FERRAMENTAS PARA O DESENVOLVIMENTO DE PROGRAMAS PARALELOS DISTRIBUÍDOS

Astrid Luise Harriet Hellmuth

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

Aprovada por:

Prof. Valmir C. Barbosa, Ph.D.
(Presidente)

Prof. Cláudio Luis de Amorim, Ph.D.

Prof. Ageu Cavalcanti Pacheco Júnior, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
JANEIRO DE 1991

HELLMUTH, ASTRID LUISE HARRIET

Ferramentas Para o Desenvolvimento de Programas
Paralelos Distribuídos [Rio de Janeiro] 1991
VIII, 65 p., 29.7 cm, (COPPE/UFRJ, M. Sc.,
Engenharia de Sistemas e Computação, 1991)
TESE — Universidade Federal do Rio de Janeiro, COPPE
1. Processamento Paralelo
I. COPPE/UFRJ II. Título(série).

Este trabalho é dedicado à Mutti e a todos os meus amigos.

Desejo agradecer a todos os meus colegas da COPPE-Sistemas, pelo apoio e estímulo, em especial ao meu Orientador, por sua compreensão e confiança, à Ana e à Lúcia por suas constantes colaborações e ao Lélio por sua inestimável ajuda.

Agradeço também a convivência com o pessoal do Laboratório de Computação Gráfica, sempre incentivadora.

Seria impossível enumerar todas as pessoas que contribuiram de diversas formas para que eu pudesse realizar este trabalho. Muitas vezes apenas com a sua distante torcida, com um sorriso, com a sua companhia, me transmitiram aquela energia que nos faz ir adiante. Sou imensamente grata a todos esses amigos queridos.

Por todos estes motivos, desejo agradecer com todo o carinho ao Cláudio Esperança por estar sempre ao meu lado.

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

Ferramentas para o Desenvolvimento de Programas Paralelos Distribuídos

Astrid Luise Harriet Hellmuth

Janeiro de 1991

Orientador: Valmir C. Barbosa
Programa: Engenharia de Sistemas e Computacão

Este trabalho aborda diversos aspectos relacionados ao projeto e implementação de programas distribuídos. As principais fontes de dificuldade para o desenvolvimento de tais programas são investigadas com o objetivo de identificar procedimentos que possam ser realizados com o auxílio de ferramentas. Apresentamos, então, um ambiente de programação que oferece um modelo simples para o projeto de aplicações e um conjunto de ferramentas para geração automática do código para a aplicação descrita e seu mapeamento de maneira eficiente em uma arquitetura.

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

Tools for Developing Parallel Distributed Programs

Astrid Luise Harriet Hellmuth
January, 1991

Thesis Supervisor: Valmir C. Barbosa
Department: Engenharia de Sistemas e Computacão

This work covers many aspects related do the design and implementation of distributed programs. The main sources of difficulties in developing such programs are investigated in order to identify activitires that can be performed by assistant tools. A programming environment that offers a simple model for application design and a set of tools for automatic code generation and efficient mapping on an architecture is presented.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo I

Introdução

Este trabalho aborda o desenvolvimento de um ambiente de programação de aplicações para sistemas distribuídos ponto-a-ponto. Definimos um conjunto de ferramentas visando os seguintes objetivos:

1. facilitar o desenvolvimento das tarefas que compõem uma aplicação, utilizando um modelo abstrato, que permite ao programador especificar sua aplicação com um formato similar ao encontrado na literatura para descrever algoritmos distribuídos;

2. dar suporte ao modelo de comunicação entre tarefas de modo que ela ocorra de forma segura, sem a ocorrência de interbloqueios (*deadlocks*), uma vez que esta é uma das principais fontes de dificuldade encontrada na programação paralela;

3. extrair um bom desempenho da aplicação do usuário, balanceando a carga de processamento com o emprego de um algoritmo para distribuição de tarefas entre processadores;

4. livrar o programador de qualquer preocupação com relação a arquitetura da máquina onde a aplicação será executada, realizando a configuração automática da aplicação, isto é, o ambiente se encarrega de todas as operações necessárias para atribuição das tarefas aos processadores e manutenção da rede de comunicação entre tarefas.

Os conceitos básicos relacionados ao processamento paralelo são apresentados no Capítulo II.

Descrevemos as principais abordagens para o desenvolvimento de aplicações paralelas no Capítulo III. Identificamos as propriedades comuns às diversas descrições de algoritmos encontradas na literatura e as principais fontes de dificuldade encontradas durante a implementação desses algoritmos.

No Capítulo IV propomos um modelo genérico para a elaboração de programas paralelos, que permite ao programador expressar apenas as características essenciais ao comportamento de uma aplicação. A capacidade de armazenamento de mensagens trocadas entre as tarefas é uma questão crucial para a implantação do modelo de comunicação, que também analisamos nesse Capítulo.

Em seguida, no Capítulo V, expomos os problemas a serem resolvidos durante o mapeamento de aplicações em arquiteturas multiprocessadas, e descrevemos a utilização de um algoritmo para o balanceamento da carga de processamento.

Mostramos, no Capítulo VI, que a linguagem OCCAM oferece uma forma adequada para expressar o paralelismo de uma aplicação projetada de acordo com o modelo. No Capítulo VII descrevemos um ambiente para a geração automática do código correspondente à descrição fornecida nos termos do modelo de programação proposto.

Durante o projeto e implementação das ferramentas, procuramos empregar os resultados obtidos em trabalhos anteriores, reunindo-os de forma a obter um conjunto integrado de soluções para os problemas que se apresentam durante o desenvolvimento de uma aplicação distribuída. A utilização de um ambiente do tipo que propomos facilita a geração de programas corretos e confiáveis, com bom desempenho, proporcionando um aumento de produtividade ao programador.

Capítulo II

Processamento Paralelo

II.1 Multiprocessadores

Conforme computadores mais poderosos são oferecidos à comunidade de usuários, maiores se tornam as suas exigências. Existe porém um limite para a máxima velocidade alcançável com um computador baseado em um único processador. Além do mais o seu custo cresce mais rapidamente de acordo com a proximidade de tal limite. Uma solução radicalmente diferente é mover de uma arquitetura uniprocessada para uma nova arquitetura empregando paralelismo, ou seja, um certo número de processadores cooperantes [1].

A essência do paralelismo não é nova. A natureza é cheia de casos onde criaturas aparentemente frágeis e incapazes de grandes feitos, tais como formigas e abelhas, realizam façanhas incríveis através do empenho coletivo. A própria tecnologia que possibilitou o surgimento destas recentes arquiteturas é o resultado de esforços humanos coletivos: graças aos avanços da tecnologia VLSI tornou-se possível a fabricação, com baixo custo, de módulos de circuito integrado contendo diversos processadores, ou processador, memória e circuitos de comunicação.

As formas usualmente encontradas para organização desse novo tipo de arquitetura são conhecidas como SIMD e MIMD [2]. SIMD é uma abreviação para uma Seqüencia de Instruções, Múltiplas seqüências de Dados e corresponde aos chamados processadores vetoriais, que executam simultaneamente a mesma seqüência de instruções sobre diferentes conjuntos de dados. MIMD é uma abreviação para Múltiplas seqüências de Instruções, Múltiplas seqüências de Dados, i.e., podemos ter diferentes seqüências de controle executando diferentes instruções e manipulando diversos conjuntos de dados. A segunda opção é a mais flexível e corresponde aos chamados multiprocessadores.

II.2 O Conceito de Processamento Concorrente

O *processamento concorrente* é definido como o uso de diversas entidades cooperantes (idênticas ou não), trabalhando em conjunto para atingir um objetivo comum [3].

Na computação concorrente, as entidades são computadores e o objetivo pode ser um cálculo científico de larga escala (como por exemplo previsão meteorológica), ou uma aplicação de inteligência artificial (tal como ganhar um campeonato de xadrez ou controlar o sistema de defesa de uma nação).

Uma computação típica consiste de um algoritmo aplicado a um extenso conjunto de dados denominado *domínio*. A concorrência é alcançada por decomposição desse domínio, i.e, o conjunto de dados original é divido em partes (*grains*) que são atribuídas a cada computador. Os algoritmos executados nos computadores são similares aos empregados no processamento seqüencial, a menos de duas diferenças fundamentais:

- o conjunto de dados sobre o qual aplica-se o algoritmo é quantitativamente diferente, uma vez que ele corresponde a uma parte apenas do domínio;
- a necessidade de cooperação entre os computadores tende a envolver alguma forma de comunicação entre eles.

Como cada computador trata apenas de uma parcela do conjunto original de dados, espera-se que o tempo gasto para tratar todo o domínio do problema seja menor do que se fosse tratado por um único computador. Intuitivamente, a razão entre o tempo para execução seqüencial em um único computador, e o tempo gasto durante o processamento concorrente deve ser muito próximo ao número de computadores empregados.

II.3 Concorrência e Paralelismo

De acordo com o contexto, os termos **concorrente** e **paralelo** são por vezes usados com significados similares porém distintos. Diz-se que duas entidades são executadas em paralelo se, em algum instante de tempo, ambas estão realmente sendo executadas. Em contrapartida, duas entidades são descritas como concorrentes se existe a possibilidade de executá-las em paralelo. Uma linguagem de programação concorrente possui, portanto, mais de uma seqüência distinta de controle, e os objetos que possam vir a ser executados em paralelo são diretamente representados [4].

Acreditamos que tal rigor na utilização dos termos **concorrente** e **paralelo** só se faz necessário quando se deseja distinguir entre a execução de um

programa concorrente em um único processador, ou seja, quando de fato não existe paralelismo, e em diversos processadores. Por esse motivo, utilizaremos ambos os termos livremente, sem distinção, uma vez que a capacidade de execução paralela estará sempre presente.

II.4 Processos e Tarefas

Os objetos que podem ser executados simultaneamente em um ambiente MIMD são normalmente conhecidos como **processos**. De maneira informal, cada processo pode ser visto como uma instância de um programa. Assim como um circuito elétrico pode possuir várias instâncias de um certo tipo de componente, uma computação concorrente pode conter várias instâncias de um ou mais programas [5].

Outro termo empregado com freqüência como sinônimo de processo é **tarefa**, especialmente quando o tratamento do processamento paralelo é baseado na idéia de processos seqüenciais comunicantes. Por exemplo, em [6] uma aplicação é definida como uma coleção de uma ou mais tarefas (ou processos) concorrentes.

Não é raro, entretanto, encontrar o termo tarefa, na literatura técnica, designando um conjunto de processos. Uma boa descrição para este tipo de distinção entre processos e tarefas é dada em [7] onde um programa a ser executado em um multiprocessador também é visto como uma coleção de elementos, denominados tarefas. Nesse contexto, cada tarefa é uma unidade de escalonamento a ser atribuída a um ou mais processadores e pode consistir de um ou mais processos. Cada processo é uma coleção de instruções de programa, e é definido como uma unidade indivisível com respeito à alocação do processador.

Em outras situações, o termo **tarefa** não é usado de todo. É o caso da linguagem de programação concorrente OCCAM, onde um programa, em seu nível mais alto, é visto como um único processo. Um processo pode conter outros processos, variando desde uma única atribuição até um programa completo, de forma que uma estrutura hierárquica é suportada. Ou seja, processos maiores são elaborados a partir dos chamados *processos primitivos* com o auxílio de elementos denominados *construtores* [8,9].

Enfim, ao investigar estes e outros exemplos da literatura, percebemos o seguinte:

- Os termos processo e tarefa são utilizados de forma intercambiável, i.e., na grande maioria dos casos é possível substituir uma palavra pela outra sem causar ambigüidade.
- No tratamento de problemas tais como o de divisão de um procedimento em diversos processos, ou *particionamento*, e atribuição de processos a processadores, ou seja, o *escalonamento*, como por exemplo em [14] é confortável considerar:

processo como uma *unidade de computação* e

tarefa como uma *unidade de escalonamento.*

II.5 Estados de um Processo

Para dar suporte ao conceito de processo, cada processador executa um código que permite a coexistência de múltiplos processos. Assim sendo, o número de processos concorrentes envolvidos em uma única computação pode exceder em muito o número de processadores nela envolvidos.

Uma vez que tenha sido criado, um processo pode se encontrar em diversos estados. Os três estados primários para um processo são [10]:

- executando
  Quando um processo estiver usando um processador para executar instruções;
- pronto ou executável
  Quando um processo pode iniciar (ou continuar) sua execução porém não há processador disponível;
- suspenso ou bloqueado
  Quando um processo estiver esperando a ocorrência de algum evento. Por exemplo, um processo pode ser temporariamente suspenso devido à indisponibilidade momentânea de algum recurso (processador, dados de entrada, memória, comunicação e assim por diante). Nesse caso um outro processo é escalonado para execução no processador.

II.6 Comunicação e Sincronização

Apesar de cada um dos processos possuir seu código próprio e variáveis particulares, eles raramente são independentes uns dos outros. As maiores dificuldades associadas à programação paralela surgem justamente quando a interação entre os processos é tratada; isso é, quando se especifica como a transferência de dados entre os processos ocorre e como eles devem sincronizar suas ações.

Existem dois esquemas complementares de comunicação entre processos [11]:

- Memória Compartilhada
- Troca de mensagens

Sistemas de memória compartilhada requerem que processos comunicantes compartilhem algumas variáveis. Espera-se que os processos troquem informações através do uso dessas variáveis. Variáveis compartilhadas são objetos aos quais dois ou mais processos têm acesso; a partir da leitura e/ou escrita dessas variáveis os dados são passados de um processo a outro.

Sistemas de troca de mensagens permitem que os processos envolvidos na comunicação enviem seus dados de maneira explícita através de mensagens.

Associado ao ato de comunicar dados está o conceito de sincronização de processos, que descrito de maneira simplificada, corresponde à necessidade de um processo adquirir algum conhecimento acerca das atividades de um outro processo, a fim de coordenar a execução de ambos. Por exemplo, para que um processo receba uma mensagem é necessário que algum outro processo a tenha enviado.

A sincronização é uma forma especial de comunicação, na qual o dado é uma informação de controle [12]. Ela serve ao duplo propósito de assegurar o correto seqüenciamento de processos e o acesso mutuamente exclusivo a dados compartilhados. Por exemplo, os mecanismos de sincronização podem ser usados para:

- Controlar um processo produtor e um processo consumidor de forma que o processo produtor não atualize dados que ainda não foram utilizados pelo consumidor, ou impedir que o processo consumidor obtenha dados inválidos;
- Proteger os dados em um banco de dados de forma que não sejam permitidos acessos concorrentes para escrita em um mesmo registro. Esse tipo de acesso pode levar à perda de uma ou mais atualizações se dois processos executarem uma leitura em seqüencia e depois uma escrita em seqüencia dos dados atualizados.

II.7 Sistemas Distribuídos Ponto-a-ponto

A computação onde os diversos elementos processadores não compartilham memória, e, portanto, a comunicação entre processadores é realizada exclusivamente através da troca de mensagens, também é denominada de computação distribuída [13]. Um dos critérios para classificá-la é justamente quanto à forma pela qual a comunicação entre os processadores acontece. Segundo esse critério, uma divisão bem ampla pode ser feita entre sistemas distribuídos onde a comunicação se efetua por difusão (*broadcast*), e aqueles em que a comunicação se dá de forma ponto-a-ponto.

Quando a comunicação é realizada por **difusão**, todos os processadores do sistema podem se comunicar diretamente com todos os outros, através do uso de um ou mais canais de comunicação compartilhados por eles. Já os sistemas **ponto-a-ponto** podem ser representados por grafos conexos em que cada

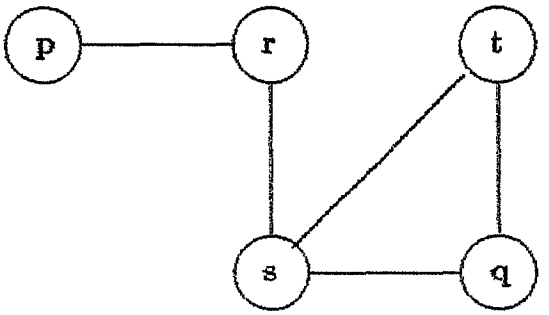


Figura II.1: Sistema Distribuído Ponto-a-Ponto

nó representa um processador e cada arco entre dois nós representa um canal de comunicação. Esse canal de comunicação é de uso exclusivo dos dois processadores representados por aqueles nós.

Essa forma de comunicação gera a necessidade de mensagens precisarem ser enviadas ao longo de rotas que passam por outros processadores que não sua origem e seu destino. Por exemplo, na figura ilustrando um sistema distribuído ponto-a-ponto, uma mensagem enviada pelo processador **p** com destino a **q** deverá, necessariamente passar pelos processadores **r** e **s**.

Neste trabalho, a discussão acerca do desenvolvimento de aplicações para sistemas MIMD de processamento paralelo, se referirá àqueles sistemas que possam ser identificados como sistemas distribuídos ponto-a-ponto.

Ω

Capítulo III

Desenvolvimento de Programas Paralelos

III.1 A Atividade de Programação

Antes de abordar o projeto de ferramentas para o auxílio à programação, é interessante analisar a atividade de programação em si, para determinar que mecanismos ou procedimentos podem ser incluídos para simplificá-la ou métodos que evitem complicações desnecessárias.

Quando nos propomos a resolver um determinado problema através do computador, precisamos, em primeiro lugar, elaborar um método ou algoritmo, que se seguido, nos dará o resultado deste problema. Podemos dizer que um algoritmo é uma seqüência de passos que manipulam informações a fim de obter um resultado. Um algoritmo é perfeitamente executável por uma pessoa, no entanto, não pode ser compreendido pelo computador. Necessário se faz traduzi-lo para alguma forma de linguagem que seja inteligível para a máquina ou linguagem de programação. Ao algoritmo assim traduzido damos o nome de programa.

O algoritmo é uma abstração descrevendo um procedimento, cuja generalidade transcende detalhes específicos de qualquer implementação. Como resultado, quando um algoritmo é especificado na literatura técnica, vários detalhes são omitidos propositadamente, ou, no máximo considerados implícitos, pois têm pouca ou nenhuma relevância na operação do algoritmo.

A programação é uma atividade de conversão de um *algoritmo* para uma forma chamada *programa*, que pode ser executada em um computador. Os detalhes omitidos precisam ser definidos no curso da programação uma vez que o computador só segue instruções explícitas. Portanto, a programação será mais fácil ou difícil, conforme a forma do algoritmo for ou não semelhante à forma do programa desejado.

Um exemplo simples seria a implementação de um algoritmo seqüencial, no qual o mecanismo de recursão é usado, através de uma linguagem de programação não-recursiva. Nesse caso, a programação se tornará difícil porque é necessário, de fato, implementar um pacote de suporte à recursão dentro dos mecanismos existentes na linguagem.

Assim sendo, é possível reduzir em grande parte as dificuldades encontradas na programação mantendo uma especificação ajustada àquela usada na literatura para descrever algoritmos, isto é, minimizando a conversão a ser realizada.

III.2 Ambientes de Programação

A forma original dos algoritmos pode, muitas vezes, parecer inatingível, uma vez que ela é revelada por uma sintaxe ou semântica pré-ordenada, dirigidas aos seus leitores e não às máquinas. Porém, freqüentemente encontramos características comuns em diversas especificações de algoritmos. A partir dessas propriedades podemos desenvolver mecanismos para o auxílio na tarefa de programação, que denominamos *ferramentas*.

Um *ambiente de programação* seria então a coleção de todas as ferramentas de linguagem e sistema operacional necessárias para suportar a programação, integradas em um único sistema [15] .

III.3 Geração de Programas Paralelos

Atualmente, a maioria das aplicações para sistemas distribuídos são desenvolvidas em um ambiente do tipo hospedeiro/máquina-alvo. Os programas são escritos em um computador hospedeiro e depois são carregados para execução no sistema alvo. O hospedeiro normalmente é uma estação de trabalho conectada ao multiprocessador através de um barramento ou de uma rede local. Vários multiprocessadores são programados dessa forma: iPSC, Cosmic Cube, NCUBE, etc [16].

Esse é o nível mais básico, e sempre presente, de ferramentas, ou seja, pacotes de programas executados no hospedeiro, que permitem a edição, compilação, depuração de aplicações e o seu carregamento na máquina-alvo para execução. Exemplos deste tipo de pacote são:

- Concurrent Workbench — pacote para desenvolvimento de programas para o iPSC, da Intel [17], contendo basicamente, compiladores C e FORTRAN, depurador e bibliotecas de rotinas vetoriais ;

- TDS (Transputer Development System) — sistema de desenvolvimento que permite a edição, compilação e execução de programas OCCAM para um transputer ou para uma rede de transputers [18].

No entanto, as facilidades oferecidas por tais sistemas não são suficientes para lidar com todos os aspectos envolvidos na programação paralela. Comparada à escrita de programas seqüenciais, a complexidade da programação para ambientes distribuídos é aumentada devido a fatores tais como: a concorrência, a comunicação e a sincronização entre os processos, e ainda à necessidade de mapear os diversos processos no conjunto de processadores disponível.

III.4 A Necessidade de Ferramentas de Auxílio à Programação Paralela

Em teoria, é fácil argumentar que um multiprocessador é mais rápido do que um computador com um único processador: divide-se uma tarefa em subtarefas, que são executadas simultaneamente em diferentes processadores, almejando alcançar um tempo de execução igual ao tempo gasto para executar a tarefa em um único processador dividido pelo número de processadores empregado na execução paralela. Embora válido, este raciocínio implica em considerar desprezível a despesa incorrida durante a decomposição e implementação de um algoritmo, o que está longe de ser uma verdade.

Existem basicamente três abordagens para o desenvolvimento de programas paralelos [14] :

- A primeira abordagem considera a paralelização um problema tão complexo que só pode ser resolvido manualmente. Por exemplo, Garg [19], divide os esforços de pesquisa para o tratamento de programas com múltiplas tarefas em dois grupos ou linhas: — manual, que deu origem a diversos sistemas para provar determinadas propriedades de algoritmos, tais como segurança, garantia de terminação, etc. ; e automática, que deu origem, por exemplo, a sistemas para análise automática de programas concorrentes, que exploram todos os comportamentos possíveis de um sistema.

  O tratamento e análise manual de programas distribuídos não é nada eficaz. Programadores tendem a cometer erros e não são muito eficientes em resolver problemas com muitas tarefas. Por exemplo, o interbloqueio entre tarefas do sistema (*system deadlock*) é um problema comum, difícil de detectar uma vez que o programa tenha sido desenvolvido.

- Outra forma de tratar o paralelismo seria através de compiladores para transformação automática de programas seqüenciais em programas paralelos. Smith [20], por exemplo, aponta como uma das dificuldades para o desenvolvimento

de programas concorrentes, a aprendizagem e compreensão das construções utilizadas para controlar o paralelismo. Assim sendo, em uma fase inicial, a paralelização automática, tal como a realizada por compiladores para otimização, pode ser um bom auxílio na superação deste tipo de dificuldade. No entanto, o paralelismo revelado dessa forma é restrito pelos algoritmos incluídos nos programas seqüenciais, e muitas vezes permite apenas a conversão de algoritmos seqüenciais determinísticos em algoritmos paralelos determinísticos equivalentes.

- Por fim, a terceira abordagem reconhece que algumas tarefas, tais como o projeto do algoritmo, são criativas, enquanto outras, como a inserção de sincronização, são solucionadas eficientemente utilizando ferramentas de programação. Dessa forma, um desempenho ótimo pode ser obtido com aumento de produtividade.

Na realidade, o processo de implementação de um algoritmo paralelo em um multiprocessador é usualmente específico de cada máquina e cheio de detalhes de nível mais baixo. O algoritmo paralelo deve ser expresso em uma linguagem suportada pelo multiprocessador. Essa linguagem pode exigir referências explícitas à organização da memória ou à estrutura de interconexão dos processadores, aos protocolos de sincronização e E/S, etc. A vantagem de computar em um ambiente multiprocessado é freqüentemente superada pela complexidade do processo de implementação.

Tais dificuldades na implementação de algoritmos paralelos tornam a computação multiprocessada inacessível a não ser para uma dedicada comunidade de usuários. Ela se tornará acessível a uma comunidade maior quando puder oferecer não só máquinas mais rápidas e poderosas como também ferramentas de programação com as quais se possa usá-las.

III.5 Características Comuns das Especificações de Algoritmos Paralelos

III.5.1 Introdução

O primeiro passo em direção à construção de um ambiente que minimize os esforços de desenvolvimento de aplicações para sistemas MIMD de processamento paralelo é identificar as características comuns às diversas descrições de algoritmos encontradas na literatura. Como vimos, os sistemas distribuídos ponto-a-ponto podem ser representados por grafos conexos em que cada nó representa um processador e cada arco entre dois nós representa um canal de comunicação. O projeto de um algoritmo distribuído envolve a especificação da computação a ser executada por cada nó do sistema, bem como da comunicação entre os diversos nós.

As principais propriedades comumente exibidas pelas descrições de algoritmos para o modelo de computação paralela com memória distribuída, são as seguintes:

- Um grafo cujos vértices representam processos e o cujos arcos representam a necessidade de troca de informações entre eles;
- Um conjunto de procedimentos descrevendo os tipos de atividade computacional encontrado no algoritmo e as interações entre elementos separados de computação;
- Declarações de entrada/saída descrevendo a forma assumida pelos dados e o formato dos resultados.

Embora possam existir outras características comumente exibidas pela especificação de algoritmos paralelos, esse conjunto de propriedades resume o que cientistas da computação descrevem sobre um algoritmo que desejam explicar.

III.5.2 O Grafo Descrevendo a Aplicação

É interessante notar que o grafo é na realidade um representante de uma *família* de grafos. Problemas com diferentes tamanhos de entrada exigirão grafos de diferentes tamanhos. Um bom exemplo de *detalhe comumente omitido* citado anteriormente como fonte de dificuldades nas implementações dos algoritmos, é a forma de tratamento dos casos onde o grafo tem mais vértices que o número de processadores disponíveis. Omite-se tais detalhes quando eles são óbvios, ou seus efeitos são inconseqüentes, ou porque são irrelevantes para a descrição do algoritmo em si. O grafo, ao contrário, é fundamental.

Comentamos anteriormente que as maiores dificuldades associadas à programação paralela surgem quando a interação entre os processos é tratada. São inúmeros os pormenores envolvidos quando a troca de informações entre os processos é especificada. Discutiremos os aspectos relacionados à troca de mensagens mais adiante.

III.5.3 O Conjunto de Procedimentos

Quanto à implementação do conjunto de procedimentos descrevendo as atividades computacionais dos algoritmos, dispomos basicamente de duas opções:

- utilizar qualquer linguagem de programação acrescida de primitivas de comunicação, tal como a linguagem C paralelo distribuída pela INMOS [6] para desenvolvimento de aplicações para *transputers*, ou

- utilizar uma linguagem de programação que ofereça as construções necessárias para expressar a concorrência entre os elementos de um programa paralelo. Exemplos de linguagem desse tipo são a linguagem LADY desenvolvida no projeto INCAS [21] e a linguagem OCCAM [4].

Uma linguagem oferecendo meios para a direta representação dos objetos executando em paralelo e suas formas de interação pode simplificar a tarefa de programação apresentando, em um nível mais alto, um modelo abstrato das computações distribuídas. A utilização de um modelo abstrato para a computação facilita não só a compreensão como a manutenção dos programas desenvolvidos.

III.5.4 Entrada e Saída

A declaração de E/S é curiosa: ela deve ser conhecida para programar o conjunto de processos, ela pode influenciar a sincronização, e pode ser crítica durante a demonstração da correção de alguns algoritmos. No entanto, ela parece entrar no processo explicitamente apenas após o programa ter sido escrito e estar pronto para ser executado. Usualmente, nos ambientes do tipo hospedeiro/máquina-alvo já citados, incluem-se rotinas de suporte à entrada e saída de dados entre as ferramentas e programas de apoio ao desenvolvimento de aplicações. É comum, por exemplo, a criação de processos com a finalidade única de trocar dados entre o hospedeiro e aqueles processos que, de fato, executam a aplicação.

III.6 A Troca de Mensagens

No modelo de computação paralela com memória distribuída, a troca de informações entre os processos é feita exclusivamente por intermédio de mensagens. Normalmente, a linguagem de programação concorrente provê operações primitivas para o envio e recepção de mensagens que denotaremos da seguinte forma:

- **envie** (*mensagem*)
- **receba** (*mensagem*)

Para que dois processos troquem mensagens entre si deve existir um elo (*link*) de comunicação entre eles. Existem diversas maneiras de implementar logicamente o elo de comunicação e as operações de transmissão e recepção de mensagens. Naturalmente, os métodos utilizados para implementação de tais elos de comunicação, também chamados de **canais**, influenciam diretamente a forma de programação de uma aplicação ([10] e [11]).

A primeira questão a ser levantada em relação à implementação de um elo de comunicação é quanto à sua capacidade de transmissão de mensagens

em uma direção ou em ambas as direções entre dois processos, i. e., os canais de comunicação entre os processos, representados pelos arcos no grafo que descreve a aplicação, podem ser:

- unidirecionais, ou
- bidirecionais.

Dizemos que um canal de comunicação é *unidirecional* se cada processo ligado ao canal pode receber ou enviar mensagens, mas não ambos. Quando cada processo ligado pelo canal puder tanto enviar como receber mensagens através do mesmo, o canal será chamado *bidirecional*.

Outro importante item a ser definido é a forma pela qual o destino e a fonte das mensagens serão identificadas. Nesse sentido podemos ter:

- *comunicação direta*

  Os processos transmissores e receptores devem fornecer uma identificação explícita uns dos outros. Essa forma de comunicação apresenta as seguintes características:

  - Quando há simetria no endereçamento, tanto o transmissor quanto o receptor precisam ser nomeados, e as primitivas de envio e recepção de mensagens são da seguinte forma:

    envie (P,*mensagem*)
    Envie *mensagem* para o processo **P**.
    receba (Q,*mensagem*)
    Receba *mensagem* do processo **Q**.

  - Quando o endereçamento é assimétrico, apenas o transmissor nomeia o processo receptor. O receptor não precisa nomear o transmissor. As primitivas de envio e recepção de mensagens são da seguinte forma:

    envie (P,*mensagem*)
    Envie *mensagem* para o processo **P**.
    receba (*id*, *mensagem*).
    Receba *mensagem* de qualquer processo; *id* é preenchido com o nome do processo com o qual a comunicação ocorreu.

  - Um elo de comunicação bidirecional é estabelecido automaticamente entre cada par de processos que desejam se comunicar.
  - Entre cada par de processos comunicantes existe apenas um elo de comunicação.

  A principal desvantagem da forma direta de comunicação é a modularidade limitada das definições dos processos resultantes. A troca do nome de um processo pode necessitar de um exame das definições de todos os outros processos. Todas as referências ao nome antigo devem ser achadas, de forma a modificá-la com o novo nome.

- *comunicação indireta*

  Nesse caso, os processos envolvidos na comunicação, nomeiam, de forma única, uma entidade intermediária, conhecida usualmente por caixa-postal ou porta. Informalmente, uma porta ou caixa-postal é um objeto onde mensagens podem ser colocadas e removidas por processos. Essa forma de comunicação apresenta as seguintes características:

  - Dois processos só podem se comunicar caso compartilherm alguma caixa-postal ou porta. As primitivas de envio e recepção de mensagens são da seguinte forma:

    envie (A,*mensagem*)
    Envie *mensagem* para a porta **A**.
    receba (A, *mensagem*).
    Receba *mensagem* da porta **A**;

  - Um elo de comunicação pode ser unidirecional ou bidirecional, e só é estabelecido entre dois processos quando eles compartilham alguma caixa-postal ou porta.
  - Entre um par de processos pode haver mais de um elo de comunicação, conforme os processos compartilhem de mais de uma porta ou caixa-postal.
  - Um processo pode se comunicar com vários outros, utilizando diversas portas diferentes.

  O uso da forma indireta de comunicação facilita tanto a modularização dos programas como a sua eventual modificação. É mais fácil acomodar modificações se a comunicação entre processos se dá entre intermediários explicitamente definidos.

III.7 Tipos de Mensagens

As mensagens enviadas por um processo podem ser de três tipos:

- mensagens com tamanho-fixo,
- mensagens com tamanho-variável,
- mensagens cujo formato é determinado por declarações de tipo fornecidas pela linguagem de programação.

Se apenas mensagens de tamanho fixo podem ser enviadas, a implementação é imediata. Essa restrição, no entanto, torna a tarefa de programação mais difícil. Por outro lado, mensagens de tamanho variável requerem uma implementação mais complexa, mas a programação se torna mais simples. O último caso

é basicamente aplicável apenas à comunicação indireta, e permite associar um determinado formato às mensagens trocadas por intermédio de uma caixa-postal ou porta.

III.8 Armazenamento de Mensagens

A operação de recepção de mensagens fornecida pela linguagem de programação concorrente usualmente apresenta uma característica *bloqueante*: o processo que a executa é suspenso até que a mensagem seja efetivamente recebida. Já a operação de transmissão de mensagens pode se comportar de maneira bloqueante ou não:

- Quando o envio de mensagens é feito de maneira *bloqueante*, o processo transmissor deve esperar até que o processo receptor esteje apto a recebê-la. Os dois processos precisam ser sincronizados para ocorrer a transferência da mensagem. Essa sincronização é chamada de *rendez-vous* entre os processos.
- Quando o envio de mensagens se realiza de forma *não-bloqueante*, permite-se que o processo transmissor continue a sua execução, mesmo antes de o processo destinatário da mensagem recebê-la. Diremos, informalmente, que o processo transmissor continua sua execução, sem esperar, enquanto a mensagem *está a caminho* ou *em trânsito*.

A característica bloqueante ou não-bloqueante do mecanismo de troca de mensagens depende diretamente de uma propriedade do elo de comunicação denominada *capacidade*. A capacidade de um elo de comunicação determina o número de mensagens que podem nele residir temporariamente. Podemos ver essa propriedade como uma fila de mensagens associada ao elo de comunicação, que pode ser implementada de três formas:

- *capacidade zero*

  A fila de mensagens associada ao elo de comunicação tem comprimento máximo igual a zero, portanto não pode haver nenhuma mensagem armazenada.

- *capacidade limitada*

  A fila de mensagens associada ao elo de comunicação possui comprimento finito n, portanto no máximo n mensagens podem estar temporariamente armazenadas no elo de comunicação.

- *capacidade ilimitada*

  A fila tem potencialmente comprimento infinito; assim sendo, é possível manter qualquer número de mensagens esperando na fila.

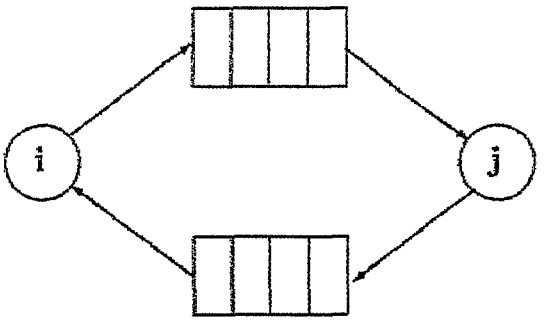


Figura III.1: Interbloqueio (*Deadlock*) na Comunicação

O caso de capacidade zero é às vezes referenciado como sistema de mensagens sem armazenamento; os outros casos fornecem armazenamento automático.

De acordo com essa classificação relativa à capacidade de armazenamento podemos dizer que:

- Operações de transmissão *bloqueantes* não requerem qualquer forma de armazenamento de mensagens, podendo ter lugar em elos de comunicação com *capacidade zero*. O processo transmissor deve aguardar o recebimento da mensagem antes de prosseguir a sua execução.

- Operações de transmissão *não-bloqueantes* requisitam o armazenamento de mensagens, ou seja, elos de comunicação com *capacidade não-nula* de armazenamento. Quando a capacidade de armazenamento for ilimitada, o processo transmissor não será suspenso. Já em um elo com capacidade de armazenamento finita $n, n > 0$, uma operação de envio de mensagem suspenderá o processo transmissor se o número de mensagens em trânsito naquele elo de comunicação for igual a n, i.e., se o número de mensagens armazenadas for igual à capacidade do canal. Se uma ou mais tarefas forem suspensas pode ocorrer um interbloqueio *deadlock* na comunicação. Por exemplo, suponha o caso ilustrado na Figura III.1. As tarefas i e j trocam mensagens através de dois canais: um canal da tarefa i para a tarefa j, e outro da tarefa j para a tarefa i. Cada um dos canais pode manter até quatro mensagens.

  Se em dado momento, exatamente quatro mensagens estiverem em trânsito, ou seja, armazenadas em cada um dos canais, e a tarefa i como a tarefa j tentarem uma outra transmissão, então ambas as tarefas serão suspensas. Uma condição de interbloqueio (*deadlock*) entre as tarefas é causada, pois nenhuma das duas poderá prosseguir a sua execução.

III.9 Comunicação Síncrona e Assíncrona

O modelo de passagem de mensagens onde a transmissão é feita de maneira bloqueante é, algumas vezes, referenciado como comunicação *síncrona*. Em contrapartida, quando a transmissão é não-bloqueante, a comunicação é dita *assíncrona*.

O comportamento síncrono na comunicação é uma das fontes de dificuldades na programação paralela. Existem autores que consideram a comunicação síncrona como uma desvantagem para a execução paralela, uma vez que que ela restringe o paralelismo possível e utiliza as operações de comunicação como o principal mecanismo de sincronização. Esse fato é especialmente ressaltado em [32], onde ainda se argumenta que, para muitas aplicações, o comportamento assíncrono é muito mais natural, e portanto, mecanismos para dar suporte à comunicação assíncrona devem ser oferecidos para a programação de tais aplicações.

Capítulo IV

Um Modelo Para Programação Distribuída

IV.1 Introdução

O objetivo principal para o desenvolvimento do nosso ambiente é fornecer ao programador uma visão de sua aplicação tal que minimize os seus esforços durante o processo de implementação. Para tanto, procuramos dar suporte a uma representação unificada de aplicações distribuídas, que permita ao programador preocupar-se somente com aquelas informações relevantes ao seu problema particular. Tendo em mente uma representação, ou modelo, para o sistema a ser desenvolvido, torna-se mais fácil raciocinar e analisar seu comportamento, sem que o programador se perca em meio a uma avalanche de detalhes de implementação.

IV.2 Descrição da Aplicação

Assim sendo, uma aplicação paralela pode ser projetada como um conjunto de tarefas concorrentes, com as seguintes características:

- Cada tarefa, ou processo, é uma unidade independente do projeto. O seu comportamento em relação às outras tarefas da aplicação pode ser descrito exclusivamente pelas mensagens que ela envia e recebe.
- Cada tarefa se comunica com outras tarefas através de canais ponto-a-ponto, ou seja, um canal conecta uma tarefa a exatamente uma única outra tarefa.
- Os canais são capazes de transmitir mensagens em apenas uma direção. Se for necessária a comunicação em ambas as direções entre duas tarefas, dois canais podem ser usados.

- Cada tarefa pode ter qualquer número de canais de entrada e saída.
- Os canais funcionam como entidades intermediárias, de forma que não é preciso endereçar as mensagens. Dito de outra forma, a comunicação é realizada de maneira indireta.
- Nenhuma restrição é imposta quanto ao tamanho e o formato das mensagens trocadas entre as tarefas.

Comentamos, anteriormente, que os sistemas ponto-a-ponto podem ser representados por grafos conexos em que cada nó representa um processador, e cada arco entre dois nós representa um canal de comunicação, de uso exclusivo dos dois processadores representados por aqueles nós. É natural, portanto, descrever uma aplicação para um sistema desse tipo também por um grafo, onde os nós seriam as tarefas, ou processos, e os arcos seriam os canais de comunicação entre as tarefas.

Não há necessariamente uma correlação entre o grafo da aplicação e o grafo do sistema onde ela será executada: tanto o número de nós como a estrutura de ligação entre eles podem ser completamente diferentes nos dois grafos.

Descreveremos, então, uma aplicação distribuída por um grafo direcionado, conexo

$$G_T = (N_T, E_T),$$

onde:

- N_T é um conjunto de processos ou tarefas
- E_T representa um conjunto de canais unidirecionais
- para cada $i \in N_T$, denotaremos por $In(i) \subseteq E_T$ o conjunto de canais chegando em i
- para cada $i \in N_T$, denotaremos por $Out(i) \subseteq E_T$ o conjunto de canais saindo de i

IV.3 Modelo Para Desenvolvimento das Tarefas

Uma forma confortável e simples de imaginar uma tarefa i do conjunto N_T seria da maneira seguinte:

envie uma mensagem em cada canal pertencente a um
subconjunto (possivelmente vazio) de $Out(i)$;
repita
receba mensagem no canal $c_1 \in In(i) \rightarrow$
execute alguma computação
e, eventualmente,
envie uma mensagem em cada canal pertencente
a um subconjunto (possivelmente vazio) de $Out(i)$;
ou ...
ou
receba mensagem no canal $c_m \in In(i) \rightarrow$
execute alguma computação
e, eventualmente,
envie uma mensagem em cada canal pertencente
a um subconjunto (possivelmente vazio) de $Out(i)$;
enquanto uma condição de terminação não for satisfeita

Após o envio de mensagens iniciais, a tarefa entra em um comando de repetição, onde uma entre as possíveis fontes de recepção de mensagens é escolhida. Para expressar essa escolha utilizamos a notação baseada naquela empregada para comandos guardados:

<guarda> → *<lista de comandos>*

Um *guarda* pode conter expressões booleanas e um comando de entrada. Se a avaliação das expressões booleanas resultar verdadeira e existir alguma mensagem a ser recebida (alguma outra tarefa executou uma operação de transmissão para a tarefa em questão), então dizemos que o guarda está *pronto*, e a *lista de comandos* pode ser executada. Em cada repetição exatamente um dos m comandos guardados é selecionado para execução, entre aqueles que possuírem guardas prontos, se houver algum. Se nenhum guarda estiver pronto a tarefa espera até que algum esteja.

IV.4 Modelo Para a Comunicação

Desenvolver uma aplicação distribuída como um conjunto de tarefas do tipo descrito torna-se muito simples para o programador, caso ele possa considerar os canais como estando sempre aptos a transmitir mensagens, sem causar a suspensão de suas tarefas. Portanto, o modelo de comunicação oferecido será o de comunicação assíncrona. O usuário poderá desenvolver suas tarefas considerando todos os canais com capacidade de armazenamento infinita.

IV.5 Prevenindo Interbloqueios

Na realidade, não existem canais de comunicação com capacidade infinita, pois sempre há um limite para o número máximo de mensagens que podem ser mantidas em um canal. Podemos, no entanto, oferecer um modelo de comunicação assíncrona entre as tarefas baseado na utilização de canais de capacidade finita, tomando o cuidado para que não ocorram interbloqueios *deadlocks* na comunicação, devido à falta de espaço para armazenamento de mensagens.

A fim de prover um mecanismo de comunicação onde não ocorram *deadlocks* durante a troca de mensagens entre as tarefas, é necessário algum tipo de informação sobre o número máximo de mensagens que podem vir a estar em trânsito em cada um dos canais. Na maioria dos casos, a determinação de limites superiores para o número de mensagens em cada canal é uma tarefa relativamente simples. O próprio programador da aplicação pode estabelecer tais limites com facilidade, como podemos verificar com o exemplo dado em [33]:

Exemplo: *Um caso especial de sincronizador* α

Seja uma aplicação cujo grafo é tal que sempre que um canal $(i \rightarrow j)$ existir, o canal $(j \rightarrow i)$ também faz parte do grafo. Cada processo inicia enviando uma mensagem a todos os seus vizinhos. Ele então espera o recebimento de mensagens do mesmo tipo de todos os seus vizinhos e repete a seqüência. Conseqüentemente, um processo i não enviará nunca uma outra mensagem no canal $(i \rightarrow j)$sem antes receber uma mensagem no canal $(j \rightarrow i)$. Logo, sempre haverá no máximo uma mensagem em trânsito em cada uma das direções.

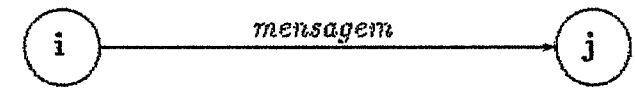
Se o número máximo de mensagens em cada canal for conhecido, podemos adotar uma estratégia de armazenamento tal que a ocorrência de *deadlocks* não seja possível. No caso desses limites serem dados para todos os canais, o método mais imediato é alocar um número correspondente de *buffers* para as mensagens. No entanto, essa forma é muito dispendiosa. É necessário então, empregar um algoritmo tal que resolva o seguinte problema:

Dado para cada canal c pertencente a E_T:

$r(c)$ — número máximo de mensagens em trânsito no canal c, se ele tivesse capacidade infinita.

Determinar:

$b(c)$ — número de *buffers* a serem alocados exclusivamente para o canal c.



canal (i → j) sem capacidade de armazenamento

canal (i → j) com capacidade de armazenamento

Figura IV.1: Introdução de Processos para Armazenamento de Mensagens

Ou seja, encontrar uma alocação ótima de *buffers*, i.e., que minimize o espaço de memória utilizada ou maximize a concorrência entre as tarefas, de forma a prevenir *deadlocks* na comunicação. O método empregado para solucionar este problema foi desenvolvido em um trabalho anterior e sua descrição pode ser encontrada em [33].

A reserva do espaço adequado para armazenamento de mensagens pode ser feita através de pedidos ao sistema operacional, se a operação **envie** (*mensagem*) fornecida pela linguagem de programação for uma primitiva não-bloqueante. Quando a operação de transmissão é definida de maneira bloqueante, esta forma explícita de armazenamento não é possível. Neste caso, podemos utilizar processos intermediários contendo variáveis locais para o armazenamento de mensagens ligados em série com canais de capacidade zero, conforme ilustra a Figura IV.1. O código executado pelo processo para armazenamento é do tipo:

repita
    receba (*mensagem*) → **envie** (*mensagem*)
enquanto uma condição de terminação não for satisfeita

Capítulo V

O Mapeamento de Aplicações em Arquiteturas

V.1 Introdução

Até agora vimos como desenvolver um programa correspondendo a um algoritmo distribuído. O programa resultante é composto por módulos computacionais que se interligam de acordo com uma estrutura descrita por um grafo. O desenvolvimento pode ser feito seguindo um modelo abstrato que facilita o projeto e a implementação de cada um dos módulos e garante que a comunicação se dará de forma segura e livre de interbloqueios (*deadlocks*).

Omitimos, propositadamente, qualquer discussão relativa à execução da aplicação em uma dada máquina paralela, com o intuito de ressaltar os seguintes pontos:

- durante a conversão do algoritmo em um programa distribuído, o programador deve estar atento apenas às propriedades e ao comportamento de sua aplicação. A adequação do programa a uma arquitetura específica pode e deve ser relegada a uma etapa posterior [28]
- o programador deve ter sempre em mente que um dado algoritmo paralelo pode ser executado em uma variedade de arquiteturas multiprocessadas, com significativas variações de desempenho de uma em relação à outra, determinadas pelas características de sua aplicação particular ([22], [23] e [24]).
- o desenvolvimento de programas voltados exclusivamente para uma arquitetura vem se tornando especialmente desaconselhado com o advento de sistemas paralelos reconfiguráveis tais como redes de *transputers* [27], a máquina CHiP [25], o multiprocessador PASM [26], etc.

V.2 O Mapeamento da Aplicação

A fim de executar o programa paralelo precisamos vincular os seus componentes aos recursos encontrados na máquina alvo. Assim, de um lado temos diversas tarefas e suas interrelações definidas pelo grafo de comunicação, e de outro temos diversos processadores (não necessariamente idênticos) e sua estrutura de interconexão. Portanto é necessário *mapear* um conjunto no outro, de forma eficiente, para extrair o rendimento potencial oferecido pela divisão da carga computacional entre diversas unidades de processamento.

Duas diferenças principais podem existir entre o grafo de comunicação entre tarefas e a estrutura de interconexão:

- primeiro, o número de tarefas pode ser diferente do número de processadores. No caso do número de tarefas ser maior, algumas abordagens para o problema de mapeamento sugerem a realização de uma etapa inicial de contração do grafo de tarefas ([14] e [30]), para acomodar o número de módulos para menor ou igual ao número de processadores. Usualmente, a etapa de contração procura minimizar o tempo de execução para o grafo contraído, e a etapa seguinte, ou seja, a de encaixe do grafo na estrutura de interconexão, procura atribuir cada nó do grafo de tarefas aos processadores minimizando o tráfego total de comunicação. Se o número total de tarefas for menor ou igual ao número de processadores, a etapa de contração é suprimida, e passa-se direto para o encaixe do grafo. Existem, no entanto, métodos eficientes para o mapeamento do grafo de tarefas, que consideram a possibilidade de atribuir mais de uma tarefa aos processadores, sem a necessidade de dividir o procedimento em duas etapas.

- a outra diferença é que a topologia (estrutura de comunicação) do grafo da aplicação pode não corresponder à topologia (estrutura de interconexão) do multiprocessador. Assim sendo, torna-se necessário prover algum tipo de suporte, em tempo de execução, para a aplicação distribuída, que inclua, por exemplo, mecanismos para o roteamento de mensagens, para a multiplexação dos canais de comunicação entre processadores, etc.

V.3 Alocação de Tarefas

O problema de alocação de tarefas corresponde à atribuição de tarefas de um programa aos processadores de um computador paralelo, com o objetivo de alcançar o melhor desempenho possível na execução da aplicação na arquitetura em questão.

Existem exemplos onde a alocação ou escalonamento de tarefas é realizado em tempo de execução, dinamicamente, tal como é proposto no sistema

SCHEDULE [31]. No entanto, o tempo gasto durante o escalonamento dinâmico das tarefas contribui para o tempo total para execução do programa. Essa contribuição não é desprezível e pode prejudicar, significativamente, o desempenho do sistema [1].

Quando a distribuição é feita na etapa inicial, antes das tarefas serem colocadas em execução no sistema, ele é denominado *balanceamento estático de carga* ou *alocação estática de tarefas.*

O problema de alocação estática de tarefas aos processadores pode ser formulado como um problema de isomorfismo de grafos, com o objetivo de minimizar uma função de custo. Para especificar o multiprocessador, a sua estrutura de conexão é representada, em geral, por um grafo não direcionado, uma vez que os canais de comunicação entre os processadores são, usualmente, bidirecionais. Dado um grafo de interação entre as tarefas, com pesos nos vértices caracterizando a carga computacional das tarefas, e pesos nos arcos que capturam a demanda de comunicação entre as tarefas, o problema é atribuir os vértices do grafo de tarefas aos processadores de maneira a otimizar algum critério de desempenho. Esse problema é intratável (conhecido como NP-completo), mesmo para estruturas simples como um hipercubo.

Conforme é apontado em [30], a maioria dos trabalhos sobre o assunto restringe a máquina-alvo a um tipo fixo, especializando-se na alocação de tarefas em arquiteturas determinadas, por exemplo, as arquiteturas em malha, hipercúbicas ou processadores vetoriais.

Um número limitado de trabalhos permitem não só a variação dos algoritmos quanto das arquiteturas, utilizando características de ambos para modelar o processo de alocação. Por exemplo, a abordagem proposta em [34] é bastante flexível e adequada para a geração automática de mapeamentos de programas em arquiteturas, apresentando as seguintes propriedades principais:

- não só os processadores podem ter diferentes capacidades de processamento, como as ligações entre eles também podem apresentar diferentes capacidades de comunicação;
- a estrutura de interconexão não é limitada à nenhum padrão específico: supõe-se a existência de uma estrutura de roteamento entre cada par de processadores;
- qualquer tarefa pode ser executada em qualquer processador, mesmo que duas tarefas que se comunicam sejam alocadas a processadores não conectados diretamente;
- uma alocação de tarefas a processadores é avaliada por uma função de custo: procura-se minimizar a comunicação entre processadores, mantendo, simultaneamente, um balanço da carga computacional entre os processadores. Esses

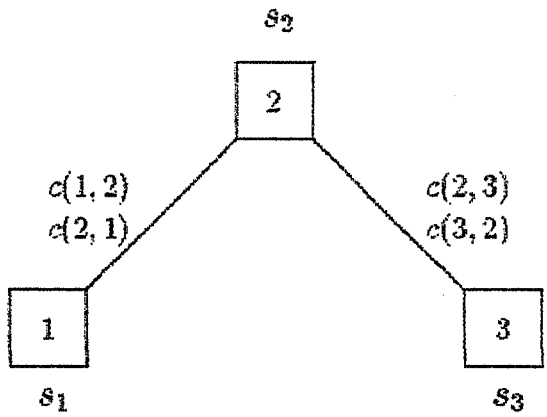


Figura V.1: Exemplo de Grafo de Processadores

dois objetivos são conflitantes: a necessidade de balancear a carga computacional tende a espalhar as tarefas entre os diversos processadores, ao passo que a necessidade de evitar muita comunicação tenderá a concentrá-las.

Vejamos, então, como o problema é formulado em [34]. As características de um multiprocessador são representadas por um grafo não-direcionado conexo (Figura V.1)

$$G_P = (N_P, E_P)$$

onde:

- N_P representa um conjunto de processadores
- E_P representa um conjunto de canais de comunicação bidirecionais entre pares de processadores
- a cada processador p em N_P é associada uma velocidade relativa do processador, representada por s_p
- para cada canal de comunicação entre processadores p e q em E_P é associada uma velocidade relativa do canal dada por $c(p,q)$
- o grafo G_P possui uma *estrutura fixa de roteamento* associada, dada pelo mapeamento :

  $$R : N_P \times N_P \rightarrow 2^{E_P}$$

  de forma que $R(p,q)$ é o conjunto de arcos na rota de p a q (possivelmente distinta de $R(q,p)$).

A velocidade relativa em cada canal de comunicação é considerada a mesma em ambas as direções, ou seja, $c(p,q) = c(q,p)$. Caso não exista ligação

direta entre p e q tem-se $c(p,q) = c(q,p) = 0$. Esse valores relativos são tais que, por exemplo, a razão s_p/s_q indica quão mais rápido o processador p é em relação ao processador q. O mesmo se aplica às velocidades dos canais de comunicação.

No caso da comunicação entre processadores não-vizinhos, a decisão do caminho que a mensagem tomará (roteamento) é dada pela matriz $R(p,q)$. Esta matriz é considerada como um dado do problema, fixa e conhecida previamente. De acordo com a descrição acima, se uma tarefa alocada ao processador p precisa se comunicar com outra alocada ao processador q, $R(p,q)$ será o conjunto de canais de comunicação por que deve passar a mensagem de forma que ela possa chegar a q. No grafo exemplo, temos $R(1,3) = \{(1,2),(2,3)\}$, ou seja, para o processador 1 enviar uma mensagem ao processador 3, ele deve fazê-lo através dos canais (1,2) e (2,3).

As características da aplicação são representadas por um grafo direcionado conexo (Figura V.2)

$$G_T = (N_T, E_T)$$

onde:

- N_T denota um conjunto de tarefas
- os arcos do conjunto E_T representam canais de comunicação entre as tarefas
- a cada tarefa i em N_T é associada uma demanda relativa de processamento dada por σ_i
- a cada arco $(i \rightarrow j)$ do conjunto E_T é associada uma demanda relativa de comunicação entre a tarefa i e a tarefa j, dada por $\gamma_{i,j}$, possivelmente diferente de $\gamma_{j,i}$
- a razão σ_i/σ_j indica o quanto mais processamento a tarefa i requer em relação à tarefa j, o mesmo valendo para as requisições de comunicação.

Uma *alocação* de tarefas a processadores é dada pelo mapeamento

$$A : N_T \rightarrow N_P$$

onde a representação $A(i) = p$ significa que a tarefa i está alocada para execução no processador p.

A carga total de uma alocação devida ao processamento num processador p é dada por:

$$PL_p = \frac{1}{s_p} \sum_{i|A(i)=p} \sigma_i$$

A *carga de processamento* PL_P é a soma das demandas de processamento σ_i de todas as tarefas i atribuídas por A para execução no processador p, com peso dado pelo inverso da velocidade relativa do processador p, s_p. O mesmo

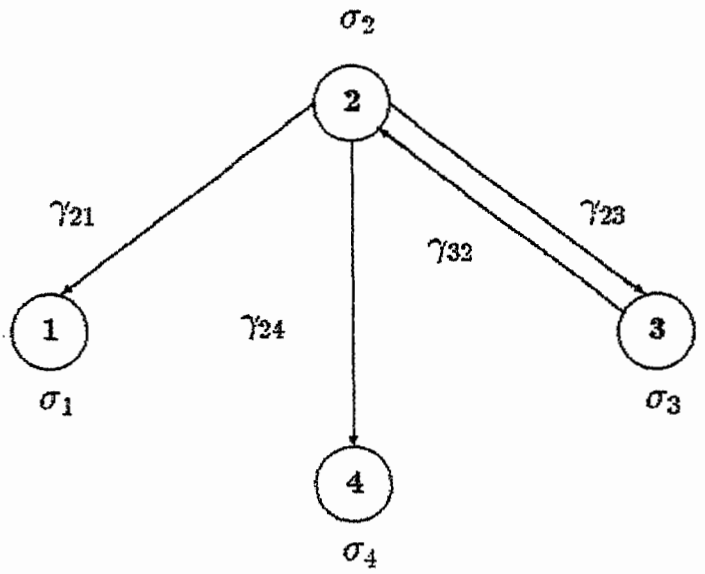


Figura V.2: Exemplo de Grafo de Tarefas

conjunto de tarefas, quando atribuídas para execução em um processador q mais rápido que p, o carregaria proporcionalmente menos, i.e., nós teríamos:

$$\frac{PL_p}{PL_q} = \frac{s_q}{s_p} > 1.$$

Da mesma forma, a carga de comunicação do canal (p, q) é denotada por $CL_{(p,q)}$, e é dada por:

$$CL_{(p,q)} = \frac{1}{c(p,q)} \sum_{i \neq j | (p,q) \in R(A(i),A(j))} \gamma_{i,j}.$$

A *carga de comunicação* $CL_{(p,q)}$ é composta das contribuições $\gamma_{i,j}$ de todos os pares ordenados de tarefas i e j tais que a rota de $A(i)$ para $A(j)$, conforme especificado por R, inclui o canal (p, q). As contribuições têm como peso o inverso da capacidade relativa do canal $c(p, q)$. Canais com capacidade maior do que $c(p, q)$ seriam proporcionalmente menos carregados pelo mesmo conjunto de tarefas comunicantes.

O *custo* da atribuição A é medido por uma função $H(A)$, composta pela soma de uma parcela de processamento $H_P(A)$ e uma parcela de comunicação $H_C(A)$.

A componente $H_C(A)$ de $H(A)$ é dada por

$$H_C(A) = \sum_{(p,q)} CL_{(p,q)}.$$

Se a parcela de processamento, $H_P(A)$, fosse dada por um simples somatório dos custos de cada processador devido ao processamento:

$$H_P(A) = \sum_p PL_p$$

$H(A)$ seria minimizado atribuindo todas as tarefas ao processador mais rápido, pois nesse caso, $H_C(A) = 0$. Portanto esse critério seria inapropriado. Por esse motivo, em [34] a parcela $H_P(A)$ é dada por

$$H_P(A) = \sum_p \sum_{i,j|A(i)=A(j)=p} g(\sigma_i, \sigma_j)$$

onde g pode ser qualquer função que encoraje a distribuição da carga entre processadores, ou seja, deve ser positiva, e tal que $g(\sigma_i, \sigma_j) = g(\sigma_j, \sigma_i)$. Por exemplo:

$$H_P(A) = \sum_p (PL_P)^2 = \sum_p \left[\sum_{i|A(i)=p} (\frac{\sigma_i}{s_p})^2 + \sum_{i \neq j|A(i)=A(j)=p} \frac{\sigma_i \sigma_j}{{s_p}^2} \right].$$

Assim sendo, a função de avaliação de uma alocação é dada por:

$$H(A) = H_P(A) + H_C(A)$$

ou

$$H(A) = \sum_p (PL_P)^2 + \sum_{(p,q)} CL_{(p,q)}.$$

Para minimizar $H(A)$ e obter uma boa alocação A, a necessidade de diminuir $H_P(A)$ tenderá a atribuir tarefas processadores mais rápidos e distribuí-los, ao passo que a necessidade de reduzir $H_C(A)$ tenderá a concentrá-las.

Exemplo: Considere um sistema com dois processadores p e q, e as tarefas i e j. Se a atribuição A_1, aloca a tarefa i ao processador p e a tarefa j ao processador q, temos:

$$H(A_1) = (\frac{\sigma_i}{s_p})^2 + (\frac{\sigma_j}{s_p})^2 + \frac{1}{c(p,q)}(\gamma_{i,j} + \gamma_{j,i})$$

Se uma atribuição A_2, aloca ambas as tarefas ao processador p, temos:

$$H(A_2) = (\frac{\sigma_i}{s_p})^2 + (\frac{\sigma_j}{s_p})^2 + 2\frac{\sigma_i \sigma_j}{{s_p}^2}$$

Dependendo da relação entre as demandas relativas, uma alocação pode ser melhor que a outra. Por exemplo, se $s_p = s_q$, então a primeira atribuição é melhor se o custo adicional de processamento das duas tarefas no mesmo processador for maior que o custo de comunicação através do canal (p,q). Caso contrário a segunda atribuição será preferível.

O algoritmo utilizado para encontrar a alocação A de tarefas a processador, minimizando a função de custo $H(A)$ é descrito em [34] .

Capítulo VI

Usando a Linguagem OCCAM

VI.1 Propriedades da Linguagem OCCAM

A linguagem OCCAM é fundamentada em um modelo simples e seguro de concorrência. OCCAM possui uma pequena variedade de construções que nos permitem expressar, com igual facilidade, trechos de algoritmos seqüenciais, paralelos e não-determinísticos.

A simplicidade do modelo da linguagem de programação OCCAM estimula o uso da concorrência permitindo a sua expressão de forma clara e direta [36]. OCCAM não é uma linguagem seqüencial à qual adicionou-se alguma forma para expressar concorrência. A noção de processo é um conceito fundamental da linguagem OCCAM, e a comunicação entre processos é colocada no mesmo nível que a operação de atribuição.

O modelo básico para um programa OCCAM é o de uma rede de processos comunicantes. Um sistema é projetado em termos de um conjunto de processos interconectados, onde cada processo pode ser visto como uma unidade independente operando sobre suas próprias variáveis, completamente especificada pelas mensagens que ela envia e recebe, através de canais de comunicação ponto-a-ponto. Cada processo inicia, executa um certo número de ações e termina. Uma ação pode ser um conjunto de processos seqüenciais, executados um após o outro, tal como na programação convencional, ou um conjunto de processos paralelos a serem executados ao mesmo tempo. Vale dizer, um processo também pode ser projetado como um conjunto de processos interconectados. Assim sendo, a elaboração do programa é hierárquica: um programa, em seu nível mais alto é considerado como um único processo. Como um processo é também composto de outros processos, alguns dos quais podem ser executados em paralelo, cada processo pode exibir qualquer nível de concorrência interna.

Embora sendo uma linguagem de programação abstrata, o desenvolvimento da linguagem OCCAM está intimamente relacionado ao desenvolvimento

dos *transputers*. Um *transputer* é um dispositivo VLSI programável contendo quatro elos (*links*) de comunicação para conecção ponto-a-ponto. O modelo de concorrência OCCAM foi bastante influenciado pela necessidade de fornecer as mesmas técnicas de programação para um único *transputer* e para uma rede de *transputers*. Cada *transputer* implementa os conceitos OCCAM de concorrência e comunicação. Uma importante propriedade da linguagem OCCAM é que ela fornece uma noção bem clara de comportamento lógico: garante-se que o comportamento lógico de um programa não é alterado em função da distribuição dos processos entre *transputers*, ou pela velocidade de processamento ou comunicação. Conseqüentemente, mesmo que uma aplicação deva ser executada em uma rede de *transputers*, ela pode ser escrita, executada e testada em um único computador utilizado para o desenvolvimento de programas.

VI.2 Os Elementos da Linguagem

Em última análise todos os processos OCCAM são construídos a partir de três processos primitivos: atribuição, leitura e escrita em canais.

Uma atribuição computa o valor de uma expressão e dá esse valor a uma variável. Em OCCAM, ela é expressa da seguinte maneira:

```
V := e
```

onde `V` é uma variável e `e` é uma expressão do mesmo tipo que `V`.

Sempre que dois processos precisam trocar dados um elemento intermediário, ou canal, é definido. O canal é unidirecional e só pode ser usado por um processo transmissor e um processo receptor. O processo transmissor só pode conter operações de escrita no canal, o mesmo valendo para o processo receptor, que só pode realizar operações de leitura. Se a comunicação for necessária em ambas as direções entre dois processos dois canais devem ser definidos.

Os processos para ler e escrever em um canal possuem uma forma bem simples:

- para escrever em um canal c o valor da expressão e, o seguinte comando deve ser executado:

  ```
  c ! e
  ```

  onde o símbolo `!` indica saída de dado através de um canal;

- para ler desse canal, em um outro processo, para a variável V, deve-se executar:

  ```
  c ? V
  ```

onde o símbolo ? indica entrada de dado a partir de um canal.

Como a comunicação é sincronizada, o primeiro processo a executar um dos dois comandos acima é suspenso, até que o outro processo esteja pronto para completar a comunicação. O efeito final é equivalente a executar:

```
V := e
```

onde V é uma variável em um processo e e é uma expressão do mesmo tipo que V, computada por um outro processo.

Processos mais elaborados são formados combinando processos simples e processos primitivos com o auxílio de construtores do tipo:

- SEQ — execução seqüencial
- IF — execução condicional
- CASE — execução selecionada
- WHILE — execução iterativa
- PAR — execução paralela
- ALT — execução de alternativas guardadas

Processos seqüenciais são formados de maneira similar à utilizada na programação convencional, executando os processos em seqüência, condicionalmente ou iterativamente, com os quatro primeiros construtores citados acima. Por exemplo:

```
CHAN in, out:
WHILE TRUE
  VAR x:
  SEQ
    in ? x
    out ! x
```

é um processo que permanentemente recebe valores por um canal de entrada in, armazenando-os em uma variável x para copiá-los em um canal de saída out.

Processos concorrentes são formados com os construtores PAR e ALT e utilizam a comunicação entre canais. O construtor PAR é utilizado para indicar que dois ou mais processos são concorrentes e podem ser executados ao mesmo tempo. Podemos usar o construtor PAR para unir dois processos do tipo anterior através de um canal intermediário, formando um único processo para armazenar dois elementos:

```
CHAN in, out, c:
PAR
  WHILE TRUE
    VAR x:
    SEQ
      in ? x
      c ! x
  WHILE TRUE
    VAR x:
    SEQ
      c ? x
      out ! x
```

Ambos os elementos da construção paralela acima possuem uma forma similar que pode ser declarada como um procedimento:

```
PROC buf (CHAN c.in, c.out)
  WHILE TRUE
    VAR x:
    SEQ
      c.in ? x
      c.out ! x
:
```

Definir um procedimento desse tipo em OCCAM, equivale a dar um nome a um processo e especificar, como parâmetros, os canais de que o processo dispõe para se comunicar. O nome fornecido na declaração PROC é usado para criar instâncias desse processo. Uma instância de um procedimento tem o mesmo efeito que executar o processo cujo código é dado no corpo da declaração PROC:

```
CHAN in, out, c:
PAR
  buf (in, c)
  buf (c, out)
```

A construção paralela acima funciona como uma fila onde é possível guardar duas mensagens temporariamente. Para enfileirar um número maior de mensagens podemos repetir o procedimento de criação de instâncias do processo buf interligadas em seqüência como em um *pipeline*. A linguagem OCCAM oferece uma forma concisa e elegante para expressar a produção de diversas instâncias de um mesmo processo. Por exemplo, para criar um processo para o armazenamento de um número n de mensagens:

```
VAL INT n IS <mensagens a armazenar> :
[n-1] CHAN c:
PAR
    buf (in, c[0])
    PAR i = 0 FOR n-2
      buf (c[i], c[i+1])
    buf (c[n-2], out)
```

Quando a comunicação entre processos se dá por intermédio de mensagens, é necessária a existência de algum comando que permita uma escolha guardada entre diversas possíveis fontes de comunicação. Em OCCAM essa possibilidade é oferecida pela construção ALT, que tem a seguinte forma:

```
ALT
  G1
    P1
  G2
    P2
  ...
  Gn
    Pn
```

onde Gi é um guarda e Pi é um processo. Freqüentemente, os guardas combinam uma expressão booleana com uma operação de leitura de canal, podendo, no caso mais simples, conter apenas uma leitura de canal. Se houver algum dado a ser recebido e a expressão booleana (se existir) for verdadeira, diz-se que o guarda está pronto. Na execução do processo ALT, caso não exista nenhum guarda pronto, o processo ALT é suspenso, até que exista algum. Se mais de um guarda estiver pronto, apenas um é escolhido, arbitrariamente, e o processo correspondente é executado. O exemplo mais simples do uso do construtor ALT é a reunião de duas seqüências de objetos recebidos por dois canais de entrada em uma única seqüência transmitida em um canal de saída:

```
CHAN c.in1, c.in2, c.out:
WHILE TRUE
  VAR x:
  ALT
    c.in1 ? x
      c.out ! x
    c.in2 ? x
      c.out ! x
```

ao número de objetos especificados no protocolo. Por exemplo, a recepção de uma mensagem em um canal com o protocolo registros tal como dado acima, é indicada de forma um pouco mais elaborada do que as que vimos até agora:

```
CHAN OF registros le.reg:
INT tam.registro:
[max.tam.registro] BYTE reg.lido:
INT num.erro:
INT tam.msg.erro:
[256] BYTE msg.erro:
SEQ
  le.reg ? CASE
    registro ; tam.registro :: reg.lido
      <utiliza dados do registro>
    erro     ; num.erro; tam.msg.erro ::[] msg.erro
      <trata erro>
```

Dada a variedade de formatos possíveis representaremos um processo para armazenamento de uma mensagem qualquer da seguinte maneira:

```
PROC buf (CHAN OF < protocolo > in, out,
          CHAN OF INT stop)
  <declara conjunto de elementos>
  <para armazenar uma mensagem do tipo >
  <especificado em protocolo >
  INT sinal:
  BOOL ativo:
  SEQ
    ativo := TRUE
    WHILE ativo
      ALT
        in ?  <conjunto de elementos>
          out !  <conjunto de elementos>
        stop ?  sinal
          ativo := FALSE
:
```

VI.3 O Programa do Usuário

As características da linguagem OCCAM que apresentamos até agora facilitam em muito a programação de aplicações de acordo com o modelo proposto no Capítulo V. O esquema sugerido para o desenvolvimento de cada uma

como o *protocolo* do canal. O protocolo associado a um canal é fornecido como o tipo do canal na sua declaração. Por exemplo,

```
CHAN OF BYTE letras:
```

declara um canal `letras` com um protocolo do tipo BYTE, fixando que o canal transfere valores do tipo BYTE durante a comunicação. Uma saída para esse canal poderia ser:

```
letras ! 'E'
```

Os protocolos também podem receber nomes através de definições do tipo:

```
PROTOCOL caracter IS BYTE:
```

Dada a definição acima, o canal `letras` poderia ser declarado da seguinte forma:

```
CHAN OF caracter letras:
```

A definição de protocolos é bastante flexível e permite a livre elaboração de diversos formatos de mensagens. É possível especificar a transferência de tipos simples, mensagens com tamanho especificado por ocasião da comunicação e mensagens formadas por seqüências de valores com tipo determinado. A linguagem OCCAM oferece ainda a possibilidade de definir protocolos variantes, de forma a utilizar mensagens com diferentes formatos em um mesmo canal. Cada formato é precedido por uma etiqueta usada para identificá-lo, dada na declaração do protocolo, tal como no exemplo:

```
PROTOCOL registros:
  CASE
    registro ; INT ::[] BYTE
    erro     ; INT; BYTE ::[] BYTE
:
```

As entradas e saídas em um canal devem ser compatíveis com o seu protocolo associado, isto é, elas devem concordar em relação ao tipo, à forma e

A construção ALT também oferece uma forma simples de promover a finalização de processos, incluindo um canal de sinalização de término da execução:

```
PROC buf (CHAN c.in, c.out, CHAN stop)
  VAR x:
  BOOL ativo:
  SEQ
    ativo := TRUE
    WHILE ativo
      ALT
        c.in ? x
          c.out ! x
        stop ? x
          ativo := FALSE
:
```

Essa nova versão para um processo *buffer* recebe valores por um canal de entrada `c.in`, armazena-os em uma variável `x` e comunicando-os por um canal de saída `c.out`, enquanto não for recebida uma mensagem de controle através do canal `stop`. Qualquer recepção através deste último canal, causará o término da execução do processo. Se criarmos mais de uma instância do processo acima, a fim de produzir uma fila de mensagens, o sinal de terminação deverá ser enviado a cada uma das instâncias em seqüência, de maneira análoga ao esvaziamento de um *pipeline*.

```
PROC buffer (CHAN c.in, c.out, CHAN stop)
  PROC buf (CHAN c.in, c.out, CHAN stop)
    < corpo do procedimento >
  :
  VAL INT n IS <mensagens a armazenar > :
  [n-1] CHAN c:
  [n]CHAN stp:
  PAR
    buf (in, c[0], stp[0])
    PAR i = 0 FOR n-2
      buf (c[i], c[i+1], stp[i+1])
    buf (c[n-2], out, stp[n-1])
    INT sinal:
    SEQ
      stop ?  sinal
      SEQ i = 0 FOR n
        stp[i] !  sinal
:
```

Os canais constituem o único meio de transferir valores entre dois processos concorrentes. O formato e o tipo dos dados transferidos é especificado

das tarefas pode ser expresso com a construção ALT. Cada tarefa pode ser definida por um procedimento, onde a lista de parâmetros define os canais de entrada e saída da tarefa com seus respectivos protocolos:

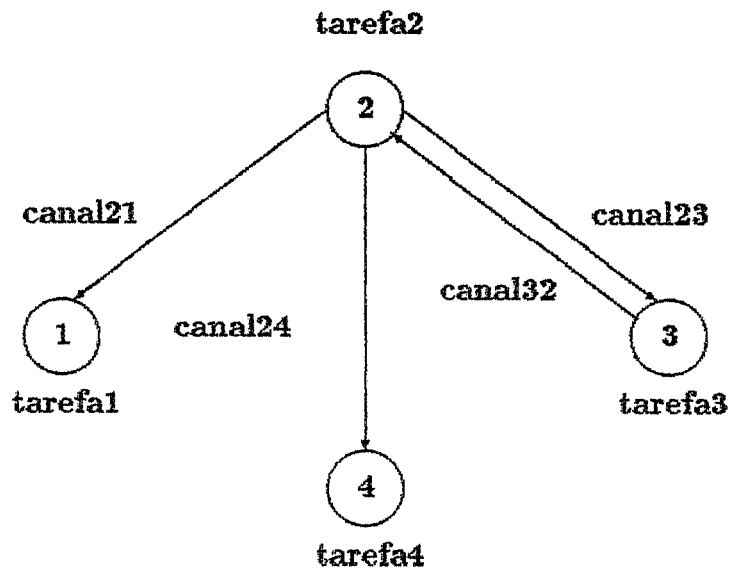
```
PROC nome.tarefa (< lista de canais>)
  < declarações locais>
  SEQ
    <inicialização>
    WHILE <condição>
      ALT
        canal.1 ? mensagem
          SEQ
            <execute alguma computação>
            <envie mensagens>
        ...
        canal.n ? mensagem
          SEQ
            <execute alguma computação>
            <envie mensagens>
:
```

O código dos procedimentos descrevendo as tarefas podem ser fornecidos em unidades de compilação separada (SC — *Separate Compilation unit*) ou bibliotecas. Para especificar quais instâncias devem ser executadas concorrentemente e suas interligações utiliza-se o construtor PAR. De forma que, em seu nível mais alto, uma aplicação bem estruturada pode ser apresentada da seguinte maneira em OCCAM:

```
<declaração de procedimentos ou inclusão de bibliotecas>
<declaração de protocolos>
<declaração de canais>
PAR
  nome.tarefa.1 (<lista de canais tarefa.1>)
  nome.tarefa.2 (<lista de canais tarefa.2>)
  ...
  nome.tarefa.n (<lista de canais tarefa.n>)
```

Em outras palavras, as instruções acima proporcionam um meio de descrever um grafo de comunicação. Cada instância indicada corresponde a um nó do grafo. Os canais declarados antes do construtor PAR são associados aos arcos, conforme eles forem referenciados nas listas de parâmetros para as instâncias.

Como cada canal representa uma ligação unidirecional entre um par de processos apenas duas instâncias podem apresentar um mesmo nome de canal




```
# USE tarefas
PROTOCOL tipo.msg IS INT:
CHAN OF tipo.msg canal21, canal23, canal32, canal24:
PAR
  tarefa1 (canal21)
  tarefa2 (canal21, canal23, canal32, canal24)
  tarefa3 (canal23, canal32)
  tarefa4 (canal24)
```

Figura VI.1: Exemplo de Grafo para a Aplicação

em suas listas. Chamaremos a uma delas de *tarefa origem* para o canal e a outra de *tarefa destino*. A direção do canal não pode ser depreendida das informações dadas nas instruções acima. Ela é estabelecida pelo código para os procedimentos, de maneira que a *tarefa origem* executará apenas operações de transmissão e a *tarefa destino* executará apenas operações de recepção de mensagens no canal em questão.

VI.4 Execução de um Programa

Um programa OCCAM com o formato descrito na seção anterior pode ser instalado para execução em um equipamento sem considerações adicionais. Nesse caso, o sistema assumirá que todas as tarefas serão mapeadas em um único *transputer*.

Para tirar proveito da multiplicidade de processadores o usuário deve efetuar uma etapa suplementar de instalação chamada configuração. Nessa etapa, de posse da arquitetura particular onde a aplicação será executada, o programador especifica uma alocação determinada para cada tarefa bem como a localização física para cada canal.

Cada *transputer* dispõe de quatro ligações (*links*) bidirecionais para interconexão em rede. Cada ligação é identificada por um endereço de entrada e outro de saída. O mapeamento de um canal OCCAM é feito relacionando o seu nome a um desses endereços através de uma declaração PLACE do tipo:

```
PLACE <nome.canal> AT <endereco.link>:
```

A construção PLACED PAR é então utilizada para indicar a execução das instâncias dos procedimentos em diferentes processadores. Cada instância é precedida por uma declaração PROCESSOR fornecendo uma identificação lógica e o tipo para o processador. Uma configuração possui portanto o seguinte formato:

```
<declaração de protocolos>
<declaração dos canais entre processadores>
PLACED PAR
  ...
  PROCESSOR < num > < tipo >
    ...
    PLACE <nome.canal> AT <endereço.link>:
    ...
    <instância para esse processador>
  ...
```

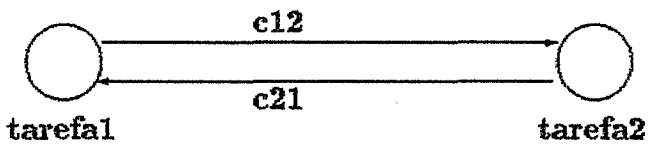


Figura VI.2: Aplicação Para o Exemplo de Configuração

Ilustraremos a seguir a criação da configuração para uma aplicação bem simples, cujo grafo mostrado na Figura VI.2, é descrito pelo código OCCAM:

```
# USE biblioteca
PROTOCOL tipo.msg IS INT:
CHAN OF tipo.msg c21, c12 :
PAR
  tarefa1 (c21, c12)
  tarefa2 (c12, c21)
```

Omitimos no trecho acima os códigos para os procedimentos das tarefas propositadamente, a fim de frisar, mais uma vez, que estes usualmente são organizados para compilação em separado, em bibliotecas ou SC's. Exemplos para ambos os procedimentos são dados a seguir:

```
PROC tarefa1 (CHAN OF tipo.msg in, out)
  INT x:
  SEQ
    out ! 1
    in ? x
:
PROC tarefa2 (CHAN OF tipo.msg in, out)
  INT x:
  SEQ
    in ? x
    out ! x + 1
:
```

O formato para a configuração da aplicação para execução em uma arquitetura tal como mostramos na Figura VI.3 é como se segue :

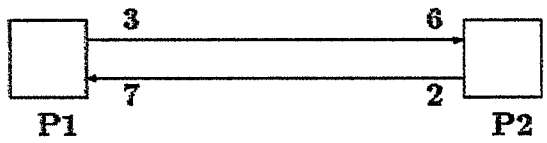


Figura VI.3: Arquitetura Para o Exemplo de Configuração

```
PROTOCOL tipo.msg IS INT:
CHAN OF tipo.msg c21, c21:
PLACED PAR
  PROCESSOR 1 T8
    PLACE c12 AT 3 :
    PLACE c21 AT 7 :
    tarefa1 (c21, c12)
  PROCESSOR 2 T8
    PLACE c12 AT 6 :
    PLACE c21 AT 2 :
    tarefa2 (c12, c21)
```

Capítulo VII

O Ambiente de Programação

VII.1 Introdução

A linguagem OCCAM descrita no capítulo anterior provê a plataforma sobre a qual montamos o Ambiente de Programação. Este Ambiente oferece, além dos benefícios da linguagem pura e simples os seguintes melhoramentos:

- comunicação assíncrona e livre de *deadlocks*
- balanceamento da carga de processamento
- programação desvinculada da arquitetura da máquina
- configuração automática

O funcionamento do Ambiente de Programação envolve basicamente o tratamento de uma aplicação OCCAM elaborada segundo o modelo proposto no capítulo VI, acompanhada de uma série de valores estimados para parâmetros tais como demanda de processamento e de comunicação e previsão de mensagens em trânsito. O resultado é um programa OCCAM pronto para execução em uma arquitetura de *transputers* (Figura VII.1).

VII.2 Funcionamento do Ambiente

A montagem do programa para a aplicação distribuída, consiste de três fases:

1. análise do programa original para a determinação do grafo de comunicação entre as tarefas e interação com o usuário para levantamento dos dados sobre a aplicação,

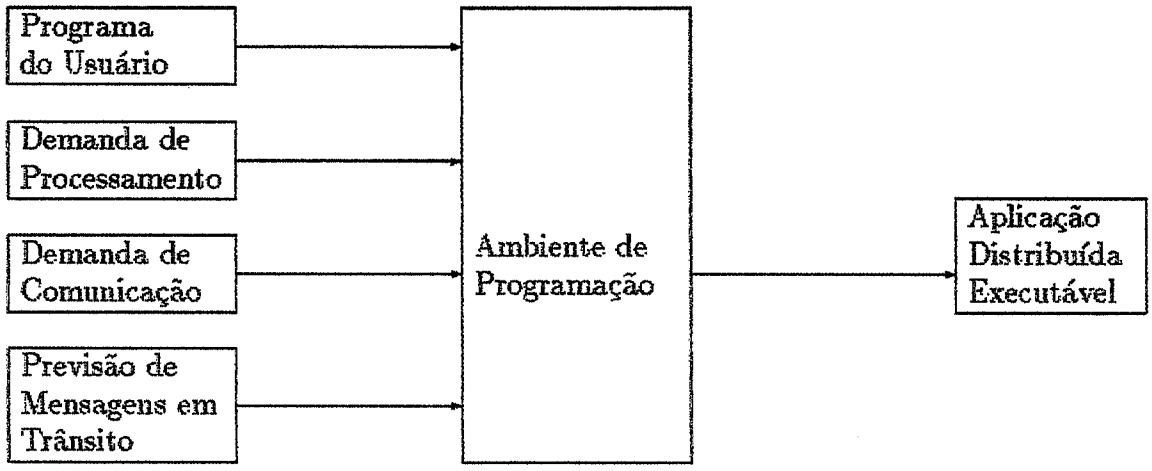


Figura VII.1: O Ambiente de Programação

2. aplicação dos algoritmos de balanceamento de carga e de alocação de espaço de armazenamento e, finalmente,

3. reconstrução da aplicação através da inclusão de elementos auxiliares que implementam a funcionalidade do Ambiente.

A interação entre as fases pode ser apreciada na Figura VII.2.

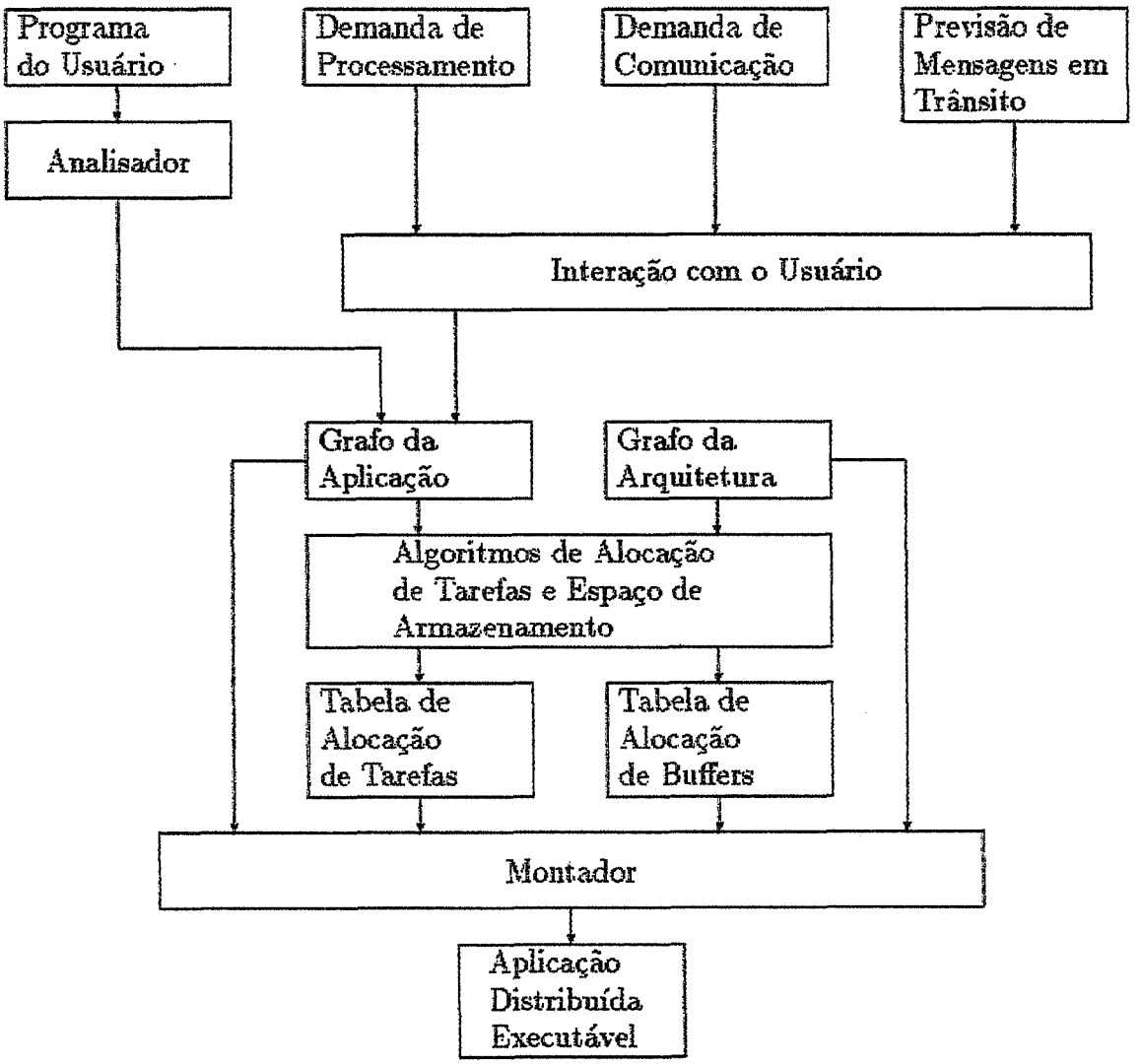


Figura VII.2: Diagrama de Funcionamento do Ambiente

O restante deste capítulo será dedicado à exposição detalhada de cada uma dessas fases.

VII.3 Programa Exemplo

Na discussão que se segue empregamos um programa exemplo para ilustrar as diversas etapas efetuadas pelo Ambiente de Programação.

O código OCCAM para descrição do grafo de comunicação entre tarefas da aplicação exemplo é dado a seguir:

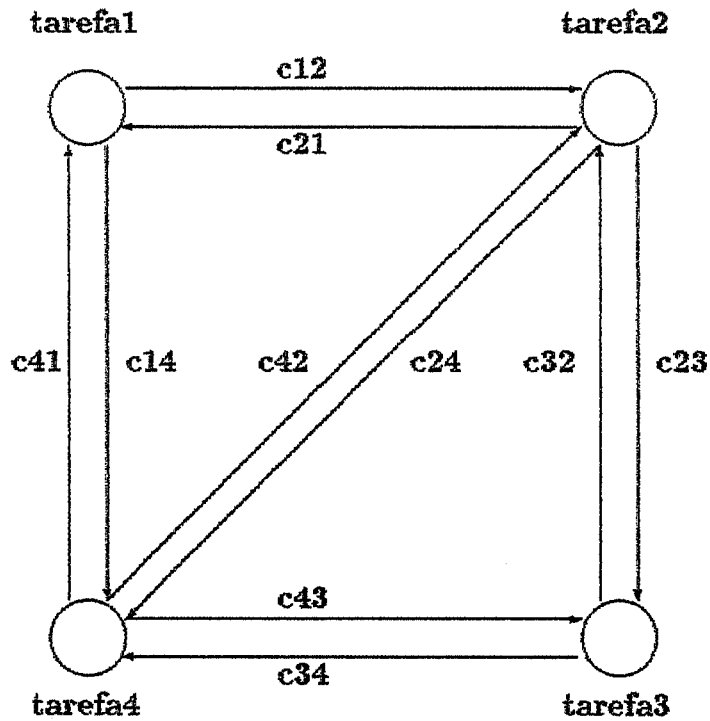


Figura VII.3: Grafo para a Aplicação Exemplo

```
# USE tarefas
PROTOCOL tipo.msg IS INT:
CHAN OF tipo.msg c12, c21, c14, c41, c23, c32, c24, c42, c34, c43 :
PAR
  tarefa1 (c21, c41, c12, c14)
  tarefa2 (c12, c32, c42, c21, c23, c24)
  tarefa3 (c23, c43, c32, c34)
  tarefa4 (c14, c24, c34, c41, c42, c43)
```

As tarefas neste exemplo se comportam conforme descrevemos para o caso especial de sincronizador α na seção IV.5.

VII.4 O Grafo da Aplicação

A primeira etapa realizada pelo Ambiente consiste em analisar o código fornecido pelo usuário a fim de montar uma estrutura de dados descritiva do grafo da aplicação. No exemplo, o grafo tem a forma ilustrada na Figura VII.3

Durante o processo de análise são incorporadas à estrutura de dados

do grafo as informações relativas aos nós (identificação das tarefas e suas respectivas listas de canais) e aos arcos (identificação dos canais e seus protocolos). Uma vez adquiridas essas informações, o Ambiente solicitará do usuário os seguintes dados suplementares:

- demanda relativa de processamento para cada tarefa,
- demanda relativa de comunicação para cada canal,
- número máximo de mensagens em trânsito em cada canal,
- orientação dos canais (identificação da tarefa origem e tarefa destino)

Esse último item é necessário em razão de não ser possível depreender o sentido de tráfego de mensagens no canal apenas pela análise do código que descreve o grafo da aplicação. Para tanto seria preciso analisar o código para o procedimento de cada uma das tarefas.

Tais dados são agregados à estrutura do grafo que se torna então uma descrição completa dos recursos a serem consumidos durante a execução.

Considerando o nosso exemplo de sincronizador α, vimos na seção IV.5 que haverá no máximo uma mensagem em trânsito em cada canal. Admitiremos também que as demandas de comunicação e de processamento de cada tarefa são idênticas para essa aplicação (valores relativos iguais a 1).

VII.5 O Grafo da Arquitetura

Todos os dados referentes ao grafo da arquitetura onde se deseja executar a aplicação são armazenados em um arquivo de configuração para o Ambiente de Programação. Para cada nó do grafo, indica-se o tipo do processador e sua velocidade. Para cada arco indica-se a velocidade de comunicação e o endereço do *link* utilizado naquela ligação.

A aplicação exemplo será executada em uma rede de *transputers* interligada da conforme mostramos na Figura VII.4. Por simplicidade, assumimos todos os processadores e canais com a mesma velocidade.

Como o Ambiente proporciona a capacidade de enviar mensagens entre qualquer par de processadores, torna-se necessário armazenar ainda a forma pela qual as mensagens são encaminhadas na rede. No nosso exemplo, a estrutura de roteamento se dá de acordo com o esquema mostrado na Tabela VII.1:

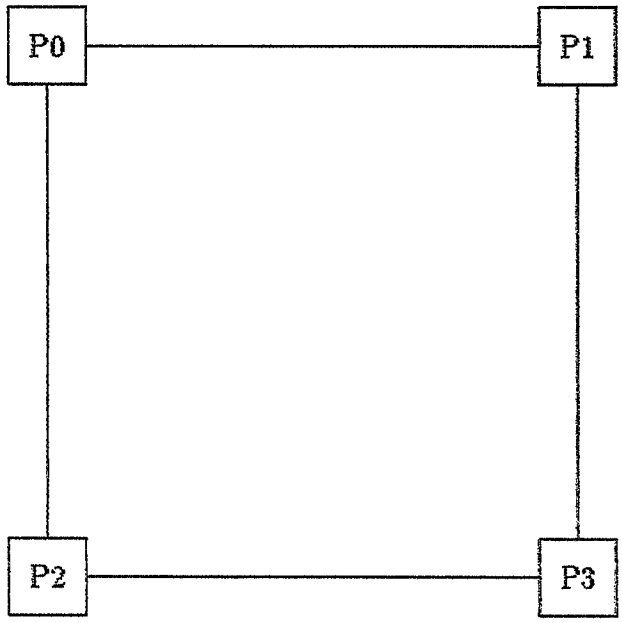


Figura VII.4: Grafo para a Arquitetura Exemplo

Processador Origem	Processador Destino	Rota
0	1	(0,1)
0	2	(0,2)
0	3	(0,2) (2,3)
1	0	(1,0)
1	2	(1,0) (0,2)
1	3	(1,3)
2	0	(2,0)
2	1	(2,0) (0,1)
2	3	(2,3)
3	0	(3,1) (1,0)
3	1	(3,1)
3	2	(3,2)

Tabela VII.1: Esquema de Roteamento

Tarefa	Processador
1	0
2	1
3	3
4	2

Tabela VII.2: Tabela de Alocação de Tarefas

VII.6 Alocação de Tarefas

A distribuição de tarefas entre os processadores é feita automaticamente para o usuário no Ambiente desenvolvido. Para garantir um bom desempenho, o algoritmo proposto em [34] e citado no capítulo V é empregado.

O algoritmo de balanceamento de carga utiliza os dados previamente armazenados nas estruturas dos grafos da aplicação e da arquitetura e gera uma *tabela de alocação de tarefas*, indicando em qual processador cada uma das tarefas deve ser executada.

Para o programa e a arquitetura dados como exemplo, a tabela de alocação de tarefas seria tal como mostra a Tabela VII.2:

VII.7 Alocação de Espaço de Armazenamento

A comunicação assíncrona proporcionada pelo Ambiente é baseada na implementação de canais com capacidades adequadas de armazenamento, que garantam a ausência de *deadlocks*. A capacidade de armazenamento em cada canal é calculada utilizando o algoritmo proposto em [33], a partir das previsões para o número máximo de mensagens em trânsito previamente fornecidas pelo programador e armazenadas no grafo da aplicação. As informações computadas por este algoritmo vão formar o que denominamos *tabela de alocação de buffers*.

Executando o algoritmo para a aplicação exemplo resulta em termos os canais `c12`, `c42` e `c43` com capacidade nula e todos os demais com capacidade para armazenar uma mensagem. A Tabela VII.3 resume essa situação:

Cada entrada na tabela corresponde à capacidade de armazenamento necessária para o canal com origem na tarefa indicada pela linha e tarefa destino indicada pela coluna.

	1	2	3	4
1	—	0	—	1
2	1	—	1	1
3	—	1	—	1
4	1	0	0	—

Tabela VII.3: Tabela de Alocação de *buffers*

VII.8 Construção da Aplicação Distribuída

A esta altura do processamento do programa original, dispomos de todas as informações necessárias para proceder à construção de um código escrito em linguagem OCCAM equivalente à configuração para a arquitetura alvo. Resta-nos apenas acrescentar os mecanismos de comunicação oferecidos pelo Ambiente, ou seja, precisamos agregar uma série de processos auxiliares que implementem as facilidades de Comunicação Assíncrona e de Roteamento de Mensagens. O texto destes processos auxiliares encontra-se em uma forma semi-acabada em um arquivo denominado biblioteca de processos auxiliares. É papel do Montador efetuar a seleção e adequação destes trechos de código para gerar, junto com as tarefas do usuário, a aplicação executável.

O programa resultante consistirá na distribuição pelos diversos processadores da arquitetura das tarefas da aplicação original (conforme especificado na tabela de alocação de tarefas), processos de armazenamento de mensagens (conforme especificados na tabela de alocação de buffers), e ainda de uma série de outros processos que implementam a comunicação entre os processadores.

VII.9 Mecanismo de Comunicação Assíncrona

Ligações de capacidade não-nula são criadas quando, ao invés de conectarmos a tarefa origem de um canal diretamente à tarefa destino, como expressa o código:

```
CHAN OF protocolo canal:
PAR
  origem(canal)
  destino(canal)
```

fizermos com que as mensagens sejam recebidas por um processo de armazenamento e então retransmitidas ao seu destino.

As tarefas origem, destino e o processo de armazenamento devem ser executados em paralelo. Por outro lado, quando as tarefas que se comunicam, agora através de um intermediário, terminarem, um sinal deve ser enviado à tarefa armazenadora de mensagens indicando o final da comunicação, e que, por conseguinte, ela deve finalizar a sua execução. Por esse motivo, o processo de armazenamento é executado em paralelo com uma construção seqüencial, para que após o término da execução paralela das tarefas origem e destino um sinal de término seja enviado ao processo de armazenamento:

```
CHAN OF protocolo canal.in, canal.out:
CHAN OF INT stop:
PAR
  buffer(canal.in,canal.out,stop)
  SEQ
    PAR
      origem(canal.in)
      destino(canal.out)
    stop ! sinal
```

O código do processo `buffer` é implementado conforme ilustramos durante a discussão sobre a linguagem OCCAM, de acordo com o protocolo do canal.

A capacidade de armazenamento de cada canal corresponde ao tamanho máximo da fila de mensagens mantida pelo processo `buffer` e é armazenada na tabela de alocação de buffers. Para a aplicação exemplo serão necessários sete processos de armazenamento, cada um deles com capacidade para uma mensagem.

Quanto à localização dos processos origem e destino, podemos ter, além da situação descrita acima, isto é, ambos instalados no mesmo processador, também o caso onde os processos são instalados em processadores diferentes. Nesse segundo caso, o processo de armazenamento será instalado no mesmo processador que a tarefa origem.

VII.10 Sistema de Comunicação

Dada uma alocação de tarefas a processadores é necessário assegurar que o grafo original de comunicação seja mantido. Portanto é necessário prover mecanismos que adeqüem a estrutura de ligação entre as tarefas, que varia conforme a aplicação, à estrutura de ligação entre processadores, que é fixa.

Tal mecanismo deve preencher as seguintes funções :

- permitir que quaisquer duas tarefas possam se comunicar usando um número

arbitrário de canais, independentemente da arquitetura ou de suas localizações nesta,

- assegurar o correto encaminhamento de mensagens entre tarefas alocadas em processadores diferentes.

Tais funções são implementadas com o auxílio dos processos auxiliares que descrevemos a seguir.

- Multiplexadores

  São processos que concentram todos os canais de transmissão de mensagens destinadas a tarefas em outros processadores da rede.

- Demultiplexadores

  São processos que concentram todos os canais de recepção de mensagens de outros processadores da rede.

- Roteadores

  São os agentes que implementam a estrutura de roteamento de mensagens preservando sua ordem de transmissão e garantindo que a comunicação entre processadores também esteja livre de *deadlocks*.

Em cada processador o Ambiente de Programação instala um módulo roteador, e opcionalmente, um módulo multiplexador e/ou um módulo demultiplexador, articulados como na Figura VII.5

VII.11 Encaminhamento de Mensagens

Mensagens trocadas entre tarefas localizadas em um mesmo processador são encaminhadas diretamente nos canais correspondentes, com ou sem o auxílio de processos de armazenamento.

Mensagens trocadas entre tarefas localizadas em processadores distintos são entregues ao Multiplexador que as organiza como uma seqüência de blocos rotulados com as informações necessárias para que estes alcancem o seu destino.

Tais blocos são passados ao Roteador que as transmite pela rede. Similarmente, o processo Roteador correspondente ao processador em que se localiza a tarefa destino coletará este fluxo de blocos entregando-os ao Demultiplexador.

Por sua vez, após examinar o rótulo de cada bloco, o Demultiplexador remontará a mensagem na sua forma original despachando-a para a tarefa destino.

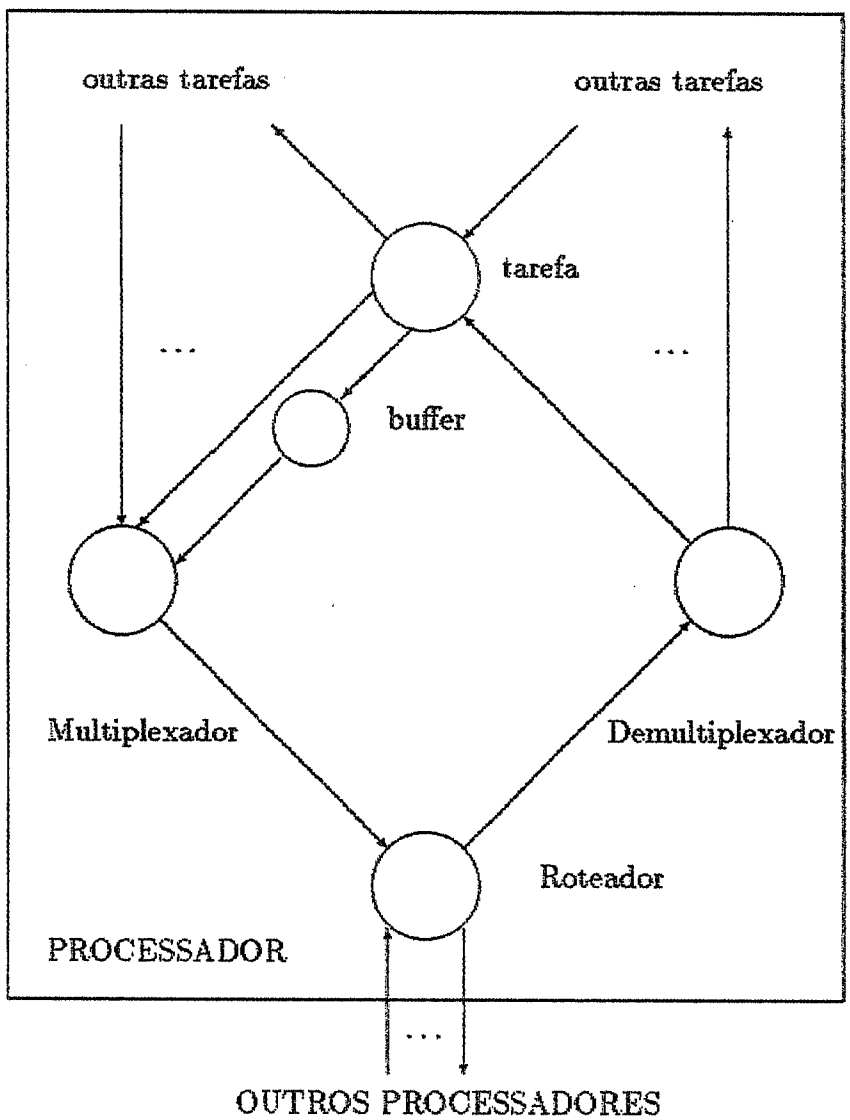


Figura VII.5: Processos Instalados em um Processador

O formato e o tamanho de cada mensagem trocada entre duas tarefas é especificado pelo protocolo associado ao canal de comunicação entre elas. Não há nenhuma limitação quanto à criação de um protocolo qualquer para a troca de mensagens entre as tarefas da aplicação. No entanto, as mensagens trocadas entre os roteadores obedecem a um protocolo fixo, definido na sua implementação. Esse protocolo determina um número máximo de elementos e um tipo para cada elemento compondo uma mensagem. Assim sendo, o processo Multiplexador, ao dividir as mensagens em blocos, adequa o seu formato original, ditado pela aplicação, para o formato utilizado durante a comunicação entre os roteadores. O mesmo acontece — de forma inversa — nos Demultiplexadores, que recuperam o formato original da mensagem antes de entregá-la à tarefa destino.

VII.12 Multiplexadores

A criação do código do Multiplexador a ser instalado num determinado processador é feita com base na relação dos canais cujas tarefas de origem se encontram neste mesmo processador e cujas tarefas destino estão localizadas em outros processadores da rede. Vale lembrar que essas informações já estão disponíveis uma vez montados o grafo da aplicação e a tabela de alocação de tarefas.

Observe também que o Multiplexador precisa ter essas informações disponíveis em tempo de execução. Assim, uma das tarefas do Montador, ao gerar o código desse processo, é incluir neste um *mapa de canais*. Esse mapa contém, para cada canal de entrada no Multiplexador, o número do processador que abriga a tarefa destino e uma identificação para a ligação entre tarefas.

O comportamento do Multiplexador, ao receber uma mensagem, pode ser descrito pelos seguintes passos:

- adequa o formato da mensagem ao utilizado pelo Roteador, arranjando-a em um ou mais blocos
- obtém no mapa de canais o número do processador de destino e a identificação para a ligação
- monta o rótulo do(s) bloco(s) com as informações extraídas do mapa
- encaminha os blocos ao Roteador

VII.13 Demultiplexadores

De maneira análoga, a criação do código do Demultiplexador a ser instalado num determinado processador é feita com base na relação dos canais cujas

MUX0			MUX1		
Canal de Entrada	Número Processador	Número Ligação	Canal de Entrada	Número Processador	Número Ligação
c12	1	1	c21	0	3
c14	2	2	c23	3	4
			c24	2	5
MUX2			MUX3		
Canal de Entrada	Número Processador	Número Ligação	Canal de Entrada	Número Processador	Número Ligação
c41	0	6	c32	1	9
c42	1	7	c34	2	10
c43	3	8			

Tabela VII.4: Mapa de Canais dos Multiplexadores

tarefas destino se encontram neste mesmo processador e cujas tarefas origem estão localizadas em outros processadores da rede.

Para o Demultiplexador, o Montador deve criar um *mapa de canais* que relaciona cada canal de saída do Demultiplexador, a uma indentificação de ligação.

O Demultiplexador executa os seguintes passos:

- recebe blocos rotulados do Roteador
- interpreta o rótulo de cada bloco como sendo uma identificação de ligação entre tarefas
- consulta no mapa de canais em qual canal de saída deve ser enviada a mensagem para a tarefa destino
- de acordo com a ligação, reúne o número de blocos necessários para recuperar o formato original da mensagem
- envia mensagem para a tarefa destino da ligação

VII.14 Roteadores

O Roteador tem basicamente três funções:

- transmitir pela rede as mensagens recebidas do Multiplexador

DUX0		DUX1	
Número Ligação	Canal de Saída	Número Ligação	Canal de Saída
3	c21	1	c14
6	c24	9	c32
		7	c42
DUX2		DUX3	
Número Ligação	Canal de Saída	Número Ligação	Canal de Saída
2	c14	4	c23
5	c24	8	c43
10	c34		

Tabela VII.5: Mapa de Canais dos Demultiplexadores

- recolher mensagens da rede que se destinam ao processador onde ele está instalado e entregá-las ao Demultiplexador
- atuar como elo de ligação retransmitindo mensagens da rede de forma que elas alcancem o seu destino

O principal componente do Roteador, responsável pela execução do algoritmo de roteamento propriamente dito, é conhecido como *Processador Virtual de Comunicação* (CVP - *Communication Virtual Processor*). Trata-se de um componente desenvolvido com a finalidade de fornecer uma série de serviços de comunicação, sendo que destes, o Ambiente faz uso de um pequeno subconjunto. Uma descrição detalhada do projeto e da implementação do CVP pode ser encontrada em [37].

Há ainda um segundo componente que possui exclusivamente a função de sincronizar o término dos CVPs ao final da execução da aplicação distribuída.

VII.15 Terminação da Aplicação

Ao término das tarefas da aplicação do usuário é necessário terminar os processos auxiliares para permitir a reutilização da máquina por uma outra aplicação.

Da mesma forma que os processos de armazenamento construímos todos os demais processos auxiliares possuindo um canal especial para recepção de um sinal de término de execução (vide seção VII.9). A construção abaixo nos permite enviar sinais de término para todos os processos auxiliares uma vez encerrado o processamento das tarefas da aplicação de um mesmo processador.

```
PAR
  < processos auxiliares >
  SEQ
    PAR
      < tarefas alocadas no processador >
    < envia sinal para os processos auxiliares >
```

Se o programador especificou corretamente o funcionamento de suas tarefas, elas não finalizarão a sua execução antes de efetuar toda a transmissão e/ou recepção de mensagens. Entretanto é possível que algumas das mensagens transmitidas pelas tarefas não tenham ainda deixado o processador. Em outras palavras tais mensagens podem ainda se encontrar em trânsito nos processos de armazenamento, no Multiplexador ou no Roteador. Uma forma de garantir o completo encaminhamento dessas mensagens é atribuir ao envio do sinal de término uma prioridade menor que a dos processos auxiliares. O código a seguir implementa esta solução:

```
PRI PAR
  PAR
    < processos auxiliares >
  SEQ
    PAR
      < tarefas alocadas no processador >
    < envia sinal para os processos auxiliares >
```

Um aspecto do problema de terminação necessita ainda ser considerado. A execução do Roteador não deve ser interrompida em função apenas da terminação das tarefas locais a um processador. Isto porque o Roteador ainda poderá estar contribuindo para o encaminhamento de outras mensagens na rede. Sua terminação deve portanto estar condicionada ao final da aplicação distribuída como um todo.

A deteção do momento em que o serviço de roteamento não é mais necessário requer que se contabilize todos os processadores que receberam o sinal de término. Esta tarefa é entregue a um dos Roteadores que mantém um contador que é incrementado à medida que todos os demais Roteadores o informa do recebimento do sinal de terminação local. Quando este contador atinge o número total de processadores, um sinal especial é difundido pela rede. Somente quando este sinal especial é recebido cada Roteador encerra sua atividade.

VII.16 Código Gerado para o Exemplo

Podemos oferecer uma noção mais exata de como os diversos trechos de código gerados pelo Ambiente são reunidos para formar a aplicação completa. Para executar uma aplicação na arquitetura exemplo (Figura VII.4), o código da configuração teria o seguinte formato geral:

```
< procedimento processador 0 >
< procedimento processador 1 >
< procedimento processador 2 >
< procedimento processador 3 >
< declaração do protocolo de comunicação entre processadores >
< declaração dos canais entre entre processadores >
PLACED PAR
  PROCESSOR 0 T8
    PLACE link10 AT 6:
    PLACE link20 AT 7:
    PLACE link01 AT 2:
    PLACE link02 AT 3:
    processador0 (0, link10, link20, link01, link02)
  PROCESSOR 1 T8
    ...
  PROCESSOR 2 T8
    ...
  PROCESSOR 3 T8
    ...
```

onde < procedimento processador i > corresponde ao texto do procedimento a ser executado no *transputer* i. Na realidade, quando carregamos a aplicação, a partir do ambiente de desenvolvimento TDS, o texto da configuração acima é dividido em dois módulos EXE e PROGRAM, conforme é explicado no manual [18]. Cada um dos procedimentos possui uma forma semelhante. Listamos abaixo o código relativo à parte da aplicação exemplo a ser instalada no processador 0.

```
PROC processador0 (VAL INT num.proc, CHAN OF INST in0, in1, out0, out1)
  < declaração dos protocolos da aplicação >
  < inclusão bibliotecas com o procedimento das tarefas >
  < inclusão bibliotecas de comunicação >
  < procedimento para o Multiplexador >
  < procedimento para o Demultiplexador >
  < procedimento para o Buffer >
  < declaração de canais >
PRI PAR
```

```
  PAR
    Buffer.c14 (c14.in, c14.out, stop.buf.c14)
    Roteador (num.proc, tabela.roteamento, to.roteador, from.roteador,
          in0,in1,out0,out1, stop.roteador)
    Multiplexador (c12, c14.out, to.roteador, stop.multiplexador)
    Demultiplexador (c21, c41, from.roteador, stop.demultiplexador)
  SEQ
    PAR
      tarefa1 (c21, c41, c12, c14.in)
    stop.buf.c14 ! 0
    stop.demultiplexador ! 0
    stop.multiplexador ! 0
    stop.roteador ! 0
:
```

Ω

Capítulo VIII

Considerações Finais

O ambiente original sobre o qual implementamos este trabalho, apesar de bastante funcional requer, por parte do programador uma considerável dose de paciência e engenhosidade para se encarregar de toda uma série de passos necessários para produzir um código executável. Mais grave ainda é a estrita dependência desse código em relação à máquina alvo. Vale dizer que um esforço suplementar deverá ser dispendido no porte da aplicação para uma outra arquitetura. O principal mérito do Ambiente de Programação é livrar o usuário dessas preocupações.

O Ambiente tal como foi implementado além de sua utilidade intrínseca provê uma base para agregar novas ferramentas tais como: simulador, depurador. Além disso, devido a sua estrutura modular, é possível experimentar com outros algoritmos de balanceamento de carga e/ou alocação de espaço de armazenamento ou mesmo substituir os diversos processos que compõem o sistema de comunicação por outros que porventura possam oferecer melhor desempenho ou funcionalidade.

As decisões de projeto tomadas durante a implementação foram influenciadas mais pelo desejo de ter um *software* modular expansível e fácil de usar. Não nos preocupamos demasiado em assegurar um desempenho ótimo.

Durante a experiência com a utilização do Ambiente por diversos programadores, observamos que a sua característica mais desconfortável refere-se à necessidade de fornecer estimativas sobre o comportamento das aplicações. Esse aspecto é tão mais negativo, quanto mais complexa a aplicação. Para eliminar a fase de entrada de dados seria preciso analisar o código completo da aplicação ou usar algoritmos mais sofisticados (isto é, que não requeiram essas informações) ou então implementar processos capazes de lidar com o balanceamento de carga, armazenamento e roteamento de mensagens de forma adaptativa ou seja, a tempo de execução.

Referências Bibliográficas

[1] AGRAWAL, D. P. e JANAKIRAM, V. K., "Evaluating the Performance of Multicomputer Configurations", *IEEE Computer*, Vol. 19, no 5, May 1986, pp 23–37, 1986.

[2] FLYNN, M. J., "Very High-Speed Computing Systems", *Proceedings of the IEEE*, Vol. 54, no 12, December 1966, pp 1901–1909, 1966.

[3] FOX, G., JOHNSON, M., LYZENGA, G., OTTO, S., SALMON, J. e WALKER, D., "Solving Problems on Concurrent Processor, Volume I, General Techniques and Regular Problems", *Prentice-Hall International, Inc.*, 1988.

[4] BURNS, A., "Programming in Occam 2", *Addison-Wesley Publishing Company*, 1988.

[5] ATHAS, C.W. e SEITZ, C.L., "Multicomputers: Message-Passing Concurrent Computers", *IEEE Computer*, August 1988, pp 9–24, 1988.

[6] "Parallel C User Guide", *3L Ltd.*, 1988.

[7] GAJSKI, D.D. E PEIR, J., "Essential Issues in Multiprocessor Systems", *IEEE Computer*, June 1985, pp 9–27, 1985.

[8] POUNTAIN, D. e MAY, D., "A tutorial introduction to occam2", *BSP Professional Books*, 1987.

[9] " Occam 2 Reference Manual", *Inmos Limited, Prentice Hall*, 1988.

[10] MAEKAWA, M., OLDEHOEFT, A.E., OLDEHOEFT, R.R., "Operating Systems, Advanced Concepts", *The Benjamim/Cummings Publishing Company*, 1988.

[11] PETERSON, J.L. e SILBERSCHATZ, A., "Operating Systems Concepts", *Addison-Wesley Publishing Company*, Second Edition, pp 349–358, 1983.

[12] DUBOIS, M. E SCHEURISCH, C., "Synchronization, Coherence and Event Ordering in Multiprocessors", *IEEE Computer*, February 1988, pp 9–21, 1988.

[13] AMORIM, C.L., BARBOSA, V.C. e FERNANDES, E.S.T., "Uma Introdução à Computação Paralela e Distribuída", *VI Escola de Computação*, Campinas, 1988, Cap. 4, 1988.

[14] WU, M. e GAJSKI, D.D., "Hypertool: A Programming Aid for Multicomputers", *Proceedings of the 1989 International Conference on Parallel Processing*, August 8–12, Vol.II, pp 15–18, 1989.

[15] SNYDER, L., "Parallel Programming and the Poker Programming Environment", *IEEE Computer*, July 1984, pp 27–36, 1984.

[16] BEMMERL, T., "An Integrated and Portable Tool Environment for Parallel Computers", *Proceedings of the 1988 International Conference on Parallel Processing*, Vol. I, pp 50–53, 1988.

[17] ERTEL, D., "Environment for Enhancing iPSC/2 Programmer Productivity", *The Third Conference on Hypercube Concurrent Computers and Applications*, Vol. I, pp 513–519, 1988.

[18] "Transputer Development System, User Guide and Reference Manual", *Inmos Limited, Prentice Hall*, 1988.

[19] GARG, V.K., "Analysis of Distributed Systems With Many Identical Processes", *The 8th International Conference on Distributed Computing Systems*, June 13–17, pp 358–365, 1988.

[20] SMITH, K.e APPELBE, W. F., "PAT — An Interactive Fortran Parallelizing Assistant Tool", *Proceedings of the 1988 International Conference on Parallel Processing*, Vol. I, pp 58–61, 1988.

[21] BUHLER, P. e WYBRANIETZ, D., "Tools for Distributed Programming in the INCAS Projetct", *Proceedings EUROMICRO 89: Design Tools for the 90's*, September 4–8, pp 199-206, 1989.

[22] MARTIN, J. "Mapping Applications to Architectures", *Proceedings of the Second International Conference on Supercomputing 87*, Vol I, pp 475, 1987.

[23] JAMIESON, L. H., "Features of Parallel Algorithms", *Proceedings of the Second International Conference on Supercomputing 87*, Vol I, pp 476–478, 1987.

[24] JAMIESON, L. H., "Characterizing Parallel Algorithms", *The Characteristics of Parallel Algorithms*, The MIT Press, 1987.

[25] SNYDER, L., "Introduction to the Configurable Highly Parallel Computer", *IEEE Computer*, January 1982, pp 47–56, 1982.

[26] CASAVANT, T.L., DIETZ, H.G., SCHWEDERSKI, P.C., SHEU, Y., SIEGEL, H.J., "Software Plans for PASM", *Proceedings of the Second International Conference on Supercomputing 87*, Vol I, pp 428–439, 1987.

[27] " Transputer Reference Manual" *Inmos Limited, Prentice Hall*, 1988.

[28] CHANDY, K.M e MISRA, J. "Parallel Program Design, a Foundation", *Addison-Wesley Publishing Company*, 1988.

[29] SCHWANN, K., BO, W., BAUMANN, N., SADAYAPPAN, P., ERCAL, F., "Mapping Parallel Applications to a Hypercube" *Proceedings of the Second Conference on Hypercube Multiprocessors*, September 1986, pp 141-151, 1986.

[30] BERMAN, F., "Experience with an Automatic Solution to the Mapping Problem", *The Characteristics of Parallel Algorithms*, The MIT Press, 1987.

[31] DONGARRA, J.J. e SORENSEN, D.C, "SCHEDULE: Tools for Developing and Analysing Parallel Fortran Programs", *The Characteristics of Parallel Algorithms*, The MIT Press, 1987.

[32] JÃNINCHEN, S. e KORDECKI, C., "The Effective Implementation of a Flexible Communication Mechanism for Distributed and Parallel Programming" *Proceedings of the Workshop on the Future Trends of Distributed Computing Systems in the 1990's*, September 1988, pp 225–281, 1988.

[33] BARBOSA, V.C, "Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs", *Technical Report*, Universidade Federal do Rio de Janeiro — COPPE/SISTEMAS, Março 1989.

[34] BARBOSA, V.C e HUANG, K.H, "Static Task Allocation in Heterogeneous Distributed Systems", *submitted to Parallel Computing*, Junho 1988.

[35] " Communicating Process Architecture", *Inmos Limited, Prentice Hall*, 1988.

[36] WELCH, P.H., "An Occam Approach to Transputer Engineering", *The Third Conference on Hypercube Concurrent Computers and Applications*, January 1988, Vol I. pp 138–147, 1988.

[37] DRUMMOND, L. e BARBOSA, V.C., "Projeto e Implementação de um Processador Virtual de Comunicação", *Dissertação de Tese de Mestrado*, COPPE-UFRJ, Agosto 1990.

Ω