate** que finalizará os *daemons* em todos os nós e fechará todas as conexões previamente abertas. Em caso de sucesso, o valor 0 (zero) é retornado. Caso contrário, retorna-se -1.

Observe que **av_init** e **av_terminate** foram desenvolvidas com foco exclusivo de introdução em Anahy, por isso suas nomenclaturas não seguem o padrão adotado às demais funções da biblioteca.

## 6.3 ANÁLISE DOS SERVIÇOS

A implementação atual de Anahy está disponível para arquiteturas SMP. A utilização deste ambiente em aglomerados de computadores requer a implementação de serviços de comunicação que possibilitem a troca de tarefas e dados entre os nós do aglomerado. A partir daí, os serviços representados na Figura 22 foram detectados como necessários. Observa-se nesta figura que a estrutura de Anahy é mantida (ver Figura 19), havendo, no entanto, a inclusão de um novo elemento: o *daemon* de comunicação.

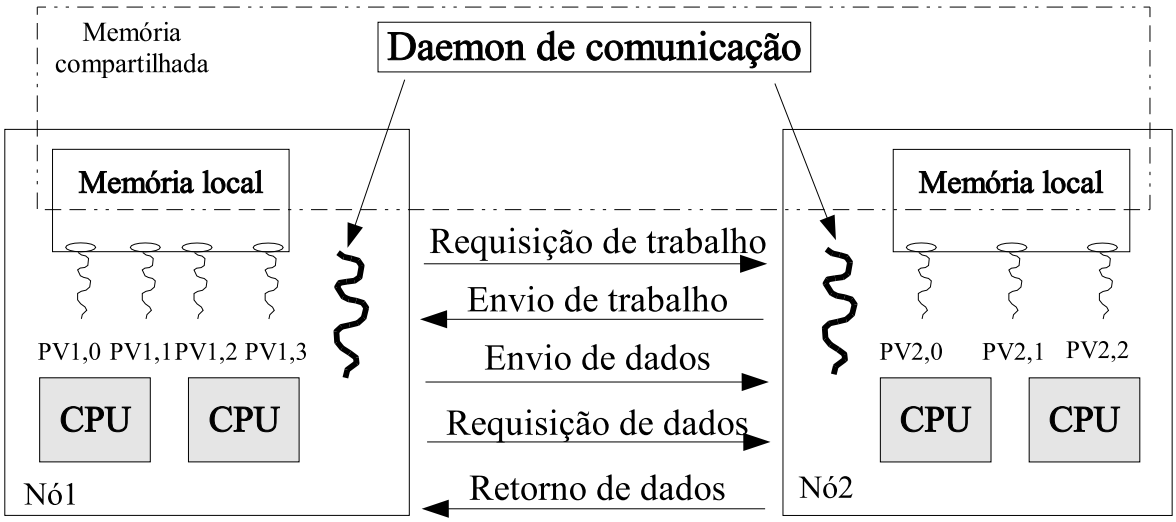


Figura 22: Introdução de suporte à comunicação em Anahy.

Na Figura 21 observa-se uma separação da estrutura em global e local. Estes níveis correspondem a exploração da concorrência entre-nós e intra-nó, respectivamente. A concorrência intra-nó, ou seja, a exploração dos recursos computacionais internos a um

nó (mais de um processador, por exemplo), é realizada pela versão atual de Anahy. Nesta versão estão implementadas todas as funcionalidades para controle das listas de tarefas, criação e sincronização de tarefas e escalonamento. A implementação destes serviços está baseada no uso da memória local ao nó. A exploração da concorrência entre-nós, por sua vez, exige que estas funcionalidades sejam igualmente aplicáveis a um contexto distribuído. Para tanto, serão necessárias extensões no núcleo executivo que possibilitem o funcionamento do escalonamento global e a manutenção do grafo de tarefas de forma distribuída.

O mecanismo de Mensagens Ativas implementado permite que, com a inexistência de tarefas em uma lista local, ocorram migrações de tarefas entre os nós, que serão possíveis através dos serviços de *i)* requisição de trabalho, *ii)* envio de trabalho e *iii)* envio de dados. As trocas de dados podem ser necessárias ao ocorrerem chamadas à operação *join* de Anahy, sendo implementados, para tanto, os serviços de *iv)* requisição de dados e *v)* retorno de dados. Tais serviços são comentados a seguir.

### 6.3.1 Requisição de Trabalho

Um nó do aglomerado, ao buscar uma tarefa na sua lista de tarefas local e esta estiver vazia, fará uma **requisição de trabalho** aos demais nós, sinalizando que está ocioso e disponível para executar novas tarefas. Essa divisão da carga de trabalho entre os nós visa diminuir o tempo total de execução da aplicação. Na Figura 22, este serviço é identificado pela primeira seta direcionado ao nó 2, indicando que o nó 1 está requisitando trabalho ao nó 2.

Em termos de implementação, este serviço consiste, basicamente, em uma mensagem ativa contendo informações referentes a origem e destino da requisição e um código identificando a operação a ser executada, no caso a requisição de trabalho.

### 6.3.2 Envio de Trabalho

A operação de **envio de trabalho** ocorre como resposta a uma requisição de trabalho descrita anteriormente. Na Figura 22, esta comunicação é representada pela segunda seta, onde o nó 2 envia trabalho ao nó 1, como resposta ao pedido de trabalho realizado pelo nó 1.

Novamente, este serviço consistirá de uma mensagem ativa enviada, neste caso, pelo nó que decidiu migrar alguma de suas tarefas para o nó que requisitou trabalho. Esta mensagem conterá informações confirmando ou negando a existência de trabalho a ser migrado naquele momento. Caso seja confirmada a existência de trabalho, a mensagem ativa enviada como resposta terá a descrição do trabalho a ser executado e os dados a serem manipulados.

### 6.3.3 Envio de Dados

O **envio de dados**, representado na Figura 22 pela terceira seta apontando do nó 1 para o nó 2, pode ser visto como um serviço complementar a requisição de trabalho, pois é realizado pelo nó que pediu trabalho a outro, utilizando o serviço de requisição de trabalho descrito anteriormente. O serviço de envio de dados consiste em uma sinalização comunicando o término da execução da tarefa que foi migrada e enviando os dados resultantes ao nó pai da tarefa, ou seja, ao nó de onde ela migrou.

A implementação deste serviço será feita através de uma mensagem ativa que conterá a identificação do serviço de envio de dados e os dados propriamente ditos.

### 6.3.4 Requisição de Dados

Este serviço é executado por um fluxo de execução que necessita de dados produzidos por outro fluxo de execução, que pode estar ou não localizado no mesmo nó. A implementação deste serviço é norteada pelo mesmo conceito da operação *join* acrescida de transparência de localização, já que o programador não necessita preocupar-se com o local

em que os dados serão buscados, precisando, apenas, identificar o fluxo que manipulou a tarefa e produziu os dados desejados.

### 6.3.5 Retorno de Dados

Assim como o envio de dados, a operação de **retorno de dados** também pode ser vista como um serviço complementar, neste caso ocorrendo em resposta à operação de requisição de dados. O nó, ao qual foram requisitados os dados de uma determinada tarefa, verifica a existência dessa tarefa e se seus dados já estão disponíveis. Caso estejam, são enviados por meio de uma mensagem ativa ao nó que requisitou o retorno dos dados. Caso ainda não estejam disponíveis, o nó requisitado aguarda pelo término da tarefa para, então, enviar os dados ao destino, no caso, o nó requerente.

## 6.4 EXTENSÃO DA INTERFACE DE PROGRAMAÇÃO DE ANAHY

Embora a interface de programação de Anahy forneça um estilo de programação *multithreaded*, programas podem ser executados em arquiteturas com memória distribuída, como aglomerados de computadores. Conseqüentemente, *threads* devem ser migradas entre os nós do aglomerado. O mecanismo de escalonamento foi desenvolvido para migrar *threads* transparentemente, sem que o programador preocupe-se com o mapeamento das tarefas em processadores ou de dados nos módulos de memória. No entanto, o programador deve fornecer informação sobre os dados requeridos (parâmetros) e produzidos (resultados) pelas *threads*, possibilitando a transferência dos dados.

Desta forma, propõe-se um mecanismo baseado em operações *pack*/*unpack*, considerando que o programador sabe manipular um tipo `void *` requerido pela *thread*. O uso deste mecanismo tem por objetivo viabilizar a transferência de *threads* Anahy entre diferentes nós. Os protótipos das funções *pack*/*unpack* para uma dada *thread* podem ser exemplificados por:

```
int packInFunc(void *in, char **buff);
int unpackInFunc(void *in, char **buff);
int packOutFunc(void *res, char **buff);
int unpackOutFunc(void *res, char **buff);
```

O primeiro parâmetro de cada função *pack/unpack* representa os dados a serem enviados (*in*) ou produzidos (*res*) para/por uma *thread*. O segundo parâmetro (*buff*) indica o *buffer* onde os dados de entrada da *thread* devem ser empacotados ou de onde devem ser lidos para serem desempacotados. Cada função retorna o tamanho (em bytes) dos dados empacotados/desempacotados na operação.

Como as operações *pack/unpack* estão relacionadas a *threads*, o programador associa específicas funções *pack/unpack* às *threads* no `athread_attr_t`:

```
int athread_attr_setpackinput(athread_attr_t * attr,
                              int (*func) (void *in, char **buff));
int athread_attr_setunpackinput(athread_attr_t * attr,
                                int (*func) (void *in, char **buff));
int athread_attr_setpackoutput(athread_attr_t * attr,
                               int (*func) (void *res, char **buff));
int athread_attr_setunpackoutput(athread_attr_t * attr,
                                 int (*func) (void *res, char **buff));
```

Uma vez que as operações *pack/unpack* são atributos de uma *thread*, elas podem ou não ser informadas no `athread_attr_t` (o valor default é NULL). No caso onde estes atributos não são especificados, a *thread* executará no nó onde foi criada.

## 6.5 COMENTÁRIOS FINAIS

Este capítulo apresentou uma camada desenvolvida para suporte de Anahy em aglomerados de computadores. Esta camada consiste em um mecanismo que emprega *multithreading* e Mensagens Ativas para exploração eficiente dos recursos de um aglomerado. Tal mecanismo foi construído, pois as bibliotecas de Mensagens Ativas encontradas na literatura não atenderam as necessidades do projeto Anahy.

Apesar de estar inserido no contexto do projeto Anahy, o mecanismo implementado foi construído como um módulo independente do restante do ambiente. Assim, algumas

primitivas de comunicação estão disponíveis ao usuário e podem ser empregadas para trocas de dados e tarefas entre os nós do aglomerado. No entanto, o escalonamento de tarefas, o controle da ordem de execução de tarefas e o acesso aos dados compartilhados ficam ao encargo do programador.

A integração do mecanismo de Mensagens Ativas com o ambiente Anahy tornará o ambiente responsável pelo controle semântico e pelo escalonamento também no contexto distribuído. Com isso, o usuário Anahy não precisará preocupar-se nem com o escalonamento nem com a sincronização de suas tarefas. Para tanto, foram descritos alguns serviços que devem ser implementados de forma a possibilitar a troca de tarefas e dados entre os nós do aglomerado. Também mostrou-se necessário estender a interface de Anahy, de forma a viabilizar a comunicação de *threads* entre nós.

# 7 ANÁLISE DE DESEMPENHO

Para verificar o funcionamento do mecanismo implementado, foram realizados uma série de experimentos explorando a biblioteca. Os experimentos envolveram duas aplicações: cálculo do Número de Fibonacci e alinhamento de seqüências biológicas. O cálculo do Número de Fibonacci possui pouca demanda computacional no que se refere a tempo de processamento e consumo de memória, mas quantidade de comunicações (troca de dados) suficiente para uma boa avaliação do mecanismo. O alinhamento de seqüências, ao contrário de Fibonacci, apresenta alto consumo de memória, maior tempo de processamento e manipula mensagens de grande tamanho.

Este capítulo destina-se a apresentar as aplicações desenvolvidas no que diz respeito a seus fluxos de execução (dependências entre tarefas) e como foram implementadas utilizando o mecanismo desenvolvido. Também são apresentados os resultados de desempenho obtidos com as execuções das aplicações, de maneira a verificar o comportamento das mesmas quando executadas de forma distribuída com o suporte do mecanismo implementado. As seções 7.1 e 7.2 apresentam, respectivamente, o cálculo do Número de Fibonacci e o processo de alinhamento de seqüências, com o grafo de dependências entre tarefas, algoritmo e resultados de desempenho para cada caso. Na Seção 7.3 é descrito como as operações de lógica de controle semântico e escalonamento implementadas nas aplicações foram mapeadas em serviços Anahy. Por fim, na Seção 7.4 são feitos alguns comentários finais sobre os resultados obtidos no capítulo.

Os experimentos foram executados em um aglomerado de computadores formado por 5 máquinas XEON 2.8Ghz, biprocessadas, com 2GB de memória principal, executando

sistema operacional Gobo Linux kernel 2.6.5 e as aplicações foram compiladas com o compilador gcc 3.3.1. Os tempos apresentados nas Seções 7.1.3, 7.2.5 e 7.2.6 estão referenciados em segundos e são uma média de 100 execuções para cada caso apresentado.

## 7.1 CÁLCULO DO NÚMERO DE FIBONACCI

A conhecida seqüência de Fibonacci, dada por:

$$\begin{aligned} &F(0) = 0 \\ &F(1) = 1 \\ &F(n) = F(n - 1) + F(n - 2), \text{ para } n \geq 2 \end{aligned}$$

apresenta-se como um interessante problema a ser calculado em paralelo, não tanto pelo seu uso prático, mas sim pela sua capacidade de representar aplicações com alto grau de concorrência e dependências de dados. Exemplos de seu uso para avaliação de sistemas paralelos são encontrados em (BLUMOFE et al., 1995; GALLILÉE et al., 1998; PRICE; LOWENTHAL, 2003; BENITEZ et al., 2004). A demanda computacional deste problema no que diz respeito a tempo de processamento e utilização de memória, pode ser considerada pequena se comparada a de outros problemas. No entanto, o número de tarefas que podem ser executadas de forma concorrente e a quantidade de comunicações exigidas no caso de uma execução distribuída, o tornam uma excelente ferramenta para avaliação do comportamento de mecanismos de comunicação.

### 7.1.1 Grafo de Dependências

O cálculo da seqüência de Fibonacci pode ser realizado em paralelo, explorando relações de independência entre tarefas. O cálculo de um número `n` qualquer gera duas sub-árvores principais, as quais são formadas por outras sub-árvores para cálculo dos números `n-1` e `n-2` e assim recursivamente, até se chegar ao cálculo de um dos dois valores "folha" (zero ou um). São essas sub-árvores que podem ser calculadas concorrentemente. Este processo é ilustrado na Figura 23, para `n` = 4.

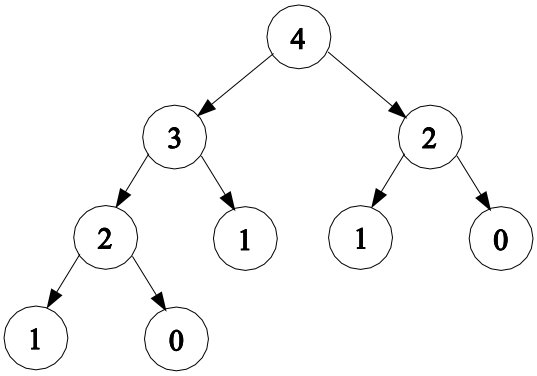


Figura 23: Exemplo do cálculo de Fibonacci do número 4.

No exemplo da Figura 23 são geradas duas sub-árvores principais a partir do número 4, uma cuja raiz é o número 3 e outra cuja raiz é o número 2. Essas duas sub-árvores são divididas em outras duas, as quais podem ser calculadas independentemente uma da outra, possibilitando a introdução de concorrência. Para o caso do 2, é gerada uma sub-árvore com raiz 1 e outra com raiz 0. Neste caso, ambos valores são folha e, conforme o algoritmo, retornam seus respectivos valores para o nó pai (2). No caso do 3, é gerada uma sub-árvore com raiz 2 e outra com raiz 1. A com raiz 1 retorna imediatamente seu valor para o nó pai (3), enquanto a com raiz 2 é dividida em outras duas sub-árvores - uma com raiz 1 e outra com raiz 0, que retornam imediatamente seus valores para o nó pai (2). Cada nó pai, ao receber o(s) valor(es) de seu(s) filho(s), os soma e retorna para seu pai, até que se chegue ao valor raiz, no caso o 4, onde serão somados os valores retornados por seus dois filhos (3 e 2). Neste exemplo de execução, foram geradas 9 tarefas.

Desejando-se calcular, por exemplo, o Fibonacci do número 5, que geraria duas sub-árvores principais, uma com raiz 4 e outra com raiz 3, a sub-árvore com raiz 4 seria exatamente igual a apresentada na Figura 23. A outra teria como raiz o número 3 e seria exatamente igual a sub-árvore da esquerda apresentada na Figura 23. A Figura 24 apresenta o exemplo de cálculo do Fibonacci de 5. Neste caso são geradas 15 tarefas.

Observando-se os dois exemplos, nota-se a considerável quantidade de novos valores a serem calculados quando se aumentou em apenas uma unidade o Número de Fibonacci

que desejávamos calcular - de 4 para 5. Considerando-se que 4 e 5 são números pequenos, pode-se ter idéia da quantidade de valores a serem calculados para números maiores que 20, por exemplo. Conseqüentemente, tem-se idéia da quantidade de tarefas que podem ser executadas concorrentemente e da quantidade de comunicações que podem ser necessárias caso os cálculos ocorram em uma arquitetura distribuída.

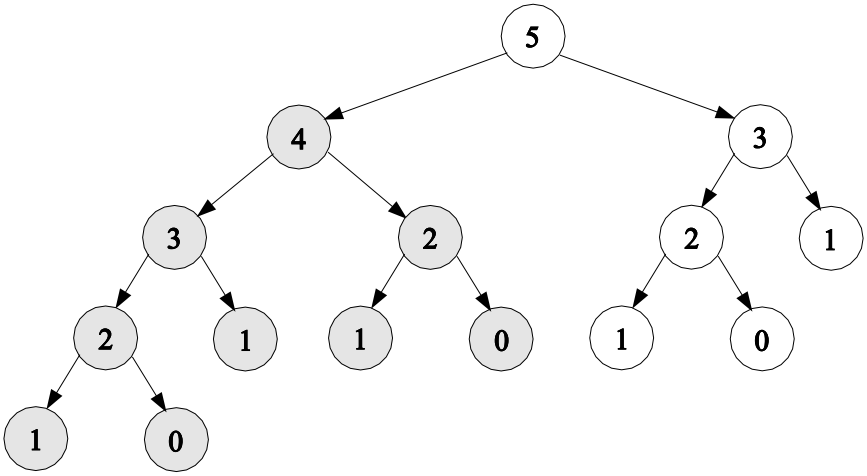


Figura 24: Exemplo do cálculo de Fibonacci do número 5.

### 7.1.2 Algoritmo

A Figura 25 apresenta a parte principal do algoritmo distribuído implementado para o cálculo do Número de Fibonacci.

Como dito no capítulo anterior, as duas principais características do mecanismo implementado são a presença de um *daemon* de comunicação, responsável por retirar da rede as mensagens enviadas a um nó e que é parte integrante da biblioteca de comunicação, e por processadores virtuais, que conferem a característica *multithreading* ao mecanismo, sendo responsáveis por executar efetivamente o cálculo desejado e cuja responsabilidade de implementação fica a cargo do usuário.

O trecho apresentado na Figura 25 corresponde ao trabalho que deve ser realizado pelos processadores virtuais. Cada processador virtual corresponde a uma *thread*, criada logo no início da execução e que permanece bloqueada enquanto a lista de tarefas do nó estiver vazia (linhas 2 e 3). Ao ser inserida uma tarefa na lista, uma das *threads* bloqueadas

é liberada para execução, tomando para si o valor incluído (linha 4). As demais linhas do trecho apresentado referem-se ao cálculo de Fibonacci propriamente dito.

```
1 enquanto(1){
2       enquanto lista de tarefas prontas estiver vazia
3               espera
4       n = primeiro elemento da lista de tarefas prontas
5
6       se(n >= 2){
7               n = n - 1
8               cria mensagem
9               inicializa mensagem
10              empacota mensagem
11              escolhe nó randômico para enviar mensagem
12              envia mensagem para nó randômico inserir n na
                lista de tarefas
13
14              n = n - 1
15              cria mensagem
16              inicializa mensagem
17              empacota mensagem
18              escolhe nó randômico para enviar mensagem
19              envia mensagem para nó randômico inserir n na
                lista de tarefas
20      }
21      senão{
22              se(n != 0){
23                      cria mensagem
24                      inicializa mensagem
25                      empacota mensagem
26                      envia mensagem para nó 0 acumular valor
27              }
28      }
29 }
```

Figura 25: Algoritmo implementado para o cálculo do Número de Fibonacci.

Entre as linhas 6 e 20 trata-se o caso onde o número n a ser calculado é maior ou igual a 2. Neste caso, dois novos cálculos de Fibonacci devem ser disparados, um para (n-1) e outro para (n-2). Para tanto, são geradas duas mensagens (uma para cada valor a ser calculado), escolhidos dois nós randomicamente e enviado a cada nó a mensagem correspondente. Cada mensagem enviada, carrega consigo a indicação de qual função deve ser executada no nó receptor e os dados necessários à execução desta função. No caso do cálculo de Fibonacci implementado, a função a ser executada quando tal mensagem for recebida, corresponde à inserção do dado recebido como parâmetro na lista de tarefas prontas para serem calculadas. Se houver alguma *thread* bloqueada, a inserção na lista causará o desbloqueio da mesma. Caso não haja, o dado permanece na lista de tarefas até que uma *thread* termine seu serviço e busque algum outro valor para calcular.

O trecho compreendido entre as linhas 21 e 28 trata os casos de n igual a zero ou 1. Se n for igual a zero, nenhuma atitude é tomada, simplesmente o valor é ignorado. Caso contrário, uma mensagem é gerada, tendo como parâmetros a indicação de uma função a ser executada no receptor e o dado que deve ser utilizado pela função. Neste caso, o nó receptor refere-se ao nó 0 (zero) e a função será para acúmulo dos valores recebidos. Ao detectar que o resultado acumulado é igual ao resultado esperado, o nó 0 envia uma mensagem a todos os nós indicando que a execução deve ser encerrada.

### 7.1.3 Resultados

Os resultados de desempenho apresentados nesta seção referem-se ao cálculo de Fibonacci dos números 10 e 15 e foram obtidos a partir de execuções utilizando a arquitetura especificada nas páginas 102 e 103. Como comentado, a carga de processamento para este problema é muito pequena, o que não exige elevados tempo de processamento e consumo de memória. Porém, são muitas as tarefas que podem ser executadas concorrentemente, a quantidade de comunicações exigidas no caso de uma execução distribuída é grande e o número de sincronizações entre tarefas necessárias para evolução da execução também é elevado. Os resultados de desempenho obtidos são apresentados como gráficos nas Figuras 26 (Fibonacci de 10) e 27 (Fibonacci de 15).

Observando-se as curvas de ambas figuras, percebe-se que a introdução de concorrência intra-nó, com o aumento do número de processadores virtuais, possibilitou ganho de desempenho. Os ganhos mais consideráveis foram obtidos com a introdução de até quatro processadores virtuais. A partir daí, até oito processadores virtuais, ainda obteve-se algum ganho, embora pequeno. Com mais de oito processadores virtuais, o tempo se manteve ou aumentou, mesmo que pouco, devido ao *overhead* introduzido pela criação de tarefas.

Por fim, observa-se que quando dobrou-se a concorrência entre-nós, aumentado-se de um para dois o número de nós, conseguiu-se obter cerca de 50% de ganho de

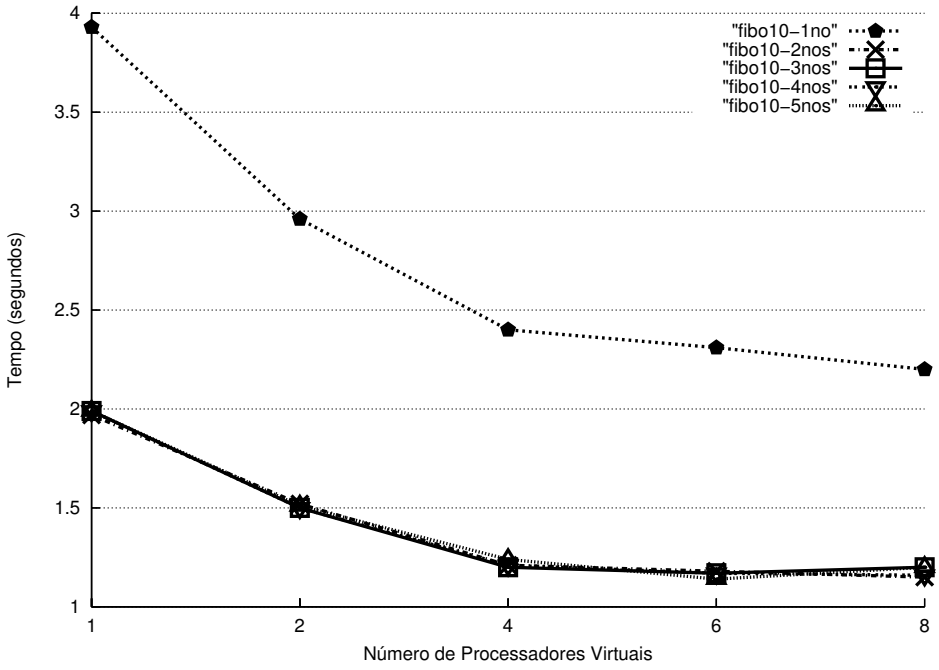


Figura 26: Tempos de execução para o Fibonacci de 10 com diferentes números de nós e processadores virtuais.

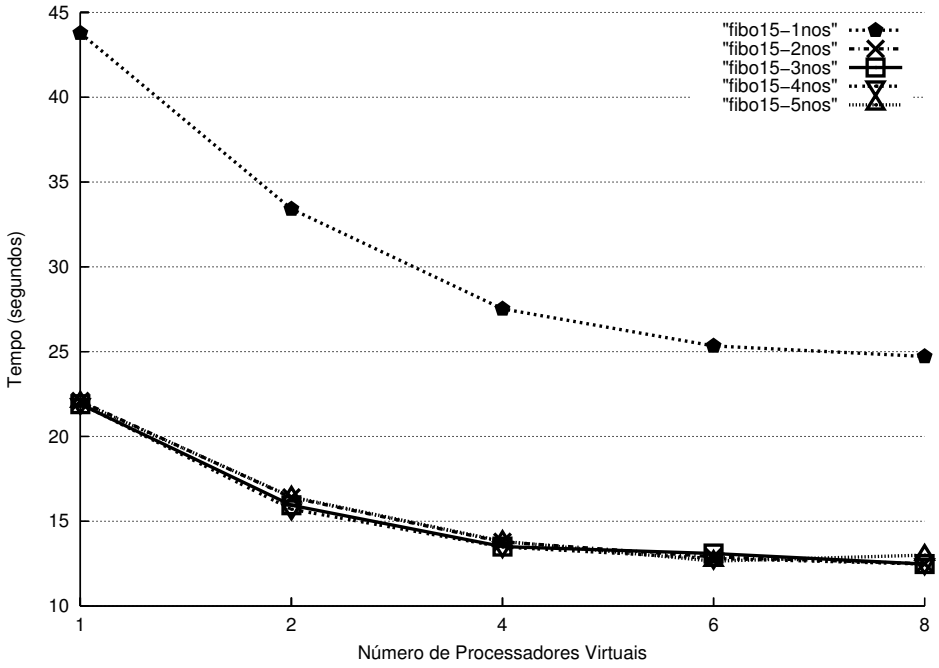


Figura 27: Tempos de execução para o Fibonacci de 15 com diferentes números de nós e processadores virtuais.

desempenho para os dois casos. Isso mostra que o *overhead* introduzido pela biblioteca de Mensagens Ativas não é grande.

## 7.2 ALINHAMENTO DE SEQÜÊNCIAS

O problema do alinhamento de seqüências apresentado em capítulos anteriores mostrou-se um excelente problema real a ser explorado. Tal interesse deve-se, principalmente, a grande demanda por poder computacional, no que se refere a tempo de processamento e consumo de memória; características que podem levar ao emprego de processamento de alto desempenho em arquiteturas com memória distribuída. Além disso, a forma como o cálculo da matriz de similaridades é realizado possibilita a introdução de execução concorrente no mesmo e controle de execução através das relações de dependências. Algumas estratégias foram apresentadas quando se tratou de trabalhos relacionados a paralelização do método de programação dinâmica. Esta seção destina-se a discutir a estratégia de introdução de concorrência no cálculo da matriz de similaridades adotada neste trabalho e apresentar uma avaliação de seu desempenho.

A relação de recorrência utilizada para o preenchimento da matriz de similaridades, apresenta três dependências para o cálculo de uma célula da matriz, conforme mostrado esquematicamente na Figura 28.

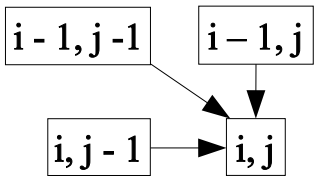


Figura 28: Dependência no cálculo de um elemento da matriz de similaridades.

Dadas as dependências de dados entre os cálculos, a matriz de similaridades pode ser preenchida (*i*) linha a linha, (*ii*) coluna a coluna ou, ainda, (*iii*) pelas antidiagonais em um processo de "inundação", segundo estratégias comentadas na Seção 2.4. O problema das duas primeiras estratégias é que a maioria dos elementos em uma linha ou coluna da matriz depende diretamente dos outros elementos da mesma linha ou coluna. Desta maneira, as linhas ou colunas não podem ser calculadas em paralelo devido ao grande número de sincronizações. A terceira estratégia, cálculo da matriz por suas antidiago-

nais, não apresenta este problema, pois uma antidiagonal depende somente dos elementos das outras antidiagonais previamente calculadas. Mesmo apresentando um grau de concorrência mais elevado, esta estratégia apresenta problemas para uma implementação paralela eficiente. Um dos problemas é que o tamanho das antidiagonais varia durante o preenchimento da matriz, causando, assim, uma não uniformidade no número de tarefas durante a execução do programa. Outro problema desta estratégia está na definição da granulosidade. Se para cada célula da matriz for criada uma tarefa, o número total de tarefas criadas será muito elevado, proporcional ao produto do tamanho das seqüências em análise. Em dados biológicos reais, o número de tarefas pode vir a ser grande o suficiente para que o sobrecusto (*overhead*) de sincronização entre estas sobreponha o potencial ganho de uma execução paralela.

Uma solução ao problema de granulosidade é dividir a matriz de similaridades em blocos de elementos, como mostrado na Figura 29. Neste exemplo, um programa em execução inicia pelo cálculo do bloco 1, para, em seguida, calcular os blocos 2 e 3, visto que as dependências foram resolvidas pelo cálculo do bloco 1. Toda a matriz de similaridades é calculada desta forma e as dependências existentes entre os blocos (conjunto de células) são as mesmas das existentes entre as células (Figura 28).

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 6 | 10 | 15 |
| 2 | 5 | 9 | 14 | 19 |
| 4 | 8 | 13 | 18 | 22 |
| 7 | 12 | 17 | 21 | 24 |
| 11 | 16 | 20 | 23 | 25 |

Figura 29: Cálculo em blocos da matriz de similaridades.

### 7.2.1 Grafo de Dependências

Uma outra forma de representação da matriz da Figura 29 consiste em um grafo de dependências, conforme ilustrado na Figura 30. Nesta figura, os nós do grafo representam

os blocos da matriz e as setas indicam as dependências existentes entre os blocos. Dessa forma, uma seta saindo de um nó e incidindo em outro, indica que o nó de onde parte a seta produz dados que serão utilizados pelo nó onde chega a seta, sendo necessária a existência de uma sincronização entre os dois nós. Assim, um nó com três setas incidentes, por exemplo, necessita realizar três operações de sincronização antes de iniciar sua execução.

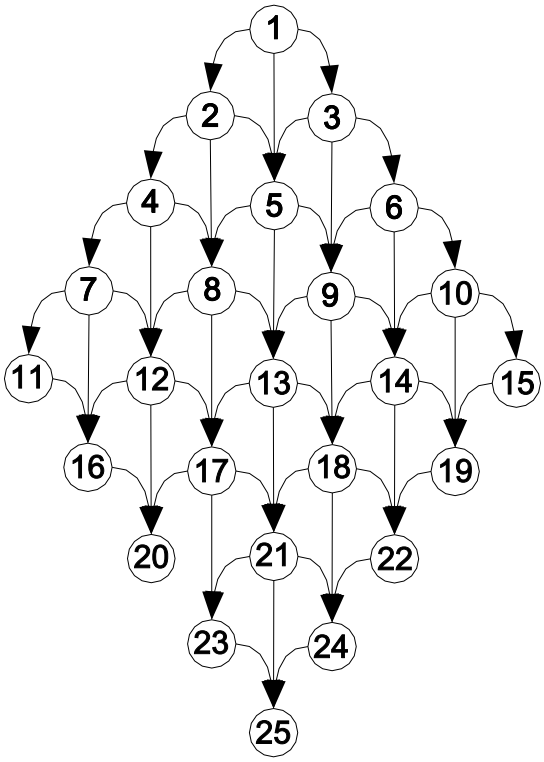


Figura 30: Grafo de dependências da matriz de similaridades.

### 7.2.2 Resultados Preliminares

Os estudos preliminares envolvendo o alinhamento de seqüências biológicas realizados no presente trabalho, resultaram na implementação de uma aplicação que emprega o algoritmo de Smith-Waterman para realizar o alinhamento entre duas seqüências(LERMEN; PERANCONI; CAVALHEIRO, 2004b, 2004a). Com o intuito de analisar o comportamento desta aplicação, foi realizada uma série de experimentos sobre uma arquitetura biprocessada (2 XEON 2.8GHz, 1 GB RAM, Gobo Linux, kernel 2.4.21). Considerando a arquitetura utilizada, mais especificamente a quantidade de memória disponível, foi possível realizar o alinhamento de 3981 resíduos do genoma da bactéria

*Mycoplasma genitalium* com 4026 resíduos do genoma da bactéria *Mycoplasma pneumoniae*, originando uma matriz de similaridades com cerca de 16.000.000 de elementos. Os resultados de desempenho apresentados nesta seção referem-se ao tempo de execução da aplicação, nas suas diferentes versões, em segundos. Cada resultado apresentado consiste na média de 100 execuções com o desvio padrão inferior a 10%.

O primeiro conjunto de resultados a ser discutido encontra-se na Tab. 8, que apresenta os tempos obtidos pela execução de diferentes versões da aplicação na arquitetura utilizada. Salienta-se que, os resultados apresentados nesta tabela, referem-se a execuções não paralelas, ou seja, exploram apenas um dos processadores da arquitetura anteriormente descrita. Estes resultados permitem analisar alguns dos custos associados à execução destas diferentes versões.

Tabela 8: Tempos para diferentes versões da aplicação em execução seqüencial.

| **Versões da aplicação** | **Medida** | **Tempo** |
|---|---|---|
| Seqüencial - cálculo por elementos | S | 17,03 |
| Concorrente Anahy | $T_1$ | 6,06 |
| Seqüencial - cálculo por blocos | $T_s$ | 6,02 |

Em um primeiro momento foi coletado o tempo de execução $S$ da versão seqüencial (cálculo por elementos) da aplicação, o qual corresponde à implementação direta do algoritmo de programação dinâmica. Neste caso, o algoritmo inicia percorrendo toda a matriz de similaridades, da esquerda para a direita e de cima para baixo, realizando o cálculo elemento por elemento, varrendo a matriz linha a linha. Como define o método de programação dinâmica, cada elemento é calculado considerando os valores de seus vizinhos (norte, oeste e noroeste) e os valores máximos encontrados em cada linha e coluna. O tempo obtido para esta versão seqüencial foi de $S = 17{,}03$s.

O segundo experimento realizado refere-se à análise dos tempos obtidos pela execução da implementação concorrente sobre recursos de processamento não paralelos. Esta medida encontra-se referenciada na Tab. 8 como $T_1$, tendo sido aferido como $T_1 = 6{,}06$s. Salienta-se que, diferentemente da versão seqüencial, a versão concorrente realiza as operações por blocos de elementos de forma a aumentar a granulosidade das tarefas

de cálculo (no caso dos dados apresentados nesta tabela, os blocos possuem tamanho de 15x15 elementos). Somente foi possível coletar o tempo $T_1$ por ter sido utilizado como mecanismo de suporte à concorrência a ferramenta Anahy.

No caso da medida $T_1$, a máquina virtual de Anahy foi configurada com apenas 1 processador virtual. Desta forma, as tarefas concorrentes definidas pelo programa em execução são executadas em um fluxo único. Assim, embora seja agregado ao tempo de execução todo custo adicional envolvido na gestão da concorrência pelo ambiente Anahy, não é explorado o ganho que pode ser obtido na execução paralela do algoritmo. Portanto, a diferença entre os tempos $S$ e $T_1$ representa o *overhead* da descrição e da gestão da concorrência.

Chama atenção que a versão concorrente sendo executada de forma não paralela possui um tempo de execução inferior à versão seqüencial. Este fato advém das diferentes formas adotadas pelos algoritmos para realização dos cálculos. A versão seqüencial realiza o cálculo elemento por elemento, pesquisando cada linha e coluna para encontrar o maior elemento a cada elemento calculado. A versão concorrente realiza o cálculo por blocos, utilizando como entrada um vetor com os máximos valores de linha e coluna já encontrados, evitando um processo de busca mais demorado.

Uma nova versão seqüencial foi composta para igualmente realizar o cálculo por blocos, recebendo como entrada um vetor com os máximos já encontrados em cada linha e coluna. O tempo obtido para esta nova versão (para o mesmo tamanho de bloco) foi de $T_s = 6{,}02$s. Portanto coerente com a expectativa de que existe a introdução de *overhead* para obter uma versão concorrente de um algoritmo seqüencial. A conclusão da análise dos tempos apresentados indica que o algoritmo concorrente pode ser suportado sem a introdução de sobrecusto significativo na execução. Além disso, Anahy não introduz um grande custo de gerência da execução das atividades concorrentes.

Na seqüência, foram tomados os tempos com execuções paralelas do algoritmo concorrente, sobre os dois processadores da arquitetura SMP disponível, considerando di-

ferentes tamanhos de blocos. Para tanto, a aplicação foi reestruturada para oferecer dois suportes à concorrência: *threads* POSIX e Anahy. Destaca-se que estas versões possuem estrutura e comportamento idênticos. Os tempos medidos para as execuções paralelas encontram-se na Tab. 9. Nesta tabela, a primeira coluna identifica o número de blocos utilizados no experimento, refletindo o número de tarefas concorrentes (*threads*) e a granulosidade destas. A segunda coluna relata os tempos obtidos pela execução do programa com suporte de *threads* POSIX. As colunas restantes referem-se à execução do programa com suporte de Anahy tendo sua máquina virtual configurada com diferentes números de processadores virtuais.

Tabela 9: Tempos para execuções de versões concorrentes da aplicação.

| **Número de threads** | **POSIX** | **Anahy** | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 PV | 2 PVs | 5 PVs | 10 PVs | 15 PVs | 20 PVs |
| 2x2 | 14,97 | 7,19 | 7,36 | 7,04 | 7,02 | 7,04 | 7,04 |
| 3x3 | 12,74 | 6,45 | 6,64 | 6,04 | 6,04 | 6,04 | 6,04 |
| 4x4 | 11,83 | 6,18 | 6,31 | 5,73 | 5,71 | 5,72 | 5,72 |
| 5x5 | 11,51 | 6,07 | 6,22 | 5,74 | 5,60 | 5,60 | 5,61 |
| 10x10 | 11,18 | 6,04 | 6,49 | 6,02 | 5,59 | 5,53 | 5,54 |
| 15x15 | 11,16 | 6,06 | 6,43 | 6,16 | 5,69 | 5,64 | 5,59 |

Na versão POSIX, cada uma das atividades concorrentes definida pelo programa é transformada em uma unidade de cálculo paralelo, ou seja, em uma *thread* POSIX. Estas *threads* são, portanto, criadas e destruídas à medida que o programa evolui, existindo, então, a possibilidade de diversas *threads* estarem ativas em um determinado instante de tempo. No caso do estudo realizado, até 15 *threads* ativas quando da realização do cálculo dividindo-se a matriz em 15x15 blocos (durante o cálculo da maior antidiagonal da matriz). Os tempos obtidos nesta versão indicam que um certo grau de paralelismo é necessário para tirar proveito do hardware (biprocessado) disponível. Porém, o número de atividades paralelas definidas pelo programa deve ser limitado à capacidade efetiva da máquina.

Um melhor mapeamento da concorrência da aplicação no paralelismo da arquitetura (conforme Tab. 9), foi obtido com o uso de Anahy como suporte à execução. Neste

caso, o número de atividades em execução simultânea é limitado pela capacidade de processamento da máquina virtual, a qual é determinada pelo número de processadores virtuais empregados. Assim, embora possam ocorrer situações onde diversas *threads* definidas pela aplicação estejam aptas a executar, o número de atividades que ocorrem efetivamente em paralelo está limitado ao número de processadores virtuais disponíveis, reduzindo custos advindos da gestão da concorrência definida pela aplicação. Os tempos obtidos reforçam a idéia de que Anahy introduz pouco sobrecusto frente a uma execução seqüencial da aplicação, permitindo ainda, embora modesto, algum ganho de desempenho. A expectativa é que Anahy ofereça um maior conforto de programação, uma vez que, face ao uso de *threads* POSIX, permite uma maior liberdade para o programador definir a granulosidade de seu programa em função da concorrência natural da aplicação.

Os resultados de desempenho apresentados demonstraram que a introdução de concorrência na aplicação foi capaz de reduzir o tempo de processamento total. No entanto, as seqüências alinhadas ainda são pequenas, dado o limite de memória da arquitetura utilizada. Buscando aumentar o tamanho das seqüências alinhadas, foi desenvolvida uma versão da aplicação capaz de executar em arquiteturas com memória distribuída tendo suporte do mecanismo de Mensagens Ativas implementado.

### 7.2.3 Implementação em Memória Distribuída

Com a matriz de similaridades dividida em blocos, necessita-se adotar uma estratégia que distribua os blocos a serem calculados entre os nós do aglomerado, buscando-se, sempre, diminuir o número de comunicações necessárias entre os nós. Diferentes estratégias podem ser adotadas, como pode ser visto na apresentação dos trabalhos relacionados (conforme Seção 2.4).

No presente trabalho, a estratégia de distribuição adotada está baseada no que se chamou de "fluxo de execução". Um fluxo de execução é formado por todos os nós do grafo cujas relações de dependência encontrem-se em alguma das antidiagonais da matriz.

Estas relações de dependência podem ser vistas na Figura 31 através da seqüência formada por nós tendo setas verticais incidentes ou de saída. No caso da figura, são identificados 9 fluxos de execução.

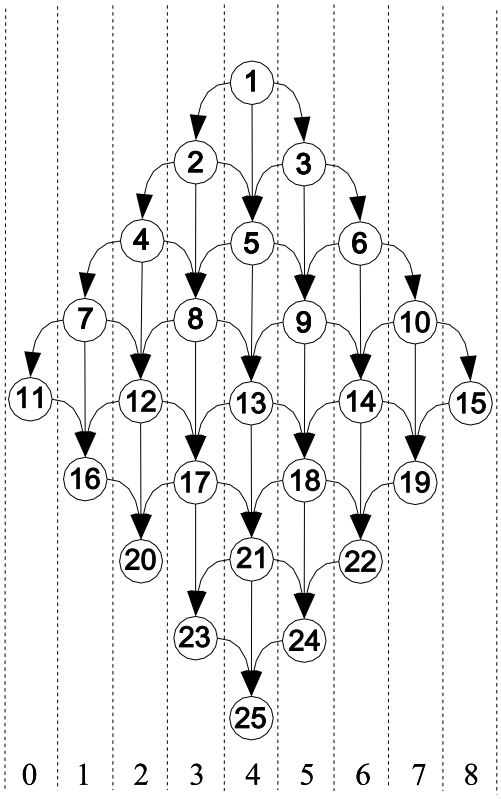


Figura 31: Estratégia de escalonamento adotada.

A idéia, neste caso, é fazer com que nós do grafo, pertencentes a um mesmo fluxo de execução, sejam atribuídos ao mesmo nó do aglomerado, de forma a explorar a localidade das tarefas, diminuindo o número de comunicações exigidas para troca de dados.

Assim, para se determinar o fluxo de execução a qual pertence um determinado bloco, utiliza-se a Fórmula 7.1. Nesta fórmula, `i` e `j` indicam, respectivamente, a linha e coluna da matriz onde se encontra o bloco, sendo que ambos os valores começam a ser contados a partir de 1 indo até `m` para o caso de `i` e `n` para o caso de `j`. Os valores de `m` e `n` indicam, respectivamente, a quantidade de blocos horizontais e verticais em que a matriz de similaridades deve ser dividida.

$$
\begin{aligned}
&se(i == j) \\
&\qquad fluxo = tfluxo/2 \\
&senão \\
&\qquad fluxo = (j - i) + (tfluxo/2)
\end{aligned}
\tag{7.1}
$$

O valor de `tfluxo` que aparece na Fórmula 7.1 representa a quantidade de fluxos gerados durante a execução do programa e pode ser obtido a partir da fórmula 7.2.

$$tfluxo = m + n - 1 \tag{7.2}$$

Sabendo-se a que fluxo de execução pertence um bloco, resta atribuí-lo a um nó do aglomerado para que sejam realizadas as operações sobre o mesmo. O cálculo do nó para o qual o bloco será atribuído está representado na Fórmula 7.3. Observe-se que somente é considerada a parte inteira da divisão. O valor de `nhosts` indica a quantidade de nós do aglomerado.

$$no = (fluxo * nhosts)/tfluxo \tag{7.3}$$

Aplicando-se os cálculos apresentados para determinação de fluxo e nó de execução em uma matriz hipotética de tamanho 5x5 blocos como a da Figura 29, o nó onde cada bloco será calculado é indicado pelos números na linha superior da Figura 32, enquanto que o fluxo de execução é indicado pelos números da linha inferior da mesma figura.

Conforme a Figura 32 mostra, a estratégia de distribuição de trabalho entre os nós também responde ao requisito de localidade. Fluxos de execução próximos, portanto que possuem relações de dependência, são alocados no mesmo nó.

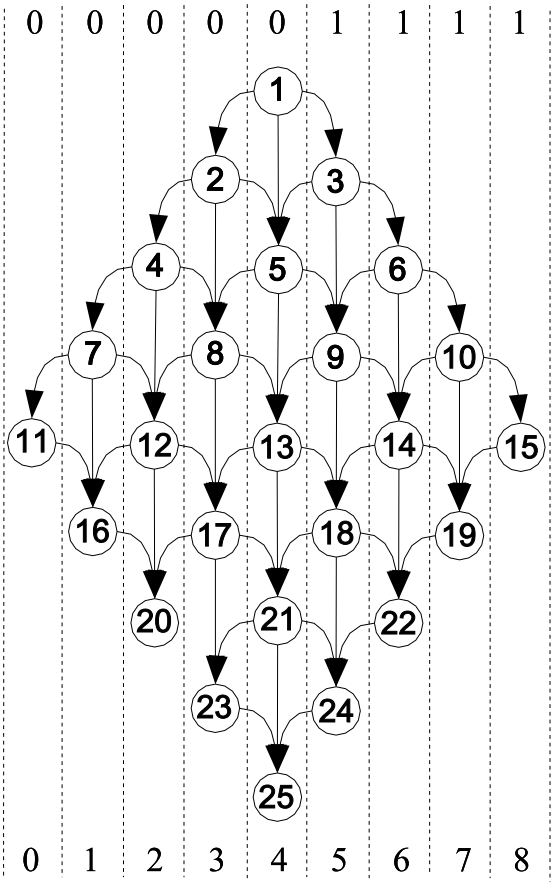


Figura 32: Estratégia de escalonamento adotada com indicação de fluxo e nó de execução.

### 7.2.4 Algoritmo

A Figura 33 apresenta a parte principal do algoritmo distribuído implementado para o cálculo concorrente da matriz de similaridades de acordo com a estratégia apresentada anteriormente.

O algoritmo apresentado na Figura 33 indica o trabalho que deve ser realizado por um processador virtual durante o processo de alinhamento de duas seqüências. Como no caso do cálculo de Fibonacci, cada processador virtual corresponde a uma *thread*, criada logo no início da execução e que permanece bloqueada enquanto a lista de tarefas do nó estiver vazia (linhas 2 e 3). Ao ser inserida uma tarefa na lista, uma das *threads* bloqueadas é liberada para execução, tomando para si o bloco incluído (linha 4). Após ser retirado da lista de tarefas prontas, o bloco passará a ser calculado (preenchido) de acordo com a relação de recorrência para preenchimento da matriz de similaridades apresentada em

```
1 enquanto(1){
2       enquanto lista de tarefas prontas estiver vazia
3               espera
4       b = primeiro bloco da lista de tarefas prontas
5
6       calcula bloco b
7
8       se((b.i != m) && (b.j != n)){
9               b1.i = b.i;
10              b1.j = b.j + 1;
11              calcula fluxo a que pretence b1
12              calcula nó em que b1 deve executar
13              cria mensagem
14              inicializa mensagem
15              empacota mensagem
16              envia mensagem
17
18              b1.i = b.i + 1;
19              b1.j = b.j;
20              calcula fluxo a que pretence b1
21              calcula nó em que b1 deve executar
22              cria mensagem
23              inicializa mensagem
24              empacota mensagem
25              envia mensagem
26
27              b1.i = b.i + 1;
28              b1.j = b.j +1;
29              calcula fluxo a que pretence b1
30              calcula nó em que b1 deve executar
31              cria mensagem
32              inicializa mensagem
33              empacota mensagem
34              envia mensagem
35      }
36      senão{
37              se((b.i == m) && (b.j == n)){
38                      cria mensagem
39                      inicializa mensagem
40                      empacota mensagem
41                      envia mensagem de encerramento
42              }
43              senão{
44                      se(b.i == m){
45                              b1.i = b.i;
46                              b1.j = b.j + 1;
47                              calcula fluxo a que pretence b1
48                              calcula nó em que b1 deve executar
49                              cria mensagem
50                              inicializa mensagem
51                              empacota mensagem
52                              envia mensagem
53                      }
54                      senão{
55                              b1.i = b.i + 1;
56                              b1.j = b.j;
57                              calcula fluxo a que pretence b1
58                              calcula nó em que b1 deve executar
59                              cria mensagem
60                              inicializa mensagem
61                              empacota mensagem
62                              envia mensagem
63                      }
64              }
65      }
66 }
```

Figura 33: Algoritmo implementado para o cálculo da matriz de similaridades.

capítulos anteriores. É importante destacar que um bloco somente é inserido na lista de tarefas prontas quando todas suas dependências estiverem resolvidas; enquanto isso não ocorrer, o bloco permanece armazenado em uma lista de tarefas bloqueadas.

Depois de preenchido, o bloco irá enviar mensagens de forma a "avisar"aos blocos que dependem dele que ele já está pronto. O trecho de código correspondente a este caso está compreendido entre as linhas 8 e 66.

O trecho que vai da linha 8 até a 35 refere-se a blocos que devem enviar mensagens para outros três blocos. Utilizando-se as fórmulas apresentadas anteriormente, calculam-se o fluxo de execução a que pertence cada bloco e em que nó ele deverá ser executado. Depois disso, envia-se uma mensagem ao nó indicado para que este calcule o bloco que lhe foi enviado. O recebimento de um bloco para ser calculado resulta na execução de uma função onde será verificado se as dependências do bloco já estão resolvidas. Caso estejam, o bloco recebido é inserido na lista de tarefas prontas para serem calculadas. Caso contrário, o bloco permanece em uma lista de tarefas bloqueadas.

Se o bloco que acabou de ser calculado corresponde ao último bloco da matriz (elemento mais inferior à direita), isso significa que todo o restante da matriz já foi calculado, pois todas as dependências do bloco já foram resolvidas antes dele ser lançado para execução. Assim, ao se detectar o término do cálculo do último bloco da matriz, uma mensagem é enviada para todos os nós do aglomerado para que encerrem suas execuções. O trecho da Figura 33 correspondente a essa descrição encontra-se entre as linhas 37 e 42.

Além dos dois casos anteriores, outro que deve ser tratado refere-se ao cálculo de um bloco pertencente à última linha da matriz. Os blocos dessa linha são tratados diferentemente, pois devem enviar apenas uma mensagem para liberação do bloco à sua direita. Com isso, calculam-se o fluxo de execução e o nó onde o bloco será calculado e envia-se a mensagem correspondente ao nó indicado. Essas operações estão indicadas entre as linhas 44 e 53.

Assim como os blocos da última linha são tratados de maneira diferente, os blocos

da última coluna também o são. A diferença neste caso está no fato de um bloco da última linha liberar o bloco que está abaixo dele, ao invés de um à sua direita. Como antes, calculam-se o fluxo de execução e o nó para o qual o bloco será enviado e envia-se a mensagem ao nó correspondente (linhas 54 a 63).

### 7.2.5 Resultados Reais em Memória Distribuída

Os resultados apresentados nesta seção correspondem ao alinhamento de duas seqüências de 100.000 elementos cada uma [1] e foram obtidos a partir de execuções utilizando-se a arquitetura descrita nas páginas 102 e 103. Com isso, originou-se uma matriz de 100.000 × 100.000 elementos (10.000.000.000 de elementos) definidos como caracteres, totalizando uma matriz com 10.000.000.000 de bytes. Esta matriz foi dividida em 20 × 20 blocos (400 blocos), originando blocos de 5.000 × 5.000 elementos (25.000.000 de elementos) cada um. As mensagens trocadas entre os nós têm o tamanho correspondente a um bloco (25.000.000 de bytes) mais dois inteiros identificando as posições $i$ e $j$ do bloco a ser calculado, resultando em 25.000.008 bytes. Os resultados de desempenho para este caso estão apresentados como gráfico na Figura 34.

Observando-se as curvas desta figura, percebe-se que foi possível obter-se ganho de desempenho com a introdução de concorrência tanto em número de processadores virtuais (concorrência intra-nó), quanto em número de nós do aglomerado (concorrência entre-nós). Observa-se, ainda, que a adição de concorrência entre-nós possibilitou ganho considerável de desempenho, com uma redução de cerca de 50% do tempo de processamento quando aumentou-se de 1 para 2 o número de nós.

É importante destacar, ainda, o ganho de desempenho obtido mesmo com um único processador virtual em cada nó. Este ganho foi possível, pois existe, em cada nó, uma segunda *thread* dedicada exclusivamente à comunicação entre os nós - o *daemon* de comunicação. A existência do *daemon* permite que o processador virtual dedique-se exclusivamente ao cálculo do bloco destinado a ele.

[1] As seqüências utilizadas são fictícias, não referindo-se a nenhum organismo específico.

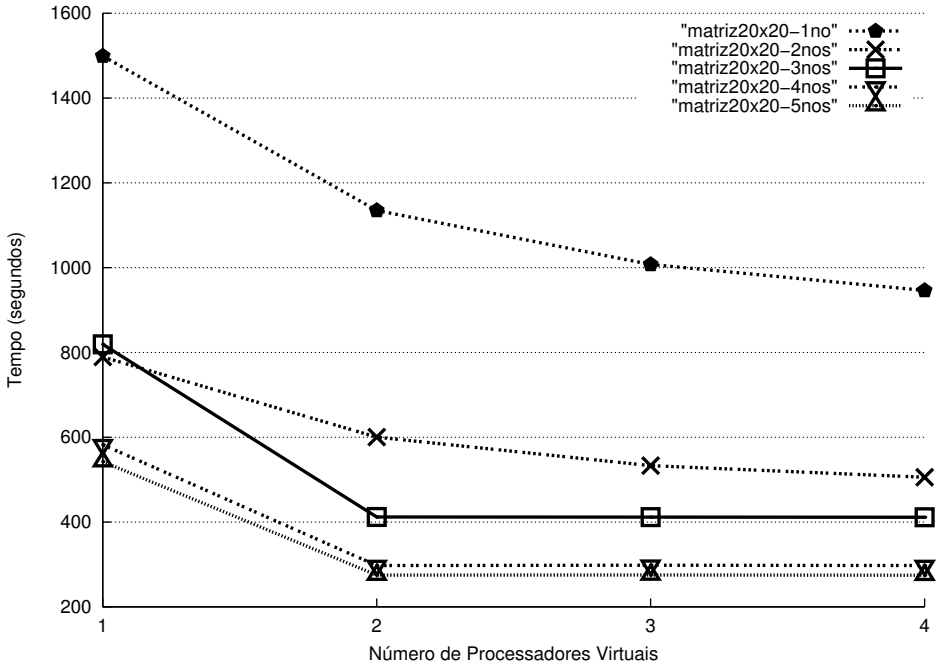


Figura 34: Tempos de execução para uma matriz de 10.000.000.000 de elementos dividida em 400 blocos, com diferentes quantidades de nós e de processadores virtuais.

Por fim, observa-se que, a partir da inclusão do terceiro processador virtual, o ganho deixa de ser considerável, pois os processadores virtuais encontram-se executando as operações de cálculo intensivo do alinhamento.

### 7.2.6 Resultados Sintéticos em Memória Distribuída

Através dos resultados apresentados na seção anterior, observou-se que a aplicação real mostrou-se de grande peso computacional, tanto em tempo de processamento quanto em consumo de memória, dados os tamanhos das seqüências que puderam ser alinhadas. Seqüências completas são bem maiores do que as que puderam ser alinhadas em tempo hábil para coleta de resultados. Assim, questões relativas ao consumo de memória devem possuir tratamento por um mecanismo especial de escalonamento, não estando, no entanto, entre os objetivos deste trabalho.

Para avaliar efetivamente o comportamento da biblioteca de Mensagens Ativas, a mesma estrutura do programa de alinhamento foi utilizada com cargas sintéticas de processamento fazendo as vezes da carga computacional associada ao alinhamento. Duas

cargas de processamento foram utilizadas para uma matriz de tamanho 100.000.000 × 100.000.000, variando o tamanho do bloco comunicado. Os resultados de desempenho para estes casos estão apresentados nos gráficos das figuras 35 até 38.

A Figura 35 apresenta os resultados obtidos quando a matriz foi dividida em 5 × 5 blocos (25 blocos), com 20.000.000 × 20.000.000 de elementos cada um. As mensagens trocadas entre os nós possuem o tamanho de um bloco (400.000.000.000.000 de bytes) mais dois inteiros indicando as posições $i$ e $j$ do bloco, totalizando 400.000.000.008 bytes. Os gráficos apresentados referem-se às duas cargas de processamento utilizadas, com diferentes números de nós e processadores virtuais.

No caso da Figura 36, os gráficos apresentados referem-se aos resultados de desempenho obtidos ao se dividir a matriz em 10 × 10 blocos (100 blocos), com 10.000.000 × 10.000.000 de elementos cada um. Cada mensagem trocada entre os nós possui 100.000.000.000.000 de bytes (o tamanho de um bloco) mais dois inteiros indicando as posições $i$ e $j$ do bloco, o que totaliza 100.000.000.000.008 de bytes. Os gráficos representam os resultados para as duas granulosidades de cálculo utilizadas, com variação do número de nós e de processadores virtuais em cada nó.

Os gráficos apresentados na Figura 37, apresentam os resultados obtidos com a divisão da matriz em 15 × 15 blocos (225 blocos), com 6.666.667 × 6.666.6667 elementos cada. Assim, as mensagens trocadas entre os nós têm 44.444.448.888.889 bytes, referentes ao tamanho de um bloco, mais dois inteiros que indicam as posições $i$ e $j$ do bloco, totalizando 44.444.448.888.897 bytes. Cada um dos gráficos representa os resultados obtidos para uma das cargas computacionais, com diferentes números de nós e de processadores virtuais.

A última figura desta seção, Figura 38, apresenta os gráficos com os resultados obtidos para a divisão da matriz em 20 × 20 blocos (400 blocos). Neste caso, cada bloco possui 5.000.000 × 5.000.000 de elementos (25.000.000 de bytes). As mensagens trocadas entre os nós possuem, assim, 25.000.008 bytes, referentes ao tamanho do bloco mais os

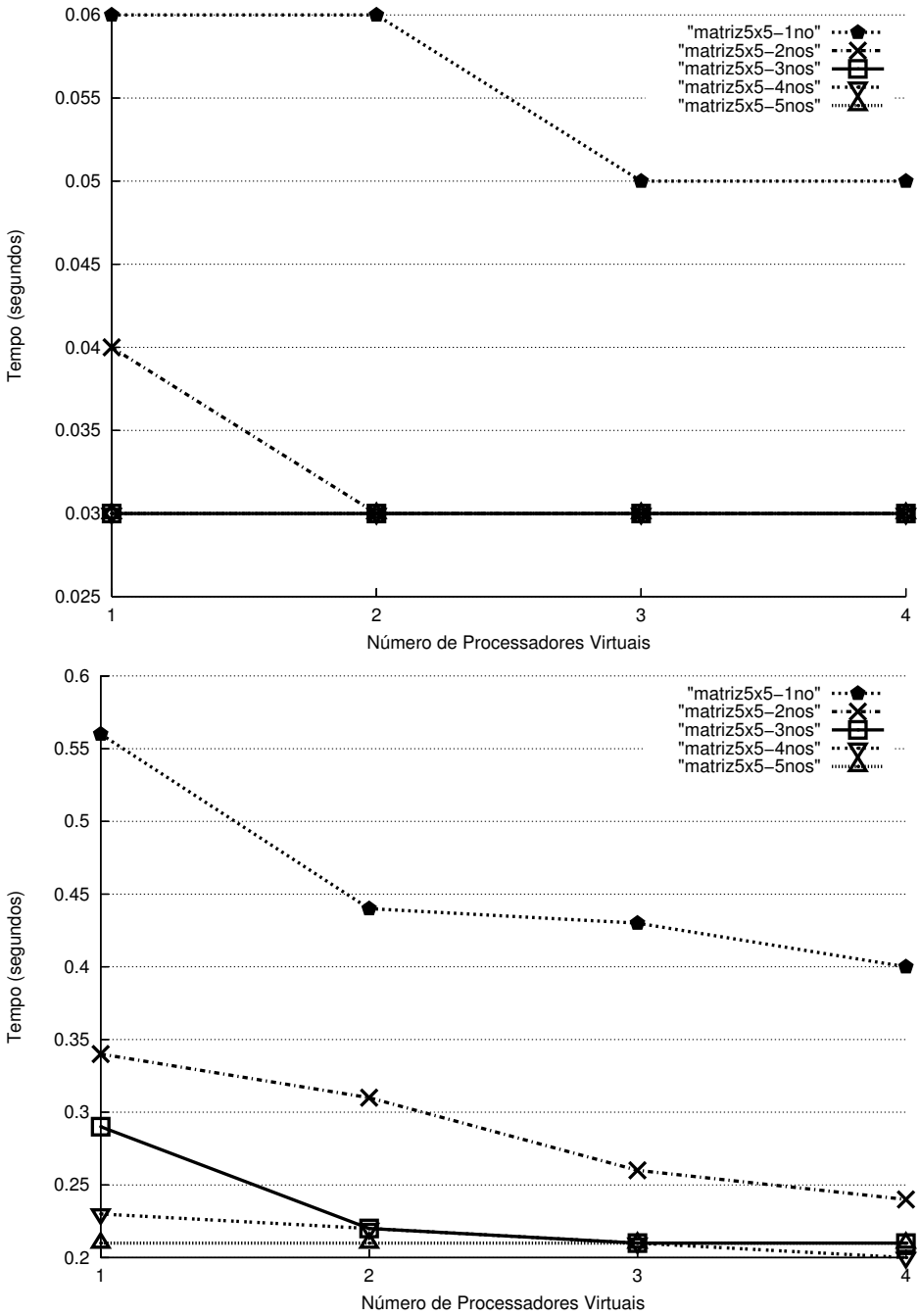


Figura 35: Tempos de execução para uma matriz de 100.000.000 × 100.000.000 de elementos dividida em 25 blocos, com diferentes quantidades de nós e de processadores virtuais. Gráfico superior representa granulosidade média e inferior, granulosidade grossa.

dois inteiros que indicam as posições $i$ e $j$ do bloco. Os gráficos apresentados referem-se às duas granulosidades utilizadas, com variações na quantidade de nós e de processadores virtuais.

Analisando-se os gráficos apresentados nesta seção, observa-se que o comportamento dos mesmos refletiu os resultados de desempenho da aplicação com carga real de

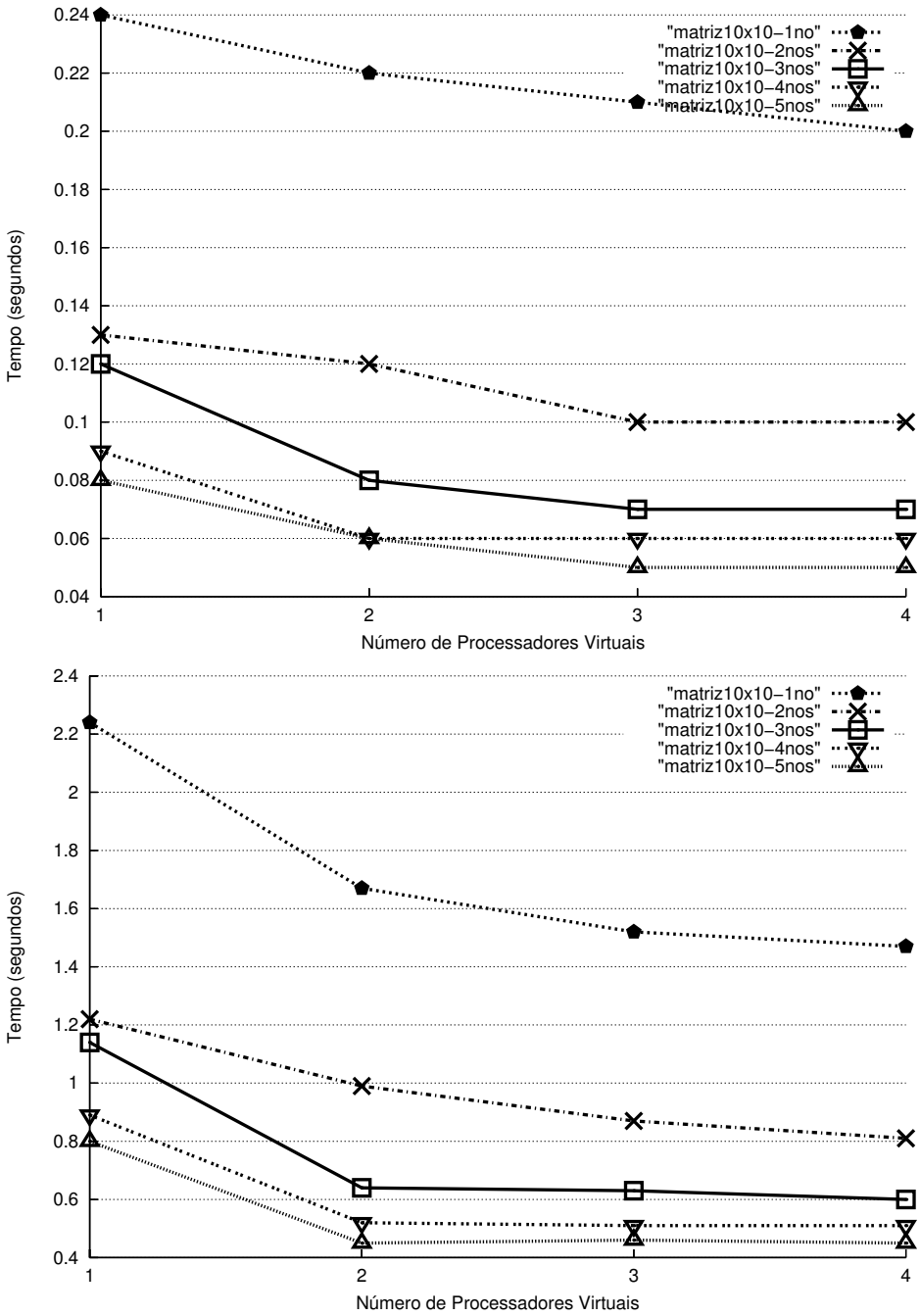


Figura 36: Tempos de execução para uma matriz de 100.000.000 × 100.000.000 de elementos dividida em 100 blocos, com diferentes quantidades de nós e de processadores virtuais. Gráfico superior representa granulosidade média e inferior, granulosidade grossa.

processamento. A introdução de concorrência intra-nó, através do aumento do número de processadores virtuais, quando as aplicações executaram em um único nó, possibilitaram a obtenção de ganho de desempenho. Introduzindo-se concorrência entre-nós (aumentando-se o número de nós), ainda foi possível obter-se ganho. Como antes, a partir da introdução do terceiro processador virtual, o ganho deixou de ser considerável.

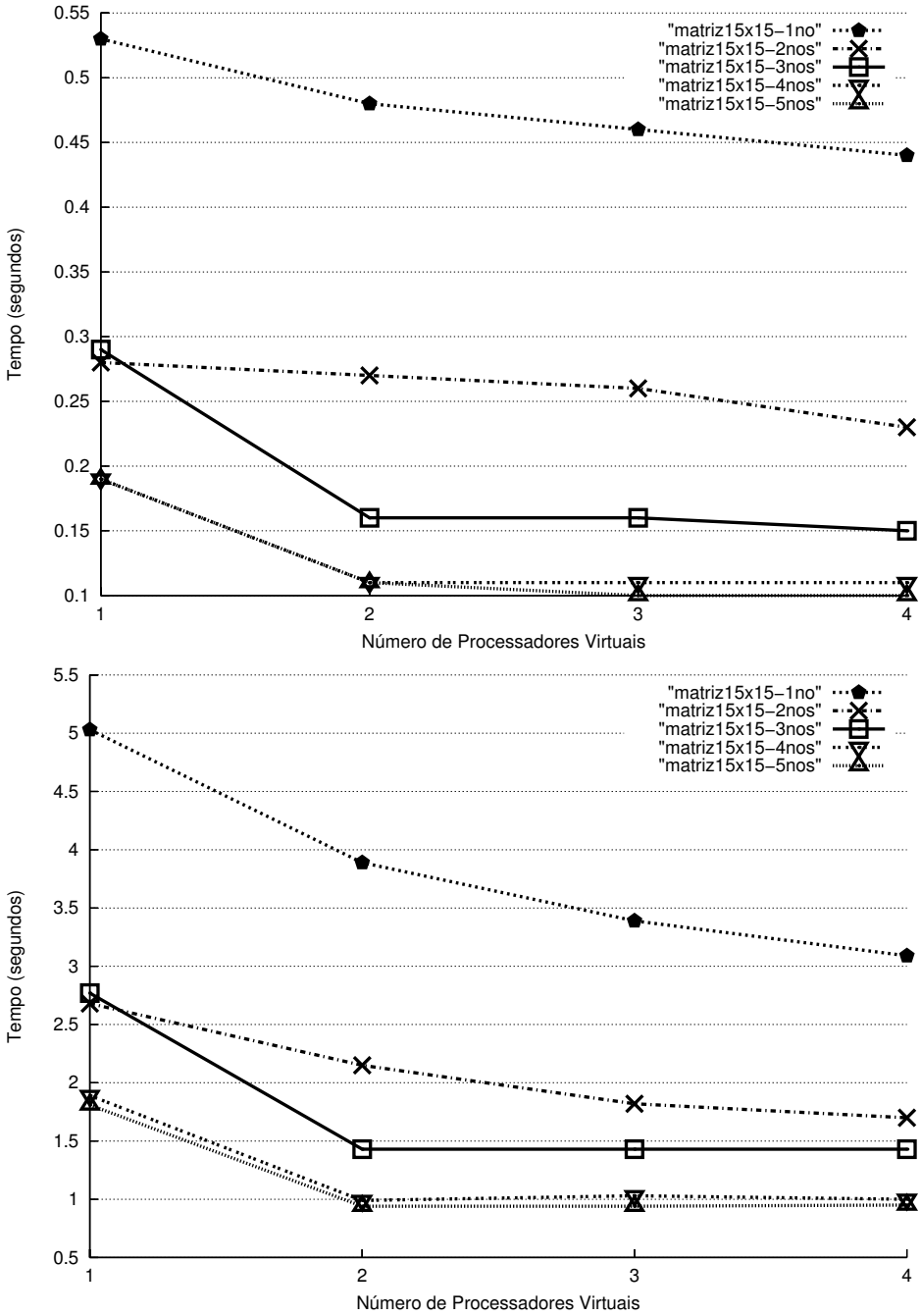


Figura 37: Tempos de execução para uma matriz de 100.000.000 × 100.000.000 de elementos dividida em 225 blocos, com diferentes quantidades de nós e de processadores virtuais. Gráfico superior representa granulosidade média e inferior, granulosidade grossa.

## 7.3 MAPEAMENTO DAS OPERAÇÕES EM SERVIÇOS ANAHY

A interface de programação de Anahy permite a descrição da concorrência de uma aplicação em termos de atividades concorrentes e dependência de dados. Seu núcleo

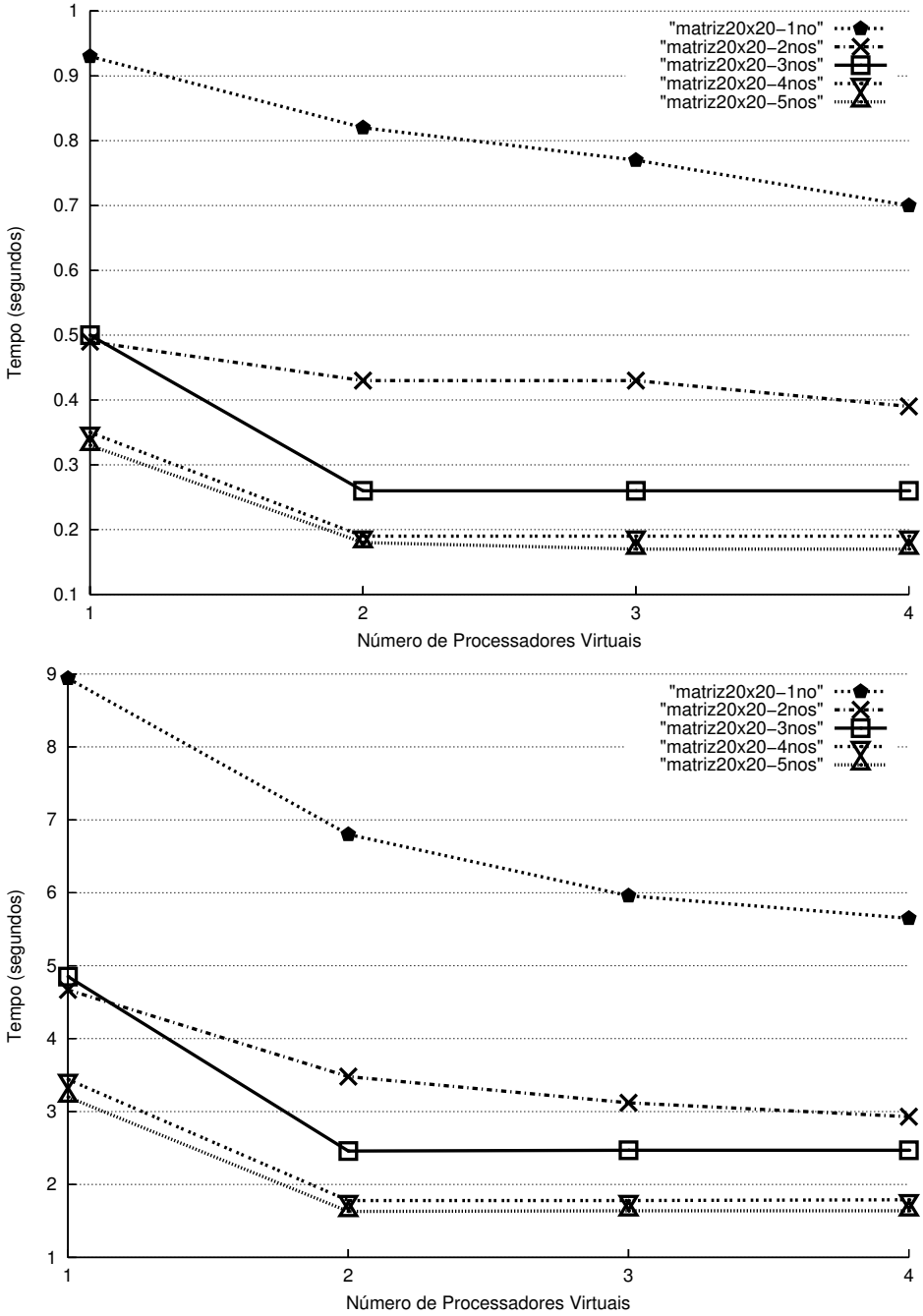


Figura 38: Tempos de execução para uma matriz de 100.000.000 × 100.000.000 de elementos dividida em 400 blocos, com diferentes quantidades de nós e de processadores virtuais. Gráfico superior representa granulosidade média e inferior, granulosidade grossa.

executivo tem a função de construir, em tempo de execução, um grafo de dependências de tarefas e de realizar a execução do programa através do escalonamento destas tarefas. Uma característica particular do algoritmo de escalonamento é explorar as informações do grafo de dependências de forma a otimizar a execução da aplicação considerando as tarefas presentes no seu caminho crítico.

Neste contexto, as aplicações Número de Fibonacci e alinhamento de seqüências biológicas possuem características interessantes para serem exploradas. Os códigos implementados para ambas aplicações produzem grafos com grande número de tarefas, porém com características distintas nas dependências e carga de comunicações, conforme discutido nas avaliações de desempenho apresentadas neste capítulo. No entanto, não estando disponível uma versão para Anahy em ambientes distribuídos, esses códigos não foram escritos utilizando a interface de programação de Anahy e sim sobre os recursos de programação utilizados para implementação do suporte executivo de Anahy: *threads* do padrão POSIX e a própria biblioteca de Mensagens Ativas. O cuidado tomado foi de construir estas aplicações obedecendo as características do modelo de programação e de execução de Anahy.

Versões dos códigos das aplicações utilizando a interface aplicativa de Anahy são apresentadas nas figuras 39 (Fibonacci) e 40 (alinhamento de seqüências). Os códigos são apresentados na sua integralidade no que diz respeito ao tratamento das questões relativas às aplicações, mas referem-se a versões efetivamente construídas ((BENITEZ; CAVALHEIRO, 2004; BENITEZ; MOSCHETTA; CAVALHEIRO, 2004; LERMEN; PERANCONI; CAVALHEIRO, 2004b)) para a versão Anahy-SMP.

O aspecto que chama atenção ao utilizar Anahy é que o tratamento da distribuição de carga (ativação remota de cálculo) e da comunicação não são feitos pelo programador. Cabe ao ambiente realizá-las, em reação as chamadas às operações de criação e sincronização de *threads* Anahy. No entanto, cabe ressaltar que o mecanismo de escalonamento de Anahy foi desenvolvido de forma a otimizar a execução das tarefas no caminho crítico da aplicação em resposta à demanda de dados. Como o grafo somente é conhecido em tempo de execução, o posicionamento das operações de criação e sincronização reflete o desempenho final do programa. Variações neste posicionamento podem alterar o tempo total de execução:

- O cálculo do Número de Fibonacci gera um grafo desbalanceado, uma vez que cada

```
void *fibo( void *n ){
      athread_t t1, t2;
      int n1,  n2, *r1, *r2;

      if( (int *)*n > 2 ) {
            n1 = (int *)*n;
            athread_create( &t1, NULL, fibo, &n1 );
            n2 = (int *)*n;
            athread_create( &t2, NULL, fibo, &n2 );
            athread_join( t1, &r1 );
            athread_join( t2, &r2 );
            *r1 += *r2;
            free((void *) r2;
      }
      else {
            r1  = (int *) malloc(sizeof(int));
            *r1 = 1;
      }
      return r1;
}
```

Figura 39: Cálculo de Fibonacci em Anahy.

*thread* gera duas novas *threads*, uma com peso de trabalho $n-1$ e outra com peso $n-2$. Como o código da Figura 39 realiza primeiro a sincronização com a *thread* `t1` e após com a *thread* `t2`, o mecanismo de execução reage dando preferência para execução das *threads* com maior carga computacional.

- Na computação dos alinhamentos (Figura 40), estratégia semelhante é empregada, buscando-se satisfazer primeiro as dependências diagonais de cada *thread*.

Em se tratando de uma ferramenta de programação, estas variações de comportamento são esperadas. O programador deve conhecer características da ferramenta de forma a construir seu código da maneira que a execução tenha bons índices de eficiência.

## 7.4 COMENTÁRIOS FINAIS

Este capítulo discutiu as principais características das duas aplicações utilizadas para validação da biblioteca de Mensagens Ativas: cálculo do Número de Fibonacci e alinhamento de seqüências. Foram apresentados os grafos de dependências de dados, o

```
struct data_in {
      int i, j, dim; //Identificação do bloco; dim - Tamanho do bloco (dimxdim)
       char *b;   //O bloco de tamanho dim x dim
       athread_t up, left, diag; //Proveniência da entrada
};

void *alinha( void *in ){
       struct data_in *bloco = in;
       char *b_local, *b_diag, *b_up, *b_left;

       b_local = malloc((bloco.dim*bloco.dim)*sizeof(char));

       if( bloco.diag != NULL ) athread_join( bloco.diag, &b_diag );
       if( bloco.up   != NULL ) athread_join( bloco.up,   &b_up );
       if( bloco.left != NULL ) athread_join( bloco.left, &b_left );

       b_local = ... //Recebe cálculo do alinhamento
       return b_local;
}

int main(){
      athread_t m[lins][cols];
      struct data_in bloco;

      bloco.b = ... //Recebe o bloco
      bloco.i = bloco.j = 0; //Primeiro bloco
      bloco.dim = tamanhoDoBloco;
      bloco.up = bloco.left = bloco.diag = NULL; //Sem dependências
      pthread_create( &(m[0][0]), NULL, alinha, bloco);

      //Cria tarefas da primeira linha de blocos da matriz
      for( j = 1 ; j < tamanhoDoBloco ; j++ ){
            bloco.b = ... //Recebe o bloco
            bloco.i = 0; bloco.j = j;
            bloco.dim = tamanhoDoBloco;
            bloco.up = bloco.diag = NULL; //Sem dependências
            bloco.left = m[0][j-1]; //Depende do bloco à esquerda
            pthread_create( &(m[0][j]), NULL, alinha, &bloco);
      }
      //Cria tarefas da primeira coluna de blocos da matriz
      for( i = 1 ; i < tamanhoDoBloco ; i++ )
            bloco.b = ... //Recebe o bloco
            bloco.i = i; bloco.j = 0;
            bloco.dim = tamanhoDoBloco;
            bloco.left = bloco.diag = NULL; //Sem dependências
            bloco.up = m[i-1][0]; //Depende do bloco acima
            pthread_create( &(m[i][0]), NULL, alinha, &bloco);
      }
      //Cria tarefas restantes
      for( i = 2 ; i < tamanhoDoBloco ; i++ )
            for( j = 2 ; j < tamanhoDoBloco ; j++ ) {
                  bloco.b = ... //Recebe o bloco
                  bloco.i = i; bloco.j = j;
                  bloco.dim = tamanhoDoBloco;
                  bloco.left = m[i][j-1];
                  bloco.diag = m[i-1][j-1];
                  bloco.up = m[i-1][j];
                  pthread_create( &(m[i][j]), NULL, alinha, &bloco);
            }
      //Aguarda final de execução (último bloco)
      athread_join( m[tamanhoDoBloco][tamanhoDoBloco], &b );
      return 0;
}
```

Figura 40: Alinhamento de seqüências biológicas em Anahy.

algoritmo e os resultados para ambas as aplicações. Para o alinhamento de seqüências foram apresentados resultados com cargas real e sintéticas de processamento.

Os resultados de desempenho apresentados mostraram que tanto para o Número de Fibonacci quanto para o alinhamento de seqüências, conseguiu-se obter ganho com a introdução das concorrências intra e entre-nós. Para ambas as aplicações, considerável ganho foi obtido quando dobrou-se a concorrência entre-nós, aumentando-se de 1 para 2 o número de nós.

A exploração efetiva destas aplicações em uma versão distribuída de Anahy também foi discutida. Foi ressaltado que, por possuir um núcleo de escalonamento empregando uma estratégia própria, o programador deve tomar cuidados em construir sua aplicação, conduzindo as decisões de escalonamento a privilegiar a execução das tarefas no caminho crítico.

# 8 CONSIDERAÇÕES FINAIS

Esta dissertação apresentou o desenvolvimento de uma biblioteca de Mensagens Ativas e sua utilização como suporte ao desenvolvimento de aplicações em aglomerados de computadores. O uso desta biblioteca foi validado através de sua utilização em uma aplicação para alinhamento de seqüências biológicas e de outra para o cálculo do Número de Fibonacci.

Inicialmente, foram apresentados alguns conceitos básicos envolvendo a Biologia Molecular, com ênfase no problema de alinhamento de seqüências. A utilização do método de programação dinâmica pelos algoritmos de alinhamento apresentou-se como melhor alternativa, pois produz resultados ótimos. Em contrapartida, consome grande quantidade de recursos computacionais no que se refere a tempo de processamento e consumo de memória. Foram destacados alguns trabalhos encontrados na literatura que introduzem concorrência à programação dinâmica de maneira a alinhar seqüências em arquiteturas com memória distribuída. Com isso, pode-se alinhar seqüências grandes em tempo relativamente pequeno.

A programação concorrente foi discutida no Capítulo 3. Destacou-se as dificuldades de desenvolvimento de aplicações capazes de executar segundo este modelo de programação e os níveis de concorrência que podem ser explorados quando se emprega este tipo de programação. Considerando-se um aglomerado de computadores, pode-se explorar tanto a concorrência intra-nó quanto a entre-nós, sendo que, para tirar o máximo proveito dos recursos de processamento deste tipo de arquitetura, o ideal é empregar ferramentas capazes de explorar os dois níveis de concorrência simultaneamente.

Como ferramenta para exploração da concorrência intra-nó, destacou-se a multiprogramação leve, onde múltiplos fluxos de execução compartilham os recursos de processamento de um único nó. A concorrência entre-nós pode ser explorada através da execução simultânea de fluxos de execução em diferentes nós do aglomerado. Eventualmente, fluxos executando em nós diferentes necessitam comunicar-se para troca de dados e tarefas. Para possibilitar a comunicação em aglomerados, foram destacadas algumas ferramentas, como troca de mensagens, chamada remota de procedimento e Mensagens Ativas. Detectou-se nas Mensagens Ativas a melhor solução quando se deseja obter alto desempenho em arquiteturas com memória distribuída.

O Capítulo 4 foi destinado ao detalhamento da ferramenta de Mensagens Ativas, com sua conceituação e apresentação de quatro diferentes implementações. Dessas quatro implementações, somente uma mostrou-se satisfatória quando se trata de aplicações altamente paralelas, o modelo de fila de execução. Foram apresentados, também, alguns ambientes de programação paralela e distribuída que empregam o mecanismo de Mensagens Ativas como suporte à comunicação em arquiteturas com memória distribuída.

No Capítulo 5 foi apresentado o ambiente de programação Anahy. Foram destacados os componentes deste ambiente, enfatizando as principais contribuições do mesmo como ambiente de programação paralela e distribuída. Uma das facilidades introduzidas por Anahy é permitir que o programador descreva a concorrência de sua aplicação de forma independente dos recursos computacionais disponíveis na arquitetura. Além disso, Anahy também garante a ordem de execução das tarefas da aplicação, de maneira que a atribuição das tarefas aos nós de processamento, a criação e sincronização de tarefas e o controle de acesso aos dados compartilhados, sejam atribuições do ambiente.

O Capítulo 6 apresentou a camada desenvolvida para suporte de Anahy em aglomerados de computadores. Esta camada foi concebida sob a forma de uma biblioteca que emprega *multithreading* e Mensagens Ativas para exploração eficiente dos recursos de processamento disponíveis em um aglomerado. A biblioteca foi implementada como um

módulo independente de Anahy, estando disponíveis ao usuário um conjunto de primitivas de comunicação para serem empregadas para trocas de dados e tarefas entre os nós do aglomerado. Apesar de independente, a biblioteca foi desenvolvida respeitando os requisitos de Anahy tanto de implementação, utilizando software livre, portabilidade, entre outros, quanto da vocação de Anahy para processamento de alto desempenho. Porém, a biblioteca deixa ao encargo do programador o escalonamento de tarefas, o controle da ordem de execução de tarefas e o controle de acesso aos dados compartilhados. Foram, ainda, descritos os serviços que necessitam ser implementados para possibilitar a troca de dados e tarefas entre os nós do aglomerado, de maneira a viabilizar a utilização de Anahy em arquiteturas com memória distribuída.

Uma análise de desempenho utilizando a biblioteca desenvolvida foi realizada no Capítulo 7. A primeira aplicação implementada consistiu do cálculo do Número de Fibonacci. Esta é uma aplicação com pouca carga computacional, mas com grande número de trocas de dados, tarefas e sincronizações. A segunda aplicação foi o alinhamento de seqüências biológicas. Foram destacadas algumas características desta aplicação e apresentada a estratégia de distribuição de tarefas entre os nós adotada neste trabalho. Para esta aplicação foram utilizadas dois tipos de carga de processamento, chamadas de real e sintética. A carga real confirmou que o alinhamento é, de fato, muito pesado, conforme discutido no Capítulo 2. A carga sintética auxiliou na avaliação efetiva do comportamento da biblioteca implementada. Tanto os resultados de desempenho obtidos com o cálculo do Número de Fibonacci quanto os obtidos com o alinhamento de seqüências, demonstraram-se satisfatórios utilizando a biblioteca desenvolvida. Índices de desempenho mais significativos foram obtidos quando dobrou-se a concorrência entre-nós, aumentado-se de 1 para 2 o número de nós de processamento. Para as duas aplicações o tempo de processamento reduziu cerca de 50%.

Trabalhos para continuação da presente dissertação a serem desenvolvidos em curto prazo, referem-se à extensão da interface de programação de Anahy, de forma a viabilizar a transferência (transparentemente ao usuário) de *threads* Anahy entre diferentes nós de

um aglomerado, e à introdução efetiva da biblioteca de Mensagens Ativas no ambiente Anahy. Trabalhos a serem desenvolvidos em médio/longo prazo, destinam-se ao desenvolvimento de estratégias de escalonamento que possibilitem alinhar grandes seqüências e à viabilização do alinhamento de grandes seqüências utilizando todo o potencial de Anahy para a exploração de alto desempenho.

# REFERÊNCIAS

ALMEIDA JR., N. F. *Tools for genome comparison*. Tese (Doutorado) — IC/Universidade de Campinas, Campinas, SP, Brasil, maio 2003.

ALMEIDA JR., N. F. et al. Comparison of genomes using high-performance parallel computing. In: *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2003)*. São Paulo, SP, Brazil: Brazilian Computer Society (SBC) and IEEE Computer Society, 2003. p. 142–148.

ALTSCHUL, S. F. et al. Basic local alignment search tool. *Journal of Molecular Biology*, v. 215, p. 403–410, 1990.

ALVES, C. E. R. et al. A parallel wavefront algorithm for efficient biological sequence comparison. In: *Proceedings of the International Conference on Computational Science and its Applications (ICCSA 2003)*. Montreal, Canada: [s.n.], 2003. Lecture Notes in Computer Science, volume 2668, pp. 249–258, 2003.

AMERICAN NATIONAL STANDARDS INSTITUTE. *IEEE standard for information technology: Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension [C language]*. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1994. xxiii + 590 p. IEEE Std 1003.1b-1993 (formerly known as IEEE P1003.4; includes IEEE Std 1003.1-1990). Approved September 15, 1993, IEEE Standards Board. Approved April 14, 1994, American National Standards Institute. ISBN 1-55937-375-X.

BARCELLOS, A. M. P.; GASPARY, L. P. Tecnologias de rede para processamento de alto desempenho. In: $3^a$ *Escola Regional de Alto Desempenho*. Santa Maria - Rio Grande do Sul - Brasil: SBC, 2003. p. 67–102.

BENITEZ, E. D.; CAVALHEIRO, G. G. H. Análise comparativa do uso de mpi e sockets aplicados na convolução de imagens. In: $4^a$ *Escola Regional de Alto Desempenho*. Pelotas - Rio Grande do Sul - Brasil: SBC, 2004. p. 321–324. ISBN 85-88442-74-4.

BENITEZ, E. D. et al. Avaliação de desempenho de anahy em aplicações paralelas. In: *Anais do III Workshop em Desempenho de Sistemas Computacionais e de Comunicação (em CD)*. Salvador, Brasil: [s.n.], 2004.

BENITEZ, E. D.; MOSCHETTA, E.; CAVALHEIRO, O. C. C. G. G. H. Modelo para a exploração eficiente de paralelismo em aplicações grão fino. In: *V Workshop em Sistemas Computacionais de Alto Desempenho*. Foz do Iguaçú, Brasil: SBC, 2004. p. 11–18.

BIRRELL, A.; NELSON, B. Implementing remote procedure calls. *ACM Transactions on Computers Systems*, v. 2, n. 1, p. 39–59, feb 1984.

BLACK, D. L. Scheduling support for concurrency and parallelism in the Mach operating system. *Computer*, v. 23, n. 5, p. 35–43, maio 1990. ISSN 0018-9162.

BLUMOFE, R. D. et al. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, v. 30, n. 8, p. 207–216, ago. 1995. ISSN 0362-1340.

BRIAT, J.; GINZBURG, I.; PASIN, M. Athapascan-0b: un noyau exécutif parallèle. *Lettre du Calculateur Parallèle*, v. 10, n. 3, p. 273–293, 1998.

BUYYA, R. *High performance cluster computing: systems and architectures*. [S.l.]: Prentice Hall PTR, 1999.

CARISSIMI, A. S. *Athapascan-0: exploitation de la multiprogrammation légère sur grappes de multiprocesseurs*. Tese (Doutorado) — Institut National Polytechnique de Grenoble, Grenoble, France, set. 1999.

CAVALHEIRO, G. G. H. A general scheduling framework for parallel execution environments. In: *Proceedings of First IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2001)*. Brisbane, Australia: IEEE Computer Society, 2001.

CAVALHEIRO, G. G. H. Princípios da programação concorrente. In: $4^a$ *Escola Regional de Alto Desempenho*. Pelotas - Rio Grande do Sul - Brasil: SBC, 2004. p. 3–39. ISBN 85-88442-74-4.

CAVALHEIRO, G. G. H.; DALL'AGNOL, E. C.; VILLA REAL, L. C. Uma biblioteca de processos leves para a implementação de aplicações altamente paralelas. In: *IV Workshop em Sistemas Computacionais de Alto Desempenho*. São Paulo - Brasil: SBC, 2003. p. 117–124.

CAVALHEIRO, G. G. H.; DENNEULIN, Y.; ROCH, J.-L. A general modular specification for distributed schedulers. *Lecture Notes in Computer Science*, v. 1470, p. 373–377, 1998. ISSN 0302-9743.

CAVALHEIRO, G. G. H.; GARZÃO, A. S.; VILLA REAL, L. C. Ferramentas para desenvolvimento de um ambiente de programação sobre agregados. In: *Workshop sobre Software Livre*. Porto Alegre - Brasil: [s.n.], 2001. v. 1, p. 36–39.

CHANG, C.-C. et al. Low-latency communication on the ibm risc system/6000 sp. In: *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (CDROM)*. Pittsburgh, Pennsylvania, United States: ACM Press, 1996. p. 24.

CORDEIRO, O. C. et al. Exploiting multithreaded programming on cluster architectures. In: *Inernational Symposium on High Performance Computing Systems and Applications*. [S.l.: s.n.], 2005.

COSTA, C. M. da; STRINGHINI, D.; CAVALHEIRO, G. G. H. Programação concorrente: Threads, mpi e pvm. In: $2^a$ *Escola Regional de Alto Desempenho*. São Leopoldo - Rio Grande do Sul - Brasil: SBC, 2002. p. 31–65. ISBN 85-88442-16-7.

CULLER, D. E. et al. Parallel programming in split-c. In: *Supercomputing*. [s.n.], 1993. p. 262–273. Disponível em: <citeseer.ist.psu.edu/culler93parallel.html>.

DALL'AGNOL, E. C. et al. Construção de um mecanismo de comunicação para ambientes de processamento de alto desempenho. In: *Anais do V Workshop em Sistemas Computacionais de Alto Desempenho*. Foz do Iguaçú, Brasil: [s.n.], 2004. p. 169–175.

DALL'AGNOL, E. C. et al. Introdução de mensagens ativas em um ambiente para processamento de alto desempenho. In: *Anais da V Escola Regional de Alto Desempenho*. Canoas, Brasil: [s.n.], 2005. p. 185–188.

DALL'AGNOL, E. C. et al. Portabilidade na programação para o processamento de alto desempenho. In: *IV Workshop em Sistemas Computacionais de Alto Desempenho*. São Paulo - Brasil: SBC, 2003. p. 141–148.

DAYHOFF, M. O.; SCHWARTZ, R. M.; ORCUTT, B. C. A model of evolutionary change in proteins. In: DAYHOFF, M. O. (Ed.). *Atlas of Protein Structure*. Silver Spring, Md.: National Biomedical Reasearch Foundataion, 1979. v. 5(Suppl. 3), p. 345–352.

DEHNE, F. K. H. A.; FABRI, A.; RAU-CHAPLIN, A. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal of Computational Geometry and Applications*, v. 6, n. 3, p. 379–400, 1996.

DEITEL, H. M.; DEITEL, P. J. *Java: como programar*. Porto Alegre: Bookman, 2000.

DENNEULIN, Y.; NAMYST, R.; MÉHAUT, J.-F. Architecture virtualization with mobile threads. In: *Parallel Computing: Fundamentals, applications and new directions (ParCo '97)*. [S.l.]: Elsevier Science Publishers, 1997. (Advances in parallel computing, 12), p. 477–484.

EICKEN, T. von et al. Active messages: a mechanism for integrated communication and computation. In: *Proceedings the 19th Annual International Symposium on Computer Architecture,ACM SIGARCH*. Gold Coast, Australia: [s.n.], 1992. v. 20, n. 5, p. 256–266.

FORUM, M. MPI : A message - passing interface standard. *The International Journal of Supercomputing and High Performance Computing*, p. 159–416, 1994.

FOSTER, I. et al. *Nexus: An Interoperability toolkit for parallel and distributed computer systems*. Argonne, 1993.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Nexus task-parallel runtime system. In: *Proc. 1st Intl Workshop on Parallel Processing*. [S.l.]: Tata McGraw Hill, 1994. p. 457–462.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, v. 37, n. 1, p. 70–82, ago. 1996. ISSN 0743-7315.

GALLILÉE, F. et al. Athapascan-1: On-line building data flow graph in a parallel language. In: *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*. Paris: IEEE Computer Society Press, 1998. p. 88–95.

GALPER, A. R.; BRUTLAG, D. L. *Parallel Similarity Search and Alignment with the Dynamic Programming Method*. [S.l.], abr. 1990.

GEIST, A. et al. *PVM: Parallel Virtual Machine*. Cambridge, Massachussetts: MIT Press, 1994.

GINZBURG, I. *Athapascan-0b: intégration efficace et portable de multiprogrammation légère et de communication*. Tese (Doutorado) — Institut National Polytechnique de Grenoble, Grenoble, France, 1997.

GOLDMAN, A. Modelos para a computação paralela. In: $3^a$ *Escola Regional de Alto Desempenho*. Santa Maria - Rio Grande do Sul - Brasil: SBC, 2003. p. 35–66. ISBN 85-88442-44-2.

GRAHAM, R.; KNUTH, D.; PATASHNIK, O. *Concrete Mathematics*. [S.l.]: Addison-Wesley, 1989.

GUSFIELD, D. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. [S.l.]: CUP, 1997.

HAINES, M.; CRONK, D.; MEHROTRA, P. On the design of Chant: A talking threads package. In: ACM/IEEE. *Proceedings of Supercomputing*. Washington D.C., 1994. p. 350–359.

HALL, P. A. V.; DOWLING, G. R. Approximate string matching. *ACM Computing Surveys*, v. 12, n. 4, p. 381–402, 1980.

HENIKOFF, S.; HENIKOFF, J. Amino acid substitution matrices from protein blocks. In: *Proceedings of the National Academy of Sciences of the USA*. [S.l.: s.n.], 1991. p. 6565–6572.

HUNT, J. W.; SZYMANSKY, T. An algorithm for differential file comparison. *Communications of the ACM*, v. 20, p. 350–353, 1977.

IEEE. *IEEE 1003.1c-1994: Standard for Information Technology — Portable Operating System Interfaces (POSIX) — Part 1: System Application Program Interface (API ) — Amendment 2: Threads Extension [C language]*. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1994.

ISHIKAWA, M. et al. Multiple sequence alignment by parallel simulated annealing. *Comp. Appl. BioSci*, v. 9, n. 3, p. 267–273, 1993. TR-730, Institute for New Generation Computing (ICOT), 1992.

KIM, J.; COLE, J. Alignment of possible secondary structures in multiple rna sequences using simulated annealing. *Comput. Applic. BioSci*, v. 12, p. 259–267, 1993.

LAU, F. *An Integrated Approach to Fast, Sensitive, and Cost-Effective Smith-Waterman Multiple Sequence Alignment*. 2000. `http://bioinfo.mbb.yale.edu/mbb452a/2000/projects/Frank--Lau.pdf`. Acesso em fevereiro, 2004.

LECOMPTE, O. et al. Multiple alignment of complete sequences (macs) in the post-genomic era. *Gene*, v. 270, p. 17–30, 2001.

LERMEN, G.; PERANCONI, D. S.; CAVALHEIRO, G. G. H. Alinhamento de seqüências de dna em ambientes com memória compartilhada. In: *Anais do V Simpósio de Informática do Planalto Médio (em CD)*. Passo Fundo, Brasil: [s.n.], 2004.

LERMEN, G.; PERANCONI, D. S.; CAVALHEIRO, G. G. H. Framework para alinhamento de seqüências biológicas com o auxílio de programação concorrente. In: *Anais do V Workshop em Sistemas Computacionais de Alto Desempenho*. Foz do Iguaçú, Brasil: [s.n.], 2004. p. 121–129.

LERMEN, G.; PERANCONI, D. S.; CAVALHEIRO, G. G. H. Implementação paralela do algoritmo smith-waterman utilizando threads posix. In: *Anais da 4ª Escola Regional de Alto Desempenho*. Pelotas, Brasil: [s.n.], 2004. p. 185–188.

LEVIN, E. Grand challenges to computational science. *Communications of the ACM*, v. 32, n. 12, p. 1456–1457, dez. 1989.

LUMETTA, S. S.; MAINWARING, A.; CULLER, D. E. Multi-protocol active messages on a cluster of SMPs. In: *Proceedings of Supercomputing'97 (CD-ROM)*. San Jose, CA: ACM SIGARCH and IEEE, 1997. University of California, Berkeley.

MARTINS, W. S. et al. Whole genome alignment using a multithreaded parallel implementation. In: *Proceedings of the Symposium on Computer Architecture and High Performance Computing*. Pirenópolis, Brazil: [s.n.], 2001. p. 1–8.

MARTINS, W. S. et al. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In: *Proceedings of the Pacific Symposium on Biocomputing*. Big Island of Hawaii: [s.n.], 2001. p. 311–322.

MOUNT, D. W. *Bioinformatics: Sequence and Genome Analysis*. [S.l.]: Cold Spring Harbor Laboratory Press, 2001.

NAVAUX, P. O. A. Execução de aplicações em ambientes concorrentes. In: 1ª *Escola Regional de Alto Desempenho*. Gramado - Rio Grande do Sul - Brasil: SBC, 2001. p. 179–193.

NEEDLEMAN, S. B.; WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, v. 48, p. 443–453, 1970.

NOTREDAME, C.; HIGGINS, D. G. Saga: sequence alignment by genetic algorithm. *Nucleic Acids Research*, v. 24, n. 8, p. 1515–1524, 1996.

PASIN, M.; KREUTZ, D. L. Arquitetura e administração de aglomerados. In: 3ª *Escola Regional de Alto Desempenho*. Santa Maria - Rio Grande do Sul - Brasil: SBC, 2003. p. 3–34. ISBN 85-88442-44-2.

PEARSON, W. R.; LIPMAN, D. J. Improved tools for biological sequence comparison. In: *Proc. Natl. Acad. Sci (USA)*. [S.l.: s.n.], 1988. p. 2444–2448. Published as Proc. Natl. Acad. Sci (USA)., volume 85, number 8.

PERANCONI, D. S.; CAVALHEIRO, G. G. H. Seqüenciamento de DNA em arquiteturas com memória distribuída. In: 4ª *Escola Regional de Alto Desempenho*. Pelotas - Rio Grande do Sul - Brasil: SBC, 2004. p. 167–168. ISBN 85-88442-74-4.

PERANCONI, D. S.; CAVALHEIRO, G. G. H. Alinhamento de seqüências de DNA em aglomerados de computadores. In: 5ª *Escola Regional de Alto Desempenho*. Canoas - Rio Grande do Sul - Brasil: SBC, 2005. p. 107–108.

PEVZNER, P. A. *Computational molecular biology: an algorithmic approach*. [S.l.]: MIT Press, 2001.

PRICE, G. W.; LOWENTHAL, D. K. A comparative analysis of fine-grain threads packages. *J. Parallel Distrib. Comput.*, Academic Press, Inc., v. 63, n. 11, p. 1050–1063, 2003. ISSN 0743-7315.

RINARD, M. C.; LAM, M. S. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, v. 20, n. 3, p. 483–545, maio 1998. ISSN 0164-0925. Disponível em: <`http://www.acm.org:80/pubs/citations/journals/toplas/1998-20-3/p483-%rinard/`>.

ROCHA, E. *Módulo de BioInformática - Análise de seqüências*. 2004. `http://labbi.uesc.br/apostilas`. Acesso em fevereiro, 2004.

ROLOFF, E.; CARISSIMI, A. S.; CAVALHEIRO, G. G. H. Variações de mensagens ativas para aglomerados de computadores. In: $4^a$ *Escola Regional de Alto Desempenho*. Pelotas - Rio Grande do Sul - Brasil: SBC, 2004. p. 289–292. ISBN 85-88442-74-4.

SANKOFF, D. Minimal mutation trees of sequences. *SIAM Journal of Applied Mathematics*, v. 28, p. 35–42, 1975.

SCHMIDT, B.; SCHRÖDER, H.; SCHIMMLER, M. Massively parallel solutions for molecular sequence analysis. In: *16th International Parallel and Distributed Processing Symposium (IPDPS '02 (IPPS & SPDP))*. Washington - Brussels - Tokyo: IEEE, 2002. p. 186–186. ISBN 0-7695-1573-8.

SEBESTA, R. W. *Conceitos de linguagens de programação*. Porto Alegre: Bookman, 2000.

SELLERS, P. H. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, v. 1, p. 359–373, 1980.

SETUBAL, J.; MEIDANIS, J. *Introduction to Computational Molecular Biology*. [S.l.]: Brooks-Cole, 1997. 320 p.

SILBERSCHATZ, A.; GALVIN, P. B. *Operating System Concepts*. New York: John Wiley & Sons, Inc., 1997. ISBN 0-471-36414-2.

SILVA, A. C. R. da et al. Comparison of the genomes of two xanthomonas pathogens with differing host specifications. *Nature*, v. 417, n. 6887, p. 459–463, maio 2002.

SKILLICORN, D. *Foundations of Parallel Programming*. [S.l.]: Cambridge, Great Britain, 1994.

SMITH, T.; WATERMAN, M. Identification of common molecular subsequences. *Journal of Molecular Biology*, v. 147, p. 195–197, 1981.

STEVENS, W. R. *UNIX Network Programming - Networking APIs: Sockets and XTI*. 2. ed. Upper Saddle River, NJ: Prentice Hall PTR, 1998.

TANENBAUM, A. S. *Modern Operating Systems*. Englewood Cliff, NJ: Prentice Hall, 1992. ISBN 0-13-588187-0.

THEOBALD, K. B. Definition of the EARTH model. out. 07 1999. Disponível em: <http://citeseer.ist.psu.edu/271556.html; ftp://ftp.capsl.udel.edu/pub/doc/memos/memo033.ps.gz>.

TICONA, W. G. C. *Aplicação de Algoritmos Genéticos Multi-Objetivo para Alinhamento de Seqüências Biológicas*. Dissertação (Mestrado) — ICMC-USP, São Carlos, Brazil, fev. 2003.

VAHALIA, U. *UNIX Internals*. Englewood Cliffs: Prentice Hall, 1976.

VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM*, v. 33, n. 8, p. 103–111, ago. 1990.

WALLACH, D. A. et al. Optimistic active messages: a mechanism for scheduling communication with computation. *ACM SIGPLAN Notices*, v. 30, n. 8, p. 217–226, ago. 1995. ISSN 0362-1340.

WILKINSON, B.; ALLEN, M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, New Jersey: Prentice-Hall, 1999.

WU, S.; MANBER, C. Fast text searching allowing errors. *Communication of the ACM*, v. 35, p. 83–91, 1992.

YANG, B. H. W. *A Parallel Implementation of Smith-Waterman Sequence Comparison Algorithm*. 2002. `http://cmgm.stanford.edu/biochem218/Projects\%202002/Byang.pdf`. Acesso em fevereiro, 2004.

YAP, T. K.; FRIEDER, O.; MARTINO, R. L. Parallel computation in biological sequence analysis. *IEEETPDS: IEEE Transactions on Parallel and Distributed Systems*, v. 9, n. 3, p. 283–294, 1998.

ZAHA, A.; FERREIRA, H. B.; PASSAGLIA, L. M. P. *Biologia Molecular Básica*. [S.l.]: Mercado Aberto, 2003. 424 p.

ZALIZ, R. R. *Reconocimiento y descripción de objetos complejos em biologia molecular*. Dissertação (Mestrado) — Departamento de Computación, Universidad de Buenos Aires, Buenos Aires - Argentina, 2001.

ZHANG, C.; WONG, A. K. C. A genetic algorithm for multiple molecular sequence alignment. *Comput. Applic. Biosci*, v. 13, n. 6, p. 565–581, 1997.