UM AMBIENTE PARA DESENVOLVIMENTO DE ALGORITMOS DISTRIBUÍDOS

Adriana Magalhães de Barros

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Mario Roberto Folhadela Benevides, Ph.D.

Prof. Lúcia Maria de Assumpção Drummond, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO DE 1999

BARROS, ADRIANA MAGALHÃES DE

Um ambiente para desenvolvimento de algoritmos distribuídos [Rio de Janeiro] 1999

XII, 128 pp., 29.7 cm, (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1999)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 Um ambiente para desenvolvimento de algoritmos distribuídos

2 Algoritmos Distribuídos

I. COPPE/UFRJ II. Título (série)

Aos Meus Pais e Irmãos

Agradecimentos

Agradeço à Deus inicialmente pela oportunidade de me dar existência e em segundo lugar por ter permitido que eu concluísse este trabalho.

Agradeço aos meus pais, Manuel e Lucinda, pelo amor, carinho e força que me oferecem sempre, e principalmente durante todo o percurso deste trabalho, como também pela compreensão pelos vários momentos em que tive de me ausentar das reuniões familiares por questões acadêmicas.

Agradeço aos meus irmãos, Iria e Nelson Luis, pelo carinho, voto de coragem, compreensão e incentivo.

O meu agradecimento muito especial ao orientador deste trabalho, Prof. Valmir, sem o qual este trabalho não existiria. Agradeço também pela compreensão e pelas palavras sábias, acima de tudo encorajadoras, nos momentos de dificuldades da realização deste trabalho.

Agradeço a todos os meus amigos da COPPE, incluindo aqui todos os professores, alunos e funcionários, que de alguma forma contribuíram para a concretização deste trabalho. Dentre eles posso citar Lúcia, Guto, Lúcio, Adriana e Eduardo, pelos longos dias e noites de estudo; Prof. Cláudio Esperança, Gabriel e Júlio pelo suporte na implementação desta tese.

Agradeço aos meus amigos da UERJ, Cláudia, Antônio Carlos, Sérgio e Weber, pelas palavras de incentivo, desde o início deste trabalho, principalmente nos momentos finais.

Agradeço ao amigo Adriano pela contribuição na implementação da interface gráfica, assim como pelas palavras sempre de incentivo.

Enfim, agradeço a todos que direta ou indiretamente colaboraram durante todo o desenvolvimento deste trabalho.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UM AMBIENTE PARA DESENVOLVIMENTO DE ALGORITMOS DISTRIBUÍDOS

Adriana Magalhães de Barros

Setembro/1999

Orientador : Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

Esta tese apresenta uma ferramenta para desenvolvimento de algoritmos distribuídos síncronos e assíncronos. Os algoritmos a serem investigados devem estar escritos segundo o modelo para descrição de algoritmos distribuídos apresentado no livro "An Introduction to Distributed Algorithms" de Valmir C. Barbosa. O ambiente foi desenvolvido utilizando o padrão para passagem de mensagens MPI (Message Passing Interface) e a linguagem de programação C. A interface gráfica com o usuário foi desenvolvida para a Internet, utilizando o padrão HTML (Hypertext Markup Language) e CGI (Common Gateway Interface). O ambiente implementado permite que o usuário realize várias experimentações, variando o número de nós, a topologia, o número de processadores e a capacidade do buffer de mensagens, e tem como resultado um programa executável que realiza a execução do algoritmo informado. No caso de algoritmos distribuídos síncronos, para gerar o programa, foi utilizado um sincronizador para converter o mesmo em um algoritmo assíncrono. Para realizar esta conversão, o usuário poderá indicar um dos dois tipos de sincronizadores implementados: Alfa ou Beta.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

AN ENVIRONMENT TO DEVELOPMENT OF DISTRIBUTED ALGORITHMS

Adriana Magalhães de Barros

September/1999

Advisor: Valmir Carneiro Barbosa

Department: Systems Engineering and Computer Science

This thesis presents a tool for the development of asynchronous and synchronous distributed algorithms. The algorithms to be investigated must be written according to the template given in "An Introduction to Distributed Algorithms" by Valmir C. Barbosa. The environment was developed using the MPI (Message Passing Interface) standard for message-passing and the C programming language. The user interface was developed for the Internet, using the HTML (Hypertext Markup Language) standard and CGI (Common Gateway Interface). The environment allows the user to perform several experiments varying the number of nodes, the topology, the number of processors and the size of message buffers, and yields as a result an executable program that runs the algorithm under development. In the case of synchronous distributed algorithm, a synchronizer is used to convert it into an asynchronous distributed algorithm. In order to perform this conversion, the user may indicate one of the two implemented synchronizers: Alpha or Beta.

Sumário

1 Introdução **1**

1.1 Objetivos 1

1.2 Metodologia 2

1.2.1 World-Wide Web e Internet 3

1.2.2 HTML 4

1.2.3 CGI 5

1.3 Características do Ambiente 5

1.3.1 Interface Gráfica 5

1.3.2 Geração do Programa 6

1.4 Organização da Tese 7

2 Algoritmos Distribuídos **8**

2.1 Programas Orientados a Passagem de Mensagens 9

2.2 Algoritmos Assíncronos e Síncronos 12

2.2.1 Modelo Assíncrono 13

2.2.2 Modelo Síncrono 15

2.3 Algoritmos Sincronizadores 17

2.3.1 Sincronizador Alfa 18

2.3.2 Sincronizador Beta 21

2.3.3 Sincronizador Gama 25

3 Implementação do Ambiente **26**

3.1 Interface Gráfica 26

3.1.1 Tela Inicial do Ambiente 27

3.1.2 Criando um Algoritmo Novo 29

3.1.3 Editando um Algoritmo já Criado 43

3.2 Estrutura do Ambiente 44

3.2.1 Formato dos Arquivos 45
3.2.2 Recursos do MPI 63
3.2.3 Geração do Programa 65
3.3 Desenvolvendo Algoritmos Distribuídos no Ambiente 70
3.3.1 Criação de um Algoritmo Assíncrono 71
3.3.2 Criação de um Algoritmo Síncrono 74

4 Alguns Resultados Obtidos **79**
4.1 Algoritmo Assíncrono 80
4.1.1 Speed Up 82
4.2 Algoritmo Síncrono 84
4.2.1 Sincronizador Beta 86
4.2.2 Speed Up 89

5 Assuntos Relacionados **91**
5.1 Ambiente de Desenvolvimento e Avaliação de Algoritmos de Exclusão Mútua para Sistemas Distribuídos 91
5.2 Dome: Programação Paralela em um Ambiente de Computação Distribuída 93
5.3 Exploração Visual Interativa de Computações Distribuídas 94

6 Conclusões **96**

A Pré-Requisitos para Execução **98**
A.1 MPICH e Dispositivo ch_p4 98
A.2 Arquivo .cshrc 98
A.3 Arquivo .rhosts 99
A.4 Gerenciamento do Ambiente 99
A.4.1 Inicialização 99
A.4.2 Arquivos Gerados 99

B Manual de Utilização do Ambiente **101**
B.1 Iniciando... 101
B.1.1 Tela Inicial do Ambiente 102
B.1.2 Informações Gerais 104
B.1.3 Informações Específicas do Algoritmo 107

B.2 Formatos 108
B.2.1 Topologia 108
B.2.2 Alocação a Processadores 109
B.2.3 Arquivo para Pares Entrada/Ação 109
B.3 Pares Entrada/Ação 110

C MPI **114**
C.1 Introdução 114
C.2 Objetivos do MPI 115
C.3 O Que Está Incluído no MPI? 116
C.4 Termos e Convenções do MPI 117
C.4.1 Especificação de Procedimentos 117
C.4.2 Termos de Semântica 117
C.4.3 Características com a Linguagem C 118
C.5 Comunicação Ponto-a-Ponto 118
C.5.1 Modos para Comunicação Ponto-a-Ponto no MPI 121
C.5.2 Alocação de Buffer de Comunicação 122
C.5.3 Tipos Definidos pelo Usuário 123
C.5.4 Comunicações Coletivas 123
C.6 Tratamento de Erros 124

Lista de Figuras

3.1 Interface Gráfica - Tela Inicial 28
3.2 Interface Gráfica - Dados Gerais I 30
3.3 Interface Gráfica - Dados Gerais II 31
3.4 Interface Gráfica - Dados Específicos I 35
3.5 Interface Gráfica - Dados Específicos II 36
3.6 Interface Gráfica - Resultado da Geração do Programa 40
3.7 Interface Gráfica - Salvar Arquivos 42
3.8 Esquema das Fases de Geração do Arquivo 44
3.9 Grafo em Anel - Não Direcionado e Direcionado 54
3.10 Grafo em Malha 55
3.11 Grafo Torus 56
3.12 Grafo Hipercubo 58
3.13 Grafo Completo com 8 nós 59
3.14 Exemplo de Topologia Geral 68

4.1 Algoritmo Assíncrono - Grafo em Anel 80
4.2 Algoritmo Assíncrono - Grafo em Malha 81
4.3 Algoritmo Assíncrono - Grafo Completo 82
4.4 Algoritmo Assíncrono - Speed Up para Grafo em Anel 83
4.5 Algoritmo Assíncrono - Speed Up para Grafo em Malha 83
4.6 Algoritmo Assíncrono - Speed Up para Grafo Completo 84
4.7 Algoritmo Síncrono - Grafo em Anel 85
4.8 Algoritmo Síncrono - Grafo em Malha 85
4.9 Algoritmo Síncrono - Grafo Completo 86
4.10 Árvore Geradora X Algoritmo Síncrono - Grafo em Anel 87
4.11 Árvore Geradora X Algoritmo Síncrono - Grafo em Malha 88
4.12 Árvore Geradora X Algoritmo Síncrono - Grafo Completo 88

4.13 Algoritmo Síncrono - Speed Up para Grafo em Anel 89
4.14 Algoritmo Síncrono - Speed Up para Grafo em Malha 90
4.15 Algoritmo Síncrono - Speed Up para Grafo Completo 90

B.1 Interface Gráfica - Tela Inicial do Ambiente 103
B.2 Topologia Geral 109

Capítulo 1

Introdução

1.1 Objetivos

A proposta desta tese é implementar uma ferramenta que permita o desenvolvimento de algoritmos distribuídos síncronos e assíncronos, o que vem auxiliar no aprendizado e avaliação destes algoritmos, nos meios educacionais e de pesquisa.

Nesta ferramenta, os dados referentes ao algoritmo distribuído devem ser escritos seguindo os formatos padronizados apresentados no livro *An Introduction to Distributed Algorithms* de Valmir C. Barbosa. [2]

Segundo o modelo do livro, os algoritmos distribuídos podem ser escritos através de uma seqüência de pares consistindo de uma Entrada e Ações associadas a esta Entrada.

Para os algoritmos assíncronos, a computação de um nó só pode ser realizada no momento do recebimento de uma mensagem, ou no início de sua computação, caso este nó esteja no conjunto dos nós que iniciam espontaneamente o algoritmo. Neste caso, dois tipos de pares podem ser escritos. Um par referente às ações a serem tomadas inicialmente, e os demais pares referentes às ações a serem tomadas ao receber mensagens.

A Entrada de cada par identifica o tipo de mensagem recebida. No caso do par indicando as ações dos nós que iniciam espontaneamente, chamamos esta Entrada de NIL, que representa o recebimento de nenhuma mensagem.

Nos algoritmos síncronos, todos os nós são guiados por um relógio global, que gera intervalos de tempo de duração fixa e não nula. Garante-se também que o tempo necessário para uma mensagem ser transmitida entre vizinhos é estritamente menor que a duração de um intervalo do relógio global. De forma semelhante ao algoritmo assíncrono, aqui também podemos dividir os pares Entrada/Ação em dois tipos. O

par referente ao pulso zero do relógio, ou seja, o primeiro pulso do algoritmo, e o par referente aos demais intervalos do algoritmo.

Neste caso, a Entrada de cada par representa um intervalo de tempo. Para o primeiro par, esta Entrada será "s=0", e para os demais pares o intervalo de tempo correspondente a cada ação diferente.

Além dos pares Entrada/Ação, algumas informações adicionais são necessárias para a implementação do algoritmo distribuído, como por exemplo, as variáveis do algoritmo, o número de nós que devem executar, a topologia, ou seja, a forma como os nós se comunicam, entre outros.

Para o ambiente implementado, foi definido que os dados referentes ao código do algoritmo devem ser informados utilizando a linguagem de programação C, por esta ser uma linguagem largamente utilizada nos meios educacionais, e também por ser a linguagem utilizada na implementação do ambiente.

1.2 Metodologia

O ambiente implementado como objeto desta tese foi desenvolvido utilizando a linguagem de programação C [15] e o padrão MPI (Message Passing Interface) que realiza a comunicação entre os processos que executam o programa. O MPI é explicado com mais detalhes no apêndice C. Este padrão possibilitou a implementação de toda a comunicação entre os processos, assim como o controle da capacidade de um *buffer* para armazenar as mensagens enviadas por um processo.

A interface gráfica com o usuário foi implementada para a Web, através da Internet, utilizando a linguagem padrão HTML (Hyper-Text Markup Language) na geração dos formulários, e CGI (Common Gateway Interface) para tratamento dos dados informados no formulário HTML. Para implementação dos programas CGI foi, também, utilizada a linguagem de programação C.

No momento da implementação da interface gráfica para o ambiente desenvolvido, foi realizada um pesquisa sobre em que ferramenta a mesma poderia ser implementada. Chegamos a iniciar a implementação utilizando a linguagem Tcl-Tk [20], que permite a criação de uma interface gráfica para o ambiente Unix. Porém, optamos por desenvolvê-la em HTML [6] [14], pois desta forma, o acesso e a utilização do ambiente poderia ser realizado de qualquer plataforma, além da facilidade na criação das telas, e ainda contando com a vantagem de desenvolver os programas

utilizando a linguagem C.

1.2.1 World-Wide Web e Internet

A World-Wide Web (*WWW, W3*) é descrita oficialmente como uma "grande área de informações *hypermedia*", que permite acesso universal a um imenso universo de documentos. O que o projeto da World-Wide Web tem feito é prover aos usuários em uma rede de computadores com um meio consistente de acessar uma variedade de mídias de uma forma simplificada. Utilizando uma interface de software popular para a Web, como Mosaic (primeira interface desenvolvida) ou Netscape, o projeto da Web tem mudado a forma com que as pessoas vêem e criam informações. [9]

As primeiras idéias desses sistemas tiveram como principal objetivo o avanço da ciência e educação. Apesar do projeto da World-Wide Web ter sido direcionado para fazer um significante impacto nessas áreas, ele está preparado para revolucionar muitos setores da sociedade, incluindo comércio, política e literatura.

A Internet é a palavra que tem sido usada para descrever a maciça rede de computadores. A World-Wide Web é geralmente utilizada na Internet; elas não significam a mesma coisa. A Web se refere ao corpo da informação - um espaço abstrato de conhecimento, enquanto que a Internet se refere ao lado físico da rede global, uma massa gigante de cabos e computadores.

A World-Wide Web utiliza a Internet para transmitir documentos *hypermedia* entre computadores usuários internacionalmente. As pessoas são responsáveis pelos documentos que elas constróem e tornam disponíveis publicamente na Web. Através da Internet, centenas de milhares de pessoas no mundo inteiro estão disponibilizando informações de suas casas, escolas e empresas.

A Web oferece uma interface muito simples de utilizar ao invés dos recursos tradicionalmente difíceis da Internet. Provavelmente, essa facilidade de uso assim como a popularidade da maioria das interfaces gráficas para a Web são as responsáveis pela explosão de tráfego na Web.

O potencial de utilizar hipertexto e multimídia na rede tem ajudado a maioria dos usuários a criar e explorar incontáveis aplicações novas na Internet. Talvez não seja surpresa que mais usuários da área educacional estejam na Web do que era esperado.

1.2.2 HTML

A linguagem padrão que a Web usa para criar e reconhecer documentos *hypermedia* é HTML (*Hypertext Markup Language*) [6]. Ela é relacionada à SGML (*Standard Generalized Markup Language*), um método de representar linguagens de formatação de documentos, mas a HTML, tecnicamente, não é um subconjunto da mesma. Linguagens como HTML, que seguem o formato SGML, permitem que os escritores de documentos separem informações da apresentação do documento - ou seja, documentos contendo a mesma informação pode ser apresentado de diferentes formas. Os usuários têm a opção de controlar elementos visuais como as fontes, tamanhos das fontes e espaçamento entre parágrafos sem alterar a informação original.

HTML é extensamente elogiada pela sua facilidade de uso. Documentos Web são tipicamente escritos em HTML e são normalmente nomeados com o sufixo ".html". Documentos HTML são nada mais que arquivos ASCII padrão com códigos de formatação que contém informação sobre *layout* (estilos de texto, títulos de documentos, parágrafos, listas) e *hyperlinks.*

A linguagem HTML também fornece a possibilidade de retornar informações ao servidor Web para que uma determinada ação seja executada, através dos formulários. Na verdade, os formulários são produzidos com uma cooperação de elementos de HTML e CGI [6] [19] [5]. O programa CGI cuida do processamento dos dados e HTML é usada para apresentar o formulário nos *browsers.*

Os controles oferecidos pela linguagem HTML para a construção de formulários são:

TEXT Cria uma caixa de texto na tela, onde pode-se digitar um texto

CHECKBOX Exibe uma caixa quadrada na tela, que pode ser marcada, indicando seleção de um item. Mais de um checkbox pode estar marcado.

RADIO Oferece botões de opção (ou rádio) que se comportam como caixas de seleção, onde apenas uma opção pode ser selecionada.

SUBMIT Oferece um botão que, quando pressionado, envia os dados do formulário para o servidor.

RESET Oferece um botão que, quando pressionado, limpa os dados do formulário retomando à sua condição inicial.

SELECT Cria uma caixa de lista de opções alternativas. Cada opção é incluída através do atributo OPTION.

TEXTAREA Oferece uma caixa de texto multilinhas.

1.2.3 CGI

CGI (Common Gateway Interface) não é realmente uma linguagem ou um protocolo no sentido mais estrito desses termos. É, na realidade, apenas um conjunto de variáveis de nomes e convenções comuns para a passagem de informações do servidor para o cliente e vice-versa. [19] [5]

A "Common Gateway Interface", ou CGI, é a interface que possibilita que scripts e programas rodem no servidor Web. Ao contrário do que algumas pessoas pensam, não basta fazer um formulário em html para que ele funcione, é necessário que exista um programa executando as instruções que permitem trabalhar as informações.

Arquivos CGI que permitem o processamento de formulários, a realização de operações matemáticas, etc. são chamados scripts. Os scripts CGI são pequenos programas, geralmente escritos na linguagem C, C++, Visual Basic ou Perl. Esta última têm a preferência por ser limitada, porém robusta, e muito comum em ambiente UNIX, predominantemente na Internet.

1.3 Características do Ambiente

O resultado final do ambiente para desenvolvimento de algoritmos distribuídos implementado é um programa executável que realiza a execução do algoritmo desejado. Pode-se dividir a implementação deste ambiente basicamente em duas partes: a primeira se refere à interface gráfica, onde o usuário deverá informar os dados relativos ao algoritmo que deseja desenvolver; a segunda parte diz respeito ao tratamento dos dados do algoritmo informado para a geração do programa final.

1.3.1 Interface Gráfica

É na interface gráfica que o usuário deve informar os dados referentes ao algoritmo distribuído que deseja desenvolver. Pode-se dividir estes dados em duas categorias: informações gerais, que não fazem parte do código do algoritmo, propriamente dito; e os dados do código do algoritmo.

Dentre os dados gerais pode-se citar o nome do algoritmo, tipo de algoritmo (Assíncrono ou Síncrono), topologia, número de nós do grafo.

Quanto à topologia, o usuário pode selecionar uma topologia pré-definida (anel, malha, hipercubo, etc.) ou definir um grafo geral, seguindo um formato definido para isso. Uma outra alternativa oferecida ao usuário é indicar o nome de um arquivo que contenha a descrição da topologia desejada.

Uma característica bastante interessante do ambiente implementado é a possibilidade do usuário definir a capacidade de um *buffer* de mensagens, e desta forma realizar diferentes experimentos. Esta informação é fornecida considerando-se o número máximo de mensagens em trânsito, ou seja, que o *buffer* pode conter.

O ambiente implementado permite que o usuário informe em que plataforma / sistema o programa gerado deverá executar, como por exemplo uma rede de *workstations* com sistema Solaris, ou uma máquina SP2. Desta forma o programa é gerado seguindo as características do sistema escolhido.

O usuário ainda poderá determinar o número de processadores em que o programa deverá ser executado, assim como a alocação de cada nó ao processador desejado, ou seja, em um grafo com 4 nós, os nós 0 e 2 podem estar executando no processador A, enquanto os nós 1 e 3 executam no processadores B e C, respectivamente.

1.3.2 Geração do Programa

A partir dos dados do algoritmo distribuído fornecidos pelo usuário, são criados arquivos que servirão de entrada para um programa que tem como resultado o programa executável que realiza a execução do algoritmo distribuído.

Caso aconteça algum problema durante a geração do programa, o erro é apresentado ao usuário para que ele possa alterar seus dados e gerar o programa corretamente.

A geração do programa para algoritmos assíncronos e síncronos é executada por dois programas diferentes. Para o algoritmo síncrono, a idéia de um relógio global é difícil de ser reproduzida em um ambiente real. Por esse motivo, um sincronizador foi utilizado com o intuito de transformar o algoritmo síncrono em um assíncrono, de forma a preservar o seu funcionamento.

Os dois sincronizadores implementados neste ambiente são Alfa e Beta, que são bastante conhecidos na literatura de Algoritmos Distribuídos. Quando o usuário deseja desenvolver um algoritmo síncrono, ele terá a possibilidade de escolher qual

o sincronizador será utilizado na conversão.

1.4 Organização da Tese

Esta seção descreve como a parte escrita desta tese está organizada.

O capítulo 1 apresenta uma visão geral dos objetivos deste trabalho, metodologias utilizadas e características do ambiente.

No segundo capítulo são apresentados os conceitos básicos de algoritmos distribuídos, assim como os modelos para descrição destes algoritmos, nos quais a implementação do ambiente de desenvolvimento foi baseada.

O capítulo 3 descreve as características do ambiente desenvolvido, assim como detalhes de implementação, formato de arquivos gerados, apresentação das telas que compõem a interface gráfica e dois exemplos de como utilizar o ambiente.

No capítulo 4 são apresentados alguns resultados para a execução de um algoritmo assíncrono e um algoritmo síncrono, sendo que este último foi experimentado utilizando os dois tipos de sincronizadores disponíveis: Alfa e Beta.

O capítulo 5 apresenta a descrição de três projetos que se assemelham à proposta desta tese. No capítulo 6 são apresentadas conclusões sobre este trabalho e possíveis trabalhos futuros.

No apêndice A são descritos os requisitos necessários para a execução do programa gerado por este ambiente. O apêndice B apresenta um Manual para utilização do ambiente e a descrição de cada campo contemplado nas telas disponíveis ao usuário. O apêndice C apresenta uma descrição das principais características do padrão MPI e as funções deste padrão que foram utilizadas na implementação do ambiente.

Capítulo 2

Algoritmos Distribuídos

Um sistema distribuído é considerado como um conjunto de processadores conectados em rede. O sistema pode ser físico, quando os computadores são conectados fisicamente, ou lógico, quando um conjunto de processos (software) são conectados através de um mecanismo de passagem de mensagens.

Os conceitos de passagem de mensagens e memória distribuída estão intimamente relacionados um com o outro. Sistemas de memória distribuída contemplam um conjunto de processadores interconectados em rede de alguma forma por canais de comunicação. Dependendo do sistema considerado, essa rede pode conter ligações ponto-a-ponto, nesse caso, cada canal de comunicação é responsável pelo tráfego de informações entre dois processadores, exclusivamente; ou conter canais de transmissão que permitem o tráfego entre os processadores de um grupo, chamados canais 'broadcast'.

Os processadores não podem compartilhar a memória fisicamente, e dessa forma, toda a troca de informação entre eles deve, necessariamente, ser realizada por passagem de mensagens através dos canais que interligam os processadores.

Os programas que executam em sistemas distribuídos podem ser vistos como *entidades* de código seqüencial, cada uma executando em um processador, e talvez mais de uma por processador. Tais *entidades* podem ser chamadas de tarefas, processos, ou threads, dependendo das particularidades do sistema.

Enquanto que no nível dos processadores em um sistema distribuído não há escolha quanto a confiar na comunicação por passagem de mensagens, no nível dos processos, existem várias possibilidades. Quando os processos executam no mesmo processador, eles podem se comunicar um com o outro através do uso explícito da memória do processador, ou pelo uso de passagem de mensagens de um forma bem

simples. Os processos que executam em processadores diferentes, também podem se comunicar destas duas formas. Eles podem se comunicar utilizando passagem de mensagens contando com os mecanismos que oferecem comunicação entre processadores, ou podem empregar mecanismos para emular o compartilhamento da memória. Uma forma híbrida para esta questão é implementar o uso de memória compartilhada entre processos em um mesmo processador e passagem de mensagens entre processos de processadores distintos.

2.1 Programas Orientados a Passagem de Mensagens

O desenvolvimento de algoritmos distribuídos utilizando o ambiente proposto se baseia no modelo apresentado em [2].

Segundo esse modelo, um algoritmo distribuído é representado por um grafo direcionado conectado $G_T = (N_T, D_T)$, onde N_T representa um conjunto de tarefas e D_T representa um conjunto de canais de comunicação unidirecionais. Para uma tarefa t, o conjunto In_t denota o conjunto de arestas direcionadas para t e Out_t, o conjunto de arestas direcionadas para fora de t. Os canais contidos em In_t são aqueles sobre os quais t recebe mensagens e os canais contidos em Out_t são aqueles sobre os quais t envia mensagens. Pode-se também definir $n_t = |In_t|$, onde n_t representa o número de canais de onde t pode receber mensagens.

Uma tarefa t é uma entidade reativa, ou seja, direcionada a mensagens, quando ela somente realiza computação (incluindo o envio de mensagens para outras tarefas) como resposta ao recebimento de uma mensagem de outra tarefa. Uma exceção a esta regra é que pelo menos uma tarefa deve ser capaz de enviar mensagens espontaneamente para outras tarefas no início de sua execução, sem a necessidade de receber uma mensagem. Isso porque, caso contrário, o caráter "dirigido a mensagem" das tarefas assumido implicaria que toda a tarefa permaneceria indefinidamente inativa e nenhuma computação aconteceria. Uma tarefa também pode inicialmente realizar computação para fins de inicialização.

O algoritmo **Task_t**, apresentado a seguir, descreve o comportamento geral de uma tarefa t genérica. Apesar de nesse algoritmo a tarefa realizar computação e enviar mensagens, isso não é obrigatório.

Algoritmo Task_t:

Faça alguma computação;

Envie uma mensagem em cada canal de um subconjunto (pode ser vazio) de Out_i;

Repita

Receba uma mensagem em $c_1 \in In_i$ e B_1

Faça alguma computação;

Envie uma mensagem em cada canal de um subconjunto (pode ser vazio) de Out_i;

ou ...

ou

Receba uma mensagem em $c_n \in In_i$ e B_n

Faça alguma computação;

Envie uma mensagem em cada canal de um subconjunto (pode ser vazio) de Out_i;

até que terminação global seja atingida por t.

Algumas considerações importantes podem ser ressaltadas ao se observar o algoritmo **Task_t**. A primeira delas é com relação a como a computação começa e termina para a tarefa t. Como já foi notado anteriormente, a tarefa t inicia realizando alguma computação e enviando mensagens para algumas tarefas (ou nenhuma) que estão conectadas a ela no grafo G_T, através de uma aresta que sai de t. As mensagens são enviadas através da operação *send*. A partir daí, t iterage até que uma condição de terminação global seja satisfeita para t, e nesse momento a computação termina. A cada iteração, t realiza computação e pode enviar mensagens.

A segunda observação importante é sobre a construção da estrutura **Repita... até** e a semântica associada a ela. Cada iteração desse loop possui **n** *comandos guardados*, agrupados pelos conectivos **ou**. Um *comando guardado* é normalmente denotado pela expressão

guarda $\Rightarrow$ comando;

onde , no nosso contexto, guarda é uma condição no formato

Receba uma mensagem em $c_k \in In_t$ **e** B_k

para alguma condição booleana B_k, onde $1 \leq k \leq n$. O comando **Receba**, que aparece na descrição do guarda, é uma operação para a tarefa receber mensagens, e estas mensagens são recebidas através da operação *receive*. Dizemos que o guarda está pronto quando existe uma mensagem disponível para recepção imediata no canal c_k e, além disso, a condição B_k é verdadeira.

A semântica geral do loop **Repita... até** é descrita a seguir: A cada iteração, executar o comando de apenas um guarda o qual esteja pronto. Se nenhum guarda está pronto, então a tarefa é suspensa até que exista um pronto. Se mais de um guarda está pronto, então, um deles é selecionado arbitrariamente.

Uma observação final quanto ao algoritmo **Task_t** é a semântica associada às operações **receive** e **send**. A operação receive é geralmente de natureza bloqueante. Um receive bloqueante tem o efeito de suspender a tarefa até que uma mensagem chegue no canal especificado, a menos que uma mensagem já esteja lá para ser recebida, nesse caso a recepção acontece e a tarefa retoma sua execução imediatamente.

A operação send pode ser bloqueante ou não-bloqueante. Se ela é bloqueante, então a tarefa é suspensa até que a mensagem possa ser entregue diretamente à tarefa receptora, a menos que a tarefa receptora já esteja suspensa esperando receber uma mensagem no canal correspondente quando o send é executado. Um send bloqueante e um receive bloqueante constituem o que chamamos de *rendez-vouz*, que é um mecanismo para sincronização de tarefas.

Se a operação send tem uma natureza não-bloqueante, então a tarefa transmite a mensagem e imediatamente retoma sua execução. A versão não-bloqueante do send requer bufferização, ou seja, armazenamento em um buffer, das mensagens que foram enviadas mas ainda não foram recebidas, que são as mensagens *em trânsito* no canal.

Às vezes, as operações de send bloqueante e não-bloqueante são também referenciadas como síncronas e assíncronas, respectivamente, com a intenção de enfatizar o efeito de sincronismo que existe no primeiro caso.

2.2 Algoritmos Assíncronos e Síncronos

Seguindo o modelo já introduzido para o algoritmo **Task_t**, um algoritmo distribuído é representado por um grafo direcionado conectado $G_T = (N_T, D_T)$. Nesse grafo, N_T é um conjunto de tarefas e D_T é um conjunto de canais de comunicação unidirecionais. Tarefas em N_T são entidades direcionadas a mensagens, e assume-se que os canais em D_T têm capacidade infinita, isto é, nenhuma tarefa está suspensa para sempre tentando enviar uma mensagem em um canal.

Para fins de simplificação, vamos passar a referenciar o grafo $G_T = (N_T, D_T)$ simplesmente por $G = (N, D)$, com $n = |N|$ e $m = |D|$. Para $1 \leq i, j \leq n$, n_i denota um membro de N, referenciado para simplificar como um nó, e para $j \neq i$ a expressão $(n_i \rightarrow n_j)$ denota um membro de D, referenciado para simplificar como uma aresta direcionada (ou simplesmente uma aresta). O conjunto de arestas direcionadas para fora de n_i é denotado por $Out_i \subseteq D$, e o conjunto de arestas direcionadas para dentro de n_i é denotado por $In_i \subseteq D$. Claramente, $(n_i \rightarrow n_j) \in Out_i$ se e somente se $(n_i \rightarrow n_j) \in In_j$. Diz-se que os nós n_i e n_j são vizinhos um do outro se e somente se $(n_i \rightarrow n_j) \in D$ ou $(n_j \rightarrow n_i) \in D$. O conjunto de vizinhos de n_i é denotado por $Neig_i$, e contém duas partições, I_Neig_i e O_Neig_i, cujos membros são, respectivamente, n_j tal que $(n_j \rightarrow n_i) \in D$ e n_j tal que $(n_i \rightarrow n_j) \in D$.

Freqüentemente G é tal que $(n_i \rightarrow n_j) \in D$ e $(n_j \rightarrow n_i) \in D$ e nesse caso, é mais conveniente ver essas duas arestas como uma única aresta não-direcionada (n_i, n_j). Analisando esse caso de grafo não-direcionado, G é denotado por $G = (N, E)$, e então $m = |E|$. Os membros de E são referenciados simplesmente por arestas. No caso não-direcionado, o conjunto de arestas incidentes a n_i é denotado por $Inc_i \subseteq E$. Dois nós n_i e n_j são vizinhos se e somente se $(n_i, n_j) \in E$. O conjunto de vizinhos de n_i continua a ser denotado por $Neig_i$.

Os algoritmos distribuídos assíncronos podem ser representados utilizando o modelo assíncrono completo (ou simplesmente assíncrono), que é caracterizado pelas duas seguintes propriedades:

- Cada nó é guiado por sua própria base de tempo local e independente, referenciada pelo seu relógio local.
- O atraso que uma mensagem sofre para ser transmitida entre vizinhos é finito mas imprevisível.

O assincronismo completo suposto nesse modelo o torna mais realístico do ponto de vista que de alguma maneira reflete algumas características dos sistemas de memória distribuída.

2.2.1 Modelo Assíncrono

De acordo com o Algoritmo Task_t, a computação de um nó no modelo assíncrono pode ser descrita fornecendo as ações a serem tomadas inicialmente (se aquele nó inicia sua computação e envia mensagens espontaneamente, ao invés de fazê-lo ao acordar com a recepção de uma mensagem) e as ações a serem tomadas ao receber mensagens quando uma certa condição booleana é satisfeita. Tal descrição é dada pelo Algoritmo A_Template, que é um padrão para o desenvolvimento dos algoritmos assíncronos baseado no qual o ambiente de desenvolvimento, que é o objetivo desta tese, foi implementado.

O Algoritmo A_Template descreve a computação realizada por $n_i \in N$. Nesse algoritmo vamos denotar $N_0 \subseteq N$ como um conjunto não vazio de nós que podem enviar mensagens espontaneamente.

O Algoritmo A_Template apresentado a seguir foi definido para o caso em que G é um grafo direcionado. Para o caso de grafo não-direcionado, tudo o que é necessário fazer no algoritmo é substituir todas as ocorrências de In_i e Out_i por Inc_i.

Algoritmo A_Template:

▷ **Variáveis:**

Variáveis usadas por n_i e seus valores iniciais são listados aqui.

▷ **Entrada:**

$msg_i = nil$.

Ação Se $n_i \in N_0$:

Faça alguma computação;

Envie uma mensagem em cada aresta de um subconjunto (pode ser vazio) de Out_i.

▷ **Entrada:**

msg_i tal que $origem_i(msg_i) = c_k \in In_i$ com $1 \leq k \leq |In_i|$.

Ação quando B_k:

Faça alguma computação;

Envie uma mensagem em cada aresta de um subconjunto (pode ser vazio) de Out_i.

Algumas observações importantes podem ser feitas considerando o algoritmo A_Template. A primeira observação é que o algoritmo é dado pela listagem das variáveis que ele emprega (assim como seus valores iniciais) e então uma série de pares Entrada/Ação. Cada um desses pares, ao contrário do algoritmo Task_t, é dado para um tipo específico de mensagem, e pode corresponder então a mais que um comando guardado no Algoritmo Task_t, com a Entrada correspondendo à recepção da mensagem no guarda, e a Ação correspondendo à parte do comando a ser executado, quando a condição booleana expressada no guarda é verdadeira.

Cada comando guardado no Algoritmo Task_t pode também corresponder a mais de um par Entrada/Ação, no Algoritmo A_Template. Além disso, a fim de preservar o funcionamento do Algoritmo Task_t, ou seja, que um novo comando guardado é somente considerado para execução na próxima iteração, depois que o comando dentro do comando guardado selecionado tenha sido executado e completado, fazemos a suposição que cada ação no Algoritmo A_Template é uma ação atômica. Uma ação atômica é uma ação que é permitida de prosseguir até terminar antes de qualquer interrupção.

Uma outra observação é que a mensagem associada com uma Entrada para a expressão "Se $n_i \in N_0$" é tratada como se $msg_i = nil$, pois nesse caso nenhuma mensagem realmente existe para disparar a ação de n_i. Quando uma mensagem existe, assume-se que sua origem é conhecida por n_i. Em muitos casos, pode-se considerar que a origem da mensagem recebida é $n_j \in I\_Neig_i$ onde $(n_j \rightarrow n_i)$ (isto é, n_j é o nó de onde msg_i originou).

Da mesma forma, enviando uma mensagem em uma aresta em Out_i é em muitos casos equivalente a enviar uma mensagem para $n_j \in O\_Neig_i$ se aquela aresta é $(n_i \rightarrow n_j)$. Porém, devemos evitar estabelecer isso como suposições gerais porque elas não valem para o caso de sistemas anônimos. Quando elas valem e G é um grafo não-direcionado, então todas as ocorrências de I_Neig_i e de O_Neig_i no Algoritmo

A_Template modificado podem ser substituídas pelas ocorrências de $Neig_i$.

Como observação final, segundo o modelo apresentado, sempre que no Algoritmo A_Template n_i envia mensagens para um subconjunto de Out_i contendo mais que uma aresta, assume-se que todas essas mensagens podem ser enviadas em paralelo.

2.2.2 Modelo Síncrono

Em adição ao modelo assíncrono, apresentado anteriormente, um outro modelo relacionado com as características de tempo de G é o modelo síncrono completo, para o qual valem as duas propriedades seguintes:

- Todos os nós são guiados por uma base de tempo global, referida como um relógio global, que gera intervalos de tempo (ou simplesmente intervalos) de duração fixa e não nula.
- O atraso que uma mensagem sofre para ser transferida entre vizinhos é diferente de zero e estritamente menor que a duração de um intervalo do relógio global.

Nesta seção será apresentado um esboço do funcionamento de um algoritmo distribuído, chamado algoritmo síncrono, designado sob as suposições do modelo síncrono. O início de cada intervalo do relógio global é indicado por um pulso. Para $s \geq 0$, o pulso s indica o início do intervalo s. No pulso $s = 0$, os nós em N_0 enviam mensagens em algumas (ou nenhuma) das arestas direcionadas para fora deles. No pulso $s > 0$, todas as mensagens recebidas no pulso $s-1$ por suposição já chegaram, e então os nós em N podem computar e enviar mensagens.

Uma suposição que se fez implicitamente é que a computação realizada pelos nós durante um intervalo não gasta nenhum tempo. Sem essa suposição, a duração de um intervalo não seria suficiente para ambas as computações locais serem realizadas e as mensagens serem transferidas, porque essa transferência pode levar a duração inteira de um intervalo para acontecer. Uma outra forma equivalente para aproximar isso seria dizer que, para algum período de tempo $d \geq 0$, estritamente menor do que a duração de um intervalo, a computação local não leva mais do que um tempo d, enquanto mensagens levam estritamente menos do que a duração de um intervalo menos d para serem transmitidas. Para atender a essa suposição, o que se faz é tornar $d = 0$.

O conjunto de nós N_0 que podem enviar mensagens no pulso $s = 0$ tem, no caso síncrono, a mesma interpretação que o conjunto de nós que enviam mensagens espontaneamente no algoritmo assíncrono. Porém, no caso síncrono faz sentido os nós realizarem computação sem terem recebido mensagens, pois eles são direcionados por um relógio global, e não pelo recebimento de mensagens. Dessa forma, em princípio, o algoritmo síncrono não requer nenhuma mensagem para executar, e os nós podem continuar a computação mesmo se $N_0 = \emptyset$. Entretanto, para que a computação global tenha algum significado diferente da paralelização de n computações seqüenciais completamente independentes, pelo menos uma mensagem tem de ser enviada por pelo menos um nó. Então, o conjunto N_0 tem pelo menos o emissor desta mensagem como elemento.

Abaixo está representado o Algoritmo S_Template, que estabelece as convenções de como descrever um algoritmo síncrono. Para $s \geq 0$ e $N_i \in N$, no algoritmo S_Template, $MSG_i(s)$ é um conjunto vazio (se $s = 0$) ou denota o conjunto de mensagens recebidas por n_i durante o intervalo $s - 1$ (se $s > 0$), que também pode ser vazio. O algoritmo para n_i é dado para o caso em que G é um grafo direcionado. Para o caso de grafo não-direcionado, basta simplesmente substituir In_i e Out_i por Inc_i no algoritmo.

Algoritmo S_Template:

▷ **Variáveis:**

Variáveis usadas por n_i e seus valores iniciais são listados aqui.

▷ **Entrada:**

$s = 0$, $MSG_i(0) = \emptyset$

Ação Se $n_i \in N_0$:

Faça alguma computação;

Envie uma mensagem em cada aresta de um subconjunto (pode ser vazio) de Out_i.

▷ **Entrada:**

$s > 0$, $MSG_i(1), ..., MSG_i(s)$ tal que $origem_i(msg) = c_k \in In_i$ com $1 \leq k \leq |In_i|$ para $msg \in \bigcup_{r=1}^{s} MSG_i(r)$.

Ação:

Faça alguma computação;

Envie uma mensagem em cada aresta de um subconjunto (pode ser vazio) de Out_i.

Como no caso do algoritmo A_Template, o algoritmo S_Template é apresentado como um conjunto de pares Entrada/Ação. As entradas agora incluem informação do relógio global, representado por s que é, como já foi visto, o que realmente guia os nós. A atomicidade das ações vem como conseqüência das características do modelo síncrono, porque nenhum nó executa mais do que uma ação por intervalo do relógio global. Na verdade, é simples de ver que todos os nós executam exatamente uma ação por intervalo do relógio global, porque as ações são agora incondicionais, ou seja, na descrição do algoritmo S_Template, não existem as condições que apareciam no algoritmo A_Template que eram provenientes do algoritmo Task_t. A razão porque isto pode ser feito é que tais condições são, no caso síncrono, avaliadas somente na ocorrência dos pulsos, e isso pode ser tratado dentro da ação.

Uma outra observação importante considerando o algoritmo S_Template, é que é permitido a n_i ter acesso, durante sua computação no intervalo $s > 0$, a todos os conjuntos $MSG_i(1), ..., MSG_i(s)$, embora normalmente somente $MSG_i(s)$ seja necessário.

2.3 Algoritmos Sincronizadores

Um algoritmo síncrono pode ser transformado em um algoritmo assíncrono acrescentando um custo adicional de mensagens e tempo para suprir a falta de uma base de tempo global, resultando um algoritmo que garantidamente funciona como no modelo síncrono.

A propriedade essencial para preservar o funcionamento do algoritmo síncrono durante a tradução é que o nó n_i só avança para o pulso $s + 1$ depois que todas as mensagens enviadas a ele no pulso s tenham sido entregues e incorporadas ao seu conjunto de mensagens. Com a finalidade de garantir essa propriedade, considera-se

que todas as mensagens enviadas no algoritmo síncrono sejam reconhecidas. Essas mensagens são denotadas por *comp_msg* (ou mensagem de computação) e os reconhecimentos por *ack*. Pode-se dizer que um nó está seguro em relação ao pulso s se, e somente se, ele recebeu um *ack* para todas as *comp_msg*'s que ele enviou no pulso s.

Para garantir que essa propriedade essencial funciona para n_i no pulso s, é suficiente que n_i receba informação indicando que todos os seus vizinhos estão seguros com respeito ao pulso s. A função do sincronizador é transferir essa informação para todos os nós considerando todos os pulsos da computação síncrona.

Pode-se definir um sincronizador como sendo um algoritmo assíncrono que é repetido a cada pulso do algoritmo síncrono, com a finalidade de passar para todos os nós a informação de "seguro".

Nas seções seguintes serão apresentados os três tipos de sincronizadores muito utilizados no estudo de algoritmos distribuídos, que são os algoritmos Alfa, Beta e Gama.

2.3.1 Sincronizador Alfa

Como já foi visto, a função principal de um sincronizador é passar para todos os nós e para todos os pulsos a informação de que todos os seus vizinhos estão "seguros" em relação àquele pulso.

No sincronizador Alfa, a informação de que todos os vizinhos de um nó estão seguros em relação a um pulso $s \geq 0$ é passada diretamente para cada um de seus vizinhos através da mensagem $safe(s)$. Desta forma, um nó só pode avançar para o pulso $s+1$ quando ele tiver recebido uma mensagem $safe(s)$ de cada um de seus vizinhos.

Vamos apresentar a seguir o algoritmo *A_Alg(Alfa)* que mostra o funcionamento do sincronizador Alfa. [2]

Algoritmo A_Alg(Alfa):

▷ **Variáveis:**

$s_i = 0$;

$MSG_i(s) = \emptyset$ para todo $s \geq 0$;

$initiated_i = false$;

$go_i^j = false$ para todo $n_j \in Neig_i$;

$expected_i(s) = 0$ para todo $s \geq 0$;

$safe_i^j(s) = false$ para todo $n_j \in Neig_i$ e todo $s \geq 0$;

▷ **Entrada:**

$msg_i = nil$

Ação Se $n_i \in N_0$:

$initiated_i = true$;

Envie *startup* para todo $n_i \in Neig_i$;

▷ **Entrada:**

$msg_i = startup$ tal que $origem_i(msg_i) = (n_i, n_j)$

Ação:

Se não $initiated_i$ então

Início

$initiated_i = true$;

Envie *startup* para todo $n_k \in Neig_i$;

Fim

$go_i^j = true$;

Se go_i^j para todo $n_j \in Neig_i$ então

Início

Faça alguma computação;

Envie $comp\_msg(s_i)$ em cada aresta de um subconjunto (pode ser vazio) de Out_i;

Se $expected(s_i) = 0$ então

Envie $safe(s_i)$ para todo $n_k \in Neig_i$;

Fim

▷ **Entrada:**

$msg_i = comp\_msg(s)$ tal que $origem_i(msg_i) = (n_i, n_j)$

Ação:

$MSG_i(s+1) = MSG_i(s+1) \bigcup \{msg_i\}$;

Envie $ack(s)$ para n_j;

▷ **Entrada:**

$msg_i = ack(s)$

Ação:

$expected_i(s) = expected_i(s) - 1$;

Se $expected_i(s) = 0$ então

Envie $safe(s)$ para todo $n_k \in Neig_i$;

▷ **Entrada:**

$msg_i = safe(s)$ tal que $origem_i(msg_i) = (n_i, n_j)$

Ação:

$safe_i^j(s) = true$;

Se $safe_i^k(s_i)$ para todo $n_k \in Neig_i$ então

Início

$s_i = s_i + 1$;

Faça alguma computação;

Envie $comp\_msg(s_i)$ em cada aresta de um subconjunto (pode ser vazio) de Out_i;

Se $expected_i(s_i) = 0$ então

Envie $safe(s_i)$ para todo $n_k \in Neig_i$;

Fim

Como se pode observar, algumas variáveis foram incluídas para garantir a sincronização do algoritmo assíncrono. A variável $expected_i(s)$, inicialmente igual a zero, guarda para todo $s \geq 0$ o número de $ack(s)$'s que n_i espera. Assume-se que essa variável é incrementada toda vez que n_i envia uma $comp\_msg(s)$, portanto essa ação está embutida na linha do algoritmo "Envie $comp\_msg$...".

O nó n_i também mantém a variável $safe_i^j(s)$ para cada vizinho n_j para todo $s \geq 0$, inicialmente recebendo o valor $false$ e utilizado para indicar que uma mensagem $safe(s)$ foi recebida de n_j.

Apesar da simplicidade do sincronizador Alfa, ao implementar as ações iniciais do algoritmo A_Alg(Alfa) apresentado anteriormente é necessário considerar algumas observações. Um nó em N_0 se comporta inicialmente como no modelo síncrono. No algoritmo síncrono, um nó que não está em N_0 permanece ocioso por um determinado número de pulsos. Já no algoritmo A_Alg(Alfa) esses nós executam uma ação em todos os pulsos, porque, caso contrário seus vizinhos nunca receberiam as mensagens $safe$ que eles deveriam enviar e a computação não teria continuidade.

Para contornar essa questão foi incluída uma mensagem adicional, chamada *startup*, que é enviada pelos nós em N_0 para todos os seus vizinhos quando iniciam sua computação. Essa mensagem, uma vez que alcança um nó que não se encontra em N_0 pela primeira vez, tem o objetivo de "acordar" este nó e para que ele possa passar essa informação a todos os seus vizinhos.

Pode-se imaginar essa mensagem *startup* como sendo uma mensagem $safe(-1)$ que é propagada pelo grafo, e tem o objetivo de transmitir aos nós que não estão em N_0 a informação de que eles devem participar do pulso $s = 0$ também, assim como em todos os outros pulsos.

Todos os nós somente seguem executando o pulso $s = 0$ após receber a mensagem *startup* de todos os vizinhos. Isso é controlado através da variável go_i^j, inicializada com $false$, que indica se um *startup* foi recebido de n_j. Uma variável adicional, $initiated_i$, inicializada com $false$, indica se n_i pertence a N_0.

2.3.2 Sincronizador Beta

O sincronizador Beta requer que uma árvore geradora (spanning tree) seja estabelecida no grafo G, antes de inicializar o algoritmo propriamente dito, pois é através desta árvore que as mensagens de comunicação do sincronizador irão circular.

Além da determinação da spanning tree, também é necessário eleger um líder nessa árvore. A função do líder no sincronizador Beta é coletar de todos os outros nós a informação de "seguro" necessária para prosseguir para os próximos pulsos, e em seguida passar essa informação para todos eles.

O sincronizador funciona da seguinte maneira: quando um nó que não é o líder se torna seguro em relação a um determinado pulso, e já recebeu a mensagem $safe$

de todos os seus vizinhos (exceto o pai na árvore), ele envia uma mensagem $safe$ para o único vizinho de quem ele não recebeu um $safe$. Como já foi dito, um nó se torna seguro quando recebeu um *ack* para todas as mensagens de computação que ele enviou em um determinado pulso.

O líder, ao receber a mensagem $safe$ de todas as arestas da árvore que são incidentes a ele, e estando ele seguro em relação àquele pulso, transmite uma mensagem através da árvore indicando que a computação de um novo pulso pode ser iniciada. Essa mensagem é semelhante a um $safe$, e então, a regra para um nó prosseguir para outro pulso é fazê-lo depois de ter recebido essa mensagem de seu pai na árvore.

Vamos apresentar a seguir o algoritmo *A_Alg(Beta)* que mostra o funcionamento do sincronizador Beta. Neste algoritmo, $Filhos_i$ representa o conjunto dos nós que são filhos de n_i na árvore geradora.

Algoritmo A_Alg(Beta):

▷ **Variáveis:**

$s_i = 0$;

$MSG_i(s) = \emptyset$ para todo $s \geq 0$;

$expected_i(s) = 0$ para todo $s \geq 0$;

$safe_i^j(s) = false$ para todo $n_j \in Neig_i$ e todo $s \geq 0$;

▷ **Entrada:**

$msg_i = nil$

Ação Se n_i é líder:

Envie $start0$ para todo $n_i \in Filhos_i$;

Se $n_i \in N_0$:

Início

Faça alguma computação;

Envie $comp\_msg(s_i)$ em cada aresta de um subconjunto (pode ser vazio) de Out_i;

Fim

▷ **Entrada:**

$msg_i = start0$ tal que $origem_i(msg_i) = (n_i, n_j)$

Ação

Envie $start0$ para todo $n_i \in Filhos_i$;

Se $n_i \in N_0$:

Início

Faça alguma computação;

Envie $comp\_msg(s_i)$ em cada aresta de um subconjunto (pode ser vazio) de Out_i;

Fim

Se $Filhos_i = \emptyset$ e $expected(s_i) = 0$ (nó é folha)

Envie $safe(s_i)$ para $parent_i$;

Senão

Envie $start0$ para todo $n_i \in Filhos_i$;

▷ **Entrada:**

$msg_i = startnew$ tal que $origem_i(msg_i) = (n_i, n_j)$

Ação:

$s_i = s_i + 1$;

Se $Filhos_i \neq \emptyset$ (nó não é folha)

Envie $startnew$ para todo $n_k \in Filhos_i$;

Faça alguma computação;

Envie $comp\_msg(s_i)$ em cada aresta de um subconjunto (pode ser vazio) de Out_i;

▷ **Entrada:**

$msg_i = comp\_msg(s)$ tal que $origem_i(msg_i) = (n_i, n_j)$

Ação:

$MSG_i(s+1) = MSG_i(s+1) \bigcup \{msg_i\}$;

Envie $ack(s)$ para n_j;

▷ **Entrada:**

$msg_i = ack(s)$

Ação:

$expected_i(s) = expected_i(s) - 1$;

Se $expected(s_i) = 0$ então

Se $safe_i^k(s_i)$ para todo $n_k \in Filhos_i$ então

Se n_i não é líder

Envie $safe(s_i)$ para $parent_i$;

Senão (nó é líder)

Início

$s_i = s_i + 1$;

Envie *startnew* para todo $n_k \in Filhos_i$;

Faça alguma computação;

Envie $comp\_msg(s_i)$ em cada aresta de um subconjunto (pode ser vazio) de Out_i;

Fim

▷ **Entrada:**

$msg_i = safe(s)$ tal que $origem_i(msg_i) = (n_i, n_j)$

Ação:

$safe_i^j(s) = true$;

Se $safe_i^k(s_i)$ para todo $n_k \in Filhos_i$ então

Se $expected_i(s_i) = 0$ então

Se n_i não é líder

Envie $safe(s_i)$ para $parent_i$;

Senão (nó é líder)

Início

$s_i = s_i + 1$;

Envie *startnew* para todo $n_k \in Filhos_i$;

Faça alguma computação;

Envie $comp\_msg(s_i)$ em cada aresta de um subconjunto (pode ser vazio) de Out_i;

Fim

Como se pode observar, algumas variáveis foram incluídas para garantir a sincronização do algoritmo assíncrono. A variável líder identifica se o nó é a raiz da árvore. Como no sincronizador Alfa, a variável $expected_i(s)$, inicialmente igual a zero, guarda para todo $s \geq 0$ o número de $ack(s)$'s que n_i espera. Assume-se que essa variável é incrementada toda vez que n_i envia uma $comp\_msg(s)$, portanto essa ação está embutida na linha do algoritmo "Envie $comp\_msg$...". Neste algoritmo, $Filhos_i$ identifica o conjunto dos nós que são filhos de n_i na árvore geradora, que será vazio no caso de n_i ser uma folha.

O nó n_i também mantém a variável $safe_i^j(s)$ para cada filho na árvore geradora n_j, para todo $s \geq 0$, inicialmente recebendo o valor $false$ e utilizado para indicar que uma mensagem $safe(s)$ foi recebida de n_j.

2.3.3 Sincronizador Gama

O sincronizador gama requer uma fase de inicialização na qual a rede é dividida em grupos (clusters). Dentro de cada grupo o algoritmo funciona conforme o sincronizador Beta e entre os grupos o algoritmo executa como o sincronizador Alfa.

Cada grupo possui uma árvore geradora e um líder, que é a raiz da árvore, necessários para a execução do sincronizador Beta. Para cada dois grupos vizinhos (dois grupos são vizinhos se existe uma aresta entre os nós dos grupos) uma das arestas entre os dois grupos é escolhida como predileta. Através desta aresta ocorre a comunicação entre dois clusters. [16]

No caso extremo, onde cada cluster consiste de apenas um nó, o algoritmo Gama se comporta como se fosse o Alfa, enquanto que no caso onde existe apenas um cluster contendo todos os nós, Gama funciona como se fosse o Beta.

Capítulo 3

Implementação do Ambiente

Tendo em vista que o resultado final deste ambiente de desenvolvimento de algoritmo distribuídos é um programa executável que realiza a execução do algoritmo desejado, pode-se dividir a implementação deste ambiente basicamente em duas partes: a primeira se refere à interface gráfica, onde o usuário deverá informar os dados relativos ao algoritmo que deseja desenvolver; a segunda parte diz respeito à utilização dos dados do algoritmo informado na geração do programa final. As seções seguintes descrevem cada uma destas partes.

3.1 Interface Gráfica

A interface gráfica do ambiente proposto foi desenvolvida para Web, utilizando o padrão HTML (Hypertext Markup Language) e a interface CGI (Common Gateway Interface). Os programas para CGI foram desenvolvidos utilizando a linguagem de programação C, por esta oferecer os recursos necessários para a implementação do ambiente.

O ambiente gerado apresenta ao usuário basicamente cinco telas diferentes. Na primeira tela, o usuário pode escolher se deseja criar um novo algoritmo distribuído ou alterar os dados de um algoritmo já criado.

A segunda e terceira telas permitem que o usuário informe os dados gerais e específicos referentes ao algoritmo desejado. A partir destes dados fornecidos, o programa executável correspondente ao algoritmo é gerado, e o resultado deste processo é apresentado ao usuário numa quarta tela, onde ele poderá observar se o resultado foi com sucesso ou ocorreu algum erro na geração.

Na quinta tela, os arquivos gerados são disponibilizados ao usuário, para que sejam gravados em diretórios locais. Cada uma dessas telas são descritas com mais

detalhes nas seções seguintes.

3.1.1 Tela Inicial do Ambiente

Assim que o usuário inicia o ambiente de desenvolvimento, uma primeira tela é apresentada onde ele tem à sua disposição duas funcionalidades: a primeira é a criação de um algoritmo distribuído, ou seja, um algoritmo que ainda não foi utilizado neste ambiente; a segunda é alterar a especificação dos dados informados referentes a um algoritmo que já foi utilizado. Esta tela é apresentada na figura 3.1:

Como se pode observar são apresentadas três opções para seleção: Novo Algoritmo, Abrir Algoritmo já existente; Abrir Algoritmo já existente com alteração no número de pares Entrada/Ação.

A opção "Novo Algoritmo" permite que um algoritmo que ainda não foi utilizado nesse ambiente seja gerado. A segunda possibilidade é alterar os dados de um algoritmo que já foi criado, neste caso a opção "Abrir Algoritmo já existente" deve ser escolhida. A última opção, "Abrir Algoritmo já existente com alteração no número de pares Entrada/Ação" permite que, além de modificar os dados gerais do algoritmo, o número de pares Entrada/Ação também possa ser alterado, ou seja, pode-se incluir ou excluir alguma ação de um algoritmo que já havia sido criado.

A separação referente às opções (segunda e terceira) de alterações nos dados do algoritmo se deve ao fato do número de pares Entrada/Ação ser gerado dinamicamente, assim que o usuário informa esse número.

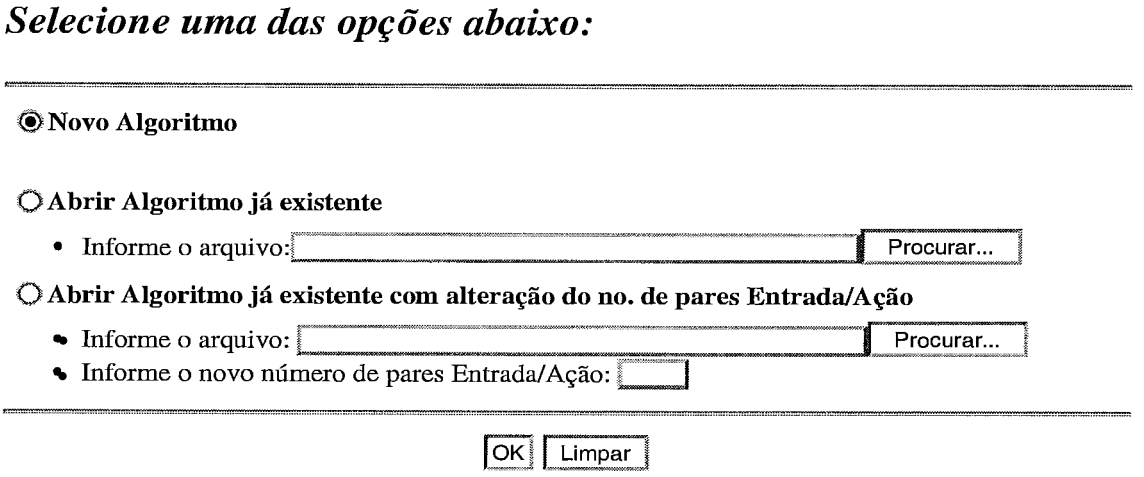

Figura 3.1: Interface Gráfica - Tela Inicial

3.1.2 Criando um Algoritmo Novo

Na tela inicial de seleção, o usuário deve selecionar a opção "Novo Algoritmo" quando desejar criar um algoritmo que ainda não tenha sido gerado nesse ambiente de desenvolvimento. Após selecionar esta opção, o usuário pode clicar no botão **OK** que envia esta informação para o servidor WWW. O servidor retorna uma tela para que o usuário forneça as informações gerais do algoritmo que está sendo desenvolvido. Esta tela é ilustrada nas figuras 3.2 e 3.3.

Estas informações gerais contemplam dados como o nome do algoritmo; o número de nós ou processos que devem executar; a topologia desejada, ou seja, como estes processos se ligam; se o algoritmo é síncrono ou assíncrono; entre outros. Dentre estes dados alguns são considerados obrigatórios para a geração do programa que realizará a execução. São eles:

- Nome do Algoritmo
- Tipo de Algoritmo (Síncrono ou Assíncrono)
- Tipo de Topologia
- Número de Nós
- Tipo e número de elementos da mensagem
- Número máximo de mensagens em trânsito
- Plataforma utilizada
- Processadores onde cada processo irá executar
- Número de Pares Entrada/Ação

O primeiro campo da tela corresponde ao nome do algoritmo novo, e como já foi dito, este campo é obrigatório. Este nome será utilizado na geração dos programas necessários.

O próximo campo determina o tipo de algoritmo que está sendo criado. Pode ser Assíncrono, ou seja, as ações ocorrem no momento do recebimento de uma mensagem, ou Síncrono, quando as ações acontecem dependendo do tempo de um relógio global.

Informações Gerais:

Informe o Nome do algoritmo:

Informe o Tipo de Algoritmo: (•) Assíncrono ( ) Síncrono

Caso tenha escolhido Tipo de Algoritmo "Síncrono", selecione o sincronizador desejado:

Sincronizador: Alpha

Escolha a Topologia: anel nao-direcionado

Informe o Número de Nós/Dx/Dy: (Dx e Dy são preenchidos somente para topologia malha e torus)

Caso tenha escolhido a opção "grafo generico" no campo de Topologia, informe um dos dois campos abaixo:

Descrição da Topologia:

Informe o Arquivo que possui a descrição da Topologia:

Procurar...

Nós que iniciam o Algoritmo (No alg. assíncrono são os nós que iniciam espontaneamente; e no alg. síncrono são os nós que realizam ação no pulso 0)

Número de nós que iniciam o Algoritmo:

☐ **Marque aqui se TODOS os Nós iniciam o Algoritmo**

OU

Digite o Número dos Nós que iniciam o Algoritmo:

OU

Informe o Arquivo com o número dos nós que iniciam o Algoritmo:

Procurar...

Tipo (em C) e número de elementos da mensagem:(ex: char 2)

Número máximo de mensagens em trânsito:

Figura 3.2: Interface Gráfica - Dados Gerais I

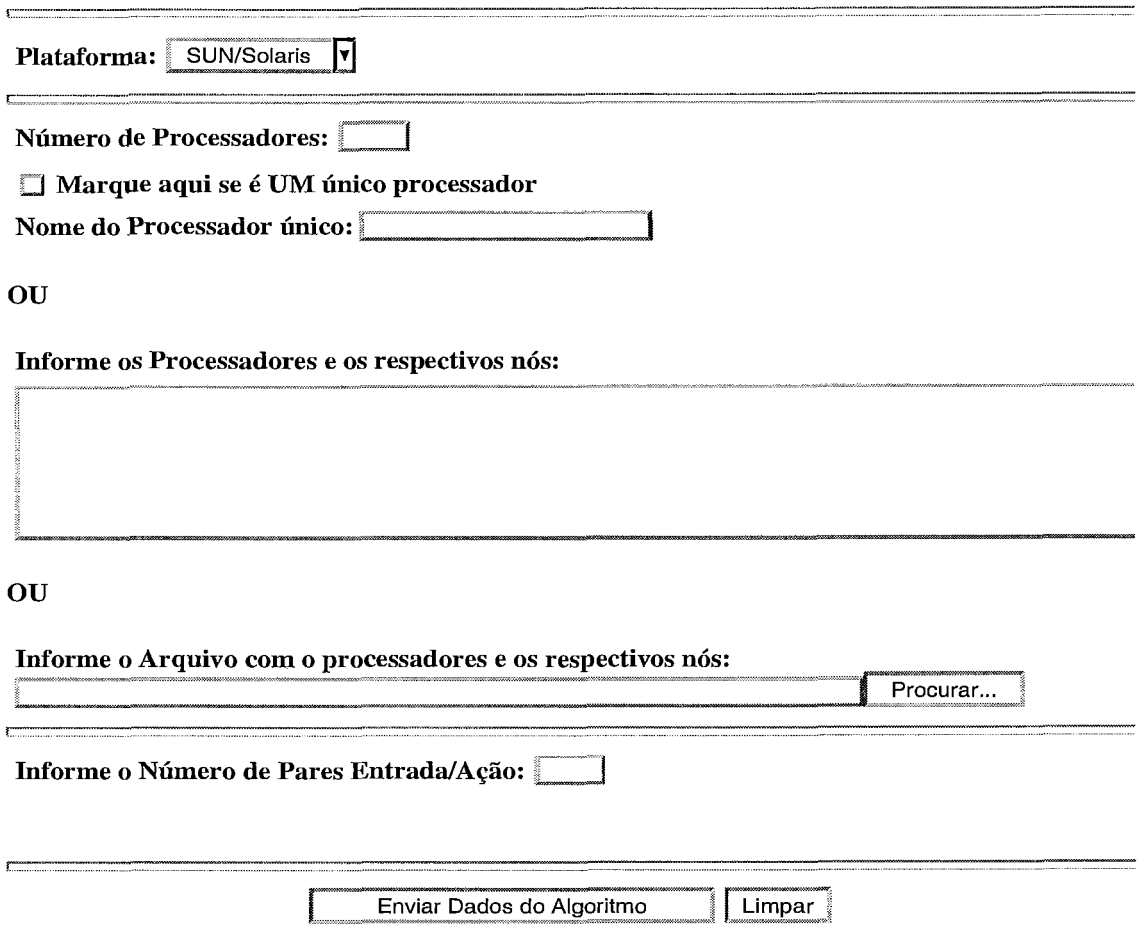

Figura 3.3: Interface Gráfica - Dados Gerais II

O campo Topologia identifica como os processos se interligam. Nesta lista existem opções que correspondem a grafos pré-definidos, como anel não-direcionado, anel direcionado, malha, torus, hipercubo, grafo completo, e uma opção geral que é especificada como grafo genérico não-direcionado e grafo genérico direcionado.

Neste último caso, o usuário deve indicar como os processos se interligam. Esta informação deve ser descrita utilizando o formato definido para este campo, onde cada linha contém os vizinhos do processo correspondente ao número da linha. Somente os vizinhos de número menor (no caso de grafo não-direcionado) ou vizinhos de saída (no caso de grafo direcionado) devem ser descritos. Para identificar a topologia, quando as opções "grafo genérico não-direcionado" e "grafo genérico direcionado" são selecionadas, o usuário poderá descrevê-la no campo Descrição da Topologia ou indicar o caminho e o nome de um arquivo no campo Arquivo, logo abaixo do campo Descrição da Topologia. Tanto a descrição como o conteúdo do arquivo devem ser digitados segundo o formato específico definido para topologia.

O campo Número de Nós/Dx/Dy contém a descrição de três informações. A primeira delas corresponde ao número total de nós que deverão executar o algoritmo e é sempre obrigatória. Os outros dois dados identificam as dimensões X e Y , respectivamente, para os tipos de topologia Malha e Torus.

Por exemplo, uma malha de seis processos onde a dimensão X é de tamanho dois e a dimensão Y é de tamanho três, seria indicada neste campo como *6 2 3* (os dados são separados por espaço(s) em branco).

O campo Número de nós que iniciam o algoritmo determina quantos nós (ou processos) iniciam o algoritmo espontaneamente, para o caso de algoritmo assíncrono. No caso de algoritmo síncrono este campo determina o número de processos que executam o primeiro pulso do algoritmo.

Para discriminar quais são estes nós, o usuário possui três alternativas: marcar o *checkbox Todos*, indicando que todos os nós iniciam; especificar o número destes nós, separados por espaço(s) em branco na caixa de texto, ou informar o nome e o caminho de um arquivo que contenha estes nós.

Os campos Tipo (em C) e número de elementos da mensagem se referem às mensagens que irão transitar durante a comunicação entre os processos. Todas as mensagens devem possuir um único tipo. No caso de um tipo composto, ou seja, uma estrutura na linguagem C, o nome da estrutura deve ser informado e a declaração desta estrutura deve ser informada no campo Variáveis do Algoritmo, numa tela que

veremos mais adiante, que contempla a apresentação do código do algoritmo.

O número de elementos da mensagem representa um elemento único, quando é igual a 1, ou um *array* quando é maior que 1. Tanto o tipo de mensagem quanto o número de elementos são campos obrigatórios e devem sempre ser informados.

O próximo campo é o Número máximo de mensagens em trânsito. Este campo permite que o usuário controle a capacidade do buffer de comunicação, podendo realizar diferentes experimentos.

No campo Plataforma o usuário poderá identificar em qual sistema o programa gerado irá executar. As opções disponíveis na lista são Sun/Solaris, IBM/Aix e SP2, porém somente a opção Sun/Solaris foi implementada.

Os campos Número de Processadores e Processadores determinam como os nós se distribuem entres os processadores da rede. O primeiro campo identifica o número de processadores diferentes nos quais os nós serão alocados.

Para determinar a alocação, o usuário tem disponível três opções: marcar a opção "Um único processador" e indicar o nome deste processador na caixa de texto ao lado, significando que todos os nós estarão alocados àquele processador; descrever um a um os processadores na caixa de texto abaixo; ou indicar o caminho e nome do arquivo que contenha as informações de alocação.

Esta informação deve ser indicada no formato definido para este campo, ou seja, cada linha contém o nome do processador ou endereço IP, seguido do número de todos os nós que estão alocados a ele. Por exemplo, supondo que existam dois processadores chamados "miami" e "porto" e três nós, se o usuário desejar que os nós 0 e 2 executem em "porto" e o nó 1 execute na máquina "miami", esta descrição seria como segue:

```
miami 1
porto 0 2
```

Finalmente, o campo Número de Pares Entrada/Ação determina o número de ações diferentes que podem ser disparadas durante a execução do algoritmo.

A descrição de cada uma destas ações será realizada na tela de Informações do Algoritmo.

Ao clicar o botão **Enviar Dados do Algoritmo**, as informações contidas nesta tela são enviadas ao servidor WWW através de um programa CGI. Neste programa

os dados são verificados e, caso sejam válidos, o programa exibe uma tela que contempla os dados gerais do algoritmo já informados além de campos para a entrada dos dados referentes à programação do algoritmo. Esta tela é ilustrada nas figuras 3.4 e 3.5. Se algum dos dados obrigatórios não foi informado ou houve algum erro, o programa retorna uma mensagem de erro específica ao usuário.

Esta tela permite que o usuário descreva os dados de programação do algoritmo. Seguindo o modelo apresentado em [2], pode-se implementar o código de um algoritmo se baseando nas seguintes informações:

- Variáveis / Inicialização das Variáveis
- Descrição dos pares Entrada/Ação
- Condição de Término do Algoritmo

A descrição destas informações deve ser realizada utilizando a linguagem C, pois esta foi definida como a linguagem de desenvolvimento do programa gerado como resultado do ambiente.

Na interface gráfica, o campo "Variáveis" contém a descrição das variáveis do algoritmo, assim como a declaração de tipos que possam ser utilizados no mesmo. Caso o usuário defina um tipo de mensagem que não seja um tipo básico do C, isto é, que necessite definir uma estrutura em C, é neste campo que esta estrutura deve ser definida. Além disso, esta declaração deve utilizar o padrão "typedef struct" da linguagem C. O usuário também poderá informar a descrição das variáveis através da indicação do caminho e o nome de um arquivo no campo Arquivo, logo abaixo do campo Variáveis.

O próximo campo representa a "Inicialização das Variáveis", ou seja, o usuário tem a opção de inicializar suas variáveis no momento da declaração das mesmas, ou neste campo, o que permite maior flexibilidade, como a possibilidade de utilizar um código do tipo "for(i=0; i=10; i++)...", entre outros.

Este campo é opcional, podendo ser deixado em branco, quando desejado. O usuário também poderá informar a inicialização das variáveis através da indicação do caminho e o nome de um arquivo no campo Arquivo, contendo esta informação.

O campo "Condição de Término" representa uma expressão booleana que, quando satisfeita para um determinado processo, força o término da execução do algoritmo neste processo. Ela é formada de apenas uma linha que contém a expressão

Informe as Bibliotecas C utilizadas pelo Algoritmo (Opcional):

OU

Informe o arquivo que contém as Bibliotecas:

Procurar...

Informe as Variáveis do Algoritmo:

OU

Informe o arquivo com as variáveis:

Procurar...

Informe o código para Inicialização de Variáveis (Opcional):

OU

Informe o arquivo para inicialização das variáveis:

Procurar...

Informe a Condição de Término do Algoritmo:

Figura 3.4: Interface Gráfica - Dados Específicos I

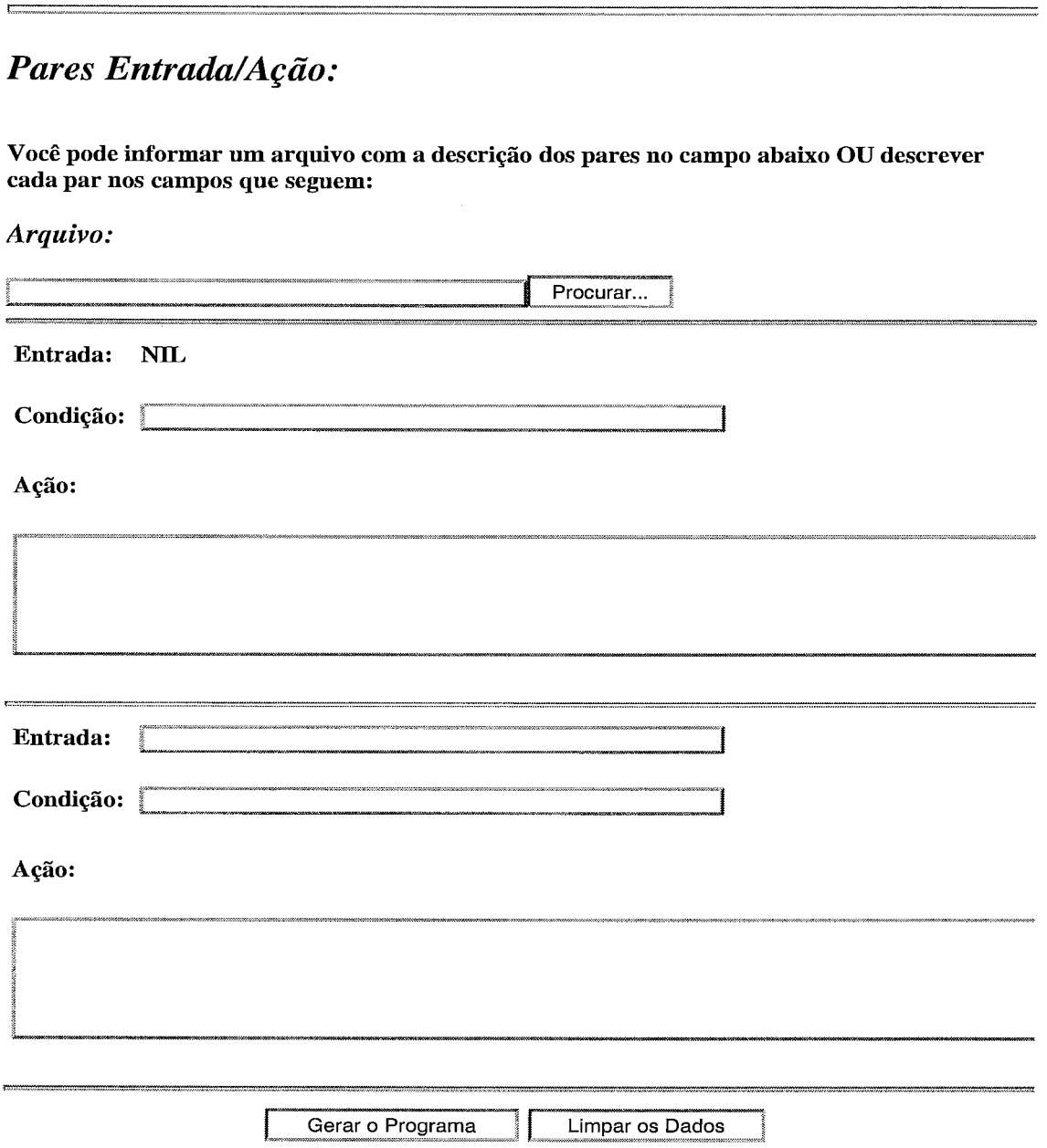

Figura 3.5: Interface Gráfica - Dados Específicos II

descrita na linguagem C. Um exemplo desta expressão seria *cont==10* indicando que, quando a variável *cont* chegar ao valor 10, o programa será finalizado.

O ambiente implementado se utiliza apenas da condição de término local para um processos do algoritmo, ou seja, somente os algoritmos que possuem uma grande regularidade podem ser desenvolvidos neste ambiente. Os algoritmos distribuídos que não possuem muita regularidade precisam ser analisados de uma perspectiva geral, através de uma condição de terminação global. [2]

De acordo com o número de pares Entrada/Ação indicado anteriormente, na tela de Informações Gerais, são criados os campos para a entrada dos dados referentes à cada par. O número e o conteúdo destes campos varia de acordo com o tipo de algoritmo escolhido, ou seja, se é síncrono ou assíncrono.

Para algoritmos assíncronos, a Entrada de cada par está associada a uma Condição. No campo "Entrada" o usuário deve informar um TAG, ou seja, uma constante que identifica o tipo de mensagem que corresponde à ação do par. No primeiro par do algoritmo assíncrono este TAG é fixo e igual a **NIL**, representando que a ação correspondente será executada por todos os processos que iniciam espontaneamente, independente do recebimento de uma mensagem, conforme padrão no qual a implementação deste ambiente foi baseada. Para os demais pares, este TAG deve ser iniciado por uma letra, e é sempre obrigatório. No programa gerado, este TAG será uma constante em C, que identifica o tipo da mensagem.

O campo "Condição" permite que o usuário teste uma condição antes de realizar a ação referente ao par em questão. É constituído de uma expressão booleana descrita utilizando a linguagem de programação C. Caso o usuário não preencha este campo, assume-se que a ação será executada, incondicionalmente, no momento do recebimento da mensagem, o que corresponde a uma condição sempre verdadeira, ou "1" na linguagem C.

O campo "Ação" consiste numa seqüência de comandos na linguagem C, que devem ser executados no momento do recebimento de uma mensagem do tipo especificado no campo "Entrada" caso a condição especificada no campo "Condição" seja satisfeita. Além dos comandos da linguagem C, algumas rotinas foram implementadas e estão disponíveis ao usuário, de forma a facilitar a codificação do algoritmo. São elas:

- send_to

- send_all
- vizinho
- calc_num_viz

A rotina **send_to** realiza o envio de uma mensagem para um dos vizinhos do processo que a chama. A declaração desta função é send_to(dest tag msg), onde *dest* corresponde ao processo que vai receber a mensagem, *tag* representa o tipo da mensagem e *msg* é o nome da variável que contém a mensagem a ser enviada.

Outra rotina disponível ao usuário é **send_all** que realiza o envio de uma mensagem para todos os vizinhos do processo que a invocou. Sua declaração é send_all(tag msg), sendo *tag* o tipo da mensagem que será enviada e *msg* o nome da variável que contém esta mensagem. Pode-se notar que a diferença entre as rotinas **send_to** e **send_all** é que a primeira possui um parâmetro a mais que se refere ao processo destino do send.

Um exemplo destas duas rotinas se encontra abaixo:

```
send_to(3 TAG1 nome);
send_all(TAG2 letra);
```

No primeiro exemplo, uma mensagem será enviada para o processo 3 com tag 'TAG1' cuja mensagem é o conteúdo da variável nome. De forma semelhante, o segundo exemplo identifica que uma mensagem com tag 'TAG2' será enviada para todos os vizinhos do processo , e a mensagem é o conteúdo da variável letra.

A rotina **vizinho** retorna um indicador se o processo passado como parâmetro é vizinho do processo que está chamando a rotina. Sua declaração é vizinho(proc, tipo) onde *proc* corresponde ao número do processo que se deseja saber se é vizinho e *tipo*, que determina qual a natureza do vizinho. Os tipos de vizinhos possíveis, correspondendo ao segundo parâmetro da função são 'I' de entrada (**I**nput), 'O' de saída (**O**utput) e 'B' se for ambos (**B**oth), ou seja, de entrada e saída.

A rotina **calc_num_viz**, como o próprio nome diz, retorna o número de vizinhos de um determinado tipo, que é passado como parâmetro. Ela possui apenas um parâmetro, correspondendo ao tipo de vizinho em questão, e seu retorno corresponde ao número de vizinhos referentes ao tipo indicado. Sua declaração é então calc_num_viz(tipo), onde tipo pode ser 'I' , 'O' ou 'B', como descrito anteriormente.

Diferente dos algoritmos assíncronos, nos algoritmos síncronos, não existe uma Condição atrelada ao campo "Entrada", pois o que vai determinar a execução da ação é o pulso do relógio e não o recebimento da mensagem. O campo "Entrada" contém, nesse caso, o intervalo de tempo correspondente à Ação do par considerado. Por exemplo, para o intervalo de s=2 até s=4, o algoritmo executa uma determinada ação. Para este par Entrada/Ação, o campo "Entrada" recebe, por exemplo, o valor (s>1&&s<5). Pode-se observar que este campo contém uma expressão booleana que representa um intervalo de tempo.

O primeiro par Entrada/Ação para algoritmos síncronos corresponde ao instante s==0, que é o momento do início do algoritmo. Isso é equivalente à ação dedicada aos processos que iniciam espontaneamente no algoritmo assíncrono.

O campo "Ação", de forma análoga ao algoritmo assíncrono, corresponde à uma lista de comandos na linguagem C, que serão executados caso a expressão descrita como Entrada para o par seja satisfeita. Todas as observações feitas para este campo em algoritmos assíncronos, vale para os algoritmos síncronos.

Uma outra forma de informar os pares Entrada/Ação é indicando o caminho e o nome de um arquivo no campo Arquivo, que contém a informação de todos os pares necessários, seguindo um padrão que foi definido para este arquivo, que pode ser verificado com mais detalhes no Apêndice B.

Após preencher todas as informações necessárias do algoritmo, o usuário pode iniciar a geração dos programas, clicando no botão **Gerar**. As informações contidas nesta tela serão então enviadas ao servidor WWW através de um programa CGI. Neste programa, além da validação de todos os dados informados pelo usuário, é realizada, efetivamente, a criação de um programa que executa o algoritmo distribuído especificado. O resultado deste processo de geração do programa é apresentado ao usuário na tela que é ilustrada na figura 3.6.

Nesta tela, além do resultado da geração do programa, são apresentadas ao usuário algumas informações do algoritmo para conferência.

Dados do Algoritmo:

Nome do Algoritmo = alg_teste

Número de Nós: 4

Número de Pares Entrada/Ação: 2

Tipo de Sincronizador: Alpha

Tipo de Algoritmo: Síncrono

Topologia: Anel Não-Direcionado

Tipo e tamanho da Msg: int 4

Número Max de Msg = 1

Número de Nós que iniciam espontaneamente = 4

Nós que iniciam espontaneamente = 0 1 2 3

Topologia: 4 Nós
Nó 0 possui 2 vizinhos:
Nó 3 (entrada e saída)
Nó 1 (entrada e saída)

Nó 1 possui 2 vizinhos:
Nó 2 (entrada e saída)
Nó 0 (entrada e saída)

Nó 2 possui 2 vizinhos:
Nó 3 (entrada e saída)
Nó 1 (entrada e saída)

Nó 3 possui 2 vizinhos:
Nó 2 (entrada e saída)
Nó 0 (entrada e saída)

RESULTADO DA GERAÇÃO DO PROGRAMA

SUCESSO - PROGRAMA GERADO

Salvar Arquivos

Figura 3.6: Interface Gráfica - Resultado da Geração do Programa

Se os programas forem gerados com sucesso, o usuário recebe uma mensagem indicando a finalização do processo. Neste momento, o usuário pode salvar localmente os arquivos gerados, para posteriormente executá-los. Para isto, basta clicar no botão **Salvar Arquivos**, que apresenta uma tela com os arquivos gerados, e um arquivo chamado "readme.txt" que explica ao usuário como o programa deve ser executado. Esta tela pode ser verificada na figura 3.7.

Além do arquivo readme.txt, mais quatro arquivos são disponibilizados para usuário. O primeiro deles é um arquivo executável, que não possui extensão, e seu nome é o mesmo do algoritmo gerado. O segundo é um programa que contém o mapeamento dos processos nos processadores. O terceiro é um arquivo que inicia a execução do programa executável gerado, utilizando as rotinas do padrão MPI. E finalmente, o quarto arquivo é o que contém os dados do algoritmo que foram informados pelo usuário. Este arquivo possui a extensão ".cfg" e pode ser utilizado quando o usuário desejar fazer alguma alteração no algoritmo.

Index of /arquivos/alg_teste

Name	Last modified	Size	Description
Parent Directory	04-Jul-1999 11:00	0k	
alg_teste	04-Jul-1999 11:00	337k	
alg_teste.aloc	04-Jul-1999 11:00	1k	
alg_teste.cfg	04-Jul-1999 11:00	1k	
execalg_teste	04-Jul-1999 11:00	1k	
readme.txt	04-Jul-1999 11:00	1k	

Ambiente para Desenvolvimento de Algoritmos Distribuidos

Este diretorio possui cinco arquivos referentes ao algoritmo 'alg_teste':

alg_teste - arquivo executavel do algoritmo

alg_teste.aloc - arquivo de alocacao de processos a processadores

alg_teste.cfg - arquivo de configuracao. (utilizado para fazer alteracoes

execalg_teste - arquivo que inicia a execucao do algoritmo distribuido

readme.txt - arquivo que contem as informações apresentadas aqui

OBS: 1 - A palavra PATH_COMPLETO no arquivo '.aloc' deve ser alterada para o ca
completo de onde estah o arquivo executavel

2 - Antes de executar o arquivo execalg_teste, sua permissao deve ser alte
atraves do comando 'chmod 755 execalg_teste'.
O mesmo deve ser feito para o programa alg_teste atraves do comando
'chmod 755 alg_teste'

Figura 3.7: Interface Gráfica - Salvar Arquivos

3.1.3 Editando um Algoritmo já Criado

Após ter gerado os programas referentes a um algoritmo, o usuário tem a possibilidade de alterar os dados informados, e gerar uma situação diferente para o mesmo algoritmo. Por exemplo, se o usuário gerou um algoritmo para uma topologia em anel com 4 processos, pode gerar novamente o mesmo algoritmo para a topologia em malha com 8 processos, bastando, para isso alterar os dados da topologia e gerar novamente.

Existem duas opções para o usuário editar os dados de um algoritmo que já foi gerado. A primeira delas não permite a alteração no número de pares Entrada/Ação, ou seja, se o algoritmo foi definido com 4 pares, este número não poderá ser alterado. Esta opção é utilizada quando o usuário já possui a estrutura do algoritmo definida e está apenas variando topologia, número de processos, etc.

Para selecionar esta opção, o usuário deve clicar em **Abrir Algoritmo já existente** e informar no campo **Arquivo**, logo abaixo, o caminho e o nome do arquivo que contém os dados de configuração do algoritmo desejado, que possui a extensão ".cfg".

Após informado o nome do arquivo, o usuário pode clicar no botão **OK**, que envia as informações do arquivo para o servidor WWW. Uma tela contendo os dados gerais e de código do algoritmo é apresentada ao usuário, para que este possa realizar as alterações desejadas. Esta tela é idêntica àquela utilizada na criação do algoritmo.

Quase todos os campos permitem alterações, porém dois deles são muito específicos, e sua alteração exige a criação de um novo algoritmo. O primeiro deles é o tipo de algoritmo (síncrono ou assíncrono), e o outro é o número de pares Entrada/Ação. A razão pela qual este segundo campo não pode ser alterado é que, o número de campos na tela para entrada desta informação é gerado dinamicamente a partir do valor informado no arquivo de configuração.

Se o usuário deseja um número de pares Entrada/Ação diferente daquele definido na última vez que o algoritmo foi gerado, ele deve selecionar a opção **Abrir Algoritmo já existente com alteração do no. de pares Entrada/Ação** na tela de seleção, na qual, além do nome do arquivo de configuração ele deve informar, também, o novo número de pares Entrada/Ação que deseja implementar.

Analogamente, após clicar no botão **OK**, que envia as informações para o servidor WWW, é apresentada ao usuário uma tela com as informações do algoritmo especificadas no arquivo. Apenas o número de campos para os pares Entrada/Ação

é alterado, de acordo com o número informado pelo usuário. Conforme ocorre na opção anterior, também não é permitida a alteração do tipo de algoritmo.

Feitas todas as alterações desejadas, o usuário pode gerar os programas para este derivado do algoritmo anterior, pressionando o botão **Gerar**. As informações contidas nesta tela serão então enviadas ao servidor WWW para um programa CGI. Neste programa, os dados são validados e o programa que executa o algoritmo distribuído especificado é gerado. O resultado deste processo é apresentado ao usuário na tela seguinte. Após a geração, o usuário poderá salvar localmente os arquivos gerados, clicando no botão **Salvar Arquivos**.

3.2 Estrutura do Ambiente

Este ambiente de desenvolvimento foi implementado utilizando arquivos como forma de armazenamento das informações do algoritmo distribuído que foram fornecidas pelo usuário na interface gráfica. A criação desses arquivos se realiza em três fases. O diagrama ilustrado na figura 3.8, apresenta o funcionamento das fases para criação do programa fonte.

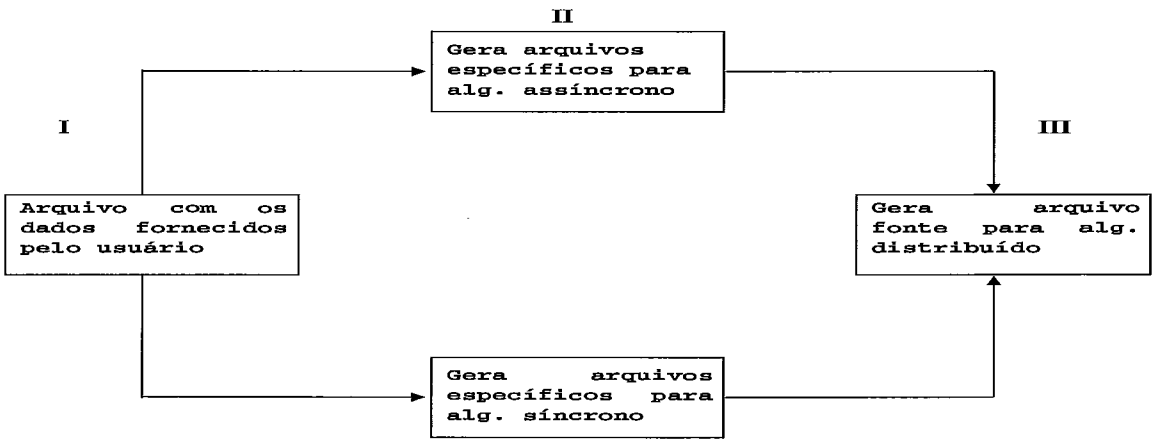

Figura 3.8: Esquema das Fases de Geração do Arquivo

Na primeira fase, as informações do usuário são guardadas em um arquivo, que foi chamado de 'arquivo de configuração'. O nome deste arquivo é constituído pelo nome do algoritmo, que foi indicado pelo usuário na interface gráfica, seguido da extensão '.cfg'. Neste arquivo, os dados se encontram, basicamente, da mesma forma

que o usuário informou nas telas da interface gráfica, apenas algumas codificações foram definidas, como por exemplo, para cada tipo de topologia foi associado um número.

Este arquivo gerado na primeira fase é utilizado na segunda fase como entrada de um programa, que divide este arquivo em vários outros pequenos arquivos, de acordo com as diversas características do ambiente (topologia, variáveis, etc.). O nome destes arquivos gerados iniciam por 'MPI' seguido do nome do algoritmo, informado pelo usuário, e uma extensão que identifica o tipo do arquivo. Por exemplo: se o nome do arquivo que o usuário deseja é "exemplo", os arquivos gerados terão o nome de: MPIexemplo.var; MPIexemplo.top, MPIexemplo.inic, etc.

O programa que realiza esta divisão depende do tipo de algoritmo, que pode ser assíncrono ou síncrono. Isso se deve ao fato de que cada um destes tipos possui um funcionamento diferente. Para os algoritmos síncronos, este programa, além de separar o arquivo de configuração em pequenos arquivos, ainda inclui um código adicional, referente à utilização de um sincronizador para converter este algoritmo em um algoritmo assíncrono.

Finalmente, na terceira fase, é gerado o arquivo fonte para o algoritmo distribuído, utilizando os arquivos criados na segunda fase. Esse arquivo fonte é um programa escrito na linguagem C utilizando o padrão MPI (Message Passing Interface). Esse arquivo é compilado, e o programa executável gerado é disponibilizado para o usuário.

3.2.1 Formato dos Arquivos

Este ambiente foi projetado para funcionar utilizando arquivos com as informações do algoritmo distribuído e, a partir de então gerar um código em C correspondente ao algoritmo desejado. Para isso, foram definidos formatos para cada um desses arquivos de forma a padronizar a identificação das informações necessárias.

Vários tipos de arquivos são gerados quando o usuário deseja desenvolver um algoritmo distribuído neste ambiente. O primeiro deles é o arquivo de configuração, que contempla todas as informações referentes ao algoritmo. Os demais são arquivos mais específicos, que foram criados para servirem de entrada para o programa que vai gerar o programa final.

Nas seções seguintes, cada um desses arquivos são apresentados, assim como o formato que foi definido para cada um deles.

Arquivo de Configuração

O formato do arquivo de configuração, gerado na primeira fase, foi definido de forma a facilitar a decodificação e a identificação de cada característica do algoritmo. Este arquivo contempla todas as informações necessárias para a geração de um programa fonte que executará o algoritmo. Seu formato é apresentado a seguir:

##CONFIG
[nome_algoritmo]
[número de nós]
[número de pares Entrada/Ação]
[sincronizador]
[orientação do grafo]
[tipo de grafo]
[tipo de sistema (SUN/Solaris, IBM/Aix, etc)]
[tipo e número de elementos da msg]
[número máximo de mensagens em trânsito]
[número de nós que iniciam espontaneamente]
[lista de nós que iniciam espontaneamente]
##TOPOL
...
descrição da topologia utilizando o formato específico
...
##PROC
...
descrição da alocação dos processos utilizando o formato específico
...

##HEADER

...

lista de bibliotecas (includes da linguagem C) e definição de constantes

...

##VAR

...

descrição das variáveis do algoritmo

...

##INIC

...

inicialização das variáveis do algoritmo

...

##TC

condição booleana que determina o término do algoritmo, quando satisfeita

##IAP0

Entrada para o par 0 (NIL para alg. assíncronos e s==0 para alg. síncronos)

condição para que a ação seja satisfeita (só se aplica para algoritmos assíncronos)

...

código em C que corresponde à ação para esse par

...

##IAP1

Entrada para o par 1

condição para que a ação seja satisfeita (só se aplica para algoritmos assíncronos)

...

código em C que corresponde à ação para esse par

...

##IAPn

Entrada para o par n

condição para que a ação seja satisfeita (só se aplica para algoritmos assíncronos)

...

código em C que corresponde à ação para esse par

...

##FIM

Como se pode observar, as diversas características que compõem o algoritmo são delimitadas por separadores constituídos pelos caracteres '##' seguidos de um texto que as identifica. Cada porção destas possui um formato próprio como podemos ver a seguir:

Dados Gerais de Configuração

A primeira linha é '##CONFIG' que identifica esta porção do arquivo.

A segunda linha, identificada por [nome_algoritmo], corresponde ao nome do algoritmo que foi informado pelo usuário no momento da entrada dos dados do algoritmo.

A linha [número de nós] identifica o número de processos que devem executar o algoritmo distribuído.

A próxima linha, referenciada por [número de pares Entrada/Ação] apresenta o número de pares Entrada/Ação que constituem o algoritmo.

O conteúdo da linha [sincronizador] identifica o nome do sincronizador que será utilizado no caso de algoritmo síncrono, que pode ser "Alpha" ou "Beta". Se o algoritmo em questão for assíncrono, esta linha contém o texto "Nenhum" indicando que nenhum sincronizador será utilizado.

Na linha [orientação do grafo] é incluído um número que identifica se as arestas que constituem o grafo (topologia) possui uma orientação, ou seja, se as arestas possuem uma direção única em que as mensagens transitam. Este número pode ser:

0 - grafo não é orientado (as mensagens são enviadas em ambas as direções sobre a aresta)

1 - grafo é orientado (as mensagens são enviadas em uma única direção sobre a aresta)

A linha [tipo de grafo] determina um número que identifica o tipo de grafo (topologia), isto é, como os processos se interligam. Abaixo segue a lista dos tipos de grafo contemplados pelo sistema:

0 – Anel não-direcionado

1 – Anel direcionado

2 – Malha

3 – Torus

4 – Hipercubo

5 – Grafo Completo

6 – Grafo genérico não-direcionado

7 – Grafo genérico direcionado

As duas últimas opções indicam que a descrição da topologia se encontra descrita no arquivo de configuração. Neste caso, ou a topologia foi digitada na tela da interface gráfica, ou o nome de um arquivo que contém esta descrição foi indicado pelo usuário.

A linha seguinte, referenciada por [tipo de sistema (SUN/Solaris, IBM/Aix, etc)], identifica em qual sistema o algoritmo deve ser executado. As possibilidades permitidas para esta opção são:

0 - SUN/Solaris
1 - IBM/Aix
2 - SP2

Todas as mensagens que serão utilizadas para a comunicação entre os processos para um mesmo algoritmo devem ser de um único tipo. Isto porque o formato da função que realiza o recebimento de mensagens (MPI_Receive) exige que o tipo da mensagem seja informado na chamada da mesma.[17]

Na linha [tipo e número de elementos da msg] o tipo de mensagem, descrito na linguagem C, é informado seguido do número de elementos desse tipo. Este campo pode ser um tipo pré-definido no C, como char, int, etc., assim como um tipo definido pelo usuário. Neste caso, o conteúdo desta linha é o nome do novo tipo, seguido pelo número de elementos deste tipo. A descrição do novo tipo deve ser incluída junto com a descrição das variáveis do algoritmo.

A linha seguinte, referenciada como [número máximo de mensagens em trânsito], identifica o número máximo de mensagens que podem estar transitando em um canal (aresta), ou seja, seria equivalente à capacidade de um buffer de mensagens. Para o processo de passagem de mensagens, foi implementada uma alternativa oferecida pelo MPI que é a utilização de um buffer para armazenar as mensagens enviadas para cada processo. Através dessa alternativa, pode-se personalizar o tamanho desse buffer de acordo com o tipo de experiência desejada.

A linha [número de nós que iniciam espontaneamente] identifica quantos processos devem iniciar o algoritmo, isto é, para algoritmos distribuídos assíncronos, são os processos que executam uma ação antes de receber qualquer mensagem; e para algoritmos síncronos, são os processos que executam uma ação no instante 0 (zero).

A última linha desta porção do Arquivo de Configuração contempla a relação dos processos que devem iniciar o algoritmo.

Topologia

Esta porção do Arquivo de Configuração apresenta a descrição de como os processos se interligam (topologia). Essa porção só consta nesse arquivo, caso o usuário tenha digitado essa descrição na interface gráfica, ou tenha informado um arquivo o qual contempla essa descrição. Isso significa que, se o usuário escolher uma topologia como anel, malha, torus, hipercubo ou grafo completo, a topologia não é incluída nesse arquivo.

Esta porção é iniciada pela linha com o texto '##TOPOL' e, as linhas seguintes constituem a descrição da topologia, que deve seguir o formato definido para essa característica. Nesse formato, cada linha possui os vizinhos correspondentes ao nó que possui a identidade igual ao valor do número da linha. Este formato é apresentado com mais detalhes no Apêndice B.

Alocação de Processadores

A descrição de como os processos estão alocados é incluída nesta porção do arquivo, a qual é iniciada pelo texto '##PROC'. As linhas seguintes constituem a relação dos processadores e processos, que deve seguir o formato definido para essa característica, onde existe uma linha correspondente a cada processador e nesta linha constam o nome do processador (máquina) seguido da lista dos processos que devem executar nesse processador. Este formato é apresentado com mais detalhes

no Apêndice B.

Definição das Bibliotecas C e Constantes

Esta porção do arquivo é iniciada pela expressão '##HEADER'. É aqui que o usuário deve identificar quais as bibliotecas C necessárias para a execução do algoritmo distribuído. A declaração das constantes também deve ser feita nessa porção do arquivo de configuração.

Um exemplo desse arquivo segue abaixo:

```
#include <stdio.h>
#include <stdlib.h>

#define NUMERO 10
#define TRUE 1
```

Definição das Variáveis

A expressão '##VAR' inicia a descrição das variáveis que fazem parte do algoritmo distribuído desejado. Essas variáveis devem ser definidas utilizando a linguagem C.

O usuário pode definir um tipo novo, o qual será o tipo das mensagens que serão enviadas na comunicação dos processos. Neste caso, este tipo novo deve ser declarado utilizando o padrão 'typedef struct', caso contrário, será apresentada uma mensagem de erro, e o programa não poderá continuar.

Segue abaixo um exemplo com a descrição de variáveis:

```
int num;
char nome[10];
typedef struct novo_tipo
   {
      int campo1;
      char campo2[10];
   } novo_tipo;
novo_tipo mensagem;
```

Definição da Inicialização das Variáveis

Esta porção do arquivo de configuração possui um código em C que atribui valores iniciais para as variáveis do algoritmo. Esta inicialização de variáveis é opcional, isto é, o usuário pode ou não definir. A expressão '##INIC' inicia esta porção do arquivo.

Segue abaixo um exemplo:

```
for(i=0; i<10; i++)
{
   dobro[i] = 2*i;
}
```

Condição de Término

Esta porção do arquivo é iniciada pela expressão '##TC' (*Termination Condition*) e define uma condição booleana que determina, quando satisfeita, o término do algoritmo para um processo. Esta condição é um texto de uma linha, escrito na linguagem C, que corresponde a uma expressão booleana. A seguir é apresentado um exemplo:

```
x == y
```

Corresponde à expressão se x é igual a y. Isso significa que, quando o valor da variável x for igual ao valor da variável y, o algoritmo para aquele processo deve terminar.

Definição dos Pares Entrada/Ação

Esta porção do arquivo contém a descrição dos pares que vão identificar o funcionamento do algoritmo. A estrutura seguinte se repete para cada par:

##IAPn
Entrada para o par n
Condição para que a ação seja satisfeita (só se aplica para algoritmos assíncronos)
...
código em C que corresponde à ação para esse par
...

Como se pode observar, a primeira linha contém um texto que indica o número do par Entrada/Ação. A segunda linha corresponde à entrada para o par, ou seja, o tipo de mensagem. A terceira linha identifica a condição booleana para que a ação seja executada, caso uma mensagem do tipo identificado como entrada seja recebido. E, a seguir, um texto, descrito em linguagem C, que corresponde à ação do par em questão.

O par inicial, que corresponde ao "IAP0", é um caso especial. Este representa, no caso de algoritmo assíncrono, a ação e condição que se aplica aos processos (nós) que iniciam espontaneamente, isto é, que iniciam sem que nenhuma mensagem seja recebida. No caso de algoritmo síncrono, corresponde a ação relativa ao pulso zero. Esses processos (nós) são os responsáveis pela inicialização do algoritmo distribuído. Para algoritmos assíncronos a Entrada para este par é 'NIL', e para algoritmos síncronos, é 's==0', onde s corresponde ao pulso do relógio global.

Finalização

O arquivo de configuração é finalizado por uma linha contendo o texto '##FIM'.

Arquivos Específicos

A partir do arquivo de configuração são gerados pequenos arquivos, separados por funcionalidade. Estes arquivos servirão de entrada para o programa responsável pela geração do programa final.

O arquivo de Topologia apresenta como os processos que irão executar o algoritmo distribuído se interligam. Seu nome é iniciado pela expressão 'MPI' seguida do nome do algoritmo e com a extensão '.top'.

O usuário possui três alternativas para informar a topologia. Selecionar uma topologia pré-definida ou informar uma topologia geral que não se enquadra nas topologias pré-definidas, de duas formas, como veremos a seguir.

Quando o usuário informa que deseja utilizar uma topologia pré-definida, o arquivo de topologia só é gerado na segunda fase do processo de criação do arquivo fonte. As topologias pré-definidas disponíveis ao usuário são anel, malha, torus, hipercubo e grafo completo.

Cada uma destas topologias pré-definidas possui características particulares. [8]

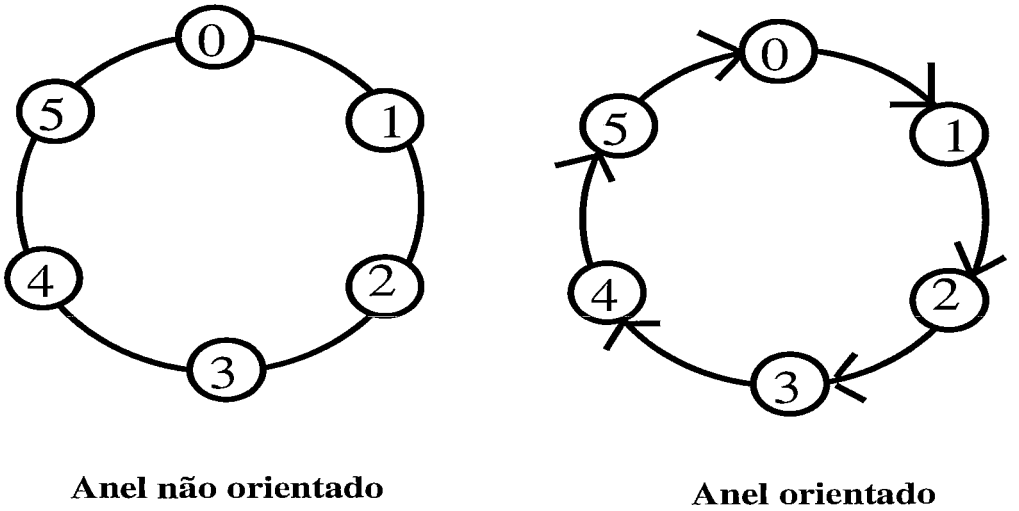

Figura 3.9: Grafo em Anel - Não Direcionado e Direcionado

O tipo de topologia anel é bastante popular, e se obtém conectando os dois processos terminais de um *array* linear através de uma aresta extra, ou seja, todos os processos estão ligados em linha e suas extremidades são unidas. O anel pode ser unidirecional, quando todas as arestas estão orientadas no mesmo sentido, ou bidirecional, quando as mensagens nas arestas podem ser enviadas em ambos os sentidos. No ambiente de desenvolvimento implementado, o anel unidirecional e bidirecional correspondem à topologia 'anel direcionado' e 'anel não-direcionado', respectivamente.

Para a geração do arquivo de topologia para anel, foi realizado um *loop* que define

os vizinhos de cada processo. No caso do processo 0 (zero), este possui dois vizinhos que são 1 e N–1, sendo N o número total de processos. Já o processo 2, possui vizinhos 1 e 3, e assim por diante. Desta forma três expressões foram identificadas:

processo	vizinhos
0	1 e N–1
x $\neq$ 0 e N–1	x–1 e x+1
N–1	0 e N–2

No caso de anel direcionado, foi definido que a orientação será no sentido horário. Desta forma, as expressões acima podem ser alteradas como se segue:

processo	vizinhos
0	1 (saída) e N-1 (entrada)
x $\neq$ 0 e N-1	x-1 (entrada) e x+1 (saída)
N-1	0 (saída) e N-2 (entrada)

O segundo tipo de topologia disponibilizado pelo ambiente é a malha. Na figura 3.10 pode-se observar uma malha 4 X 4. Em geral, em uma malha de dimensão k com N=n^k processos, os processos internos possuem grau igual a 2k e o diâmetro do grafo é k(n–1).

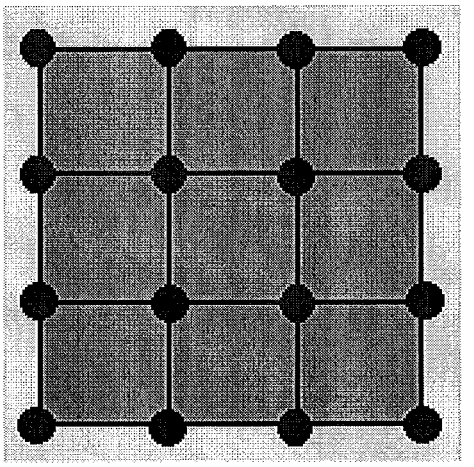

Figura 3.10: Grafo em Malha

Ao selecionar este tipo de topologia, além do número de processos, que já é obrigatório para as demais topologias, o usuário deve informar a dimensão X (número de processos no eixo x) e Y (número de processos no eixo y).

O arquivo de topologia resultante é construído, baseando-se nas informações das dimensões X e Y fornecidas pelo usuário. Inicialmente, uma matriz com as mesmas

dimensões da malha é alocada dinamicamente, onde cada elemento corresponde a um processo do grafo. Esta matriz é então inicializada da esquerda para direita e de cima para baixo, ou seja, o elemento [0][0] corresponde ao processo 0, [0][1] corresponde ao processo 1, e assim por diante.

Após inicializada a matriz, pode-se determinar facilmente os vizinhos de cada processo, bastando observar os dois índices da matriz para cada processo. Pode-se afirmar que cada processo possui, no máximo, quatro vizinhos, que seria o equivalente a incrementar e decrementar de um, cada um de seus índices. Por exemplo, seja uma malha de 3 X 3, o processo 4, que se localiza na posição [1][1] da matriz, possui os vizinhos [1-**1**][1], [1+**1**][1], [1][1-**1**], [1][1+**1**].

Uma atenção especial deve ser tomada em relação aos processos que se encontram nos limites, ou seja, quando algum de seus índices é igual a 0, Dx–1 ou Dy–1. Nestes casos, ao subtrair ou somar um a estes índices, irá gerar um elemento inexistente, e assim sendo o número de vizinhos para este processo é menor que quatro.

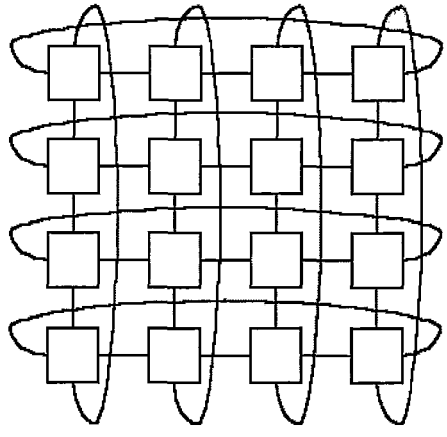

Figura 3.11: Grafo Torus

O torus é um tipo de topologia que pode ser visto como uma combinação de anel e malha e é extensivo a dimensões maiores. Possui conexões em forma de anel ao longo de cada linha e ao longo de cada coluna do *array*. Em geral, os processos de um torus binário **n X n** possuem grau 4 e o diâmetro de $2\lfloor n/2 \rfloor$.

A geração do arquivo de topologia para o tipo torus é bastante semelhante ao da topologia malha. Neste tipo, o usuário também deve informar a dimensão X (número de processos no eixo x) e Y (número de processos no eixo y) para a topologia, além do número de processos.

Uma matriz com as dimensões fornecidas pelo usuário é alocada dinamicamente,

onde cada elemento corresponde a um processo do grafo. Da mesma forma que a topologia Malha, esta matriz é inicializada da esquerda para direita e de cima para baixo.

Pode-se observar o mesmo procedimento realizado para topologia em malha. Neste caso, pode-se afirmar que cada processo possui, no máximo, quatro vizinhos, que seria o equivalente a incrementar e decrementar cada um de seus índices. Seguindo o procedimento realizado para malha, deve-se decrementar e incrementar de um cada um dos índices de cada processo.

Quando algum dos índices do processo é igual a 0, Dx–1 ou Dy–1, este se encontra nos limites do torus. Para estes casos, pode-se extrair duas regras como segue:

índice = –1
  substituir por $Dx - 1$ (eixo horizontal)
  substituir por $Dy - 1$ (eixo vertical)
índice = Dx ou Dy
  substituir por 0

Vejamos um exemplo: Em um torus de dimensão 3 X 3, o processo 0 (zero) corresponde ao elemento [0][0] da matriz. Ao determinar os vizinhos deste processo, verificamos que eles são [0–**1**][0], [0+**1**][0], [0][0–**1**] e [0][0+**1**].

O primeiro vizinho encontrado deve ser substituído pelo processo [2][0], pois essa variação corresponde ao eixo vertical, cuja dimensão é 3. O mesmo ocorre para o terceiro vizinho, que deve ser o processo [0][2], pois a variação ocorre no eixo horizontal, cuja dimensão é 3. Podemos, então, concluir que os vizinhos do processo 0 (zero) em um torus de dimensão 3 X 3 são 6, 3, 2 e 1.

A topologia Hipercubo é uma arquitetura cubo binário n-dimensional (*binary n-cube*). Em geral, um cubo n-dimensional consiste de $N = 2^n$ nós expandidos ao longo de n dimensões, com dois nós por dimensão. A figura 3.12 apresenta uma seqüência com hipercubos de dimensão 1, 2, 3 e 4. Uma topologia 4-cubo pode ser construído pela interconexão dos nós correspondentes em dois 3-cubos. Desta forma, para obter-se um cubo n-dimensional basta conectar os nós correspondentes de dois cubos (n–1)-dimensional.

O grau de cada nó em um n-cubo (cubo n-dimensional) é igual n, assim como o diâmetro da rede. Na realidade, o grau de cada processo (nó) aumenta linearmente

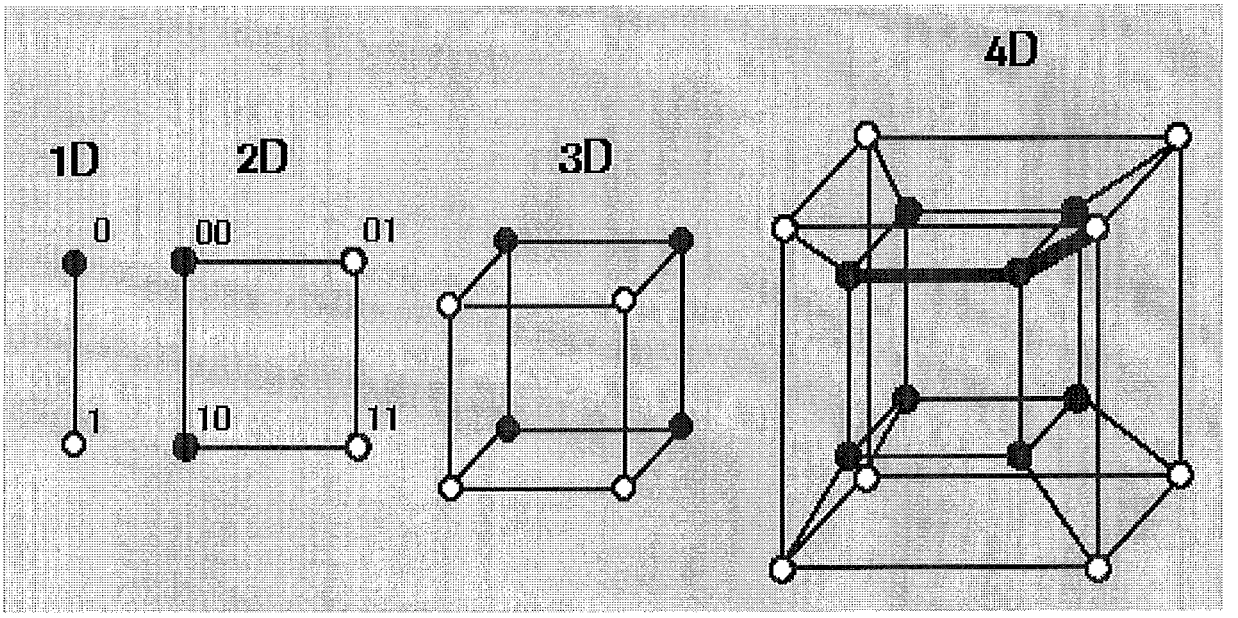

Figura 3.12: Grafo Hipercubo

com relação à dimensão.

Para a geração do arquivo da topologia hipercubo foram utilizadas duas características deste tipo de grafo. A primeira é que um hipercubo de dimensão k possui 2^k nós e cada nó possui k vizinhos. E, além disso, cada nó é ligado a todos os nós que diferem de seu valor de apenas um bit.

Por exemplo, para k = 4 (hipercubo de dimensão 4), os processos podem ser descritos utilizando 4 bits. O processo 3 (0011) possui os vizinhos 11 (**1**011), 7 (0**1**11), 1 (000**1**) e 2 (001**0**).

O grafo completo é a topologia onde cada processo do grafo se conecta com todos os demais processos do grafo. Desta forma, cada processo se comunica diretamente com cada processo do grafo, através de uma única aresta. A figura 3.13 ilustra um grafo completo com 8 nós.

A geração do arquivo para este tipo de topologia é o mais simples, pois podemos afirmar que todos os processos são vizinhos de todos os processos.

Quando o usuário deseja utilizar uma topologia diferente das pré-definidas, ele tem duas formas de informá-la: ou descrevendo-a no formulário, ou indicando um arquivo que contém a mesma.

Na primeira alternativa, o usuário pode informar a topologia desejada, selecionando o tipo de topologia "grafo genérico direcionado" ou "grafo genérico não-direcionado", e descrever a mesma no campo do formulário "Descrição da Topolo-

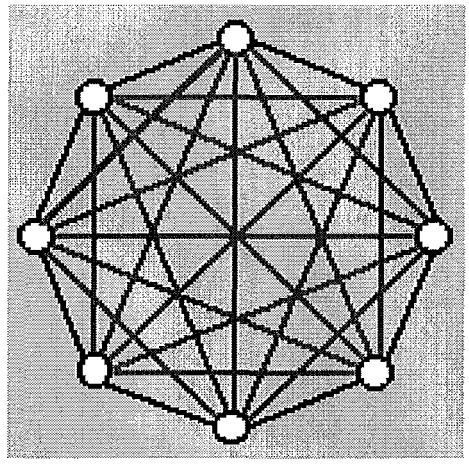

Figura 3.13: Grafo Completo com 8 nós

gia". Essa descrição deve seguir o padrão definido para topologia, que é apresentado no Apêndice B.

A segunda possibilidade que o usuário tem para informar a topologia é indicar o nome e o caminho de um arquivo que contemple a descrição da topologia desejada para realizar o algoritmo distribuído. Este arquivo deve possuir o formato baseado no padrão definido para topologia, o qual é apresentado com mais detalhes no Apêndice B.

Considerando as duas últimas alternativas, isto é, ou digitando a topologia no formulário ou indicando um arquivo que contenha esta descrição, esta informação não é gerada na segunda fase, como ocorre na primeira alternativa (topologia pré-definida). Nestes casos, a descrição é incluída no arquivo de configuração exatamente como o usuário informou.

Um outro arquivo gerado é o de Alocação de Processos a Processadores. Este arquivo contém a informação de como os processos se distribuem nos diversos processadores da rede. Seu nome é iniciado pela expressão 'MPI' seguida do nome do algoritmo e com a extensão '.aloc'.

Esses dados são fornecidos pelo usuário, que tem a possibilidade de descrevê-los de três formas diferentes.

A primeira delas consiste em todos os processos executarem em um mesmo processador. Desta forma, o usuário deve informar apenas o nome do processador desejado e o sistema gera automaticamente o arquivo de alocação.

A segunda alternativa é o usuário descrever no campo do formulário "Processadores" quais processos estão associados a quais processadores. Esta informação

deve ser fornecida seguindo o padrão estabelecido, que é baseado no formato utilizado pelo ch_p4 para a execução do programa gerado. [10] Neste padrão, cada linha corresponde a um processador. O primeiro argumento é o nome do processador e os dados subseqüentes são os números dos processos que irão executar no processador em questão, separados por espaço(s) em branco.

Abaixo segue um exemplo do padrão referenciado acima.

```
miami 1 2
lisboa 0 3 5
porto 4
```

Neste exemplo, estamos considerando que o algoritmo distribuído será executado por seis processos em três processadores (miami, lisboa e porto): os processos 1 e 2 serão executados no processador "miami"; os processos 0, 3 e 5 em "lisboa"; e o processo 4 no processador "porto".

O processo 0 (zero) tem um significado especial. Este é o processo inicializador da execução do algoritmo. Portanto, o processador no qual este processo deve estar alocado deve ser o mesmo no qual a execução deve ser disparada (started).

A terceira alternativa que o usuário tem para informar os processadores e seus respectivos processos é indicar o nome e o caminho de um arquivo que contemple esta informação. Este arquivo deve possuir o formato baseado no padrão definido anteriormente.

O número de linhas deste arquivo constitui o número total de processadores onde o algoritmo distribuído deverá executar.

O arquivo de Header contempla as bibliotecas da linguagem C (includes) necessárias para a execução do algoritmo distribuído que o usuário deseja executar, assim como a definição das constantes. Seu nome é iniciado pela expressão 'MPI' seguida do nome do algoritmo e com a extensão '.header'.

Para fornecer esses dados, o usuário deve se basear no código que será gerado para a execução do algoritmo, referente às ações dos pares Entrada/Ação e à inicialização das variáveis do algoritmo. O usuário possui duas alternativas para informar o Header. Pode-se digitar essa informação no campo do formulário, ou identificar o nome e caminho de um arquivo que contemple estes dados.

Esses dados devem ser descritos no formato padrão da linguagem C. Segue abai-

xo um exemplo deste arquivo:

```
#include <stdio.h>
#include <string.h>
#define NOME "Adriana"
#define CODIGO 01
```

O arquivo de Variáveis é composto do trecho de código em C que descreve a declaração das variáveis e tipos que serão utilizados pelo algoritmo distribuído. Seu nome é iniciado pela expressão 'MPI' seguida do nome do algoritmo e a extensão '.var'.

Foi especificado um tratamento especial para o caso da mensagem que será transmitida entre os processos não possuir um tipo padrão da linguagem C. Nesse caso, o usuário deve declarar esse tipo utilizando a estrutura "typedef" que se compõe da seguinte forma:

```
typedef struct NOME_TIPO
{
    tipo1 variavel1;
    tipo2 variavel2;
    ...
} NOME_TIPO;
NOME_TIPO variavel_msg;
```

Uma outra restrição para essa situação é que os tipos das variáveis dentro da estrutura têm de ser tipos padrão da linguagem C.

O arquivo com a Inicialização das Variáveis é destinado ao código que atribui valores iniciais para as variáveis que, por algum motivo, não foram inicializadas no momento da declaração. Seu nome é iniciado pela expressão 'MPI' seguida do nome do algoritmo e com a extensão '.inic'. Esse arquivo pode ser vazio, pois a informação para inicializar as variáveis não é obrigatória.

Outro arquivo criado é o que contempla a Condição de Término, uma expressão booleana que determina o fim da execução do algoritmo para os processos. Seu nome é iniciado pela expressão 'MPI' seguida do nome do algoritmo e a extensão

ıo se compõe de apenas uma linha e esta expressão booleana deve ser declarada utilizando a linguagem C.

Para representar os Pares Entrada/Ação foi criado um arquivo para cada par. Estes arquivos contemplam as ações para cada tipo de mensagem recebida (Entrada) por cada processo. Seu nome é iniciado pela expressão 'MPI' seguida do nome do algoritmo e a extensão '.iap.XX', onde 'XX' determina o número do par (ex. MPInome.iap.3).

Para cada par, a primeira linha corresponde a uma expressão identificando seu número, como por exemplo, "##IAP3", determinando que as linhas a seguir descrevem o terceiro par Entrada/Ação.

Em seguida, um TAG (Entrada) determina o tipo de mensagem correspondente ao par em questão. Este TAG será transformado em uma constante no programa gerado, e será utilizado para enviar e receber mensagens através das funções MPI_Send e MPI_Receive do MPI. [17].

Para os algoritmos assíncronos é possível atribuir uma condição para que, enquanto esta não for satisfeita, a ação referente ao recebimento de uma mensagem não será executada. Esta condição é uma expressão booleana, descrita na linguagem C, e é acrescentada neste arquivo na linha seguinte à Entrada do par.

As linhas seguintes correspondem à descrição da ação referente ao par Entrada/Ação. Esta ação se compõe de um código descrito em linguagem C espelhando o que deve acontecer quando a mensagem é recebida, de acordo com o algoritmo distribuído.

Além dos comandos da linguagem C, alguns procedimentos foram criados de forma a facilitar a codificação das ações para os pares. As rotinas relativas à comunicação entre processos são *send_to* e *send_all*. A rotina *send_to* é responsável por enviar uma mensagem para um determinado processo vizinho. Já a rotina *send_all*, é bastante semelhante à anterior, porém a mensagem será enviada para todos os vizinhos do processo, no caso de grafo não orientado e para todos os vizinhos de saída, no caso de grafo orientado.

Existem mais duas funções que auxiliam o usuário no geração do código para as ações. São elas *vizinho* e *calc_num_vizinho*. A primeira indica se um determinado processo é vizinho do processo que faz a chamada da função.

A outra função criada é aquela que calcula o número de vizinhos de um determinado tipo para um processo. Ela possui apenas um parâmetro, correspondendo

ao tipo de vizinho em questão, e seu retorno corresponde ao número de vizinhos referentes ao tipo indicado.

Algumas variáveis internas ao programa gerado também são visíveis ao usuário, para que o código possa ser escrito mais facilmente. A variável *id* determina o número do processo que está executando o algoritmo.

Outra variável também importante para a codificação é *source* que determina, no momento do recebimento da mensagem, qual o processo emissor (remetente) da mesma. Desta forma é possível determinar códigos diferentes dependendo do processo que enviou a mensagem.

A variável *recbuf* contém a última mensagem recebida pelo processo. Seu tipo e tamanho são os mesmos definidos pelo usuário no arquivo de configuração.

A variável *processor_name* determina o nome do processador onde o processo está executando.

3.2.2 Recursos do MPI

Alguns recursos oferecidos pelo MPI (Message Passing Interface) foram utilizados para implementar algumas características necessárias na implementação desse ambiente.

Conforme já foi visto, este ambiente de desenvolvimento permite que o usuário defina um tipo de mensagem diferente dos tipos de dados disponíveis na linguagem C (ex. char, int, etc.). Todas as funções de comunicação do MPI têm um argumento tipo de dado. O MPI possui uma generalização importante que resulta da possibilidade de tipos definidos pelo usuário, onde os tipos primitivos podem ocorrer. Eles são tipos onde o MPI somente toma conhecimento deles através de funções construtoras de tipos. Dentre estas funções, foi utilizada a mais geral delas, que é a função MPI_Struct.

Esta função recebe como parâmetros o número de campos do novo tipo, o tipo, tamanho e o nome do novo tipo. Desta forma, é criado um tipo derivado do MPI com os campos desejados, que pode ser utilizado como 'tipo' das mensagens que serão responsáveis pela comunicação entre os processos. Esse recurso faz com que o usuário não fique limitado aos tipos de dados padrão do C e a comunicação possa ser realizada com estruturas de dados complexas, tais como estruturas que contém combinações de tipos de dados primitivos.

Um outro recurso do MPI que foi utilizado na implementação deste ambiente foi a

opção de utilizar um buffer, dimensionado pelo usuário, para realizar a comunicação entre os processos.

No MPI, a operação de envio de mensagem (*send*) no modo bufferizado, pode ser iniciada independente de um recebimento (*receive*) correspondente ter sido inicializado. Esta função é a MPI_BSend, onde a letra 'B' indica que deve existir um buffer alocado para receber as mensagens enviadas através desta função.

O envio da mensagem pode se completar antes do *receive* ter sido inicializado. Então, se um send é executado e nenhum *receive* correspondente é inicializado, o padrão MPI deve guardar a mensagem em um buffer, de forma a permitir que a chamada seja completada. O MPI gerará um erro caso não haja nenhum tratamento de erro para o caso de não haver espaço suficiente no buffer para armazenar uma mensagem recebida.

O usuário pode controlar o tamanho do buffer de mensagens, e desta forma, realizar vários experimentos com seu algoritmo. O espaço disponível no buffer é definido pelo usuário na interface gráfica através da informação do número máximo de mensagens em trânsito. A partir desta informação, o tamanho do buffer é atribuído, através da função MPI_Buffer_attach do MPI.

Um dos recursos mais interessantes disponíveis no MPI é a possibilidade de manipular alguns erros que ocorram durante chamadas de MPI. Estes podem incluir erros que geram exceções ou *traps*, tais como erro de ponto flutuante ou violações de acesso. O conjunto de erros que são manipulados pelo MPI é dependente da implementação. Cada erro gera uma exceção MPI.

Na implementação deste ambiente, esse recurso foi utilizado especificamente para o tratamento do erro que ocorre quando uma mensagem é enviada e não há espaço no buffer de mensagens. Esse erro é considerado "fatal" pelo MPI, e frente à sua ocorrência, o programa é abortado.

Existem dois manipuladores de erro pré-definidos disponíveis no MPI:

MPI_ERRORS_ARE_FATAL - O programa aborta todos os processos em execução.
MPI_ERRORS_RETURN - O manipulador simplesmente retorna o código do erro ocorrido para o usuário, não abortando o programa.

A cada vez que a função MPI_Bsend, que é responsável pelo envio das mensagens entre os processos, é invocada, foi acrescentado o tratamento para o erro de buffer

cheio.

Esse tratamento é realizado através de duas funções do MPI que são responsáveis por resgatar e atribuir o manipulador de erros ao programa. Estas funções são *MPI_Errhandler_get* e *MPI_Errhandler_set*, respectivamente. Inicialmente, o manipulador de erros atual é armazenado em uma variável auxiliar. Logo em seguida, atribui-se o manipulador pré-definido do MPI, o *MPI_ERRORS_RETURN*, que possui apenas a função de retornar o código do erro para o programa. O código retornado é comparado com o código para o erro "Buffer Cheio", que no MPICH é igual a 513. O programa permanece dentro de uma estrutura 'while' que envia novamente a mensagem até que esse erro não ocorra mais. Assim que a mensagem é enviada, o manipulador de erros original é restituído, pois este é o único erro tratado pelo programa.

3.2.3 Geração do Programa

Utilizando os dados do algoritmo distribuído informados pelo usuário, um programa em C é gerado. Este programa realiza a execução do algoritmo distribuído em questão.

Para a geração deste programa, foi utilizado o padrão para comunicação entre processos MPI (Message Passing Interface). Além disso, foram definidas algumas estruturas onde as informações principais para a execução do algoritmo devem ser guardadas, e alguns padrões para nomes de variáveis, constantes, etc. Todas essas características serão descritas a seguir.

Inicialmente foi desenvolvido um "esqueleto", ou seja, um modelo onde foram definidas as partes necessárias para o funcionamento de um algoritmo distribuído utilizando os modelos apresentados em [2].

Foi definido então que este esqueleto deveria possuir as seguintes partes:

1 – Header (Includes C)

2 – Definição de Constantes

3 – Definição das Variáveis MPI e Gerais

4 – Definição das Variáveis do Usuário

5 – Definição dos Procedimentos

6 – Inicialização do MPI

7 – Inicialização das Variáveis do Usuário e Gerais

8 – Inicialização da Estrutura para Topologia

9 – Inicialização do vetor dos processos que iniciam espontaneamente

10 – Tratamento para Inicialização do Algoritmo

11 – Tratamento para Recebimento das Mensagens. (pares Entrada/Ação)

12 – Finalização do MPI

Abaixo se encontra uma descrição mais detalhada de cada uma destas porções do programa gerado.

A primeira parte é constituída das bibliotecas da linguagem C necessárias para a execução do programa. Primeiramente, são incluídas as bibliotecas para o Programa Geral. Vamos chamar de Programa Geral a porção do programa que é comum a todos os algoritmos, isto é, que independe do algoritmo distribuído escolhido pelo usuário. As bibliotecas do Programa Geral são "mpi.h", "stdio.h" e "string.h". Logo abaixo destas, são incluídas as bibliotecas informadas pelo usuário como Header.

Na parte dois são contempladas as constantes do programa. O usuário pode definir no campo identificado como Header as constantes que deseja para seu algoritmo. Logo após a inclusão das constantes do usuário, são acrescentadas as constantes para o Programa Geral. Duas delas podem ser bastante úteis ao usuário. A constante para o número de processos CTE_NUM_NOS, e CTE_NUM_MAX_MSG que indica o número máximo de mensagens em trânsito em um determinado canal ou aresta. Como se pode observar, todas as constantes pertencentes ao Programa Geral possuem a expressão 'CTE_' antes de seu nome.

A terceira parte contempla a definição das variáveis do programa. O padrão MPI exige que algumas variáveis sejam definidas para que o processo de comunicação possa ser realizado. É neste momento que estas são declaradas. Nesta parte também são declaradas as variáveis do Programa Geral, como a identificação (número) do processo corrente (*id*), identificação do processo emissor de uma mensagem (*source*), etc; assim como as estruturas criadas para o programa.

Foi definido que as variáveis do Programa Geral são iniciadas pela expressão 'var_' para que possam ser diferenciadas das variáveis do usuário, exceto as duas

citadas acima, pois estas tem um significado especial para o usuário. As estruturas do programa gerado possuem o texto 'type_' antes de seus nomes.

Na quarta porção, as variáveis do algoritmo distribuído, definidas pelo usuário, são acrescentadas. Esta porção pode conter também declarações de tipos, caso o usuário tenha definido.

A parte cinco possui a declaração dos procedimentos do programa geral. Dentre eles estão rotinas para execução das ações para os pares Entrada/Ação, testes para verificar se as condições dos pares são satisfeitas e tratamento para enviar mensagens para todos os vizinhos. Todos os procedimentos do Programa Geral, exceto as funções vizinho, calc_num_vizinhos que estão disponíveis ao usuário, possuem o texto 'proc_' antes de seu nome.

A parte seis contempla o trecho de código C correspondente à inicialização das funções do padrão MPI, assim como a inicialização do programa principal, com a função 'main()'.

Na sétima parte, as variáveis que por algum motivo não foram inicializadas no momento da declaração, recebem aqui seus valores iniciais.

Uma estrutura foi criada para armazenar os dados que constituem a topologia (grafo) que retrata a ligação entre cada processo participante do algoritmo que se deseja executar. É na parte oito, que esta estrutura é inicializada. Essa informação é guardada em dois vetores: 'var_indice' e 'var_arestas'. A variável 'var_indice' é um vetor de inteiros que guarda a soma dos graus (número de vizinhos) dos processos de 0 a N, onde N é o número total de processos. A variável 'var_arestas' é um vetor de estruturas que guarda todas as arestas de cada processo em ordem crescente do número de processos, isto é, primeiro arestas do processo 0, depois arestas do processo de 1, e assim por diante. Cada elemento do vetor é uma estrutura de dois campos. O primeiro indica o número do processo vizinho e o segundo indica o tipo do vizinho, que pode ser 'I' de entrada (Entrada), 'O' de saída (output) ou 'B' para ambos (entrada e saída).

Vejamos um exemplo desta estrutura:

No exemplo acima, o processo 0 possui três vizinhos, o processo 1 possui um vizinho, e os processos 2 e 3 possuem dois vizinhos cada um. Desta forma, a variável 'var_indice' recebe os seguintes valores:

$$\text{var\_indice}[4] = \{\ 3\ ,\ 4\ ,\ 6\ ,\ 8\ \}$$

Pode-se observar que o grau do processo p, exceto o processo zero, corresponde

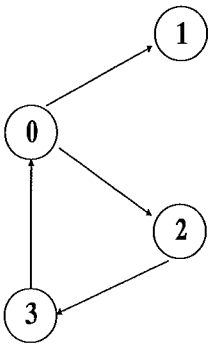

Figura 3.14: Exemplo de Topologia Geral

à diferença var_indice[p]–var_indice[p–1]. No caso do processo zero, o grau equivale ao conteúdo da variável var_indice[0]. Para esse exemplo, a variável 'var_arestas' recebe os seguintes valores:

var_arestas[8] = {['O', 1], ['O', 2], ['I', 3], ['I', 0], ['I', 0], ['O', 3], ['I', 2], ['O', 0]}

Neste caso, os três primeiros elementos correspondem às arestas do processo 0, o terceiro elemento corresponde ao processo 1, os elementos quatro e cinco são arestas do processo 2 e os elementos seis e sete são arestas do processo 3.

Esse procedimento de incluir a soma dos graus dos processos visa facilitar a identificação das arestas de cada processo, os quais constam na variável 'var_arestas'. Também pode-se observar que utilizando esta estrutura, obtém-se uma enorme economia de memória, transferindo a complexidade da decodificação para a programação.

Voltando à descrição das partes do programa gerado, a nona parte corresponde à inicialização do vetor que contém os processos que iniciam espontaneamente, no caso do algoritmo assíncrono, e os processos que executam no pulso zero, para o caso de algoritmo síncrono. O nome da variável deste vetor é var_N0 e sua dimensão é igual ao número de processos que iniciam espontaneamente. Essa informação é indicada pelo usuário na interface gráfica.

Na décima parte, o código para a inicialização do algoritmo é preparado. Para os algoritmos assíncronos, este corresponde ao par Entrada/Ação de número zero, onde a Entrada é igual a NIL. Já para os algoritmos síncronos, este código é referente

à ação para o instante s=0. Apenas os processos que iniciam espontaneamente executam este trecho de código.

Na parte onze são contempladas as ações do algoritmo distribuído propriamente dito. É aqui que ocorre o tratamento de cada mensagem recebida, isto é, a implementação de um código para os pares Entrada/Ação. As mensagens são recebidas e tratadas dentro de um *loop* que termina quando a condição de término, que foi informada pelo usuário, é satisfeita. O trecho de código para esta parte do programa segue o algoritmo abaixo:

```
Enquanto (não terminou)
{
   Aguarda recebimento mensagem
   Testa se condição para msg recebida é verdadeira
   Verifica se msgs na lista de pendências já estão satisfeitas
}
```

A ação de aguardar o recebimento de uma mensagem corresponde à chamada de uma função do padrão MPI. Nesta chamada, o programa fica aguardando até que um processo vizinho envie uma mensagem.

Ao receber a mensagem, o programa verifica se a condição referente àquele tipo de mensagem está sendo satisfeita. No caso de algoritmos síncronos, esta condição é sempre verdadeira. Se esta condição estiver satisfeita, a ação correspondente é executada, caso contrário, esta mensagem é incluída em um lista de mensagens pendentes, cujas condições ainda não foram satisfeitas. Isso só pode ocorrer para algoritmos assíncronos. Cada elemento desta lista contém um campo que identifica o tipo de mensagem (TAG) e outro campo que determina um apontador (ponteiro) para o próximo elemento da lista. Não há necessidade de guardar a condição nesta lista pois, a partir do tipo de mensagem, pode-se determinar facilmente a condição associada.

Ao final do tratamento de uma mensagem recebida, a lista de mensagens pendentes é varrida e todas as condições contempladas são testadas. À medida que estas vão sendo satisfeitas, são excluídas da lista.

A última parte do programa gerado corresponde à finalização do padrão MPI, assim como a desalocação da memória reservada para o buffer utilizado para arma-

zenar as mensagens relativas à comunicação entre os processos.

Algoritmos Síncronos

Uma característica importante desse ambiente é a forma com que o algoritmo síncrono foi implementado. Nesse caso, um sincronizador foi utilizado para converter o algoritmo informado para um algoritmo assíncrono. Dois tipos de sincronizadores foram utilizados: o sincronizador Alpha e o Beta.

A utilização do sincronizador é transparente ao usuário, de forma que os dados do algoritmo distribuído informados pelo usuário são os mesmos, independente do sincronizador escolhido pelo usuário.

O código responsável pela implementação do sincronizador é incluído durante a geração do programa final.

Para o sincronizador Beta, há a necessidade da criação de uma árvore geradora. No ambiente implementado, esta árvore é construída antes da execução do algoritmo, propriamente dita. O nó zero é tomado como raiz da árvore, e a partir daí, a árvore é construída através do algoritmo de propagação de informação com realimentação. Neste algoritmo, um nó, ao receber uma mensagem, envia a mesma para todos os seus vizinhos, exceto para o vizinho de quem recebeu a primeira mensagem. O nó que inicia espontaneamente envia mensagem para todos os seus vizinhos. No nosso caso este nó é o nó zero, que corresponde à raiz da árvore. Quando um nó recebe a mensagem de todos os seus vizinhos, ele envia uma mensagem para seu pai, informando que todos os seus vizinhos já receberam a mensagem. Quando o nó zero recebe a mensagem de retorno de todos os seus vizinhos, significa que a árvore já foi criada, e cada nó já sabe quem é seu pai e seus filhos. [2]

3.3 Desenvolvendo Algoritmos Distribuídos no Ambiente

O ambiente implementado nesta tese tem como objeto final um programa que realiza a execução do algoritmo distribuído informado pelo usuário.

Vamos mostrar a seguir como é simples a utilização deste ambiente para o desenvolvimento de algoritmos distribuídos. Na primeira seção são apresentados os passos necessários para a geração do programa referente a um algoritmo assíncrono. E, na seção seguinte, os passos referentes a geração do programa para um algoritmo

síncrono.

3.3.1 Criação de um Algoritmo Assíncrono

Um algoritmo bastante utilizado no estudo de algoritmos distribuídos é aquele que considera o problema de se determinar a distância mínima de um nó a todos os demais nós de uma rede, assim como o próximo nó na rota para chegar a cada um deles. Essa distância está sendo considerado como sendo o número de arestas de um nó a outro nó da rede.

Pode-se descrever o algoritmo citado acima no formato apresentado em [2], obtendo-se como resultado o algoritmo descrito a seguir:

Algoritmo A_Compute_Distances:

▷ **Variáveis:**

$dist_i^i = 0$;
$dist_i^k = n$ para todo $n_k \in N$ tal que $k \neq i$;
$first_i^k = nil$ para todo $n_k \in N$ tal que $k \neq i$;
$set_i = \{id_i\}$;
$level_i^j = -1$ para todo $n_j \in Neig_i$;
$state_i = 0$;
$initiated_i = false$;

▷ **Entrada:**

$msg_i = nil$
Ação Se $n_i \in N_0$:
$initiated_i = true$;
Envie set_i para todo $n_j \in Neig_i$;

▷ **Entrada:**

$msg_i = set_j$ tal que $origem_i(msg_i) = (n_i, n_j)$
Ação:
Se não $initiated_i$ então
Início
$initiated_i = true$;
Envie set_i para todo $n_k \in Neig_i$;
Fim
Se $state_i < n - 1$ então
Início
$level_i^j = level_i^j + 1$;
Para todo $id_k \in set_j$ faça
Se $dist_i^k > level_i^k + 1$ então
Início
$dist_i^k = level_i^j + 1$;
$first_i^k = n_j$
Fim

Se $state_i \leq level_i^j$ para todo $n_j \in Neig_i$ então
Início
$state_i = state_i + 1$;
$set_i = \{id_k | n_k \in N$ e $dist_i^k = state_i\}$;
Envie set_i para todo $n_k \in Neig_i$;
Fim
Fim

No ambiente implementado, os dados do algoritmo devem ser fornecidos descritos na linguagem de programação C. Dessa forma, os dados do algoritmo apresentado anteriormente podem ser informados como descrito abaixo:

▷ **Variáveis:**

```
int dist[N], first[N], set[N];
int level[N];
int state=0, initiated=0;
int i, k, indice;
int achou;
int cont_msg[N];
int final = 0;
```

▷ **Inicialização das Variáveis:**

```
for(i=0; i<N; i++)
{
    if(i==id)
        dist[i] = 0;
    else
        dist[i] = N;
    first[i] = -1;
    set[i] = -1;
    level[i]=-1;
    cont_msg[i]=0;
}
set[0] = id;
```

▷ **Entrada:**
NIL
Ação:
Condição para a Ação: 1 (true)

```
initiated=1;
send_all(MSG set);
```

▷ **Entrada:**
MSG
Ação:
Condição para a Ação: 1 (true)

```
cont_msg[source]++;
if(!(initiated))
{
    initiated=1;
    send_all(MSG set);
}
```

```
    for(i=0; i<N; i++)
    set[i] = recbuf[i];
if(state < N-1)
{
    level[source]++;
    for(i=0; i<N; i++)
    {
        if(set[i] != -1)
            if(dist[set[i]] > (level[source]+1))
            {
                dist[set[i]] = level[source]+1;
                first[set[i]] = source;
            }
    }
    achou=0;
    for(i=0; i<N; i++)
        if(vizinho(i,'B'))
            if(state > level[i])
                achou=1;
    if(achou==0)
    {
        state++;
        indice=0;
        for(k=0; k < N; k++)
            set[k]=-1;
        for(k=0; k < N; k++)
        {
            if(dist[k]==state)
            {
                set[indice]=k;
                indice++;
            }
        }
        send_all(MSG set);
    }
}
final=1;
for(i=0; i<N; i++)
{
    if(vizinho(i,'B'))
        if(cont_msg[i] != N)
            final=0;
}
```

Além das informações contempladas no modelo do livro, alguns dados adicionais são necessários para permitir a criação do programa. Dentre eles está o número de nós, a topologia e uma condição booleana que identifica, quando satisfeita, o fim da participação do nó na execução do algoritmo.

No caso deste algoritmo, esta condição booleana, que na interface gráfica chamamos de *Condição de Término*, está representada pela variável *final*. O final da

execução de um nó neste algoritmo se dá quando ele recebe N mensagens de cada um de seus vizinhos, onde N é o número de nós da rede. Isso é fácil de observar, pois cada nó envia N mensagens, uma quando inicia o processamento, ou seja, no primeiro para Entrada/Ação e N–1 mensagens no segundo par. Por isso foi incluída a variável *cont_msg[j]* que é incrementada toda vez que um nó recebe uma mensagem de um vizinho j. Esta variável é então verificada e, caso seja igual a N para todos os vizinhos, a variável *final* recebe o valor 1, e o algoritmo termina para o nó.

Os procedimentos "send_all" e "vizinho" foram utilizados na codificação do algoritmo. Estes são procedimentos implementados no ambiente de forma a facilitar a programação por parte do usuário e têm a função de enviar uma mensagem para todos os vizinhos e descobrir se um nó é vizinho, respectivamente.

Pode-se observar na codificação do algoritmo que aparece uma constante N que determina a condição de fim de alguns *loops*. Esta constante é declarada juntamente com a declaração das bibliotecas do C (includes) no campo destinado a esta informação na interface gráfica.

Um outro dado necessário é a identificação do tipo em C da mensagem que deverá transitar realizando a comunicação entre os nós. No caso desse algoritmo, essa mensagem é um vetor de no máximo N posições, contendo números que correspondem à identificação dos nós, representada pela variável *set*.

Os demais dados requeridos pelo ambiente têm a finalidade de permitir variações sobre o algoritmo que se deseja criar. Ou seja, pode-se variar a topologia, o número de nós, estipular o tamanho máximo do buffer de mensagens, definir em que processador cada nó estará executando.

3.3.2 Criação de um Algoritmo Síncrono

O algoritmo síncrono escolhido para ser demonstrado nesta seção foi o mesmo utilizado na versão assíncrona, que trata o problema de se determinar a distância mínima de um nó a todos os demais nós de uma rede, considerando o número de arestas.

Pode-se descrever este algoritmo no formato apresentado em [2], obtendo-se como resultado o algoritmo descrito a seguir:

Algoritmo S_Compute_Distances:

▷ **Variáveis:**

$dist_i^i = 0$;

$dist_i^k = n$ para todo $n_k \in N$ tal que $k \neq i$;

$first_i^k = nil$ para todo $n_k \in N$ tal que $k \neq i$;

$set_i = \{id_i\}$;

▷ **Entrada:**

$s = 0$, $MSG_i(0) = \emptyset$

Ação Se $n_i \in N_0$:

Envie set_i para todo $n_j \in Neig_i$;

▷ **Entrada:**

$0 < s \leq n - 1$

Ação:

$set_i = \emptyset$

Para todo $set_j \in MSG_i(s)$ faça

Para todo $id_k \in set_j$ faça

Se $dist_i^k > s$ então

Início

$dist_i^k = s$;

$first_i^k = n_j$

$set_i = set_i \bigcup \{id_k\}$;

Fim

Envie set_i para todo $n_k \in Neig_i$;

Pode-se observar a seguir como os dados do algoritmo síncrono apresentado acima podem ser descritos na linguagem de programação C.

```
▷ Variáveis:
  int dist[N], first[N], set[N];
  int i, prox_set;
▷ Inicialização das Variáveis:
  for(i=0; i<N; i++)
  {
     if(i==id)
        dist[i] = 0;
     else
        dist[i] = N;
     first[i] = -1;
     set[i] = -1;
  }
  set[0] = id;
▷ Entrada:
  s == 0
  Ação:
  send_all(MSG set);
▷ Entrada:
  s > 0 && s < N
  Ação:
  for(i=0; i<N; i++)
     set[i] = -1;
  prox_set = 0;
  msg_aux = message[s%2];
  while(msg_aux != NULL)
  {
```

```
        for(i=0; i<N; i++)
        {
            if(msg_aux->message[i]!=-1)
                if(dist[msg_aux->message[i]]>s)
                {
                    dist[msg_aux->message[i]]=s;
                    first[msg_aux->message[i]]=msg_aux->source;
                    set[prox_set]=msg_aux->message[i];
                    prox_set++;
                }
        }
        msg_aux = msg_aux->prox;
    }
    send_all(COMP_MSG set);
```

Algumas considerações devem ser feitas com relação ao modelo apresentado acima. Para armazenar as mensagens recebidas durante um pulso foi criada uma lista encadeada que é representada pelo variável *message*. Esta variável é um vetor de duas posições. Isso é decorrente da utilização de um sincronizador para converter o algoritmo síncrono em assíncrono.

Quando utilizamos um sincronizador, um nó pode receber mensagens dos seus vizinhos referentes ao pulso corrente (s) ou ao pulso posterior (s+1), pois todas as mensagens dos pulsos anteriores já foram recebidas, e nenhum vizinho pode enviar mensagens de pulsos maiores que s+1 antes que o nó tenha enviado suas mensagens do pulso s+1. [16] [13]

Para implementar a variável *message*, que representa o conjunto de mensagens recebidas em um determinado pulso, foi definido que esta variável seria um vetor de duas posições. Cada posição é um apontador para uma lista que contém as mensagens referentes a um pulso. A solução mais simples seria identificar o número máximo de pulsos e criar um apontador para cada pulso, mas esta solução despende muita memória.

O cálculo para se identificar índice no vetor da variável *message* referente a um determinado pulso P é bem simples. Este índice será o resto da divisão inteira de P por 2.

Cada elemento da variável message corresponde a uma estrutura composta de três campos, como se pode ver a seguir:

```
typedef struct type_message
{
    int source;
```

```
    int message[N];
    struct type_message *prox;
}type_message;
```

O primeiro campo é o *source* que indica qual o nó que enviou a mensagem. A seguir vem o campo message, que guarda o conteúdo da mensagem propriamente dito. O tipo C desse campo deve ser o mesmo da mensagem que transita entre os nós, que no caso do algoritmo de exemplo é um vetor de inteiros de N posições. E o terceiro campo é um ponteiro para identificar o próximo elemento da lista.

Voltando ao algoritmo apresentado, pode-se observar que as variáveis *msg_aux* e *message* não foram declaradas como variáveis do algoritmo. Estas variáveis fazem parte da implementação do sincronizador e podem ser de grande auxílio ao usuário para identificar as mensagens recebidas em um pulso. O código em C utilizado para "varrer" a lista de mensagens pode ser utilizado em todos os demais algoritmos síncronos. Este código é referente à seguinte parte:

```
msg_aux = message[s%2];
while(msg_aux != NULL)
{
    ...
    msg_aux = msg_aux->prox;
}
```

A variável *msg_aux* é uma variável auxiliar que aponta para um elemento da lista de mensagens. Inicialmente ele aponta para o primeiro elemento da lista e vai avançando para os próximos elementos através do comando

```
msg_aux = msg_aux->prox;
```

O valor de cada elemento da lista pode ser consultado através da variável "msg_aux->message".

Como no algoritmo assíncrono, alguns dados adicionais são necessários para que o programa final seja gerado. No caso deste algoritmo síncrono, a condição booleana que determina o fim da participação de um nó no algoritmo é quando ele atingir o

pulso N, e o campo *Condição de Término* na interface gráfica deve ser preenchido com "s == N". Isso pode ser facilmente explicado. Observando o funcionamento deste algoritmo, no pulso s $\geq$ 0, um nó envia para seus vizinhos um conjunto contendo a identificação de todos os nós que estão a exatamente "s" arestas de distância dele. Para s=0, esse conjunto contém apenas a própria identificação do nó. Para s > 0, esse conjunto contempla a identificação de todos os nós que recebidos durante o intervalo s–1, exceto aqueles que já haviam sido recebidos em algum intervalo anterior a s–1. Dessa forma, pode-se observar que não mais que N pulsos são necessários.

Como no algoritmo assíncrono, a constante N está declarada juntamente com a declaração das bibliotecas do C (includes) no campo *Header* da interface gráfica. O tipo de mensagem também pode ser identificado como um vetor de N posições, contendo números que correspondem à identificação dos nós.

Para visualizar o funcionamento desses algoritmos, o usuário pode acrescentar comandos de escrita nas Ações dos pares. Por exemplo, para observar o resultado dos vetores de distância e o primeiro nó na rota do algoritmo apresentado acima pode-se incluir o seguinte código no final do segundo par Entrada/Ação:

```
if(s==N-1)
{
   printf("Processo %d: \n dist = { ", id);
   for(i=0; i<N; i++)
      printf("%d ", dist[i]);
   printf("} \n");
   printf(" first = { ");
   for(i=0; i<N; i++)
      printf("%d ", first[i]);
   printf("} \n");
}
```

Utilizando o código acima, o usuário poderá imprimir na tela o conteúdo das variáveis *dist* e *first*. O usuário poderá também, optar por direcionar essa saída para um arquivo, ao invés de enviar para a tela.

Capítulo 4

Alguns Resultados Obtidos

Para apresentação de alguns resultados obtidos utilizando o ambiente implementado, foi escolhido o algoritmo distribuído para o cálculo das distâncias de um nó a todos os demais nós de uma rede. Foram analisadas as versões assíncrona e síncrona deste algoritmo e os parâmetros de variação foram topologia, número de nós e número de processadores. Para a experimentação com algoritmos síncronos, além destes parâmetros, os dois tipos de sincronizadores, Alfa e Beta, foram verificados.

As topologias verificadas nas experimentações foram anel, malha e grafo completo e, para cada uma delas o número de nós foi variado entre 4, 8 e 16 nós. Cada uma destas combinações foi executada utilizando-se 1, 2, 3 e 4 processadores.

Para distribuir os processos nos processadores, utilizou-se o seguinte critério:

- 1 processador - todos os processos alocados no processador único
- 2 processadores - os processos pares foram alocados no primeiro processador, e os demais processos, no segundo processador;
- 3 processadores - os processos 0, 3, 6, 9, 12, 15 (quando existentes) foram alocados ao primeiro processador; processos 1, 4, 7, 10, 13 (quando existentes) ao segundo processador e os demais, ao terceiro processador;
- 4 processadores - os processos 0, 4, 8, 12 (quando existentes) foram alocados ao primeiro processador; processos 1, 5, 9, 13 (quando existentes) foram alocados ao segundo processador; 2, 6, 10, 14 (quando existentes) foram alocados ao terceiro, e os demais alocados ao quarto processador.

Nas seções seguintes se encontram os resultados obtidos para o algoritmo verificado. Como resultados, considerou-se a média dos tempos de processamento de cada uma das combinações experimentadas, que foram executadas 5 vezes.

4.1 Algoritmo Assíncrono

Os gráficos apresentados nas figuras 4.1, 4.2 e 4.3 espelham os resultados obtidos na execução dos programas gerados pelo ambiente implementado, para o tipo de algoritmo assíncrono.

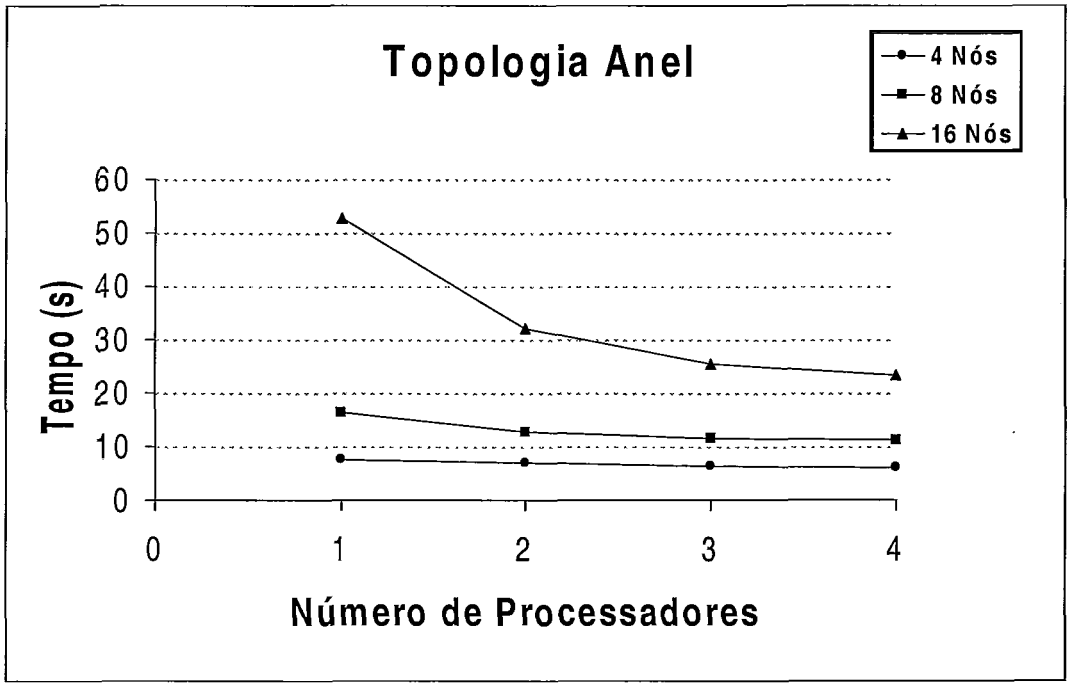

Figura 4.1: Algoritmo Assíncrono - Grafo em Anel

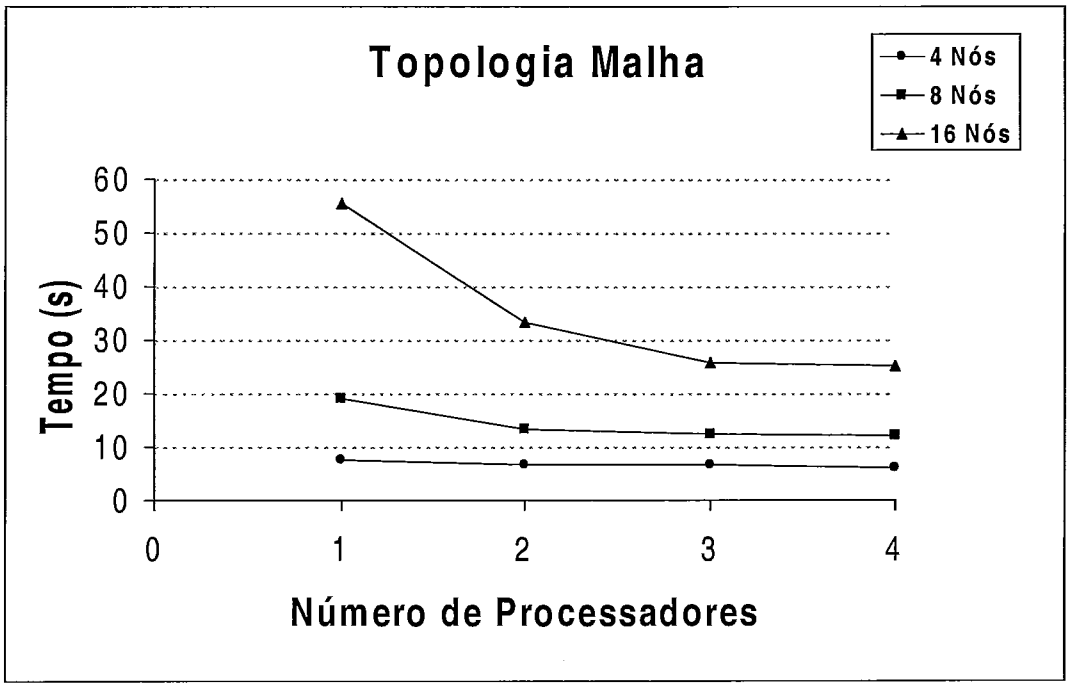

Figura 4.2: Algoritmo Assíncrono - Grafo em Malha

Antes de fazer qualquer observação a respeito dos resultados obtidos, deve-se considerar que diversos fatores influenciam no tempo de processamento do programa, dentre eles estão:

- o tráfego de processos executando no processador utilizado;
- características físicas do processador, como a quantidade de memória e velocidade;
- forma como os processos foram alocados aos processadores

Estes fatores são responsáveis por gerar alguns resultados fora do esperado.

Observando os gráficos apresentados, pode-se verificar que, ao aumentar o número de processadores, o tempo de processamento foi reduzido. Isso é justificado pelo conceito de paralelismo, pois o objetivo de se executar um programa em vários processadores é reduzir o tempo de processamento do mesmo.

Pode-se observar, também, que, em geral, a diferença entre os tempos de $x+1$ e x processadores, diminui conforme se aumenta o número de processadores. Isso pode ser facilmente explicado. Para um grafo de 16 nós, ao se utilizar dois processadores, o número de processos por máquina foi reduzido em 8 (16 para 8). Quando

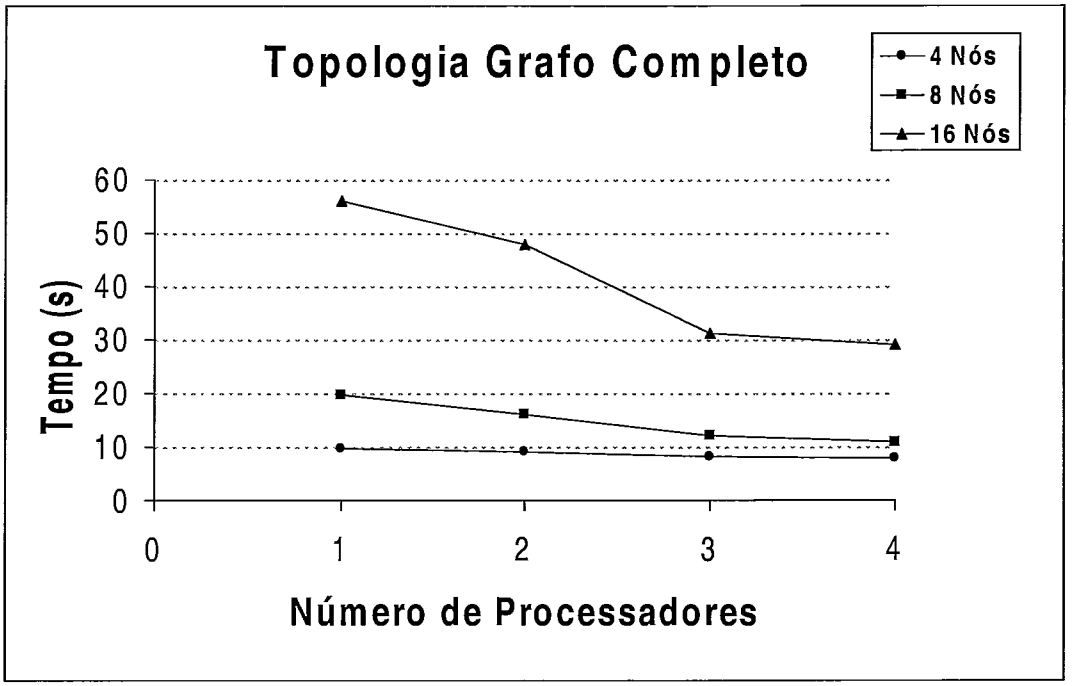

Figura 4.3: Algoritmo Assíncrono - Grafo Completo

utilizamos três processadores, este número reduz de 3 (8 para 5) visto que dois processadores estarão com 5 nós e um processador estará com 6 nós. No caso de quatro processadores, esta redução é ainda menor, ou seja, 4 processos executando em cada máquina, obtendo-se uma redução de 1 (5 para 4).

Desta forma, pode-se observar também que a queda no tempo de processamento aumenta conforme o número de processadores se torna maior.

4.1.1 Speed Up

O *speed-up* é o tempo de processamento de um algoritmo executado numa máquina seqüencial, dividido pelo tempo de processamento deste mesmo algoritmo numa máquina paralela. Para as três topologias verificadas, foi calculado o speed-up, como se pode observar nas figuras 4.4, 4.5 e 4.6.

Observando os gráficos apresentados, pode-se notar que para a topologia Grafo Completo, o speed-up obtido ao se utilizar 16 processos apresentou um valor maior

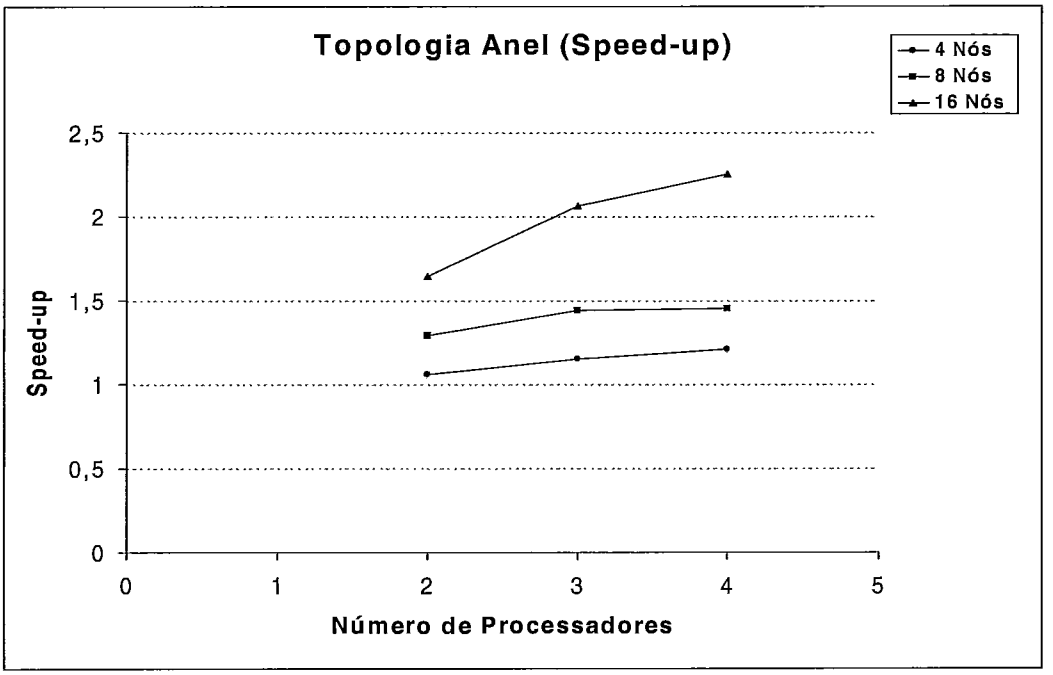

Figura 4.4: Algoritmo Assíncrono - Speed Up para Grafo em Anel

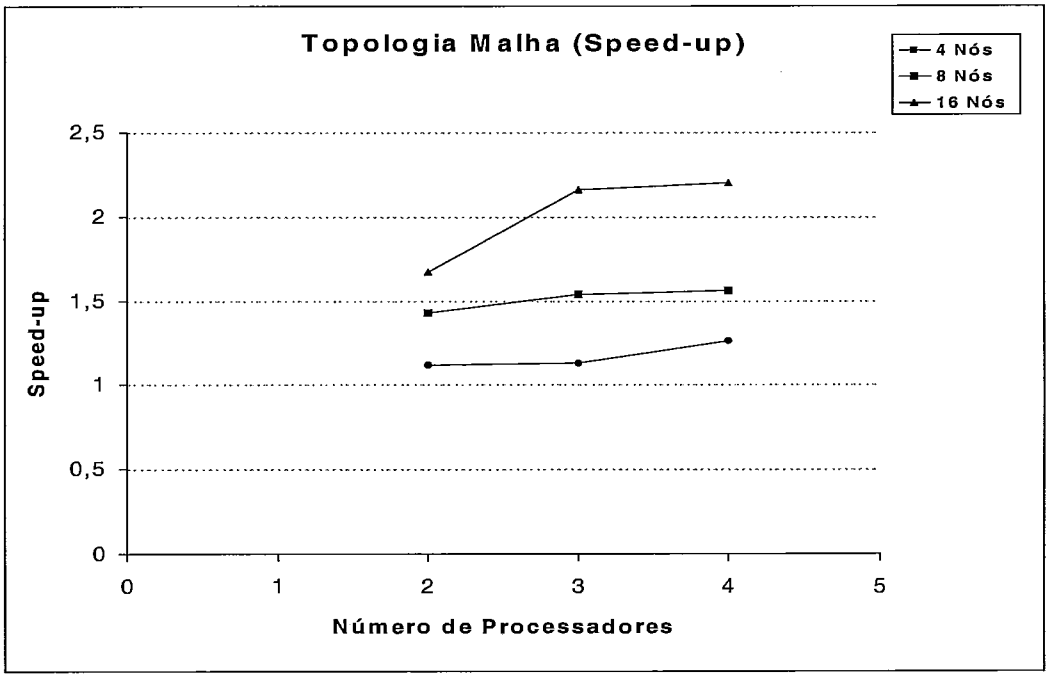

Figura 4.5: Algoritmo Assíncrono - Speed Up para Grafo em Malha

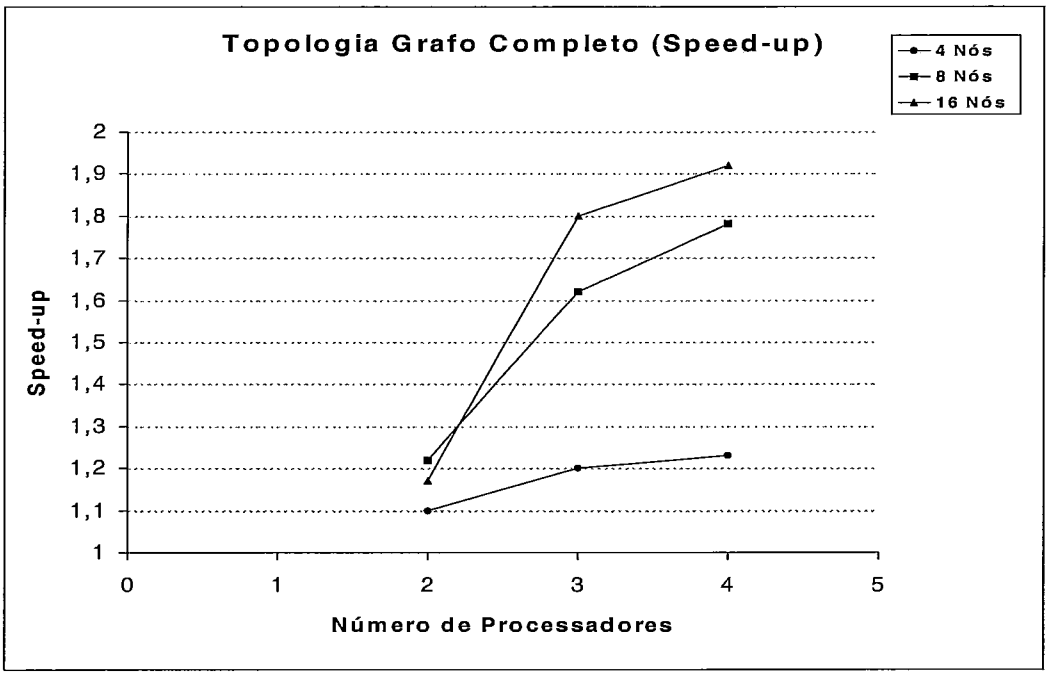

Figura 4.6: Algoritmo Assíncrono - Speed Up para Grafo Completo

comparando com os valores obtidos para 8 processos, exceto para 2 processadores. Normalmente, isso não deveria acontecer, mas, como já foi dito no início deste capítulo, diversos fatores do sistema podem ter influenciado nestes resultados.

4.2 Algoritmo Síncrono

Os gráficos ilustrando os resultados obtidos durante a execução do algoritmo síncrono, são apresentados nas figuras 4.7, 4.8 e 4.9.

As observações consideradas na seção anterior, para o algoritmo assíncrono, também se reflete nos resultados obtidos nesta seção, visto que, ao gerar um programa para o algoritmo síncrono, o mesmo é transformado em um algoritmo assíncrono correspondente.

O fato novo que aparece neste tipo de algoritmo diz respeito aos dois tipos de sincronizadores utilizados: Alfa e Beta. Ao se comparar estes dois sincronizadores, pode-se observar alguns fatores importantes: O sincronizador Alfa apresenta mais

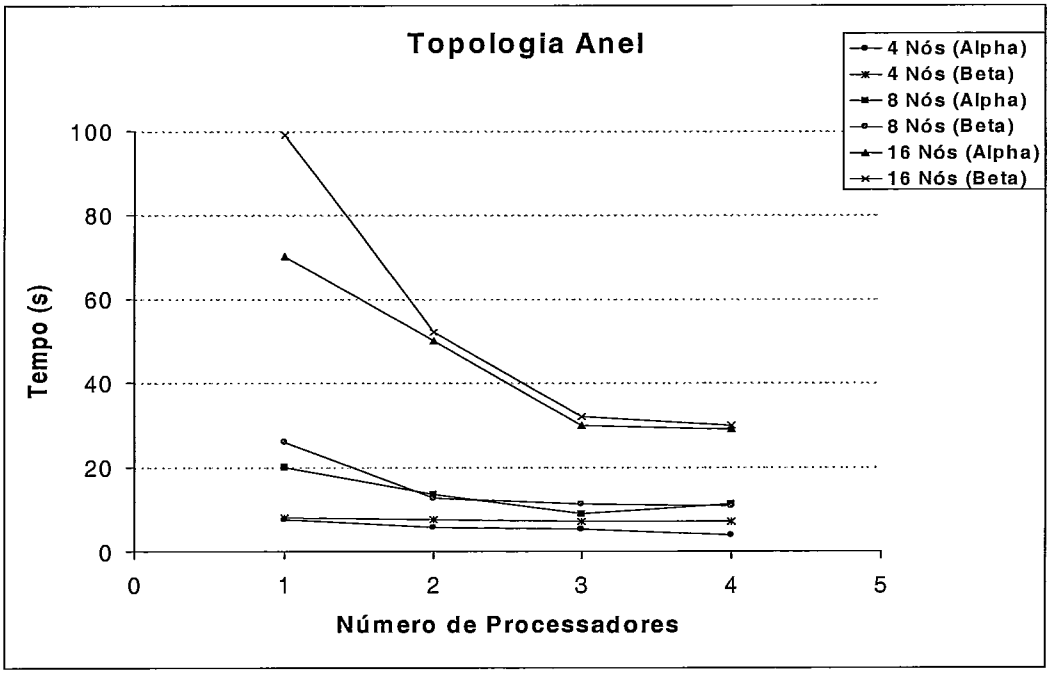

Figura 4.7: Algoritmo Síncrono - Grafo em Anel

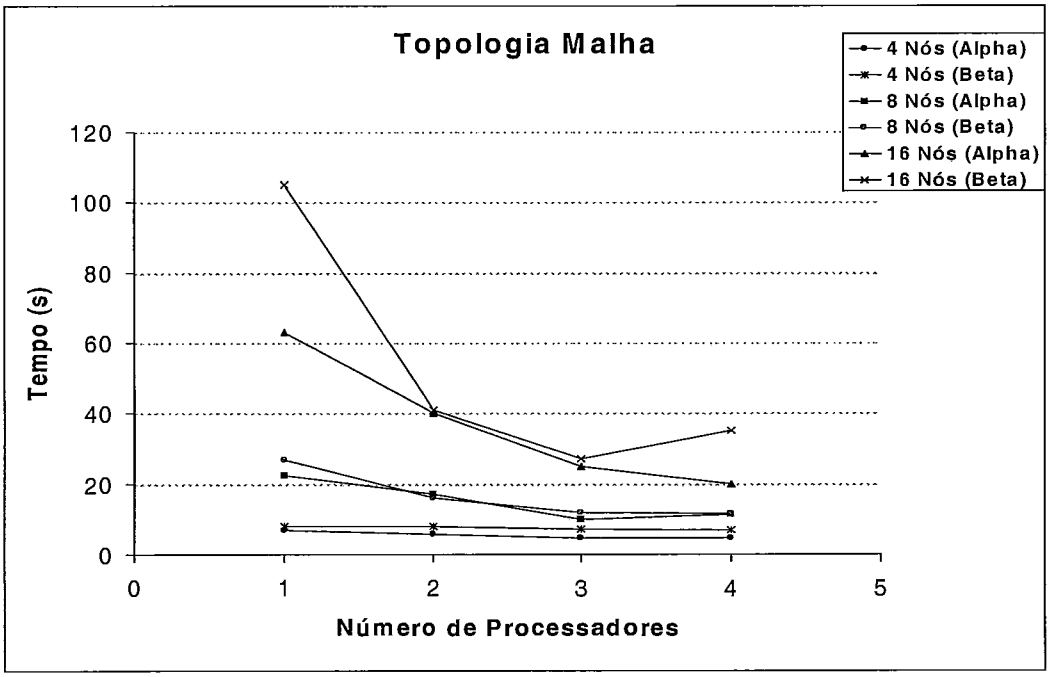

Figura 4.8: Algoritmo Síncrono - Grafo em Malha

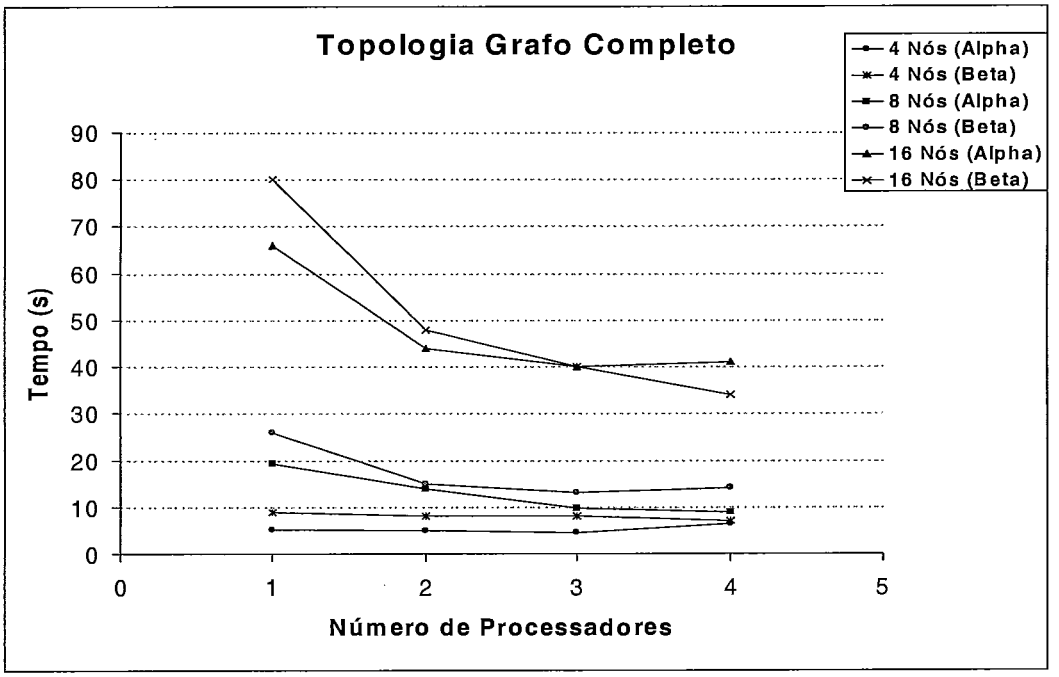

Figura 4.9: Algoritmo Síncrono - Grafo Completo

comunicação e menos processamento, enquanto que o sincronizador Beta possui menos comunicação, porém o tempo de processamento é bem maior, considerando que este último exige a construção de uma árvore geradora. Levando em conta que o tempo de comunicação é muito pequeno em relação ao tempo de processamento, é de se esperar que a execução do programa gerado utilizando o sincronizador Alfa seja mais rápida.

4.2.1 Sincronizador Beta

Para o sincronizador Beta, além dos tempos de comunicação e processamento, e a forma de alocação dos processos ao processadores, existe mais um fator que influencia nos resultados obtidos. A forma como a árvore geradora é construída pode tornar a execução do algoritmo mais lenta ou mais demorada. Dependendo da profundidade da árvore, o sincronizador levará mais tempo para propagar as informações por toda a árvore, resultando num tempo maior para finalizar o algoritmo.

No ambiente implementado, a árvore geradora é construída antes da execução do algoritmo, propriamente dita. O nó zero é tomado como raiz da árvore, e a partir daí, a árvore é construída através do algoritmo de propagação de informação com

realimentação. Neste algoritmo, um nó, ao receber uma mensagem, envia a mesma para todos os seus vizinhos, exceto para o vizinho de quem recebeu a primeira mensagem. O nó que inicia espontaneamente envia mensagem para todos os seus vizinhos. No nosso caso este nó é o nó zero, que corresponde à raiz da árvore. Quando um nó recebe a mensagem de todos os seus vizinhos, ele envia uma mensagem para seu pai, informando que todos os seus vizinhos já receberam a mensagem. Quando o nó zero recebe a mensagem de retorno de todos os seus vizinhos, significa que a árvore já foi criada, e cada nó já sabe quem é seu pai e seus filhos. [2]

Deve-se considerar que, cada vez que o algoritmo é executado, uma árvore diferente pode ser gerada, influenciando no tempo de processamento do algoritmo. Os gráficos nas figuras 4.10, 4.11 e 4.12 apresentam uma comparação entre o tempo para criação da árvore geradora e o tempo de processamento do algoritmo.

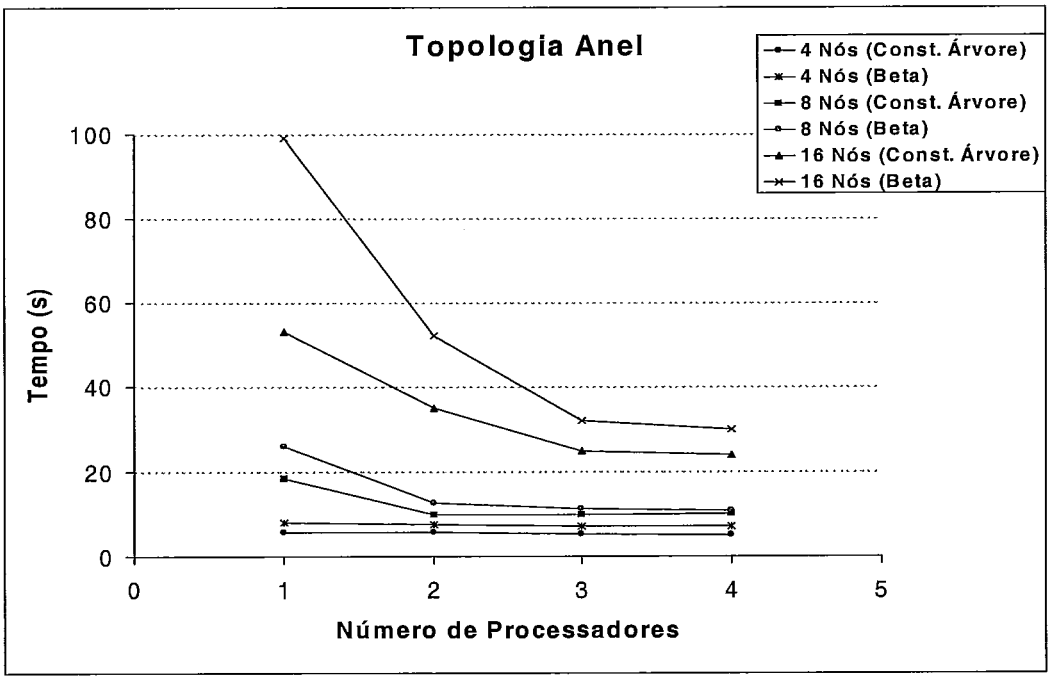

Figura 4.10: Árvore Geradora X Algoritmo Síncrono - Grafo em Anel

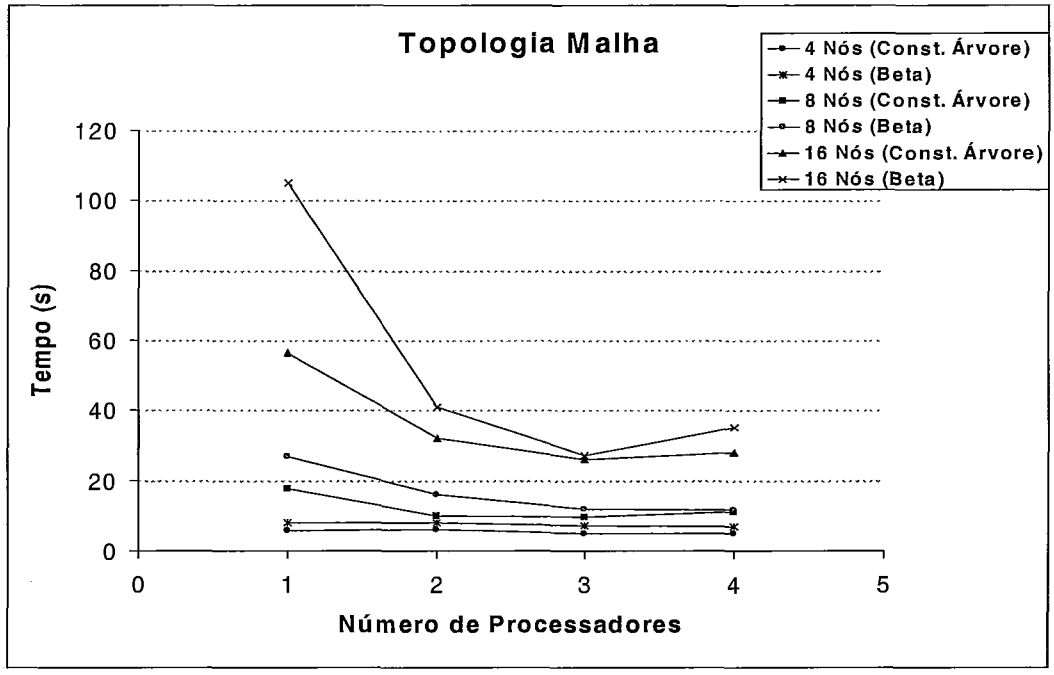

Figura 4.11: Árvore Geradora X Algoritmo Síncrono - Grafo em Malha

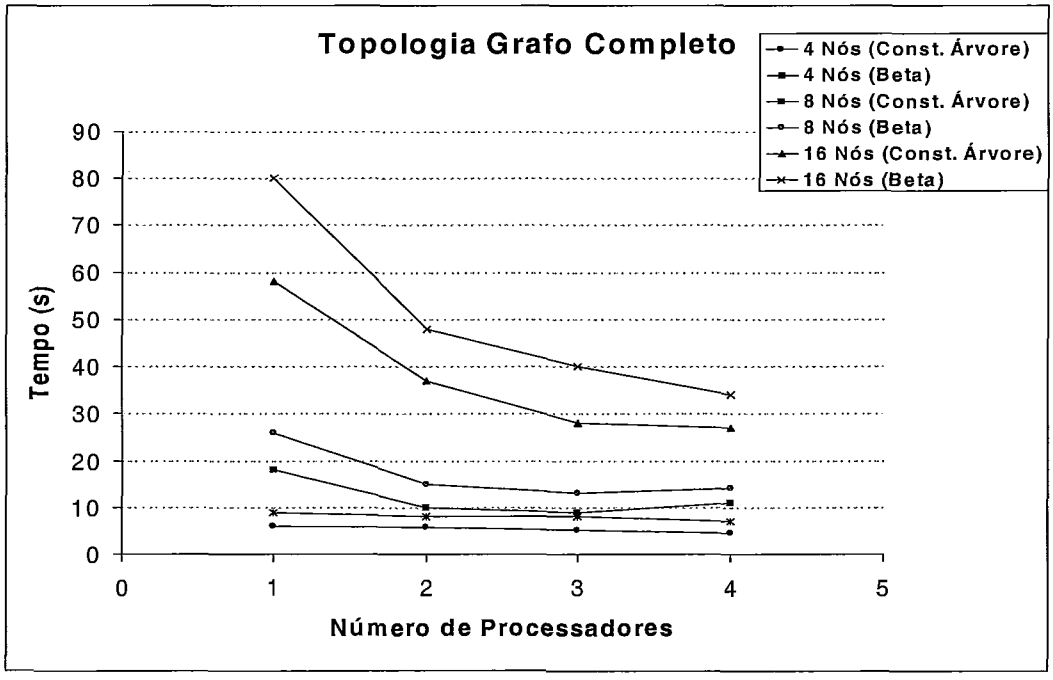

Figura 4.12: Árvore Geradora X Algoritmo Síncrono - Grafo Completo

Considerando as topologias analisadas espera-se que o tempo para a geração da árvore na topologia grafo completo seja menor que os demais tempos. Isso porque, como nesta topologia todos os nós se comunicam diretamente, a árvore mais provável de ser construída é aquela com profundidade 2, onde a raiz é o nó zero e todos os demais nós são filhos da raiz.

Porém, deve-se considerar que para grafo completo, o número de mensagens é bem maior, pois todos os nós enviam mensagens para todos os nós do grafo.

Observando os gráficos apresentados, pode-se notar que o tempo para a construção da árvore representa uma porção bastante considerável do tempo total do processamento do algoritmo.

4.2.2 Speed Up

Também para o algoritmo síncrono, foi calculado o *speed-up* para as três topologias verificadas, como se pode observar nas figuras 4.13, 4.14 e 4.15.

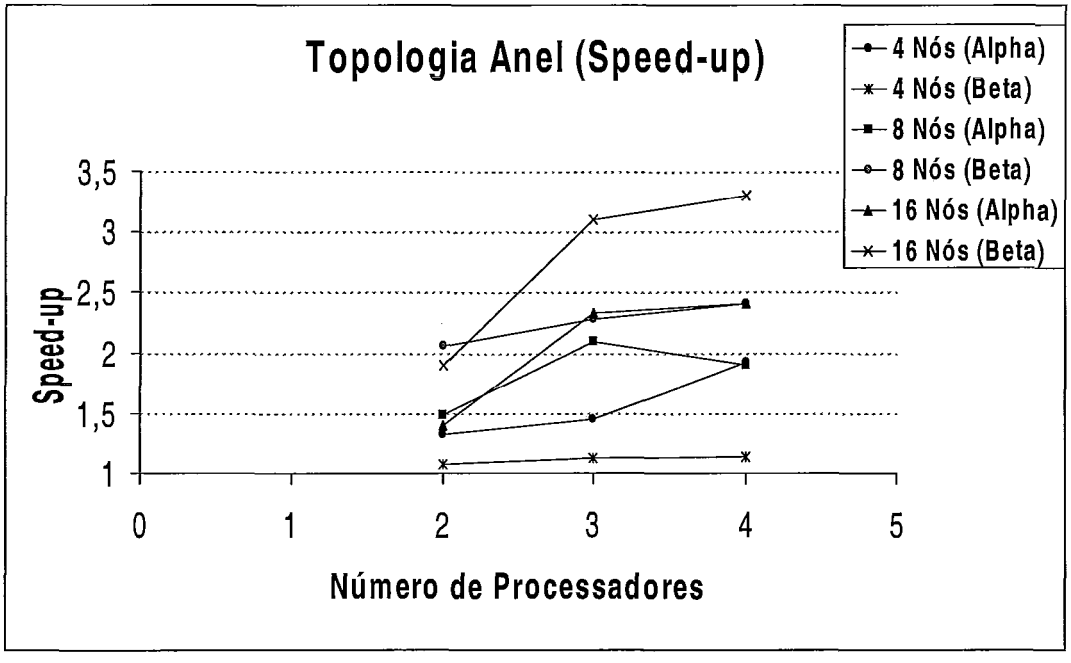

Figura 4.13: Algoritmo Síncrono - Speed Up para Grafo em Anel

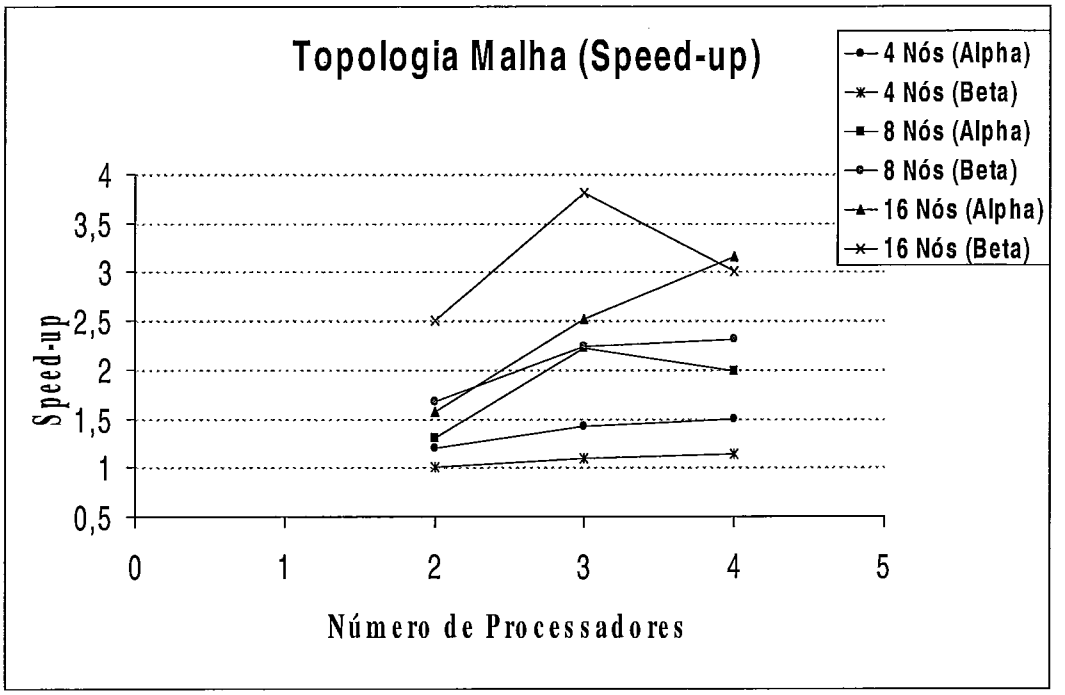

Figura 4.14: Algoritmo Síncrono - Speed Up para Grafo em Malha

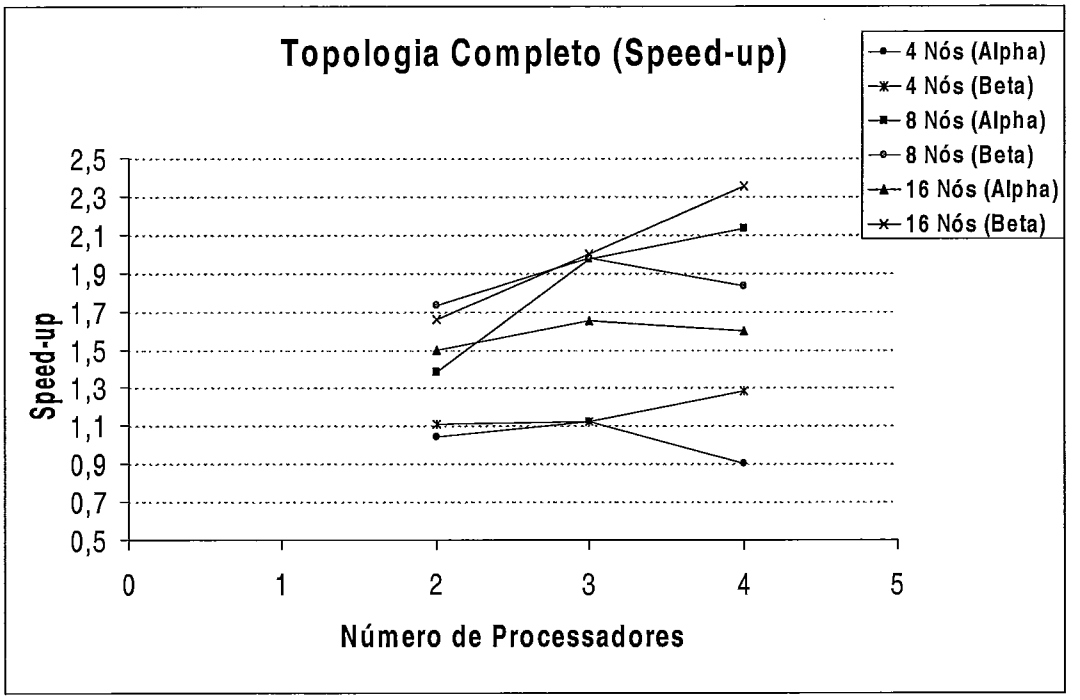

Figura 4.15: Algoritmo Síncrono - Speed Up para Grafo Completo

Capítulo 5

Assuntos Relacionados

Contemplando o assunto que é objeto desta tese, foram estudados alguns ambientes para desenvolvimento de algoritmos já implementados cuja finalidade se assemelha à do nosso ambiente.

5.1 Ambiente de Desenvolvimento e Avaliação de Algoritmos de Exclusão Mútua para Sistemas Distribuídos

Este primeiro sistema apresenta um ambiente de desenvolvimento, avaliação e comparação por simulação de algoritmos de exclusão mútua para sistemas distribuídos. Ele foi desenvolvido sobre a plataforma Unix/XWindow System. Entretanto, a maior parte do código é independente de plataforma.[3]

O núcleo do ambiente funciona como um emulador de um sistema distribuído. Simulações são realizadas submetendo-se o algoritmo implementado pelo usuário às condições desse sistema. O sistema apresenta as seguintes características:

- há um único recurso compartilhado
- os nodos se comunicam via troca de mensagens
- cada nodo pode se comunicar com todos os outros nodos do sistema
- o tempo de transmissão de mensagens pode sofrer atrasos aleatórios, mas a transmissão é livre de erros
- mensagens de um nodo i para um nodo j são entregues na ordem de envio;

- cada nodo do sistema assegura a serialização de sua demanda interna, isto é, no máximo uma demanda está por vez e o recurso é liberado antes que o nodo passe a processar a próxima demanda
- o algoritmo em questão é executado igualmente por todos os nodos, sem qualquer falha
- as rotinas do algoritmo são executadas de maneira local e indivisível

Inicialmente, o usuário deve codificar o algoritmo a ser avaliado num arquivo em linguagem C++ e ligá-lo com a biblioteca de primitivas disponíveis SiMEA, gerando um arquivo executável. Na segunda fase, o usuário de criar um arquivo que contenha a parametrização necessária à configuração do sistema distribuído a ser simulado. Concluídas essas fases, o programa gerado pode ser executado, acessando o arquivo de configuração e gerando como resultado arquivos de dados que servirão para análise e amostragem do comportamento do algoritmo.

Dois serviços complementares são oferecidos pela biblioteca SiMEA para utilização pelos nós do sistema:

- *Send*: Primitiva para troca de mensagens entre nodos
- *Eat*: Função que provoca a entrada de um nodo em sua região crítica

Acima foram vistas as funções que efetivamente participam do processo de especificação de um algoritmo. Para a execução das simulações estão disponíveis mais duas funções, são elas:

- *Config*: Processa o arquivo de configuração de simulação
- *Simulator*: Responsável por todo o processo de simulação

O arquivo de configuração de simulação (.simrc) deve ser fornecido pelo usuário antes da execução da simulação. Tal arquivo informa ao Núcleo os parâmetros sobre os quais a simulação será executada. Os seguintes parâmetros devem ser definidos:

- número de nodos do sistema
- tempo médio de latência de cada nodo
- tempo médio de utilização do recurso de cada nodo
- tempo médio de transmissão de mensagens entre um nodo e os demais nodos

do sistema

- desvio padrão associado a cada tempo acima
- distribuições estatísticas
- tempo de simulação

O algoritmo especificado pelo usuário através do Núcleo pode eventualmente conter algum erro lógico. A detecção de erros leva ao término da execução , invalidando logicamente os resultados que seriam gerados. Durante a execução, o Núcleo detecta a ocorrência de dois tipos de erros:

- Violação de Exclusão Mútua: Ocorre quando mais de um nodo utiliza, ao mesmo tempo, o recurso compartilhado;
- Ocorrência de *Deadlock*: É determinada pela formação de ciclos, isto é, um grupo de nodos não consegue acessar o recurso compartilhado.

5.2 Dome: Programação Paralela em um Ambiente de Computação Distribuída

O segundo sistema é o Dome (Distributed Object Migration Environment) que possui as características de balanceamento de carga e tolerância a falhas. [1]

Dome provê controle de processos, distribuição de dados, comunicação e sincronização para programas Dome executando em um ambiente de computação paralelo e heterogêneo. O programador paralelo escreve um programa em C++ usando objetos Dome que são automaticamente particionados e distribuídos sobre a rede de computadores. Dome incorpora o balanceamento de carga que ajusta, automaticamente, o mapeamento de objetos nas máquinas em tempo de execução, mostrando um ganho de desempenho significante em relação aos programas de troca de mensagens padrão executando em um sistema sem balanceamento. Dome também oferece *checkpointing* do estado do programa de uma forma independente da arquitetura, permitindo que programas Dome sejam verificados em uma arquitetura e reiniciados em outra.

Esse ambiente foi implementado como uma biblioteca de classes C++ que usa PVM para comunicação e controle de processos. A biblioteca Dome usa *operator*

overloading para permitir ao programador da aplicação simples manipulação dos objetos Dome e para esconder os detalhes de paralelismo.

Quando um programa utilizando a biblioteca Dome é executado, Dome primeiro cria os processos que constituem o programa distribuído usando um modelo *Single Program Multiple Data* (SPMD). No modelo SPMD, o programa do usuário é replicado na máquina virtual, e cada cópia do programa, executando em paralelo, executa sua computação sobre um subconjunto de dados em cada objeto Dome.

Dome oferece algumas possibilidades diferentes para o método de particionamento de dados. A diretiva *Whole* indica que todos os elementos de um dado objeto são replicados em todos os processos distribuídos. A distribuição *Block* indica que os elementos de dados do objeto Dome são equilibradamente divididos entre os processos. E finalmente o *Dynamic* indica que os elementos são inicialmente distribuídos equilibradamente, mas o dado é reparticionados entre os processos periodicamente através do balanceamento de carga realizado em dados intervalos. O usuário pode indicar o método de particionamento para um dado objeto Dome quando esse objeto é declarado.

5.3 Exploração Visual Interativa de Computações Distribuídas

O objetivo desse sistema é prover aos programadores não familiarizados com uma computação distribuída específica, desenvolver um entendimento razoável do funcionamento de um programa, sem requerer que eles examinem os detalhes de seu código. [7]

Seguindo esse objetivo, foi proposta a visualização baseada em *query* (query-based visualization), um método de exploração para entender computações distribuídas. As características básicas do método são o uso de *queries* como um dispositivo para pesquisar o espaço de estados, técnicas de apresentação visual adaptadas à animação do programa, e a possibilidade de navegação através do espaço de estado utilizando interações visuais. Todas as visões (*views*) correspondem aos retratos (*snapshots*) globais consistentes da computação.

A estratégia utilizada trata a computação distribuída como um banco de dados contendo o estado de processos individuais executando através de uma rede. O estado de cada processo se altera devido à computações locais, como o resultado de

atividades de passagem de mensagem, e como conseqüência da criação e terminação de processos. As *queries* oferecem o mecanismo pelo qual a exploração é conduzida. A navegação através do espaço de estados da computação acarreta o refinamento gradual das *queries* propostas pelo programador.

Com a finalidade de evitar o tipo de informação enganosa que pode resultar da presença de atrasos de comunicação, todas as *queries* são avaliadas logicamente com respeito aos estados globais consistentes do sistema que está executando. *Queries* típicas são avaliadas depois de cada alteração no sistema de estados. Algoritmos de *Snapshots* eficientes são fundamentais para o sucesso desse método.

A informação é apresentada com animações on-line tridimensionais do estado de sistema envolvido. A implantação inicial emprega a biblioteca PVM e conta com suposições seguras sobre a forma como a computação é estruturada, a fim de realizar o processamento da *query* e a coleção de *snapshot* eficientes.

Capítulo 6

Conclusões

O propósito desta tese foi a implementação de uma ferramenta para desenvolvimento de algoritmos distribuídos. Esta ferramenta vem suprir uma atual necessidade, para fins educacionais e de pesquisa, de uma ferramenta que permita explorar as diversas possibilidades de algoritmos distribuídos síncronos e assíncronos.

O ambiente implementado permite a experimentação de algoritmos, variando diversos dados, como número de nós, topologia, número de processadores onde os nós estarão alocados, tamanho do *buffer* de mensagens, entre outros. Com a variação de cada um destes fatores, é possível analisar o funcionamento do algoritmo distribuído desejado e fazer comparações entre cada resultado obtido.

Nesta tese foi apresentado um exemplo de como desenvolver um algoritmo distribuído utilizando o ambiente implementado e, pôde-se observar que isso representa uma tarefa bastante fácil. O desenvolvimento de algoritmos distribuídos utilizando este ambiente não demanda nenhum conhecimento de uma linguagem paralela. Apenas é necessário conhecer o formato para apresentação de algoritmos distribuídos, apresentado em [2] e um conhecimento razoável da linguagem C.

O fato de ter sido implementado para a Web, permite que o ambiente possa ser facilmente acessado de qualquer plataforma, tornando-o uma ferramenta bastante acessível.

Nesta tese foram apresentados resultados referentes ao algoritmo para cálculo das distâncias de um nó a todos os demais nós de uma rede. Foram analisadas as versões assíncrona e síncrona deste algoritmo e os parâmetros de variação foram topologia, número de nós e número de processadores.

Os valores testados para o parâmetro topologia foram anel, malha e grafo completo e o número de nós variou entre 4, 8 e 16 nós. Cada uma das combinações de

topologia e número de nós foi experimentada utilizando de 1 a 4 processadores. Para a versão síncrona, os dois tipos de sincronizadores, Alfa e Beta, foram verificados, além dos parâmetros avaliados para algoritmos assíncronos.

Analisando os resultados obtidos, viu-se que, na maioria dos casos, atenderam às expectativas, sendo que apenas alguns deles não corresponderam ao esperado por causa de algumas características do sistema operacional.

Para o caso de algoritmos síncronos utilizando o sincronizador Beta, também foram apresentados resultados da comparação entre o tempo gasto na construção da árvore geradora e o tempo total para execução do algoritmo síncrono gerado.

Para algoritmos síncronos e assíncronos foi calculado o *speed-up* relativo ao número de processadores utilizados.

Além do algoritmo escolhido para a geração dos resultados, também foram verificados outros algoritmos, como: propagação de informação, propagação de informação com realimentação, cálculo de uma função F em um anel, entre outros.

Finalmente, podemos verificar que o ambiente implementado atende à proposta desta tese, apresentando-se como ponto de partida para novas implementações. Uma das melhorias que poderia ser desenvolvida é a geração do programa para as plataformas IBM e SP2, visto que este ambiente foi implementado num escopo reduzido, gerando programas apenas para a plataforma Sun.

Uma outra sugestão de extensão deste trabalho seria a implementação de uma condição de terminação global para os algoritmos distribuídos que não possuem muita regularidade, pois neste caso, uma condição de término local não é suficiente.

Apêndice A

Pré-Requisitos para Execução

A.1 MPICH e Dispositivo ch_p4

Para a execução do programa gerado é necessário que o MPICH esteja instalado na rede. O MPICH é uma implementação do padrão MPI disponível gratuitamente, que executa em uma grande variedade de sistemas. Os arquivos para instalação estão disponíveis na Web, na URL *http://www.mcs.anl.gov/mpi* . Nesse mesmo endereço encontram-se também os manuais para instalação e de usuário. [10] [11] [12]

O programa gerado como resultado deste ambiente utiliza algumas rotinas do padrão MPI específicas para a implementação MPICH, como por exemplo, o código de erro que identifica que o *buffer* de mensagens está cheio, e o procedimento para tratamento de erros.

O ch_p4 é um dispositivo que já vem incluído no MPICH, que pemite a execução do *secure server*. Como cada estação requer que o usuário execute um novo *login* na mesma, e este processo despende muito tempo de consumo, o MPICH provê um programa que pode ser utilizado para acelerar esse processo, que é o *secure server*. [11]

A.2 Arquivo .cshrc

Para utilizar-se o *secure server* para o ch_p4, citado na seção anterior, é necessário adicionar as seguinte definições ao ambiente:

```
setenv MPI_USEP4SSPORT yes
setenv MPI_P4SSPORT 1234
```

O valor de MPI_P4SSPORT deve ser a porta na qual o *secure server* é inicializado. Quando estas variáveis de ambiente estão inicializadas, o mpirun, que é responsável por iniciar o programa distribuído, tenta utilizar o *secure server* para iniciar os programas que utilizam o dispositivo ch_p4.

Para que estas variáveis sejam sempre inicializadas, pode-se incluí-las no arquivo ".cshrc" que se localiza no diretório raiz da conta do usuário.

A.3 Arquivo .rhosts

A execução do algoritmo distribuído em máquinas diferentes exige que durante a inicialização do programa na máquina remota seja executado um *login* na mesma. Para que esse *login* remoto ocorra é necessário que o nome dessas máquinas remotas apareçam no arquivo ".rhosts". Vejamos um exemplo: se o usuário algdist deseja executar um algoritmo distribuído nas máquinas miami, porto e lisboa, esse arquivo deveria conter, pelo menos, essas três linhas, como segue:

```
miami algdist
porto algdist
lisboa algdist
```

A.4 Gerenciamento do Ambiente

A.4.1 Inicialização

O servidor Web responsável pela execução do ambiente implementado é um servidor *Apache*, que está instalado na conta do usuário algdist na máquina *miami*. Para que o ambiente funcione normalmente, o processo responsável pela inicialização deste servidor deve "estar no ar".

O processo que inicializa o servidor se chama start_server e se encontra na conta do usuário algdist, no diretório '~/.apache'. Este programa deve ser executado na máquina miami.

A.4.2 Arquivos Gerados

Os arquivos criados durante a geração dos programas no ambiente implementado são armazenados em um diretório temporário da máquina miami, e são apagados todos

os domingos através de um comando no programa *crontab* do sistema operacional Unix.

Apêndice B

Manual de Utilização do Ambiente

B.1 Iniciando...

O ambiente implementado provê ao usuário a possibilidade de desenvolver algoritmos distribuídos de uma forma simples, bastando para isso um conhecimento, não muito profundo, da linguagem de programação C.

Este ambiente está disponível na Web, na página do programa da Coppe Sistemas, através da URL **http://www.cos.ufrj.br/~algdist**.

Como já foi visto no Capítulo 3, o ambiente gerado apresenta ao usuário basicamente cinco telas diferentes. Na primeira tela, o usuário pode escolher se deseja criar um novo algoritmo distribuído ou alterar os dados de um algoritmo já criado.

A segunda tela permite que o usuário informe os dados gerais referentes ao algoritmo desejado, como topologia, número de nós, entre outros, enquanto que na terceira tela, o usuário deverá informar os dados relativos ao código do algoritmo. A partir destes dados fornecidos, o programa executável correspondente ao algoritmo é gerado, e o resultado deste processo é apresentado ao usuário numa terceira tela, onde ele poderá observar se o resultado foi com sucesso ou ocorreu algum erro na geração.

Quando o usuário está alterando um algoritmo que já foi gerado, a segunda e a terceira tela são unidas em uma única tela, onde são apresentados os dados do algoritmo informado para alteração.

Na quarta tela, os arquivos gerados são disponibilizados ao usuário, para que sejam gravados em diretórios locais.

B.1.1 Tela Inicial do Ambiente

Na tela inicial, o usuário pode selecionar uma entre três opções como pode-se ver na figura B.1:

A opção **Novo Algoritmo** permite o desenvolvimento de um novo algoritmo distribuído, ou seja, um algoritmo que ainda não foi utilizado neste ambiente. Para selecionar esta opção, o usuário deve marcá-la e clicar no botão OK, quando uma tela será apresentada ao usuário onde ele deverá descrever as informações gerais referentes ao novo algoritmo.

Após preencher as informações gerais, dois botões podem ser acionados. O botão **Limpar** tem a função de limpar os dados que foram preenchidos pelo usuário e retoma a situação inicial da tela.

Ao clicar o botão **Enviar Dados do Algoritmo**, os dados informados pelo usuário são verificados e, caso sejam válidos, uma nova tela é apresentada, contemplando os dados gerais do algoritmo já informados, além de campos para a entrada dos dados referentes à programação do algoritmo. Esta tela permite que o usuário altere os dados já informados na tela de informações gerais, editada anteriormente.

Após preencher todas as informações específicas para programação do algoritmo, o usuário poderá iniciar a geração dos programas, clicando no botão **Gerar**. As informações contidas nesta tela serão validadas, e é realizada, efetivamente, a criação de um programa que executa o algoritmo distribuído especificado. O resultado deste processo de geração do programa é apresentado ao usuário na tela seguinte.

A segundo opção, **Abrir Algoritmo já existente**, permite que os dados de um algoritmo que já foi criado neste ambiente possam ser alterados. Para selecionar esta opção, o usuário deverá marcá-la, informar, no campo "Arquivo", o nome do arquivo de configuração gerado durante a criação do mesmo, e clicar no botão OK. O arquivo de configuração tem a extensão ".cfg". Ao clicar no botão "Browse..." ou "Procurar", ao lado deste campo, o usuário poderá selecionar o arquivo desejado.

Após selecionar esta opção, uma tela com os dados do algoritmo informado é apresentada ao usuário, onde ele poderá alterá-los. O usuário não poderá alterar o número de pares Entrada/Ação, nem o tipo de algoritmo (síncrono ou assíncrono).

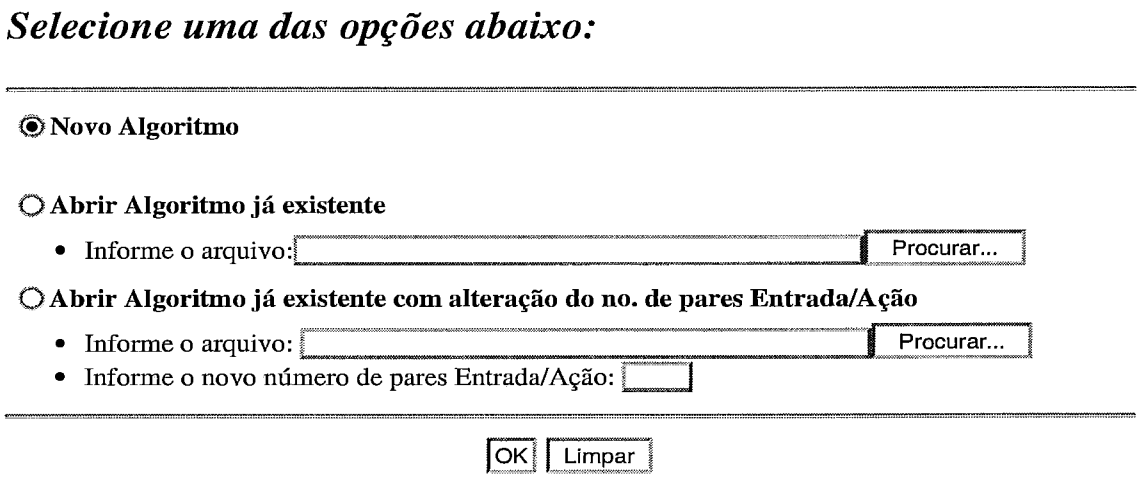

Figura B.1: Interface Gráfica - Tela Inicial do Ambiente

Assim que o usuário tiver realizado as alterações desejadas, ele deverá clicar no botão **Gerar**, e então os dados serão verificados e uma tela será apresentada com o resultado da geração do programa.

Na opção **Abrir Algoritmo já existente com alteração no número de pares Entrada/Ação**, o usuário poderá alterar além dos dados do algoritmo, o número de pares Entrada/Ação. Para selecionar esta opção, o usuário deverá marcá-la, informar, no campo "Arquivo", o nome do arquivo de configuração gerado durante a criação do mesmo, indicar o novo número de pares Entrada/Ação, e clicar no botão OK. O arquivo de configuração tem a extensão ".cfg". Ao clicar no botão "Browse..." ou "Procurar...", ao lado deste campo, o usuário poderá selecionar o arquivo desejado.

Após selecionar esta opção, uma tela com os dados do algoritmo informado é apresentada ao usuário, assim como o número de campos para a informação dos pares Entrada/Ação de acordo com o valor informado pelo usuário. Nesta tela, os dados poderão ser alterados.

Análogo ao processo realizado para a opção anterior, após realizar as alterações desejadas, o usuário poderá gerar o novo programa, clicando no botão **Gerar**.

Após a geração do programa que executa o algoritmo distribuído criado ou alterado, o usuário poderá salvar os arquivos gerados em um diretório local. Para isto, basta clicar no botão **Salvar Arquivos**, que apresenta um tela com os arquivos gerados. O arquivo chamado "Readme.txt" explica ao usuário como o programa deve ser executado, e o que contém cada um dos demais arquivos.

Além do arquivo Readme.txt, mais três arquivos são disponibilizados para usuário. O primeiro deles é um arquivo executável, que não possui extensão, e seu nome é o mesmo do algoritmo gerado. O segundo é um programa que contém o mapeamento dos processos nos processadores (extensão '.aloc'). E finalmente, o terceiro é um arquivo que inicia a execução do programa gerado.

B.1.2 Informações Gerais

Chamamos de Informações Gerais, os dados que não fazem parte do código do algoritmo distribuído, propriamente dito. São elas:

Nome do Algoritmo Corresponde ao nome do algoritmo, que pode ter até 10 caracteres (Obrigatório)

Tipo de Algoritmo Indica se o algoritmo é Assíncrono ou Síncrono. Default: Assíncrono. (Obrigatório)

Sincronizador Indica o tipo de sincronizador Alfa ou Beta. Default: Alfa. (Obrigatório para algoritmos síncronos)

Topologia Indica a forma como os nós estão ligados. Opções disponíveis: Anel não-direcionado; Anel direcionado; Malha; Torus; Hipercubo; Grafo Completo; Grafo genérico não-direcionado; Grafo genérico direcionado. (Obrigatório)

Número de Nós/Dx/Dy Indica o número de nós que deverão executar o algoritmo, dimensão X e dimensão Y no grafo, respectivamente. Esses dados devem ser apresentados separados por espaço(s) em branco. (Número de Nós: obrigatório; Dx e Dy: obrigatório para topologia Malha ou Torus)

Descrição da Topologia Indica como os nós estão ligados numa topologia geral. A topologia deve ser informada seguindo o formato especificado para este campo (Ver seção "Formatos")

Arquivo para Descrição da Topologia Indica o caminho + nome do arquivo que possui a descrição da topologia, seguindo o formato especificado para este campo (Ver seção "Formatos"). Ao clicar no botão "Browse..." ou "Procurar...", ao lado deste campo, o usuário poderá selecionar o arquivo desejado.

Número de nós que iniciam o Algoritmo Para o algoritmo assíncrono são os nós que iniciam espontaneamente, e para o algoritmo síncrono são os processos que realizam ação no pulso 0. (Obrigatório quando a opção "Todos" não está marcada)

Marque aqui se TODOS os Nós iniciam o Algoritmo Quando marcada, indica que todos os nós iniciam o algoritmo.

Número dos Nós que iniciam o Algoritmo Lista com o número dos nós que iniciam o algoritmo, separados por espaço(s) em branco. (Obrigatório quando a opção "Todos" não está marcada e um arquivo não foi indicado)

Arquivo com o número dos nós que iniciam o Algoritmo Indica o caminho + nome do arquivo que possui a descrição dos números dos nós que iniciam o algoritmo, separados por espaço(s) em branco. Ao clicar no botão "Browse..."

ou "Procurar...", ao lado deste campo, o usuário poderá selecionar o arquivo desejado.

Tipo (em C) e número de elementos da mensagem Indica o tipo de dado em C e o número de elementos deste tipo referentes às mensagens responsáveis pela comunicação entre os processos. Todas as mensagens enviadas pelo algoritmo devem ser de um mesmo tipo. (Obrigatório)

Número máximo de mensagens em trânsito Determina o tamanho máximo do *buffer* de comunicação, considerando número de mensagens. (Obrigatório)

Plataforma Identifica o sistema no qual o programa gerado irá executar. As opções disponíveis na lista são Sun/Solaris, IBM/Aix e SP2, porém somente a opção Sun/Solaris está implementada.

Número de Processadores Indica em quantos processadores diferentes o algoritmo deverá executar. (Obrigatório quando a opção "UM único processador" não está marcada)

Marque aqui se é UM único processador Quando marcada, indica que todos os nós estarão alocados em um único processador.

Nome do Processador único Indica o nome do processador (máquina) em que todos os nós devem ser alocados. (Obrigatório quando a opção "UM único processador" está marcada)

Informe os Processadores e os respectivos nós Indica como os nós estão alocados aos processadores da rede. A descrição desta informação deve seguir o formato especificado para este campo (Ver seção "Formatos")

Informe o Arquivo com o processadores e os respectivos nós Indica o caminho + nome do arquivo que possui a descrição da alocação dos nós aos processadores da rede, seguindo o formato especificado para este campo (Ver seção "Formatos") Ao clicar no botão "Browse..." ou "Procurar...", ao lado deste campo, o usuário poderá selecionar o arquivo desejado.

Informe o Número de Pares Entrada/Ação Indica o número de Pares Entrada/Ação necessários para a execução do algoritmo distribuído. Este número deve ser, no máximo, 20.

B.1.3 Informações Específicas do Algoritmo

Chamamos de "Informações Específicas do Algoritmo", os dados referentes ao código do algoritmo distribuído, seguindo o modelo apresentado em [2]. Todas estas informações devem ser descritas utilizando a linguagem de programação C. São elas:

Bibliotecas C utilizadas pelo Algoritmo Indica as bibliotecas necessárias para a execução do código referente ao algoritmo distribuído desejado (Opcional)

Arquivo que contém as Bibliotecas C Indica o caminho + nome do arquivo que possui a descrição das Bibliotecas da linguagem C. Ao clicar no botão "Browse..." ou "Procurar...", ao lado deste campo, o usuário poderá selecionar o arquivo desejado.

Variáveis Apresenta a descrição das variáveis do algoritmo, assim como a declaração de tipos que possam ser utilizados no mesmo. (Obrigatório)

Arquivo com as Variáveis Indica o caminho + nome do arquivo que possui a descrição das variáveis. Ao clicar no botão "Browse..." ou "Procurar...", ao lado deste campo, o usuário poderá selecionar o arquivo desejado.

Inicialização das Variáveis Apresenta um código em C que permite a inicialização de variáveis. (Opcional)

Arquivo com a Inicialização das Variáveis Indica o caminho + nome do arquivo que possui a descrição da inicialização das variáveis. Ao clicar no botão "Browse..." ou "Procurar...", ao lado deste campo, o usuário poderá selecionar o arquivo desejado.

Condição de Término Representa uma expressão booleana que, quando satisfeita para um determinado processo, força o término da execução do algoritmo neste processo. Ele é formada de apenas uma linha que contém a expressão descrita na linguagem C. (Obrigatório)

Arquivo para Pares Entrada/Ação Indica o caminho + nome do arquivo que possui a descrição dos pares Entrada/Ação, seguindo o formato especificado para este campo (Ver seção "Formatos"). Ao clicar no botão "Browse..." ou "Procurar...", ao lado deste campo, o usuário poderá selecionar o arquivo desejado.

Descrição dos Pares Entrada/Ação Contempla a descrição das Entradas e respectivas Ações para cada par que constitui o algoritmo distribuído. O número de campos apresentados na tela para a descrição dos pares corresponde ao número informado pelo usuário na tela de Informações Gerais. No caso de algoritmos assíncronos, uma condição também pode ser informada, o que restringirá a execução da Ação no momento do recebimento da mensagem. (Ver seção Pares Entrada/Ação) (Obrigatório)

B.2 Formatos

B.2.1 Topologia

No formato definido para a informação da topologia, cada linha possui os vizinhos correspondentes ao nó com o valor do número da linha. No caso de grafo orientado apenas os vizinhos de saída são incluídos (nós para os quais este nó pode enviar mensagens), e no caso de grafo não-orientado, apenas os vizinhos com valor menor que o próprio devem ser incluídos. Quando um determinado nó possui mais de um vizinho que se enquadre nessas condições, eles devem ser inseridos separados por espaço(s) em branco. Caso o nó não possua vizinhos nas condições acima, deve-se deixar a linha referente a esse nó em branco.

Um exemplo desse formato é apresentado abaixo:

```
1 2
3
4

0
```

No exemplo acima, o nó 0 possui vizinhos de saída 1 e 2; o nó 1 possui o vizinho 3; o nó 2 possui o vizinho de saída 4; o nó 3 não possui vizinhos de saída; e o nó 4 possui o vizinho 0 como saída. Uma figura que retrata a topologia deste exemplo é apresentada na figura B.2.

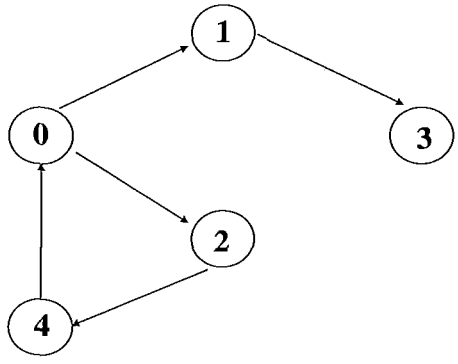

Figura B.2: Topologia Geral

B.2.2 Alocação a Processadores

No formato definido para a informação dos processadores, cada linha corresponde a um processador. O primeiro argumento é o nome do processador e os dados subseqüentes são os números dos processos que irão executar no processador em questão, separados por espaço(s) em branco. Abaixo segue um exemplo do padrão referenciado acima.

```
miami 1 2
lisboa 0 3 5
porto 4
```

Neste exemplo, estamos considerando que o algoritmo distribuído será executado por seis processos em três processadores (miami, lisboa e porto): os processos 1 e 2 serão executados no processador "miami"; os processos 0, 3 e 5 em "lisboa"; e o processo 4 no processador "porto".

O processo 0 (zero) tem um significado especial. Este é o processo inicializador da execução do algoritmo. Portanto o processador no qual ele deve executar deve ser o mesmo no qual a execução deve ser disparada (started).

B.2.3 Arquivo para Pares Entrada/Ação

No formato definido para a informação dos pares Entrada/Ação, cada par é separado por uma *string*, definida como uma seqüência de cinco caracteres '-'. A primeira linha de cada par corresponde à Entrada.

No caso de algoritmos assíncronos, a segunda linha corresponde a uma condição que deve ser testada no momento do recebimento da mensagem, e se for satisfeita, implica na execução da Ação. Esta linha é obrigatória, de forma que, se não existir uma condição específica para o par, esta linha deve conter o número '1', indicando condição '*true*'. No caso de algoritmos síncronos, está linha não é necessária.

Em seguida, o código referente à Ação para o par deve ser descrito. Para facilitar a codificação do algoritmo, foram criadas algumas rotinas, que são descritas na próxima seção (Pares Entrada/Ação).

B.3 Pares Entrada/Ação

Os pares Entrada/Ação consistem em três campos, para algoritmo assíncronos e dois campos para algoritmos síncronos. São eles, Entrada/Condição/Ação e Entrada/Ação, respectivamente.

Para algoritmos assíncronos, a Entrada de cada par pode estar associada a uma Condição. No campo "Entrada" o usuário deve informar um TAG, ou seja, uma constante que identifica o tipo de mensagem que corresponde à ação do par. No primeiro par do algoritmo assíncrono este TAG é fixo e igual a **NIL**, representando que a ação correspondente será executada por todos os processos que iniciam espontaneamente, independente do recebimento de uma mensagem, conforme padrão no qual a implementação deste ambiente foi baseada. [2] Para os demais pares, este TAG deve ser iniciado por uma letra, e é sempre obrigatório. No programa gerado, este TAG será uma constante em C, que identifica o tipo da mensagem.

O campo "Condição" permite que o usuário teste uma condição antes de realizar a ação referente ao par em questão. É constituído de uma expressão booleana descrita utilizando a linguagem de programação C. Caso o usuário não preencha este campo, assume-se que a ação será executada no momento do recebimento da mensagem, o que corresponde a uma condição sempre verdadeira, ou "1" na linguagem C. Se a condição não for satisfeita, a mensagem recebida é incluída em uma lista. Toda vez que uma mensagem é recebida, esta lista é percorrida, e a condição é testada novamente, até que seja satisfeita.

O campo "Ação" consiste numa seqüência de comandos na linguagem C, que devem ser executados no momento do recebimento de uma mensagem do tipo especificado no campo "Entrada" caso a condição especificada no campo "Condição" seja

satisfeita. Além dos comandos da linguagem C, algumas rotinas foram implementadas e estão disponíveis ao usuário, de forma a facilitar a codificação do algoritmo. São elas:

- send_to
- send_all
- vizinho
- calc_num_viz

A rotina *send_to* realiza o envio de uma mensagem para um dos vizinhos do processo que chama a mesma. A declaração desta função é send_to(dest tag msg), onde *dest* corresponde ao processo que vai receber a mensagem, *tag* representa o tipo da mensagem e *msg* é o nome da variável que contém a mensagem a ser enviada.

Outra rotina disponível ao usuário é **send_all** que realiza o envio de uma mensagem para todos os vizinhos do processo que invocou essa rotina. Sua declaração é send_all(tag msg), sendo *tag* o tipo da mensagem que será enviada e *msg* o nome da variável que contém esta mensagem. Pode-se notar que a diferença entre as rotinas *send_to* e *send_all* é que a primeira possui um parâmetro a mais que se refere ao processo destino do send.

Um exemplo destas duas rotinas se encontram abaixo:

```
send_to(3 TAG1 nome);
send_all(TAG2 letra);
```

No primeiro exemplo, uma mensagem será enviada para o processo 3 com tag 'TAG1' cuja mensagem é o conteúdo da variável nome. De forma semelhante, o segundo exemplo identifica que uma mensagem com tag 'TAG2' será enviada para todos os vizinhos do processo , e a mensagem é o conteúdo da variável letra.

A rotina **vizinho** retorna um indicador se o processo passado como parâmetro é vizinho do processo que está chamando a rotina. Sua declaração é vizinho(proc, tipo) onde *proc* corresponde ao número do processo que é se deseja saber se é vizinho e *tipo*, que determina qual a natureza do vizinho. Os tipos de vizinhos possíveis, correspondendo ao segundo parâmetro da função são 'I' de entrada (**I**nput), 'O' de

saída (**O**utput) e 'B' se for ambos (**B**oth), ou seja, de entrada e saída. Esta rotina retorna **0** caso do processo indicado não ser vizinho, e **1**, caso contrário.

A rotina **calc_num_viz**, como o próprio nome diz, retorna o número de vizinhos de um determinado tipo, que é passado como parâmetro. Ela possui apenas um parâmetro, correspondendo ao tipo de vizinho em questão, e seu retorno corresponde ao número de vizinhos referentes ao tipo indicado. Sua declaração é então calc_num_viz(tipo), onde tipo pode ser 'I' , 'O' ou 'B', como descrito anteriormente.

Diferente dos algoritmos assíncronos, nos algoritmos síncronos, não existe uma Condição atrelada ao campo "Entrada", pois o que vai determinar a execução da ação é o pulso do relógio e não o recebimento da mensagem. O campo "Entrada" contém, nesse caso, o intervalo de tempo correspondente à Ação do par considerado. Por exemplo, para o intervalo de s=2 até s=4, o algoritmo executa uma determinada ação. Para este par Entrada/Ação, o campo "Entrada" recebe o valor (s>1&&s<5). Pode-se observar que este campo contém uma expressão booleana que representa um intervalo de tempo.

O primeiro par Entrada/Ação para algoritmos síncronos corresponde ao instante s==0, que é o momento do início do algoritmo. Isso é equivalente à ação dedicada aos processos que iniciam espontaneamente no algoritmo assíncrono.

O campo "Ação", de forma análoga ao algoritmo assíncrono, corresponde à uma lista de comandos na linguagem C, que serão executados caso a expressão descrita como Entrada para o par seja satisfeita. Todas as observações feitas para este campo em algoritmos assíncronos, vale para os algoritmos síncronos.

Algumas variáveis internas ao programa gerado também são visíveis ao usuário, para que o código possa ser escrito mais facilmente. A variável **id** determina o número do processo que está executando o algoritmo.

Outra variável também importante para a codificação é **source** que determina, no momento do recebimento da mensagem, qual o processo emissor (remetente) da mesma. Desta forma é possível determinar códigos diferentes dependendo do processo que enviou a mensagem.

A variável **recbuf** contém a última mensagem recebida pelo processo. Seu tipo e tamanho são os mesmos definidos pelo usuário no arquivo de configuração.

A variável **processor_name** determina o nome do processador onde o processo está executando.

No algoritmo síncrono, as mensagens recebidas durante um determinado pulso

são armazenadas em uma variável chamada **message**. Esta variável é um vetor de duas posições. Cada posição é um apontador para uma lista que contém as mensagens referentes a um pulso. O cálculo para se identificar o índice no vetor da variável **message** referente a um determinado pulso P é bem simples. Este índice será o resto da divisão inteira de P por 2.

Cada elemento da variável **message** corresponde a uma estrutura composta de três campos, como se pode ver a seguir:

```
typedef struct type_message
{
   int source;
   int message[N];
   struct type_message *prox;
}type_message;
```

O primeiro campo é o "*source* que indica qual o nó que enviou a mensagem. A seguir vem o campo *message*, que guarda o conteúdo da mensagem propriamente dito. O tipo C desse campo é o mesmo da mensagem que transita entre os nós. E o terceiro campo é um ponteiro para identificar o próximo elemento da lista.

O código apresentado abaixo pode ser utilizado para "varrer" a lista de mensagens recebidas num determinado pulso 's'.

```
msg_aux = message[s%2];
while(msg_aux != NULL)
{
   ...
   msg_aux = msg_aux->prox;
}
```

A variável **msg_aux** é uma variável auxiliar que aponta para um elemento da lista de mensagens. Inicialmente ele aponta para o primeiro elemento da lista e vai avançando para os próximos elementos através do comando "msg_aux = msg_aux->prox;". O valor de cada elemento da lista pode ser consultado através da variável *msg_aux->message*.

Apêndice C

MPI

C.1 Introdução

O MPI (Message Passing Interface) é um padrão para passagem de mensagens que facilita o desenvolvimento de bibliotecas e aplicações paralelas. MPI está disponível desde 1994, executando em máquinas de processamento paralelo "massivo" (MPP-massively parallel processing) e redes de workstations (NOWs - networks of workstatios). Grande parte dessas máquinas têm oferecido MPI pelos seus fabricantes. Por isso MPI tem alcançado um dos seus objetivos - adicionar credibilidade para a computação paralela. [17] [18] [4]

O principal objetivo do MPI, como a maior parte dos padrões, é o grau de portabilidade entre diferentes máquinas. Isso significa que o mesmo código fonte da passagem de mensagem pode ser executado em uma variedade de máquinas desde que a biblioteca MPI esteja disponível, enquanto que algumas melhorias podem ser necessárias para tirar maior vantagem das características de cada sistema.

Um outro tipo de compatibilidade oferecido pelo MPI é a capacidade de executar transparentemente em sistemas heterogêneos, isto é, conjuntos de processadores com arquiteturas distintas. O usuário não precisa se preocupar se o código está enviando mensagens entre processadores de arquiteturas iguais ou diferentes. As implementações MPI farão automaticamente qualquer conversão de dados necessária e utilizar o protocolo de comunicação correto.

Um importante objetivo do projeto do MPI foi permitir implementações eficientes em máquinas com características diferentes. MPI tenta evitar a especificação de como as operações acontecerão, ele somente especifica o que uma operação faz logicamente.

Escalabilidade (por exemplo, permitir que um sistema possa rodar tanto em um

sistema pequeno - 2 processadores - como em um sistema grande - 1000 processadores) é um outro objetivo do processamento paralelo. MPI oferece ou suporta escalabilidade através de várias de suas características.

Como todos os bons padrões, MPI é valioso pois ele define um comportamento mínimo e conhecido das implementações de passagem de mensagens. Isso libera o programador de se preocupar com certos problemas que podem ocorrer. MPI garante que a transmissão básica de mensagens é confiável. O usuário não precisa verificar se uma mensagem foi recebida corretamente.

O padrão MPI é usado para especificar a comunicação entre um conjunto de processos formando um programa concorrente. O paradigma de passagem de mensagens é atrativo por causa da grande portabilidade e escalabilidade. É facilmente compatível tanto com multicomputadores de memória distribuída como com multiprocessadores de memória compartilhada, NOWs, e uma combinação desses elementos. A interface é adequada para uso de programas Multiple Instruction Multiple Data, (MIMD), ou programas Multiple Program Multiple Data (MPMD), onde cada processo segue um caminho de execução distinto através do mesmo código, ou mesmo executa um código diferente. É também adequado para programas escritos num estilo mais restrito de Single Program Multiple Data (SPMD), onde todos os processos seguem o mesmo caminho de execução através do mesmo programa.

Ainda que muito do MPI siga as práticas comuns de sistemas de passagem de mensagem já existentes, MPI vai além na definição de características avançadas como tipos de dados definidos pelo usuário, portas de comunicação permanentes, operações para comunicação coletiva poderosas e diferentes mecanismos para comunicação. Nenhum sistema já existente possui todos esses fatores.

C.2 Objetivos do MPI

O objetivo do MPI é desenvolver um padrão largamente utilizado para programas de passagem de mensagens. Para isso, os seguintes fatores são observados:

- Desenvolver uma interface para programação de aplicações
- Permitir comunicação eficiente, evitando cópias memória-memória e suportando sobreposição de computação e comunicação
- Permitir implementações que possam ser utilizadas em ambientes heterogêneos

- Permitir ligação (*binding*) conveniente da linguagem C e Fortran 77 para a interface
- Oferecer uma interface de comunicação confiável
- Definir uma interface não muito diferente das atuais, oferecendo extensões o que permite uma maior flexibilidade
- Definir uma interface que pode ser implementada em várias plataformas de fabricantes sem alterações significantes no software de comunicação básica e de sistema
- O projeto da semântica da interface ser independente da linguagem
- O projeto da interface permitir o sistema de thread seguro (*thread safety*)

C.3 O Que Está Incluído no MPI?

O padrão MPI inclui:

- Comunicação ponto-a-ponto, ou seja, troca de mensagens entre pares de processos
- Comunicações coletivas, ou seja, operações de comunicação ou sincronização envolvendo grupos inteiros de processos
- Grupos de processos. Como os grupos de processos são usados e manipulados
- Comunicadores. Um mecanismo para oferecer escopos de comunicação separados para módulos e bibliotecas. Cada comunicador especifica um espaço distinto para processos e um contexto de comunicação distinto para mensagens.
- Topologias de Processos. Funções que permitem a manipulação conveniente de rótulos de processos quando os processos são considerados como formando uma topologia particular.
- Ligações (*Binding*) para Fortran 77 e ANSI C

- *Profiling interface.* A interface é designada de forma que o *profiling* de tempo de execução ou as ferramentas para monitoração de performance podem ser unidas ao sistema de passagem de mensagens
- Gerenciamento de Ambiente e funções de pesquisa

C.4 Termos e Convenções do MPI

C.4.1 Especificação de Procedimentos

Os argumentos de chamadas de procedimentos utilizados pelo MPI são indicados como:

IN - a rotina usa mas não atualiza o argumento apontado como IN
OUT - a rotina pode atualizar um argumento apontado como OUT
INOUT - a rotina usa e atualiza um argumento apontado como INOUT

Existe um caso especial - se um argumento é um handle para um objeto opaco, e o objeto é atualizado pela chamada à rotina, então o argumento é apontado como OUT. Ele é marcado dessa forma mesmo que o handle não seja modificado - nós utilizamos o atributo OUT para denotar que "o que" o handle referencia é atualizado.

A definição do MPI tenta evitar o uso de argumentos INOUT, pois estes são propensos a erros.

Um argumento do tipo OUT ou INOUT não pode ser atribuído (aliased) a qualquer outro argumento passado para a rotina MPI.

Exemplo:

```
void procedure (int *pin, int *pout)
 *pout++ = *pin++;
```

C.4.2 Termos de Semântica

Processos

Um programa MPI consiste de processos autônomos, executando seus próprios códigos, em um estilo MIMD. Os processos se comunicam via chamadas para primitivas de comunicação do MPI. Normalmente, cada processo executa em seu próprio

espaço de endereçamento, ainda que implementações com memória compartilhada de MPI sejam possíveis.

Um processo pode ser seqüencial ou pode ser multi-threaded, com threads possivelmente executando concorrentemente. O atual padrão não provê criação ou deleção dinâmica de processos durante a execução do programa. Os processos são identificados de acordo com seu rank no grupo, que é um inteiro.

Tipos de Rotinas MPI

As operações de comunicação podem ser do seguinte tipo:

local: se o término da rotina depende somente do processo que está executando localmente, tal operação não requer uma comunicação com um outro processo usuário.

não-local: se o término da rotina pode requerer a execução de alguma rotina MPI sobre outro processo.

bloqueante: se o retorno da rotina indica que o usuário está autorizado a reusar recursos especificados na chamada da rotina.

não-bloqueante: se a rotina pode retornar antes que a operação iniciada pela chamada se complete, e antes que o usuário seja autorizado a reusar os recursos especificados na chamada da rotina.

coletiva: se todos os processos no grupo de processos precisam invocar a rotina.

C.4.3 Características com a Linguagem C

A definição de constantes protótipos de funções e definições de tipos devem ser fornecidas no arquivo de include mpi.h.

O código de retorno de sucesso será MPI_SUCCESS, mas os códigos de retorno de erros são dependentes de implementação.

Os argumentos de arrays são indexados desde zero.

Flags lógicos são inteiros com valor 0 significando false e 1 significando true.

C.5 Comunicação Ponto-a-Ponto

O envio e o recebimento de mensagens por processos, é o mecanismo de comunicação básico da MPI. As operações básicas da comunicação ponto-a-ponto são o **send** e

receive. A sua utilização é ilustrada no exemplo abaixo. [11]

```
#include "mpi.h"
main(argc, argv)
int argc;
char **argv;
{
    char msg[20];
    int myrank;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
    if(my_rank == 0) /* código para o processo zero */
    {
        strcpy(msg, "Hello, there");
        MPI_Send(msg, strlen(msg), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Recv(msg, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

Neste exemplo, o processo zero (myrank = 0) envia uma mensagem ao processo um, utilizando a operação de send MPI_SEND. Esta operação especifica o buffer de send na memória do processo emissor de onde o dado da mensagem é buscado. No exemplo acima, o buffer de send consiste na parte da memória onde está localizada a variável *msg*, no processo zero. A locação, o tamanho e o tipo do buffer de send são especificados através dos 3 primeiros parâmetros da operação send. A mensagem enviada irá conter 13 caracteres desta variável. Além disso, a operação de send associa um envelope à mensagem. Este envelope especifica o destino da mensagem e contém informações que a diferenciam e que podem ser utilizadas pela operação

de receive para selecionar uma mensagem em particular. Os últimos 3 parâmetros da operação de send especificam o envelope para a mensagem enviada.

O processo um (myrank = 1) recebe esta mensagem através da operação de receive MPI_RECV. A mensagem a ser recebida é selecionada de acordo com o valor do seu envelope, e o dado da mensagem é armazenado no buffer de receive. No exemplo acima, o buffer de receive consiste na parte da memória onde está localizada a string *msg*, no processo um. Os 3 primeiros parâmetros da operação de receive especificam a locação, o tamanho e o tipo do buffer de receive. Os próximos 3 parâmetros são utilizados para selecionar a mensagem recebida. O último parâmetro é utilizado para retornar informação sobre a mensagem recebida.

MPI provê um conjunto de funções **send** e **receive** que permitem a comunicação de dados com um tag associado. O tag permite a seleção de mensagens do lado receptor; um programa usuário pode receber sobre um tag específico, ou o programa pode mascarar esse valor permitindo a recepção de mensagens com qualquer tag. Isso também ocorre no lado que envia.

MPI_COMM_WORLD é um comunicador default. Entre outras coisas, um comunicador define o conjunto de processos que se permite participar da operação de comunicação. Processos possuem um número inteiro (rank) que serve de identificação do processo e são identificados através de uma pesquisa ao comunicador. O tipo da mensagem é evidenciada na função para envio ou recepção. A variável status, atualizada pela função MPI_Recv(), informa sobre a fonte e o tag da mensagem, e o número de elementos recebidos.

MPI provê funções send e receive bloqueantes e não-bloqueantes. No caso bloqueante, a chamada send bloqueia até que o buffer send possa ser recuperado; isto é, depois do send, o processo que o executou pode reutilizar a variável que continha a mensagem enviada. Similarmente, a função receive bloqueia até que o buffer receive realmente possua o conteúdo da mensagem. As funções não bloqueantes sempre vêm em duas partes: funções de envio, que começam a operação solicitada, e a função teste-término, que permite que o programa descubra se a operação solicitada foi completada.

Existem dois aspectos da comunicação: as primitivas da semântica da comunicação e os protocolos básicos que as implementam. Se um usuário deseja implementar um protocolo minimizando a cópia e o armazenamento de dados, a semântica mais natural seria a versão rendevouz onde o término do send implica que o recei-

ve tenha iniciado (pelo menos). Por outro lado, um protocolo que tenta bloquear processos por um intervalo de tempo mínimo necessariamente termina fazendo mais armazenamento e cópias de dados.

Como o objetivo fundamental do MPI é padronizar operações de passagem de mensagens sem prejudicar a performance, a decisão tomada foi incluir todas as opções importantes para a semântica ponto-a-ponto no padrão.

C.5.1 Modos para Comunicação Ponto-a-Ponto no MPI

Tanto a comunicação bloqueante como a não-bloqueante possuem modos. O modo permite que o usuário escolha a semântica da operação send, como conseqüência, influencia o protocolo básico para transferência de dados. Os modos para comunicação disponíveis no MPI são:

- **Padrão** (standard)
- **Buffered**
- **Síncrono**
- **Ready**

No modo **Padrão**, o MPI decide quando as mensagens serão bufferizadas. Se isto ocorrer, uma chamada send pode completar antes que o receive seja chamado. Por outro lado, o espaço de buffer pode não estar disponível, ou o MPI pode optar por não bufferizar mensagens por razão de performance. Neste caso, a chamada send não irá completar até que o receive correspondente tenha sido ativado e o dado tenha sido transferido para o receptor.

Então, um send **Padrão** pode ser inicializado independente do receive ter iniciado. Ele pode ser completado indepente do receive ter iniciado. O modo padrão pode não ser local, neste caso, a finalização correta de um send pode depender da ocorrência de um receive correspondente.

Uma operação de send no modo **Bufferizado**, pode ser iniciada independente de um receive correspondente ter sido inicializado. Ela pode completar antes do receive ter sido inicializado. Diferentemente do send padrão, esta operação é local e, sua terminação não depende da ocorrência de um receive correspondente. Então, se um send é executado e nenhum receive correspondente é inicializado, o MPI

deve guardar a mensagem em um **buffer**, de forma a permitir que a chamada seja completada. Um erro ocorrerá se o espaço no **buffer** for insuficiente. A quantidade de espaço disponível no **buffer** é controlado pelo usuário. Alocação de **buffer** pelo usuário pode ser requerida para o modo bufferizado a ser efetivado.

Um send que usa o modo **Síncrono** pode ser iniciado independente de um receive ter sido iniciado. Entretanto, o send será completado com sucesso somente se um receive correspondente tiver sido iniciado, e a operação de receive tenha iniciado o recebimento da mensagem enviada pelo send síncrono. Então, a finalização de um send síncrono não só indica que o buffer do send pode ser reutilizado, mas também indica que o receptor já atingiu um determinado ponto de sua execução. Se ambos, send e receive, são operações bloqueantes, então o uso do modo Síncrono fornece uma semântica de comunicação síncrona : uma comunicação não estará completa enquanto ambas as operações tenham finalizado. Um send executado nesse modo é não-local.

Um send que utiliza o modo de comunicação **Ready** somente pode ser iniciado após um receive correspondente ter sido iniciado pelo processo receptor. Esse modo permite que o usuário explore conhecimento extra para simplificar o protocolo e potencialmente realizar uma performance mais alta.

C.5.2 Alocação de Buffer de Comunicação

O MPI permite que o usuário especifique um buffer para ser utilizado em mensagens enviadas pelo send no modo buffered. A alocação do buffer é realizada pelo processos emissor, através da rotina MPI_Buffer_attach. Essa rotina tem como parâmetros o endereço inicial e o tamanho do buffer, provendo ao MPI um buffer na memória do usuário para ser utilizado para armazenar mensagens. O buffer é utilizado apenas por mensagens enviadas no modo buffered. Apenas um buffer pode ser definido a um processo de cada vez.

Através da rotina MPI_Buffer_detach o buffer pode ser desalocado na memória. Essa operação será bloqueada até que todas as mensagens que estão no buffer tenham sido transmitidas.

Quando nenhum buffer é associado , o MPI se comporta como se o um buffer de tamanho zero estivesse associado ao processo. Nesse caso, ao enviar uma mensagem no modo buffered, um erro ocorrerá por falta de espaço no buffer.

C.5.3 Tipos Definidos pelo Usuário

Normalmente, a comunicação ponto-a-ponto envolve apenas buffers contíguos contendo uma seqüência de elementos do mesmo tipo. Esta condição é restritiva demais, por dois motivos. Freqüentemente, desejamos passar mensagens que contêm diferentes tipos de dados (p. ex., um contador inteiro, seguido de uma seqüência de números reais); e freqüentemente queremos enviar dados não-contíguos (p. ex., um sub-bloco de uma matriz). Uma solução para este problema é compactar os dados não-contíguos num buffer contíguo no emissor da mensagem e descompactá-los novamente no receptor. Este procedimento tem a desvantagem de requerer operações de cópia memória-a-memória adicionais em ambos os sites, mesmo quando o subsistema de comunicação tem capacidade de dispersão-agrupamento. Por outro lado, o MPI fornece mecanismos para especificar buffers de comunicação mais gerais, mistos e não-contíguos.

Buffers de comunicação mais gerais são especificados substituindo-se os tipos de dados básicos por tipos de dados derivados, que são construídos a partir de tipos de dados básicos, utilizando construtores oferecidos pelo MPI. Através de tipos definidos pelo usuário, MPI suporta comunicação de estruturas de dados complexas, tais como estruturas que contém combinações de tipos de dados primitivos.

C.5.4 Comunicações Coletivas

As comunicações coletivas transmitem dados através de todos os processos especificados pelo objeto comunicador. Uma função, a barreira, serve para sincronizar processos sem passagem de dados. MPI oferece as seguintes funções de comunicação coletiva:

- Sincronização por barreira (barrier) através de todos os processos
- Broadcast de um processo para todos
- Coletando (gathering) dados de todos os processos para um
- Dispersando (scattering) dados de um para todos
- Allgather - como o gather seguido por um broadcast da saída do gather
- Alltoall - como um conjunto de gathers no qual cada processo recebe um resultado distinto

- Operações de redução global - como soma, max, min, e funções definidas pelo usuário

- Scan (ou prefix) através de processos

Para manter o número de funções e suas listas de argumentos a um nível de complexidade razoável, os projetistas do MPI fizeram as funções coletivas mais restritas que as funções ponto-a-ponto. Uma restrição é que a quantidade de dados enviada deve ser exatamente igual à quantidade de dados especificada pelo receptor. Essa restrição foi imposta para evitar a necessidade de variáveis de status como um argumento para as funções.

A principal simplificação é que as funções coletivas vêem somente nas versões bloqueantes. E uma simplificação final diz respeito aos modos, que possui somente um tipo que pode ser considerado como análogo ao modo padrão (standard) do ponto-a-ponto.

C.6 Tratamento de Erros

Uma implementação MPI pode ou não manipular alguns erros que ocorram durante chamadas de MPI. Estes podem incluir erros que geram exceções ou traps, tais como erro de ponto flutuante ou violações de acesso. O conjunto de erro que são manipulados pelo MPI é dependente da implementação. Cada erro gera uma exceção MPI.

Um usuário pode associar um manipulador de erros à um comunicador. A rotina de tratamento de erros especificada será usada para qualquer exceção de MPI que ocorra durante uma chamada à MPI para uma comunicação, com este comunicador. Chamadas MPI que não estão relacionadas a nenhum comunicador, são consideradas ligadas ao comunicador MPI_COMM_WORLD. A ligação de manipuladores de erros com os comunicadores é puramente local: diferentes processos podem ligar diferentes manipuladores de erros para o mesmo comunicador.

Um comunicador posteriormente criado herda o manipulador de erros que é associado com o comunicador "pai". Em particular, o usuário pode especificar um manipulador de erros "global" para todos os comunicadores, associando este manipulador com o comunicador MPI_COMM_WORLD imediatamente após a inicialização.

Vários manipuladores de erro pré-definidos estão disponíveis no MPI. São eles:

- MPI_ERRORS_ARE_FATAL - O manipulador, quando chamado, faz com que o programa aborte todos os processos em execução. Isto causa o mesmo efeito que uma chamada à MPI_ABORT pelo processo que invoca o manipulador.

- MPI_ERRORS_RETURN - O manipulador não tem outro efeito além de retornar o código de erro para o usuário.

As diferentes implementações do MPI podem prover manipuladores de erros pré-definidos adicionais e os programadores podem codificar seus próprios manipuladores de erro.

O manipulador de erros MPI_ERRORS_ARE_FATAL é associado, por default, com MPI_COMM_WORLD, após a inicialização. Então, se o usuário escolhe não controlar a manipulação de erros, todo erro que MPI manipular é tratado como fatal. Como (quase) todas as chamadas MPI retornam um código de erro, o usuário pode escolher manipular erros em seu código principal, testando o código de retorno das chamadas MPI e executando um código de recuperação adequado, quando a chamada não teve sucesso. Neste caso, o manipulador de erro MPI_ERRORS_RETURN será usado. Geralmente é mais conveniente e mais eficiente não testar os erros após a chamada MPI, e ter tal erro manipulado por um manipulador de erro MPI, não trivial.

Após um erro ter sido detectado, o estado do MPI é indeterminado. Ou seja, usando um manipulador de erros definido pelo usuário, ou MPI_ERRORS_RETURN, não é aconselhável ao usuário continuar usando o MPI após um erro ter sido detectado. O propósito desses manipuladores de erros, é permitir ao usuário emitir mensagens de erro apropriadas e tomar ações não relacionadas ao MPI (tal como "esvaziar" o buffer de I/O) antes de finalizar o programa.

O padrão MPI permite a criação de novos manipuladores de erros e a associação destes manipuladores ao communicador. As funções utilizadas para isto são MPI_ERRHANDLER_CREATE, que registra uma função do usuário, informada como parâmetro, como um manipulador de exceções MPI; e a função MPI_ERRHANDLER_SET, que associa o novo manipulador de erros ao comunicador informado como parâmetro no processo chamador. Note que um manipulador de erro está sempre associado com o comunicador.

Para dissociar um manipulador de erros de um comunicador, também são utilizadas duas funções:

MPI_ERRHANDLER_GET, que identifica qual o manipulador de erros associado;

MPI_ERRHANDLER_FREE, que marca o manipulador de erros associado com o comunicador para desalocação.

O manipulador de erros será desalocado após todos os comunicadores associados à ele terem sido desalocados.

Referências Bibliográficas

[1] Árabe, José Nagib Cotrim, Beguelin, Adam, Lowekamp, Bruce, Seligman, Erik, Starkey, Mike, Stephan, Peter, "Dome: Parallel Programming in a Distributed Computing Environment", Proceedings - 10th International Parallel Processing Symposium - Honolulu, Hawai - IEEE COMPUTER SOCIETY PRESS, pag. 218-224, 1996

[2] Barbosa, Valmir C., An Introduction to Distributed Algorithms. The MIT Press Cambridge, Massachusetts, London, England, 1996.

[3] Couto, Kêmio de Oliveira, Mendes, Marco Aurélio de Souza, Carvalho, Osvaldo S. F., "Ambiente de Desenvolvimento e Avaliação de Algoritmos de Exclusão Mútua para Sistemas Distribuídos", XIV Congresso da Sociedade Brasileira de Computação - VI SBAC/PAD - Caxambu - MG Brasil, pag. 123-135, 1994

[4] Dongarra, Jack J., Otto, Steve W., Snir, Marc, Walker, David, "A Message Passing Standard for MPP and Workstations", Communications of The ACM - Vol. 39, No. 7, pag. 84-90, July 1996

[5] Felton, Mark, CGI: Internet Programming with C++ and C. Prentice Hall 1997

[6] Graham, Ian S., HTML A Referência Completa para HTML 3.2 e extensão HTML. Editora Campus 1997

[7] Hart, Delbert, Kraemer, Eileen, Roman, Gruia-Catalin, "Interactive Visual Exploration of Distributed Computations", Proceedings - 11th International Parallel Processing Symposium, Geneva, Switzerland - IEEE COMPUTER SOCIETY PRESS, pag. 121-127, 1997

[8] Hwang, K., Advanced Computer Architecture: Parallelism, Scalability, Programmability. McGraw-Hill Computer Science Series, 1993

[9] LeVitus, Bob, Webmaster - How to build your own World Wide Web Server without really trying. AP Professional 1994

[10] Lusk, Ewing and Butler, Ralph, User's Guide to the p4 Parallel Programming System. Argonne National Laboratory 1994

[11] Lusk, Ewing and Gropp, William, User's Guide for mpich, a Portable Implementation of MPI. Argonne National Laboratory 1996

[12] Lusk, Ewing and Gropp, William, Installation Guide to mpich, a Portable Implementation of MPI. Argonne National Laboratory 1996

[13] Lynch, Nancy A., Distributed Algorithms. MK(Morgan Kaufmann) 1996

[14] Mullen, Robert, HTML4 - Guia de Referência do Programador. Editora Ciência Moderna 1997

[15] Schildt, H., C The Complete reference. 3 ed, 1997.

[16] Tel, Gerald, Introduction to Distributed Algorithms. Cambridge 1994

[17] University of Tennessee. MPI : A Message Passing Interface Standard. Manual de MPI. Knoxville, Tennessee. June, 12, 1995.

[18] Walker, D.W., Dongarra, J.J., MPI : a standard Message Passing Interface, Supercomputer, v.20,n.63, pp. 56-68, 1996.

[19] Weinman, William E., Manual de CGI. Makron Books 1997

[20] Welch, Brent B., Practical Programming in Tcl and Tk. Prentice Hall PTR, 1997