UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

YALI, Uma Extensão do Modelo Linda para Programação Paralela em Redes Heterogêneas

por

ANDRÉA SCHWERTNER CHARÃO

Dissertação submetida à avaliação, como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Celso Maciel da Costa
Orientador

Prof. Cláudio Fernando Resin Geyer
Co-orientador

Porto Alegre, outubro de 1996.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Charão, Andréa Schwertner

YALI, Uma Extensão do Modelo Linda para Programação Paralela em Redes Heterogêneas / por Andréa Schwertner Charão.—Porto Alegre: CPGCC da UFRGS, 1996.

157 f.: il.

Dissertação (mestrado)—Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1996. Orientador: Costa, Celso Maciel da. Co-orientador: Geyer, Cláudio Fernando Resin

1. Ferramentas para Programação Paralela. 2. Programação Paralela. 3. Linda. 4. Heterogeneidade. 5. Sistemas Distribuídos. I. Costa, Celso Maciel da. II. Geyer, Cláudio Fernando Resin. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitor de Pós-Graduação: Prof. José Carlos Ferraz Hennemann
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. Flávio Rech Wagner
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Agradecimentos

Ao longo do desenvolvimento deste trabalho, contei com o apoio incondicional de diversas pessoas. Gostaria de agradecer, em especial:

- Ao professor Celso Maciel da Costa, pela sua orientação exemplar, pelo seu total apoio ao trabalho, pelo seu suporte técnico e científico, e pelo seu incentivo e compreensão nos momentos mais difíceis;
- Ao professor Wolfgang Pandikow, cujo entusiasmo e empenho constantes serviram como motivação em várias etapas deste trabalho;
- Ao professor Cláudio Geyer, pelo seu espírito de cooperação junto ao Grupo de Pesquisa em Processamento Paralelo e Distribuído da UFRGS;
- Ao professor João Paulo Kitajima, pelo seu interesse e disposição em discutir o trabalho;
- Ao Benhur, pelo seu incansável incentivo, pelas suas preciosas sugestões ao trabalho e pelo seu apoio constante mesmo à distância, sem os quais este trabalho ainda não estaria concluído;
- À Ritinha, colega há tanto tempo, e amiga com quem eu pude contar em todas as horas.

De maneira geral, gostaria de agradecer também aos vários colegas com os quais convivi ao longo do curso, que sempre foram uma ótima companhia dentro e fora do Instituto, e àqueles funcionários que sempre se esforçaram em facilitar nossas longas jornadas no laboratório.

Sumário

Lista de Abreviaturas 8
Lista de Figuras 10
Lista de Tabelas 11
Resumo 12
Abstract 14

1 Introdução **16**

2 O Modelo Linda **19**
2.1 Tuplas 19
2.2 Primitivas para Manipulação de Tuplas 21
2.2.1 A Primitiva OUT 21
2.2.2 A Primitiva IN 21
2.2.3 A primitiva RD 23
2.2.4 Equivalência de Tuplas e *Templates* 23
2.2.5 A Primitiva EVAL 24
2.2.6 Primitivas Não-Bloqueantes 24
2.3 Paralelismo, Comunicação e Sincronização em Linda 24
2.3.1 Expressão do Paralelismo 25
2.3.2 Comunicação 25
2.3.3 Sincronização 29
2.4 Programação Paralela em Linda 30
2.5 Limitações do Modelo 32
2.6 Sumário 33

3 Implementações do Modelo Linda **35**
3.1 Projeto do Ambiente de Programação 36
3.1.1 Estrutura das Aplicações Paralelas 36
3.1.2 Conjunto de Primitivas 36
3.1.3 Biblioteca × Pré-Compilador 37
3.1.4 Ferramentas Adicionais 38

3.2 Projeto do Ambiente de Execução 38
3.2.1 Política de Distribuição de Tuplas 38
3.2.2 Grau de Compartilhamento do Espaço de Tuplas 45
3.2.3 Implementação de EVAL 45
3.2.4 Implementação de INP e RDP 46
3.2.5 Suporte à Heterogeneidade 47
3.3 Linda em Redes: Estudo de Casos 47
3.3.1 Glenda 48
3.3.2 POSYBL 52
3.3.3 p4-Linda 57
3.3.4 Network Linda 61
3.4 Sumário 68

4 O Ambiente YALI **72**
4.1 Objetivos Preliminares 72
4.1.1 Facilidade de Aprendizado e Uso 72
4.1.2 Heterogeneidade 73
4.1.3 Desempenho 73
4.2 Ambiente de Programação 74
4.2.1 Estrutura das Aplicações 74
4.2.2 Conjunto de Primitivas 75
4.2.3 Componentes do Sistema 89
4.3 Ambiente de Execução 93
4.3.1 Política de Distribuição de Tuplas 93
4.3.2 Grau de Compartilhamento do Espaço de Tuplas 94
4.3.3 Implementação de EVAL 94
4.3.4 Implementação de INP e RDP 95
4.3.5 Suporte à Heterogeneidade 95
4.4 Sumário 96

5 Implementação de YALI **98**
5.1 Plataformas de Implementação 98
5.2 Estrutura Geral do Ambiente de Execução 99

5.3 Estrutura de Armazenamento de Dados 101
5.3.1 Espaço de Tuplas Local 101
5.3.2 Tabela de Requisições de Tupla Pendentes 102
5.3.3 Tabela de Barreiras 103
5.3.4 Tabela de Funções Globais 103
5.3.5 Tabela de Processos 104
5.4 Programação *Multithreaded* 104
5.4.1 Portabilidade 104
5.4.2 Aspectos de Programação 105
5.5 Mecanismos de Comunicação entre Processos 106
5.6 Suporte à Heterogeneidade 109
5.7 Implementação das Primitivas 111
5.7.1 Considerações Gerais 111
5.7.2 Primitivas de Manipulação de Tuplas 114
5.7.3 Operações Globais 117
5.7.4 Primitivas para Expressão do Paralelismo 120
5.7.5 Outras Primitivas 124
5.8 Inicialização e Término de Aplicações 124
5.8.1 Criação de Processos 125
5.8.2 Inicialização Individual de Processos 125
5.8.3 Procedimento de Término de Aplicações 126
5.9 Implementação do Pré-Processador 126
5.10 Sumário 127

6 Avaliação do Sistema **130**
6.1 Comparação com os Sistemas Estudados 130
6.2 Expressividade do Sistema 130
6.2.1 Iteração de Jacobi 131
6.2.2 Cálculo de π em Paralelo 135
6.2.3 Determinação de Números Primos em um Intervalo 137
6.3 Multiplicação de Matrizes 142
6.4 Desempenho 143

6.4.1 Ping-Pong 143
6.4.2 Geração de Fractais 145

7 Conclusão 147
7.1 Avaliação Geral do Trabalho 147
7.2 Trabalhos Futuros 148

Bibliografia 151

Lista de Abreviaturas

ASCII	American Standard Code for Information Interchange
BSD	Berkeley Standard Distribution
DEC	Digital Equipment Corporation
FTP	File Transfer Protocol
IBM	International Business Machines
IEEE	Institute of Electrical and Electronic Engineers
MPMD	Multiple-Program Multiple-Data
NFS	Network File System
POSIX	Portable Open System Interface eXchange
POSYBL	PrOgramming SYstem for distriButed appLications
PVM	Parallel Virtual Machine
RPC	Remote Procedure Call
SPMD	Single-Program Multiple-Data
TCP	Transmission Control Protocol
TLI	Transport Level Interface
TP	Thread Processadora
TR	Thread Receptora
TS	Tuple Space
TU	Thread Usuária
UDP	User Datagram Protocol
XDR	External Data Representation
YALI	Yet Another Linda Implementation

Lista de Figuras

Figura 2.1 - Comunicação entre processos em Linda. 25

Figura 2.2 - Comunicação entre processos que executam em tempos e locais diferentes. 26

Figura 3.1 - Alternativas para uma política de distribuição uniforme. . . . 41

Figura 3.2 - Trecho de programa em p4-Linda. 59

Figura 3.3 - Manipulação de tuplas em Network-Linda. 63

Figura 3.4 - Programa em Network-Linda para cálculo de π em paralelo. 64

Figura 3.5 - Implementação de **in** e **out** em Network-Linda. 67

Figura 3.6 - Protocolo Network-Linda para manipulação de tuplas grandes. 68

Figura 5.1 - Processo YALI. 100

Figura 5.2 - Aplicações YALI executando simultaneamente. 101

Figura 5.3 - Estrutura de um Espaço de Tuplas Local. 102

Figura 5.4 - Estrutura de uma Tabela de Requisições de Tupla Pendentes. 103

Figura 5.5 - Estrutura de uma Tabela de Barreiras. 103

Figura 5.6 - Estrutura de uma Tabela de Funções Globais. 104

Figura 5.7 - Situação de *deadlock* entre duas *threads*. 106

Figura 5.8 - Conjunto de sockets de um processo YALI. 110

Figura 5.9 - Processamento de primitivas baseadas em *hashing*. 113

Figura 5.10 - Exemplo de implementação da primitiva **y_out**. 115

Figura 5.11 - Exemplo de implementação das primitivas **y_in/y_rd**. 116

Figura 5.12 - Exemplo de implementação das primitivas **y_gather/y_reduce**. 119

Figura 5.13 - Exemplo de implementação da primitiva **y_barrier**. 121

Figura 5.14 - Implementação de barreiras através de uma combinação de primitivas. 122

Figura 5.15 - Exemplo de implementação de **y_global** e **y_globeval**. 124

Figura 5.16 - Código fonte YALI. 127

Figura 5.17 - Código fonte gerado pelo pré-processador. 128

Figura 6.1 - Código do processo mestre para implementação do algoritmo de Jacobi. 133

Figura 6.2 - Código dos processos trabalhadores para implementação do algoritmo de Jacobi. 134

Figura 6.3 - Código otimizado para implementação do algoritmo de Jacobi (processo mestre). 136

Figura 6.4 - Código otimizado para implementação do algoritmo de Jacobi (processos trabalhadores). 137

Figura 6.5 - Versão seqüencial do programa para cálculo de π. 138

Figura 6.6 - Programa YALI para cálculo de π em paralelo. 139

Figura 6.7 - Programa YALI para o cálculo da peneira de Eratóstenes (principal). 140

Figura 6.8 - Programa YALI para o cálculo da peneira de Eratóstenes (funções auxiliares) 141

Figura 6.9 - Processo mestre para multiplicação de matrizes em paralelo . 143

Figura 6.10 - Processo trabalhador para multiplicação de matrizes em paralelo 144

Figura 6.11 - *Speedup* para a implementação do algoritmo de Mandelbrot. 146

Lista de Tabelas

Tabela 3.1 - Resumo das características do ambiente de programação de cada sistema estudado. 70

Tabela 3.2 - Resumo das características do ambiente de execução de cada sistema estudado. 71

Tabela 4.1 - Funções pré-definidas para uso em **y_reduce**. 87

Tabela 6.1 - Comparação entre os ambientes de programação de YALI e dos demais sistemas estudados. 131

Tabela 6.2 - Comparação entre os ambientes de execução de YALI e dos demais sistemas estudados. 131

Tabela 6.3 - Tempo médio de uma iteração Ping-Pong em YALI. 145

Tabela 6.4 - Tempo médio de uma iteração Ping-Pong em POSYBL. 145

Resumo

Com a disponibilidade de redes que ligam estações cada vez mais poderosas a baixos custos, o interesse em torno de ferramentas que suportam a programação paralela em arquiteturas deste tipo tem aumentado significativamente. Esta dissertação trata do projeto e implementação de YALI (*Yet Another Linda Implementation*), uma ferramenta destinada ao desenvolvimento e execução de programas paralelos em redes heterogêneas de computadores.

Com o objetivo de oferecer uma interface simples e flexível para os usuários programadores, YALI baseia-se no modelo Linda[GEL85], que destaca-se por utilizar uma abstração de alto nível para a cooperação entre processos. Em Linda, processos interagem por intermédio de uma memória associativa logicamente compartilhada, denominada Espaço de Tuplas. Entre outras vantagens deste modelo pode-se citar a simplicidade de suas primitivas e a possibilidade de incorporá-las a uma linguagem seqüencial conhecida, o que contribui fortemente para sua fácil assimilação, mesmo por usuários com pouca experiência em programação paralela.

Após uma descrição detalhada do modelo Linda, este trabalho discute várias questões envolvidas no projeto e implementação de sistemas nele baseados. Para oferecer uma visão prática das soluções mais freqüentemente adotadas para estas questões, quatro sistemas que implementam o modelo para programação paralela em redes são apresentados e avaliados. São eles: Glenda, uma implementação do modelo baseada na ferramenta PVM (*Parallel Virtual Machine*); POSYBL (*PrOgramming SYstem for distriButed appLications*), um sistema construído através de recursos de sistemas operacionais compatíveis com Unix; p4-Linda, construído a partir da ferramenta de programação paralela p4 e, por fim, Network-Linda, uma implementação comercial do modelo.

Depois do estudo dos quatro sistemas acima, o projeto de YALI é discutido detalhadamente. Decidiu-se, inicialmente, que YALI deveria incorporar o modelo Linda à linguagem C, que é largamente utilizada no desenvolvimento de programas de propósito geral. Além disso, optou-se por estender o modelo com algumas novas primitivas, de modo a oferecer maior poder de expressão ao usuário. Basicamente, as primitivas que YALI acrescenta ao modelo servem para dar suporte a operações globais e à criação dinâmica de *threads*. Operações globais servem para expressar a comunicação e a sincronização entre múltiplos processos, sendo utilizadas com bastante freqüência em vários tipos de programas paralelos. YALI suporta operações globais de maneira totalmente ortogonal ao modelo Linda, garantindo melhor desempenho sem afetar o nível de abstração oferecido. O suporte a criação dinâmica de *threads*, por outro lado, tem o objetivo de permitir a exploração de um paralelismo de granularidade fina, adequado até mesmo à execução de rotinas simples em paralelo.

Para suportar o desenvolvimento e execução de aplicações paralelas, YALI é implementado através de três componentes distintos. O primeiro é um pré-processador, que garante uma interface simplificada com o usuário. O segundo é uma biblioteca, que contém as rotinas de suporte às primitivas YALI e deve

ser ligada aos programas de usuários. O terceiro componente, por fim, é um utilitário destinado a controlar a inicialização e o término de aplicações paralelas, que baseia-se em uma configuração estabelecida pelo usuário para distribuir processos sobre uma rede de computadores.

Ao contrário da maioria dos sistemas baseados em Linda, YALI implementa um espaço de tuplas distribuído entre os processos que compõem uma aplicação paralela, dispensando o uso de processos especializados no gerenciamento de tuplas. Para isso, YALI utiliza múltiplas *threads* em cada processo definido pelo usuário, e distribui tuplas sobre estes processos através de um mecanismo baseado em *hashing*. A implementação de YALI leva em conta a heterogeneidade inerente a ambientes de rede, permitindo que máquinas com diferentes arquiteturas e sistemas operacionais sejam utilizadas na execução de programas paralelos. Por fim, YALI é totalmente implementado a partir de recursos presentes em sistemas compatíveis com Unix, de modo a aumentar sua portabilidade e garantir sua eficiência.

Palavras-Chave: programação paralela, Linda, heterogeneidade, sistemas distribuídos.

TITLE: "YALI, AN EXTENSION TO THE LINDA MODEL INTENDED FOR PARALLEL PROGRAMMING IN HETEROGENEOUS COMPUTER NETWORKS"

Abstract

With the availability of networks connecting powerful workstations at a low cost, increasing interest has been devoted to systems that support parallel programming in such architectures. This document describes the design and implementation of YALI (*Yet Another Linda Implementation*), a tool that allows the development and execution of parallel programs in heterogeneous computer networks.

Aiming to provide a simple and flexible interface for its users, YALI is based on the Linda parallel programming model[GEL85], that outstands in providing a high level abstraction for cooperation between processes. In Linda, communication and synchronization take place through an associative, logically shared memory called Tuple Space. Among the advantages of this model, one can mention the simplicity of its primitives, and the possibility of incorporate them in a well-known sequential language. These characteristics make Linda easy to learn, even to users with little experience in parallel programming.

After a detailed description of the Linda model, this document discusses some design and implementation issues related to Linda-based systems. In order to provide a practical view of some usual solutions to address these issues, four Linda-based systems are presented and evaluated. These systems are: Glenda, an implementation of Linda built on top of PVM (*Parallel Virtual Machine*); POSYBL (*PrOgramming SYstem for distriButed appLications*), that relies on features provided by Unix-like operating systems to implement the model; p4-Linda, built on top of p4 parallel programming tool and, at last, Network-Linda, a comercial product based on Linda. All these systems, as YALI, are specially tailored to parallel programming in computer networks.

Following the study of the four systems, this documents presents the design of the YALI system. One of the first design decisions was to incorporate the Linda primitives to the C language, that is broadly used as a general purpose programming language. In addition, a set of new primitives was designed as an extension to the original model, in order to increase YALI's expressivenes. Basically, the new primitives support global operations and dynamic thread creation. Global operations are useful to express communication and synchronization among multiple processes, and are frequently used many classes of parallel programs. YALI gives support to global operations in a way that is totally ortoghonal to the Linda model, ensuring better performance without affecting the abstraction level inherent to Linda-based systems. The support to dynamic thread creation, on the other hand, is helpful to explore lightweight parallelism, which allows the execution of simple routines in parallel.

To support the development and execution of parallel applications, YALI is made up of three distinct components. The first is a pre-processor, that provides a simple user interface. The second is a library, that must be linked to the user

programs since it's where YALI primitives are actually implemented. Finally, the third component is an utility that controls initialization and termination of parallel applications, which takes configuration parameters from the user to distribute processes over a newtork.

In contrast with most Linda-based systems, YALI relies on a tuple space that is distributed among the processes in the same parallel application, so that intermediate tuple managers are not necessary. To implement that, multiple threads are embedded in each user process, and tuples are spread over the processes in the basis of a hashing mechanism. YALI's implementation takes in account the inherent heterogeneity of network environments, allowing machines with different architectures and operating systems to be used in the execution of parallel programs. Finally, YALI is build on top of common features of Unix-like operating systems, in order to increase its efficiency and portability.

Keywords: parallel programming, Linda, heterogeneity, distributed systems.

1 Introdução

O alto crescimento na utilização de redes que interligam vários tipos de computadores, aliado à disponibilidade de tecnologias de rede de alta velocidade, tem motivado cada vez mais o uso de redes de computadores como uma alternativa para explorar-se as vantagens do paralelismo a custos relativamente baixos. Esta possibilidade, no entanto, só é realmente efetivada com o uso de ferramentas de *software* construídas especialmente para suportar o desenvolvimento e execução de programas paralelos em plataformas distribuídas.

Um programa paralelo consiste em múltiplos processos que executam simultaneamente e interagem quando necessário. Num ambiente de rede, esta interação entre processos ocorre fisicamente através de mensagens, devido à ausência de memória compartilhada entre as máquinas interligadas. Grande parte das ferramentas para programação paralela disponíveis atualmente apóia-se em um paradigma de cooperação entre processos baseado em troca de mensagens, que reflete claramente a organização física inerente a um ambiente de rede. Apesar de eficiente, este paradigma é de baixo nível de abstração, fazendo com que a construção de programas paralelos se torne uma tarefa bem mais trabalhosa do que o desenvolvimento de programas seqüenciais.

Uma abordagem alternativa para a cooperação entre processos é o uso de um paradigma baseado em compartilhamento lógico de memória, que tenta esconder do programador a distribuição característica de um ambiente de rede, diminuindo algumas das dificuldades envolvidas na programação paralela[BAL90]. Dentro desta abordagem de mais alto nível, uma solução que se destaca é aquela proposta pelo modelo Linda[GEL85], que utiliza a abstração de uma memória compartilhada associativa denominada espaço de tuplas. Em Linda, toda interação entre processos acontece por intermédio deste espaço de tuplas, através de um pequeno conjunto de primitivas que manipulam os elementos básicos de armazenamento desta memória: as tuplas. Estas primitivas podem ser adicionadas a uma linguagem seqüencial já existente, tornando-a adequada para a programação paralela.

O alto nível de abstração de Linda, a simplicidade de suas primitivas, e a possibilidade de incorporá-las à uma linguagem convencional são características que contribuem fortemente para que o modelo seja de fácil assimilação e utilização, mesmo por usuários com pouca experiência no desenvolvimento de programas paralelos. Apesar disso, o modelo é suficientemente flexível para expressar diversos padrões de interação entre processos, sendo atualmente utilizado em aplicações de diversas áreas, como análise financeira, projeto de dispositivos eletrônicos, pesquisa farmacêutica, *ray-tracing* e aplicações ligadas à exploração de petróleo[CAR94, CAG93].

Pelas vantagens que Linda oferece no sentido de facilitar a programação paralela, este modelo foi escolhido como base para o projeto de YALI (*Yet Another Linda Implementation*), uma ferramenta destinada a suportar o desenvolvimento

e execução de programas paralelos em redes de computadores[1]. O trabalho em torno desta ferramenta tem como um dos seus principais objetivos a construção de uma interface de programação simples e flexível, que possa ser usada tanto por programadores iniciantes na área de processamento paralelo como por programadores experientes que buscam maior desempenho para suas aplicações.

Uma importante contribuição deste trabalho é que decidiu-se não apenas projetar e implementar um sistema baseado em Linda, mas também incorporar algumas extensões ao modelo. Com algumas novas primitivas, que dão suporte a operações globais e à criação dinâmica de *threads*, espera-se oferecer maior poder de expressão ao usuário e, ao mesmo tempo, tornar mais eficiente a interação entre múltiplos processos e a expressão do paralelismo em YALI.

Também de grande importância neste trabalho é a preocupação com a heterogeneidade, que é uma característica cada vez mais marcante em ambientes de rede atuais. Levando-se em conta esta realidade heterogênea, decidiu-se oferecer ao usuário a possibilidade de utilizar máquinas de arquiteturas diferentes na execução cooperativa de programas paralelos, de modo a aumentar a flexibilidade da ferramenta e, também, permitir um melhor aproveitamento dos recursos computacionais disponíveis em cada ambiente de rede.

O projeto de uma ferramenta com as características descritas acima envolve a busca de soluções para diversas questões, requerendo não somente conhecimentos na área de paralelismo e sistemas distribuídos, mas também várias noções em outras áreas, como arquitetura e redes de computadores. A implementação da ferramenta também é crítica, exigindo o uso de técnicas avançadas de programação para lidar com questões de distribuição e de desempenho. Desta maneira, também é objetivo deste trabalho adquirir experiência e consolidar alguns conhecimentos já adquiridos, através da busca de soluções eficientes para as questões envolvidas no projeto e implementação de YALI.

Este texto, que tem o objetivo de descrever as várias etapas do trabalho realizado, está estruturado da seguinte maneira:

- o segundo capítulo descreve o modelo Linda em maior detalhe, salientando suas vantagens para a programação paralela e também identificando algumas limitações do modelo, que motivaram as extensões incluídas em YALI;
- o terceiro capítulo trata de diversas questões relacionadas ao projeto de sistemas baseados em Linda, e inclui um estudo de casos sobre sistemas que implementam este modelo para programação em redes de computadores;
- o quarto capítulo apresenta o projeto de YALI, descrevendo sua interface, seus componentes, e as soluções adotadas para várias questões de projeto;

[1] Embora a programação paralela em redes de computadores seja comumente referenciada como "programação distribuída", o paradigma adotado em YALI tem o propósito de tornar transparente esta distribuição, de modo que, ao longo deste texto, usa-se somente o termo "programação paralela" para referenciar qualquer técnica de programação envolvendo múltiplos processos executando simultaneamente.

- o quinto capítulo descreve a implementação da ferramenta, salientando algumas particularidades importantes na construção de YALI;
- o sexto capítulo faz uma avaliação de YALI, comparando o sistema com outras ferramentas e incluindo alguns dados de desempenho do protótipo implementado;
- finalmente, o sétimo capítulo é destinado à conclusão, fornecendo uma visão dos aspectos importantes do trabalho e apresentando algumas sugestões de trabalhos futuros.

2 O Modelo Linda

O modelo Linda[GEL85] consiste em um pequeno conjunto de primitivas que permitem estender uma linguagem seqüencial, tornando-a adequada para programação paralela. O paradigma de programação suportado por este modelo baseia-se em uma memória global associativa denominada **espaço de tuplas** (*Tuple Space* — TS). Esta memória é compartilhada entre os processos que compõem uma aplicação paralela, e sua unidade de armazenamento são seqüências de dados chamadas **tuplas**. Em Linda, processos interagem por intermédio do espaço de tuplas, utilizando primitivas que servem, basicamente, para inserir ou recuperar tuplas do TS.

Este capítulo trata de vários aspectos ligados à semântica do modelo Linda. Inicialmente, a seção 2.1 apresenta uma descrição detalhada das tuplas, que são elementos fundamentais do modelo. Logo após, na seção 2.2, são apresentadas as primitivas oferecidas em Linda para manipulação de tuplas. Como qualquer linguagem para programação paralela deve oferecer meios de expressar o paralelismo, a comunicação e a sincronização entre processos, a seção 2.3 apresenta uma abordagem do modelo Linda com ênfase nestes três aspectos. A seguir, a seção 2.4 trata da estruturação de aplicações paralelas em Linda e, por fim, a seção 2.5 discute algumas limitações que têm sido atribuídas ao modelo.

2.1 Tuplas

Tuplas são formadas por seqüências de dados de tipos bem definidos. Os tipos de dados permitidos são, em geral, aqueles suportados pela própria linguagem seqüencial hospedeira de Linda. Tuplas não são referenciadas por endereços – como ocorre com variáveis em uma memória convencional – mas sim associativamente, isto é, pelo seu próprio conteúdo. A especificação de uma tupla se dá pela determinação do **valor** ou do **tipo** de cada um de seus campos de dados. Campos que possuem um valor associado são ditos **reais**, e servem como uma chave estruturada de identificação da tupla. Campos sem valores associados, por sua vez, são ditos **formais**, e geralmente funcionam como recipientes de dados. Para exemplificar a composição de tuplas em Linda, considere a definição das seguintes variáveis na linguagem C:

```
int  x = 10;
int  y = 20;
```

Exemplos típicos de tuplas são aquelas formadas por uma chave alfanumérica seguida de zero ou mais valores, como em:

```
("mutex"),
("pi", 3.1416) e
("index", x, y).
```

A primeira tupla é composta por apenas um campo real: a cadeia de caracteres `"mutex"`. No segundo exemplo tem-se uma tupla com dois campos reais: uma cadeia de caracteres (`"pi"`) e um número de ponto flutuante (3.1416). O terceiro exemplo mostra uma tupla com três campos, sendo o primeiro uma cadeia de caracteres (`"index"`), e os restantes dois números inteiros (10 e 20). Nesta tupla, os valores dos dois últimos campos são dados, respectivamente, pelas variáveis x e y. Pode-se notar, pelos exemplos fornecidos, que o tipo e o valor associados a um campo real são iguais ao tipo e ao valor da constante ou variável usada para representá-lo. Embora seja comum utilizar cadeias alfanuméricas no primeiro campo de uma tupla, é possível usar chaves de diferentes tipos, como nos exemplos abaixo:

```
(1002, "Mr. Jones")
('A', 1, 2, 5166)
```

Acima, os primeiros campos das tuplas apresentadas são, respectivamente, um número inteiro (1002) e um caracter (`'A'`). Tuplas também podem ser compostas por campos formais. Para exemplificar isso, considere a definição das seguintes variáveis:

```
int z;
int w;
```

As tuplas a seguir utilizam estas variáveis como campos formais:

```
("index", ?z, ?w)
('A', 1, ?w, ?z)
```

Acima, o símbolo **'?'** é usado para indicar que um determinado campo é formal, isto é, não possui valor associado. Campos formais são representados por variáveis definidas na linguagem hospedeira de Linda. O tipo de um campo formal é sempre igual ao tipo da variável que o representa. Nas tuplas acima, as variáveis w e z representam campos formais do tipo inteiro.

Tuplas podem compor **estruturas de dados distribuídas** dentro do TS. Uma estrutura de dados distribuída tem a vantagem de poder ser manipulada simultaneamente por vários processos paralelos[AHU86]. Por exemplo, uma matriz distribuída pode ser representada no TS por tuplas da forma

(<nome da matriz>, <numero linha>, <numero coluna>, <valor>)

Assim, a tupla

```
("A", 1, 2, 4.5)
```

armazena o elemento da primeira linha e da segunda coluna da matriz "A". Outra forma de organizar uma matriz distribuída é agrupar linhas ou colunas inteiras em uma tupla, como a seguir:

```
("A", 1, <elementos da primeira linha de A>)
("A", 2, <elementos da segunda coluna de A>)
```

Outras estruturas de dados usuais também podem ser representadas de forma distribuída através de tuplas. Uma fila, por exemplo, pode ter cada um de seus elementos armazenados em tuplas no TS. Para garantir a ordem de recuperação dos elementos, cada um deve ter um número de seqüência associado, e tuplas auxiliares podem ser utilizadas para indicar o início e o fim da fila, como mostra o exemplo abaixo:

```
("fim", <numero de sequencia>)
("inicio", <numero de sequencia>)
("elemento", <numero de sequencia>, <valor>)
```

Como elementos devem ser adicionados no final da fila, o número de seqüência de um novo elemento deve ser aquele contido na tupla `"fim"`. Por outro lado, elementos são sempre retirados do início da fila, e por isso o número de seqüência do elemento a recuperar deve ser aquele dado pela tupla `"inicio"`. Conforme são adicionados ou removidos elementos da fila, os números de seqüência contidos nas tuplas `"fim"` e `"inicio"` devem ser devidamente atualizados.

2.2 Primitivas para Manipulação de Tuplas

Existem quatro operações definidas em Linda para manipulação de tuplas: **out**, **in**, **rd** e **eval**[CAR89, AHU86, GEL85]. Tais primitivas são sempre processadas atomicamente. O efeito de várias operações simultâneas executadas sobre a mesma tupla é igual aquele obtido com a execução seqüencial das operações[BAL89]. Do ponto de vista de sua utilização, as primitivas podem ser vistas como chamadas de procedimentos que recebem como parâmetro os campos da tupla a ser manipulada.

2.2.1 A Primitiva OUT

Esta primitiva serve para inserir uma tupla no TS. A execução de

out(`"VET"`, 1, 3.5)

causa a inserção da tupla `("VET", 1, 3.5)` no TS. A primitiva **out** é assíncrona, isto é, o processo que a executa continua mesmo que a tupla ainda não tenha sido de fato depositada no TS. É possível incluir no TS várias cópias da mesma tupla. Toda tupla inserida permanece no TS até ser explicitamente removida. Geralmente, tuplas fornecidas como parâmetro para **out** são compostas por campos reais. No entanto, o modelo também permite que campos formais apareçam numa chamada **out**.

2.2.2 A Primitiva IN

A função desta primitiva é remover uma tupla do TS. A tupla fornecida como argumento a **in** é chamada de *template*, e pode conter tanto campos reais como

formais. Os campos reais do *template* formam uma chave para busca da tupla, enquanto os campos formais são variáveis que recebem algum valor quando uma tupla equivalente ao *template* é encontrada e retirada do TS. As regras de equivalência entre tuplas e *templates* serão detalhadas na seção 2.2.4 mas, basicamente, para que uma tupla satisfaça um determinado *template*, seus campos devem ser equivalentes em número e tipo, e seus campos reais devem apresentar os mesmos valores.

Como exemplo, considere as variáveis e a operação abaixo:

```
int     i;
float   x;

in("VET", ?i, ?x)
```

Esta operação **in** deve procurar no TS uma tupla com três campos, sendo o primeiro a cadeia de caracteres "VET", o segundo um número inteiro e o último um número de ponto flutuante. A tupla ("VET", 1, 3.5), inserida no TS pela operação **out** da seção anterior, poderia satisfazer esta chamada **in** e, neste caso, as variáveis i e x receberiam, respectivamente, os valores 1 e 3.5. Quando nenhuma tupla equivalente é encontrada no TS, o processo que executa **in** é bloqueado até que seja depositada alguma tupla que satisfaça a operação. É possível que várias tuplas satisfaçam um mesmo *template* mas, nesta situação, somente uma delas é escolhida aleatoriamente para remoção. Por exemplo, se existissem no TS as tuplas

```
("VET", 2, 4.5)
e
("VET", 3, 5.5)
```

qualquer uma delas poderia ter sido escolhida para satisfazer a operação **in** acima. No entanto, se a operação a ser realizada fosse, por exemplo

```
in("VET", 2, ?x)
```

somente a primeira das tuplas acima poderia satisfazer o *template* especificado, e a variável x receberia o valor 4.5. A recuperação de tuplas através da especificação do valor de alguns dos seus campos é semelhante à operação **select** existente em bancos de dados relacionais[GEL85, AHU86]. Esta maneira de identificar uma tupla é chamada de **nomeação estruturada**, e geralmente é empregada para selecionar uma tupla entre outras que apresentam alguns campos idênticos.

Tuplas são sempre removidas atomicamente. Se dois processos executam **in** esperando pela mesma tupla, somente um deles irá obtê-la, enquanto o outro ficará bloqueado esperando que uma tupla equivalente seja depositada. A ordem com que as operações **in** são satisfeitas, entretanto, é arbitrária.

2.2.3 A primitiva RD

A primitiva **rd**[1], assim como **in**, serve para recuperação de uma tupla que satisfaz um dado *template*. A tupla selecionada, porém, não é removida do espaço de tuplas. Considerando que a tupla ("VET", 3, 5.5) esteja disponível no TS, a execução de

rd("VET", 3, ?x)

fará com que o valor 5.5 seja atribuído à variável x, mas a tupla permanecerá no TS.

Se um processo executa **rd** esperando por uma tupla, e outro processo executa **in** para recuperar a mesma tupla, uma destas situações pode ocorrer: se a operação **rd** é executada antes, a operação **in** poderá posteriormente retirar a mesma tupla. Ao contrário, se **in** é executada antes, a tupla é removida, e a operação **rd** bloqueará até que outra tupla esteja disponível.

2.2.4 Equivalência de Tuplas e *Templates*

A recuperação de tuplas do TS, através de **in** ou **rd**, baseia-se na equivalência entre tuplas e *templates*. Existem quatro condições para que se verifique tal equivalência[LEI89, NAR89]:

1. a tupla e o *template* devem ter o mesmo número de campos;
2. ambos devem ter a mesma **assinatura**. A assinatura de uma tupla ou *template* consiste na descrição do tipo de cada um de seus campos. Por exemplo, a assinatura da tupla ("VET", 3, 4.5) é (char *, int, float), que é idêntica à assinatura do *template* ("VET", ?i, ?x), considerado na seção 2.2.3;
3. a **polaridade** de ambos deve ser equivalente. Define-se a polaridade de um campo como sendo real ou formal. Assim, a polaridade de uma tupla ou *template* consiste na identificação da polaridade de cada um de seus campos. Por exemplo, a polaridade do *template* ("VET", ?i, ?x) é (real, formal, formal). Para que as polaridades de uma tupla e de um *template* sejam equivalentes deve-se garantir que campos correspondentes sejam reais ou de polaridades diferentes. Por exemplo, se houver um campo formal no *template* (ou tupla), o campo correspondente na tupla (ou no *template*) deverá ser real;
4. campos reais correspondentes na tupla e no *template* devem ter o mesmo valor.

[1]Originalmente, esta primitiva chamava-se read. Porém, read é uma função implementada em muitas bibliotecas padrões da linguagem C, e por isso novas implementações de Linda têm adotado o nome rd para esta primitiva.

2.2.5 A Primitiva EVAL

Assim como a operação **out**, **eval** também deposita uma tupla no TS. Porém, antes que a tupla esteja disponível para recuperação, cada um de seus campos é avaliado por um processo criado implicitamente pela operação. Por exemplo, suponha que **dif(x,y)** seja uma função que retorna a diferença entre x e y, ambos números inteiros. Então, a execução de

eval("DIF", dif(55, 23))

criará um novo processo para executar a função dif(55,23). Este processo irá prosseguir em paralelo com o processo que executou **eval** e, ao terminar, a tupla resultante, ("DIF", 32), estará disponível no TS. Pela definição de **eval**, todos os campos fornecidos como argumento a esta primitiva deveriam ser avaliados em paralelo, por processos diferentes. Na prática, entretanto, somente aqueles campos que representam uma chamada de procedimento é que são de fato executados por um novo processo[CAR94].

2.2.6 Primitivas Não-Bloqueantes

Algumas implementações de Linda incluem versões não-bloqueantes das primitivas **in** e **rd**, chamadas, respectivamente, **inp** e **rdp**. Estas operações não bloqueiam o processo que as executa quando não existe no TS uma tupla equivalente ao *template* especificado. Ao contrário, **inp** e **rdp** retornam um valor Booleano, verdadeiro ou falso, indicando se a tupla está ou não presente no TS. Caso a tupla seja encontrada no TS, estas primitivas comportam-se exatamente como suas versões originais.

As primitivas **inp** e **rdp** foram originalmente definidas em [CAR87], com a finalidade de oferecer maior flexibilidade ao programador, especialmente quando é preciso decidir como responder a um evento que indica que um programa terminou ou completou uma fase da sua computação. No entanto, alguns autores [LEI89, BJO92] têm considerado que estas primitivas não são de fato necessárias, e dificilmente podem ser implementadas de maneira eficiente.

2.3 Paralelismo, Comunicação e Sincronização em Linda

Toda linguagem que se propõe a permitir a programação paralela deve oferecer meios de expressar e controlar o paralelismo, além de permitir que processos se comuniquem e sincronizem suas ações. Nesta seção o modelo Linda será abordado com ênfase nestes três aspectos.

2.3.1 Expressão do Paralelismo

A expressão do paralelismo em Linda se dá unicamente através da primitiva **eval**, que cria um processo dinamicamente para executar alguma função especificada. Esta primitiva está automaticamente associada à inserção de uma tupla, o que é bastante expressivo em situações onde a função executada produz um resultado que é esperado por outro processo. No entanto, quando apenas deseja-se executar procedimentos independentes em paralelo, esta semântica pode forçar a produção de um resultado que não é de fato necessário.

Uma questão relacionada à expressão do paralelismo em Linda é a granularidade das tarefas que são executadas em paralelo. Como o paralelismo é expresso a nível de funções, pode-se ter uma granularidade mais fina ou mais grossa de acordo com a quantidade de processamento associado a cada função. No entanto, como a criação dinâmica de um processo é sempre uma operação relativamente demorada, a execução de funções muito simples em paralelo pode ser pouco vantajosa. Desta maneira, o paralelismo através de **eval** tende a ser pouco apropriado para aplicações de granularidade fina.

2.3.2 Comunicação

Em Linda, a comunicação entre processos é feita **indiretamente**, por intermédio do espaço de tuplas. Para isso, um processo usa **out** para depositar no TS uma tupla contendo dados, enquanto outro processo posteriormente a recupera usando **in** ou **rd**, e fornecendo um *template* adequado. A figura 2.1 ilustra a comunicação entre dois processos em Linda.

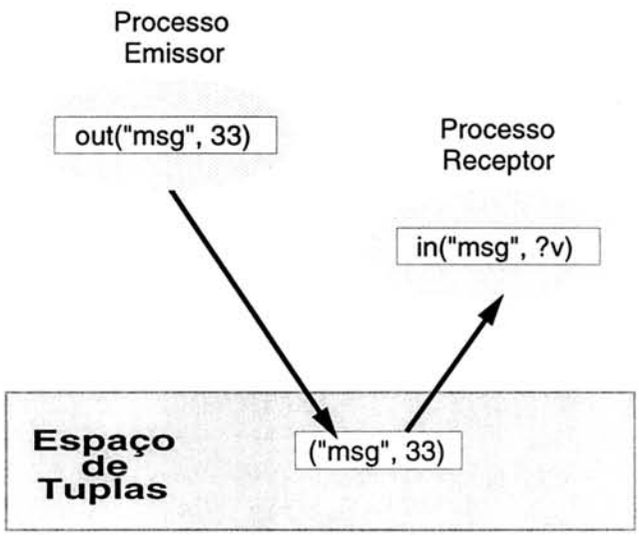

Figura 2.1 - Comunicação entre processos em Linda.

Como nenhuma informação sobre a identificação ou localização dos processos é necessária para a troca de dados entre eles, diz-se que a comunicação em Linda é **anônima**. Nenhum dos processos envolvidos (emissores e receptores) precisa conhecer a identificação dos outros processos, por isso a comunicação também é

dita **ortogonal**. Em alguns sistemas baseados em troca de mensagens esta propriedade não é verificada, pois certas primitivas para comunicação exigem que os emissores identifiquem o processo receptor, mas não o contrário.

Uma característica que distingue Linda da maioria das linguagens para programação paralela é a possibilidade de comunicação entre processos que executam em tempos ou em locais diferentes, isto é, os programas Linda podem ser distribuídos no tempo ou no espaço. A **distribuição no tempo** é possível porque as tuplas são mantidas no TS até que sejam explicitamente removidas, o que pode ocorrer após o término do processo que as inseriu. Na figura 2.2(a) é ilustrada a comunicação entre dois processos (P1 e P2) disjuntos no tempo.

A **distribuição no espaço**, por sua vez, garante que todos os processos, mesmo residentes em diferentes processadores, devem ter a mesma visão do espaço de tuplas. A figura 2.2(b) ilustra a comunicação, via espaço de tuplas, entre dois processos (P3 e P4) disjuntos no espaço.

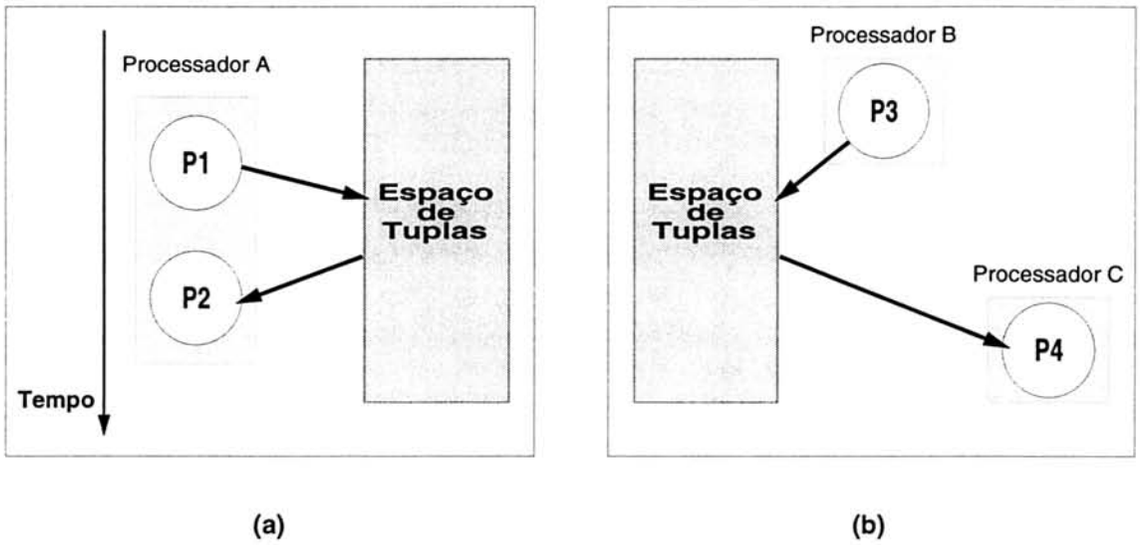

Figura 2.2 - Comunicação entre processos que executam em tempos e locais diferentes.

Embora a comunicação em Linda esteja fortemente ligada ao compartilhamento do espaço de tuplas, o modelo também permite simular vários padrões de comunicação baseados em troca de mensagens[GEL85]. As primitivas definidas em Linda, apesar de simples, são suficientemente flexíveis para expressar, por exemplo, difusão de mensagens e chamadas remotas de procedimento. A seguir será visto como estes dois padrões de comunicação podem ser simulados em Linda.

Difusão de Mensagens

A difusão de mensagens é empregada quando é necessário propagar uma mesma informação para um grupo de processos. Ao contrário da comunicação ponto-a-ponto, que envolve apenas dois processos, a difusão caracteriza-se por envolver vários processos receptores. Por isso é comum referenciar a difusão de mensagens como comunicação um-para-muitos ou comunicação multi-ponto.

Em Linda, para que uma mensagem seja recebida por vários processos, basta depositá-la no espaço de tuplas, onde ela pode ser lida por todos os processos que compartilham o mesmo TS. Assim, por exemplo, um processo que deseja propagar uma mensagem contendo um conjunto de valores deve reunir estes dados numa tupla e inseri-la no TS:

```
out("msg", 13, 62, 73)
```

Os processos que devem receber a mensagem, por sua vez, devem consultar a tupla no TS:

```
rd("msg", ?v1, ?v2, ?v3)
```

Chamada Remota de Procedimento

Uma chamada remota de procedimento (*Remote Procedure Call* — RPC) é um mecanismo que permite a um determinado processo (cliente) invocar um procedimento implementado por outro processo (servidor). Pode-se simular um mecanismo de RPC em Linda utilizando-se uma combinação das primitivas **out** e **in**[GEL85]. Um processo cliente pode implementar uma chamada remota de procedimento da seguinte maneira:

```
out(PROC, "cliente-id", <parametros>)
in("cliente-id", ?<resultados>)
```

Acima, o campo PROC identifica o procedimento a ser invocado remotamente, "cliente-id" é um nome que distingue o processo cliente e <parametros> representa um conjunto de informações necessárias à execução do procedimento. Após depositar a tupla que requisita a execução do procedimento, o cliente espera pelos resultados da execução remota. É importante que a chave "cliente-id" identifique de maneira única o processo cliente, a fim de evitar que repostas destinadas a outros processos sejam recebidas por engano.

Para que o mecanismo de RPC esteja completo, o processo servidor deve estar preparado para receber chamadas dos processos clientes:

```
in(PROC, ?cliente, ?<parametros>)
/* executa procedimento */
out(cliente, <resultados>)
```

O servidor utiliza a operação **in** para receber chamadas e, após executar o procedimento com os parâmetros fornecidos, utiliza **out** para devolver os resultados ao cliente. Esta implementação de RPC, no entanto, não é adequada quando o servidor implementa vários procedimentos que podem ser invocados à distância. Isto porque, no exemplo acima, o servidor fica bloqueado esperando pela chamada de um procedimento em particular, o que impede que outros procedimentos invocados por outros clientes sejam executados. Para permitir que o mecanismo de RPC funcione nesta situação mais genérica, onde um servidor implementa

múltiplos procedimentos, algumas alterações devem ser efetuadas nos códigos executados pelo cliente e pelo servidor:

```
cliente:
out("chamada", "cliente-id", PROC1);
out("parametros", "cliente-id", <parametros>);
in("cliente-id", ?<resultados>);


servidor:
in("chamada", ?cliente, ?proc);
switch(proc) {  /* seleciona procedimento a executar */

    case PROC1: {
        in("parametros", cliente, ?<parametros>);
        /* executa procedimento */
        out(cliente, <resultados>);
    }
}
```

Nesta segunda implementação de RPC em Linda, o cliente efetua a chamada depositando duas tuplas no TS: a primeira seleciona o procedimento a executar, enquanto a segunda fornece os parâmetros para sua execução. O servidor, por sua vez, retira do TS a tupla com a identificação do procedimento que deve ser executado e, a seguir, recupera a tupla correspondente que contém os parâmetros para o procedimento. Esta dissociação em duas tuplas é necessária porque é provável que os diferentes procedimentos implementados pelo servidor recebam argumentos potencialmente diferentes, tanto em número como em tipo. Por isso, os argumentos de cada procedimento devem ser tratados separadamente. Após a execução, a tupla contendo resultados é depositada no TS e retirada pelo cliente como no exemplo anterior. Não é necessária nenhuma alteração nesta tupla, já que o identificador "cliente-id" é único. A tupla somente teria que ser alterada se o mesmo processo cliente executasse chamadas de procedimento em paralelo utilizando o mesmo identificador.

Uma outra maneira de implementar um mecanismo de RPC é através de uma combinação das primitivas **eval** e **in**:

```
eval(FUNC, f(x,y));
in(FUNC, ?resultado);
```

Neste exemplo, o processo servidor é criado implicitamente pelo cliente através de **eval**, a fim de executar a função f(x,y). A seguir, a tupla contendo o resultado é recuperada através de **in**. Este método, no entanto, é apropriado somente à execução de uma função que retorna um único valor como resultado.

2.3.3 Sincronização

A sincronização entre processos no modelo Linda ocorre automaticamente quando um processo espera por uma tupla utilizando as primitivas bloqueantes **in** ou **rd**. A atomicidade no processamento das primitivas Linda garante a exclusão mútua no acesso às tuplas compartilhadas. Quando dois ou mais processos tentam remover uma tupla, só um deles é bem sucedido. Além disso, não existe uma primitiva para alterar os campos de uma tupla. Para fazer isso, um processo deve remover a tupla, modificá-la e inseri-la novamente no TS. Se um processo tenta acessar uma tupla que está sendo modificada, sua execução é suspensa até que a tupla seja novamente inserida no TS, o que preserva a integridade dos dados compartilhados.

Uma tupla que contém um único campo é funcionalmente equivalente a um semáforo[GEL85]. As operações **in**(sem) e **out**(sem) são equivalentes, respectivamente, às operações P(sem) e V(sem) realizadas sobre um semáforo 'sem'. Um processo que executa P pode bloquear esperando por um recurso compartilhado, assim como um processo que utiliza **in** bloqueia se a tupla desejada não está disponível. Em Linda, entretanto, se dois ou mais processos repetidamente concorrem no acesso a uma tupla, é possível que um processo jamais consiga removê-la, já que a ordem com que as operações são atendidas é arbitrária. Isso geralmente não ocorre com semáforos, cuja implementação envolve a manipulação de uma fila de processos bloqueados em operações P. Conforme ocorre a liberação dos recursos (através da operação V), os processos na fila são desbloqueados numa ordem FIFO.

Semáforos não são usados intensivamente em aplicações paralelas, ao contrário do que ocorre em aplicações concorrentes[CAR89a]. No caso de aplicações paralelas, semáforos são mais empregados para sincronização do que para exclusão mútua, já que o compartilhamento de recursos não é tão comum neste tipo de aplicação. Em aplicações paralelas com múltiplas *threads*, no entanto, semáforos podem ser usados para evitar condições de corrida entre *threads* de um mesmo processo.

As primitivas Linda também podem ser usadas para expressar outros padrões de sincronização entre processos, como barreiras, por exemplo. Uma barreira é um tipo de operação global[FOS95] que serve para sincronizar a execução de múltiplos processos. Cada processo que alcança uma barreira permanece bloqueado até que um determinado número de processos também alcance esta mesma barreira. Um mecanismo simples de sincronização de barreira pode ser implementado com uma seqüência de operações Linda. Inicialmente, é necessário inicializar um contador com o número de processos que deverão sincronizar suas execuções:

```
out("barreira", N);
```

Ao atingir uma fase da computação onde é necessária a sincronização de barreira, cada processo deve executar as seguintes operações:

```
in("barreira", ?n);
out("barreira", n-1);
rd("barreira", 0);
```

Nesta implementação, proposta em [CAR89a], cada processo que alcança a barreira decrementa o contador de processos que devem se sincronizar, e a seguir bloqueia esperando que este contador seja decrementado até zero.

2.4 Programação Paralela em Linda

Em Linda, a possibilidade de manipular estruturas de dados distribuídas oferece o suporte ideal a um método de programação paralela conhecido como **mestre/trabalhador** ou **trabalhadores replicados**. Segundo este método, o programa paralelo é composto por vários processos replicados, que executam essencialmente o mesmo tipo de computação. A tarefa a ser computada, no entanto, é dividida em várias sub-tarefas, cujas descrições são armazenadas em uma estrutura de dados distribuída acessível a todos os processos trabalhadores. Cada um destes processos repetidamente retira uma sub-tarefa desta estrutura de dados, realiza a computação necessária, e deposita o resultado também em uma estrutura de dados distribuída.

A decomposição de uma tarefa complexa ou extensa em várias sub-tarefas mais simples pode ser feita dinâmica ou estaticamente. No primeiro caso, cada processo trabalhador também é responsável pela geração de novas sub-tarefas, cujas informações são adicionadas à estrutura de dados. É possível que um processo mestre seja encarregado de gerar a descrição inicial de uma tarefa e, a seguir, se torne também um processo trabalhador, auxiliando na computação das sub-tarefas, bem como na decomposição dinâmica da tarefa inicial ou das próprias sub-tarefas. O número de sub-tarefas a executar, neste caso, pode variar dinamicamente.

A decomposição estática, por sua vez, é realizada previamente por um processo mestre, que organiza as sub-tarefas sob a forma de uma estrutura de dados distribuída. Neste caso, os processos trabalhadores não se envolvem na decomposição de tarefas, e o número de sub-tarefas a executar é fixo, determinado pelo processo mestre no início do programa paralelo.

Além da escolha do método de decomposição de tarefas, aplicações paralelas desenvolvidas no estilo mestre/trabalhador também precisam adotar um método para detectar o fim da computação de todas as tarefas. Geralmente, esta tarefa é responsabilidade do processo mestre, que recolhe os resultados parciais de cada sub-tarefa para, posteriormente, fornecer o resultado final. Neste caso, o fim das computações é detectado pelo processo mestre quando todos os resultados parciais esperados estiverem disponíveis. Uma estratégia alternativa consiste na utilização de um contador de sub-tarefas a serem executadas. Cada processo trabalhador deve ter acesso a este contador, e deve decrementá-lo sempre que uma

nova sub-tarefa é processada. A detecção do término das computações, neste caso, pode ser feita tanto pelo processo mestre como pelos trabalhadores.

O uso do paradigma de trabalhadores replicados em Linda tem algumas vantagens importantes. Aplicações desenvolvidas segundo este paradigma são altamente extensíveis (*"scalable"*), pois a complexidade envolvida na paralelização destas aplicações não depende do número de processos trabalhadores envolvidos. Um programa que emprega uma dezena de trabalhadores pode ser facilmente estendido para empregar uma centena destes processos e, assim, executar mais rapidamente. Outra vantagem é que este paradigma permite o balanceamento dinâmico de carga entre os processos trabalhadores. Como cada trabalhador é responsável pela busca de novas tarefas a executar, a distribuição de tarefas ocorre de acordo com a capacidade de processamento de cada trabalhador. Mais claramente, processos que executam em nodos mais carregados levam mais tempo para completar uma tarefa, e por isso tendem a processar um menor número de tarefas, em relação àqueles processos que executam em nodos menos carregados ou mais potentes. Segundo [CAR95], o paradigma mestre/trabalhador também se mostra bastante apropriado para facilitar o aproveitamento dos nodos desocupados de uma rede. Este é o principal objetivo de um sistema chamado Piranha[CAR95], que suporta paralelismo adaptativo[2] em programas Linda desenvolvidos segundo este paradigma.

Linguagens que não suportam estruturas de dados distribuídas também podem ser usadas para implementar aplicações segundo o paradigma de trabalhadores replicados. No entanto, para simular uma estrutura de dados compartilhada por vários processos, muitas linguagens requerem o uso de um processo gerente[AHU86], que serializa todos os acessos aos dados compartilhados.

Em Linda, o paradigma de trabalhadores replicados é implementado através de tuplas cujos campos descrevem cada sub-tarefa a ser processada. Estas tuplas são depositadas no TS, onde podem ser recuperadas por qualquer processo trabalhador. Para ilustrar a utilização deste paradigma em Linda, vamos considerar uma implementação do algoritmo de Mandelbrot para geração de fractais. Esta implementação consiste em um processo mestre e no mínimo um processo trabalhador. O processo mestre recebe como argumento as coordenadas na tela onde a figura de Mandelbrot deve ser apresentada, e divide a área da figura em blocos cujo tamanho é especificado pelo usuário. Durante a fase inicial desta aplicação, o processo mestre deposita no TS tuplas que contêm as coordenadas de cada um dos blocos para os quais será executado o algoritmo. Estas tuplas têm a forma:

("coords", <x inicial>, <x final>, <y inicial>, <y final>)

Cada trabalhador repetidamente retira do TS uma tupla com coordenadas da área a processar, executa o algoritmo de Mandelbrot para as coordenadas especificadas, e deposita no TS uma tupla contendo informações para apresentação do blo-

[2] O paralelismo adaptativo caracteriza-se por permitir que uma aplicação paralela seja executada sobre um conjunto variável de processadores. Este conjunto pode diminuir quando ocorre a sobrecarga de algum processador, ou pode aumentar quando surgem processadores ociosos.

co processado.

```
in("coords", ?xmin, ?xmax, ?ymin, ?ymax);
mandel(xmin, xmax, ymin, ymax, bloco);
out("bloco", xmin, xmax, ymin, ymax, bloco);
```

Acima, **bloco** é um vetor de bytes que, após a execução da função **mandel**, contém a cor de cada ponto do bloco calculado. O processo mestre recolhe as tuplas depositadas pelos processos trabalhadores, apresentando na tela os pontos de cada bloco que compõe a figura de Mandelbrot. A terminação desta aplicação também é gerenciada pelo processo mestre, que insere no TS tuplas contendo coordenadas nulas. São inseridas tantas tuplas quantos forem os processos trabalhadores. Ao retirar uma tupla deste tipo, cada trabalhador encerra sua execução.

2.5 Limitações do Modelo

Nas seções anteriores procurou-se mostrar as vantagens marcantes que o modelo Linda oferece para o desenvolvimento de aplicações paralelas. Linda, no entanto, também possui algumas limitações, que serão discutidas ao longo desta seção. Embora algumas críticas ao modelo estejam relacionadas à sua implementação eficiente, esta seção não tratará de problemas de implementação, mas sim de algumas limitações decorrentes da semântica inerente ao modelo Linda. Aspectos envolvidos na implementação do modelo serão abordados no próximo capítulo.

Uma das críticas ao modelo diz respeito ao modo com que estruturas de dados distribuídas são manipuladas em Linda. Segundo alguns trabalhos[KAA89, BAL94] que exploram o paradigma proposto por Linda, o modelo permite representar eficientemente estruturas de dados simples, como vetores e matrizes, mas suas primitivas dificultam a construção de estruturas de dados distribuídas mais complexas, como listas e grafos. Embora as primitivas Linda possam ser consideradas de baixo nível neste caso, a solução para este problema provavelmente envolveria recursos somente implementáveis a nível de linguagem, porém Linda não se propõe a ser uma nova linguagem de programação paralela. De fato, os próprios autores dos trabalhos que criticam Linda sob este ponto de vista são também os autores da linguagem Orca[BAL92], que propõe um mecanismo alternativo de suporte a estruturas de dados distribuídas.

Outra limitação do modelo é a ausência de mecanismos que permitam expressar o não-determinismo num programa paralelo. Frequentemente, um processo pode precisar esperar até que uma entre várias condições seja satisfeita, mas não pode prever qual condição ocorrerá primeiro. Algumas linguagens, como Ada[DOD83] e SR[AND93], oferecem **guardas** ou construções do tipo **select**, que permitem expressar este não-determinismo. Linda, entretanto, não suporta mecanismos deste tipo, mas permite simulá-los através das primitivas não-

bloqueantes **inp** ou **rdp**, como mostra o exemplo abaixo:

```
while(TRUE) {
    if (inp("cond1")) { /* testa primeira condiç˜ao */
        proc1();
    }
    if (inp("cond2")) { /* testa segunda condicao */
        proc2();
    }
}
```

Neste exemplo, cada condição é testada repetidamente com **inp** e, quando uma delas é satisfeita, uma ação correspondente é executada. Esta solução para expressar o não-determinismo tem a desvantagem de que o processo é mantido em espera ocupada (*busy-waiting*), o que leva ao desperdício de ciclos do processador.

Existem casos onde o não-determinismo pode ser tratado de outra maneira. Por exemplo, na implementação de servidores que devem esperar por vários tipos de requisições, pode-se utilizar **eval** para disparar processos dedicados ao processamento de cada requisição, evitando-se, assim, que os processos servidores permaneçam em *busy-waiting*.

Por fim, outra limitação que tem sido associada ao modelo Linda quando este é comparado com outros modelos para programação paralela [DOU93, MAT94, MAT95] é a falta de suporte a operações globais, que permitem expressar comunicação e sincronização entre múltiplos processos. Operações deste tipo podem ser implementadas através de primitivas Linda (ver implementação simplificada de barreiras na seção 2.3.3), mas seriam mais eficientes e mais simples de utilizar se fossem suportadas diretamente pelo modelo.

2.6 Sumário

O capítulo apresentou o modelo Linda sob vários aspectos, desde a definição de seus componentes até a avaliação das vantagens e limitações que o modelo oferece para o desenvolvimento de aplicações paralelas. A principal abstração do modelo é o **espaço de tuplas**, que é uma memória global associativa capaz de armazenar coleções de tuplas. Uma **tupla** é basicamente uma seqüência de campos contendo dados de tipos bem definidos. Tuplas podem compor estruturas de dados distribuídas dentro do TS, que podem ser acessadas simultaneamente por vários processos. O modelo prevê um conjunto de primitivas que servem para inserir tuplas no TS (**out** e **eval**), inspecioná-las (**rd**), e removê-las quando necessário (**in**).

O conjunto de primitivas é pequeno, mas tem um poder de expressão muito grande. O paralelismo é expresso através de **eval**, que cria processos dinamicamente para avaliar campos de uma tupla, antes de inseri-la no TS. A comunicação

entre processos é feita indiretamente, por intermédio do espaço de tuplas, de forma anônima e ortogonal. A abstração do espaço de tuplas permite a interação entre processos disjuntos no tempo e no espaço. O modelo permite simular vários padrões de comunicação, como difusão de mensagens ou RPC. A sincronização é automática através das primitivas bloqueantes **in** e **rd**, mas pode-se implementar outros mecanismos de sincronização, como barreiras, por exemplo.

Linda oferece o suporte ideal para o desenvolvimento de aplicações segundo o paradigma mestre-trabalhador, onde um processo mestre divide uma tarefa em várias sub-tarefas menores, que são processadas de forma independente e paralela por processos trabalhadores. Em Linda, estas sub-tarefas podem ser armazenadas no TS, tornando-se acessíveis a qualquer processo trabalhador que esteja livre para processá-las. Uma das principais vantagens disto é permitir um balanceamento dinâmico de carga entre os processos trabalhadores, já que cada um deles é responsável pela busca de novas tarefas a executar, de modo que processos em nodos menos carregados tendem a processar mais tarefas que aqueles que residem em nodos sobrecarregados.

Mesmo suportando um paradigma de alto nível para a interação entre processos, o modelo Linda também tem suas limitações. As principais críticas quanto à semântica do modelo podem ser resumidas na ausência de mecanismos para expressão do não-determinismo em programas paralelos, e na falta de operações globais para expressar interações de alto nível entre vários processos ao mesmo tempo.

3 Implementações do Modelo Linda

Existe atualmente um grande número de sistemas que implementam o modelo Linda total ou parcialmente. A maioria dos sistemas destina-se à programação de aplicações paralelas, incorporando o modelo a linguagens tradicionais como C[SCI92, SCH91], C++[CAL91a], Pascal[PIN91a] e Prolog[MDO90]. Alguns sistemas, como Glenda[SEY93], p4-Linda[BUT93] e Eilean[CAE94b] implementam o modelo utilizando recursos de outras ferramentas para programação paralela, enquanto outros adicionam-lhe novas características, como tolerância a falhas[BAK94] ou mesmo novas primitivas[DOU95]. Existem sistemas baseados em Linda para as mais variadas arquiteturas paralelas, sejam elas com memória compartilhada (como Sequent, Encore e Alliant) ou com memória distribuída (como iPSC[BJO89], Transputer[SHE92] e S/Net[CAR86]). Por fim, existem implementações destinadas à programação a nível de sistema operacional, como por exemplo QIX[LEL90] e Minix/Linda[CIA93], e até uma implementação do modelo a nível arquitetural[AHU88].

O crescente aparecimento de sistemas que implementam Linda sugere uma ampla aceitação deste modelo como uma alternativa para programação paralela. Uma característica que claramente contribui para a aceitação do modelo é sua simplicidade inerente, que se deve tanto ao conjunto pequeno de primitivas quanto à utilização de um paradigma baseado em uma memória compartilhada. Apesar da simplicidade do modelo, o projeto e a implementação de sistemas baseados em Linda envolvem diversas questões, principalmente quando se deseja implementar o modelo em arquiteturas que não dispõem de memória fisicamente compartilhada, como é o caso das redes de computadores e das máquinas paralelas com memória distribuída.

As questões envolvidas no projeto de um sistema baseado em Linda podem ser divididas em dois grupos. De um lado estão as questões que dizem respeito ao **ambiente de programação**, isto é, ao conjunto de recursos e ferramentas que permitem ao usuário desenvolver e executar um programa paralelo segundo o modelo Linda. De outro lado, existem questões relacionadas ao **ambiente de execução** do sistema, isto é, ao conjunto de mecanismos capazes de suportar a execução do modelo Linda em termos de controle do paralelismo, comunicação e sincronização. Um usuário sempre tem conhecimento das questões relacionadas ao ambiente de programação, mas normalmente não precisa conhecer detalhes sobre o ambiente de execução do sistema.

O objetivo deste capítulo é apresentar e discutir algumas questões relevantes no projeto e implementação de sistemas para programação paralela baseados no modelo Linda, especialmente em arquiteturas sem memória compartilhada. As seções 3.1 e 3.2 identificam algumas questões de projeto relacionadas, respectivamente, aos ambientes de programação e execução de sistemas deste tipo, e apresentam algumas alternativas comuns para solução de cada questão. O restante do capítulo é dedicado ao estudo de alguns sistemas que implementam Linda para programação paralela em redes de computadores. Serão descritos os ambientes

de programação e execução de cada sistema, juntamente com alguns detalhes de implementação relacionados.

3.1 Projeto do Ambiente de Programação

Um ambiente de programação consiste em um conjunto de recursos e ferramentas que permitem principalmente o desenvolvimento e execução de programas segundo um determinado modelo de programação. No caso de sistemas baseados no modelo Linda, o ambiente de programação deve no mínimo permitir a construção de programas que interajam através de operações sobre o espaço de tuplas. Respeitando-se esta exigência, existem questões relacionadas ao ambiente de programação cujas soluções podem variar de sistema para sistema. Algumas destas questões serão discutidas a seguir.

3.1.1 Estrutura das Aplicações Paralelas

Em Linda, uma aplicação paralela é constituída por processos que interagem através do espaço de tuplas. O modelo, no entanto, não determina como a aplicação deve ser estruturada. Existem dois métodos básicos de estruturação de uma aplicação paralela. Num deles, o programador decompõe a aplicação em vários processos que executam funções diferentes, e por isso diz-se que o método permite **paralelismo funcional**. É comum referenciar este método de estruturação de aplicações como modelo de processos comunicantes, ou ainda modelo MPMD (*Multiple-Program Multiple-Data*). No segundo método, conhecido como modelo SPMD (*Single-Program Multiple-Data*), a aplicação é estruturada através de processos idênticos, onde cada um manipula uma parte dos dados a serem processados. Por isso, diz-se que o método permite **paralelismo de dados**.

Uma das questões relevantes para o projeto do ambiente de programação de um sistema baseado em Linda é, portanto, a definição do método de estruturação de aplicações que o usuário deverá utilizar. Para oferecer maior flexibilidade ao usuário, é possível projetar um ambiente de programação que permita ao usuário selecionar o método mais adequado a cada aplicação, ao invés de exigir que somente um método seja utilizado.

3.1.2 Conjunto de Primitivas

Um sistema baseado em Linda deve disponibilizar ao usuário um conjunto de primitivas para manipulação de tuplas. Pode-se oferecer primitivas idênticas àquelas definidas no modelo Linda, mas muitos sistemas introduzem algumas modificações na interface Linda original. É comum, por exemplo, a existência de sistemas que não oferecem as primitivas não-bloqueantes definidas em Linda, por motivos que serão discutidos na seção 3.2.4. Igualmente, muitos sistemas modificam a semântica da primitiva **eval**, a fim de facilitar sua implementação (ver

seção 3.2.3). Existem sistemas, entretanto, que estendem o modelo adicionando novas primitivas, como é o caso de Eilean[CAE94b].

Além das modificações no conjunto de primitivas, muitos sistemas alteram a própria estrutura das tuplas, como é o caso de Parlin[CAE94a], Brenda[BRA90] e TsLib[SIL94]. Nesses sistemas, tuplas são implementadas como seqüências de bytes precedidas por um número inteiro, que serve como identificador das tuplas. Esta simplificação da estrutura das tuplas tem como objetivo aumentar a eficiência dos sistemas, à medida que a busca associativa fica resumida a uma simples comparação dos identificadores das tuplas.

3.1.3 Biblioteca × Pré-Compilador

Sistemas baseados em Linda se caracterizam por incorporar o modelo a alguma linguagem para programação seqüencial. Para isso, a maioria dos sistemas adota uma das duas alternativas a seguir:

- reunir as primitivas Linda em uma biblioteca, que deve ser ligada aos programas dos usuários. Sistemas como POSYBL[SCH91], p4-Linda[BUT93] e LiPS[ROT93] adotam esta alternativa, cuja grande vantagem é a facilidade de implementação.

- construir, além da biblioteca, um pré-compilador para a linguagem seqüencial estendida com primitivas Linda. Um pré-compilador deve ser capaz de reconhecer e analisar operações Linda em um programa, de modo a produzir um código fonte otimizado que possa ser compilado normalmente por um compilador da linguagem hospedeira. Sistemas como Network-Linda, Lucinda[BUT91] e AUC C++[CAL91a] empregam esta alternativa, que tem a vantagem de permitir várias otimizações na implementação das primitivas, como será visto na seção 3.3.4 mais adiante.

Embora a implementação somente a nível de biblioteca seja mais simples, esta alternativa geralmente exige mudanças na interface originalmente definida para as primitivas Linda. Na interface original, as primitivas recebem como parâmetro uma lista qualquer de campos, ficando implícito o tipo de cada campo usado. Esta informação sobre o tipo dos campos é essencial para a implementação da associatividade do espaço de tuplas, e pode ser obtida facilmente por um pré-compilador. Entretanto, quando as primitivas são implementadas como funções em uma biblioteca, o tipo de cada campo precisa ser fornecido pelo próprio usuário, o que modifica a interface original e torna mais trabalhoso o uso das primitivas.

A construção de um pré-compilador, ao contrário, é uma alternativa bem mais complexa, mas permite oferecer uma interface simples para o usuário e, além disso, geralmente leva a uma implementação mais eficiente, devido à possibilidade de otimizações em tempo de compilação.

Para combinar algumas vantagens de ambas altenativas, alguns sistemas têm optado por construir apenas um pré-processador, capaz de identificar automaticamente os tipos dos campos em cada operação Linda e, a seguir, substituir cada operação por uma função da biblioteca do sistema. Um pré-processador, no entanto, não analisa um programa com o objetivo de introduzir possíveis otimizações no código.

3.1.4 Ferramentas Adicionais

Além dos recursos para a construção de programas, um ambiente de programação pode incluir diversas ferramentas adicionais. Muitos sistemas baseados em Linda incluem, por exemplo, uma ferramenta capaz de distribuir os processos de uma aplicação sobre uma determinada rede de processadores, a fim de iniciar a execução da aplicação paralela. Também podem ser oferecidas, por exemplo, ferramentas que auxiliem na depuração, na monitoração e na avaliação do desempenho das aplicações desenvolvidas.

3.2 Projeto do Ambiente de Execução

O ambiente de execução de um sistema baseado em Linda deve implementar eficiente e transparentemente a abstração do espaço de tuplas, além de permitir a execução de primitivas para expressão do paralelismo, comunicação e sincronização. Algumas das principais questões relacionadas ao projeto de um ambiente de execução serão descritas a seguir.

3.2.1 Política de Distribuição de Tuplas

Quando o modelo Linda é implementado numa arquitetura com memória compartilhada, o espaço de tuplas pode ser alocado em uma região de memória comum a todos os processos. Em arquiteturas com memória distribuída, entretanto, é necessária a implementação de mecanismos capazes de permitir o compartilhamento lógico do TS entre processos que executam em diferentes nodos ou processadores. Para isso, todo sistema adota uma **política de distribuição de tuplas**, que consiste em algumas regras para decidir onde armazenar cada tupla, e como localizá-la quando necessário. Em outras palavras, uma política de distribuição define regras para a própria implementação do espaço de tuplas.

A escolha de uma política deve levar em conta as características oferecidas pela arquitetura e pelo sistema de interconexão e, quando possível, os tipos de aplicações que devem executar no sistema. Uma política pode ser bastante eficiente para uma determinada combinação das características acima, mas pode tornar-se insatisfatória quando utilizada, por exemplo, em outra arquitetura. No entanto, pode-se identificar algumas características desejáveis para uma política de distribuição genericamente satisfatória:

- **Tráfego de Mensagens:** a implementação de uma determinada política sempre exige um determinado tráfego de mensagens entre os processadores ou nodos envolvidos. Desde que um tráfego intenso pode sobrecarregar o sistema de interconexão, quanto menor o tráfego de mensagens gerado pela política escolhida, melhor será o desempenho do sistema.

- **Grau de Distribuição:** políticas que distribuem o espaço de tuplas sobre muitos nodos têm a vantagem de dividir o custo da busca associativa entre vários processadores, já que a busca pode ocorrer simultaneamente nos vários segmentos que compõem o espaço de tuplas. Além disso, quanto maior o grau de distribuição, menor é a chance de ocorrer a sobrecarga de algum nodo, o que evita o surgimento de gargalos no sistema.

- **Uso de Memória:** algumas políticas replicam tuplas e *templates*, armazenando exatamente os mesmos dados em muitos nodos. Isto requer mais espaço em memória em relação às políticas que não utilizam replicação. Como *templates* são tipicamente menores que tuplas[LEI89], políticas que replicam apenas *templates* são preferíveis em relação àquelas que replicam tuplas sobre muitos nodos.

- **Adaptação aos Padrões de Acesso a Tuplas:** as operações definidas em Linda podem ser combinadas de várias maneiras pelos processos de uma aplicação, produzindo diferentes padrões de acesso a tuplas. Em geral, pode-se classificar estes padrões de acesso da seguinte forma:

  - Tuplas para sincronização: são usadas para implementar algum mecanismo de sincronização através do espaço de tuplas (semáforos, por exemplo). Em geral, tuplas usadas com esta finalidade são somente inseridas e removidas do espaço de tuplas, não sendo comum uma operação de leitura sobre estas tuplas. Além disso, seu conteúdo geralmente não varia, já que a sincronização normalmente não é baseada nas informações contidas nas tuplas, mas sim condicionada à existência das mesmas no espaço de tuplas;
  - Tuplas migratórias: tuplas deste tipo servem para armazenar dados que "migram"de um processo produtor para um processo consumidor. Estas tuplas são recuperadas através de **in** ou **rd** sempre por um processo apenas;
  - Tuplas freqüentemente lidas: servem para armazenar informações que precisam ser consultadas por vários processos diferentes. Operações **rd** são freqüentes para este tipo de tuplas;
  - Tuplas freqüentemente modificadas: servem para armazenar dados que são modificados com freqüência por vários processos diferentes. Tuplas deste tipo são removidas, modificadas e reinseridas no espaço de tuplas muitas vezes.

  Uma política de distribuição capaz de permitir a implementação eficiente destes diversos padrões de acesso a tuplas poderia ser considerada ideal.

No entanto, a maioria das políticas privilegia um ou outro padrão de acesso, de modo que a solução ideal seria identificar os diversos tipos de tuplas manipulados em uma aplicação e adotar políticas de distribuição diferentes para tipos de tuplas distintos. A identificação automática dos tipos de tuplas, entretanto, é uma tarefa bastante complexa.

Alguns sistemas podem ser projetados para suportar uma classe específica de aplicações, de modo que, em alguns casos, é possível prever o padrão de acesso a tuplas que será predominante. Nesta situação, deve-se adotar uma política que favoreça o padrão de acesso identificado, pois isto irá garantir um bom desempenho geral do sistema.

- **Extensibilidade:** algumas políticas têm bom desempenho em sistemas com pequeno número de nodos, mas se comportam insatisfatoriamente em sistemas maiores. Uma política de distribuição extensível deve manter os mesmos níveis de desempenho para qualquer número de nodos utilizados pelo sistema.

A seguir serão descritas as políticas de distribuição mais freqüentemente empregadas, juntamente com alguns detalhes de implementação associados.

Política Centralizada

Numa política centralizada, tuplas são armazenadas em um único nodo, onde reside um processo servidor encarregado de gerenciar o espaço de tuplas. Segundo esta política, operações de inclusão ou recuperação de tuplas executadas por qualquer processo sempre geram mensagens ao servidor centralizado.

A implementação centralizada do espaço de tuplas tem a vantagem de ser uma solução bastante simples. Em particular, a localização de tuplas é facilitada, já que o servidor concentra todas as informações sobre o espaço de tuplas. Por outro lado, o servidor centralizado constitui um ponto crítico do sistema, pois pode ser sobrecarregado com um grande número de requisições a tratar. Quanto maior o número de processos clientes envolvidos (executando potencialmente em diferentes processadores), maior é a chance de o servidor centralizado tornar-se um gargalo. Esta política portanto, não pode ser considerada extensível.

Apesar das desvantagens citadas, uma política centralizada pode ser satisfatória em alguns casos. As experiências com Parlin[CAE94a], um sistema que implementa Linda em transputers, têm concluído que, para sistemas de pequeno e médio porte (isto é, com até algumas dezenas de processadores), um espaço de tuplas centralizado é a melhor solução, embora isto represente um gargalo em sistemas maiores.

Política de Distribuição Uniforme

Segundo esta política, cada tupla gerada através de uma chamada **out** é difundida para um conjunto pré-determinado de nodos, denominado *out_set*. Tuplas

requisitadas através de **in** ou **rd**, por sua vez, são procuradas em um conjunto de nodos denominado *in_set*. Diz-se que esta política distribui tuplas uniformemente porque não existem nodos que centralizam informações sobre o estado global do espaço de tuplas. Os conjuntos *out_set* e *in_set* devem ser definidos para cada nodo da rede, e deve-se garantir que cada *in_set* tenha uma interseção não vazia com os conjuntos *out_set*.

Uma política uniformemente distribuída admite várias possibilidades para escolha dos conjuntos *out_set* e *in_set* de cada nodo. Em S/Net-Linda[CAR86], por exemplo, o *out_set* de um nodo n é formado por todos os nodos da rede (incluindo o nodo n), enquanto o *in_set* é o próprio nodo n. Isto significa que uma operação **out** sempre causa a difusão de uma tupla para rede inteira, o que caracteriza um espaço de tuplas totalmente replicado. Como cada nodo mantém sua própria cópia do espaço de tuplas, operações **in** ou **rd** realizam buscas locais, e a remoção de uma tupla causa a execução de um protocolo para atualização global das cópias. A figura 3.1(a) ilustra esta alternativa para uma distribuição uniforme do espaço de tuplas.

Também é possível um esquema oposto ao anterior, onde o *out_set* de n é o próprio nodo n, e o *in_set* engloba todos os nodos da rede. Desta maneira, tuplas são armazenadas no mesmo nodo onde são geradas e, para localizar uma tupla requisitada através de **in** ou **rd**, torna-se necessária a difusão do *template* correspondente para todos os nodos da rede. Esta alternativa é adotada, por exemplo, no sistema POSYBL[SCH91], e exemplificada na figura 3.1(b).

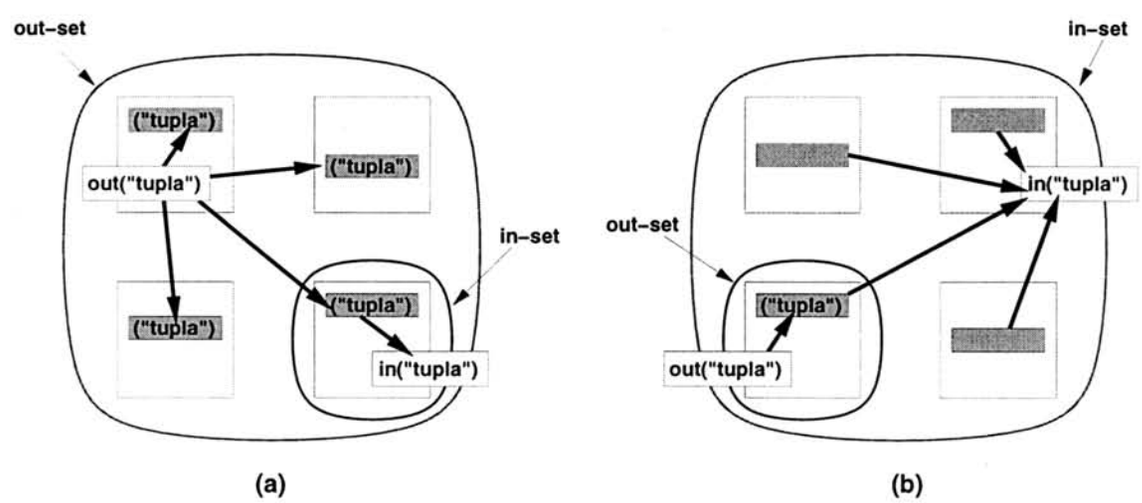

Figura 3.1 - Alternativas para uma política de distribuição uniforme.

A escolha da melhor alternativa para distribuição uniforme depende da topologia do sistema de interconexão e, também, do padrão de acesso a tuplas esperado para o sistema. Quando espera-se que tuplas sejam lidas com freqüência, o primeiro esquema é mais eficiente, já que a leitura é processada localmente (não gera tráfego na rede). Neste esquema, entretando, operações de inclusão ou remoção de tuplas têm alto custo, pois exigem operações de *broadcast*, ou seja, difusão de mensagens para todos os nodos da rede. Operações deste tipo geralmente sobrecarregam o sistema de interconexão, principalmente se o número de

nodos envolvidos é grande – esta alternativa, portanto, não favorece a extensibilidade (*"scalability"*) dos sistemas. Além disso, a replicação total das tuplas requer muito espaço de memória em cada nodo.

Nos casos onde espera-se que a inclusão ou a remoção de tuplas sejam mais freqüentes que a leitura, a segunda alternativa se mostra mais eficiente do que a primeira, já que não requer uma difusão a cada execução de **out** e não necessita de um protocolo para remover cópias de uma mesma tupla. Para localizar qualquer tupla, entretanto, esta alternativa requer uma operação de *broadcast*, neste caso para difusão de *templates*. Como visto no início desta seção, a replicação de *templates* é preferível porque requer menos espaço em memória que a replicação de tuplas.

Quando não é possível prever o padrão de acesso a tuplas que predominará no sistema, ou quando espera-se que operações de inclusão e recuperação de tuplas ocorram aproximadamente com a mesma freqüência, pode-se adotar um esquema de distribuição uniforme intermediário entre os dois anteriores. Neste esquema intermediário, o espaço de tuplas é parcialmente replicado. Supondo uma rede de N nodos, cada conjunto *out_set* ou *in_set* é composto por $\sqrt{N}$ nodos. Os conjuntos devem ser arranjados de tal maneira que cada *in_set* inclua no mínimo um membro de cada conjunto *out_set*.

O esquema intermediário apresenta algumas vantagens em relação às duas alternativas anteriores. Em particular, o número de nodos participantes de uma determinada transação **out-in** é mínimo neste esquema[AHU88]. Este número é igual a N em cada um dos esquemas anteriores, já que ambos requerem uma difusão para todos os nodos da rede, seja para a implementação de **out** (no primeiro caso) como para a implementação de **in** ou **rd** (no segundo caso). No esquema intermediário, uma transação deste tipo requer a participação de apenas $2\sqrt{N}$ nodos, sendo $\sqrt{N}$ envolvidos na operação **out**, e os outros $\sqrt{N}$ na operação **in**. Outra vantagem diz respeito ao uso de memória, que é menor em relação às alternativas anteriores, já que a replicação de tuplas e *templates* é parcial.

Um exemplo de emprego da distribuição uniforme intermediária é o projeto arquitetural de uma máquina paralela baseada no modelo Linda. Nesta máquina, chamada Linda Machine[AHU88], o espaço de tuplas é implementado em hardware sobre uma malha[1] de $\sqrt{N} \times \sqrt{N}$ processadores, onde N é o número total de nodos. A distribuição das tuplas se dá de tal maneira que o *out_set* de um nodo n é formado pelos $\sqrt{N}$ da linha de n (incluindo o próprio nodo), enquanto seu *in_set* é composto pelos $\sqrt{N}$ nodos da sua coluna.

Esta política de distribuição é empregada de maneira semelhante em outras arquiteturas paralelas com processadores conectados em forma de malha. Como exemplo pode-se citar a implementação preliminar de Linda no "SBN network computer"[GEL85] e uma implementação em redes de transputers[FAA91]. Nesta última, entretanto, o emprego da distribuição uniforme intermediária foi considerado ineficiente, devido ao *overhead* envolvido na comunicação entre os processadores da rede[FAA91].

---

[1] *mesh*

Política Baseada em *Hashing*

Numa política deste tipo, tuplas são distribuídas sobre um conjunto de processadores de acordo com uma função de *hash* aplicada a uma chave extraída de cada tupla. A localização de tuplas também se dá através desta função, que deve retornar a identificação do nodo onde uma determinada tupla pode ser encontrada. A chave fornecida como argumento para a função de *hash* pode ser constituída de várias maneiras. Pode-se utilizar, por exemplo, o valor de alguns campos das tuplas, ou então a assinatura de cada tupla, que consiste na descrição de cada um de seus campos (ver seção 2.2.4).

Para a implementação desta política pode-se empregar um processo servidor em cada nodo escolhido para armazenar tuplas. Cada processo passa a ser responsável por alguns tipos de tuplas, determinados pela política de *hashing*, e deve atender operações **out**, **in** ou **rd** relacionadas a tuplas destes tipos. Entre os sistemas que adotam uma política baseada em *hashing* estão Modula-L[TRE92] e a implementação de Linda na arquitetura hipercúbica iPSC[BJO89].

A utilização de uma política baseada em *hashing* tem várias vantagens. Na implementação das primitivas Linda, por exemplo, não é necessária a difusão de mensagens para todos os nodos da rede. A execução de qualquer primitiva gera apenas uma requisição ao servidor que reside no nodo determinado pela função de *hash*. Além disso, como não existe replicação, não é necessária a utilização de protocolos para manter a consistência do espaço de tuplas. Esta política também favorece a extensibilidade do sistema, à medida que o volume de mensagens para implementação das primitivas independe do número de nodos na rede.

Existem entretanto alguns problemas com relação à esta política. Segundo [CAE94a], o principal problema é que uma função de *hash* simples mapeia tuplas de um mesmo tipo sempre em um único nodo, o que pode causar o surgimento de gargalos no sistema. Isto é particularmente crítico quando o uso de tuplas de um mesmo tipo é freqüente. Em aplicações no estilo mestre-trabalhador, por exemplo, existem normalmente apenas dois tipos de tuplas: as que contêm tarefas a executar e as que armazenam resultados da execução destas tarefas. Numa situação como esta, o desempenho do sistema pode ser limitado pelo fato de que as tuplas acabam sendo distribuídas em apenas dois nodos.

Política Baseada em Servidores de Tipos

Esta política é semelhante a uma política baseada em *hashing* no sentido de que tuplas são distribuídas entre processos servidores de acordo com seu tipo ou, em outras palavras, de acordo com alguma informação extraída de seus campos. A diferença, no entanto, é que nesta política não se usa uma função para mapear cada tipo de tupla em um processador, pois este mapeamento é controlado por processos servidores especiais, denominados **servidores de tipos**. Toda operação de manipulação de tuplas, neste caso, é implementada em duas etapas: na primeira é feito o contato com um servidor de tipos, para determinar a localização

do servidor responsável pela tupla, e na segunda é enviada uma requisição a este processo servidor.

Um servidor de tipos pode ter conhecimento sobre todos os tipos de tuplas mantidos no sistema, ou pode conhecer apenas uma parte destes tipos. A maneira como estes servidores gerenciam as informações sobre cada tipo de tupla também varia de sistema para sistema, podendo, por exemplo, envolver operações de difusão no caso destas informações serem replicadas sobre vários servidores. Esta política requer um maior volume de mensagens para sua implementação, se comparada com uma política baseada em *hashing*. No entanto, o uso de servidores de tipos pode levar a uma melhor distribuição de tuplas, evitando o surgimento de gargalos no sistema. O uso de memória nesta política não é ideal, já que pode haver replicação de informações sobre tipos, mas pelo menos não há replicação de tuplas, que exige mais espaço em memória.

Esta política de distribuição de tuplas é utilizada, por exemplo, em D-Linda[PIN91], um sistema baseado em Linda que tem a particularidade de permitir que programas escritos em linguagens diferentes também possam interagir através de um espaço de tuplas. Em D-Linda, existe um processo servidor de tipos e um processo servidor de tuplas em cada nodo. Cada tipo de tupla é gerenciado por um servidor de tuplas em exatamente um nodo, e tem um identificador único que deve ser registrado junto a um servidor de tipos. A princípio, o servidor de tipos em um nodo mantém informações sobre tipos de tuplas armazenados localmente e, ao receber uma requisição para um tipo de tupla desconhecido, ele se comunica com servidores semelhantes em outros nodos, a fim de localizar o tipo solicitado. É possível, no entanto, que determinados servidores tenham diferentes graus de conhecimento sobre tipos de tuplas mantidos em servidores remotos, de modo a otimizar a localização de tuplas no sistema.

Política Hierárquica

Alguns sistemas têm adotado uma política de distribuição hierárquica, baseada no particionamento do espaço de tuplas global em vários domínios ou sub-espaços, sendo cada domínio diretamente acessível a um determinado conjunto de processadores. Com esta política, operações de inclusão ou recuperação de tuplas ficam, em sua maioria, restritas a um único sub-espaço, o que diminui o número de nodos participantes em cada operação, e também reduz o volume de mensagens necessárias para implementação das primitivas.

O particionamento do espaço de tuplas, entretanto, não deve impedir que processos tenham acesso a vários sub-espaços, pois, do contrário, o espaço de tuplas deixaria de ser uma memória globalmente acessível. Por isso, a implementação de uma política deste tipo geralmente utiliza duas hierarquias de processos: uma responsável pelo gerenciamento local de cada sub-espaço, e outra pela coordenação do espaço global. No projeto de Eilean[CAE94b], por exemplo, cada domínio é gerenciado por um único processo servidor. De fato, os domínios em Eilean podem ser vistos como pequenos espaços de tuplas centralizados. Além disso, um destes processos servidores, denominado Super Master, é escolhido como co-

ordenador dos demais, e passa a supervisionar os acessos ao espaço de tuplas global.

Embora a criação de sub-espaços seja uma alternativa para otimizar a distribuição de tuplas, isto só ocorre quando são minimizados os acessos a sub-espaços não-locais. Um particionamento eficiente, entretanto, é difícil de ser feito automaticamente pelo sistema, já que depende das particularidades de cada aplicação. Sistemas como Eilean, acima mencionado, e Linda-Polylith[MAT93], outra implementação que utiliza uma política hierárquica, têm deixado a tarefa de particionamento do espaço de tuplas para o usuário. Isso, no entanto, tende a comprometer a transparência do espaço de tuplas, que é uma das grandes vantagens do modelo Linda.

3.2.2 Grau de Compartilhamento do Espaço de Tuplas

Independentemente da política de distribuição de tuplas adotada, deve-se escolher o grau de compartilhamento desejado para o espaço de tuplas. A princípio, processos de uma mesma aplicação sempre compartilham um mesmo espaço de tuplas. No entanto, existem sistemas que permitem o compartilhamento entre processos de diferentes aplicações do mesmo usuário, ou até mesmo entre aplicações de usuários diferentes. Deve-se notar, entretanto, que o compartilhamento do espaço de tuplas entre aplicações e usuários diferentes pode levar a interferências indesejáveis, principalmente quando aplicações independentes utilizam tuplas idênticas.

3.2.3 Implementação de EVAL

Um aspecto crítico no projeto de um ambiente de execução baseado em Linda é o suporte à primitiva **eval**. De acordo com os próprios projetistas do modelo, **eval** é a operação mais difícil de implementar[HUP91]. Para simplificar a implementação do modelo Linda, muitos sistemas até deixam de oferecer esta operação, ou modificam consideravelmente sua semântica. Quando se opta por implementar esta primitiva, deve-se resolver duas questões importantes:

- Onde alocar o novo processo?

  Conforme definido em Linda, **eval** deve provocar a criação de um novo processo para avaliar os campos de uma tupla. No entanto, o modelo não especifica onde este processo deve ser alocado. Portanto, cada sistema é livre para decidir se processos serão criados local ou remotamente, ou ainda se ambas opções serão permitidas.

- Qual o contexto inicial do novo processo?

  O modelo Linda não define qual o contexto inicial do processo criado através de **eval**. Uma conseqüência disso, por exemplo, é que não fica definido se o novo processo pode ter acesso a variáveis globalmente definidas em

um programa Linda. No projeto do ambiente de execução, a definição do contexto inicial do novo processo deve levar em conta se **eval** pode ou não criar processos remotamente. Se **eval** só dispara a execução de processos no nodo local, o contexto pode ser herdado ou compartilhado com outro processo. Em caso contrário, o contexto pode ser totalmente independente do processo de origem, ou então pode ser transferido de um nodo para outro.

Existem várias alternativas para solucionar as questões acima. A mais simples delas é restringir a criação de processos ao nodo local, e utilizar os recursos do sistema operacional hospedeiro para criação de um novo processo. Numa implementação em sistemas Unix, por exemplo, pode-se utilizar a chamada **fork**, através da qual o novo processo herda uma cópia do estado global do processo de origem. Neste caso, a própria chamada **fork** se encarrega da criação do contexto inicial do novo processo. Outra alternativa para implementação é a criação não de um novo processo, mas sim de uma nova *thread* para avaliar os campos da tupla. Basicamente, *threads* permitem múltiplos fluxos de execucão concorrentes dentro de um único processo, sendo também conhecidas como *processos leves*. Uma *thread* compartilha o espaço de endereçamento do processo ao qual pertence, mas possui seu próprio contador de programa e sua própria pilha.

Quando o ambiente de execução deve permitir a criação de processos em diferentes nodos, pode-se utilizar os recursos disponíveis para criação remota de processos, como o comando **rsh** disponível na maioria dos sistemas Unix. O contexto inicial do novo processo, neste caso, é totalmente independente do processo de origem. Outra alternativa é implementar **eval** como uma combinação de operações **outin**, onde todas as informações necessárias à execução do novo processo são depositadas no espaço de tuplas, e um processo servidor remoto se encarrega de retirar estas informações e iniciar a execução do novo processo. Esta solução é bastante trabalhosa, já que o ambiente de execução deve encapsular todas as informações necessárias à execução do novo processo em uma tupla especial e, a seguir, depositá-la no espaço de tuplas. Neste caso, o contexto inicial do processo é transferido de um nodo para outro através do espaço de tuplas.

3.2.4 Implementação de INP e RDP

Como visto anteriormente na seção 3.1.2, alguns sistemas optam por não oferecer as primitivas não-bloqueantes **inp** e **rdp** em seus ambientes de programação. Esta decisão, no entanto, se deve principalmente à dificuldade de implementar estas operações de maneira correta e eficiente junto ao ambiente de execução.

O resultado de uma operação **inp** ou **rdp** sempre representa uma afirmação muito forte sobre o estado global do sistema: uma tupla **está** ou **não está** disponível no espaço de tuplas. Dependendo da política de distribuição de tuplas adotada, entretanto, o resultado destas operações (verdadeiro ou falso) só pode ser fornecido corretamente após a consulta de todos os segmentos que compõem o espaço de tuplas. Se esta consulta envolver um grande número de nodos, a

implementação de **inp** ou **rdp** pode gerar um alto tráfego de mensagens, comprometendo a eficiência do sistema. Além disso, deve-se garantir que, enquanto esta consulta está em andamento, tuplas não sejam transferidas de um segmento para outro, pois do contrário poderia ser produzido um resultado incorreto caso a tupla procurada fosse transferida de um segmento ainda não consultado para outro onde a consulta tenha fornecido resultado negativo (tupla não disponível).

As dificuldades mencionadas acima, entretanto, são minimizadas quando o ambiente de execução utiliza, por exemplo, uma política centralizada ou baseada em *hashing*. Nestes casos, apenas um nodo precisa ser consultado na busca de uma determinada tupla, permitindo uma implementação eficiente das primitivas não-bloqueantes. Mesmo assim, um outro argumento tem sido levantado por alguns autores contra a implementação de **inp** e **rdp**[LEI89, BAK94]: ao contrário das outras primitivas do modelo, estas primitivas introduzem uma noção de sincronia global nas aplicações onde são empregadas. Quando são utilizadas somente primitivas bloqueantes, o processo que executa **in** ou **rd** não precisa se preocupar em garantir que a operação **out** correspondente já tenha sido executada por outro processo. Por outro lado, quando se utiliza **inp** ou **rdp**, este tipo de cuidado se torna necessário, podendo causar grandes dificuldades no desenvolvimento de aplicações.

3.2.5 Suporte à Heterogeneidade

Ao projetar-se um sistema Linda para redes de computadores, deve-se considerar a possibilidade de distribuir o espaço de tuplas sobre máquinas de diferentes arquiteturas. A heterogeneidade é uma realidade marcante na maior parte dos sistemas de computação e, no caso das redes de computadores, há muito tempo vêm sendo possível interligar máquinas de arquiteturas completamente diferentes. Desta maneira, é vantajoso implementar um sistema que permita a execução de uma aplicação Linda sobre um conjunto heterogêneo de máquinas, já isto proporciona um melhor aproveitamento dos recursos computacionais. Além disso, é possível projetar sistemas que tirem proveito da heterogeneidade para executar uma aplicação mais rapidamente, distribuindo uma carga computacional maior sobre as máquinas mais poderosas disponíveis na rede.

Apesar de vantajoso, o suporte à heterogeneidade implica em diversos cuidados na implementação da comunicação entre máquinas de diferentes arquiteturas. Além disso, isso implica num *overhead* adicional no ambiente de execução, o que tem uma influência negativa no desempenho geral dos sistemas.

3.3 Linda em Redes: Estudo de Casos

Esta seção apresenta um estudo sobre cinco sistemas baseados em Linda disponíveis atualmente: Glenda, POSYBL, p4-Linda, Eilean e Network-Linda. Estes sistemas estão entre os mais conhecidos atualmente e, assim como YALI, destinam-se à programação paralela em redes de computadores. De cada sistema

serão analisados os ambientes de programação e execução, de modo a ilustrar a implementação de algumas soluções discutidas no início deste capítulo.

3.3.1 Glenda

Glenda[SEY93] é uma implementação do modelo Linda sobre PVM[SUN90] (*Parallel Virtual Machine*), um ambiente para programação paralela baseado em troca de mensagens. PVM foi especialmente projetado para suportar redes heterogêneas de computadores com sistema compatível com Unix, e tem sido implementado em várias arquiteturas, desde estações de trabalho até multiprocessadores com ou sem memória compartilhada. A portabilidade deste ambiente, aliada ao suporte a linguagens tradicionais como C, C++ e Fortran, têm contribuído para que PVM tenha um grupo muito grande de usuários no mundo inteiro, e venha sendo considerado como um "padrão de fato" no que diz respeito a ambientes para troca de mensagens.

O objetivo principal de Glenda é unir os atributos positivos tanto de Linda como de PVM[SEY93]. Assim, Glenda suporta a funcionalidade do espaço de tuplas e as primitivas Linda, mas mantém a portabilidade de PVM e também permite que processos se comuniquem diretamente quando isto for desejável. Este recurso, entretanto, não mantém o mesmo nível de abstração do modelo Linda pois, como será visto a seguir, a comunicação deixa de ser anônima e ortogonal.

Ambiente de Programação

O ambiente de programação de Glenda consiste basicamente em um pré-processador e uma biblioteca de funções para programas em C. Nesta biblioteca, as rotinas para manipulação de tuplas requerem a especificação do tipo e tamanho de cada campo recebido como argumento, mas estas informações não precisam ser fornecidas explicitamente pelo usuário. A identificação dos tipos e tamanhos dos campos é feita automaticamente pelo pré-processador do sistema, que se encarrega de introduzir estas informações em cada chamada Glenda encontrada durante o processamento de um programa fonte. O sistema é flexível quanto à estruturação das aplicações, que podem ser organizadas tanto no modelo SPMD como MPMD.

Glenda difere do modelo Linda original em alguns aspectos. Em primeiro lugar, a primitiva **eval** não é implementada. Para criação de processos, Glenda oferece a função **gl_spawn**, que é apenas uma interface para a primitiva **pvm_spawn** do sistema PVM. Além disso, Glenda exige que o primeiro campo de uma tupla ou *template* seja sempre uma cadeia de caracteres. O conjunto de funções suportadas por Glenda será apresentado a seguir.

- **gl_mytid** e **gl_exit**: a função **gl_mytid** habilita o acesso ao espaço de tuplas, e por isso deve ser utilizada antes de qualquer função de manipulação de tuplas. Além disso, **gl_mytid** retorna o número de identificação do proces-

so que a executa. A função **gl_exit**, por sua vez, serve para encerrar um processo Glenda.

- **gl_out**, **gl_in** e **gl_rd**: estas funções são equivalentes, respectivamente, às primitivas **out**, **in** e **rd** do modelo Linda. Os tipos de dados permitidos em Glenda baseiam-se nos tipos básicos suportados pela linguagem C. O sistema distingüe os seguintes tipos: caracteres (char), números inteiros (short, int ou long), números de ponto flutuante (float ou double) e cadeias de caracteres (char *). Glenda também permite que estes tipos básicos sejam combinados na forma de vetores. O trecho de programa a seguir ilustra a utilização da primitiva **gl_out** com tuplas de diferentes tipos:

```
int   i, a[10];
int   *p;
float value = 12.34;

for (i=0; i < 10; i++)
    a[i] = i;

p = &a[0];

gl_out("data", i, value);
gl_out("array", a);
gl_out("another_array", p:10);
```

Na última operação deste exemplo, o símbolo **':'** precede a especificação do tamanho de um campo. Isto é necessário quando o campo é representado por um ponteiro, pois neste caso o pré-processador não consegue determinar automaticamente o tamanho do campo, ao contrário do que acontece com vetores previamente declarados. O exemplo a seguir ilustra a recuperação das tuplas geradas pelas operações acima:

```
int   *p, b[10];
int   len;
float number;

p = (int *) malloc(sizeof(int) * 20);

gl_in("data", 10, ?number);
gl_rd("array", ?p:len);
gl_in("another_array", ?b);
```

Neste exemplo, nota-se que o símbolo **'?'** indica um campo formal. Quando um campo formal deve receber um conjunto de valores, como é o caso da operação **gl_rd** acima, o número de elementos neste conjunto (tamanho do vetor, neste caso) é atribuído à variável que segue o símbolo **':'**. A segunda operação **gl_in** acima também recebe um conjunto de valores mas, neste caso, o tamanho do vetor é implícito.

- **gl_inp** e **gl_rdp**: estas funções são equivalentes, respectivamente, às primitivas não-bloqueantes **inp** e **rdp** do modelo Linda. Se a tupla especificada em uma destas operações não é encontrada no espaço de tuplas, ambas retornam o valor 0, e não bloqueiam o processo usuário. Caso contrário, retornam o valor 1, e tupla especificada é recuperada normalmente.

- **gl_spawn**: esta função serve para criar um processo dinamicamente em Glenda, e recebe como argumento o nome do programa a ser executado pelo novo processo. Opcionalmente, pode-se fornecer também o nome da máquina onde o processo deve ser criado. Se este argumento não é fornecido, o processo é criado em uma máquina escolhida pelo sistema. Um exemplo de utilização desta função será visto junto com a descrição das funções **gl_outto** e **gl_into** a seguir.

- **gl_outto** e **gl_into**: estas funções foram adicionadas com o objetivo de aproveitar os recursos de comunicação um-para-muitos oferecidos por PVM. A função **gl_outto** pode ser utilizada para enviar uma tupla diretamente para um ou mais processos, ou seja, sem o intermédio do espaço de tuplas. Os processos receptores, por sua vez, devem usar a função **gl_into** para receber uma tupla gerada por **gl_outto**. Embora a comunicação direta possa ser mais eficiente, estas funções não mantêm o mesmo nível de abstração do modelo Linda. Como o usuário precisa informar a identificação dos processos que devem receber uma tupla, a comunicação deixa de ser anônima. Além disso, só o emissor precisa identificar o receptor, e por isso a comunicação deixa de ser ortogonal.

  Para exemplificar a utilização de **gl_outto**, considere as operações abaixo:

```
float value = 3.14;
int   i, tid[10];

for (i = 0; i < 10; i++)
    tid[i] = gl_spawn("worker");

gl_outto(tid:10, "data", value);
```

  Neste exemplo, o vetor tid contém os identificadores dos processos criados através de **gl_spawn** para executar o programa "worker". A operação **gl_outto** faz com que uma tupla com dois campos seja enviada a cada um destes processos. O símbolo **':'**, neste caso, precede a especificação do tamanho do vetor tid. Para receber a tupla, cada processo worker deve executar uma operação do tipo:

```
float   number;

gl_into("data", ?number)
```

Como será visto na próxima seção, Glenda utiliza um processo servidor para gerenciar o espaço de tuplas, que por sua vez necessita do sistema PVM para

executar. Por isso, antes de executar qualquer aplicação, o usuário deve primeiro inicializar o sistema PVM e depois invocar o processo servidor de Glenda. Durante a inicialização de PVM o usuário deve especificar os nomes das máquinas que podem ser usadas na execução de aplicações PVM. Como Glenda executa sobre PVM, o mesmo conjunto de máquinas serve também para execução de aplicações Glenda. Um usuário pode executar várias aplicações simultaneamente, utilizando o mesmo processo servidor. No entanto, para garantir que aplicações independentes não interfiram umas nas outras, o usuário deve evitar que aplicações distintas utilizem tuplas semelhantes.

Ambiente de Execução

Glenda emprega uma **política centralizada** para implementação do espaço de tuplas, e para isso utiliza um processo servidor que gerencia o TS à medida que atende a requisições de manipulação de tuplas provenientes de processos clientes. Estes processos, por sua vez, se comunicam com o servidor através das funções disponíveis na biblioteca do sistema, descritas na seção anterior.

O espaço de tuplas em Glenda pode ser compartilhado por processos clientes pertencentes a várias aplicações de um mesmo usuário. Glenda, no entanto, não implementa qualquer tipo de mecanismo capaz de impedir que aplicações independentes interfiram umas nas outras através do espaço de tuplas. Como visto na seção anterior, cabe ao usuário evitar que isto aconteça. Por outro lado, esta característica pode ser vantajosa para implementar a interação entre aplicações disjuntas no tempo, onde dados produzidos por uma aplicação permanecem no espaço de tuplas até serem utilizados por outra aplicação executada após o término da primeira.

A implementação tanto do servidor como da biblioteca do sistema é baseada principalmente nos recursos que PVM oferece para troca de mensagens entre processos. Grande parte das funções da biblioteca utiliza as rotinas **pvm_send** e **pvm_recv** de PVM, que servem para envio e recepção de mensagens, respectivamente. Estas rotinas permitem a comunicação entre processos residentes em máquinas de diferentes arquiteturas, por isso pode-se dizer que o suporte à heterogeneidade em Glenda é uma característica herdada do sistema PVM. Para implementação de **gl_out**, uma mensagem contendo a tupla a ser armazenada no TS é enviada ao servidor. Já a implementação de **gl_in**, **gl_rd**, **gl_inp** ou **gl_rdp** requer o envio de um *template* ao servidor e, a seguir, o uso de **pvm_recv** para aguardar a recepção de uma tupla equivalente. Em particular, a implementação de **gl_inp** e **gl_rdp** é facilitada pelo uso do servidor centralizado.

O servidor aguarda requisições de processos clientes através de **pvm_recv**, e envia respostas através de **pvm_send**. Ao receber uma requisição **gl_out**, o servidor armazena a tupla recebida em uma tabela *hash*, que representa fisicamente o espaço de tuplas. Quando recebe uma requisição de recuperação de tuplas, o servidor procura uma tupla equivalente na tabela *hash*, usando como chave o primeiro campo do *template* recebido. Se uma tupla é encontrada, o servidor a envia para o processo cliente que gerou a requisição e, caso a operação seja **gl_in** ou

gl_inp, a tupla é também removida do TS. Se nenhuma tupla é encontrada, e se a requisição é do tipo bloqueante (**gl_in** ou *gl_rd*), o servidor mantém o *template* em uma outra tabela *hash*, que é consultada sempre que uma nova tupla é recebida. Isto permite que a requisição pendente seja satisfeita tão logo uma tupla equivalente esteja disponível. Por outro lado, quando a requisição é do tipo não-bloqueante, o servidor simplesmente envia uma resposta indicando que a tupla não foi encontrada, e nenhuma pendência precisa ser mantida.

Como visto na seção anterior, Glenda inclui as funções **gl_outto** e **gl_into**, que manipulam tuplas sem o intermédio do servidor centralizado. A função **gl_outto** é implementada através da rotina **pvm_mcast** de PVM, que permite enviar uma mensagem diretamente para um ou mais processos. A função **gl_into**, por sua vez, usa **pvm_recv** para receber somente tuplas geradas através de **gl_outto**. É possível, entretanto, que a tupla recebida não seja equivalente ao *template* fornecido como argumento a **gl_into** (podem existir vários processos executando **gl_outto** com diferentes tuplas). Neste caso, a tupla é inserida em uma tabela *hash* mantida pelo próprio processo e, a cada nova execução de **gl_into**, esta tabela é examinada na procura de uma tupla que satisfaça imediatamente o *template* especificado.

As funções restantes de Glenda não manipulam tuplas e, portanto, são implementadas sem o intermédio do servidor. A função **gl_mytid** inicializa a tabela *hash* local (já que é através dela que o sistema Glenda é inicializado) e executa a rotina **pvm_mytid**, que retorna o número de identificação do processo junto ao ambiente de execução PVM. As funções **gl_spawn** e **gl_exit** apenas executam rotinas equivalentes do sistema PVM (**pvm_spawn** e **pvm_exit**, respectivamente). Em particular, a rotina **pvm_spawn** é bastante flexível quanto a localização do novo processo, permitindo executá-lo tanto em um nodo determinado pelo usuário como também em uma máquina escolhida pelo próprio sistema.

3.3.2 POSYBL

POSYBL[SCH91] (*PrOgramming SYstem for distriButed appLications*) é uma implementação do modelo Linda para redes de estações Sun ou DEC com sistema operacional compatível com Unix. Este sistema foi desenvolvido no Departamento de Ciência da Computação da Universidade de Creta, na Grécia, e, assim como Glenda, permite que primitivas do modelo Linda sejam utilizadas em programas escritos em C. Ao contrário de Glenda, entretanto, POSYBL não é implementado a partir de uma ferramenta de mais alto nível, mas sim através dos recursos que o próprio sistema operacional Unix oferece para lidar-se com comunicação entre processos através da rede.

Ambiente de Programação

Em POSYBL, aplicações são estruturadas estritamente conforme o modelo MPMD, e as rotinas para manipulação de tuplas são reunidas em uma biblioteca para programas em linguagem C. O sistema não dispõe de um pré-processador,

e por isso o usuário deve empregar funções pré-definidas para indicar o tipo de cada campo fornecido como argumento para operações **out**, **in** e **rd**. Existem também outras diferenças em relação ao modelo Linda original. Para facilitar a recuperação de tuplas, POSYBL convenciona que o primeiro campo de toda tupla ou *template* deve ser sempre real. Além disso, POSYBL substitui a primitiva **eval** por algumas funções que servem apenas para criação dinâmica de novos processos.

Por fim, este sistema não oferece as primitivas não-bloqueantes **inp** e **rdp**, já que sua implementação envolveria um *overhead* de comunicação muito alto em relação às outras primitivas do modelo[SCH91]. As principais funções disponíveis na biblioteca deste sistema serão descritas a seguir.

- **init**: esta função deve ser utilizada quando o usuário deseja executar programas POSYBL independentes de maneira concorrente. Em POSYBL, uma aplicação consiste em uma coleção de processos que pertencem a um mesmo Grupo de Execução (*Run Group*). Um Grupo de Execução é identificado por um número inteiro, conhecido por todos os processos deste grupo. A fim de impedir que operações executadas em diferentes Grupos de Execução interfiram umas nas outras através do espaço de tuplas, cada processo fornece o identificador do seu grupo como argumento para **init**. Além deste identificador, **init** recebe também o nome de um arquivo que enumera os nodos disponíveis para execução do processo. Quando o usuário não deseja executar mais de uma aplicação por vez, a função **init** não precisa ser usada. Neste caso, o sistema automaticamente considera que os processos são do Grupo de Execução 0, e os nodos disponíveis devem ser enumerados em um arquivo chamado ".nodefile".

  No exemplo abaixo, a função **init** é usada por um processo pertencente ao Grupo de Execução 5, e a identificação dos nodos disponíveis é fornecida no arquivo "nodes".

  ```
  int RunGroup = 5;

  init(RunGroup, "nodes");
  ```

- **eval_l**, **eval_v**, **eval_nl** e **eval_nv**: estas funções substituem a primitiva **eval** em POSYBL. As duas primeiras, **eval_l** e **eval_v**, servem para execucão remota de um dado programa no nodo menos carregado da rede, e têm sintaxe semelhante à das chamadas Unix **execl** e **execv**, respectivamente. As duas últimas, **eval_nl** e **eval_nv**, são semelhantes às primeiras, porém requerem que o usuário especifique o nome do nodo onde o programa deve ser disparado. A operação abaixo ilustra a utilização de **eval_nl**:

  ```
  eval_nl("minuano", "/users/andrea/worker", NULL);
  ```

  Nesta operação, um novo processo é criado no nodo "minuano", para executar o programa "worker" que está no diretório "/users/andrea".

- **out, in** e **rd**: estas funções manipulam tuplas em POSYBL. Os campos das tuplas podem conter dados de cinco tipos básicos: caracteres (**char**), números inteiros (**short** ou **int**), números de ponto flutuante (**float** ou **double**) ou cadeias de caracteres (**string**). Além disso, dados dos quatro primeiros tipos podem ser combinados na forma de vetores. Como mencionado anteriormente, é necessário que o próprio usuário especifique o tipo de cada campo de uma tupla. Para isso, POSYBL define o seguinte conjunto de funções para descrição de campos, onde **tipo-s** pode ser qualquer um dos tipos básicos, e **tipo-v** pode ser qualquer tipo básico exceto **string**:

  - **l[tipo-s](valor)**: indica um campo real simples;
  - **ln[tipo-v](valor, número)**: indica um campo real em forma de vetor;
  - **ql[tipo-s](endereço)**: indica um campo formal simples;
  - **qln[tipo-v](endereço, endereço)**: indica um campo formal em forma de vetor.

  Para ilustrar o uso destas funções, considere o trecho de programa abaixo:

```
short  *shortp, primes[] = {1, 2, 3, 5, 7};
char   charvar;
int    size = 5;
int    intvar = 22;
float  pi;

out(lstring("data"), lchar('x'), lshort(1),
    lfloat(3.1416), lint(intvar));
out(lstring("primes"), lnshort(primes, 5));
in(lstring("data", qlchar(&charvar),
    lshort(1), qlfloat(&pi), lint(intvar));
rd(lstring("primes"), qlnshort(&shortp, &size));
```

  Acima, a primeira operação **out** insere uma tupla com cinco campos reais simples, enquanto a segunda deposita uma tupla contendo dois campos, sendo o último um vetor de números inteiros. Sempre que um vetor de qualquer tipo é utilizado, seu número de elementos deve ser fornecido como parâmetro à função descritora correspondente (acima, o vetor tem cinco elementos, e este número é fornecido à função **lnshort**). Estas duas tuplas são recuperadas através das operações **in** e **rd** logo a seguir. A operação **in** remove a primeira tupla usando como chave o primeiro, o terceiro e o quinto campo da tupla. Tanto o segundo como o quarto campo são formais e, como devem receber um valor, são passados por referência às funções descritoras **qlchar** e **qlfloat**. Por fim, a operação **rd** recupera a segunda tupla, fazendo com que **shortp** receba o endereço do vetor recuperado, e o tamanho do vetor seja atribuído à variável **size**. Os dados do vetor ficam localizados em uma área de memória que pode ser sobrescrita após outra operação de recuperação de tuplas, e por isso o usuário deve copiá-los para outra área previamente alocada sempre que precisar manipular esses dados.

Para que usuários possam executar aplicações POSYBL, o espaço de tuplas precisa ser previamente inicializado sobre um conjunto de nodos disponíveis. Como será visto mais adiante, o espaço de tuplas em POSYBL é implementado através de processos gerenciadores de tuplas (*Tuple Managers*), que devem executar em cada nodo da rede. Para inicialização do espaço de tuplas, POSYBL fornece o utilitário **startup**, que permite distribuir os processos gerenciadores sobre um conjunto especificado de nodos. Uma vez inicializado, o espaço de tuplas é compartilhado por aplicações de diferentes usuários, que são distribuídas sobre a rede também através do programa **startup**. O sistema não permite que cada usuário inicialize seu próprio espaço de tuplas, mas impede que aplicações pertencentes a usuários diferentes interfiram umas nas outras, mesmo quando utilizam tuplas idênticas.

POSYBL oferece também um utilitário chamado *system*, que permite a interação direta com os processos gerenciadores do espaço de tuplas. Através deste utilitário, um usuário pode obter informações estatísticas sobre a utilização do espaço de tuplas, ordenar a reinicialização do espaço de tuplas para um determinado Grupo de Execução, ou mesmo encerrar a execução dos processos gerenciadores.

Ambiente de Execução

Em POSYBL, o espaço de tuplas é distribuído entre processos gerenciadores de tuplas (*Tuple Managers*) que executam em cada nodo da rede. Cada gerenciador de tuplas é, na verdade, um servidor que aceita requisições de processos clientes locais, e que se comunica com outros processos gerenciadores quando necessário. POSYBL emprega uma **política uniforme** para distribuição de tuplas, onde o *out_set* de cada nodo é o próprio nodo, e o *in_set* consiste em todos os nodos da rede. Isto quer dizer que tuplas são armazenadas no processo gerenciador do nodo onde foram geradas, existindo sempre uma única cópia de cada tupla no TS. Por outro lado, tuplas que não são localizadas no gerenciador local devem ser procuradas em todos os gerenciadores espalhados sobre a rede.

Somente um processo gerenciador pode executar em cada nodo, e por isso o espaço de tuplas pode ser compartilhado por processos de diferentes aplicações e de diferentes usuários. Para impedir que aplicações independentes interfiram umas nas outras através do espaço de tuplas, o gerenciador armazena separadamente as tuplas e *templates* pertencentes a Grupos de Execução e usuários diferentes.

A implementação do sistema baseia-se principalmente em chamadas de procedimentos remotos (RPC). POSYBL foi construído a partir de uma biblioteca de RPC para o sistema Unix, que facilita a implementação de sistemas segundo o modelo cliente-servidor. Esta biblioteca, por sua vez, utiliza o mecanismo de *sockets*[STE90] do sistema Unix para comunicação entre processos. As rotinas de manipulação de tuplas disponíveis ao usuário invocam procedimentos implementados pelo processo servidor local que, por sua vez, pode invocar procedimentos em processos servidores residentes em outros nodos. Cada processo

servidor implementa quatro procedimentos para manipulação de tuplas e *templates*, e outros três para controle do sistema. A implementação de cada um destes procedimentos será discutida a seguir.

- Manipulação de tuplas: os procedimentos de manipulação de tuplas (e *templates*) que cada servidor está preparado para processar são LindaOut, LindaIn, LindaRead e LindaTemplate. Os procedimentos LindaIn e LindaRead são invocados apenas por processos clientes locais, enquanto os dois restantes são invocados tanto por clientes locais como por outros servidores remotos.

  - LindaOut: um procedimento deste tipo é usado para implementar a rotina **out** na biblioteca do sistema. O processo cliente invoca LindaOut no servidor fornecendo como parâmetro uma tupla a ser armazenada. Ao processar uma chamada deste tipo, o servidor verifica se a tupla pode satisfazer uma chamada LindaIn ou LindaRead proveniente de algum processo local ou remoto, sempre levando em conta as identificações do usuário e do Grupo de Execução. Para isso, o servidor consulta duas tabelas de operações pendentes: uma onde são mantidos *templates* gerados por processos locais, e outra onde ficam temporariamente armazenados os *templates* provenientes de outros nodos. Se a tupla não satisfaz nenhum *template*, o servidor armazena-a numa tabela *hash* que representa fisicamente o segmento local do espaço de tuplas (existe uma tabela dessas para cada combinação de usuário e Grupo de Execução). Se a tupla satisfaz um *template* local, o servidor responde diretamente ao cliente que a solicitou. Por fim, se um *template* remoto é satisfeito, o servidor faz uma chamada LindaOut ao servidor do nodo remoto, enviando a tupla recém recebida. Em ambos os casos, se o *template* satisfeito era proveniente de uma operação **rd**, o servidor também armazena a tupla localmente.
  - LindaIn: a chamada deste procedimento faz com que o servidor procure localmente por uma tupla equivalente ao *template* recebido. Caso uma tupla seja encontrada, seu conteúdo é enviado ao processo cliente que gerou a requisição e, a seguir, a tupla é removida do espaço de tuplas. Em caso contrário, o *template* é armazenado numa tabela de pendências locais, e uma chamada LindaTemplate é difundida a todos os gerenciadores distribuídos sobre a rede. Esta operação é periodicamente repetida caso uma tupla equivalente não seja recebida através de uma chamada LindaOut.
  - LindaRead: esta chamada é implementada de maneira semelhante a LindaIn. No entanto, quando uma tupla é encontrada, o servidor responde ao processo cliente mas não remove a tupla logo a seguir. Neste caso, a tupla permanece armazenada para acessos futuros.
  - LindaTemplate: este procedimento é invocado por um processo gerenciador remoto quando uma operação **in** ou **rd** não é satisfeita localmente. O servidor que recebe uma chamada LindaTemplate deve procurar localmente uma tupla equivalente ao *template* recebido. Se uma tupla

é encontrada, o servidor responde ao gerenciador remoto através de uma chamada LindaOut. Em caso contrário, o *template* é armazenado na tabela de requisições pendentes provenientes de gerenciadores remotos, onde permanece até que uma tupla equivalente esteja disponível, ou até esgotar-se um intervalo de tempo pré-definido. Este intervalo de tempo é renovado sempre que o gerenciador remoto envia um *template* idêntico através de outra chamada LindaTemplate.

- Controle do Sistema: cada procedimento de controle do sistema é invocado simultaneamente em todos os processos gerenciadores, através do utilitário **system** que acompanha o sistema POSYBL.
  - SystemStats: este procedimento envia estatísticas sobre a utilização do espaço de tuplas ao processo cliente (**system**), que se encarrega de mostrá-las ao usuário.
  - SystemReset: reinicializa as tabelas *hash* que cada gerenciador mantém para uma determinada combinação de usuário e Grupo de Execução.
  - SystemExit: encerra a execução dos processos gerenciadores, desde que o usuário que solicita esta operação seja o mesmo que inicializou o sistema através de **startup**.

Ao contrário das rotinas de manipulação de tuplas e controle do sistema, as funções para criação de processos (**eval_l**, **eval_v**, **eval_nl**, **eval_nv**) são implementadas sem o intermédio de um processo servidor, já que não estão associadas a inserção automática de tuplas. Basicamente, POSYBL utiliza o comando **rsh** do Unix para disparar um programa executável em algum nodo da rede. Para implementação das funções **eval_l** e **eval_v**, que devem disparar um processo no nodo menos carregado da rede, POSYBL utiliza a função **getrusage** disponível em alguns sistemas Unix (SunOs, por exemplo), que serve para determinar a carga em um determinado nodo.

3.3.3 p4-Linda

O sistema p4-Linda[BUT93] acrescenta a funcionalidade do modelo Linda a um ambiente para programação paralela chamado p4[BOY87] (*Portable Programs for Parallel Processors*). Basicamente, p4 consiste em uma biblioteca de macros e subrotinas para programação paralela em vários tipos de arquitetura, desde multiprocessadores com memória compartilhada até redes heterogêneas de computadores. Esta biblioteca é disponível tanto para programas em C como Fortran, e oferece, entre outros recursos, um mecanismo baseado em monitores para programação em arquiteturas com memória compartilhada, e um variado conjunto de primitivas para troca de mensagens, destinado a arquiteturas com memória distribuída.

Aproveitando o suporte de p4 a vários tipos de arquiteturas, p4-Linda foi implementado em duas versões compatíveis: uma para arquiteturas com memória

compartilhada, e outra para redes de computadores heterogêneas, que será discutida logo a seguir. Ambas as versões de p4-Linda, assim como o próprio sistema p4, foram desenvolvidas no Laboratório Nacional de Argonne, nos Estados Unidos.

Ambiente de Programação

Em p4-Linda, aplicações paralelas devem ser estruturadas segundo o modelo SPMD, e devem utilizar rotinas para manipulação de tuplas disponíveis em uma biblioteca para programas em C (embora p4 permita desenvolver programas em Fortran, p4-Linda não inclui esta facilidade). Nesta implementação, as primitivas **out**, **in** e **rd** exigem que o usuário especifique o tipo de cada campo que compõe uma tupla. Isto é feito através de uma **máscara**, que consiste em uma cadeia de caracteres com formato semelhante ao usado na função **printf** da linguagem C, como será visto mais adiante. Além disso, a primitiva **out** só admite campos reais, as primitivas não-bloqueantes **inp** e **rdp** não são suportadas, e a semântica de **eval** também difere daquela originalmente definida no modelo Linda. As rotinas que p4-Linda disponibiliza ao usuário serão discutidas mais detalhadamente a seguir.

- **linda_init** e **linda_end**: p4-Linda requer que o usuário utilize estes procedimentos respectivamente para inicialização e término de programas. O procedimento de inicialização envolve basicamente a criação de processos que suportam o modelo Linda, além dos processos definidos pelo usuário. Isto é feito com o auxílio de um arquivo de configuração, chamado *procgroup*, onde o usuário fornece (1) o nome das máquinas onde os processos devem ser criados, (2) o número de processos que podem compartilhar uma mesma máquina e (3) o nome dos programas que devem executar em cada uma das máquinas.

  A fase de inicialização também interfere no uso de **eval**. É necessário que o usuário forneça, como argumento para *linda_init*, uma tabela identificando as funções que podem ser invocadas através de **eval**. Cada entrada na tabela deve conter o endereço de uma função e um nome a ela associado (ver descrição de **eval** mais adiante). Para ilustrar a inicialização de uma aplicação p4-Linda, considere o trecho de programa da figura 3.2. Neste exemplo é incluído o arquivo de cabeçalho sr_linda.h, que contém, além de outras definições, a declaração da estrutura linda_eval_tbl. Esta estrutura representa a tabela onde devem ser registradas as funções que podem ser invocadas através de **eval**. Na figura 3.2, esta tabela contém apenas a função **consumer**.

- **out**, **in** e **rd**: como mencionado anteriormente, estas rotinas requerem o uso de uma máscara que descreve cada um dos campos de uma tupla. Os tipos de dados válidos em p4-Linda são números inteiros, cadeias de caracteres e números de ponto flutuante. Na máscara, estes três tipos básicos são denotados, respectivamente, pelos caracteres '**d**', '**s**' e '**f**'. Para especificar se um determinado campo é real ou formal, utilizam-se os símbolos '**%**' ou '**?**'

```
#include "sr_linda.h"

main(argc, argv)
int argc;
char **argv;
{
    int consumer();
    struct  linda_eval_tbl eval_funcs[2];

    eval_funcs[0].ptr = consumer;
    strcpy(eval_funcs[0].name, "consumer");
    eval_funcs[1].ptr = NULL;

    linda_init(&argc, argv, eval_funcs);

    /* corpo do programa omitido */

    linda_end();
}

int consumer() {
    /* corpo da funçao omitido */
}
```

Figura 3.2 - Trecho de programa em p4-Linda.

precedendo o caracter descritor do tipo de dado. Também é possível utilizar agregados (neste caso, vetores formados pelos tipos básicos), que são identificados pelo símbolo ':' seguido pelo descritor do tipo básico de dado.

Existem algumas restrições que devem ser respeitadas na composição das tuplas. Em primeiro lugar, o primeiro campo de uma tupla deve ser real, podendo ser um número inteiro ou uma cadeia de caracteres. Além disso, a primitiva **out** só admite campos reais, e campos agregados fornecidos como argumento para **in** ou **rd** são sempre formais (por isso não são utilizados símbolos distintos para agregados reais ou formais). Campos agregados, em particular, devem sempre ser seguidos por um argumento que especifica seu número de elementos. Isso é exemplificado no trecho de programa a

seguir:

```
int   i, a1[10];
int   s1 = 10;
int   num1 = 100;

for (i=0; i < 10; i++)
    a1[i] = i;

out("%s%d:d", "array", num1, a1, s1);
```

No exemplo acima, **out** insere no TS uma tupla com três campos, sendo o primeiro uma cadeia de caracteres, o segundo um número inteiro, e o terceiro um vetor de inteiros. A dimensão do vetor é dada pela variável `s1`. Esta mesma tupla poderia ser recuperada, por exemplo, através da operação abaixo:

```
int   a2[10];
int   num2;
int   s2;

in("%s?d:d", "array", &num2, a2, &s2);
```

Desde que campos formais devem receber um valor, as variáveis num2 e `a2` são passadas por referência. Como `a2` é um agregado, sua dimensão é atribuída à variável `s2`.

- **eval**: em p4-Linda, a primitiva **eval** serve para disparar um novo processo para executar uma dada função. Esta função deve ter sido previamente registrada em uma tabela durante a inicialização da aplicação. Nesta tabela, o usuário deve associar um nome para cada função que pode ser executada através de **eval**. Este nome deve, então, ser usado como argumento a uma chamada **eval**. Em p4-Linda, no entanto, esta primitiva não está automaticamente associada à inserção de uma tupla, como originalmente definido em Linda. Caso isto seja necessário, a inserção da tupla deve ser feita pela própria função invocada através de **eval**. Considerando o exemplo da figura 3.2, a função **consumer** poderia ser ativada através da operação **eval** a seguir:

```
eval("%s", "consumer");
```

Ambiente de Execução

A versão de p4-Linda destinada à programação em redes implementa o espaço de tuplas de modo **centralizado**, utilizando um único processo para gerenciar o TS e processar requisições **out** **in** ou **rd**. Considerando esta política centralizada, pode-se identificar três tipos de processos em uma aplicação p4-Linda:

- um processo **mestre**, que tem a função de inicializar o ambiente de execução;
- um processo **gerenciador do espaço de tuplas**, que é criado durante a inicialização pelo processo mestre;
- processos **escravos**, que também são criados pelo processo mestre, e que utilizam as primitivas p4-Linda para comunicação entre si ou com o mestre.

A rotina **linda_init**, que deve ser executada pelo processo mestre no início de cada aplicação, é responsável por disparar o processo gerenciador do espaço de tuplas. Cada aplicação tem seu próprio espaço de tuplas, compartilhado apenas pelos processos que a compõem. Os processos escravos também são criados durante a inicialização, mas só são ativados quando existe uma chamada **eval** a processar. Para implementação de **eval**, p4-Linda deposita no TS centralizado uma tupla especial contendo o nome da função a executar. Os processos escravos permanecem bloqueados esperando que tuplas deste tipo sejam inseridas no espaço de tuplas. Cada operação **eval** é atendida por um processo cliente, que retira a tupla correspondente do TS e ativa a função nela especificada.

O processo gerenciador e as rotinas de manipulação de tuplas são implementados, basicamente, através das funções **p4_send** e **p4_recv** da biblioteca p4, que servem, respectivamente, para envio e recepção de mensagens. A primitiva **out** constrói uma estrutura baseada na tupla recebida como argumento, e a envia na forma de uma mensagem para o processo gerenciador. Este, ao receber uma requisição **out**, armazena a estrutura que representa a tupla em uma tabela *hash*, que representa fisicamente o espaço de tuplas.

As primitivas **in** e **rd**, por sua vez, enviam ao gerenciador uma estrutura representando um *template* e aguardam a recepção de uma tupla equivalente, bloqueando a execução do processo usuário de **in** ou **rd**. O processo gerenciador, ao receber uma requisição **in** ou **rd**, procura uma tupla equivalente na tabela *hash*. Quando a busca é bem sucedida, a tupla é enviada ao processo que gerou a requisição e, se necessário (operação **in**), a tupla é também removida do TS. Caso uma tupla equivalente não seja encontrada, o gerenciador armazena o *template* em uma fila de espera. Esta fila é examinada sempre que novas requisições **out** são recebidas, a fim de satisfazer, se possível, as requisições **in** e **rd** pendentes. Uma tupla pode satisfazer várias requisições **rd** pendentes, e só é de fato inserida no TS caso não satisfaça nenhuma requisição **in** da fila de espera.

3.3.4 Network Linda

A primeira implementação do modelo Linda em redes de computadores foi desenvolvida na Universidade Yale, e chamava-se TSnet[ARA89]. Este sistema deu origem a Network Linda[CAG93, CAR94], uma implementação do modelo para redes heterogêneas, comercializada pela empresa americana Scientific Computing Associates. Existem versões de Network Linda para as linguagens C e Fortran, e ambas as versões executam em uma grande variedade de estações Unix, como Sun, IBM e Silicon Graphics.

Ambiente de Programação

Entre os sistemas estudados, Network Linda é, sem dúvida, o mais sofisticado. Este sistema inclui, entre outros componentes, um pré-compilador e um pré-ligador, o que permite introduzir diversas otimizações na implementação do modelo, que serão discutidas na próxima seção. Embora um dos principais objetivos seja garantir eficiência em tempo de execução, a existência de um pré-compilador também permite desenvolver uma interface bastante simples para o usuário.

Network Linda oferece um conjunto completo de primitivas Linda: **out**, **in**, **rd**, **eval**, e as primitivas não-bloqueantes **inp** e **rdp**. A sintaxe das primitivas é idêntica àquela definida originalmente no modelo, isto é, não existe a necessidade de explicitar o tipo de cada campo fornecido como parâmetro a uma primitiva, já que esta informação pode ser obtida pelo pré-compilador. Os tipos de dados permitidos em Network Linda são aqueles suportados pela linguagem hospedeira (C ou Fortran), além de agregados formados pelos tipos básicos de dados. Na versão para a linguagem C (C-Linda[SCI92]), por exemplo, é utilizado um compilador específico, que suporta vetores e até estruturas de tamanho variável. Campos agregados, entretanto, devem ser seguidos por um símbolo ':' e por uma especificação do tamanho do campo, como no exemplo abaixo:

```
char  v[] = {'a', 'e', 'i', 'o', 'u'};

out("vogais", v:5);
```

Opcionalmente, o tamanho do campo pode ser seguido por outro símbolo ':' e pela especificação de um limite de transferência. Isto é usado em operações de recuperação de tuplas que envolvem agregados como campos formais, para garantir que um campo não receba mais dados do que pode armazenar[NAR89]. Em operações deste tipo, o tamanho real do campo é atribuído à variável que precede o limite de transferência. O exemplo na figura 3.3 emprega estruturas de tamanho variável, ilustrando o propósito do limite de transferência.

Outra vantagem decorrente da utilização de um pré-compilador é que, em Network Linda, a primitiva **eval** permite passar argumentos para as funções que devem ser executadas em paralelo. A figura 3.4 ilustra esta facilidade através de um programa C-Linda extraído de [MAT95], que implementa um algoritmo para cálculo de π em paralelo. Como pode ser visto neste exemplo, a função principal do programa de usuário deve se chamar real_main, já que o próprio sistema possui uma função principal (main) pré-definida, que inclusive é responsável pela chamada de real_main. O exemplo da figura 3.4 utiliza também uma função adicional oferecida por Network Linda, lprocs, que retorna o número de nodos envolvidos na execução da aplicação.

Em Network Linda, aplicações devem ser estruturadas segundo o modelo SPMD. Para colocar uma aplicação em execução sobre uma rede de computadores, o sistema fornece um utilitário chamado **ntsnet**, que carrega um mesmo programa Network Linda em cada nodo disponível da rede. Cabe ao usuário

```
struct  var {
    int  v_num;
    char v_name[1];
}


example() {

    struct var *sv1, *sv2;
    int size1, size2;
    char *name = "testing";

    size1 = sizeof(struct var) + strlen(name) + 1;

    if ((sv1 = (struct var *) malloc(size1)) == NULL)
            exit(1);
    sv1->v_num = 1515;
    strcopy(sv1->v_name, name);

    out("var", *sv1:size1);

    if ((sv2 = (struct var *) malloc(100)) == NULL)
            exit(1);

    in("var", ?*sv2:size2:100);

}
```

Figura 3.3 - Manipulação de tuplas em Network-Linda.

especificar quais nodos estão disponíveis ao sistema, mas os nodos realmente usados na execução de uma aplicação podem ser escolhidos de várias maneiras, dependendo dos parâmetros de configuração fornecidos a **ntsnet**. Este programa é bastante flexível, permitindo o ajuste de vários parâmetros através de arquivos de configuração ou mesmo através da linha de comando. Entre alguns recursos importantes oferecidos por **ntsnet** está a capacidade de carregar programas em nodos remotos mesmo quando a rede não dispõe de NFS (*Network File System*). A maioria dos sistemas para programação paralela em rede conta com NFS para garantir que a acessibilidade de um mesmo programa em vários nodos, mas este não é o caso de Network Linda. Na ausência de NFS, o programa **ntsnet** se encarrega de copiar programas em cada nodo onde for necessário. Outro recurso oferecido por **ntsnet** é a escolha automática dos nodos menos carregados da rede para execução de uma aplicação. Esta funcionalidade adicional tem seu preço, entretanto, já que a construção dos arquivos de configuração pode ser bastante trabalhosa[MAT95].

```
double  pi_comp(int ID, int num_steps, int num_nodes);

real_main() {

    int     j, num_steps, num_nodes;
    double  pi, partial_sum;

    printf("\n How many steps should we take? ");
    scanf("%d", &num_steps);

    num_nodes = lprocs();

    for (j=0; j<num_nodes-1; j++)
        eval("partial_sum", pi_comp(j, num_steps,
            num_nodes);

    pi = pi_comp(num_nodes-1, num_steps, num_nodes);

    for (j=1; j<num_nodes; j++) {
        in("partial_sum", ?partial_sum);
        pi = pi + partial_sum;
    }

    printf("For num_steps = %d steps, PI = %f \n",
        num_steps, pi);
}
```

Figura 3.4 - Programa em Network-Linda para cálculo de π em paralelo.

Além do utilitário **ntsnet**, Network Linda também inclui duas outras importantes ferramentas. A primeira é TupleScope[BER90], um depurador gráfico para programas Linda, que permite visualizar o conteúdo do espaço de tuplas. A outra ferramenta é uma interface para o ambiente Paragraph[HEA90], que permite monitorar o desempenho de aplicações paralelas.

Ambiente de Execução

A implementação de Network Linda envolve três componentes básicos: um pré-compilador (C-Linda ou Fortran-Linda), um pré-ligador, e uma biblioteca que implementa as rotinas chamadas em tempo de execução [CAG93, CAR94]. O pré-compilador, derivado daquele desenvolvido originalmente na Universidade Yale[CAR87, CAR90], processa um programa C-Linda ou Fortran-Linda e produz um código fonte puramente em C ou Fortran, que é então processado pelo compilador original, novamente C ou Fortran. Durante a pré-compilação, operações Linda são substituídas por funções intermediárias, inicialmente "va-

zias", que devem ser preenchidas em tempo de ligação a fim de invocar rotinas implementadas na biblioteca do sistema. Nesta fase o pré-compilador também coleta informações sobre o uso do espaço de tuplas, que são armazenadas num arquivo auxiliar (*"Linda object file"*).

Em tempo de ligação, todos os módulos objetos que compõem um determinado programa são conhecidos, e portanto o pré-ligador é capaz de analisar os os acessos ao espaço de tuplas realizados por cada primitiva Linda usada neste programa. Com base nas informações obtidas da análise de todos os arquivos auxiliares construídos em tempo de compilação (*"Linda object files"*), o pré-ligador então "preenche" as funções intermediárias geradas pelo pré-compilador. Basicamente, o que estas funções fazem é armazenar dados sobre uma tupla ou *template* em uma estrutura de dados adequada, e invocar uma rotina apropriada (selecionada pelo pré-ligador) na biblioteca do sistema. A seguir, o ligador padrão do sistema hospedeiro é usado para produzir o código executável final.

O pré-ligador é o principal responsável pelas otimizações que garantem a eficiência de um programa Network Linda. Pela análise dos dados coletados em tempo de compilação, é possível selecionar a rotina mais apropriada para implementar uma determinada operação sobre o espaço de tuplas. A biblioteca do sistema é, na verdade, uma coleção de rotinas que implementam as operações Linda sobre diferentes **classes** de tuplas ou *templates*. Estas classes são formadas com base no número de campos e na assinatura de cada tupla ou *template*, bem como nos campos reais constantes que geralmente são usados na composição destes elementos. Para cada classe é selecionada uma estrutura de dados apropriada para armazenamento das tuplas ou *templates*, juntamente com um conjunto de rotinas adequado para manipulação desta estrutura. Em tempo de execução, operações de recuperação de tuplas só precisam procurar entre as tuplas da mesma classe do *template* fornecido, o que reduz o custo da busca associativa.

Existe também uma outra otimização que pode ser realizada pelo pré-ligador. Quando são encontradas, por exemplo, operações **out** e **in** que manipulam tuplas e *templates* de uma mesma classe, e que também possuem campos constantes em comum, pode-se prever, em tempo de ligação, que estas operações devem interagir em tempo de execução. Isto pode ser visto como uma pré-busca associativa, onde a equivalência de alguns campos (constantes) é detectada em tempo de ligação. Assim, estes campos não precisam ser levados em conta na busca em tempo de execução, desde que o pré-ligador substitui as operações envolvidas por rotinas mais específicas, que interagem diretamente. A seguir são exemplificadas algumas classes de tuplas que podem ser identificadas pelo pré-ligador, a fim de ilustrar o funcionamento das otimizações recém discutidas.

- Muitas vezes, operações sobre tuplas com um único campo constante, como por exemplo **out**("sem") e **in**("sem"), são utilizadas para sincronização (ver seção 2.3.3). Através das otimizações em tempo de ligação, é possível suprimir o campo constante, e utilizar um contador para representar as tuplas pertencentes a esta classe. A operação **out**, neste caso, pode ser substituída por uma rotina que incrementa o contador correspodente à tupla usada. Já a operação **in** deve decrementar este contador, a menos que seu valor seja

igual a zero, quando **in** deve bloquear até que uma operação **out** seja executada.

- Em muitos casos, tuplas são usadas para armazenar variáveis compartilhadas. Estas tuplas são tipicamente manipuladas por operações do tipo:

  out("nome", valor)
  in("nome", ?valor)
  rd("nome", ?valor)

  Como é possível identificar esta classe de tuplas em tempo de ligação, o campo constante pode ser suprimido, e pode-se usar uma fila ou pilha para armazenar os valores variáveis.

- Quando tuplas são usadas para compor uma estrutura de dados distribuída (um vetor, por exemplo), é comum utilizar operações do tipo:

  out("A", indice, valor)
  in("A", indice, ?valor)
  rd("A", indice, ?valor)

  Acima, apenas o primeiro campo pode ser suprimido, já que os campos restantes são variáveis (seu valor não pode ser determinado antes da execução). No entanto, o pré-compilador pode detectar que o segundo campo é sempre real, e pode usar este campo como uma chave para busca de tuplas que pertencem a esta mesma classe. Neste caso, uma tabela *hash* pode ser usada para armazenar tais tuplas.

Embora o pré-compilador e o pré-ligador tenham um papel importante em Network Linda, são as rotinas da biblioteca do sistema que compõem realmente o seu ambiente de execução. Estas rotinas são responsáveis, principalmente, pela manutenção de um espaço de tuplas distribuído, e pelo mapeamento, neste espaço, das classes de tuplas identificadas em tempo de ligação. As estratégias usadas para implementar o espaço de tuplas distribuído são baseadas em [BJO92]. Network Linda não emprega uma única política de distribuição de tuplas pois, como será visto a seguir, informações coletadas tanto em tempo de compilação como de execução auxiliam o sistema a escolher a melhor política de distribuição para cada classe de tuplas.

Network Linda não emprega processos especiais para gerência do espaço de tuplas, ao contrário dos outros sistemas apresentados. Cada processo usuário age também como um servidor computacional (que executa operações **eval**) e como um servidor de tuplas, responsável pelo gerenciamento de uma porção disjunta do espaço de tuplas[NEL92, CAR94]. Como aplicações Network Linda são essencialmente SPMD, cada nodo disponível da rede executa uma instância do mesmo processo usuário. Em Network Linda, somente processos pertencentes a uma mesma aplicação podem compartilhar o mesmo espaço de tuplas.

Em tempo de execução, cada classe de tuplas é mapeada sobre um determinado nodo, de maneira que o processo usuário residente neste nodo fica responsável

por armazenar as tuplas desta classe e por atender todas as operações que manipulam estas tuplas. Em princípio, toda tupla (ou *template*), ao ser gerada, é enviada para o nodo associado a sua classe. Um nodo escolhido para gerenciar uma determinada classe de tuplas é denominado "nodo de *rendezvous*", já que serve como um ponto de encontro para *templates* e tuplas (*templates* esperam neste nodo por uma tupla equivalente, e vice versa).

As primitivas de manipulação de tuplas são implementadas diretamente através do mecanismo de *sockets* do Unix, usando o protocolo de rede UDP (*User Datagram Protocol*). Para implementação da primitiva **out**, a tupla é enviada ao nodo de *rendezvous* da classe correspondente, como mostra a figura 3.5(a). Já para as primitivas **in** e **rd**, o nodo requisitante envia o *template* ao nodo de *rendezvous* e espera pela resposta, que retorna tão logo a tupla requisitada esteja disponível (ver figura 3.5(b)).

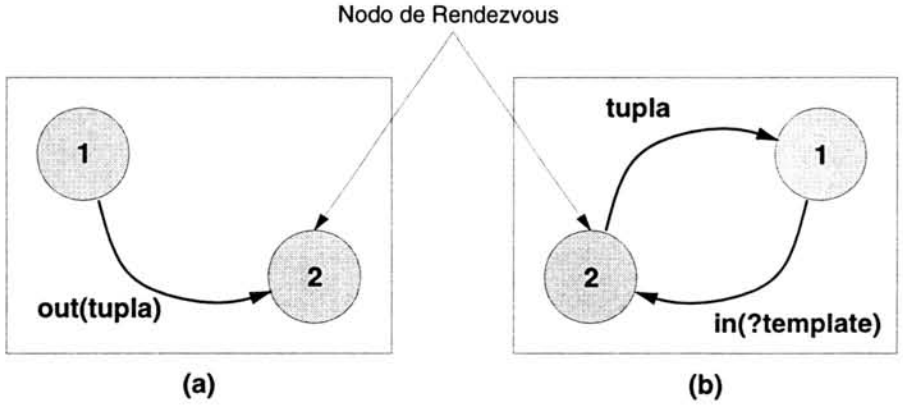

Figura 3.5 - Implementação de **in** e **out** em Network-Linda.

Quando as primitivas manipulam tuplas que incluem estruturas muito grandes, entretando, sua implementação é diferenciada. Para evitar a transferência de uma grande quantidade de dados pela rede, o nodo de *rendezvous* recebe somente uma descrição da tupla, suficiente para garantir que esta seja recuperada corretamente. Ao receber um *template* equivalente, o nodo de *rendezvous* solicita que o nodo de origem da tupla envie os dados diretamente ao nodo que gerou o *template*, como pode ser visto na figura 3.6. Isto é possível porque, segundo [CAG93], a recuperação de uma tupla muito grande geralmente não é baseada nos dados nela armazenados (campos reais), mas sim na descrição destes dados (campos formais). Operações com tuplas grandes que fogem a esta regra podem ser identificadas em tempo de ligação, e neste caso são tratadas como se manipulassem tuplas menores (figuras 3.5(a) e 3.5(b)).

Network Linda inclui algumas otimizações para garantir eficiência em tempo de execução. Em primeiro lugar, classes implementadas como tabelas *hash* podem ser gerenciadas por vários nodos de *rendezvous*, cada um ficando responsável por um segmento da tabela. Isto evita que um único nodo seja sobrecarregado e se torne um gargalo no sistema. Em segundo lugar, quando é possível prever, em tempo de ligação, que algumas tuplas serão acessadas por muitos nodos, pode-se fazer uma difusão destas tuplas para estes nodos, a fim de reduzir seu tempo de acesso. Por fim, com base na observação do tráfego de tuplas em tempo

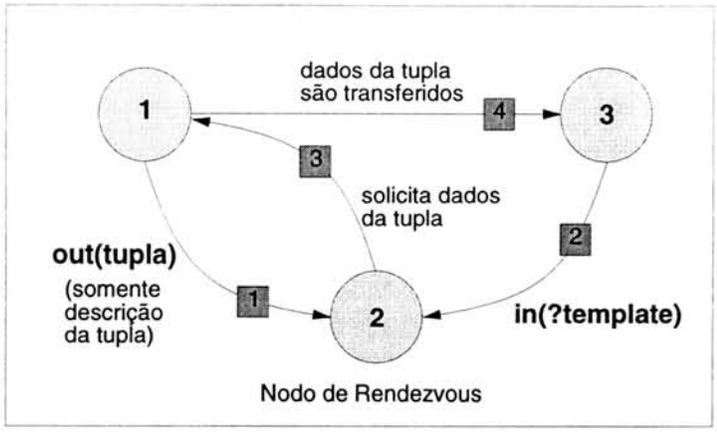

Figura 3.6 - Protocolo Network-Linda para manipulação de tuplas grandes.

de execução, é possível mapear dinamicamente uma classe de tuplas em outro nodo de *rendezvous*. Por exemplo, se um determinado processo constantemente requisita tuplas de uma determinada classe, o nodo de *rendezvous* para esta classe pode, automaticamente, passar a ser o nodo onde este processo reside. Assim, grande parte das tuplas podem ser enviadas diretamente para o nodo onde serão consumidas.

Questões relativas à implementação de **eval** são pouco discutidas na bibliografia sobre Network Linda. Por tratar-se de um produto comercial, é compreensível que alguns detalhes de implementação não sejam divulgados publicamente. Segundo [MAT96], **eval** é a primitiva menos portável em Network Linda, pois sua implementação pode envolver detalhes que variam de acordo com cada arquitetura e sistema operacional suportados. Pelas informações obtidas em [MAT96], sabe-se que os processos disparados por **ntsnet** são idênticos, mas que somente um deles executa a função principal da aplicação. Os processos restantes permanecem inativos, esperando que exista uma chamada **eval** a processar. O critério usado para escolher um processo deve computar uma chamada **eval**, e o mecanismo usado para ativar este processo, no entanto, são detalhes não completamente esclarecidos. Uma das possíveis implementações de **eval** [HUP91] consiste em armazenar no próprio espaço de tuplas o nome da função a ser computada. Neste caso, um processo é ativado ao retirar do TS uma tupla contendo uma função a ser avaliada. Como os processos são todos idênticos, qualquer processo está apto a executar qualquer função especificada numa chamada **eval**.

3.4 Sumário

Neste capítulo foram discutidas várias questões relacionadas ao projeto de um sistema baseado em Linda. Estas questões foram divididas em dois grupos: aquelas relacionadas ao **ambiente de programação** destes sistemas, que basicamente dizem respeito à variedade de recursos oferecidos aos usuários programadores, e aquelas referentes ao **ambiente de execução**, que tratam dos mecanismos capazes de suportar a execução de aplicações segundo o paradigma proposto por

Linda. Com relação ao ambiente de programação, as questões discutidas foram as seguintes:

- **Estrutura das Aplicações Paralelas**: um sistema baseado em Linda pode exigir que uma aplicação paralela seja estruturada num modelo SPMD (*Single Program Multiple Data*) ou MPMD (*Multiple Program Multiple Data*), ou ainda permitir que o usuário selecione o modelo mais adequado a cada aplicação;

- **Conjunto de Primitivas**: sistemas baseados em Linda sempre oferecem primitivas para manipulação de tuplas, mas podem introduzir algumas modificações na interface destas primitivas. Além disso, muitos sistemas modificam a semântica da primitiva **eval**, e as primitivas **inp** e **rdp** são muitas vezes excluídas do conjunto de primitivas oferecido;

- **Biblioteca** $\times$ **Pré-Compilador**: para incorporar o modelo Linda a uma linguagem seqüencial, pode-se simplesmente reunir as primitivas em uma biblioteca ou, adicionalmente, pode-se implementar um pré-processador ou um pré-compilador, que permitem oferecer uma interface mais simples ao usuário;

- **Ferramentas Adicionais**: um ambiente de programação pode incluir ferramentas que auxiliam o programador em tarefas como, por exemplo, depuração de programas paralelos e avaliação do desempenho destes programas;

Com relação ao ambiente de execução de um sistema baseado em Linda, as questões discutidas se resumiram a:

- **Política de Distribuição de Tuplas**: numa arquitetura sem memória compartilhada, a política de distribuição de tuplas é fundamental para coordenar a implementação do compartilhamento lógico do espaço de tuplas. As políticas mais freqüentemente empregadas diferem entre si em vários aspectos, como complexidade de implementação, grau de distribuição, extensibilidade e requisitos de memória. A escolha de uma política deve levar em conta as características da arquitetura onde o sistema deve executar e, se possível, os tipos de aplicações que serão suportadas;

- **Grau de Compartilhamento do Espaço de Tuplas**: o compartilhamento do espaço de tuplas é, a princípio, garantido para os processos que compõem uma mesma aplicação. Existem sistemas, entretanto, que permitem este compartilhamento entre processos de diferentes aplicações, pertencentes até mesmo a usuários diferentes;

- **Implementação de EVAL, INP e RDP**: estas primitivas constituem um aspecto crítico na implementação de sistemas baseados em Linda. No caso de **eval**, sua semântica é de difícil implementação, envolvendo vários detalhes que não estão explícitos no modelo. No caso de **inp** e **rdp**, sua implementação eficiente pode ser dificultada dependendo da política de distribuição de tuplas utilizada;

- **Suporte a Heterogeneidade**: no caso de sistemas destinados à programação paralela em redes de computadores, é interessante que uma aplicação possa ser executada em um conjunto heterogêneo de máquinas, embora isto implique em uma dificuldade adicional de implementação e, ao mesmo tempo, tenha uma influência negativa no desempenho dos sistemas.

Apesar desta divisão das questões de projeto em dois grupos, deve-se notar que muitas decisões a nível de ambiente de execução, por exemplo, podem influenciar ou serem visíveis no ambiente de programação, sendo a recíproca também verdadeira. A decisão de implementar as primitivas **inp** e **rdp**, por exemplo, é geralmente baseada em questões relacionadas ao ambiente de execução, mas isto obviamente se reflete no ambiente de programação, já que o conjunto de primitivas é modificado. Outro exemplo é a opção pelo uso de uma biblioteca, um pré-processador ou um pré-compilador, que foi tratada como questão relacionada ao ambiente de programação porque pode interferir fortemente na interface do sistema com o usuário. No entanto, esta decisão se reflete também no ambiente de execução, já que, por exemplo, a existência de um pré-compilador permite a implementação de rotinas mais otimizadas nas bibliotecas dos sistemas.

Este capítulo também apresentou um estudo sobre os sistemas Glenda, POSYBL, p4-Linda e Network-Linda, que implementam o modelo Linda para programação paralela em redes de computadores. O objetivo deste estudo foi fornecer uma idéia clara de como cada um destes sistemas lida com as questões identificadas no início do capítulo, tanto a nível de projeto como a nível de implementação. Nas tabelas 3.1 e 3.2 estão resumidas as principais soluções que cada um destes sistemas adota com relação a seus ambientes de programação e execução.

Tabela 3.1 - Resumo das características do ambiente de programação de cada sistema estudado.

Sistemas	Estrutura das Aplicações	EVAL	INP/ RDP	Incorporação à Ling. Hospedeira
Glenda	SPMD/MPMD	substituída por **gl_spawn**, que usa PVM para criação de processos	sim	pré-processador
POSYBL	MPMD	dispara a execução de um novo programa	não	biblioteca
p4-Linda	SPMD	executa função em paralelo porém sem inserção de tupla	não	biblioteca
Network-Linda	SPMD	semântica original	sim	pré-compilador

Na tabela 3.2 é possível identificar duas importantes questões de implementação que foram abordadas no decorrer de cada estudo de caso, sobre as quais é conveniente fazer-se algumas observações:

- **Mecanismo de Comunicação entre Processos**: a comunicação entre processos através da rede é fundamental para a implementação de um ambiente de execução distribuído. Pelo estudo realizado, pôde-se concluir

que existem basicamente duas soluções para isso: utilizar alguma ferramenta para programação baseada em troca de mensagens, o que facilita a implementação do ambiente de execução, ou utilizar os recursos do próprio sistema operacional hospedeiro para implementar a comunicação entre processos, o que permite a construção de um ambiente de execução mais eficiente, mas menos portável.

- **Processo Gerenciador do TS**: para o gerenciamento do espaço de tuplas, a maioria dos sistemas emprega um ou mais processos servidores, que atendem a requisições de processos clientes usuários das primitivas Linda. No entanto, também é possível implementar esta funcionalidade nos próprios processos usuários, o que é uma solução mais eficiente à medida que permite diminuir a quantidade de mensagens trocadas entre processos.

Tabela 3.2 - Resumo das características do ambiente de execução de cada sistema estudado.

Sistemas	**Política de Distribuição**	**Suporte à Heterogeneidade**	**Mecanismo de Comunicação entre Processos**	**Processo Gerenciador do TS**
Glenda	centralizada	sim	PVM	sim
POSYBL	uniforme	limitado	RPC, *sockets*	sim
p4-Linda	centralizada	sim	p4	sim
Network-Linda	várias políticas	sim	*sockets*	não

4 O Ambiente YALI

Este capítulo trata do projeto de YALI (*Yet Another Linda Implementation*), um sistema baseado no modelo Linda destinado à programação e execução de aplicações paralelas em redes de computadores. Como visto no capítulo anterior, o projeto de um sistema baseado em Linda envolve uma grande variedade de questões. Em YALI, a busca de soluções para estas questões foi norteada por alguns objetivos preliminares, que são descritos na seção 4.1. Logo a seguir, a seção 4.2 trata do ambiente de programação de YALI, descrevendo algumas decisões de projeto à medida que apresenta o conjunto de recursos do sistema para o desenvolvimento de aplicações paralelas. Por fim, a seção 4.3 apresenta as principais soluções adotadas no projeto do ambiente de execução de YALI. Maiores detalhes sobre o ambiente de execução são fornecidos no capítulo seguinte, que trata da implementação de YALI.

4.1 Objetivos Preliminares

Os objetivos preliminares discutidos a seguir serviram para adotar um conjunto coerente de soluções para as várias questões envolvidas no projeto de YALI.

4.1.1 Facilidade de Aprendizado e Uso

Uma das preocupações mais fortes no projeto de YALI foi com relação à facilidade de aprendizado e uso do sistema. Desejava-se projetar uma interface facilmente assimilável e utilizável, até mesmo por usuários inexperientes, iniciantes na tarefa de desenvolvimento de aplicações paralelas. O modelo Linda foi escolhido por apresentar várias vantagens sob este ponto de vista. Em primeiro lugar, o paradigma de memória compartilhada torna transparente vários aspectos inerentes a uma plataforma distribuída. Além disso, o pequeno número de primitivas do modelo, aliado à possibilidade de incorporá-las a uma linguagem sequencial tradicional, permite que o modelo (ou os sistemas que implementam o modelo) sejam facilmente assimilados. O mesmo não se pode dizer com relação a ambientes baseados em troca de mensagens que, além de utilizarem um paradigma de mais baixo nível, geralmente oferecem um grande número de primitivas que precisam ser dominadas (p4, por exemplo, oferece 28 funções, enquanto PVM oferece cerca de 53).

Antes de optar-se por desenvolver um sistema baseado no modelo Linda, a possibilidade de projetar-se uma nova linguagem para programação paralela também foi considerada. Esta alternativa, no entanto, foi abandonada por dois motivos. Em primeiro lugar, o projeto e implementação de uma linguagem é uma tarefa que poderia exigir um cronograma maior do que o disponível. Em segundo lugar, esta alternativa não pode ser considerada ideal sob o ponto de vista da

facilidade de aprendizado, já que a assimilação de uma nova linguagem pode ser uma atividade demorada, especialmente para usuários pouco experientes.

4.1.2 Heterogeneidade

Como não há um conjunto de *hardware* e *software* que sirva a todos os propósitos computacionais igualmente bem[GEI90], a heterogeneidade em sistemas de computação vem se tornando uma realidade cada vez mais marcante. Esta "diversidade intrínseca aos ambientes computacionais"[BAR93] pode manifestar-se, por exemplo, a nível de arquitetura, de sistema operacional ou até a nível de interfaces e aplicações. Uma análise de vários aspectos relacionados à heterogeneidade em sistemas de computação distribuídos pode ser encontrado em [BAR93].

Considerando esta realidade heterogênea, outro objetivo preliminar no projeto de YALI foi oferecer ao usuário a possibilidade de utilizar máquinas de arquiteturas potencialmente diferentes para a execução de aplicações paralelas, permitindo assim um melhor aproveitamento dos recursos computacionais disponíveis. O projeto do sistema, no entanto, não tem como objetivo explorar automaticamente a heterogeneidade para aumentar a eficiência na execução de aplicações, embora esta alternativa seja bastante interessante para trabalhos futuros.

4.1.3 Desempenho

Sistemas para programação de aplicações paralelas sempre têm como objetivo aumentar o desempenho destas aplicações ou, em outras palavras, diminuir seu tempo de execução. Uma das medidas de desempenho mais usadas é o *speedup*, que é dado por

$$sp = \frac{T_s}{T_p}$$

onde T_s e T_p são, respectivamente, os tempos de execução seqüencial e paralela de uma aplicação. Para uma aplicação executada em N processadores, o *speedup* ideal é igual a N, ou seja, espera-se que a aplicação execute N vezes mais rápido. No entanto, o *speedup* ideal é difícil de obter porque o tempo de execução paralela sempre inclui o tempo gasto pela aplicação em operações de comunicação e sincronização. Nisto está incluído, por exemplo, o tempo que uma mensagem leva para trafegar entre duas máquinas numa rede, e também o tempo que o próprio ambiente de execução dispende na implementação destas operações. No projeto de YALI, houve uma preocupação em adotar alternativas que minimizassem este *overhead* no ambiente de execução, a fim de garantir um desempenho razoável na execução de aplicações paralelas.

4.2 Ambiente de Programação

Como já se comentou anteriormente, o projeto do ambiente de programação de YALI foi bastante voltado à facilidade de aprendizado e uso do sistema. Uma das decisões influenciadas por esta preocupação foi a escolha da linguagem C para servir como hospedeira do modelo Linda. Esta linguagem é amplamente utilizada em aplicações seqüenciais de propósito geral, e existem compiladores C disponíveis para um variado conjunto de arquiteturas e sistemas operacionais. Com a escolha de uma linguagem bastante difundida, é provável que muitos usuários de YALI já tenham experiência de programação em C, ficando o aprendizado do sistema limitado às novas primitivas e a algumas noções de programação paralela.

Além da escolha da linguagem hospedeira, outras questões de projeto também foram solucionadas levando-se em conta a facilidade de aprendizado e uso de YALI. Sempre que possível, procurou-se também adotar alternativas que oferecessem maior flexibilidade ao programador. As seções seguintes apresentam o projeto completo do ambiente de programação de YALI, descrevendo as soluções adotadas para as questões identificadas na seção 3.1 do capítulo anterior.

4.2.1 Estrutura das Aplicações

Para oferecer maior flexibilidade ao usuário, uma aplicação paralela em YALI pode ser estruturada tanto em um modelo SPMD como MPMD. O usuário pode optar por codificar um só programa e replicá-lo sobre diversos nodos de uma rede, ou então decidir pela decomposição da aplicação em vários módulos, que também são distribuídos sobre os nodos da rede. Como será visto nas próximas seções, YALI provê um conjunto de primitivas adequado a ambos os modelos de estruturação de aplicações, e também fornece meios de replicar ou distribuir um ou mais programas sobre uma rede de computadores.

Com relação à estrutura de um programa YALI, esta é praticamente idêntica à estrutura de um programa C. As primitivas YALI são usadas como funções em C, como na maioria das implementações do modelo. A única diferença é que a função principal de um programa YALI deve se chamar `yali_main`, em substituição à função `main` usada em programas C.

Uma consideração importante sobre aplicações paralelas em geral é com relação à sua terminação. Como uma aplicação é composta por múltiplos processos, o término de um único processo não implica no fim da aplicação como um todo. Em YALI, portanto, foi definido que uma aplicação só deve terminar quando todos os seus processos terminarem a execução de suas funções principais (`yali_main`).

4.2.2 Conjunto de Primitivas

YALI disponibiliza ao usuário um conjunto de primitivas bastante semelhante àquele definido em Linda, principalmente no que diz respeito a primitivas para manipulação de tuplas, através das quais se dá a comunicação e a sincronização entre processos. Com relação à expressão do paralelismo, entretanto, YALI oferece primitivas cuja semântica difere significativamente daquela proposta no modelo Linda para a primitiva **eval**. Além disso, YALI estende o modelo com primitivas que facilitam a expressão de algumas operações globais, usadas para implementar a comunicação e a sincronização entre múltiplos processos. A seguir tem-se uma descrição detalhada do conjunto de primitivas YALI, organizadas de acordo com os aspectos de expressão do paralelismo, comunicação e sincronização, e operações globais.

Expressão do Paralelismo

Em YALI, a expressão do paralelismo se dá através das primitivas descritas logo a seguir, que permitem a criação dinâmica de *threads* em qualquer processo de uma aplicação. O uso de *threads* permite expressar o paralelismo de granularidade mais fina, possibilitando a execução paralela tanto de funções simples como de rotinas mais complexas.

y_eval

Esta primitiva serve para executar uma função em paralelo com o processo que a implementa. Para isso, o endereço da função é fornecido como parâmetro para **y_eval**, que dispara uma nova *thread* para executá-la. Ao contrário de **eval** no modelo Linda, esta primitiva não está automaticamente associada à inserção de tuplas. Quando isto for necessário, a própria função que é executada em paralelo deve encarregar-se de depositar uma tupla resultante no TS. Para ilustrar o uso desta primitiva, considere o exemplo abaixo:

```
void func() {
    /* corpo da funcao omitido */
}

yali_main() {
    ...
    y_eval((void (*)()) func);
    ...
}
```

Neste exemplo sintético, **y_eval** é utilizada para disparar a execução da função `func` em paralelo, o que permite, por exemplo, que `func` utilize operações bloqueantes sem impedir que outras operações independentes continuem a ser executadas pela função principal. Pelo exemplo acima,

pode-se notar que **y_eval** espera que a função passada como parâmetro seja do tipo `void`. A conversão de tipo na chamada **y_eval** acima, no entanto, é meramente ilustrativa, pois `func` já havia sido previamente declarada como `void`.

Através de sucessivas chamadas a **y_eval** é possível criar várias *threads* dentro de um mesmo processo. Todas as *threads* compartilham o mesmo espaço de endereçamento, incluindo dados globais do processo. Esta característica encoraja o uso de variáveis globais para armazenar dados que precisam ser compartilhados por múltiplas *threads*, a fim de permitir a cooperação entre elas. Esta prática, no entanto, não é prevista em YALI, uma vez que o sistema não oferece primitivas específicas para controlar o acesso das *threads* aos dados compartilhados. O ideal é que cada função disparada por **y_eval** se limite a utilizar apenas dados locais a ela, e que dados globais sejam sempre depositados no espaço de tuplas. Do contrário, a integridade dos dados compartilhados não é garantida pelo sistema.

y_global

Um processo utiliza esta primitiva para associar um nome global a uma função por ele implementada. O propósito desta associação é permitir que, posteriormente, a execução da função possa ser disparada por qualquer processo que conheça seu nome global, mesmo que este processo resida em outra máquina da rede. A execução de **y_global**, por si só, não tem efeito nenhum sobre a execução da função (isto é feito através de outra primitiva do sistema, descrita a seguir).

Como parâmetro para **y_global** devem ser fornecidos o nome global (uma cadeia de caracteres) e o endereço da função, como mostra o exemplo abaixo:

```
void other_func() {
    /* corpo da funcao omitido */
}

yali_main() {
    ...
    y_global("func", other_func);
    ...
}
```

Assim como **y_eval**, **y_global** requer que a função que recebe o nome global seja do tipo `void`. É possível que vários processos associem um mesmo nome global a uma determinada função, permitindo que sua execução seja disparada em qualquer um dos processos que a implementam. Isto pode ocorrer, por exemplo, em aplicações SPMD, onde vários processos executam o mesmo programa e implementam as mesmas funções. A vantagem disto é permitir que chamadas sucessivas a uma mesma função global sejam atendidas por processos diferentes, evitando a sobrecarga de um processo em particular. O usuário, no entanto, deve evitar que processos diferentes

definam nomes idênticos para funções distintas, pois neste caso não se pode garantir qual processo será escolhido e, conseqüentemente, qual função será executada.

y_globeval

O objetivo desta primitiva é disparar uma ou mais *threads* para executar uma função associada a um determinado nome global. As *threads* são criadas no processo que implementa a função, o qual pode residir em qualquer máquina da rede e, provavelmente, não é o mesmo processo que executa **y_globeval**. Pode-se dizer, portanto, que esta primitiva permite a criação de *threads* à distância.

Um mesmo nome global pode referenciar funções implementadas por diferentes processos. Isto pode facilmente ocorrer quando são usados processos trabalhadores replicados, por exemplo. Nesta situação, quaisquer das funções associadas ao nome global poderão ser escolhidas para execução, de modo que as *threads* podem vir a ser disparadas em diferentes processos da aplicação. Ao contrário, se nenhuma função está associada a um determinado nome global, a criação da *thread* é postergada até que algum processo faça esta associação (se isto não acontecer, a *thread* nunca será realmente criada). A execução de **y_globeval** é sempre assíncrona, de modo que o processo que executa esta operação prossegue mesmo que as *threads* não tenham sido de fato criadas.

O uso de **y_globeval** é ilustrado através do exemplo a seguir:

```
processo A:
void my_func() {
    /* corpo da funcao omitido */
}
yali_main() {
    ...
    y_global("func", my_func);
    ...
}

processo B:
yali_main() {
    ...
    y_globeval(1, "func");
    ...
}
```

Neste exemplo, o processo A implementa a função my_func e declara um nome global para ela. O processo B, por sua vez, utiliza **y_globeval** para disparar uma *thread* em A, que passa a executar my_func. Pode-se comparar o funcionamento de **y_globeval** com uma chamada remota de procedimento, onde o servidor é o processo que executa **y_global**. Ao contrário de um mecanismo de RPC, entretanto, **y_globeval** não permite a passagem

de parâmetros ao procedimento remoto, e também não bloqueia esperando por resultados. Qualquer comunicação entre o cliente e o servidor deve ser feita através do espaço de tuplas.

A existência de uma *thread* limita-se ao tempo necessário para executar a função especificada, embora uma *thread* possa dar origem a várias outras *threads*. O término de uma *thread* ocorre quando a função que ela executa chega ao fim, ou quando o comando `return` é utilizado. O usuário, no entanto, nunca pode utilizar o comando `exit` num programa YALI, pois ele não causa o término consistente da aplicação paralela. Ainda com relação à terminação de *threads*, deve-se ressaltar que o único meio de uma *thread* esperar pelo término de outra é através de primitivas de comunicação e sincronização, descritas logo a seguir. Isto é particularmente importante para as *threads* que executam funções `yali_main` em cada processo, já que, se todas elas terminam sem esperar por outras *threads*, a aplicação pode encerrar sem que todas as suas *threads* tenham terminado seu processamento. O usuário, portanto, deve garantir a correta sincronização das *threads* da aplicação, a fim de evitar uma situação semelhante à recém mencionada.

Pelas primitivas descritas acima, nota-se que YALI não oferece ao usuário meios de criar processos dinamicamente. No entanto, processos podem ser criados em qualquer quantidade durante a inicialização de uma aplicação paralela, quando são distribuídos estaticamente sobre um conjunto disponível de máquinas da rede. Após esta fase de inicialização, YALI só permite a criação de *threads*. Isto, no entanto, não constitui uma limitação muito forte, já que é possível criar *threads* para executar funções bastante complexas, que por sua vez podem criar tantas *threads* quantas forem necessárias.

O paralelismo expresso através de *threads* não se restringe apenas às novas *threads* criadas com **y_eval** ou **y_globeval**. A própria função principal de um processo (chamada `yali_main`), é executada por uma *thread*. Este uso massivo de *threads* modifica ligeiramente o escopo das primitivas que promovem comunicação e sincronização em YALI, porque este tipo de interação ocorre, na verdade, entre *threads*. Quando uma primitiva bloqueante é utilizada, por exemplo, não é todo o processo que bloqueia, mas somente a *thread* que utiliza a primitiva. Ao longo deste capítulo, toda a utilização do termo "processo" deve ser entendida como "uma *thread* do processo", sempre que se referir a algum tipo de interação que ocorre em tempo de execução numa aplicação YALI.

Comunicação e Sincronização

Em YALI, a comunicação e a sincronização entre processos são expressas exatamente como no modelo Linda, isto é, através de primitivas de manipulação de tuplas. As seguintes primitivas estão disponíveis em YALI:

- **y_out**, para inserção de tuplas;
- **y_in**, para recuperação de tuplas seguida de remoção;

- **y_rd**, para recuperação de tuplas somente;
- **y_inp**, versão não-bloqueante de **y_in** e
- **y_rdp**, versão não-bloqueante de **y_rd**.

A primitiva **y_out** é assíncrona, de modo que a *thread* que a executa continua mesmo que a tupla não tenha sido de fato inserida no espaço de tuplas. As demais primitivas são síncronas, isto é, dependem do resultado de uma consulta ao espaço de tuplas. No caso de **y_in** e **y_rd**, o resultado desta consulta deve ser necessariamente uma tupla, e por isso estas primitivas bloqueiam até que uma tupla apropriada esteja disponível. As primitivas **y_inp** e **y_rdp**, no entanto, são ditas não-bloqueantes porque liberam a *thread* que as utiliza quando a tupla solicitada não está disponível.

Os parâmetros fornecidos a estas primitivas são tuplas ou *templates* com uma estrutura bastante semelhante àquela definida em Linda. As únicas diferenças são a exigência de que o primeiro campo de uma tupla ou *template* seja sempre real, e a impossibilidade de utilização de campos formais em operações **y_out**. Ambas as modificações foram introduzidas por motivos relacionados à implementação do sistema, que será discutida no próximo capítulo.

Em YALI, os campos usados nas tuplas e *templates* podem ser de seis tipos básicos, definidos na linguagem C: `char`, `int`, `short`, `long`, `float` e `double`. Também são permitidos campos formados por vetores destes tipos básicos. Os tipos dos campos são identificados automaticamente por um pré-processador (discutido na seção 4.2.3 mais adiante), o que simplifica a interface com o usuário. A seguir tem-se alguns exemplos simples de manipulação de tuplas em YALI:

```
int   i = 10;
int   j = 20;
float x = 2.71;

y_out(i, j, 32);
y_out("num", x);
y_in("mutex");
```

Campos formais, permitidos em *templates* fornecidos a **y_in**, **y_rd**, **y_inp** e **y_rdp**, devem ser precedidos pelo símbolo '**?**'. Campos reais compostos por vetores alocados dinamicamente devem ter seu número de elementos (tamanho do vetor) explicitamente fornecido nas operações, após um símbolo '**:**'. No caso de campos formais que devem receber vetores, o símbolo '**:**' deve ser precedido por uma variável inicializada com o tamanho máximo do vetor a ser recuperado. Se uma tupla com campos equivalentes é encontrada, esta variável recebe o número de elementos do vetor de fato recuperado. O uso de campos formais e vetores é ilustrado no trecho de programa a seguir:

```
#define N 10

int i, size, a1[20], *a2, *a3;

for (i=0; i<20; i++)      /* preenche vetor a1 */
    a1[i] = i;

y_out("head", a1[0]);    /* insere primeiro elemento de a1 */
y_out("array", a1);      /* insere vetor a1 */

a2 = (int *) malloc(N * sizeof(int)); /* aloca espaco para vetor a2 */
for (i=0; i<N; i++)                   /* preenche vetor a2 */
    a2[i] = i*i;
y_out("another_array", a2:N);         /* insere vetor a2 */

y_in("head", ?i);         /* recupera tupla com primeiro elemento de a1 */

size = 20;
a3 = (int *) malloc(size * sizeof(int));
y_rd("array", ?a3:size);   /* recupera vetor a1, armazenando-o em a3 */
```

Acima, o vetor `a1` tem seu tamanho previamente declarado, e por isso esta informação não precisa ser fornecida explicitamente na operação **y_out** que insere `a1` no espaço de tuplas. Se, no entanto, fosse desejado incluir no TS apenas parte do vetor `a1`, a especificação de tamanho poderia ser usada para limitar o número de elementos inseridos. Por outro lado, o tamanho do vetor `a2`, que é alocado dinamicamente, deve ser necessariamente especificado na operação **y_out** que o insere no espaço de tuplas. O restante do exemplo ilustra o uso de campos formais, iniciando pela operação **y_in** que recupera uma tupla contendo o primeiro elemento do vetor `a1`, e atribui o valor deste elemento à variável `i`. A seguir, a operação **y_rd** recupera a tupla contendo o próprio vetor `a1`, armazenando-o em `a3`. Nesta operação, a variável `size` especifica o tamanho máximo do vetor a ser recuperado e, após a recuperação, contém o número de elementos copiados para `a3`.

Embora os exemplos fornecidos acima utilizem apenas **y_in** e **y_rd** para recuperação de tuplas, todas as observações sobre campos reais e formais se aplicam também às operações não-bloqueantes **y_inp** e **y_rdp**. Como definido em Linda, estas primitivas retornam 0 quando nenhuma tupla equivalente é encontrada no TS, e retornam 1 em caso contrário (neste caso, comportam-se exatamente como **y_in** e **y_rd**). A inclusão destas primitivas em YALI teve como objetivo aumentar a expressividade do sistema. O uso de **y_inp** e **y_rdp** facilita, por exemplo, a simulação de estruturas do tipo `select`, utilizadas quando um processo precisa esperar por mais de um tipo de tuplas. Outra aplicação de primitivas não-

bloqueantes é no controle de laços, como mostra o exemplo abaixo:

```
worker() {

    int i;
    int sum = 0;
    int count = 0;

    while (!y_inp("value", ?i)) {
        sum += i;
        count++;
    }

    /* segunda fase */

}
```

Neste exemplo, a *thread* que executa a função `worker` deve repetidamente recuperar tuplas contendo a cadeia de caracteres `"value"` seguida de um valor inteiro. Este valor é utilizado numa computação simples dentro do laço `while`, que termina quando não houverem mais tuplas equivalentes disponíveis no TS. A seguir inicia-se uma segunda fase da computação, onde provavelmente são usados os resultados produzidos na primeira fase (laço `while`). Se ao invés de **y_inp** fosse usada uma primitiva bloqueante, este código não mais poderia ser usado, pois, ao final da primeira fase (ausência de tuplas equivalentes no TS), a *thread* ficaria bloqueada e nunca iniciaria a segunda fase da computação. É possível reescrever este código utilizando-se primitivas não-bloqueantes, como mostra [LEI89], mas em geral as alternativas levam a um código mais complexo e ao uso de tuplas adicionais para controlar a própria recuperação de tuplas.

A disponibilidade de primitivas não-bloqueantes, portanto, aumenta a expressividade de YALI à medida que permite codificar certas técnicas de programação de maneira mais simples. Além disso, a implementação destas primitivas em YALI tem um custo de comunicação semelhante ao das outras primitivas do sistema, como será visto no próximo capítulo. O usuário que optar pela utilização destas primitivas, no entanto, deve estar ciente de que normalmente elas introduzem uma noção de sincronia global nas aplicações onde são empregadas. Isto pode ser entendido observando-se o exemplo acima, que só funciona se uma *thread* depositar tuplas no TS **antes** da execução de **y_inp**. Caso nenhuma tupla esteja disponível no momento em que a primeira chamada **y_inp** é executada, o laço termina sem recuperar uma tupla sequer, o que pode prejudicar a próxima fase da computação. Em casos como este, o usuário é responsável por sincronizar adequadamente as *threads* produtoras (que depositam tuplas) e consumidoras (que recuperam tuplas através de **y_inp** ou **y_rdp**).

Operações Globais

Aplicações paralelas freqüentemente necessitam expressar a comunicação e a sincronização entre múltiplos processos (ou múltiplas *threads*). Por isso, muitos ambientes de programação oferecem operações globais, capazes de facilitar e otimizar este tipo de interação. Pode-se identificar vários tipos de operações globais:

- **Difusão**: operações deste tipo são usadas para enviar uma informação para muitos processos. Uma difusão pode ser feita para um grupo restrito de processos (comunicação um-para-muitos ou multiponto) ou para todos os processos de uma aplicação (comunicação um-para-todos ou *broadcast*).
- **Reunião**: é o contrário da difusão, isto é, permite que um único processo reúna informações provenientes de múltiplos processos. Também é chamada de comunicação muitos-para-um.
- **Redução**: é semelhante a uma operação de reunião, porém as informações reunidas são processadas a fim de produzir um resultado único.
- **Barreira**: é uma operação que permite sincronizar a execução de múltiplos processos. Uma barreira pode ser vista como um ponto de encontro, onde os processos que executam em paralelo esperam uns pelos outros antes de iniciar uma nova fase de computação.

Qualquer destas operações pode ser definida igualmente substituindo-se o termo "processo" por "*thread*". O modelo Linda não oferece operações deste tipo, mas permite que o próprio usuário as implemente. Na seção 2.3.2 já foi visto que a difusão de mensagens é facilmente implementada em Linda, e em 2.3.3 foi explicado como implementar barreiras através de uma combinação de **out**, **in** e **rd**. Além disso, operações de reunião e redução também podem ser implementadas em Linda. Para que um único processo reúna informações geradas por múltiplos processos (operação de reunião), basta executar repetidamente a primitiva **in** com um determinado *template*, como mostra o exemplo abaixo:

```
int i, n;

for (i=0; i<10; i++) {
    in("num", ?n);
}
```

Embora não esteja explícito neste exemplo, cada tupla recuperada através de **in** pode ter sido gerada por um processo diferente, o que caracteriza uma comunicação muitos-para-um. Uma operação de redução, por sua vez, pode ser implementada simplesmente através do processamento dos dados coletados com

in no exemplo acima. Reescrevendo este exemplo, teríamos:

```
int i, n;
int sum = 0;

for (i=0; i<10; i++) {
    in("num", ?n);
    sum += n;
}
```

No final da execução deste trecho de programa, os valores coletados para a variável n são reduzidos a um único valor, armazenado em sum. Neste exemplo, uma operação de adição foi usada para produzir o valor final, mas a redução poderia ser feita através de qualquer outra operação condizente com o tipo dos dados recuperados.

Operações globais são bastante usadas em vários tipos de aplicações paralelas. Em aplicações que seguem o paradigma mestre/trabalhador, por exemplo, operações de reunião ou redução são frequentemente utilizadas pelo processo mestre, a fim de recolher os resultados produzidos pelos trabalhadores. O uso de barreiras, por outro lado, é comum em aplicações onde a computação paralela é dividida em fases, onde cada fase deve ser iniciada ao mesmo tempo por vários processos da aplicação. Embora seja possível implementar estas operações em Linda, a disponibilidade, em um sistema, de primitivas específicas para execução de operações globais tem pelo menos duas vantagens importantes. A primeira é o aumento da expressividade do sistema, que acaba levando a uma maior facilidade de uso, já que o usuário não precisa se preocupar em implementar ele próprio estas operações. A segunda, e talvez a mais importante, é o aumento do desempenho na execução destas operações. Isto fica evidenciado observando-se o número de chamadas necessárias para implementar barreiras, reuniões ou reduções em Linda: com o uso de primitivas específicas, reduz-se o número de chamadas e, conseqüentemente, aumenta-se o desempenho da aplicação.

Considerando as vantagens acima, e observando que a ausência de operações globais tem sido apontada como uma das limitações de Linda[DOU93, MAT94, MAT95], decidiu-se, no projeto de YALI, estender o modelo com três novas primitivas síncronas:

- **y_gather**, para reunião de tuplas;
- **y_reduce**, para redução de várias tuplas em uma, e
- **y_barrier**, para sincronização de barreira.

Uma das principais preocupações no projeto destas novas primitivas foi manter o nível de abstração proposto pelo modelo Linda, preservando características importantes como a ortogonalidade das operações e a possibilidade de interação entre processos independentemente da sua identificação ou localização. Por este

motivo, decidiu-se que uma primitiva para difusão de dados não deveria ser incluída, já que, pelo próprio paradigma de Linda, bastaria colocar uma informação no espaço de tuplas para que ela fosse acessível a vários processos através de uma operação **rd**. O funcionamento das primitivas projetadas é descrito detalhadamente a seguir.

y_gather

A primitiva **y_gather** recupera um determinado número de tuplas que satisfazem um mesmo *template* e, assim como **y_in**, também remove estas tuplas do TS. A ordem de recuperação de tuplas é arbitrária, e a *thread* que executa **y_gather** é bloqueada até que o número especificado de tuplas esteja disponível no TS. Com a execução de **y_gather**, cada campo formal do *template* recebe um **vetor** que reúne os valores dos campos correspondentes nas tuplas recuperadas. Para ilustrar o funcionamento desta primitiva, considere a operação abaixo:

```
int   a[4];
float v[4];

y_gather(4, "num", ?a, ?v);
```

Esta operação deve procurar no TS quatro tuplas com três campos, sendo o primeiro a cadeia de caracteres "num", o segundo um número inteiro e o terceiro um número de ponto flutuante. Caso existissem no TS as tuplas

```
("num", 1, 10.5)
("num", 4, 67.3)
("num", 2, 32.8)
("num", 5, 41.1)
("num", 3, 59.4)
```

a operação **y_gather** acima poderia recuperar qualquer combinação formada por quatro destas tuplas. Se fossem escolhidas, por exemplo, as duas primeiras e as duas últimas tuplas, o vetor a seria preenchido com os valores 1, 4, 5 e 3, e o vetor v receberia os valores 10.5, 67.3, 41.1 e 59.4. Pode-se notar que a ordem de preenchimento dos vetores é idêntica à ordem de recuperação das tuplas, ou seja, elementos com o mesmo índice (a[2] e v[2], por exemplo) são provenientes da mesma tupla.

Como YALI permite a existência de tuplas contendo vetores, **y_gather** deve também poder recuperar tais tuplas. Para ilustrar esta situação, considere o exemplo abaixo:

```
int   *p[3], s[3];
int   i, size = 4;

for (i=0; i < 3; i++)
    p[i] = (int *) malloc(sizeof(int) * size);

y_gather(3, "primes", ?s, ?p:size);
```

Neste exemplo, a operação **y_gather** deve procurar no TS três tuplas contendo a cadeia de caracteres "primes" no primeiro campo, um número inteiro no segundo campo, e um vetor de inteiros no terceiro campo. Cada vetor recuperado pode ter no máximo quatro elementos, conforme especificado através da variável `size` que segue o símbolo ':'. Esta variável é atualizada automaticamente com a execução de **y_gather**, de modo a indicar o tamanho do maior vetor de fato recuperado. Supondo que existissem no TS as tuplas

```
("primes", 3, <1, 2, 3>)
("primes", 3, <11, 13, 17>)
("primes", 2, <5, 7>)
("primes", 5, <23, 29, 31, 37, 39>)
```

somente as três primeiras poderiam ser recuperadas pela operação **y_gather** acima, já que a última contém um vetor com mais de quatro elementos. O vetor `s`, neste caso, é preenchido com os valores 3, 3 e 2, enquanto `p` recebe uma cópia dos vetores recuperados. A variável `size`, por sua vez, passa a conter o valor 3 após a execução desta operação. Os elementos de cada vetor contido em `p` são acessados como elementos de uma matriz, onde a linha n contém os elementos do $(n+1)$-ésimo vetor recuperado. Por exemplo, p[1][0] corresponde ao primeiro elemento do segundo vetor copiado para a variável `p`, lembrando que números de linhas e colunas começam em 0.

Ainda com relação ao exemplo acima, deve-se notar que o usuário é responsável por reservar espaço suficiente para a variável que recebe a cópia dos vetores recuperados. É recomendável que o espaço alocado seja capaz de acomodar vetores com o tamanho máximo especificado na operação, mesmo que isso leve ao desperdício de memória quando algum vetor tiver um número de elementos menor do que o máximo permitido. Além disso, também cabe ao usuário garantir que o tamanho de cada vetor recuperado possa ser determinado após a execução de **y_gather**. No exemplo acima, isto foi feito armazenando-se o tamanho de cada vetor no segundo campo de cada tupla, de modo a reunir estas informações na variável `s` fornecida como parâmetro a **y_gather**. Assim, para determinar o número de elementos do primeiro vetor contido em `p`, por exemplo, bastaria verificar o valor armazenado em s[0].

y_reduce

Assim como **y_gather**, a primitiva **y_reduce** serve para recuperar várias tuplas que satisfazem um mesmo *template* e, em seguida, removê-las do TS. Esta primitiva, no entanto, executa algumas operações simples sobre os valores coletados para determinados campos, e atribui o resultado destas operações aos campos formais especificados no *template*. A execução de **y_reduce**, portanto, equivale à redução de várias tuplas em apenas uma. O uso desta primitiva é ilustrado através do exemplo a seguir:

```
int result;

y_reduce(3, "partial_result", SUM(?result));
```

Esta operação procura no TS três tuplas que satisfaçam o *template* fornecido. Estas tuplas devem ter a cadeia de caracteres "partial_result" no primeiro campo, e um número inteiro no segundo campo. A função SUM aplicada ao segundo campo indica que a variável `result` deve receber o valor da soma dos campos correspondentes nas três tuplas encontradas. Supondo, por exemplo, que existissem no TS as tuplas

```
("partial_result", 4)
("partial_result", 10)
("partial_result", 25)
```

a operação **y_reduce** no exemplo acima retornaria o valor 39 na variável `result`. Caso não houvesse no TS um número suficiente de tuplas que satisfizessem o *template* especificado, a *thread* executando **y_reduce** seria bloqueada até que as tuplas estivessem disponíveis. Por outro lado, se o número de tuplas capazes de satisfazer o *template* fosse maior do que aquele especificado em **y_reduce**, qualquer combinação de tuplas poderia ser escolhida.

As funções que podem ser aplicadas a um campo são pré-definidas pelo sistema. Todas elas recebem como argumento um campo formal simples, e produzem um resultado do mesmo tipo do argumento. Se nenhuma função for aplicada nos campos fornecidos a **y_reduce**, esta primitiva preenche os campos formais com os valores de uma tupla qualquer dentre as tuplas recuperadas. No projeto inicial do sistema não foram incluídas funções que aceitam vetores como argumento, embora futuramente seja possível projetar funções de redução com este propósito. Existem dois grupos distintos de funções em YALI:

- Funções de Combinação: estas funções produzem um resultado que é uma combinação de vários valores coletados para um determinado campo.
- Funções de Seleção: estas funções selecionam um entre vários valores coletados para um determinado campo, e retornam este valor como resultado.

As funções suportadas pelo sistema para uso em conjunto com **y_reduce** são listadas na tabela 4.1. O uso destas funções, no entanto, deve obedecer algumas regras:

1. não se pode aplicar mais de uma função a um mesmo campo;
2. quando uma função de seleção é aplicada a um único campo formal num *template*, os campos formais restantes, se existirem, não precisam ter função aplicada. Estes campos, no caso, recebem os valores correspondentes na tupla selecionada. Como ilustração disso, considere o exemplo abaixo:

```
int   i, j;
float value;

y_reduce(2, "result", ?i, ?j, MIN(?value));
```

Se existissem no TS as tuplas

```
("result", 5, -4, 56.4)
```

e

```
("result", 2, 3, 45.1)
```

a função MIN faria com que `value` recebesse o valor 45.1 (mínimo entre 45.1 e 56.4) e, neste caso, as variáveis `i` e `j` receberiam, respectivamente, os valores 2 e 3. Como MIN é uma função de seleção e está aplicada a somente um campo do *template*, o resultado da redução está necessariamente contido em apenas uma tupla, que pode ser selecionada para fornecer os valores aos campos formais sem função aplicada. Se, no entanto, MIN fosse substituída por uma função de combinação, ou se fosse aplicada qualquer função a algum outro campo formal do *template*, não mais poder-se-ia garantir que o resultado da redução estaria contido em apenas uma tupla, e neste caso o uso de campos formais sem função aplicada não seria permitido.

3. quando uma função de combinação é aplicada a um campo formal em uma operação **y_reduce**, ou quando funções de seleção são aplicadas a mais de um campo num *template*, os campos formais restantes, se existirem, devem obrigatoriamente ter uma função aplicada. Pode-se, portanto, ter diferentes funções aplicadas a diferentes campos de um *template*, como mostra o exemplo abaixo:

```
int   i, j;
float value;

y_reduce(2, "result", SUM(?i), ABSMAX(?j),
    MIN(?value));
```

Considerando que existissem no TS as mesmas tuplas do exemplo anterior, as variáveis `i`, `j` e `value` receberiam, respectivamente, os valores 7 (soma de 5 com 2), 4 (máximo absoluto entre -4 e 3) e 45.1 (mínimo entre 45.1 e 56.4).

Tabela 4.1 - Funções pré-definidas para uso em **y_reduce**.

Função	Argumento	Resultado
SUM	float, double, int, short ou long	soma
MULT	float, double, int, short ou long	produto
MIN	float, double, int, short ou long	menor valor
MAX	float, double, int, short ou long	maior valor
ABSMIN	float, double, int, short ou long	menor valor absoluto
ABSMAX	float, double, int, short ou long	maior valor absoluto

y_barrier

Ao contrário das outras operações globais em YALI, **y_barrier** não manipula tuplas. Esta primitiva serve para sincronizar a execução de múltiplas *threads*, e para isso recebe como argumento um nome global de barreira e um número que indica a quantidade de *threads* que devem sincronizar-se nesta barreira. Ao alcançar uma barreira, isto é, ao executar uma chamada **y_barrier**, cada *thread* é bloqueada esperando que o restante das *threads* também atinja a mesma barreira. Quando isto acontece, as *threads* são desbloqueadas, garantindo que uma nova fase da computação seja iniciada ao mesmo tempo por várias *threads* de uma aplicação. Uma aplicação de **y_barrier** pode ser, por exemplo, na sincronização da função principal de um programa (`yali_main`) com as *threads* por ela criadas, como mostra o exemplo abaixo:

```
#define CHILDS 5

void func() {

    /* executa tarefa */
    ...
    /* sincroniza com yali_main() e com outras threads*/
    y_barrier("end", CHILDS+1);

}

yali_main() {

    int i;

    /* dispara varias threads */
    for (i = 0; i < CHILDS; i++)
        y_eval(func);

    /* espera pelo termino das threads */
    y_barrier("end", CHILDS+1);

}
```

Acima, a *thread* principal dispara várias *threads*, e só encerra sua execução quando todas estas *threads* tiverem terminado suas tarefas. Como se pode notar, o número de *threads* fornecido como argumento a **y_barrier** inclui, também, a *thread* principal que executa `yali_main`. Deve-se garantir que as *threads* a serem sincronizadas numa mesma barreira sempre utilizem chamadas **y_barrier** idênticas. Se, por exemplo, uma *thread* especificar erroneamente o nome de uma barreira, é provável que as *threads* envolvidas permaneçam bloqueadas indefinidamente. Se o erro ocorrer na

especificação do número de *threads* que devem alcançar a barreira, entretanto, o usuário pode detectá-lo testando o valor retornado por **y_barrier**, que deve ser igual a -1 na ocorrência do erro mencionado, ou igual a zero caso a operação tenha sido bem sucedida (no exemplo acima, a verificação do valor retornado por **y_barrier** foi omitida com fins de simplificação).

Outras Primitivas

O conjunto de primitivas YALI também inclui as seguintes funções:

- **y_id**: esta primitiva retorna a identificação do processo que a executa. O indentificador é um número inteiro de 0 a (N-1), sendo N o número de processos que fazem parte da aplicação. Cada processo tem um identificador único, que pode ser usado para selecionar o código a ser executado em programas SPMD;
- **y_nproc**: esta primitiva retorna o número de processos que compõem a aplicação paralela. Esta informação também pode ser útil para controlar a execução de aplicações SPMD.

4.2.3 Componentes do Sistema

Para suportar a funcionalidade descrita nas seções anteriores, YALI conta com três componentes distintos: um pré-processador, uma biblioteca e um programa de inicialização de aplicações paralelas. As funções de cada um destes componentes junto ao ambiente de programação de YALI serão descritas a seguir.

O Pré-Processador

Conforme discutido no capítulo anterior, as alternativas para incorporação do modelo Linda a uma linguagem seqüencial vão desde a simples utilização de uma biblioteca até a construção de um complexo pré-compilador. No projeto de YALI, entretanto, optou-se por uma alternativa intermediária, baseada no uso de um pré-processador. Basicamente, a função do pré-processador é identificar tipos e tamanhos dos campos fornecidos como parâmetro às primitivas YALI, a fim de convertê-las para um formato utilizado internamente pela biblioteca do sistema. Isto permite que a interface das primitivas seja bastante simples, já que não existe a necessidade do uso de funções ou outros artifícios para fornecer explicitamente o tipo e o tamanho de cada campo, como ocorre com alguns sistemas discutidos no capítulo anterior.

Do ponto de vista do ambiente de programação, um pré-processador é capaz de oferecer a mesma funcionalidade de um pré-compilador, satisfazendo plenamente os objetivos preliminares do projeto de YALI no que diz respeito à facilidade de aprendizado e uso do sistema. Do ponto de vista do ambiente de execução, entretanto, um pré-compilador é capaz de introduzir otimizações que permitem

aumentar o desempenho das aplicações, sendo, portanto, superior a um pré-processador. Embora isto constitua uma limitação da alternativa adotada em YALI, a construção de um pré-processador é mais simples que a implementação de um pré-compilador e, por isso, é mais adequada ao cronograma disponível. Futuramente, a funcionalidade do pré-processador pode ser estendida, a fim de atender melhor aos requisitos de desempenho.

O pré-processador do sistema, chamado ypp, recebe um ou mais nomes de programas YALI, com extensão ".y", e produz um ou mais programas com extensão ".c", que então devem ser processados por um compilador C. Com o pré-processamento, cada primitiva YALI é substituída por uma função equivalente contida na biblioteca do sistema. Para realizar esta substituição, o pré-processador precisa determinar o tipo e o tamanho de cada campo passado como argumento para as primitivas YALI, já que as rotinas da biblioteca necessitam destas informações. O usuário não precisa se preocupar com detalhes sobre o funcionamento do pré-processador, e também não precisa conhecer a sintaxe das rotinas da biblioteca. No entanto, o usuário deve estar consciente de que:

- os tipos de dados reconhecidos pelo pré-processador são aqueles suportados em YALI, ou seja, `char`, `int`, `short`, `long`, `float`, `double`, e vetores formados por estes tipos básicos. Caso deseje-se utilizar dados de outros tipos (tipos definidos pelo usuário, por exemplo), deve-se antes fazer a conversão para um dos tipos suportados pois, do contrário, o pré-processador não será capaz de fazer o reconhecimento, e retornará uma mensagem de erro. Esta conversão pode ser especialmente trabalhosa se o usuário desejar armazenar matrizes ou outras estruturas em uma tupla pois, neste caso, deve-se transformar estas estruturas em vetores de tipos conhecidos. Para acomodar estruturas com elementos de diferentes tipos, por exemplo, pode-se transformar toda a estrutura em um vetor de bytes (isto é, um vetor do tipo `char`), ou pode-se desmembrar a estrutura e colocar cada um de seus elementos em um campo da tupla;

- o pré-processador só consegue determinar o tamanho de um vetor quando esta informação é especificada na própria declaração do vetor. Como visto na seção 4.2.2, o número de elementos de um vetor dinamicamente alocado deve ser fornecido explicitamente após um símbolo ':'. Se o usuário não respeitar esta convenção, o pré-processador não poderá reconhecer que o campo é composto por vários elementos;

- em geral, é possível utilizar expressões e funções como campos de uma tupla. O pré-processador é capaz de identificar o tipo de dado resultante da expressão ou função, e por isso não há necessidade de o usuário fornecer explicitamente esta informação (a menos que o resultado seja de um tipo diferente daqueles suportados por YALI).

A Biblioteca do Sistema

A biblioteca do sistema, chamada **libyali.a**, contém as funções que implementam cada uma das primitivas descritas na seção 4.2.2. A interface destas funções é bem mais complicada que a interface descrita para as primitivas YALI. No entanto, estas rotinas nunca precisam ser utilizadas diretamente pelo usuário, pois suas chamadas são geradas automaticamente pelo pré-processador. Com relação à biblioteca, o usuário só precisa se preocupar em ligá-la aos módulos objetos gerados pelo compilador C escolhido, a fim de produzir os programas executáveis que compõem cada aplicação. Cada programa, então, pode ser executado em uma ou mais máquinas de uma rede, formando o conjunto de processos de uma aplicação.

Como visto em 4.1, um dos objetivos preliminares no projeto de YALI foi permitir a utilização de máquinas de diferentes arquiteturas para a execução de uma aplicação. A nível de ambiente de programação, uma conseqüência disso é que o usuário deve garantir que cada programa seja compilado de acordo com a(s) arquitetura(s) onde este deve executar. Para isso, a biblioteca do sistema também deve estar disponível para diferentes arquiteturas. Para que seja possível manter cópias de um mesmo programa objeto, executável, ou da própria biblioteca, compiladas para diferentes arquiteturas, YALI organiza estes elementos em diretórios separados para cada arquitetura suportada pelo sistema.

O Programa de Inicialização

Para disparar a execução de uma aplicação paralela sobre uma rede de computadores, o sistema conta com um utilitário chamado `yalistart`. Basicamente, a função deste programa é distribuir os processos de uma aplicação sobre a rede de acordo com uma configuração fornecida pelo usuário. Esta configuração consiste na especificação dos processos (programas) que compõem a aplicação e, também, das máquinas onde cada processo deve ser executado. Com `yalistart`, a configuração de uma aplicação pode ser feita de três modos:

- Linha de Comando: em aplicações SPMD, um único programa é disparado em várias máquinas da rede. Para executar aplicações deste tipo, o usuário pode fornecer as informações de configuração para `yalistart` diretamente na linha de comando, como mostra o exemplo abaixo:

  ```
  % yalistart /home/yali/progname mate cuia pala
  ```

  Neste exemplo, o programa "progname", do diretório "/home/yali", é disparado sobre as máquinas nomeadas "mate", "cuia" e "pala". É necessário que este programa esteja no mesmo diretório ("/home/yali") em cada uma das máquinas, o que se consegue com o uso de NFS. Se o programa estiver em diretórios diferentes em cada máquina, então torna-se necessário utilizar outro modo de configuração.

- Arquivo de Configuração: neste caso, a configuração da aplicação é especificada em um arquivo, que é passado como parâmetro para `yalistart`. Este

arquivo deve conter, em cada linha, o nome de um programa (incluindo a especificação completa do diretório onde o programa reside), e o nome das máquinas onde este programa deve ser executado, como mostra o exemplo abaixo:

```
% cat conf
/home/yali/master mate
/home/yali/worker1 cuia pala
/home/yali/worker2 cuia pala

% yalistart -f conf
```

Neste exemplo, a opção "-f" fornecida a `yalistart` indica que se deseja usar o arquivo "conf" para especificar a configuração da aplicação. Segundo especificado em "conf", `yalistart` deve disparar o programa "master" na máquina "mate", e os programas "worker1" e "worker2" nas máquinas "cuia" e "pala". Todos estes programas estão localizados no diretório "/home/yali", que deve existir em cada máquina especificada. Se cada programa estivesse em diretórios diferentes em cada máquina, bastaria especificar os nomes destes diretórios no arquivo de configuração.

- Modo Interativo: a configuração de uma aplicação também pode ser especificada interativamente, bastando, para isso, invocar `yalistart` com a opção "-i". Neste modo de configuração, indicado para usuários pouco experientes, `yalistart` solicita as informações necessárias sobre os processos da aplicação e sobre as máquinas onde cada processo deve executar.

É importante observar que a criação de processos em YALI só pode ser feita através de `yalistart`, e somente durante a fase de inicialização da aplicação. Como visto em 4.2.2, as primitivas YALI para expressão do paralelismo não permitem a criação dinâmica de processos, mas sim de *threads*. Embora estes processos possam executar em máquinas distintas, o dispositivo padrão de saída associado a cada um deles corresponde sempre ao terminal onde `yalistart` foi executado. Isto quer dizer, por exemplo, que o resultado de qualquer chamada à função `printf` aparece sempre no mesmo terminal, oferecendo a ilusão de que a aplicação está executando em apenas uma máquina. Operações de entrada pelo teclado, ao contrário, são restritas a um único processo da aplicação, que deve executar na mesma máquina onde `yalistart` foi invocado. Durante a configuração da aplicação, este processo deve ser especificado em primeiro lugar (por exemplo, na primeira linha do arquivo de configuração), pois do contrário a aplicação não poderá receber dados através do teclado.

Outra observação importante diz respeito ao sistema de arquivos ao qual cada processo de uma aplicação tem acesso. Em redes de não dispõem de NFS, cada processo "enxerga" o sistema de arquivos da máquina onde foi disparado, e só pode compartilhar arquivos com processos da aplicação que residam nesta mesma máquina. Quando dispõe-se de NFS, no entanto, é possível um maior compartilhamento de arquivos entre os processos da aplicação, já que alguns diretórios podem ser acessíveis a várias máquinas ao mesmo tempo.

4.3 Ambiente de Execução

No projeto do ambiente de execução de um sistema baseado em Linda, uma das principais questões a serem resolvidas é como suportar eficientemente a abstração do espaço de tuplas. Em YALI, optou-se por utilizar um espaço de tuplas distribuído, baseado numa política de *hashing*. Ao contrário da maioria das implementações do modelo, entretanto, YALI distribui o espaço de tuplas sobre cada um dos processos que compõem uma aplicação paralela, não necessitando de processos especiais para o gerenciamento do espaço de tuplas. Como será visto no próximo capítulo, esta organização tem a vantagem de exigir um menor volume de mensagens para a implementação das primitivas, influenciando positivamente no desempenho do sistema.

Além das soluções discutidas brevemente acima, o projeto do ambiente de execução de YALI envolveu várias outras decisões. O restante desta seção, portanto, apresenta e avalia as alternativas adotadas em YALI para resolver cada uma das questões de projeto que dizem respeito ao ambiente de execução de um sistema baseado em Linda.

4.3.1 Política de Distribuição de Tuplas

Já foi mencionado que YALI adota uma política baseada em *hashing* para coordenar a distribuição do espaço de tuplas. Como visto em 3.2.1, a escolha de uma política de distribuição de tuplas deve levar em conta as características da arquitetura onde o sistema deve executar e, também, os tipos de aplicações que o sistema deve suportar. No que diz respeito à arquitetura, YALI foi concebido para executar sobre uma rede heterogênea de computadores, normalmente interligados fisicamente via Ethernet. Em redes deste tipo, políticas de distribuição que requerem operações de difusão devem ser especialmente evitadas, já que uma mensagem difundida pode atingir nodos que possivelmente não estejam interessados na operação. Com relação aos tipos de aplicações, deve-se mencionar que não houve uma preocupação especial em adequar o sistema a um tipo específico de aplicação. Ao contrário, YALI destina-se a ser um sistema flexível, capaz de suportar o desenvolvimento e execução de aplicações paralelas em geral. Desta maneira, não se pode prever o padrão de acesso a tuplas predominante no sistema, e por isso a política de distribuição escolhida deve, na medida do possível, adaptar-se bem aos diversos padrões de comunicação entre processos.

As observações acima tiveram importância fundamental na escolha de uma política baseada em *hashing* para suportar a abstração do espaço de tuplas em YALI. Uma política deste tipo tem várias vantagens:

- sua implementação exige apenas mensagens ponto-a-ponto, evitando o uso de pesadas operações de difusão;
- o uso de memória é ótimo, já que não existe replicação de tuplas ou de *templates*;

- é uma política extensível, já que o número de mensagens necessárias para sua implementação não aumenta com o número de nodos da rede.

O principal problema com esta política, como foi visto em 3.2.1, é que seu grau de distribuição depende da função de *hash* escolhida. Considerando o espaço de tuplas como uma tabela *hash* distribuída, onde cada entrada é mapeada em um processo residente em um determinado nodo, é provável que uma função de *hash* simples mapeie tuplas com a mesma descrição sempre em uma mesma entrada desta tabela. Desta maneira, é possível que o espaço de tuplas fique distribuído sobre um pequeno conjunto de processos e, quanto menor for o grau de distribuição do espaço de tuplas, pior será o desempenho do sistema, já que diminui a quantidade de operações sobre o espaço de tuplas que podem ser executadas em paralelo. Este problema, no entanto, pode ser resolvido utilizando-se um mecanismo de *hashing* adaptativo[CAE94a], onde a função de *hash* leva em conta informações sobre a utilização do TS coletadas em tempo de execução.

4.3.2 Grau de Compartilhamento do Espaço de Tuplas

Em YALI, o espaço de tuplas é compartilhado somente pelos processos que pertencem a uma mesma aplicação. Do ponto de vista do ambiente de execução, esta solução é vantajosa por não requerer a utilização de mecanismos para impedir que processos de diferentes aplicações interfiram entre si através do espaço de tuplas. A principal justificativa desta decisão, entretanto, é a própria organização do espaço de tuplas em YALI, onde cada processo é responsável por gerenciar uma parte do espaço de tuplas distribuído. Esta organização determina que cada aplicação tenha seu próprio espaço de tuplas, e não permite que um espaço de tuplas tenha existência independente. Embora isto não constitua um problema para a maioria das aplicações paralelas, processos disjuntos no tempo passam a não mais poder cooperar através de um espaço de tuplas. Esta limitação foi imposta ao usuário unicamente por questões de desempenho, podendo futuramente vir a ser resolvida em novas versões de YALI.

4.3.3 Implementação de EVAL

O suporte à primitiva **eval** é um aspecto crítico no projeto do ambiente de execução de qualquer sistema baseado em Linda. Em YALI, como já foi visto na seção 4.2.2, optou-se por oferecer primitivas inspiradas em **eval**, mas que possuem uma semântica consideravelmente diferente para a expressão do paralelismo. Uma das principais diferenças é que o sistema só permite a criação dinâmica de *threads* e, adicionalmente, torna esta decisão visível ao usuário, ao contrário do que ocorre com **eval**. A opção por *threads* foi basicamente motivada por uma questão de desempenho: a criação de uma *thread* é uma operação mais rápida em relação à criação de um processo, uma vez que *threads* têm menos informações de estado associadas[TAN92].

Quando se discute a implementação de **eval**, é importante decidir se a primitiva criará processos locais ou remotos e, adicionalmente, definir o contexto inicial ao qual terão acesso estes processos. Embora YALI não suporte a criação dinâmica de processos, estas questões também foram consideradas no projeto do seu ambiente de execução, neste caso com relação a *threads*. Pelas primitivas apresentadas na seção 4.2.2, pode-se notar que YALI permite a criação de *threads* tanto local como remotamente. Esta decisão tem a vantagem de permitir que um processo ative a execução de uma *thread* em um processo residente em outra máquina, de modo a não causar aumento na sua carga local de processamento. Já com relação ao contexto inicial acessível às *threads* criadas, o próprio conceito de *thread* implica no compartilhamento do contexto com o processo de origem. Uma *thread* pode ter acesso, por exemplo, a dados e funções definidas globalmente no processo ao qual pertence. O acesso a dados globais, no entanto, não é encorajado em YALI, pois o sistema não provê meios de evitar que várias *threads* pertencentes ao mesmo processo tentem, ao mesmo tempo, alterar algum dado compartilhado. Além disso, *threads* podem ser criadas à distância e, neste caso, podem compartilhar espaços de endereçamento de processos diferentes daqueles que as invocaram. Assim, um processo poderia causar modificações em dados globais de outro processo, o que certamente seria uma operação arriscada.

4.3.4 Implementação de INP e RDP

Na apresentação do conjunto de primitivas suportado por YALI já foi visto que o sistema oferece primitivas não-bloqueantes equivalentes a **inp** e **rdp**, chamadas respectivamente de **y_inp** e **y_rdp**. Embora a disponibilidade destas primitivas sirva para aumentar a expressividade do sistema, a opção pela implementação de **y_inp** e **y_rdp** foi principalmente motivada pela política de distribuição de tuplas escolhida. Com uma política baseada em *hashing*, cada tupla é mapeada em uma única entrada da tabela *hash*, e cada entrada é gerenciada por um único processo. Por isso, para descobrir se uma determinada tupla está ou não presente no TS, basta consultar o processo responsável pela porção da tabela onde a tupla deve ter sido mapeada, o que requer um número pequeno de mensagens para implementação de **y_inp** e **y_rdp**.

4.3.5 Suporte à Heterogeneidade

Como discutido no início do capítulo, o suporte à heterogeneidade foi uma das preocupações iniciais no projeto de YALI. A princípio, considerou-se que YALI deveria suportar a heterogeneidade a nível de arquitetura, a fim de permitir que máquinas de diferentes tipos pudessem ser aproveitadas numa computação paralela. No entanto, é comum que máquinas com arquiteturas distintas utilizem sistemas operacionais diferentes, e por isso o suporte à heterogeneidade a nível de sistema operacional também seria uma característica bastante desejável. O suporte inicial a múltiplos sistemas operacionais, entretanto, seria inviável devido ao cronograma disponível, de maneira que optou-se por, inicialmente, adequar

o sistema apenas a plataformas Unix. Esta decisão, aliada à escolha da linguagem C como hospedeira, faz com que YALI seja potencialmente portável, já que sistemas compatíveis com Unix têm sido utilizados amplamente nos mais variados tipos de arquiteturas, e atualmente é possível encontrar compiladores C para praticamente qualquer combinação de arquitetura e sistema operacional.

Com relação a heterogeneidade a nível de arquitetura, o principal problema consiste em definir como deve ser feita a transferência de dados entre máquinas que adotam diferentes convenções para armazenar e representar dados. Nesse nível podem ocorrer diferenças, por exemplo, na ordem de armazenamento dos bytes na memória (*big-endian* ou *little-endian*), no tamanho da palavra do processador e na forma de representar números inteiros e valores de ponto flutuante. Quando diferenças deste tipo ocorrem entre duas máquinas que devem comunicar-se, o ambiente de execução deve ter meios de garantir que ambas as máquinas interpretem igualmente os mesmos dados. Em YALI, as soluções adotadas para implementação do suporte à heterogeneidade a nível de arquitetura são discutidas no próximo capítulo, na seção 5.6.

4.4 Sumário

Este capítulo descreveu o conjunto de soluções adotadas em YALI para resolver as várias questões envolvidas no projeto de um sistema baseado no modelo Linda. A nível de ambiente de programação, a maior parte das decisões de projeto foi tomada levando-se em conta que o sistema deveria ser de fácil aprendizado e de simples utilização. As principais características do ambiente de programação projetado são resumidas a seguir:

- YALI incorpora o modelo Linda à linguagem C, que é amplamente difundida para o desenvolvimento de aplicações de propósito geral;
- o conjunto de primitivas do sistema é pequeno, sendo portanto de fácil assimilação. YALI, no entanto, não se limita em oferecer somente as primitivas definidas em Linda, e estende o modelo através do suporte a **operações globais**, que são úteis para expressar a comunicação e a sincronização entre múltiplos processos. Convém destacar que estas operações foram incorporadas ao sistema sem qualquer prejuízo ao nível de abstração inerente ao modelo Linda;
- a expressão do paralelismo em YALI se dá através de *threads*, criadas dinamicamente para executar alguma função determinada pelo usuário. A criação dinâmica de processos não é permitida, por se tratar de uma operação muito pesada em relação à criação de *threads*. O mecanismo adotado em YALI para associação de nomes globais a funções permite a **criação de *threads* à distância**, isto é, permite que um processo cause o disparo de uma *thread* em outro processo da aplicação. Isto possibilita um certo grau de distribuição de carga, já que um processo pode ativar uma função em outra

máquina da rede, e usar os resultados desta função como se ela fosse executada localmente. Além disso, este mecanismo assemelha-se a uma chamada remota de procedimento, permitindo a estruturação de aplicações segundo um modelo cliente-servidor;

- o sistema dispõe de um pré-processador, o que permite oferecer uma interface bastante simples para o usuário. A tarefa do pré-processador é adaptar certas chamadas a funções YALI para o formato suportado pela biblioteca do sistema. As rotinas desta biblioteca, portanto, nunca são utilizadas diretamente pelo usuário, que precisa apenas preocupar-se em ligá-la aos módulos objetos que devem compor um programa executável YALI;

- aplicações YALI podem ser estruturadas tanto num modelo SPMD como MPMD. O sistema provê primitivas adequadas a ambos os modelos de estruturação de aplicações, e também dispõe de um utilitário de inicialização que distribui ou replica um ou mais programas sobre uma rede de computadores, de acordo com uma configuração estabelecida pelo usuário.

A nível de ambiente de execução, as soluções adotadas em YALI foram norteadas por dois objetivos principais: desempenho e capacidade de suporte à heterogeneidade. Algumas características do ambiente de execução projetado merecem ser salientadas:

- o espaço de tuplas suportado pelo sistema é distribuído entre os processos que compõem uma aplicação paralela, de modo que não são necessários processos intermediários para o gerenciamento do espaço de tuplas, como ocorre com muitos sistemas baseados em Linda. Isto influi favoravelmente no desempenho do sistema, diminuindo o volume de mensagens necessárias para implementação das primitivas;

- a política de distribuição de tuplas é baseada em *hashing*, adaptando-se satisfatoriamente a um ambiente de rede por ser uma política extensível e por não exigir pesadas operações de difusão na sua implementação.

5 Implementação de YALI

O capítulo anterior apresentou o projeto de YALI, fornecendo a descrição da interface oferecida ao usuário e das alternativas adotadas na concepção de seu ambiente de execução. Neste capítulo, a implementação do sistema projetado é apresentada detalhadamente, com ênfase na adaptação de YALI a duas plataformas de *hardware* diferentes. Esta implementação é feita em linguagem C e, como discutido no capítulo anterior, destina-se a sistemas operacionais compatíveis com Unix.

5.1 Plataformas de Implementação

Como YALI destina-se à programação paralela em redes heterogêneas, decidiu-se implementar a primeira versão do sistema em duas plataformas diferentes, a fim de avaliar o sistema em um ambiente realmente heterogêneo. Para isso, escolheu-se as arquiteturas SPARC com sistema operacional Solaris 2.4, e Intel 486 com sistema Mach 4.0. Estas plataformas têm algumas características em comum: ambas utilizam sistemas compatíveis com Unix, suportam o desenvolvimento de aplicações *multithreaded*, e empregam processadores de 32 bits. Existem, no entanto, várias particularidades que distingüem uma plataforma da outra, e que foram consideradas no momento de se resolver algumas questões relacionadas à implementação de YALI.

A plataforma i486/Mach utilizada se distingüe da maioria das plataformas compatíveis com Unix por empregar um sistema operacional baseado em *microkernel*. O sistema Mach[ACC86, RAS89, RAS89a], por si só, consiste em um núcleo mínimo de sistema operacional (*microkernel*), que provê apenas um pequeno conjunto de serviços (gerência e comunicação local entre processos, gerência de memória e controle de recursos físicos). Qualquer funcionalidade adicional como, por exemplo, sistema de arquivos ou comunicação em rede, é incorporada no Mach através de servidores que executam a nível de usuário. Para tornar o sistema compatível com Unix, portanto, deve-se utilizar um servidor que implemente todas as características de um sistema deste tipo através dos serviços oferecidos pelo *microkernel*. Na plataforma i486/Mach onde YALI foi implementado, a compatibilidade com Unix foi conseguida através do servidor 4.4BSD-Lite[HEL94], desenvolvido na Universidade de Helsinki, na Finlândia.

Esta organização particular do sistema Mach, no entanto, não teve influência direta na implementação de YALI. Os fatores que mais influenciaram foram, na verdade, a versão de Unix suportada pelo servidor (4.4BSD), e a biblioteca de *threads* oferecida pelo sistema, denominada **Cthreads**[COO90]. Estas duas características da plataforma i486/Mach diferem daquelas encontradas na plataforma SPARC/Solaris, que emprega um sistema compatível com a versão System V do Unix, e suporta uma biblioteca de *threads* conhecida como **Solaris *threads***[POW91, STE92]. A existência destas diferenças levou a uma escolha cuida-

dosa das chamadas de sistema e das rotinas de manipulação de *threads* utilizadas para implementação de YALI, de modo a garantir um alto nível de compatibilidade de código entre as plataformas suportadas. Em outras palavras, houve uma grande preocupação com a **portabilidade** do sistema, que é uma característica fundamental para sua adaptação a plataformas heterogêneas.

5.2 Estrutura Geral do Ambiente de Execução

Como já foi mencionado no capítulo anterior, YALI implementa um espaço de tuplas que é distribuído, através de *hashing*, entre os processos que compõem uma aplicação paralela. Esta organização foi adotada por dispensar o uso de processos servidores intermediários, o que garante maior eficiência à medida que as mensagens para implementação das primitivas podem ser trocadas diretamente pelos processos usuários. Em sistemas como POSYBL, onde o espaço de tuplas é distribuído sobre vários processos servidores, o volume de mensagens trocadas é maior, já que os processos usuários sempre precisam se comunicar com o servidor local para solicitar a execução de alguma operação.

Com esta organização, cada processo deve ser capaz de, ao mesmo tempo, gerenciar uma porção do espaço global e executar a parte que lhe cabe da computação paralela. Adicionalmente, cada processo também deve estar preparado para atender operações que não requerem a manipulação de tuplas, como por exemplo **y_eval** ou **y_barrier**. Para embutir toda esta funcionalidade em um processo, optou-se por uma implementação *multithreaded* do ambiente de execução de YALI, de modo a dividir as várias atribuições de cada processo entre um conjunto de *threads* especializadas. Pode-se dizer, então, que o ambiente de execução de YALI é formado pelas múltiplas *threads* que residem em cada processo de uma aplicação. Estas *threads* podem ser basicamente de três tipos:

- **Thread Usuária (TU)**: *threads* deste tipo executam funções definidas pelo usuário, as quais, por sua vez, utilizam primitivas YALI contidas na biblioteca do sistema. Cada primitiva tem suas particularidades de implementação mas, em geral, os parâmetros fornecidos para uma primitiva contribuem para determinar, através de *hashing*, a identificação do processo que irá executá-la. Dependendo do resultado da função de *hash*, a operação pode ser executada localmente, por uma nova *thread*, ou pode ser requisitada a um processo remoto. Além disso, se a operação for síncrona, a TU bloqueia até receber o resultado correspondente.

  Qualquer processo YALI tem pelo menos uma TU, que executa a função principal especificada pelo usuário (`yali_main`). No entanto, *threads* usuárias podem ser criadas local ou remotamente em qualquer quantidade, através, respectivamente, das operações **y_eval** e **y_globeval** (a única limitação são os recursos do sistema operacional hospedeiro).

- **Thread Receptora (TR)**: sua principal função é receber mensagens de outros processos da aplicação, e garantir que estas mensagens sejam processadas

corretamente. Basicamente, uma mensagem pode ser de dois tipos: uma **requisição**, que solicita a execução local de alguma operação, ou uma **resposta**, que contém resultados de alguma operação previamente solicitada a outro processo. Em geral, cada requisição recebida pela TR é atendida por uma *thread* especializada, criada dinamicamente para o processamento da requisição. Respostas, por sua vez, são repassadas pela TR diretamente à TU que gerou a requisição.

- **Thread Processadora (TP)**: *threads* deste tipo são criadas dinamicamente por uma TU ou pela TR para processar operações YALI. Quando uma TP é criada para processar uma operação solicitada localmente, a resposta correspondente, se houver, é fornecida diretamente à TU solicitante. Ao contrário, se a TP é criada para atender uma requisição remota, a resposta (se houver) é enviada ao processo solicitante, que então deve providenciar para que ela seja repassada à TU que gerou a requisição. Como será visto na seção 5.7, existem vários tipos de *threads* processadoras, cada uma especializada na execução de determinadas operações.

Considerando esta organização com múltiplas *threads*, um processo YALI tem a estrutura ilustrada na figura 5.1. Na figura seguinte, 5.2, é possível visualizar a estrutura global do ambiente de execução YALI, através de duas aplicações executando simultaneamente sobre três nodos de uma rede de computadores. Enquanto a primeira aplicação é formada por dois processos residentes no nodo 1 e um processo no nodo 3, a aplicação 2 é composta por apenas dois processos, residentes, respectivamente nos nodos 2 e 3. Pode-se notar, através deste exemplo, que processos de diferentes aplicações YALI podem coexistir em um nodo da rede, e que os processos de uma mesma aplicação podem executar concorrentemente ou em paralelo, dependendo do nodo em que são alocados.

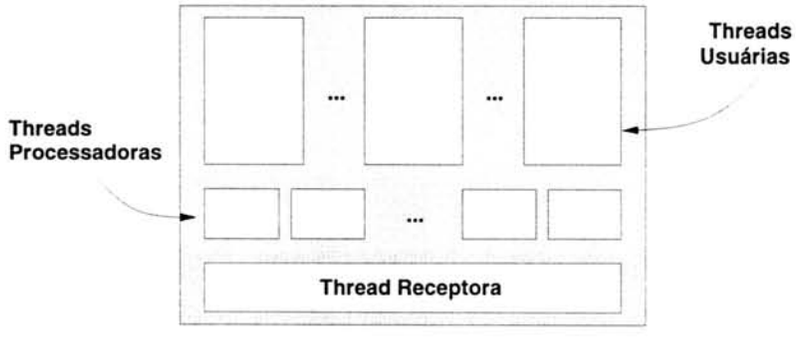

Figura 5.1 - Processo YALI.

Existem várias vantagens nesta organização. Em primeiro lugar, o uso de *threads* permite explorar a concorrência na execução das operações, o que influencia favoravelmente no desempenho do sistema. Em segundo lugar, como as *threads* são especializadas, a implementação do sistema fica bastante modular, facilitando possíveis alterações no código. Por fim, a criação dinâmica de *threads* para o processamento de requisições é mais vantajosa do que o uso de um grupo estático de *threads*, isto porque as novas *threads* não precisam bloquear esperando por

requisições a processar. Quando uma *thread* bloqueia, seu contexto precisa ser salvo para que sua execução possa ser retomada mais tarde, e esta seqüência de operações – salvamento e recuperação do contexto – tem um custo maior do que a simples criação de uma nova *thread*[TAN92].

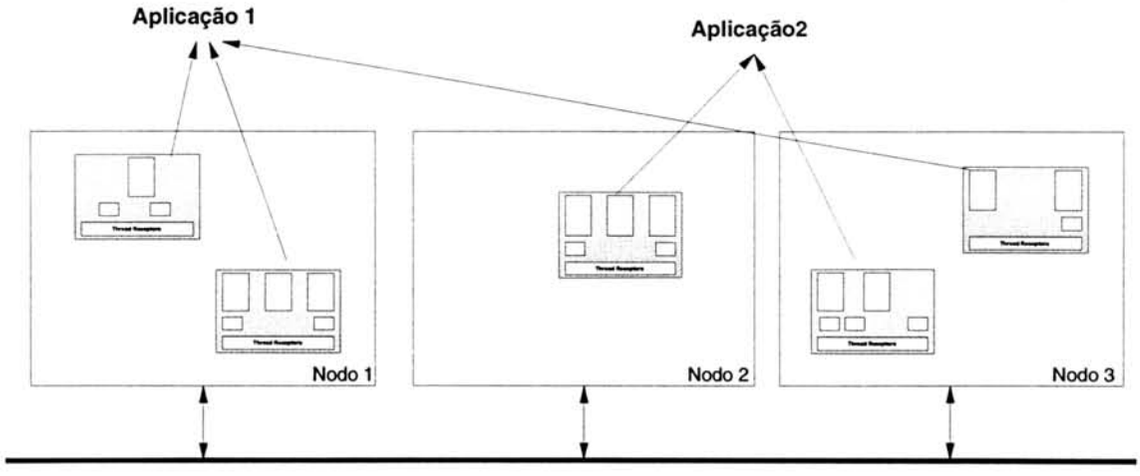

Figura 5.2 - Aplicações YALI executando simultaneamente.

Como todas as *threads* do mesmo processo compartilham o mesmo espaço de endereçamento, a interação entre elas é sempre feita através de memória compartilhada. Isto é vantajoso porque evita muitas cópias de dados mas, por outro lado, introduz uma complexidade adicional na implementação do sistema, já que todo o acesso a dados compartilhados precisa ser sincronizado.

5.3 Estrutura de Armazenamento de Dados

Cada processo YALI mantém um conjunto de estruturas de dados globais, que são manipuladas pelas *threads* processadoras durante a execução de operações YALI. A cada uma destas estruturas é associado um *mutex*, que é um tipo de semáforo binário (só pode assumir valores 0 ou 1)[TAN92] usado para garantir a exclusão mútua no acesso aos elementos da estrutura. Um *mutex* admite duas operações, *lock* e *unlock*, que são usadas, respectivamente, no ínicio e no fim de uma seção crítica, onde uma *thread* modifica alguma estrutura global. A seguir são descritas as principais estruturas mantidas por um processo YALI.

5.3.1 Espaço de Tuplas Local

Esta estrutura armazena as tuplas que a função de *hash* destina a cada processo, representando uma porção do espaço de tuplas global. É uma das mais importantes estruturas mantidas por um processo, pois é manipulada pela maioria das primitivas YALI. O espaço de tuplas é organizado sob a forma de uma tabela *hash*, onde tuplas são localizadas através de uma chave formada pela concatenação das seguintes informações:

- conteúdo do primeiro campo da tupla;
- número de campos da tupla;
- lista dos tipos de dados associados a cada campo.

A função de *hash* utilizada é baseada em [PEA90], e mapeia cada tupla em uma entrada da tabela. Como é possível que várias tuplas possuam a mesma chave, cada entrada na tabela contém um ponteiro para uma lista onde estas tuplas são encadeadas. A cada tupla é associado um conjunto de descritores de campos, onde são mantidos, além de informações sobre tipos de dados e tamanhos dos campos, também ponteiros para as posições de memória que armazenam o conteúdo de cada campo. Esta estrutura do espaço de tuplas local é ilustrada na figura 5.3.

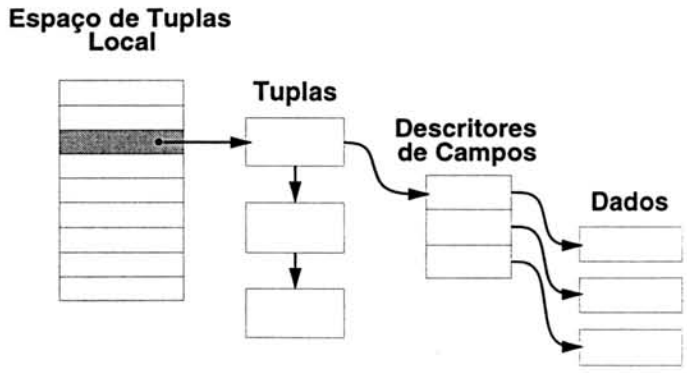

Figura 5.3 - Estrutura de um Espaço de Tuplas Local.

5.3.2 Tabela de Requisições de Tupla Pendentes

Nesta estrutura são armazenadas informações sobre requisições de recuperação de tuplas ainda não satisfeitas. Uma pendência é registrada nesta tabela sempre que o processo é escolhido para atender uma requisição **y_in**, **y_rd**, **y_gather** ou **y_reduce**, mas a tupla requisitada não é encontrada no espaço de tuplas local. Esta tabela é organizada e acessada de maneira semelhante ao espaço de tuplas, isto é, através de *hashing*. A diferença é que nesta tabela são armazenados *templates* ao invés de tuplas, e a cada um deles são adicionadas algumas informações complementares, que permitem identificar o tipo e a origem da requisição. A chave usada para *hashing* é composta pelas mesmas informações descritas na seção anterior, neste caso referentes a *templates*. Além disso, os descritores de campos de cada *template* mantêm também o tipo de cada campo, que pode ser real ou formal. Somente descritores de campos reais precisam manter ponteiros para dados, como pode ser visto na figura 5.4, que ilustra a estrutura desta tabela de requisições pendentes.

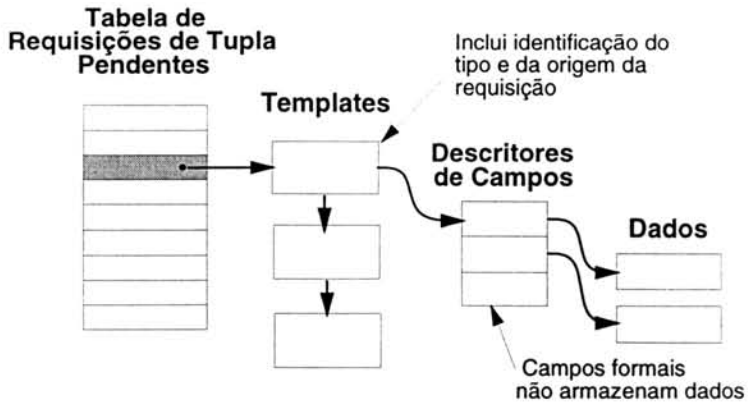

Figura 5.4 - Estrutura de uma Tabela de Requisições de Tupla Pendentes.

5.3.3 Tabela de Barreiras

Esta estrutura, também organizada como uma tabela *hash*, mantém informações sobre as barreiras gerenciadas por cada processo. A chave utilizada para *hashing* é o próprio nome da barreira, e a função é a mesma empregada nas outras estruturas descritas anteriormente. Como é possível que chaves diferentes produzam um mesmo índice (já que o número de entradas na tabela é fixo), cada entrada contém um ponteiro para uma lista encadeada de barreiras. A cada barreira estão associados o número de processos que devem alcançá-la e algumas informações sobre os processos que já a alcançaram (na verdade, todas estas informações correspondem a *threads*, não processos, já que uma operação **y_barrier** bloqueia apenas a *thread* que a executa). A figura 5.5 ilustra a organização desta tabela.

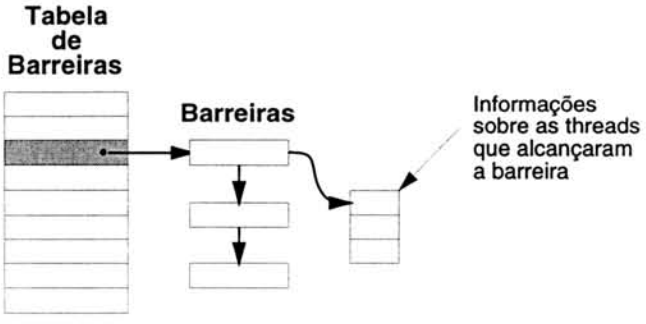

Figura 5.5 - Estrutura de uma Tabela de Barreiras.

5.3.4 Tabela de Funções Globais

Esta estrutura armazena informações sobre as funções globais gerenciadas por cada processo e, como as demais estruturas já comentadas, também é organizada como uma tabela *hash*. A chave usada para *hashing* é o nome global da função e, como é possível que chaves diferentes produzam um mesmo índice, cada entrada na tabela contém um ponteiro para uma lista encadeada de funções globais. Cada função, por sua vez, contém um ponteiro para uma lista de processos que

a implementam. Nesta lista são armazenados a identificação de cada processo e o endereço que a função global assume em cada um destes processos. Esta organização da tabela de funções globais é ilustrada na figura 5.6.

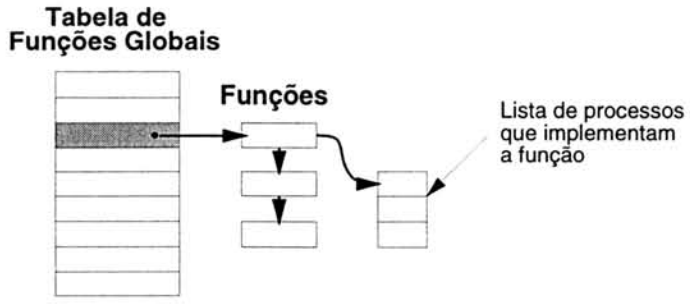

Figura 5.6 - Estrutura de uma Tabela de Funções Globais.

5.3.5 Tabela de Processos

Esta estrutura armazena várias informações sobre os processos que compõem uma aplicação paralela, necessárias para permitir o envio de requisições ou respostas a estes processos. Cada entrada nesta tabela corresponde a um processo da aplicação, e pode ser acessada utilizando-se como índice o identificador numérico deste processo. Todos os processos de uma aplicação têm uma cópia idêntica desta tabela, que é construída durante a inicialização da aplicação.

5.4 Programação *Multithreaded*

Como discutido em 5.2, o uso de *threads* na implementação do ambiente de execução de YALI tem várias vantagens. Esta solução, no entanto, envolve algumas questões de implementação adicionais, que serão discutidas a seguir.

5.4.1 Portabilidade

A opção por *threads* tem uma influência negativa na portabilidade do sistema, já que alguns sistemas operacionais compatíveis com Unix não suportam esta funcionalidade. No entanto, versões mais novas de muitos sistemas operacionais tradicionais vêm gradativamente incluindo bibliotecas de *threads*, embora nem sempre compatíveis. Um grande passo para a padronização de interfaces de *threads* foi dado relativamente há pouco tempo, com a criação da norma IEEE 1003.1c[IEE95]. Esta norma é baseada na interface POSIX, conhecida como Pthreads[DEC93], cujas versões preliminares (*drafts*) já vinham sendo implementadas em alguns sistemas (este é o caso, por exemplo, da biblioteca de *threads* do sistema Solaris).

Na implementação de YALI houve uma preocupação em manter a manipulação de *threads* o mais portável possível. Para garantir isso, a interface das bi-

bliotecas de *threads* dos sistemas Mach e Solaris (denominadas, respectivamente, Cthreads e Solaris *threads*) foi cuidadosamente analisada, de modo a evitar o uso de rotinas sem correspondência em ambas as bibliotecas. Foi possível notar que a semântica de muitas rotinas básicas de controle e sincronização de *threads* têm um alto nível de correspondência nestas bibliotecas, diferindo apenas em alguns detalhes de sintaxe. Além disso, a biblioteca do sistema Solaris é, sem dúvida, superior à biblioteca Cthreads com relação aos recursos oferecidos, principalmente no que se refere a mecanismos de sincronização de *threads*. Como conseqüência desta análise, optou-se por utilizar somente a funcionalidade básica de cada biblioteca para implementar o controle e a sincronização de *threads*, o que também aumenta a portabilidade do sistema. Com relação ao padrão POSIX, pela análise de alguma documentação disponível[SUN94, DEC93], já foi possível verificar que YALI seria facilmente adaptado à interface Pthreads.

5.4.2 Aspectos de Programação

A programação de um sistema *multithreaded* envolve alguns cuidados adicionais, principalmente com relação à sincronização das *threads*. Existem basicamente dois casos em que a sincronização é necessária entre *threads*:

- para garantir exclusão mútua em operações que modificam dados compartilhados, de modo a preservar sua consistência. Isso normalmente é feito através de um *mutex* e de suas operações associadas (*lock* e *unlock*), que devem estar disponíveis na biblioteca de *threads* utilizada. Este mecanismo de sincronização é suportado de maneira bastante semelhante nas bibliotecas de *threads* fornecidas com os sistemas Mach e Solaris;

- quando uma *thread* precisa esperar que outra *thread* termine alguma tarefa. Para isso, muitas bibliotecas de *threads* suportam **variáveis de condição**, às quais são geralmente associadas duas operações: *wait*, que bloqueia uma *thread* enquanto uma determinada condição for falsa, e *signal*, que sinaliza à *thread* bloqueada que uma condição se tornou verdadeira. Uma condição sempre se refere ao conteúdo de algum dado compartilhado, e toda variável de condição deve ser protegida por um *mutex*. Desde que não existe garantia de que a condição ainda seja verdadeira quando a *thread* bloqueada volta a executar (porque outra *thread* já pode ter conseguido modificar o dado compartilhado), a espera pela condição precisa ser da forma:

```
mutex_lock(mutex);
...
while (/* condicao falsa */)
    condition_wait(condition, mutex);
    ...
mutex_unlock(mutex);
```

  Com relação às bibliotecas de Cthreads e Solaris *threads*, ambas suportam variáveis de condição de maneira bastante similar. A biblioteca do sistema

Solaris, entretanto, inclui uma operação adicional para manipulação de variáveis de condição, que permite desbloquear não uma, mas todas as *threads* que esperam por uma determinada condição.

Ao usar os mecanismos de sincronização descritos acima, é necessário muito cuidado para evitar *deadlock*. Um *deadlock* é definido como uma situação onde uma ou mais *threads* estão permanentemente bloqueadas esperando umas pelas outras. Isto pode acontecer, por exemplo, quando duas *threads* usam duas variáveis *mutex* encadeadas em ordens diferentes, como mostra a figura 5.7. Neste exemplo, se ambas as *threads* entram na primeira seção crítica, elas permanecerão bloqueadas indefinidamente, cada uma esperando que a outra libere o *mutex* associado à sua segunda região crítica. Para evitar situações como esta, uma técnica simples é definir uma ordem global no uso de variáveis *mutex*, que deve ser respeitada por todas as *threads* que encadeiam operações *lock*, e utilizada de maneira reversa nas operações *unlock* correspondentes.

```
mutex_lock(mutex1);                    mutex_lock(mutex2);
/* primeira thread entra na            /* segunda thread entra na
   primeira secao critica */              primeira secao critica */
    ...                                    ...
    mutex_lock(mutex2);                    mutex_lock(mutex1);
    /* primeira thread entra na            ...
       segunda secao critica */            ...
        ...                                ...
    mutex_unlock(mutex2);                  mutex_unlock(mutex1);
    ...                                    ...
mutex_unlock(mutex1);                  mutex_unlock(mutex2);
```

Figura 5.7 - Situação de *deadlock* entre duas *threads*.

Outro problema encontrado na programação *multithreaded* é que muitas rotinas de biblioteca de propósito geral podem manipular variáveis compartilhadas, como é o caso da variável global `errno`, utilizada em sistemas Unix para manter o código de erro da última chamada de sistema utilizada. A menos que as bibliotecas utilizadas tenham sido projetadas para funcionar na presença de múltiplas *threads*, é aconselhável que o uso de suas rotinas seja protegido através de um *mutex*.

5.5 Mecanismos de Comunicação entre Processos

Mecanismos de comunicação entre processos são fundamentais para a implementação do ambiente de execução de YALI, já que a maioria das primitivas que o sistema suporta podem gerar mensagens a outros processos da aplicação. No capítulo 3, o estudo de alguns sistemas que implementam o modelo Lin-

da mostrou que existem basicamente duas alternativas para implementação da comunicação entre processos:

- utilizar ferramentas que auxiliam no desenvolvimento de programas distribuídos, como PVM ou p4;
- utilizar os mecanismos que o próprio sistema operacional disponibiliza para comunicação entre processos.

O uso de uma ferramenta auxiliar tem a vantagem de facilitar a implementação de um ambiente de execução distribuído, já que oferece recursos de mais alto nível que aqueles suportados pelo sistema operacional. Além disso, se a ferramenta escolhida for voltada à programação em plataformas heterogêneas, o ambiente de execução construído se torna, automaticamente, heterogêneo e portável, sem a necessidade de maiores esforços de programação. Todas estas facilidades, no entanto, têm um preço: um sistema construído a partir de ferramentas deste tipo terá, provavelmente, um desempenho menor do que outro que utiliza somente os recursos do sistema operacional, já que, a princípio, toda a comunicação entre processos será intermediada pelo ambiente de execução da ferramenta utilizada.

Diante desta desvantagem da primeira alternativa, optou-se pela construção de YALI a partir dos mecanismos de comunicação oferecidos pelo próprio sistema operacional Unix. Embora esta escolha tenha levado a um maior esforço de desenvolvimento, ela permitiu maiores otimizações na implementação do ambiente de execução, já que o controle exercido sobre a comunicação entre processos e o suporte à heterogeneidade foi maior.

Com relação aos mecanismos de comunicação suportados por sistemas compatíveis com Unix, existem duas interfaces que podem ser usadas para implementar a comunicação entre processos independentes: os *sockets*[STE90], que constituem uma interface genérica para comunicação entre processos, tanto locais como remotos, e a *Transport Level Interface* (TLI)[STE90], que se destina à comunicação através de uma rede. Enquanto *sockets* são suportados diretamente pelo sistema operacional e estão presentes em praticamente qualquer sistema compatível com Unix, a interface TLI é constituída apenas por funções de biblioteca, e geralmente só é disponível em sistemas Unix baseados na versão System V, da AT&T. Em YALI, para garantir maior portabilidade ao seu ambiente de execução, optou-se pela utilização da interface de *sockets* na comunicação entre processos.

A interface de *sockets* baseia-se em um elemento fundamental, o *socket*, que pode ser visto como um canal aberto em um processo para recepção e envio de mensagens. Quando um *socket* deve receber mensagens, é necessário atribuir-lhe um **endereço**, de modo que outros processos possam referenciá-lo. Estes endereços, por sua vez, são divididos em **domínios**. Existem dois domínios principais:

- Unix: neste domínio, os endereços atribuídos aos *sockets* são nomes válidos dentro de um sistema de arquivos. Um *socket* no domínio Unix, portanto,

permite a comunicação entre processos que residem na mesma máquina e que tenham acesso ao mesmo sistema de arquivos;

- Internet: os endereços de *sockets*, neste domínio, consistem na identificação da máquina que contém o *socket* (endereço da máquina na rede) e em um número de porta, que pode ser visto como o número de uma caixa postal. O domínio Internet, portanto, permite a comunicação entre processos que residem em diferentes máquinas de uma mesma rede.

Independemente do domínio de *sockets* utilizado, a comunicação pode seguir um protocolo orientado a conexão ou um protocolo orientado a datagrama. Uma conexão é geralmente empregada quando existe um fluxo contínuo de informações entre dois processos, enquanto datagramas são freqüentemente usados para troca de pequenas mensagens independentes. No domínio Internet, a comunicação orientada a conexão se dá através do protocolo TCP (*Transmission Control Protocol*), e a comunicação orientada a datagrama é feita através do protocolo UDP (*User Datagram Protocol*).

Para implementação do ambiente de execução de YALI foram utilizados vários tipos de *sockets*. Basicamente, processos residentes no mesmo nodo trocam requisições e respostas através de datagramas no domínio Unix, enquanto processos que residem em nodos diferentes se comunicam através do protocolo UDP. A opção por datagramas foi baseada na constatação de que uma implementação baseada em conexões fixas entre cada processo seria pouco extensível, já que o número de conexões que um processo pode manter é limitado. O uso de conexões estabelecidas dinamicamente para o envio de uma nova mensagem também não seria viável, pois o estabelecimento de uma conexão é uma operação relativamente demorada.

Datagramas, no entanto, não são apropriados para o envio de mensagens muito longas. Por isso, quando uma mensagem com mais de 8Kbytes precisa ser enviada pela rede, uma conexão TCP é estabelecida dinamicamente entre os processos envolvidos. Mensagens subseqüentes que precisem ser enviadas ao mesmo processo destinatário utilizam esta conexão já estabelecida, de modo que o alto custo do estabelecimento da conexão é bem aproveitado pelo sistema. Para implementar este esquema de comunicação, cada processo tem o conjunto de *sockets* a seguir, que é ilustrado na figura 5.8.

- dois *sockets* para recepção de datagramas (um no domínio Unix e outro no domínio Internet), que são constantemente monitorados pela *thread* receptora. A estes *sockets* são atribuídos endereços únicos (um nome de arquivo para o *socket* no domínio Unix, e um número de porta para o *socket* no domínio Internet), que são divulgados durante a inicialização da aplicação para permitir que os processos possam enviar mensagens uns aos outros. Como estes endereços de *sockets* precisam ser consultados toda vez que uma *thread* usuária ou processadora deve enviar uma mensagem, eles ficam armazenados na tabela de processos descrita em 5.3, que é acessível a todas as *threads* de um processo;

- um *socket* que aceita conexões TCP, que também é monitorado pela *thread* receptora e cujo endereço (um número de porta) também é divulgado durante a inicialização da aplicação, para a montagem das tabelas de processos. A cada pedido de conexão recebido, um novo *socket* é criado pela TR, a fim de que as mensagens provenientes do processo recém conectado sejam recebidas sem afetar o estabelecimento de novas conexões;

- um grupo de *sockets* orientados a datagrama, tanto no domínio Unix como Internet, usados para envio de mensagens pelas *threads* usuárias e processadoras. Estes *sockets* não precisam ter endereços associados, já que não são usados para recepção de mensagens. No entanto, eles são compartilhados por várias *threads*, tanto receptoras como processadoras, e por isso o acesso a eles precisa ser protegido por variáveis do tipo *mutex*. A existência de vários *sockets* deste tipo diminui a possibilidade de uma *thread* ter que esperar por outra para enviar uma mensagem, o que permite maior concorrência no processamento de requisições;

- um grupo de *sockets* TCP usados para o estabelecimento de conexões com outros processos da aplicação, e posterior envio de mensagens. Existe um *socket* neste grupo para cada processo da aplicação. Depois que uma conexão é estabelecida com algum processo, os *sockets* UDP e Unix não são mais usados para envio de mensagens a este processo, as quais passam a ser transmitidas através do *socket* conectado. Como estes *sockets* são compartilhados por todas as *threads* do processo, o acesso a eles é controlado através de variáveis *mutex*, a fim de garantir que somente uma *thread* de cada vez utilize cada *socket* deste grupo.

5.6 Suporte à Heterogeneidade

O mecanismo de *sockets* adotado para implementar a comunicação entre processos não oferece qualquer suporte à heterogeneidade, e por isso o próprio ambiente de execução de YALI se encarrega desta importante tarefa. A implementação do suporte à heterogeneidade envolve, basicamente, o uso de alguma técnica para garantir que dados sejam transferidos corretamente entre máquinas heterogêneas. Em outras palavras, isto quer dizer que o processo YALI destinatário deve enxergar os mesmos dados que o processo emissor, mesmo que residam em máquinas que adotam diferentes convenções para representação destes dados.

Existem basicamente duas técnicas para implementar a transferência de dados entre máquinas heterogêneas[COR91]:

- numa técnica conhecida como *receiver makes right*[COR91], ou "receptor corrige", o processo receptor se encarrega de fazer uma conversão adequada dos dados. Para isso, toda mensagem deve carregar consigo alguma identificação da arquitetura onde foi gerada, e devem existir procedimentos que implementam a conversão de dados entre as várias representações

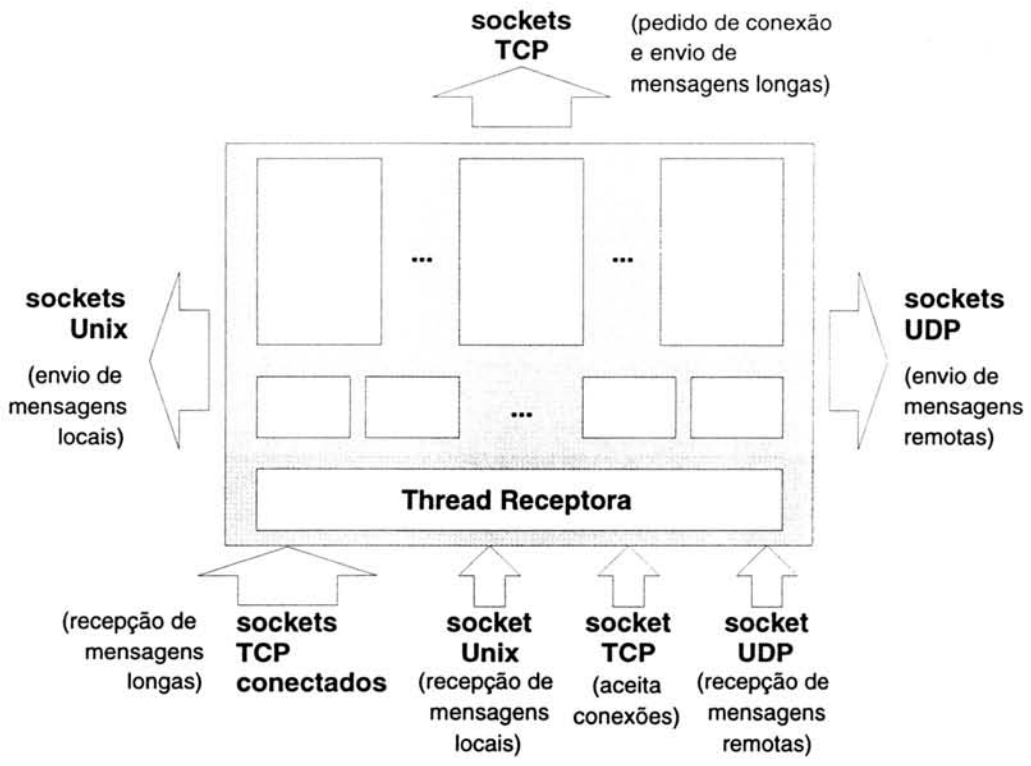

Figura 5.8 - Conjunto de sockets de um processo YALI.

possíveis. A necessidade de uma grande variedade de procedimentos de conversão pode, portanto, dificultar a implementação desta técnica. Mesmo assim, esta técnica é atraente por sua eficiência, já que não exige conversões intermediárias e, entre duas máquinas com a mesma arquitetura, nenhuma conversão se faz necessária;

- na técnica referenciada como "representação canônica", qualquer dado a ser transferido é convertido para um formato canônico, sendo depois desconvertido para a representação usada na máquina destinatária. Neste caso, os procedimentos de conversão são menos numerosos, já que, para cada arquitetura suportada, são necessárias apenas conversões do formato particular da arquitetura para o formato canônico, e vice-versa. A desvantagem desta técnica, no entanto, é que ela pode exigir conversões desnecessárias quando uma transferência ocorre entre máquinas com a mesma arquitetura, sendo, por isso, menos eficiente do que a técnica descrita no item anterior. Um dos exemplos mais comuns de emprego desta técnica é a biblioteca XDR (*External Data Representation*)[SUN87], que consiste em um conjunto de rotinas para programas C capazes de permitir a descrição de estruturas de dados arbitrárias de maneira independente de máquina. Esta biblioteca tem sido amplamente utilizada em ambientes heterogêneos por razões de portabilidade, embora utilize uma técnica menos eficiente para a transferência de dados.

Em YALI, optou-se por implementar a transferência de dados entre máquinas heterogêneas através de uma técnica do tipo "receptor corrige". O uso de XDR também foi considerado, mas constatou-se que esta biblioteca exigiria uma excessiva manipulação de *buffers* para a realização das conversões, causando um *overhead* adicional no ambiente de execução. Como muitos fabricantes de computadores baseiam-se em padrões IEEE para representação de dados[ZHO95], o número total de rotinas de conversão tende a manter-se pequeno, mesmo que o sistema seja adaptado para muitas outras arquiteturas diferentes.

Para permitir que conversões sejam feitas quando necessário, toda mensagem trocada por processos YALI possui um cabeçalho padrão, onde consta a identificação da arquitetura de origem. Sempre que uma nova mensagem é recebida, a *thread* receptora de cada processo verifica se é necessária a conversão e, em caso afirmativo, aplica procedimentos de conversão adequados ao conteúdo da mensagem. Estes procedimentos são bastante simples, e convertem cada tipo de dado para as representações adotadas nas arquiteturas SPARC e i486. Em relação às convenções utilizadas por estas arquiteturas, uma das diferenças mais marcantes está na ordem de armazenamento dos bytes na memória: *big-endian* (byte mais significativo no menor endereço de memória) na arquitetura SPARC, e *little-endian* (byte mais significativo no maior endereço de memória) na arquitetura i486. Para lidar com isso, os procedimentos de conversão rearranjam os bytes adequadamente sempre que manipulam tipos de dados que ocupam múltiplos bytes.

5.7 Implementação das Primitivas

A implementação das primitivas YALI inicia nas rotinas da biblioteca do sistema e se estende em funções desempenhadas pela *thread* receptora e pelas *threads* processadoras. O objetivo desta seção é fornecer uma descrição detalhada da implementação destas primitivas e, ao mesmo tempo, do funcionamento das várias *threads* que compõem o ambiente de execução de YALI.

5.7.1 Considerações Gerais

A seguir serão discutidos alguns procedimentos e características que são comuns à implementacão de várias primitivas.

Primitivas Baseadas em *Hashing*

Grande parte das primitivas YALI são implementadas com o auxílio de *hashing*. Esta técnica, inicialmente escolhida para gerenciar a distribuição de tuplas, foi também aplicada para dividir a tarefa de gerência de barreiras e funções globais entre os processos da aplicação. A idéia básica é que se pode extrair, de qualquer um destes elementos (tuplas, *templates*, barreiras e funções globais), uma chave que pode ser usada como argumento para uma função de *hash*. Esta função

recebe também o tamanho da tabela *hash* distribuída (neste caso, igual ao número de processos da aplicação), e produz um resultado que determina em qual entrada (processo) da tabela o elemento deve ser armazenado. Assim, cada processo fica responsável por gerenciar algumas tuplas, barreiras e funções globais, processando qualquer operação de manipulação destes elementos que venha a ser solicitada por *threads* locais ou residentes em outros processos.

A função de *hash* utilizada em YALI é bastante simples: é a mesma função que cada processo emprega para manutenção de suas tabelas *hash* locais, apenas modificando-se o argumento que indica o tamanho da tabela. A chave fornecida como argumento para esta função varia de acordo com o tipo de elemento a ser manipulado. No caso de tuplas e *templates*, a chave é composta pelas informações descritas em 5.3.1. A restrição de que o primeiro campo de qualquer tupla ou *template* deva ser sempre real foi imposta justamente para permitir a composição desta chave, a fim de se conseguir uma melhor diferenciação destes elementos e, conseqüentente, uma melhor distribuição de tuplas. No caso de barreiras e funções globais, a chave usada é mais simples, consistindo apenas na cadeia de caracteres que representa o nome destes elementos.

Com a função de *hash* adotada, elementos com a mesma chave são mapeados sempre em um mesmo processo, o que pode levar ao surgimento de um gargalo no sistema. No entanto, esta função tem a vantagem de permitir uma rápida localização dos elementos globais manipulados em YALI (tuplas, barreiras, funções), exigindo sempre um pequeno número de mensagens para implementação das primitivas, mesmo aumentando-se o número de processos participantes da aplicação. Apesar disso, futuramente pode-se estudar alguns mecanismos de *hashing* mais otimizados, como os propostos em [LUC86], [CAE94a] e [CAL91a], com vistas à sua incorporação em novas versões do sistema.

Em geral, as primitivas YALI baseadas em *hashing* (**y_out**, **y_in**, **y_rd**, **y_inp**, **y_rdp**, **y_reduce**, **y_gather**, **y_barrier**, **y_global** e **y_globeval**) são implementadas de maneira bastante semelhante na biblioteca do sistema. Basicamente, as rotinas que implementam estas primitivas constróem uma requisição a partir dos argumentos recebidos, e dela extraem uma chave que é utilizada para *hashing*. O resultado do *hashing*, como já foi mencionando, corresponde à identificação do processo que deve processar a requisição. É possível que o processo escolhido seja o mesmo onde reside a *thread* usuária requisitante, e neste caso uma *thread* processadora adequada é criada dinamicamente para atender a requisição, recebendo como argumento um ponteiro para o endereço de memória onde esta requisição está armazenada. Em caso contrário, a requisição é enviada ao processo de *hashing* através dos mecanismos de comunicação descritos na seção 5.5, onde será recebida pela *thread* receptora deste processo. Esta, por sua vez, irá disparar uma *thread* processadora capaz de atender a requisição. Estas duas situações no processamento de primitivas baseadas em *hashing* são ilustradas na figura 5.9.

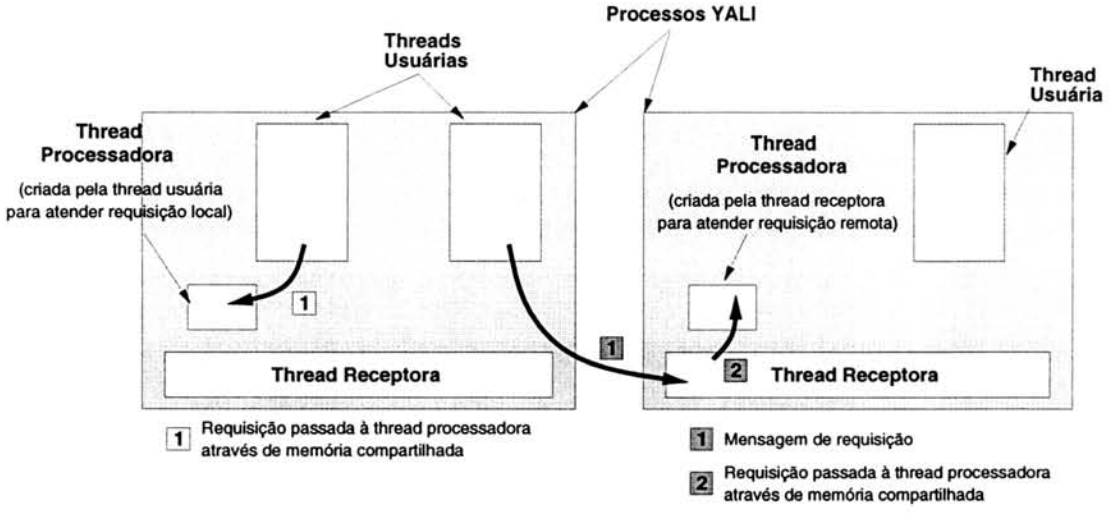

Figura 5.9 - Processamento de primitivas baseadas em *hashing*.

Primitivas Síncronas

YALI possui várias primitivas síncronas (**y_in, y_rd, y_inp, y_rdp, y_gather, y_reduce** e **y_barrier**), onde a *thread* usuária que as utiliza deve esperar pelo recebimento de algum resultado. Para isso, toda rotina que implementa uma primitiva síncrona aloca dinamicamente uma variável de condição (e um *mutex* associado) para aguardar a chegada de uma resposta, e adiciona-a à requisição construída. Além disso, se a rotina precisa manipular a resposta recebida, um ponteiro para o endereço de memória que conterá a resposta é criado dinamicamente, e seu endereço também é adicionado na requisição de operação produzida pela rotina. A resposta propriamente dita pode ser repassada à *thread* usuária de duas maneiras:

- quando a requisição é atendida por uma *thread* processadora **local** (no próprio processo), o ponteiro nela contido é atualizado, de maneira a apontar para a resposta correspondente. Após ajustar o ponteiro, a *thread* processadora desbloqueia a *thread* usuária através da variável de condição contida na requisição. A *thread* usuária, então, pode manipular a resposta adequadamente;

- quando a requisição é atendida por uma *thread* processadora **remota** (em outro processo), a resposta correspondente retorna ao processo requisitante através de uma mensagem, que é recebida pela *thread* receptora. Nesta mensagem são incluídos os endereços do ponteiro e da variável de condição originalmente contidos da requisição, para que a própria *thread* receptora seja capaz de desbloquear a *thread* usuária e repassar para ela a resposta recebida.

O objetivo do uso de variáveis de condição exclusivas para cada primitiva em execução é permitir que *threads* utilizem primitivas síncronas ao mesmo tempo, sem interferirem umas nas outras.

5.7.2 Primitivas de Manipulação de Tuplas

Na biblioteca do sistema, todas as primitivas para manipulação de tuplas têm uma interface semelhante, onde os argumentos iniciais devem fornecer uma descrição detalhada da tupla ou *template*, e os argumentos seguintes são constantes ou variáveis que representam campos reais ou formais. Esta interface, que não é utilizada diretamente pelo usuário, foi definida para facilitar a construção das estruturas de dados que representam tuplas ou *templates*.

y_out

A rotina que implementa a primitiva **y_out** inicia construindo uma requisição **y_out**, contendo a tupla a ser inserida. A chave extraída da tupla é fornecida como argumento para a função de *hash*, que determina o processo onde a tupla deve ser depositada. A partir daí, a rotina se comporta como descrito em 5.7.1, causando o disparo de uma *thread* processadora de requisições **y_out** no processo escolhido pela função de *hash*. Como a primitiva **y_out** é assíncrona, a *thread* usuária não precisa esperar nenhuma resposta da *thread* processadora que atendeu a requisição.

Uma *thread* processadora de requisições **y_out** tem a função de inserir a tupla recebida no espaço de tuplas local. Antes disso, no entanto, a *thread* precisa consultar a tabela de requisições de tupla pendentes na procura de *templates* que possam ser satisfeitos pela nova tupla. Se é encontrada uma pendência equivalente, a *thread* processadora se encarrega de respondê-la, usando uma das técnicas descritas em 5.7.1. São quatro os tipos de pendências que podem estar registrados na tabela:

- Pendência **y_in**: ao encontrar uma pendência **y_in** equivalente à tupla recebida, a *thread* processadora responde adequadamente a requisição pendente, remove a pendência e encerra sua execução. Não é necessário inserir a tupla no espaço local, já que requisições **y_in** destinam-se à remoção de tuplas;

- Pendência **y_rd**: ao encontrar uma pendência deste tipo, a *thread* processadora responde a requisição e continua procurando e respondendo pendências equivalentes, até encontrar uma pendência **y_in**, **y_reduce** ou **y_gather**. Se nenhuma pendência de um destes tipos é encontrada, a tupla é inserida no espaço local. Caso contrário, a *thread* responde a requisição pendente e encerra sua execução;

- Pendência **y_reduce**: uma pendência deste tipo contém um contador de tuplas pendentes associado, já que **y_reduce** deve remover várias tuplas.

Quando a *thread* processadora encontra uma pendência deste tipo que pode ser satisfeita pela tupla recebida, ela decrementa o contador de tuplas pendentes e responde à *thread* requisitante (como será visto mais adiante, esta *thread* requisitante é uma *thread* processadora residente no mesmo processo). A tupla não é inserida no espaço local, e a pendência só é removida quando o contador de tuplas chega a zero.

- Pendência **y_gather**: esta pendência é tratada pela *thread* processadora de maneira idêntica a uma pendência **y_reduce**.

A figura 5.10 ilustra duas situações na implementação de uma primitiva **y_out**. Na primeira, a requisição é atendida por uma *thread* processadora local, e uma pendência é encontrada, de modo que é enviada uma resposta adequada à *thread* requisitante remota. Na segunda situação, a requisição é atendida por uma *thread* processadora remota, que insere a tupla no seu espaço de tuplas local.

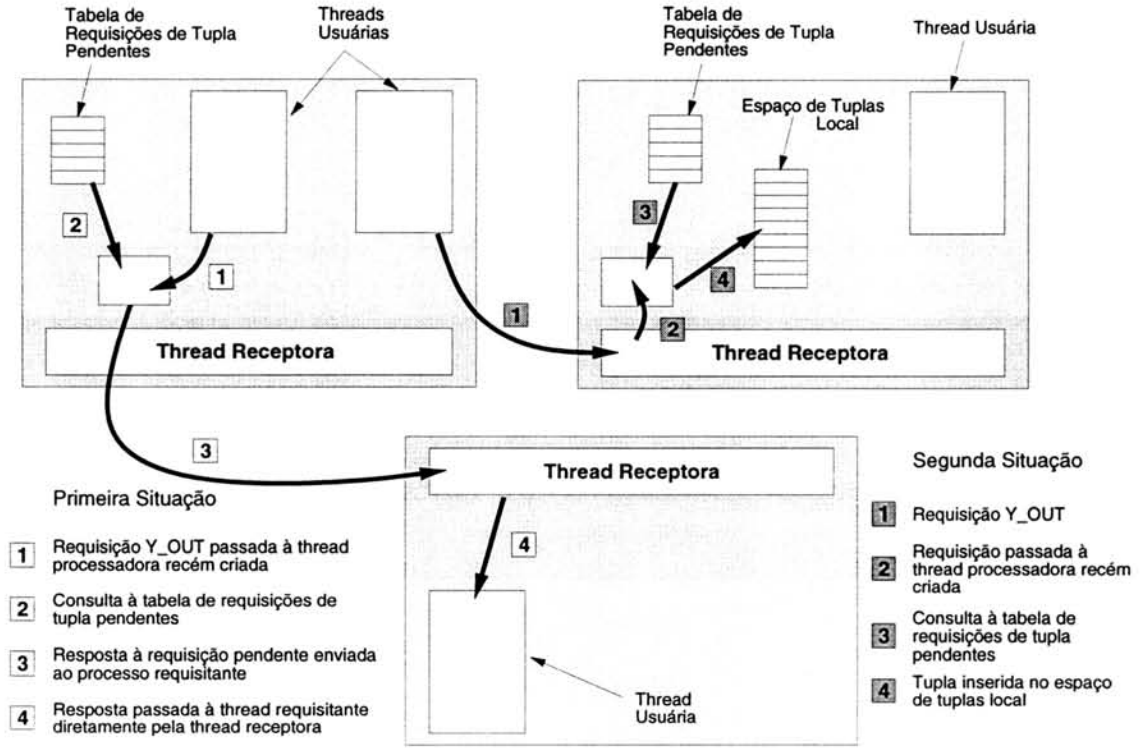

Figura 5.10 - Exemplo de implementação da primitiva **y_out**.

y_in e y_rd

As rotinas que implementam as primitivas **y_in** e **y_rd** na biblioteca do sistema são bastante semelhantes. Inicialmente, estas rotinas constróem, respectivamente, requisições **y_in** e **y_rd** contendo um *template* a ser usado na busca associativa. A seguir, através de *hashing*, é determinada a identificação do processo onde se pode encontrar tuplas equivalentes a este *template*. Dependendo do resultado do *hashing*, as rotinas podem causar o disparo de uma *thread* processadora local (no

próprio processo) ou remota (em outro processo da aplicação), conforme discutido em 5.7.1. Tanto **y_in** como **y_rd** são primitivas síncronas, portanto as rotinas que as implementam devem bloquear numa variável de condição esperando por uma resposta. Quando esta resposta é recebida, a tupla nela contida é usada para preencher os campos formais correspondentes do *template* satisfeito.

Devido à semelhança das operações **y_in** e **y_rd**, usa-se o mesmo tipo de *thread* processadora para atender estas requisições. A função de uma *thread* deste tipo é implementar uma busca associativa no espaço de tuplas local, a fim de encontrar uma tupla que satisfaça o *template* contido na requisição. Caso uma tupla seja encontrada, ela é repassada como resposta à *thread* requisitante, através de um dos métodos descritos anteriormente. Adicionalmente, se a requisição é do tipo **y_in**, a tupla é removida do espaço de tuplas local. Se nenhuma tupla é encontrada, entretanto, a requisição é armazenada na tabela de pendências local, para que possa ser satisfeita posteriormente com a chegada de novas tuplas.

A figura 5.11 ilustra duas situações na implementação de uma primitiva **y_in** ou **y_rd**. Na primeira, a requisição é atendida por uma *thread* processadora local, e a tupla é encontrada na porção local do espaço de tuplas. Assim, a resposta é passada à *thread* usuária requisistante diretamente através de memória compartilhada. Na segunda situação, a requisição é atendida por uma *thread* processadora remota, mas a tupla solicitada não é encontrada. Assim, uma pendência é armazenada na tabela de requisições pendentes do processo que contém a *thread* processadora.

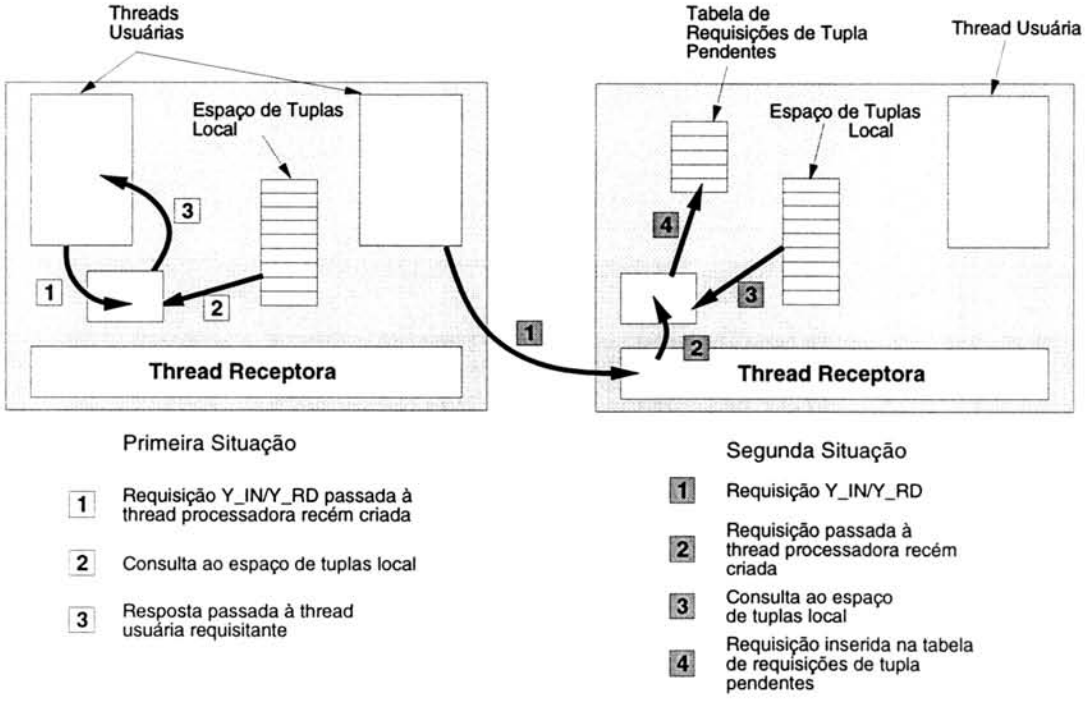

Figura 5.11 - Exemplo de implementação das primitivas **y_in/y_rd**.

y_inp e y_rdp

A implementação das primitivas **y_inp** e **y_rdp** é semelhante à implementação de suas versões bloqueantes, **y_in** e **y_rd**. A única diferença entre as rotinas que implementam **y_inp** e **y_rdp** e aquelas que implementam **y_in** e **y_rd** está na resposta esperada por elas: enquanto respostas a uma requisição **y_in** ou **y_rd** devem necessariamente conter uma tupla, respostas a uma requisição **y_inp** ou **y_rdp** não precisam satisfazer esta exigência. É importante notar que, embora estas primitivas sejam não-bloqueantes a nível de interface, as rotinas que as implementam precisam bloquear na espera de uma resposta que indique se uma tupla equivalente ao *template* requisitado foi ou não encontrada no processo escolhido pela função de *hash*. Ao receber uma resposta positiva, estas rotinas se comportam como suas versões bloqueantes, e adicionalmente retornam o valor 1. Do contrário, simplesmente retornam o valor 0, indicando que nenhuma tupla foi encontrada.

Requisições **y_inp** e **y_rdp** são atendidas por *threads* processadoras do mesmo tipo, cuja função é realizar uma busca associativa no espaço de tuplas local, a fim de determinar a existência de uma tupla que satisfaça um dado *template* e, a seguir, retornar uma resposta à *thread* requisitante. Esta resposta é enviada através de um dos métodos descritos anteriormente (diretamente à *thread* usuária ou por intermédio da *thread* receptora), e pode conter ou não uma tupla, dependendo do resultado da busca associativa. No caso desta busca ter sido bem sucedida, e sendo a requisição do tipo **y_inp**, a *thread* processadora também remove a tupla encontrada.

É importante observar que, com a política de distribuição baseada em *hashing*, o número de mensagens necessárias para a implementação de **y_inp** e **y_rdp** é o mesmo envolvido na implementação de **y_in** e **y_rd**. Quando uma requisição é executada em um processo diferente daquele que a produziu, apenas duas mensagens são necessárias (uma requisição e uma resposta). Em caso contrário, nenhuma mensagem precisa ser trocada entre processos. Esta possibilidade de implementação de **y_inp** e **y_rdp** com um pequeno número de mensagens foi um dos principais motivos da inclusão destas primitivas não-bloqueantes em YALI.

5.7.3 Operações Globais

A implementação de operações globais em YALI tem várias particularidades, mas se assemelha às primitivas já apresentadas no que diz respeito à utilização de *hashing* para determinar o processo que deve atender a uma dada requisição.

y_gather e y_reduce

As particularidades de implementação destas primitivas são visíveis tanto nas rotinas da biblioteca como no funcionamento das *threads* processadoras que as executam. No que diz respeito às rotinas da biblioteca, a implementação de **y_gather** e **y_reduce** é inicialmente semelhante à da primitiva **y_in**, já que todas

as três recebem um *template* como parâmetro, e devem enviá-lo numa requisição a um processo escolhido através de *hashing*. Diferentemente de **y_in**, entretanto, os parâmetros fornecidos a **y_gather** incluem o número de tuplas a recuperar, enquanto os parâmetros para **y_reduce** incluem, além deste número, também a identificação das funções de redução que são aplicadas aos campos formais do *template*. As requisições produzidas por cada uma destas primitivas, portanto, carregam mais informações que as demais requisições de recuperação de tuplas.

A principal particularidade, no entanto, está nas *threads* processadoras responsáveis pelo atendimento de requisições **y_gather** ou **y_reduce**. Uma *thread* processadora de **y_gather** tem a função de realizar uma busca associativa no espaço de tuplas local e coletar tantas tuplas quantas foram solicitadas na requisição, desde que sejam equivalentes ao *template* recebido. É possível, no entanto, que o número solicitado de tuplas não esteja totalmente disponível, e neste caso é necessário armazenar a requisição na tabela de pendências local. Esta requisição, como já foi mencionado anteriormente, mantém consigo o número de tuplas pendentes restantes, que é atualizado por *threads* processadoras de requisições **y_out**. Ao contrário das *threads* que processam outras primitivas de recuperação de tuplas, uma *thread* processadora de **y_gather** não encerra sua execução após o registro das pendências. Ela precisa esperar que o número especificado de tuplas esteja disponível localmente, para só então construir a resposta a ser enviada à *thread* usuária requisitante. Esta resposta contém todas as tuplas reunidas, permitindo que a rotina que implementa **y_gather** possa construir vetores com os valores coletados para cada campo formal.

Uma *thread* processadora de requisições **y_reduce** funciona de maneira bastante semelhante a uma *thread* que atende requisições **y_gather**. A única diferença é que, após a coleta das tuplas solicitadas, a *thread* processa os valores coletados para determinados campos formais utilizando as funções de redução especificadas, de modo a produzir uma única tupla como resposta à *thread* solicitante.

A figura 5.12 exemplifica a implementação de uma primitiva **y_gather** ou **y_reduce** numa situação em que a requisição é atendida por uma *thread* processadora remota, escolhida através de *hashing*. Neste exemplo, a quantidade de tuplas especificada não é encontrada no espaço de tuplas, sendo necessário o registro de uma pendência na tabela de requisições de tupla. O exemplo ilustra ainda o surgimento de uma requisição **y_out** que satisfaz a pendência **y_gather/y_reduce** recém inserida, do modo que a *thread* processadora de **y_gather/y_reduce** consegue terminar o processamento da primitiva e responder à *thread* usuária requisitante. Neste exemplo, a implementação das primitivas exigiu somente o tráfego de duas mensagens pela rede (uma requisição e uma resposta).

Deve-se notar que a implementação destas operações globais requer um pequeno número de mensagens trocadas entre processos: apenas duas quando a *thread* usuária e a *thread* processadora residem em processos diferentes, ou mesmo nenhuma quando as *threads* residem no mesmo processo (neste caso, a comunicação se dá através de memória compartilhada). Isto é claramente mais eficiente do que uma implementação destas operações através de várias chamadas **y_in**. Além disso, no caso de **y_reduce**, as reduções são executadas pelo processo que

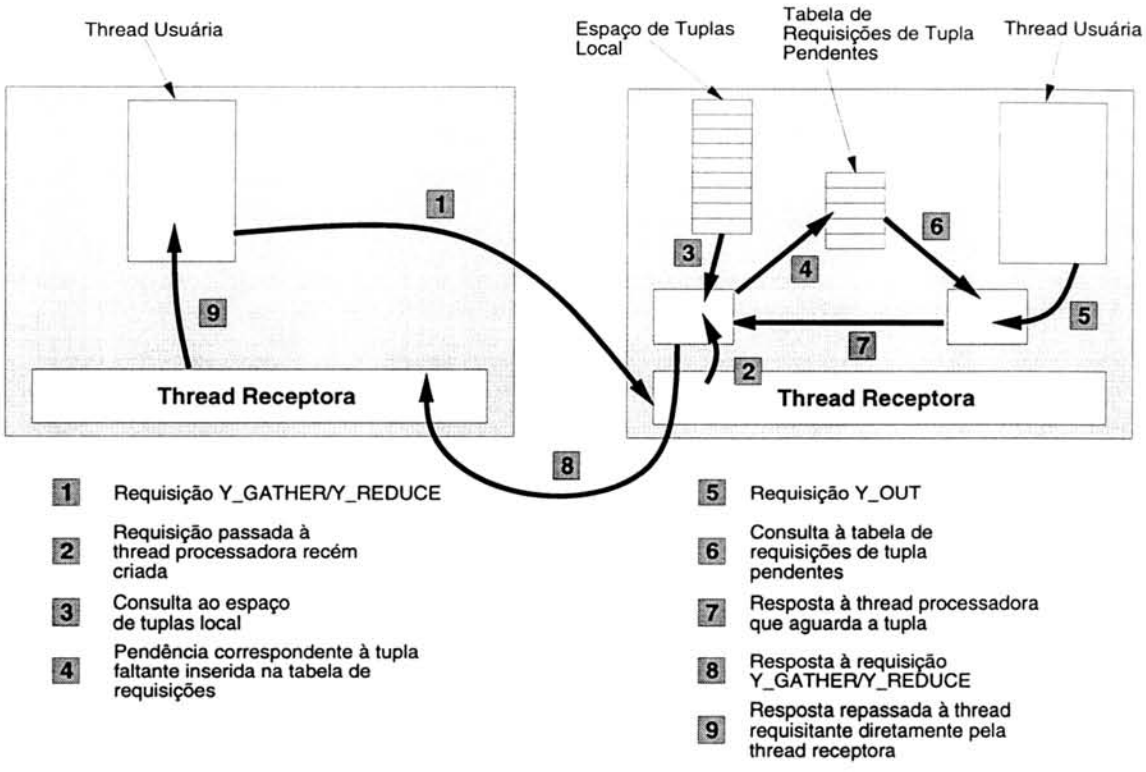

Figura 5.12 - Exemplo de implementação das primitivas **y_gather/y_reduce**.

mantém as tuplas, de modo que o processo requisitante fica livre desta carga de processamento.

y_barrier

A rotina que implementa esta primitiva recebe como argumentos apenas um nome de barreira e o número de *threads* que devem sincronizar nesta barreira. Devido à simplicidade de sua interface, uma chamada a esta primitiva não precisa ser pré-processada, podendo ser utilizada diretamente pelo usuário. Como já foi mencionado anteriormente, a implementação de **y_barrier** também é baseada em *hashing*, e emprega como chave o próprio nome associado à barreira. O funcionamento da rotina que implementa **y_barrier** é semelhante ao das outras rotinas baseadas em *hashing*, com a diferença de que a resposta consiste apenas num valor -1 ou 0, indicando, respectivamente, um erro ou uma operação bem sucedida. Como outras rotinas que implementam operações síncronas, **y_barrier** utiliza também uma variável de condição criada dinamicamente, com a finalidade de bloquear a *thread* usuária na espera de uma indicação de que as outras *threads* envolvidas também tenham alcançado a barreira.

Uma *thread* processadora de requisições **y_barrier** funciona da seguinte maneira:

- inicialmente, o nome de barreira contido na requisição é procurado na tabela local de barreiras;

- caso não seja encontrada uma barreira com este nome, um novo registro de barreira é inserido na tabela, e o contador de *threads* a serem sincronizadas é inicializado com o número fornecido na requisição. Adicionalmente, a *thread* processadora também armazena na tabela as informações necessárias para desbloquear a *thread* usuária requisitante no momento apropriado;
- caso a barreira já esteja registrada, a *thread* verifica se o número de *threads* a serem sincronizadas contido na tabela é igual ao número contido na requisição. Se os números são diferentes, a *thread* processadora responde imediatamente à *thread* requisitante, sinalizando a ocorrência de erro (-1). Se os números são iguais, a *thread* decrementa o contador de *threads* a serem sincronizadas e adiciona a identificação da nova *thread* que alcançou a barreira. Quando este contador chega a zero, a *thread* processadora causa a liberação de todas as *threads* usuárias bloqueadas e remove da tabela o registro correspondente a esta barreira. Neste caso, a resposta enviada sinaliza operação bem sucedida (0). Independentemente do sucesso da operação, toda resposta enviada pela *thread* processadora segue o procedimento descrito em 5.7.1;
- por fim, após a atualização da tabela de barreiras, a *thread* processadora encerra sua execução.

A figura 5.13 exemplifica a implementação de **y_barrier** numa situação em que três *threads* usuárias devem sincronizar na mesma barreira. Duas destas *threads* residem num processo diferente daquele que gerencia a barreira em questão, e por isso produzem requisições que são enviadas pela rede. As *threads* processadoras disparadas para atender estas requisições simplesmente atualizam a tabela de barreiras. Quando a última *thread* usuária atinge a barreira, a *thread* processadora correspondente desbloqueia todas as *threads* usuárias sincronizadas.

A implementação de barreiras em YALI é claramente mais eficiente que uma implementação envolvendo primitivas de manipulação de tuplas, como a exemplificada na figura 5.14 (esta implementação já foi discutida no capítulo 3, seção 2.3.3). Considerando-se que tupla usada neste exemplo não fosse armazenada em nenhum dos processos cujas *threads* participam da sincronização de barreira, o número de mensagens necessárias para sincronizar N *threads* seria igual a $1 + N * 5$, sendo uma mensagem para a operação **y_out** que inicializa a barreira e, para cada processo, uma mensagem a cada operação **y_out** e duas mensagens a cada operação **y_in** ou **y_rd**. Utilizando-se a primitiva **y_barrier**, entretanto, o número de mensagens necessárias para a sincronização de N *threads* seria apenas $N * 2$, também considerando-se que a barreira não fosse gerenciada por um processo participante.

5.7.4 Primitivas para Expressão do Paralelismo

As primitivas para expressão do paralelismo têm em comum o fato de que não precisam ser pré-processadas, pois exigem um pequeno número de argumentos, que podem ser facilmente fornecidos pelo usuário.

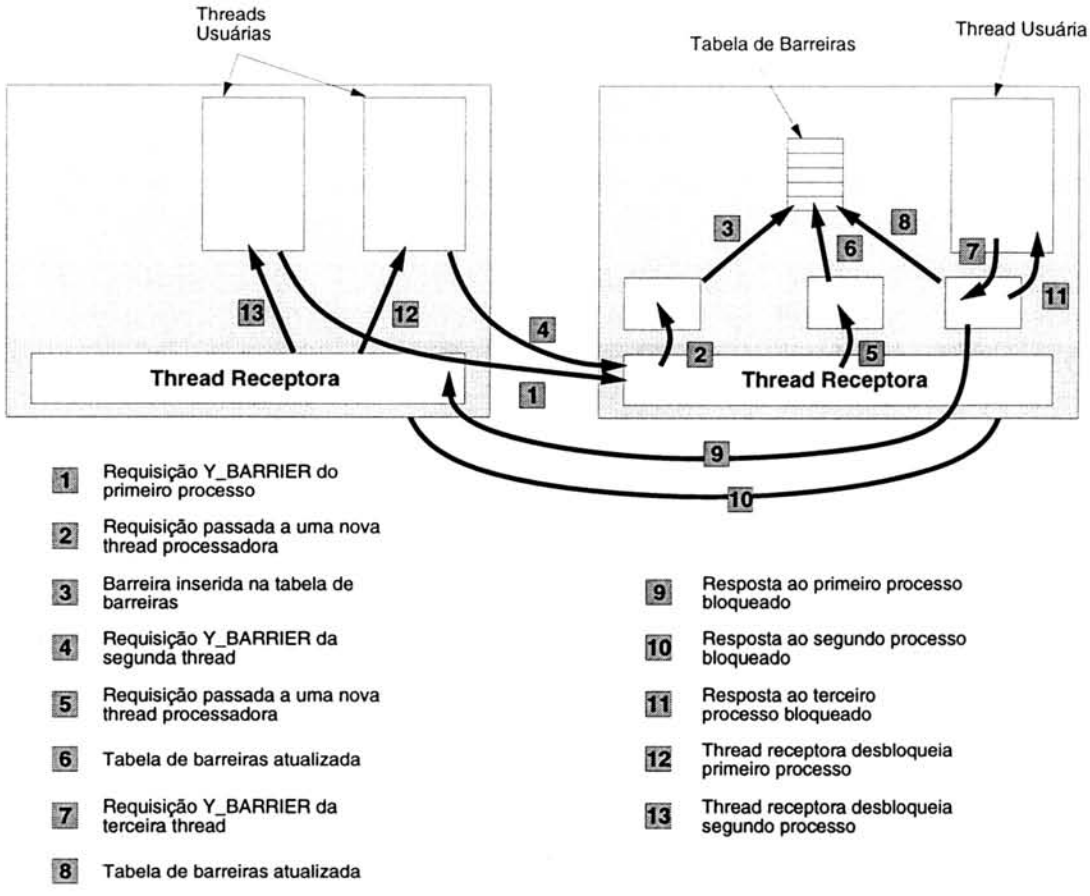

Figura 5.13 - Exemplo de implementação da primitiva **y_barrier**.

Para a implementação das primitivas que manipulam funções globais é utilizada uma estrutura de dados global (omitida na seção 5.3), que mantém exclusivamente requisições **y_globeval** pendentes. Uma pendência deste tipo acontece quando o nome de função especificado numa chamada **y_globeval** não está registrado na tabela de funções globais do processo que deveria mantê-lo. A causa disto poderia ser, por exemplo, um atraso no processamento da operação **y_global** que deveria registrar o nome global da função. Optou-se por omitir a descrição desta tabela de pendências na seção 5.3 por tratar-se de uma estrutura de dados bastante específica, cujo propósito seria melhor compreendido ao ser discutida a implementação das primitivas **y_global** e **y_globeval**.

y_eval

Esta é uma das poucas primitivas que são implementadas sem o auxílio de *hashing*. A razão disto é que **y_eval** serve para disparar uma *thread* local ao processo que a utiliza, não necessitando, portanto, do envolvimento de outros processos.

A rotina que implementa esta primitiva é bastante simples: o argumento recebido, que representa o endereço de uma função local, é passado como parâmetro para uma rotina de criação de *thread*, que então se encarrega de realmente dis-

```
/* inicializacao da barreira */
y_out("barrier", N);
y_out("barrier", n-1);
y_rd("barrier", 0);
```

```
/* codigo executado por cada thread */
y_in("barrier", ?n);
```

Figura 5.14 - Implementação de barreiras através de uma combinação de primitivas.

parar uma nova *thread*. Esta rotina deve estar disponível na biblioteca de *threads* que acompanha cada plataforma suportada por YALI. As bibliotecas do sistema Mach e do sistema Solaris suportam rotinas semanticamente semelhantes para a criação de *threads*.

y_global

A rotina que implementa **y_global** recebe como parâmetro um nome (uma cadeia de caracteres) e um endereço referentes a uma função que deve ser tornada global. Como visto em 4.2.2, uma função global pode ter sua execução disparada por qualquer processo da aplicação. Os argumentos recebidos são utilizados para construir uma requisição **y_global**, e o nome da função é utilizado como chave de *hashing* para determinar onde esta requisição deve ser processada.

O funcionamento de uma *thread* processadora de requisições **y_global** segue os passos descritos abaixo:

- o nome de função contido na requisição é procurado na tabela de funções globais;
- caso não seja encontrada uma função global com este nome, a *thread* processadora acrescenta um novo registro na tabela e inicializa uma lista de processos que implementam funções com este nome. Cada elemento desta lista contém informações que permitem a posterior execução da função (basicamente, a identificação de um processo e o endereço da função global neste processo);
- caso a função já esteja registrada, é sinal de que outro processo já implementa uma função com o mesmo nome. Nesta situação, a *thread* processadora simplesmente complementa a lista de processos que implementam a função;
- após a atualização da tabela de funções globais, a *thread* processadora consulta a tabela de requisições **y_globeval** pendentes, na procura de referências à função recém registrada. Para cada referência encontrada é criada uma nova *thread* processadora de **y_globeval**, de modo a atender requisições que ainda não tinham sido satisfeitas.
- depois de processar e remover as pendências que tenham sido encontradas, a *thread* processadora, por fim, encerra sua execução.

y_globeval

A rotina que implementa **y_globeval** na biblioteca do sistema é bastante semelhante às demais rotinas baseadas em *hashing*. O nome de função fornecido como parâmetro para **y_globeval** é usado como chave de *hashing*, a fim de identificar o processo que mantém informações sobre esta função global. Este processo, por sua vez, deve disparar uma *thread* processadora de requisições **y_globeval**.

A tarefa de uma *thread* processadora de **y_globeval** é usar o nome de função que lhe é fornecido para consultar sua tabela de funções globais, com o objetivo de identificar os processos que implementam tal função. A seguir, a *thread* processadora age como **escalonadora**, escolhendo um ou mais processos para executar a função, de acordo com o número especificado na chamada **y_globeval**. A fim de evitar que um processo seja sobrecarregado quando **y_globeval** deve disparar várias *threads* (ou quando são feitas várias chamadas consecutivas à mesma função global), a *thread* processadora utiliza uma política de escalonamento semelhante à política *round-robin* utilizada em sistemas operacionais. Segundo esta política, o processo escolhido é sempre aquele no início da lista de processos que implementam a função. A cada execução da função, no entanto, o processo escolhido é colocado no fim da lista, evitando que seja escolhido em chamadas subseqüentes à mesma função.

Após a escolha de um ou mais processos para executar a função global, a *thread* processadora deve repassar a eles uma solicitação de execução da função. É possível que um dos processos escolhidos seja o mesmo onde a *thread* processadora executa, e neste caso ela própria pode disparar uma ou mais *threads* para processar a função, já que esta faz parte de seu próprio espaço de endereçamento. No caso de *threads* que precisam ser disparadas em processos diferentes, a *thread* processadora utiliza os mecanismos de comunicação descritos em 5.5 para enviar-lhes requisições de execução apropriadas. Estas requisições serão recebidas pela *thread* receptora de cada processo, que então irão criar uma ou mais *threads* para executar a função global. Após enviar as requisições de execução (ou disparar *threads* no próprio processo), a *thread* processadora encerra sua execução, já que as novas *threads* são totalmente independentes e, por isso, não é necessário esperar por seu término.

Quando a *thread* processadora consulta sua tabela de funções globais, é possível que o nome de função procurado ainda não tenha sido registrado. Neste caso, a *thread* processadora adiciona o nome da função na tabela de requisições **y_globeval** pendentes e, a seguir, encerra sua execução.

A figura 5.15 ilustra a implementação de uma primitiva **y_global** e, a seguir, de uma chamada **y_globeval** correspondente. A chamada **y_global** gera uma requisição a uma *thread* processadora remota, que simplesmente atualiza a tabela de funções globais. A chamada **y_globeval**, executada em um processo diferente daquele que implementa a função, também produz uma requisição a uma *thread* processadora remota, que se encarrega de solicitar a execução da função no processo adequado.

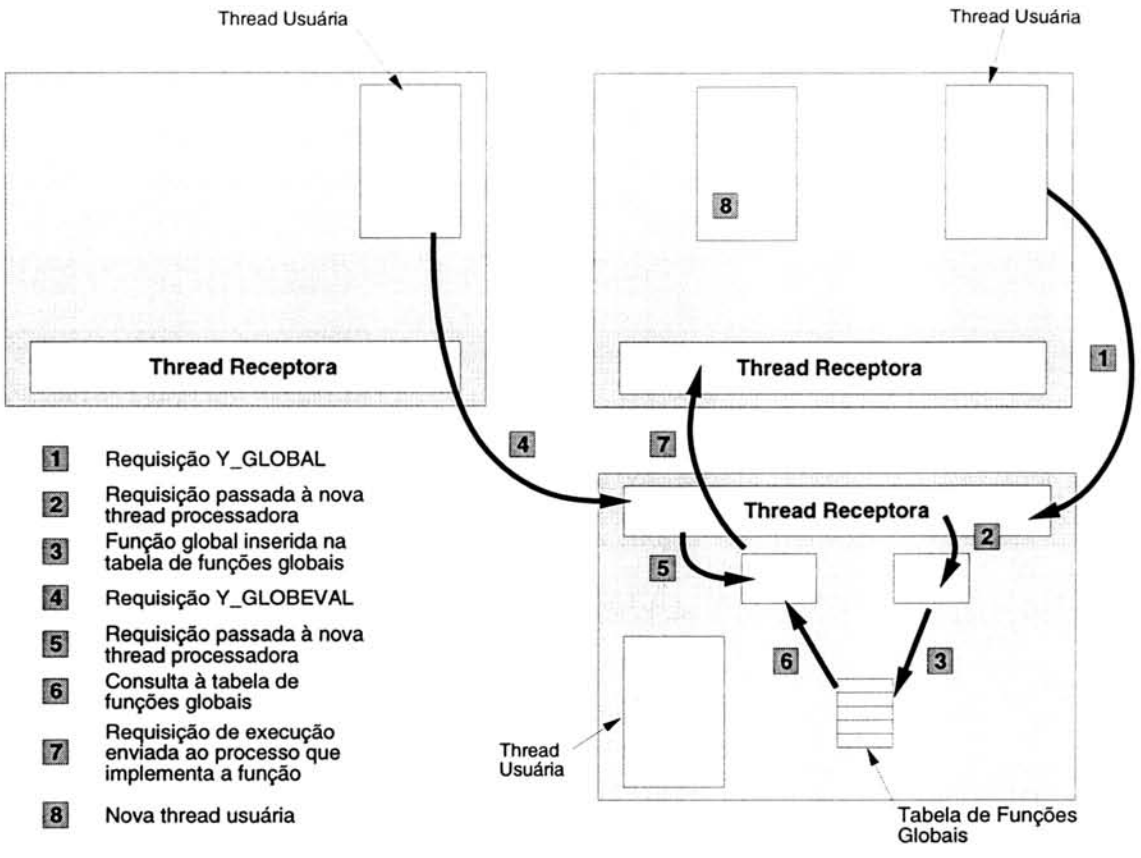

Figura 5.15 - Exemplo de implementação de **y_global** e **y_globeval**.

5.7.5 Outras Primitivas

A primitiva **y_id** deve retornar a identificação do processo que a executa, enquanto **y_nproc** deve fornecer o número de processos da aplicação. Estas informações são obtidas por cada processo durante a inicialização da aplicação, e são mantidas em variáveis globais acessíveis por todas as *threads* do processo. A implementação destas primitivas, portanto, consiste em simplesmente retornar o valor armazenado nestas variáveis.

5.8 Inicialização e Término de Aplicações

Conforme visto no capítulo anterior, a inicialização de uma aplicação YALI é feita através do utilitário `yalistart`. O procedimento geral de inicialização, no entanto, envolve também os próprios processos disparados por `yalistart`, que precisam inicializar sua estrutura local de execução (*threads* e estruturas de dados, basicamente), e precisam interagir com `yalistart` para descobrir informações sobre o restante dos processos. O procedimento geral de inicialização, portanto, pode ser resumido em duas etapas principais: **criação de processos** e **inicialização individual de processos**.

Uma particularidade de YALI é que o programa de inicialização permanece ativo enquanto a aplicação executa, podendo ser visto como uma espécie de monitor da aplicação. Após as etapas de inicialização, `yalistart` passa a monitorar os processos criados, esperando pelo término de suas funções principais, a fim de assegurar o término correto da aplicação como um todo. O procedimento de término de aplicações YALI, bem como as etapas do procedimento de inicialização, são descritos detalhadamente a seguir.

5.8.1 Criação de Processos

Em `yalistart`, a criação de processos é feita localmente através da chamada `fork` do Unix, e remotamente através do utilitário `rsh`. Antes de disparar cada processo, entretanto, `yalistart` cria um *socket* para a posterior comunicação com os processos da aplicação. O endereço deste *socket* (número de porta) e a identificação da máquina onde `yalistart` executa são informações passadas como parâmetro para cada processo criado. A estas informações também são adicionados o número de identificação do processo e o número total de processos que compõem a aplicação (não incluindo o processo `yalistart`).

Processos criados remotamente através de `rsh` têm seu dispositivo padrão de saída redirecionado, de modo que qualquer informação por eles produzida aparece no terminal onde `yalistart` executa. O dispositivo padrão de entrada, entretanto, é fechado para todos os processos com exceção do primeiro, ao qual é permitido o uso de funções de entrada quando este reside no mesmo nodo que `yalistart`.

5.8.2 Inicialização Individual de Processos

Cada processo YALI executa uma função principal (`main`) idêntica, implementada na biblioteca do sistema. Esta função executa uma série de procedimentos de inicialização individual do processo, que incluem a troca de algumas informações com `yalistart`. A seqüência de procedimentos de inicialização executada por cada processo é a seguinte:

1. as estruturas de dados globais do processo são inicializadas adequadamente, com exceção da tabela de processos, que é construída em outra etapa;
2. é criado um conjunto de *sockets* para recepção e envio de mensagens, conforme visto na seção 5.5;
3. os endereços dos *sockets* de recepção são enviados a `yalistart`;
4. após receber endereços de *sockets* de todos os processos, `yalistart` reúne as informações recebidas e as envia a cada processo da aplicação;
5. ao receber estas informações de `yalistart`, o processo constrói sua tabela de processos, que permitirá o estabelecimento de comunicação com os demais processos da aplicação;

6. a *thread* receptora é colocada em execução, passando a monitorar todos os canais (*sockets*) de recepção;

7. por fim, é criada uma *thread* para executar a função principal definida pelo usuário (`yali_main`).

5.8.3 Procedimento de Término de Aplicações

O término de uma aplicação YALI ocorre quando as funções yali_main de todos os processos encerram sua execução. Como yali_main é executada por uma *thread*, seu término não influi no funcionamento das outras *threads*, de modo que o processo pode continuar a atender requisições. O procedimento de término de uma aplicação YALI segue as seguintes etapas:

1. após os procedimentos de inicialização, a função main de cada processo aguarda o término da *thread* criada para executar yali_main;

2. quando yali_main termina, uma mensagem é enviada a `yalistart`, sinalizando esta ocorrência;

3. após reunir notificações de término de todos os processos, `yalistart` envia a cada um deles uma requisição de término;

4. esta requisição é recebida pela *thread* receptora de cada processo, que avisa a função principal (main) e, a seguir, encerra sua execução;

5. a função principal também termina, encerrando finalmente o processo;

6. após o término de todos os processos, `yalistart` também encerra sua execução.

5.9 Implementação do Pré-Processador

O pré-processador `ypp` foi construído a partir de uma gramática para a linguagem C obtida em `ftp://ftp.iecc.com/pub/articles/91-09.gz`[1]. A gramática foi modificada para suportar as primitivas YALI e, para que fosse possível a identificação de tipos e tamanhos de campos, uma tabela de símbolos foi adicionada.

Uma particularidade na implementação de `ypp` é que ele invoca o pré-processador C do sistema hospedeiro, e usa o resultado deste primeiro pré-processamento como entrada. Com isso, diretivas do tipo `#include` ou `#define` já estão processadas quando `ypp` inicia sua análise, e portanto as declarações contidas em arquivos de cabeçalho já estão incluídas. A vantagem disso está, por exemplo, em permitir uma chamada do tipo:

```
y_out("square", sqr(4));
```

[1] *newsgroup* comp.compilers, mensagem 91-09-030

onde `sqr` é uma função declarada em `math.h`, que retorna um valor do tipo `double`. Se as declarações contidas em `math.h` não fossem incluídas, o pré-processador não teria como descobrir o tipo de dado retornado por `sqr` e, conseqüentemente, a chamada acima não poderia ser traduzida corretamente.

Para ilustrar o funcionamento de `ypp` e fornecer uma idéia da interface real de algumas primitivas, a figura 5.16 contém um pequeno programa YALI, e a figura 5.17 apresenta o código correspondente gerado por `ypp` (nesta figura, parte do código gerado pelo pré-processador foi omitido, para se obter maior clareza).

```
#include <math.h>

yali_main () {

    double  num;
    int     m[45];
    int     size, *p;

    y_out("matrix", m);
    y_out("square root", sqrt(44.55));
    y_rd("square root", ?num);
    y_out(num+1);

    size = 45;
    p = (int *) malloc(size);
    linda_in("matrix", ?p:size);
}
```

Figura 5.16 - Código fonte YALI.

5.10 Sumário

O capítulo apresentou diversos detalhes sobre a implementação de YALI. Dentre as características marcantes desta implementação, convém destacar as seguintes:

- o ambiente de execução de YALI é implementado através de **múltiplas *threads*** que residem em cada processo da aplicação. Esta estrutura permite que cada processo seja capaz de gerenciar qualquer tipo de elemento do sistema (tuplas, barreiras e funções globais) e, ao mesmo tempo, executar a parte que lhe cabe da computação paralela;
- a política usada para distribuir tuplas, barreiras e funções globais entre os processos da aplicação é baseada em *hashing*, o que permite uma rápida localização destes elementos no sistema. Além disso, uma política baseada

```
extern double sqrt();

yali_main () {

  double  num;
  int     m[45];
  int     size, *p;

  y_out( ACTUAL_CHAR_ARRAY, 7, ACTUAL_INT_ARRAY, 45, 0,
         "matrix", m);
  y_out( ACTUAL_CHAR_ARRAY, 12, ACTUAL_DOUBLE, 1, 0,
         "square root",   sqrt(44.55));
  y_rd( ACTUAL_CHAR_ARRAY, 12, FORMAL_DOUBLE, 1, 0,
        "square root", &num);
  y_out( ACTUAL_DOUBLE, 1, 0, num+1);

  size = 45;
  p = (int *) malloc(size);
  linda_in( ACTUAL_CHAR_ARRAY, 7, FORMAL_INT_ARRAY, &size,
            0, "matrix", p);
}
```

Figura 5.17 - Código fonte gerado pelo pré-processador.

em *hashing* tem a vantagem de ser bastante extensível, já que exige sempre um pequeno número de mensagens para implementação das primitivas do sistema, independentemente do número de processos que compõem a aplicação;

- YALI não utiliza ferramentas intermediárias para implementar a comunicação entre processos, e não emprega bibliotecas adicionais para o suporte à heterogeneidade. Por razões de eficiência, toda a interação entre processos é implementada através de recursos suportados pelo próprio sistema operacional hospedeiro, e as transferências de dados através de máquinas heterogêneas são feitas de modo a evitar a manipulação excessiva de *buffers* para codificação e decodificação de mensagens;

- a implementação das operações globais **y_barrier**, **y_gather** e **y_reduce** é feita de maneira a exigir pouca troca de mensagens, o que é claramente mais eficiente do que a simulação destas primitivas através de uma combinação de **out**, **in** e **rd**. Além disso, no caso de **y_reduce**, as reduções são executadas pelo processo que armazena as tuplas, o que permite uma melhor distribuição da carga de processamento desta operação.

A implementação de um sistema com as características acima envolve várias dificuldades. Em primeiro lugar, ao implementar YALI em duas plataformas diferentes, vários cuidados precisam ser tomados para garantir um grau satisfatório

de portabilidade ao sistema. Além disso, o uso de *threads* no sistema, embora vantajoso, introduz uma complexidade adicional na implementação, principalmente com relação à correta sincronização dos vários tipos de *threads*. Por fim, a opção por alternativas mais eficientes para implementar a comunicação entre processos e o suporte à heterogeneidade também aumentam a complexidade da implementação mas, ao mesmo tempo, permite que se lide mais diretamente com diversas questões relacionadas à implementação de um ambiente de execução distribuído.

6 Avaliação do Sistema

O propósito deste capítulo é avaliar os recursos oferecidos por YALI tanto qualitativa como quantitativamente. Para isso, inicialmente é feita uma comparação entre YALI e os sistemas estudados no capítulo 3, levando em conta tanto aspectos de interface como de implementação. A seguir, analisa-se a expressividade do sistema através de quatro exemplos de aplicações paralelas que utilizam diferentes recursos oferecidos por YALI e, por fim, apresenta-se alguns resultados de desempenho obtidos com o protótipo do sistema.

6.1 Comparação com os Sistemas Estudados

As tabelas 6.1 e 6.2 reapresentam as principais soluções empregadas nos ambientes de programação e execução de cada sistema estudado, desta vez incluindo também as soluções que YALI adota para as várias questões consideradas. Analisando-se as informações reunidas na tabela 6.1, pode-se notar que YALI destaca-se por ser o único[1] que implementa operações globais ao modelo Linda. Além disso, o sistema possui algumas características vantajosas do ponto de vista do ambiente de programação:

- o suporte a diferentes métodos de estruturação de aplicações, que oferece maior flexibilidade aos usuários;
- a disponibilidade das primitivas **y_inp** e **y_rdp**, que aumentam o poder de expressão do sistema;
- o uso de um pré-processador para incorporar o modelo à linguagem hospedeira, que garante uma interface simples para o usuário.

Com relação ao ambiente de execução, YALI destaca-se por implementar a comunicação entre processos diretamente através de *sockets*, e por não utilizar processos especiais para gerenciamento do espaço de tuplas, o que contribui favoravelmente para o desempenho do sistema. Além disso, a política adotada para distribuição de tuplas é extensível, e facilita a implementação das primitivas não-bloqueantes, que são preteridas por alguns dos sistemas estudados.

6.2 Expressividade do Sistema

As aplicações escolhidas para ilustrar a expressividade do sistema implementam algoritmos paralelos para cálculo do Jacobiano de uma matriz[KLE96], cál-

[1]Os sistemas Glenda e p4-Linda são construídos, respectivamente, a partir das ferramentas PVM e p4, que oferecem algumas operações globais exclusivamente voltadas ao paradigma de troca de mensagens. No entanto, nem Glenda nem p4-Linda incorporam estas operações ao modelo Linda.

Tabela 6.1 - Comparação entre os ambientes de programação de YALI e dos demais sistemas estudados.

Sistemas	Estrutura das Aplicações	EVAL	INP/ RDP	Operações Globais	Incorporação à Ling. Hospedeira
Glenda	SPMD/MPMD	substituída por **gl_spawn**, que usa PVM para criação de processos	sim	não	pré-processador
POSYBL	MPMD	dispara a execução de um novo programa	não	não	biblioteca
p4-Linda	SPMD	executa função em paralelo porém sem inserção de tupla	não	não	biblioteca
Network-Linda	SPMD	semântica original	sim	não	pré-compilador
YALI	SPMD/MPMD	substituída por primitivas para criação de *threads*	sim	sim	pré-processador

Tabela 6.2 - Comparação entre os ambientes de execução de YALI e dos demais sistemas estudados.

Sistemas	Política de Distribuição	Suporte à Heterogeneidade	Mecanismo de Comunicação entre Processos	Processo Gerenciador do TS
Glenda	centralizada	sim	PVM	sim
POSYBL	uniforme	limitado	RPC, *sockets*	sim
p4-Linda	centralizada	sim	p4	sim
Network-Linda	várias políticas	sim	*sockets*	não
YALI	*hashing*	sim	*sockets*	não

culo de π com alta aproximação[KAR88], determinação de números primos num dado intervalo[CAR89a] e, por fim, multiplicação de matrizes quadradas.

6.2.1 Iteração de Jacobi

Este é um exemplo de algoritmo de relaxação conhecido como "iteração de Jacobi", usado com freqüência na solução da equação de Laplace a fim de determinar, por exemplo, a temperatura estável de um corpo quando a temperatura de suas bordas é mantida fixa. O algoritmo obtém a solução iterativamente: a cada iteração, a temperatura de um ponto é calculada como sendo a média da temperatura de quatro pontos "vizinhos". Assumindo que o espaço ocupado pelo corpo tenha sido discretizado na forma de uma matriz bi-dimensional, $T_i(x, y)$ representa a temperatura num ponto (x, y) depois da i-ésima iteração do algoritmo. A cada iteração calcula-se uma nova matriz, onde a temperatura em cada ponto é dada por:

$$T_{(i+1)}(x, y) = \frac{(T_i(x-1, y) + T_i(x+1, y) + T_i(x, y-1) + T_i(x, y+1))}{4}$$

A diferença de temperatura em cada ponto (x, y) é calculada como sendo o módulo de $T_{(i+1)}(x, y) - T_i(x, y)$. O algoritmo converge, de modo que pode ser executado até que a diferença máxima seja inferior a uma constante pré-definida, ou até que um número máximo de iterações tenha sido completado.

Este algoritmo pode ser implementado em YALI conforme o paradigma mestre-trabalhador, onde uma *thread* mestre deposita no espaço de tuplas as temperaturas correspondentes aos pontos (x, y), e dispara *threads* trabalhadoras que calculam a nova temperatura em cada ponto. Para que seja possível detectar a convergência do algoritmo, cada trabalhador deposita no TS, além da temperatura calculada, também a diferença entre o valor recém calculado e a temperatura na iteração anterior. A *thread* mestre, por sua vez, é responsável por coletar as diferenças em cada ponto e comparar a diferença máxima com a constante de erro pré-definida. Se esta diferença for maior ou igual à constante, a *thread* mestre deve ordenar às *threads* trabalhadoras que iniciem uma nova iteração. Em caso contrário, quando a convergência é detectada ou quando o número máximo de iterações é atingido, a *thread* mestre deve coletar a matriz resultante e encerrar a execução da aplicação.

Uma aplicação YALI que implemente este algoritmo pode ser estruturada de várias maneiras. Neste exemplo, adotou-se um modelo MPMD que utiliza dois tipos de processos:

- um processo mestre, cuja função principal executa as atribuições da *thread* mestre;
- vários processos trabalhadores, que executam uma ou mais *threads* trabalhadoras.

As figuras 6.1 e 6.2 ilustram, respectivamente, o código executado pelo processo mestre e pelos processos trabalhadores. Convém ressaltar que ambos os tipos de processos fazem uso de algumas operações globais disponíveis em YALI. Na figura 6.1, a *thread* mestre utiliza **y_reduce** para determinar a diferença máxima obtida entre os valores calculados na iteração atual e os valores da iteração anterior. Além disso, a primitiva **y_barrier** é utilizada para sincronizar a execução de todas as *threads* a cada iteração, de modo que uma nova iteração só inicia depois que a *thread* mestre analisa as diferenças de temperatura resultantes da iteração anterior.

Pode-se notar que esta implementação do algoritmo de Jacobi permite pouca exploração do paralelismo, já que as *threads* trabalhadoras são sincronizadas a cada iteração. Seria possível adotar uma solução mais eficiente, onde cada *thread* inicia uma nova iteração tão logo tenha terminado a iteração atual. Neste caso, a sincronização ocorreria automaticamente sempre que uma *thread* tivesse que esperar por algum valor produzido por outra *thread*. Com esta solução, no entanto, o espaço de tuplas poderia conter, num dado instante, elementos da matriz de temperaturas em várias iterações. Ao contrário, na implementação discutida anteriormente, o espaço de tuplas mantinha no máximo elementos de duas iterações, já que a cada nova iteração as *threads* trabalhadoras se encarregavam

```
#define MAXERR  0.000001  /* constante de erro */
#define MAXIT   10000     /* numero maximo de iteracoes */

yali_main() {

    int rows, cols; /* ordem da matriz de temperaturas */
    float *t;       /* matriz de temperaturas */
    int i, j, x, y, n_workers;
    float dif = MAXERR;
    int it = 1;

    /* aloca e inicializa matriz de temperaturas */
    ...

    /* armazena matriz no espaco de tuplas */
    for (x=0; x < rows; x++)
        for (y=0; y < cols; y++)
            y_out("T", 0, x, y, t[x][y]);

    /* numero de threads que participam da barreira */
    n_workers = (rows-2) * (cols-2);
    y_out("total_threads", 1 + n_workers); /* master + workers */

    /* inicializa escopo de cada thread */
    for (x=1; x < rows-1; x++)
        for (y=1; y < cols-1; y++)
            y_out("scope", x, y);

    /* dispara threads trabalhadoras */
    y_globeval(n_workers, "worker");

    while (it <= MAXIT && dif >= MAXERR) {
        y_out("it", it);
        y_reduce((rows-2)*(cols-2), "dif", ?x, ?y, MAX(?dif));
        y_in("it", it);
        y_barrier(n_workers+1, "end_of_iteration");
        it++;
    }

    /* sinaliza termino das iteracoes as threads trabalhadoras */
    y_out("it", 0);
    /* recolhe a matriz resultado do espaco de tuplas */
    for (x=0; x < rows; x++)
        for (y=0; y < cols; y++)
            y_in("T", it-1, x, y, ?t[x][y]);
}
```

Figura 6.1 - Código do processo mestre para implementação do algoritmo de Jacobi.

de remover as tuplas da iteração anterior. Para evitar que o espaço de tuplas seja saturado com uma grande quantidade de tuplas, a *thread* mestre poderia encarregar-se de retirar as tuplas anteriores a uma iteração onde a convergência já tenha sido testada.

```
yali_main() {

    y_global("worker", worker);
}

void worker() {

    int x, y, v1, v2, v3, v4, old_t, new_t, it, n_threads;

    y_rd("total_threads", ?n_threads);
    y_in("scope", ?x, ?y);
    y_rd("it", ?it);

    while (it != 0) {

        /* coleta temperatura em pontos vizinhos */
        y_rd("T", it-1, x-1, y, ?v1);
        y_rd("T", it-1, x+1, y, ?v2);
        y_rd("T", it-1, x, y-1, ?v3);
        y_rd("T", it-1, x, y+1, ?v4);

        /* calcula nova temperatura e diferenca */
        new_t = (v1 + v2 + v3 + v4)/4;
        y_rd("T", it-1, x, y, ?old_t);
        y_out("dif", x, y, abs(new_t - old_t));
        y_out("T", it, x, y, new_t);

        /* aguarda termino da iteracao */
        y_barrier(n_threads, "end_of_iteration");

        /* recolhe tupla da iteracao anterior */
        y_in("T", it-1, x, y, ?old_t);

        y_rd("it", ?it);
    }
}
```

Figura 6.2 - Código dos processos trabalhadores para implementação do algoritmo de Jacobi.

Os códigos do processo mestre e dos processos trabalhadores que implementam esta solução mais otimizada são apresentados, respectivamente, nas figuras 6.3 e 6.4. Nota-se, nestes novos exemplos, que a primitiva **y_barrier** não mais é utilizada. Além disso, as *threads* trabalhadoras utilizam uma primitiva não-bloqueante para verificar se uma nova iteração deve ou não ser iniciada. Caso esta primitiva retorne 0 como resultado, a *thread* simplesmente encerra sua execução. Se, ao contrário, fosse utilizada uma primitiva não-bloqueante equi-

valente, a *thread* permaneceria bloqueada caso não fosse necessário o cálculo de uma nova iteração.

6.2.2 Cálculo de π em Paralelo

Para ilustrar a estruturação de aplicação YALI segundo um modelo SPMD foi escolhido um programa simples para cálculo de π, que foi proposto inicialmente em [KAR88], e tem sido utilizado como exemplo em alguns trabalhos mais atuais, como [LAR94] e [MAT95]. O programa calcula o valor de π como sendo igual a:

$$\int_0^1 \frac{4.0}{1.0 + x^2}$$

Para computar a integral é utilizado um método numérico simples, baseado na regra dos trapézios. O código seqüencial que implementa este cálculo é mostrado na figura 6.5. Este programa consiste em um único laço que calcula o somatório do integrando no centro de vários sub-intervalos. Depois do laço, multiplica-se o somatório pelo tamanho dos sub-intervalos, a fim de convertê-lo para uma aproximação numérica da integral.

Embora o programa seja bastante simples, o cálculo de π com uma aproximação satisfatória exige que o número de sub-intervalos seja muito grande. Pode-se, no entanto, paralelizar o programa de modo a distribuir as iterações do laço entre vários processos que executam em paralelo. Na figura 6.6 encontra-se uma versão paralelizada deste programa, estruturada segundo o modelo SPMD. Nesta versão, todos os processos são idênticos, mas um deles age como mestre, enquanto os demais funcionam apenas como trabalhadores. Isto é possível através da primitiva **y_id**, que permite descobrir a identificação de um processo numa aplicação YALI. Esta identificação é usada para selecionar o código a ser realmente executado por cada processo, assim criando a distinção entre mestre e trabalhadores.

A função do processo mestre é obter o número de sub-intervalos desejado pelo usuário e, a seguir, depositar no TS as tuplas que indicam a parte da computação que deve ser executada por cada trabalhador. Existe uma tupla deste tipo para cada trabalhador, que deve usar as informações nela contidas para executar a função pi_comp e, depois de completá-la, depositar seu resultado no TS. Esta função recebe como parâmetro o número total de sub-intervalos, o número de processos que participam da aplicação, e um índice que é usado para determinar quais os sub-intervalos que cabem a cada processo. Como pi_comp executa apenas parte das iterações para o cálculo de π, os resultados parciais produzidos por cada processo devem ser combinados para produzir o resultado final. Isto é feito pelo processo mestre, que também executa pi_comp mas, a seguir, utiliza **y_reduce** para recolher do TS todas as somas parciais produzidas pelos trabalhadores.

```
#define MAXERR  0.000001  /* constante de erro */
#define MAXIT   10000     /* numero maximo de iteracoes */

yali_main() {

    int rows, cols; /* ordem da matriz de temperaturas */
    float *t;       /* matriz de temperaturas */
    float dif = MAXERR;
    int it = 0;
    int i, j, x, y, n_workers;

    /* aloca e inicializa matriz de temperaturas */
    ...

    /* armazena matriz no espaco de tuplas */
    for (x=0; x < rows; x++)
        for (y=0; y < cols; y++)
            y_out("T", it, x, y, t[x][y]);

    /* inicializa escopo de cada thread */
    for (x=1; x < rows-1; x++)
        for (y=1; y < cols-1; y++)
                y_out("scope", x, y);

    /* coloca tupla de inicializacao */
    y_out("continue");

    /* dispara threads trabalhadoras */
    n_workers = (rows - 2) * (cols - 2);
    y_globeval(n_workers, "worker");

    while (it <= MAXIT && dif >= MAXERR) {
        y_reduce(n_workers, "dif", it, ?x, ?y, MAX(?dif));
        y_reduce(rows*cols, "T", it, ?x, ?y, ?t[x][y]);
        it++;
    }

    /* impede que as threads trabalhadoras iniciem nova iteracao */
    y_in("continue");

    /* recolhe a matriz resultado do espaco de tuplas */
    for (x=0; x < rows; x++)
        for (y=0; y < cols; y++)
            y_in("T", it, x, y, ?t[x][y]);
}
```

Figura 6.3 - Código otimizado para implementação do algoritmo de Jacobi (processo mestre).

```
yali_main() {

    y_global("worker", worker);
}

void worker() {

    int x, y, v1, v2, v3, v4, old_t, new_t;
    int it = 0;

    y_in("scope", ?x, ?y);

    while (y_rdp("continue") != 0) {

        /* coleta temperatura em pontos vizinhos */
        y_rd("T", it, x-1, y, ?v1);
        y_rd("T", it, x+1, y, ?v2);
        y_rd("T", it, x, y-1, ?v3);
        y_rd("T", it, x, y+1, ?v4);

        /* calcula nova temperatura e diferenca */
        new_t = (v1 + v2 + v3 + v4)/4;
        y_rd("T", it, x, y, ?old_t);
        y_out("dif", it, x, y, abs(new_t - old_t));
        y_out("T", ++it, x, y, new_t);

    }
}
```

Figura 6.4 - Código otimizado para implementação do algoritmo de Jacobi (processos trabalhadores).

6.2.3 Determinação de Números Primos em um Intervalo

O algoritmo conhecido como "peneira de Eratóstenes" ou "crivo de Eratóstenes"[2][CAR89a] constitui uma solução para o problema de encontrar números primos dentro de um intervalo que vai de 1 até um dado limite. Seu funcionamento pode ser entendido imaginando-se um conjunto de números inteiros que passam por uma série de peneiras: uma delas remove múltiplos de 2, outra remove múltiplos de 3, a seguinte múltiplos de 5, e assim por diante. Um número inteiro que tenha passado pela última peneira de uma série é um novo número primo, e pode constituir uma nova peneira nesta mesma série.

Um programa paralelo que implementa este algoritmo pode ser constituído por uma seqüência de processos ou *threads* que representam peneiras, isto é, que recebem como entrada um conjunto de inteiros, e fornecem na saída somente aqueles elementos que não são múltiplos do número primo no qual a peneira é especializada. A saída de uma peneira é fornecida como entrada para o próximo

[2] *sieve of Erathostenes*

```
main() {

    int j, num_steps;
    double  step, x, sum, pi;

    printf("How many steps should we take?");
    scanf("%d", &num_steps);

    step = 1.0/num_steps;
    sum  = 0.0;

    for (j = 1; j<num_steps; j++) {
        x = (j-0.5) * step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = sum * step;

    printf("For %d steps, pi is %f\n", num_steps, pi);

}
```

Figura 6.5 - Versão seqüencial do programa para cálculo de π.

processo na seqüência, e novos processos podem ser criados a cada novo número primo encontrado.

Em [CAR89a] é encontrado um exemplo de implementação deste algoritmo em Linda. O exemplo foi adaptado para funcionar com os recursos oferecidos por YALI, produzindo o programa das figuras 6.7 e 6.8. O programa é estruturado segundo o modelo SPMD e, em tempo de execução, é composto por quatro tipos de *threads*:

- a primeira age como produtora, inserindo no TS um conjunto de tuplas contendo números inteiros ímpares. A seguir, esta *thread* aguarda até que seja computada a quantidade de números primos existentes no intervalo estabelecido. Existe apenas uma *thread* deste tipo na aplicação: é aquela que executa yali_main no processo para o qual a função **y_id** retorna 0 como identificador;

- a segunda *thread*, que executa a função sink, representa o último estágio de uma seqüência de peneiras, sendo sempre responsável pela remoção de múltiplos do maior número primo descoberto pelo algoritmo. Sempre que esta *thread* detecta um novo número primo cujo quadrado não excede o limite do intervalo estabelecido, ela dispara uma nova *thread* especializada em remover múltiplos do número primo anterior. Isto faz com que sink passe a receber somente números que já foram previamente "peneirados";

- o terceiro tipo de *thread* executa a função sieve, sendo disparada pela *thread* sink com o objetivo de remover múltiplos de um determinado número primo. Podem existir várias *threads* deste tipo, representando vários estágios

```
double pi_comp(int id, int num_steps, int num_nodes);

yali_main() {

    int     id, num_steps, num_nodes;
    double  pi, partial_sum;

    num_nodes = y_nproc();

    if (y_id() == 0) {  /* processo mestre */

        printf("\n How many steps should we take? ");
        scanf("%d", &num_steps);

        for (id=0; id<num_nodes-1; id++)
            y_out("task", id, num_steps);

        pi = pi_comp(num_nodes-1, num_steps, num_nodes);

        y_reduce(num_nodes-1, "partial_sum", SUM(?partial_sum));
        pi += partial_sum;

        printf("For num_steps = %d steps, PI = %f\n", num_steps, pi);

    } else { /* processo trabalhador */

        y_in("task", ?id, ?num_steps);
        y_out("partial_result", pi_comp(id, num_steps, num_nodes));

    }

}

double pi_comp(int id, int num_steps, int num_nodes) {

    int     j;
    double  step, x, sum;

    step = 1.0/num_steps;

    sum = 0.0;
    for (j=id+1; j < num_steps; j+=num_nodes) {
        x = (j-0.5) * step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    sum = sum * step;

    return sum;

}
```

Figura 6.6 - Programa YALI para cálculo de π em paralelo.

```
#define LIMIT 1000000

void sieve();
void sink();

yali_main() {

    int i, primes_found, out_index = 0;

    y_global("sieve", sieve);

    if (y_id() == 0) {  /* produtor */

        y_eval(sink);

        for (i=5; i<LIMIT; i += 2)
            y_out("number", 3, out_index++, i);

        y_out("number", 3, out_index, 0);

        y_in("prime_count", ?primes_found);
        printf("Number of primes between 1 and %d: %d\n", LIMIT,
             primes_found);
    }
}
```

Figura 6.7 - Programa YALI para o cálculo da peneira de Eratóstenes (principal).

de peneiras. Todas elas recebem como entrada um conjunto de tuplas contendo números produzidos num estágio, e repassam para o próximo estágio um subconjunto destas tuplas (somente aquelas que não contenham múltiplos do número primo em questão);

- o último tipo é constituído por todas as *threads* que executam yali_main em processos cujo identificador é diferente de 0. Estas *threads* têm pouca duração, e sua única ação é declarar como global a função sieve implementada pelos processos aos quais pertencem. Com isso, toda vez que sink utiliza **y_globeval**, uma *thread* sieve é disparada em um dos vários processos que implementam esta função.

Toda *thread* sieve termina após processar as tuplas destinadas ao seu estágio. Com isso, quando a *thread* sink detecta que não existem mais números a processar, todas as outras *threads* sieve já terminaram sua execução, de modo que não é necessário nenhum mecanismo adicional de sincronização destas *threads*.

Embora este programa forneça como resultado somente a quantidade de números primos encontrada, ele seria facilmente modificável a fim de fornecer uma lista destes números. Também é conveniente observar que este algoritmo permite uma baixa exploração do paralelismo, à medida que introduz uma noção de seqüencialidade com a criação de estágios de "peneiras". No entanto, este al-

```
void sink() {

    int num;
    int prime_count = 2;
    int in_index = 0;
    int prime = 3;

    while (1) {

        y_in("number", prime, in_index++, ?num);

        if (!num) break;

        if (num % prime) {
            ++prime_count;
            if (num*num < LIMIT) {
                y_out("new_sieve", prime, num, in_index);
                y_globeval("sieve");
                prime = num;
                in_index = 0;
            }
        }
    }
    y_out("prime_count", prime_count);
}

void sieve() {

    int prime, next, in_index;
    int num, out_index = 0;

    y_in("sieve", ?prime, ?next, ?index);

    while(1) {
        y_in("number", prime, index++, ?num);
        if (!num) {
            y_out("number", next, out_index, num);
            return;
        }
        if (num % prime)
            y_out("number", next, out_index++, num);
    }
}
```

Figura 6.8 - Programa YALI para o cálculo da peneira de Eratóstenes (funções auxiliares)

goritmo é interessante do ponto de vista da abstração que ele representa, a qual pode ser facilmente implementada com os recursos oferecidos por YALI.

6.3 Multiplicação de Matrizes

Este exemplo consiste no cálculo paralelo do produto de duas matrizes quadradas A e B, produzindo uma matriz resultante C de mesma ordem. A implementação é estruturada segundo o modelo MPMD, e emprega dois tipos de processos:

- um processo mestre ou coordenador, que coleta as matrizes de entrada, armazena-as de forma distribuída no espaço de tuplas, e espera que a matriz resultante seja calculada;
- vários processos trabalhadores, que são responsáveis pelo cálculo de uma ou mais linhas da matriz resultante;

As matrizes de entrada são armazenadas de forma distinta no TS: enquanto a matriz A é armazenada sob a forma de **linhas**, a matriz B é dissociada em tuplas que representam suas **colunas**. Para iniciar o cálculo da matriz resultante, o processo mestre armazena no TS a dimensão dessa matriz e, a seguir, deposita uma tupla contendo o número da primeira linha de C a ser calculada. Cada processo trabalhador que retira esta tupla é responsável por atualizá-la, devolvendo-a ao TS com o número da próxima linha a ser calculada, ou com um número que indica o fim dos cálculos (-1). Após o cálculo de uma linha, cada trabalhador volta a buscar o número de outra linha a calcular, e só termina ao retirar um número negativo, indicando que todas as linhas foram calculadas.

As figuras 6.9 e 6.10 ilustram, respectivamente, o código YALI executado pelo mestre e pelos trabalhadores para implementação deste algoritmo de multiplicação de matrizes. Para fins de simplificação, todos os elementos da matriz A são idênticos, com valor igual a 3. O mesmo ocorre com os elementos de B, que são inicializados com o valor 5. Obviamente, neste caso, os elementos da matriz resultante poderiam ser facilmente calculados pelo processo mestre. Embora este exemplo seja apenas ilustrativo, poucas modificações seriam necessárias para que fossem utilizadas quaisquer matrizes de entrada.

Pela figura 6.9, nota-se que o processo mestre coleta as linhas da matriz resultado uma por vez, através de várias chamadas a **y_in**. Alternativamente, poder-se-ia utilizar **y_gather** para reunir todas as tuplas com uma única operação. Isto não foi feito, no entanto, porque **y_gather** não permite que as tuplas sejam recuperadas numa ordem pré-estabelecida.

```
yali_main() {

    long   A_row[256], B_col[256], C_row[256];
    long   dim, index, row_index, col_index;

    /* obtem dimensao das matrizes */
    printf("Enter dimension:\n");
    scanf("%d", &dim);

    /* inicializa as duas matrizes */
    for (index = 0; index < dim; index++) {  {
        A_row[index] = 3;
        B_col[index] = 5;
    }

    /* armazena dimensao no espaco de tuplas */
    y_out("dim", dim);

    /* coloca matrizes no espaco de tuplas */
    for (index = 0; index < dim; index++) {
        y_out("row", index, A_row:dim); /* todas as linhas de A sao identicas */
        y_out("col", index, B_col:dim); /* todas as colunas de B sao identicas */
    }

    /* coloca indice da primeira linha a calcular */
    y_out("row_index", 0);

    /* coleta as linhas da matriz resultado */
    for (index = 0; index < dim; index++)
        y_in("prod", ?row_index, ?C_row:dim);
}
```

Figura 6.9 - Processo mestre para multiplicação de matrizes em paralelo

6.4 Desempenho

Para permitir uma avaliação quantitativa de YALI, algumas aplicações simples foram executadas com o auxílio do protótipo do sistema. Este protótipo executa em estações Sun com sistema operacional SunOS ou Solaris, e também na plataforma i486/Mach. Os resultados de desempenho obtidos são apresentados a seguir.

6.4.1 Ping-Pong

Inicialmente foi implementado um *benchmark* conhecido como "Ping-Pong", cujo objetivo é medir o tempo médio decorrido entre o envio de uma mensagem através do espaço de tuplas e o recebimento da resposta correspondente. O *benchmark* utiliza dois processos: um deles deposita uma tupla (`"ping"`) e retira uma tupla (`"pong"`), enquanto o outro retira a tupla (`"ping"`) e devolve a tu-

```
yali_main() {

    long    dim, col_index, row_index, next_index;
    long    A_row[256], B_col[256], C_row[256];

    /* obtem dimensao das matrizes */
    y_rd("dim", ?dim);

    while(1) {

        /* obtem indice da linha a calcular */
        y_in("row_index", ?row_index);

        /* se nao existem mais linhas a calcular, encerra execucao */
        if (row_index < 0) {
            y_out("row_index", -1); /* coloca tupla de volta no TS */
            return;
        }

        /* nova linha a calcular */
        next_index = row_index + 1;

        if (next_index < dim)
            /* existem linhas a calcular, entao deposita indice no TS */
            y_out("row_index", next_index);
        else
            /* insere tupla de terminacao */
            y_out("row_index", -1);

        /* obtem linha da matriz A */
        y_rd("row", row_index, ?A_row:dim);

        /* le cada coluna da matriz B, calculando a nova linha da matriz C */
        for (col_index = 0 ; col_index < dim; col_index++) {

            y_rd("col", col_index, ?B_col:dim);

            C_row[col_index] = 0;
            for (index = 0; index < dim; index++, rp++, cp++)
                C_row[col_index] += A_row[index] * B_col[index];

        }

        /* deposita produto no espaco de tuplas */
        y_out("prod", row_index, C_row:dim);
    }
}
```

Figura 6.10 - Processo trabalhador para multiplicação de matrizes em paralelo

pla ("pong"). Executando-se um total de 100 iterações deste tipo, com tuplas de diferentes tamanhos, obteve-se os os resultados apresentados na tabela 6.3. Para a realização destas medições foram utilizadas duas estações Sun, modelo SPARC-Classic, ligadas a uma mesma rede local. Cada processo foi mapeado em uma estação, e as medições foram feitas em um período de baixa utilização da rede.

Tabela 6.3 - Tempo médio de uma iteração Ping-Pong em YALI.

Tamanho da Tupla (em bytes)	**Tempo Médio** (em milissegundos)
100	9.6
400	10.8
1000	13.3
4000	23.2
10000	57.6

Este mesmo *benchmark* foi implementado no sistema POSYBL, que foi obtido junto ao endereço mencionado na seção 3.3.2, e foi instalado na mesma plataforma descrita acima. Assim como YALI, este sistema utiliza um espaço de tuplas distribuído, e é implementado a partir de recursos oferecidos diretamente pelo sistema operacional. A execução deste *benchmark* em POSYBL forneceu os resultados contidos na tabela 6.4. Comparando-se esta tabela com a anterior, nota-se que em YALI obteve-se melhores resultados para este *benchmark*. Isto pode ser explicado pela política de distribuição de tuplas utilizada em YALI, que requer poucas mensagens para execução das primitivas, e pelo fato de que, ao contrário de YALI, POSYBL utiliza processos especiais para gerenciamento de tuplas, que precisam ser contactados a cada operação executada por um processo usuário.

Tabela 6.4 - Tempo médio de uma iteração Ping-Pong em POSYBL.

Tamanho da Tupla (em bytes)	**Tempo Médio** (em milissegundos)
100	20.4
400	23.6
1000	30.4
4000	47.7
10000	64.3

6.4.2 Geração de Fractais

Este segundo exemplo consiste na implementação de um algoritmo para geração de fractais do conjunto de Mandelbrot, que basicamente são a representação gráfica a aplicação de uma série em uma área retangular. A área escolhida pode

ser dividida em blocos, que podem ser calculados independentemente dos demais. Este algoritmo, portanto, presta-se à implementação segundo o paradigma mestre-trabalhador, onde vários processos trabalhadores calculam em paralelo os blocos da figura a ser apresentada. As informações que descrevem um bloco a calcular são depositadas no espaço de tuplas por um processo mestre, que depois coleta as tuplas com os resultados produzidos pelos processos trabalhadores. Basicamente, uma tupla resultado contém informações sobre as cores de cada pixel que deve ser apresentado na tela pelo processo mestre.

Executando-se esta aplicação com diferentes números de processos trabalhadores e utilizando-se áreas de diferentes tamanhos, obteve-se um *speedup* que é apresentado na figura 6.11.

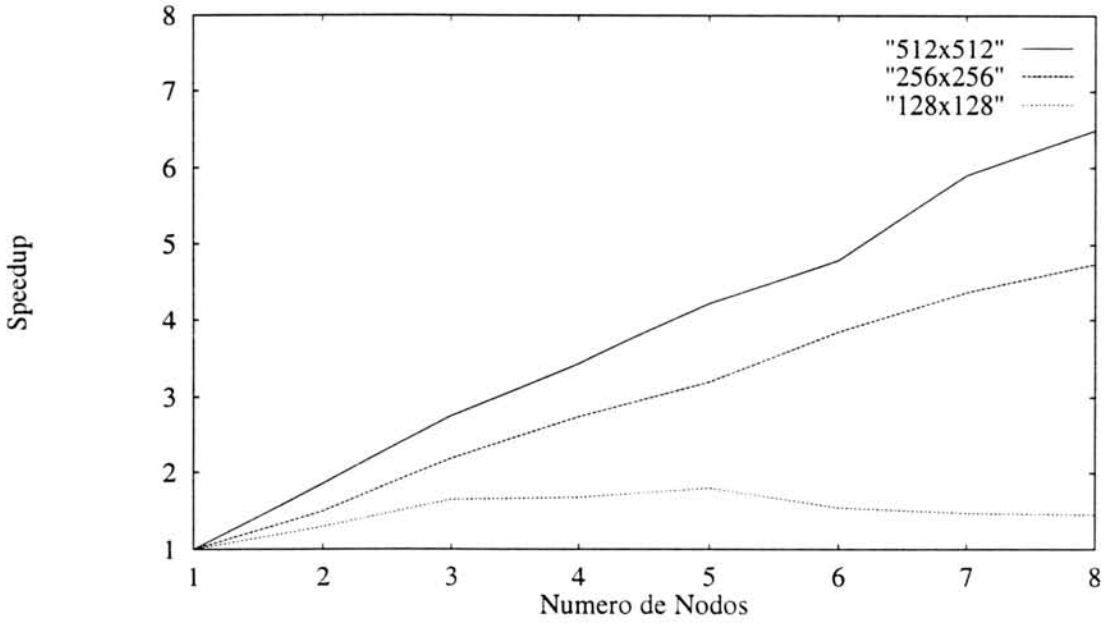

Figura 6.11 - *Speedup* para a implementação do algoritmo de Mandelbrot.

7 Conclusão

Este último capítulo está dividido em duas partes: na primeira é feita uma avaliação geral dos aspectos relevantes do trabalho, enquanto que na segunda são formuladas várias sugestões para sua futura continuação.

7.1 Avaliação Geral do Trabalho

Servindo para a programação paralela em redes heterogêneas, YALI encaixa-se em uma classe de ferramentas de *software* bastante promissora atualmente, capaz de permitir a exploração das vantagens do paralelismo em plataformas amplamente acessíveis, e de custo relativamente baixo. Além de estar em conformidade com necessidades atuais, YALI destaca-se por oferecer uma interface simples, de fácil aprendizado e utilização, que ameniza as dificuldades envolvidas no desenvolvimento de programas paralelos e, em conseqüência disso, torna esta técnica mais acessível a programadores em geral.

Na efetivação deste compromisso com a simplicidade, a escolha do modelo Linda foi de fundamental importância. Por ser baseado neste modelo, YALI oferece uma abstração de alto nível para a interação entre processos e disponibiliza um pequeno mas expressivo conjunto de primitivas, que são incorporadas a programas escritos em C. A escolha da linguagem seqüencial hospedeira também foi importante, pois o uso de uma linguagem amplamente difundida aumenta a facilidade de aprendizado do sistema.

Uma importante contribuição deste trabalho foi a incorporação de operações globais ao modelo Linda. Esta extensão foi projetada de maneira a não comprometer o grau de abstração inerente ao modelo, resultando em algumas primitivas de alto nível que permitem a comunicação e a sincronização entre múltiplos processos. Com o suporte a estas primitivas, YALI oferece maior poder de expressão ao usuário programador e, principalmente, garante a execução eficiente de operações que são freqüentemente necessárias em programas paralelos.

Além do suporte a operações globais, outro aspecto importante deste trabalho foi a semântica alternativa adotada para a expressão do paralelismo, concretizada através de primitivas para criação dinâmica de *threads*. Com esta semântica baseada em *threads*, YALI permite expressar o paralelismo de forma leve, o que é apropriado ao processamento em paralelo tanto de rotinas simples como de procedimentos mais complexos. Além disso, a possibilidade de se criar *threads* à distância em YALI oferece maior flexibilidade ao usuário, além de permitir um certo grau de distribuição de carga na execução de uma aplicação.

A opção pela incorporação de um pré-processador ao sistema também foi fundamental para garantir uma interface simples às primitivas YALI, fortalecendo ainda mais o ambiente de programação projetado, e servindo para destacá-lo em relação a alguns sistemas semelhantes. A flexibilidade que YALI permite na

estruturação das aplicações paralelas também é uma característica que merece destaque, já que nem sempre está presente em outros sistemas baseados em Linda.

O ambiente de execução de YALI foi organizado de forma bastante particular, através de múltiplas *threads* residentes em cada processo de uma aplicação. Com esta organização, cada processo executa as rotinas definidas pelo usuário e, concorrentemente, pode atender requisições de manipulação de tuplas, barreiras e funções globais. Pôde-se constatar que esta organização, além de ser bastante modular, exige menor número de mensagens para implementação das primitivas, se comparada com a alternativa de se usar processos servidores especiais para gerenciar tuplas ou outros elementos.

A política baseada em *hashing* utilizada para coordenar a distribuição dos elementos do sistema pode ser considerada demasiadamente simples, já que não prevê qualquer otimização no mapeamento destes elementos sobre os processos da aplicação. Mesmo assim, o desempenho de YALI foi razoavelmente satisfatório nos testes realizados, até mesmo superando o desempenho de outro sistema similar que foi instalado na mesma plataforma. Para isso, certamente contribuiu o fato de que o *hashing* permite uma rápida localização de um determinado elemento no sistema, exigindo pequeno tráfego de mensagens na rede. Além disso, como a comunicação entre processos e a transferência de dados entre máquinas heterogêneas são implementadas diretamente sobre os recursos do sistema operacional hospedeiro, o *overhead* que estas operações causam no ambiente de execução é mantido em níveis relativamente baixos.

Apesar de fornecer bons resultados preliminares, o sistema pode ser melhorado em alguns pontos. Como não são implementadas otimizações em tempo de compilação, o desempenho do sistema pode variar muito de aplicação para aplicação e, de modo geral, dificilmente seria comparável ao desempenho de um sistema como Network-Linda, que prevê este tipo de otimização. Além disso, como o espaço de tuplas de YALI é embutido nos processos que compõem uma aplicação, a interação entre processos completamente disjuntos no tempo fica impossibilitada, comprometendo uma importante característica do modelo Linda. Estas limitações, de um modo geral, não impedem que as vantagens do paralelismo sejam exploradas em YALI, mas poderiam ser tratadas de forma a aperfeiçoar o sistema projetado. Na próxima seção discute-se algumas possíveis soluções para estas limitações, além de outras melhorias que podem ser futuramente incorporadas ao sistema.

7.2 Trabalhos Futuros

Várias técnicas poderiam ser utilizadas para aumentar o desempenho de YALI. Uma delas seria aperfeiçoar o pré-processador do sistema, de modo a implementar algumas otimizações em tempo de compilação, como é feito em Network-Linda. A incorporação destas otimizações poderia ser feita gradualmente, iniciando pela determinação de chaves de *hash* mais precisas (através da incorporação,

quando possível, de outros campos reais à chave original), e possivelmente terminando com a capacidade de determinar a política de distribuição mais adequada a cada padrão de acesso a tuplas identificado numa aplicação.

Outra melhoria que poderia ser incorporada ao sistema seria com relação ao mapeamento eficiente de processos sobre a rede. Na versão preliminar do sistema, a escolha dos nodos que devem ser utilizados para execução de uma aplicação e o mapeamento de processos sobre estes nodos são tarefas que o usuário deve realizar manualmente. Se estas tarefas fossem realizadas pelo sistema, a distribuição seria mais transparente ao usuário, e algumas otimizações poderiam ser implementadas. Para a escolha do conjunto de nodos mais apropriado à execução de uma aplicação, uma otimização simples seria levar em conta alguma informação sobre a carga das máquinas disponíveis, de modo a, por exemplo, evitar o uso de máquinas já sobrecarregadas. Isto poderia ser implementado através de servidores adicionais, responsáveis por coletar dados sobre a utilização de cada máquina da rede, que seriam consultados por `yalistart` no momento da inicialização de uma aplicação. Para a definição dos processos a serem executados em cada nodo escolhido, um mecanismo capaz de mapear processos mais pesados nas máquinas mais poderosas ou menos carregadas seria o ideal, mas a determinação da carga de um processo antes de sua execução seria uma tarefa bastante complexa.

A discussão acima se refere somente ao mapeamento estático de processos, mas poderia ser estendida de modo a contemplar o escalonamento eficiente de *threads* criadas dinamicamente pelo usuário. O sistema já utiliza um escalonador simples para escolha do processo onde uma nova *thread* deve ser executada e, caso fossem implementados servidores adicionais para monitoração da carga da rede, este escalonador seria facilmente modificado de modo a mapear uma nova *thread* no processo menos carregado (entre os processos disponíveis, o processo menos carregado seria aquele executando em um nodo menos carregado).

Ainda visando um melhor desempenho do sistema, outra solução seria utilizar um mecanismo de *hashing* adaptativo, que permitiria principalmente uma melhor distribuição de tuplas no sistema. De fato, o mecanismo de *hashing* utilizado atualmente é bastante simples, pois mapeia tuplas com a mesma descrição em um único processo, o que pode causar o surgimento de gargalos no sistema. Com um mecanismo adaptativo, a função de *hash* levaria em conta informações sobre a utilização do espaço de tuplas coletadas em tempo de execução ou de compilação. Isso permitiria, por exemplo, que a função de *hash* mapeasse tuplas diretamente sobre o processo que iria acessá-la, reduzindo assim o número de mensagens trocadas através da rede. Outra possibilidade seria mapear tuplas de um mesmo tipo em mais de um processo, a fim de reduzir a chance de sobrecarga de um único processo. A opção por um mecanismo de *hashing* adaptativo, no entanto, deve ser feita com cuidado, já que sua implementação pode implicar num *overhead* adicional no ambiente de execução.

Paralelamente à implementação de técnicas para aumentar o desempenho do sistema, algumas melhorias poderiam ser incorporadas ao ambiente de programação de YALI. Para permitir a comunicação entre processos disjuntos no tempo, ou para permitir que aplicações diferentes também possam interagir através do

espaço de tuplas, seria possível implementar um protocolo especial executado por processos `yalistart`, cujo propósito seria mesclar os espaços de tuplas de diferentes aplicações. A execução de um protocolo deste tipo seria obviamente uma operação bastante pesada, justificável apenas no caso de aplicações inerentemente distribuídas onde não houvesse grandes preocupações com o desempenho final.

Outra possível melhoria no ambiente de programação seria dotar a primitiva **y_gather** de algum meio de expressar a ordem de recuperação das tuplas, com base nos valores de determinados campos. A utilidade disso seria, por exemplo, permitir a recuperação ordenada dos elementos de uma matriz distribuída, como aquela utilizada no exemplo da seção 6.3 do capítulo anterior, onde ilustra-se um algoritmo paralelo para multiplicação de matrizes. Isto poderia aumentar ainda mais o poder de expressão de YALI, mas a implementação desta nova funcionalidade deveria ser analisada com cuidado, pois o *overhead* causado pela ordenação das tuplas poderia prejudicar a eficiência no processamento de **y_gather**.

O sistema também poderia disponibilizar algumas ferramentas adicionais ao usuário. Uma interface gráfica para auxiliar na configuração de aplicações seria bastante útil, principalmente para usuários pouco experientes. Além disso, poderia ser desenvolvida uma ferramenta capaz de permitir a monitoração do uso de tuplas, barreiras e funções globais em todos os processos de uma aplicação, com o propósito de facilitar a depuração de programas YALI.

Finalmente, também pode-se sugerir como trabalho futuro o porte de YALI para outras arquiteturas e sistemas que suportam *threads*, como, por exemplo, Cray Y-MP com sistema UNICOS e IBM RS6000 com sistema AIX.

Bibliografia

[ACC86] ACCETTA, M. et al. Mach: A New Kernel Foundation for UNIX Development. In: USENIX SUMMER CONFERENCE, 1986, Atlanta. **Proceedings...** Berkeley: USENIX, 1986, p.93–112.

[AHU86] AHUJA, S. et al. Linda and Friends. **IEEE Computer,** New York, v. 19, n.8, p.26–34, Aug. 1986.

[AHU88] AHUJA, S. et al. Matching Language and Hardware for Parallel Computation in the Linda Machine. **IEEE Transactions on Computers,** New York, v.37, n.8, p.921–929, Aug. 1988.

[AND93] ANDREWS, G.R.; OLSSON, R.A. **The SR Programming Language:** Concurrency in Practice. Redwood City: Benjamin/Cummings, 1993. 344p.

[ARA89] ARANGO, M.; BERNDT, D. **TSnet:** A Linda Implementation for Networks of Unix-based Computers. New Haven: Department of Computer Science, Yale University, Aug. 1989. (Technical Report, 739).

[BAK94] BAKKEN, D.E. **Supporting Fault-Tolerant Parallel Programming in Linda.** Tucson: Department of Computer Science, University of Arizona, Aug. 1994. Ph.D. Dissertation.

[BAL89] BAL, H.E. et al. Programming Languages for Distributed Computing Systems. **ACM Computing Surveys,** New York, v.21, n.3, p.261–322, Sept. 1989.

[BAL90] BAL, H.E. **Programming Distributed Systems.** Hertfordshire: Prentice-Hall, 1990. 268p.

[BAL92] BAL, H.E. et al. Orca: A Language for Parallel Programming of Distributed Systems. **IEEE Transactions on Software Engineering,** New York, v.18, n.3, p.190-205, Mar. 1992.

[BAL94] BAL, H.E. A Comparative Study of Five Parallel Programming Languages. In: BRAZIER, F.; JOHANSEN, D. (Eds.). **Distributed Open Systems**. New York: IEEE Computer Society Press, 1994. p.134–151. Trabalho apresentado na EurOpen Spring 1991 Conference on Open Distributed Systems, Tromso, May 1991.

[BAR93] BARCELLOS, A.M.P. **O Sistema Operacional de Rede Heterogêneo HetNOS.** Porto Alegre: CPGCC da UFRGS, 1993. 300p. Dissertação de Mestrado.

[BER90] BERCOVITZ, P.; CARRIERO, N. **TupleScope:** A Graphical Monitor and Debugger for Linda-Based Parallel Programs. New Haven: Department of Computer Science, Yale University, Apr. 1990. (Technical Report, 782).

[BIR84] BIRREL, A.D; NELSON, B.J. Implementing Remote Procedure Calls. **ACM Transactions on Computer Systems,** New York, v.2, n.1, p.39–59, Feb. 1984.

[BJO89] BJORNSON, R.; CARRIERO, N.; GELERNTER, D. **The Implementation and Performance of Hypercube Linda.** New Haven: Department of Computer Science, Yale University, Mar. 1989. (Research Report, nº YALEU/DCS/RR-690).

[BJO92] BJORNSON, R. **Linda on Distributed Memory Multiprocessors.** New Haven: Department of Computer Science, Yale University, Nov. 1992. Ph.D. Thesis.

[BOY87] BOYLE, J. et al. **Portable Programs for Parallel Processors.** New York: Hold, Rinehart, and Winston, Inc., 1987.

[BRA90] BRANER, M. et al. Trolius: A Software Solution for Transputers and Other Multicomputers. In: CONFERENCE OF THE NORTH AMERICAN TRANSPUTER USERS GROUP (NATUG), 1., 1989, [S.l]. **Proceedings...**, Amsterdam: IOS Press, 1990. p.1–4.

[BUT91] BUTCHER, P. Lucinda. In: WILSON, G. V., **Linda-Like Systems and Their Implementation.** Edinburgh: Parallel Computing Centre, 1991. p.27–38. (EPCC Technical Report, 91-13). (disponível em `ftp://ftp.epcc.ed.ac.uk/pub/tr/91/tr9113.ps.Z`).

[BUT93] BUTLER, R.M.; LUSK, E. **p4-Linda: A Portable Implementation of Linda.** Argonne: Mathematics and Computer Science Divison, Argonne National Laboratory, Jul. 1993. (Technical Report, MCS-P374-0793). (disponível em `ftp://info.mcs.anl.gov/pub/tech_reports/reports/P374.ps.Z`).

[CAE94a] CARREIRA, J. et al. ParLin: From a Centralized Tuple Space to Adaptive Hashing. In: WORLD TRANSPUTER CONGRESS, Cernobbio, Itália, Sept. 1994. **Proceedings...** Amsterdam: IOS Press, 1994. p.91–104. (disponível em `http://pandora.uc.pt/Papers/wtc94.ps.Z`).

[CAE94b] CARREIRA, J. et al. On the Design of Eilean: A Linda-like library for MPI. In: SCALABLE PARALLEL LIBRARIES CONFERENCE, 2., Mississipi, USA, 1994. **Proceedings...** New York: IEEE Computer Society Press, 1994. (disponível em `http://pandora.uc.pt/Papers/splc94.ps.Z`).

[CAG93] CAGAN, L.; SHERMAN, A.H. Linda unites network systems. **IEEE Spectrum,** New York, v.30, n.12, p.31–35, Dec. 1993.

[CAL91a] CALLSEN, C.J. et al. **The AUC C++ Linda System.** Aalborg: Department of Mathematics and Computer Science, Aalborg University Center, 1991. p.39–73. (EPCC Technical Report, 91-13).

[CAR86] CARRIERO, N.; GELERNTER, D. The S/Net's Linda Kernel. **ACM Transactions on Computer Systems,** New York, v.4, n.2, p.110–129, May 1986.

[CAR87] CARRIERO, N. **Implementing Tuple Space Machines.** New Haven: Department of Computer Science, Yale University, Dec. 1987. Ph.D. Thesis.

[CAR89] CARRIERO, N.; GELERNTER, D. Linda in Context. **Communications of the ACM,** New York, v.32, n.4, p.444–458, Apr. 1989.

[CAR89a] CARRIERO, N.; GELERNTER, D. How to Write Parallel Programs: A Guide to the Perplexed. **ACM Computing Surveys,** New York, v.21, n.3, p.323–357, Sept. 1989.

[CAR90] CARRIERO, N.; GELERNTER, D. Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler. In: GELERNTER, D.; NICOLAU, A.; PADUA, D. (Eds.). **Languages and Compilers for Parallel Computing.** Cambridge: MIT Press, 1990. p.114–125.

[CAR94] CARRIERO, N. et al. The Linda Alternative to Message-Passing Systems. **Parallel Computing,** Eindhoven, v.20, n.4, p.633–655, Apr. 1994.

[CAR95] CARRIERO, N. et al. Adaptive Parallelism and Piranha. **IEEE Computer,** New York, v.28, n.1, p.40–49, Jan. 1995.

[CIA93] CIANCARINI, P; GUERRINI, N. Linda Meets Minix. **ACM Operating Systems Review,** New York, v.24, n.4, p.76–92, Oct. 1993.

[COO90] COOPER, E.C.; DRAVES, R.P. **C Threads.** Pittsburgh: Department of Computer Science, Carnegie Mellon University, Sept. 1990. (Technical Report, nº CMU-CS-88-154). (disponível em `ftp://reports.adm.cs.cmu.edu/usr/anon/1988/CMU-CS-88-154.ps`).

[COR91] CORBIN, J. **The Art of Distributed Applications.** Berlin: Springer-Verlag, 1991. 321p.

[DEC93] DIGITAL EQUIPMENT CORPORATION. **Guide to DECthreads.** Bedford: Digital Press, Mar. 1993. 372p.

[DOD83] U.S. DEPARTMENT OF DEFENSE. **Reference Manual for the Ada Programming Language.** Washington D.C.: U.S. Department of Defense, 1983.

[DOU93] DOUGLAS, C.G. et al. **Parallel Programming Systems for Workstation Clusters.** New Haven: Department of Computer Science, Yale University, Aug. 1993. (Technical Report, TR-975). (disponível em `ftp://casper.cs.yale.edu/pub/tr975.ps`).

[DOU95] DOUGLAS, A. et al. **ISETL-LINDA: Parallel Programming with Bags.** York: Department of Computer Science, University of York, 1995. (Technical Report, YCS-95-257). (disponível em `ftp://ftp.cs.york.ac.uk/reports/YCS-95-257.ps.Z`).

[FAA91] FAASEN, C. Intermediate Uniformly Distributed Tuple Space on Transputer Meshes. In: BANATRE, J. B., LEMETAYER, D. (Eds.). **Research Directions in High–Level Parallel Programming Languages**. Berlin: Springer-Verlag, 1992. p.157–173. (Lecture Notes on Computer Science, v.574).

[FOS95] FOSTER, I. **Designing and Building Parallel Programs.** Reading: Addison Wesley, 1995. 430p. (disponível em `http://www.mcs.anl.gov/dbpp/`).

[GEI90] GEIHS, K.; HOLLBERG, U. Retrospective on DACNOS. **Communications of the ACM,** New York, v.33, n.4, p.439-448, Apr. 1990.

[GEL85] GELERNTER, D. Generative Communication in Linda. **ACM Trans. on Programming Languages and Systems,** New York, v.7, n.1, p.80–112, Jan. 1985.

[HEA90] HEATH, M.T; ETHERIDGE, J.A. Visualizing the Performance of Parallel Programs. **IEEE Software,** New York, v.8, n.5, p. 29–39, Sept. 1991.

[HEL94] HELANDER, Johannes. **Unix under Mach:** The Lites Server. Helsinki: Department of Computer Science, Helsinki University of Technology, 1994. Master Thesis.

[HUP91] HUPFER, S. et al. Coordination Applications of Linda. In: BANATRE, J. B., METAYER, D. (Eds.). **Research Directions in High–Level Parallel Programming Languages**. Berlin: Springer-Verlag, 1992. p.187–194. (Lecture Notes on Computer Science, v.574).

[IEE95] IEEE STANDARDS DEPARTMENT. **POSIX System Application Program Interface:** Threads Extensions (C Language), IEEE Std. 1003.1c. New York, 1995.

[KAA89] KAASHOEK, M.F.; BAL, H.E. **An Evaluation of the Distributed Data Structure Paradigm in Linda.** Amsterdam: Department of Computer Science, Vrije Universiteit, May 1989. (Technical Report, IR-173).

[KAR88] KARP, A.H; BABB, R.G. A Comparison of 12 Parallel Fortran Dialects. **IEEE Software,** New York, v.5, p.52–67, 1988.

[KLE96] KLEIMAN, S. et al. **Programming with Threads.** Mountain View: Prentice-Hall (SunSoft Press), 1996. 534p.

[LAR94] LARRABEE, A.R. The p4 Parallel Programming System, the Linda Environment, and Some Experiences with Parallel Computation, **Scientific Programming,** [S.l.], v.3, p.61–71, 1994.

[LEI89] LEICHTER, J. **Shared Tuple Memories, Shared Memories, Buses and LAN's — Linda Implementations Accross the Spectrum of Connectivity.** New Haven: Department of Computer Science, Yale University, Jul. 1989. Ph.D. Thesis.

[LEL90] LELER, W. Linda Meets Unix. **IEEE Computer,** New York, v.23, n.2, p.43–55, Feb. 1990.

[LUC86] LUCCO, S. A Heuristic Linda Kernel for Hypercube Multiprocessors. In: CONFERENCE ON HYPERCUBE MULTIPROCESSORS, 1986, Knoxville, Tennessee, EUA. **Proceedings...** Philadelphia: SIAM Publications, 1986. 286p.

[MAT93] MATOS, G.; PURTILO, J. **Reconfiguration of Hierarchical Tuple Spaces: Experiments with Linda-Polylith.** Maryland: Department of Computer Science, University of Maryland, Oct. 1993. (Technical Report, CS-TR-3153). (disponível em `ftp://ftp.cs.umd.edu/pub/papers/papers/3153/3153.ps.Z`).

[MAT94] MATTSON, T.G. Programming Environments for Parallel Computing: A Comparison of CPS, Linda, P4, PVM, POSYBL, and TCGMSG. In: SOFTWARE TECHNOLOGY: HAWAII INTERNATIONAL CONFERENCE, 27., 1994, Wailea. **Proceedings...** New York: IEEE Computer Society Press, 1994. p.586–594.

[MAT95] MATTSON, T.G. Programming Environments for Parallel and Distributed Computing: A Comparison of p4, PVM, Linda and TCGMSG. **The International Journal of Supercomputer Applications and High Performance Computing,** Cambridge, v.9, n.2, p.138–161, 1995.

[MAT96] MATTSON, T.G. **Information on Network-Linda.** Correspondência trocada através de correio eletrônico (`tgm@ssd.intel.com`), Junho 1996.

[MDO90] MacDONALD, N.B. **Prolog-Linda.** Edinburgh: Department of Computer Science, University of Edinburgh, 1990. Project Report.

[NAR89] NAREM Jr., J.E. **An Informal Operational Semantics of C-Linda V2.3.5.** New Haven: Department of Computer Science, Yale University, Dec. 1989. (Technical Report, 839). (disponível em `http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda_semantics.dvi`).

[NEL92] NELKEN, I.; BJORNSON, R. Fast Lady. **Risk Magazine,** [S.l.], v.5, n.4, Apr. 1992.

[PEA90] PEARSON, P.K. Fast Hashing of Variable-Length Text Strings. **Communications of the ACM,** New York, v.33, n.6, p.677–679, June 1990.

[PIN91] PINAKIS, J. The Design and Implementation of a Distributed Linda Tuple Space. In: UWA DEPARTMENT OF COMPUTER SCIENCE RESEARCH CONFERENCE, 2., 1991, Australia. **Proceedings...** Nedlands: University of Western Australia, 1991.

[PIN91a] PINAKIS, J.; MacDONALD, C. The Inclusion of the Linda Tuple Space Operations in a Pascal-based Concurrent Language, In: AUSTRALIAN COMPUTER SCIENCE CONFERENCE, 14., 1991, Kensington, Australia. **Proceedings...** [S.l: s.n.], 1991.

[POW91] POWELL, M.L. et al. SunOS Multi-thread Architecture. In: USENIX WINTER CONFERENCE, Jan. 1991, Dallas. **Proceedings...** Berkeley: USENIX, 1991. p.21–25. (disponível em `http://www.sun.com/sunsoft/Developer-products/sig/threads/papers/sunos_mt_arch.ps`).

[RAS89] RASHID, R. et al. Mach: A System Software kernel. In: COMPUTER SOCIETY INTERNATIONAL CONFERENCE (COMPCON 89), 34., 1989, **Proceedings...** [S.l.: s.n.], Feb. 1989.

[RAS89a] RASHID, R. et al. Mach: A Foundation for Open Systems. In: WORKSHOP ON WORKSTATION OPERATING SYSTEMS (WWOS2), 2., 1989, Pacific Groove, California, EUA. **Proceedings...** [S.l.: s.n.], Sept. 1989.

[ROT93] ROTH, R.; SETZ, T. **LiPS**: A System for Distributed Processing on Workstations. Saarlandes: Universität des Saarlandes, June 1993. (disponível em `http://www-jb.cs.uni-sb.de/LiPS/doc/html/Install_and_work/Manual2.1/lipsdist`).

[SCH91] SCHOINAS, G. **Issues on the implementation of PrOgramming SYstem for distriButed appLications.** Crete: Department of Computer Science, University of Crete, Greece, 1991.

[SCI92] SCIENTIFIC COMPUTING ASSOCIATES, INC. **C-Linda User's Guide and Referenc Manual.** New Haven: Scientific Computing Associates Inc., 1992.

[SEY93] SEYFARTH, B. et al. **Glenda Installation and Use.** University of Southern Mississippi, Nov. 1993. (disponível em `http://sushi.st.usm.edu/~jbickham/glenda/glenda.html`).

[SHE92] SHEKHAR, K.; SRIKANT, Y. Linda Subsystem on Transputers. **Computer Languages,** Washington D.C., v.18, n.2, p.125–136, 1992.

[SIL94] SILVA, L.M. et al. The Helios Tuple Space Library, In: EUROMICRO WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING, 2., Malaga, 1994. **Proceedings...** p.325–333, Jan. 1994.

[STE90] STEVENS, W.R. **Unix Network Programming.** Englewood Cliffs: Prentice-Hall, 1990. 772p.

[STE92] STEIN, D.; SHAH, D. Implementing Lightweight Threads. In: USENIX SUMMER CONFERENCE, 1992, San Antonio. **Proceedings...** Berkeley: USENIX, 1992. p.1–10. (disponível em `http://www.sun.com/sunsoft/Developer-products/sig/threads/papers/impl_threads.ps`).

[SUN87] SUN MICROSYSTEMS. **XDR:** External Data Representation Standard. Mountain View: Sun Microsystems Inc., June 1987. 20p. (Request For Comments, 1014). (disponível em `ftp://ds.internic.net/rfc/rfc1014.txt`).

[SUN90] SUNDERAM, V.S. PVM: A Framework for Parallel Distributed Computing. **Concurrency: Practice and Experience,** Chichester, v.2, n.4, p.315–349, Dec. 1990.

[SUN94] SUN Microsystems Inc. **Pthreads and Solaris Threads: A Comparison of Two User Level Threads APIs.** Montain View: Sun Microsystems Inc., May 1994. 59p. (disponível em `http://www.eu.sun.com/developer-products/sig/threads/doc/pthreads_comparison.ps`).

[TAN92] TANENBAUM, A.S. **Modern Operating Systems.** Englewood Cliffs: Prentice-Hall, 1992. 730p.

[TRE92] TRESCHER, J. et al. Modula-L: Implementation of the Linda Model for Arbitrary Transputer Networks. In: **Parallel Computing: From Theory to Sound Practice**. Amsterdam: IOS Press, 1992.

[ZHO95] ZHOU, H.; GEIST, A. "Receiver Makes Right" Data Conversion in PVM. In: INTERNATIONAL PHOENIX CONFERENCE ON COMPUTERS AND COMMUNICATIONS, 14., 1995, Scottsdale. **Proceedings...** New York: IEEE Computer Society Press, 1995. p.458–464.

Informática  UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Yali, Uma Extensão do Modelo Linda para Programação Paralela em Redes Heterogêneas.

por

Andréa Schwertner Charão

Dissertação apresentada aos Senhores:

João Paulo F. W. Kitajima

Prof. Dr. João Paulo Fumio Whitaker Kitajima (DCC/UFMG)

Antônio Carlos da Rocha Costa

Prof. Dr. Antônio Carlos da Rocha Costa (PUCRS)

S S Toscani

Prof. Dr. Simão Sirineo Toscani

Vista e permitida a impressão.
Porto Alegre, 03/04/97.

Celso Costa

Prof. Dr. Celso Maciel da Costa
Orientador

Flávio Rech Wagner

Prof. Flávio Rech Wagner
Coordenador do Curso de Pós-Graduação
em Ciência da Computação - CPGCC
Instituto de Informática - UFRGS