154105-0

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TFPS - Um Sistema de Pré-processamento de Traces para Auxiliar na Visualização de Programas Paralelos

por

DENISE STRINGHINI

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

Prof. Philippe O. A. Navaux
Orientador

Porto Alegre, abril de 1997.

UFRGS

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Stringhini, Denise

TFPS - Um Sistema de Pré-processamento de Traces para Auxiliar na Visualização de Programas Paralelos / por Denise Stringhini. - Porto Alegre: CPGCC da UFRGS, 1997.

97p.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1997. Orientador: Navaux, Philippe O. A.

1. Processamento Paralelo. 2. Visualização. 3. Monitoração. 4. Avaliação de Desempenho. I. Navaux, Philippe O . A . II. 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

Agradeço a todas as pessoas que direta ou indiretamente ajudaram na conclusão deste trabalho. Aí vão os "créditos" :

Produção e Direção
Philippe O. A. Navaux

Roteiro
João Paulo F. W. Kitajima

Consultoria
Lisiane Pioner Ramos

Efeitos Especiais
Roberta Jungblut Hessel
Isabel Harb Manssour
João Frederico Lacava Schramm
Ricardo Menna Barreto
Rita de Cássia Pivetta Machado

Abstract
Míriam Pilz Albrecht
Ferdinand Baumgartner

Participações Especiais
Débora Dias Eggers
Renato Albrecht
Mirela Eidt
Alessandra Correia
Letícia Janicsek
and
Yuri Teixeira
as
"The boyfriend"

Agradecimento Especial
José Luiz Stringhini
Maria de Lourdes Stringhini
Thaís Stringhini

Patrocínio
CNPq

Sumário

Lista de Figuras 6

Lista de Tabelas 7

Resumo 8

Abstract 9

1 Introdução 10

2 Conceitos Fundamentais 15

2.1 Arquitetura de Computadores para Processamento Paralelo 15
2.2 Paradigmas de Programação Paralela 18
2.3 Bibliotecas de Programação Paralela 19
2.3.1 PVM 19
2.3.2 MPI 20
2.4 Monitoração para Avaliação de Desempenho de Programas Paralelos 22
2.5 Monitoração por *software* 25
2.5.1 Detecção de um evento 25
2.5.2 Instrumentação do programa 26
2.5.3 Formato de um arquivo de *trace* 27

3 Características das ferramentas de depuração e análise 31

3.1 Classificação Quanto ao Objetivo 33
3.2 Classificação Quanto ao Modo de Execução 34
3.3 Classificação Quanto à Apresentação dos Resultados 36
3.4 Classificação Quanto ao Modo de Utilização 37
3.5 Análise de Ferramentas para PVM e MPI 38
3.5.1 Ferramentas para PVM 38
3.5.2 Ferramentas para MPI 39
3.5.3 PVM x MPI 42

4 Descrição do *Trace File Preprocessor System* - TFPS 44

4.1 Visão Geral 44
4.2 Projeto das Janelas de Visualização 48
4.2.1 Diagrama Espaço-tempo 48
4.2.2 Diagrama Kiviat 50
4.2.3 Diagrama de Mapeamento de Processos 51
4.2.4 Grafo de Processos 52

5 Características de Implementação do Pré-processador TFP 56

5.1 Estruturas de Dados 56
5.2 Leitura do arquivo de *Trace* 58
5.3 Processamento das Informações 60
5.4 Algoritmo gerador do Grafo de Processos 63

5.3 Processamento das Informações 60
5.4 Algoritmo gerador do Grafo de Processos 63
5.5 Formato do arquivo de saída 65

6 Montagem das Janelas de Visualização 67

6.1 Janelas do Protótipo TFP View 67
6.2 Comparação com outras ferramentas 72
6.2.1 TFP View x ParaGraph 73
6.2.2 XPVM e PVaniM 79

7 Conclusão 87

Bibliografia 90

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Sistema de Biblioteca da UFRGS

6437

INF 6437 05229401

681.32.06(043) S918t

[0154105] Stringhini, Denise. TFPS : um sistema de pré-processamento de traces para auxiliar na visualização de programas paralelos. 1997. 97 f. : il.

MOD. 2.3.2

Lista de Figuras

FIGURA 2.1 - Taxonomia de Flynn estendida por Tanenbaum 16
FIGURA 2.2 - Taxonomia clássica para avaliação de desempenho 22
FIGURA 3.1 - Bibliotecas utilizadas pelo *linker* do MPI 40
FIGURA 4.1 - Descrição do TFPS 45
FIGURA 4.2 - Exemplo de Diagrama Espaço-tempo ordenado 49
FIGURA 4.3 - Exemplo de Kiviat de utilização da rede 51
FIGURA 4.4 - Exemplo do Diagrama de Mapeamento de Processos 52
FIGURA 4.5 - Exemplo do Grafo de Processos 54
FIGURA 5.1 - Lista de adjacências do TFP 56
FIGURA 5.2 - Lista de *hosts* do TFP 58
FIGURA 5.3 - Exemplo de criação dos nomes alfanuméricos 60
FIGURA 5.4 - Arquivo .g para ferramenta Nature 63
FIGURA 5.5 - Arquivo .nat para ferramenta XGraphDrawer 65
FIGURA 5.6 - Trecho de um arquivo Info do TFP 66
FIGURA 6.1 - ParaGraph com botões para ativação do TFP View 67
FIGURA 6.2 - Space-time Diagram do TFP View 68
FIGURA 6.3 - Kiviat Diagram do TFP View 69
FIGURA 6.4 - Processes Mapping Diagram do TFP View 70
FIGURA 6.5 - Grafo gerado pela ferramenta Nature 71
FIGURA 6.6 - Legenda para o Grafo de Processos 72
FIGURA 6.7 - Space-time do ParaGraph 74
FIGURA 6.8 - Kiviat do ParaGraph 76
FIGURA 6.9 - Animation do ParaGraph (janela inicial) 77
FIGURA 6.10 - Animation do ParaGraph (RING) 78
FIGURA 6.11 - Animation do ParaGraph (USER) 79
FIGURA 6.12 - Janelas do XPVM 81
FIGURA 6.13 - Causality View do PVaniM 83
FIGURA 6.14 - Message Passing View do PVaniM 84
FIGURA 6.15 - HostList do PVaniM 85
FIGURA 6.16 - Messages Sent/bytes sent do PVaniM 86

Lista de Tabelas

TABELA 2.1 - Classificação das ferramentas de monitoração 24
TABELA 2.2 - Estrutura básica de um arquivo de *trace* PICL 28
TABELA 2.3 - Pseudo-eventos do Upshot 28
TABELA 3.1 - Características das ferramentas de depuração e análise 32
TABELA 4.1 - Cabeçalho de um evento TAPE/PVM 46
TABELA 4.2 - Exemplo da parte variável dos eventos TAPE/PVM 46
TABELA 6.1 - Tabela comparativa entre as janelas analisadas 86

Resumo

O trabalho apresenta o projeto e o desenvolvimento de uma ferramenta para visualização lógica da execução de programas paralelos, a **TFPS** de *Trace File Preprocessor System*, cujo objetivo é a análise de desempenho de tais programas. O projeto é baseado no pré-processamento de arquivos de *traces* de execução dos programas. A idéia básica consiste em aproveitar as informações fornecidas pela monitoração. Estas informações, que em geral são utilizadas apenas para dirigir animação *post-mortem* destes programas, neste caso são utilizadas também na montagem das janelas de visualização.

Assim, são descritos o pré-processador e a montagem das janelas de visualização. O primeiro, é responsável principalmente pela leitura e análise das informações contidas no arquivo de *trace* e pela geração de um arquivo de saída com todas as informações necessárias à montagem das janelas. Estas, foram concebidas levando em consideração o tipo de informação que pode ser obtido de um arquivo de *trace*. Desta forma, foi possível aproximar o conteúdo das janelas de visualização o máximo possível do programa paralelo em análise. Com o objetivo de demonstrar esta aproximação foi construído um protótipo tanto para o pré-processador quanto para a ferramenta de visualização. Ambos os protótipos são descritos neste trabalho.

Palavras-chave: Processamento Paralelo, Visualização, Monitoração, Avaliação de Desempenho

Title: "TFPS - A Traces Preprocessing System to Aid in Parallel Programs Visualization"

Abstract

This study presents the project and development of a logical visualization tool for parallel programs, the **TFPS** of Trace File Preprocessor System, whose goal is the performance analysis of such programs. The project is based on the preprocessing of trace files of programs' execution. The basic idea consists in making use of the information given by the monitoring process. This information, whose general application is only to drive the *post-mortem* animation of these programs, is in this case also used to create the visualization displays.

Thus, the preprocessor and the creation of visualization displays are described. The first is mainly responsible for reading and analyzing the information present in the trace file and for generating an output file with all information necessary for creating the views. The latter was conceived by taking into consideration the type of information that can be obtained from a trace file. Therefore it was possible to make the content of the visualization displays close to the parallel program that is being analyzed. A prototype of the preprocessor as well as of the visualization tool was built up in order to demonstrate the described approach. Both prototypes are described in this study.

Keywords: Parallel Processing, Visualization, Monitoring, Performance Analysis

1 Introdução

Nos últimos anos, o processamento paralelo tem se tornado uma das principais alternativas em se tratando de computação de alto desempenho. Neste contexto, a necessidade de que se desenvolvam técnicas e ambientes de programação paralela cada vez mais apurados e próximos ao usuário é inegável. Deste ponto, depende em grande parte o avanço da computação paralela.

Assim como a computação convencional (seqüencial), a programação paralela requer ambientes que facilitem a programação, depuração e avaliação de desempenho de programas. Entretanto, ela se torna mais complexa, na medida em que cada programa é constituído de vários processos que devem interagir em busca de um resultado.

Esta complexidade reside no fato de que os vários processos que compõem um programa podem estar localizados em diferentes processadores, que podem possuir diferentes cargas de processamento a cada momento, o que torna o tempo de execução de cada processo imprevisível. Além disso, visto que os processos necessitam de comunicação entre si, a carga da rede de interconexão que os liga também pode interferir no tempo de execução. Tudo isto faz com que a execução de um programa paralelo seja completamente não-determinística, ou seja, o comportamento do programa pode variar a cada execução.

Desta forma, as ferramentas para programação paralela devem possuir uma preocupação extra que é a de não sobrecarregar o sistema com sua própria execução. Isto poderia acarretar um comportamento diferente ao programa paralelo, inclusive podendo mascarar ou ocasionar erros em sua execução que não fazem parte da execução original (realizada sem o auxílio da ferramenta).

A partir desta restrição surge um mecanismo muito utilizado com o objetivo de reduzir ao máximo esta interferência. Este mecanismo consiste na geração de um *trace* de execução do programa paralelo, que é utilizado de forma *post-mortem* (após a execução da aplicação), em contrapartida às ferramentas que executam *on-line* (juntamente com a aplicação). Entretanto, cabe salientar que ferramentas híbridas também são, com freqüência, encontradas.

O *trace* consiste na monitoração do programa paralelo durante sua execução, coletando informações que possibilitem ou sua reexecução completa, de uma forma determinística, ou uma espécie de "relatório", com informações importantes sobre o comportamento do programa. Este relatório tem o objetivo de fornecer parâmetros para que o programador tire suas próprias conclusões sobre o desenvolvimento da aplicação.

O escopo deste trabalho abrange os programas paralelos baseados em troca de mensagens. Neste tipo de programação, a interação entre os processos se dá exclusivamente através da troca de mensagens entre eles, onde, a cada interação, um faz o papel de emissor e o outro (ou outros), o de receptor(es). Não há o compartilhamento de uma memória global, o que faz com que as máquinas próprias a este tipo de paradigma de programação sejam as fracamente acopladas, incluindo redes de estações de trabalho.

Neste contexto, a visualização de processos e/ou processadores torna-se um mecanismo bastante eficaz quando se quer ter uma visão geral da aplicação. Vários trabalhos têm sido desenvolvidos nesta área, dos quais pode-se destacar o ParaGraph [HEA 91], escrito originalmente para programas que utilizam a biblioteca PICL [GEI 90], e o XPVM [GEI 96], para programas PVM [GEI 94]. O ParaGraph tem seu ponto forte na grande quantidade de janelas de visualização disponíveis, principalmente visando a avaliação de desempenho. O XPVM fornece várias

informações neste sentido e suas janelas de visualização foram construídas baseadas nas do ParaGraph.

Este trabalho tem como principal objetivo facilitar e tornar mais intuitiva a compreensão e a avaliação de desempenho de programas paralelos através da visualização. Para isto, fornece uma quantidade maior de informações em cada janela de visualização e as organiza de forma que, à primeira vista, o programador já tenha uma idéia de quais são os principais problemas da aplicação que está desenvolvendo. Tem, então, a possibilidade de executar uma animação da execução do programa em várias janelas, que fornecem diferentes parâmetros de avaliação.

O grande diferencial em relação às outras ferramentas, já citadas, é a utilização de uma grande quantidade de informações do *trace* de execução numa fase pré-animação. Através da análise do *trace* antes da montagem das janelas, é possível organizá-las de tal forma que facilite a visualização, uma vez que já serão conhecidas diversas informações relativas ao comportamento do programa durante sua execução.

O presente trabalho consiste no projeto e na construção de um protótipo de uma ferramenta que se divide em duas partes. A primeira é a fase de pré-processamento que consiste em coletar informações de um arquivo de *trace*, organizá-las e gravá-las num outro arquivo, de tal forma que possibilite a próxima etapa, que é a da montagem das janelas de visualização. O formato escolhido para o arquivo de *trace* foi o do TAPE/PVM [MAI 95], que é um *software* de monitoração para aplicações PVM.

A segunda parte consiste em captar as informações fornecidas pelo pré-processador e, a partir daí, gerar as janelas que posteriormente serão animadas. Num primeiro momento, as janelas farão parte de um protótipo implementado no ambiente ParaGraph, que possibilita a criação de novas janelas que possam executar em

conjunto com as já existentes. Isso é possível pelo fato da ferramenta TAPE/PVM incluir a possibilidade de transformação do seu formato para o PICL [WOR 92], que é reconhecido pelo ParaGraph.

Quanto à organização deste texto, no segundo capítulo são introduzidos conceitos fundamentais, necessários à compreensão do texto pelo leitor menos familiarizado com o assunto tratado no trabalho, como por exemplo arquiteturas paralelas, paradigmas de programação paralela e bibliotecas de programação. Além disso, também introduz-se o conceito de monitoração, juntamente com uma abordagem sobre o tipo de informação que é trabalhada no arquivo de *trace* e como esta informação pode ser obtida. Também são apresentados alguns formatos de arquivos de *trace* existentes, assim como suas principais características.

No terceiro capítulo será apresentada uma classificação das características de ferramentas existentes. Isto é feito antes de se aprofundar o assunto para que se situe o trabalho dentro dos diversos tipos de ferramentas de depuração existentes e que vêm sido extensivamente pesquisadas.

A seguir, no quarto capítulo, é descrito o **TFPS** (*Trace File Preprocessor System*) desenvolvido neste trabalho. Ele é composto por duas ferramentas: a **TFP**, que é o pré-processador de *traces* e a **TFP View**, que implementa as janelas de visualização. É apresentada uma visão geral da integração destas duas ferramentas com a de monitoração e é descrito o projeto das janelas de visualização.

Depois disso, no quinto capítulo, são apresentadas as principais características de implementação do pré-processador, o **TFP**. Isto é feito para reforçar a idéia de que o pré-processamento, apesar de ser uma fase a mais na animação *post-mortem*, é perfeitamente viável e sua implementação é bastante simples. Além disso,

este capítulo demonstra exatamente quais os tipos de informações que são retiradas do arquivo de *trace* e como elas são organizadas e gravadas no arquivo de saída.

No sexto capítulo é apresentado o protótipo da ferramenta **TFP View**, com todas as janelas idealizadas neste trabalho. Isto é feito tanto a título de demonstração, quanto para fazer uma espécie de validação da pesquisa, incluindo comparações com ferramentas existentes.

2 Conceitos Fundamentais

Este capítulo tem o objetivo de fornecer subsídios para que o leitor menos familiarizado com os conceitos que envolvem a programação paralela/distribuída não tenha problemas em compreender capítulos restantes. Neste, são apresentados conceitos básicos, como os de arquiteturas para processamento paralelo, paradigmas e ambientes para programação paralela. Além disso, procura-se inserir o presente trabalho neste contexto.

2.1 Arquitetura de Computadores para Processamento Paralelo

Desde o início da construção dos computadores a idéia de se ter mais de um processador atuando em paralelo já existia, o que é normal, visto que a idéia da distribuição de trabalho é intuitiva [KIT 95a]. Na época atual, onde os limites físicos das tecnologias de fabricação de processadores são constantemente questionados, o processamento paralelo tem surgido com muita força como a alternativa mais imediata ao problema da melhoria de desempenho dos sistemas de computação.

Os elementos básicos que definem uma arquitetura paralela são os seguintes: o tipo e o número dos elementos de processamento (EPs), a natureza dos módulos de memória e a rede de interconexão que interliga todos os elementos. Dentro destas características básicas, as possibilidades de diferentes combinações são inúmeras, o que faz com que surjam diferentes projetos de novas arquiteturas a todo o momento.

Desta forma, fica difícil adotar como definitiva qualquer tentativa de classificação destas arquiteturas. Dentre estas tentativas, a Taxonomia de Flynn [FLY 72 *apud* TAN 92] tem sido a mais constantemente utilizada, apesar de limitada. Ela classifica os computadores através da análise do fluxo de instruções (*instruction ou I*) e de dados (*data ou D*). Estes podem ser únicos (*single ou S*) ou múltiplos (*multiple ou M*). Desta forma, segundo Flynn, existem quatro categorias de computadores: SISD, SIMD, MISD e MIMD.

Tanenbaum [TAN 92] estendeu esta classificação incluindo os computadores SIMD vetoriais e paralelos, além de utilizar uma já amplamente difundida divisão entre computadores MIMD: memória compartilhada (os **multiprocessadores**) e distribuída (os **multicomputadores**). Estas duas categorias, por sua vez, subdividem-se levando em consideração o tipo de acesso aos módulos de memória e/ou demais EPs: por chaveamento (dinâmico) ou por barramento (estático). Esta taxonomia pode ser visualizada na figura 1.1.

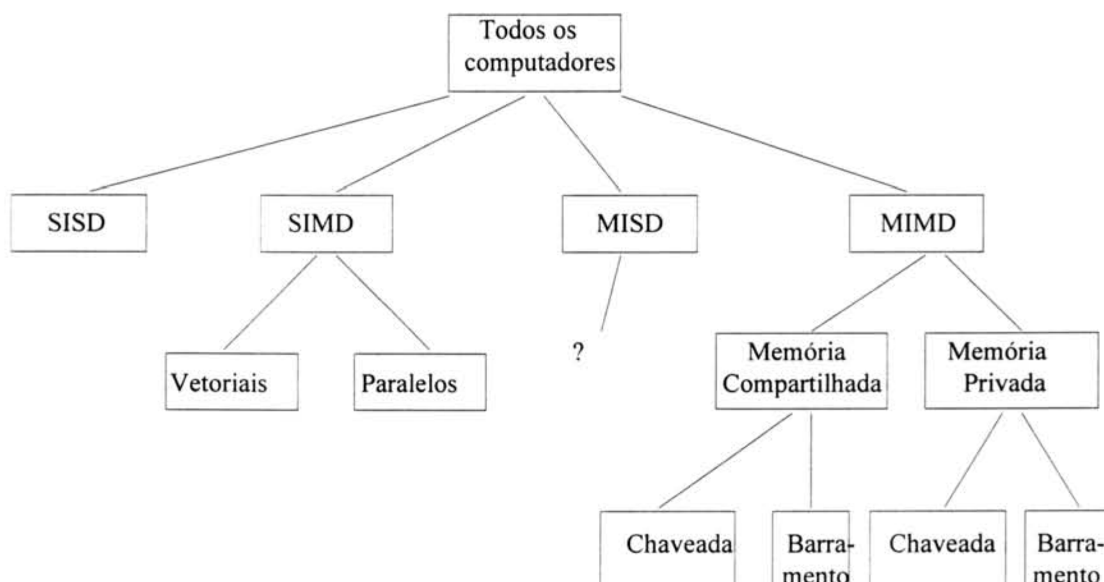


FIGURA 2.1 - Taxonomia de Flynn estendida por Tanenbaum

O escopo do presente trabalho se insere na categoria de computadores paralelos do tipo **MIMD**, com **memória privada** e comunicação entre processadores/memórias através de **barramento**. Isto se deve ao fato de que o *hardware* para o qual a ferramenta desenvolvida se destina em especial são as **redes de estações de trabalho**. Estas, com o avanço das tecnologias de interconexão, já são consideradas **multicomputadores** e como tal têm sido constante objeto de estudo. Isto se deve às grandes vantagens trazidas por este tipo de abordagem, sendo as principais o custo e a flexibilidade. Salienta-se, entretanto, que a ferramenta proposta não

apresenta nenhuma restrição quanto ao seu uso em outras plataformas de *hardware*, desde que façam parte do grupo dos multicomputadores.

As redes de estações de trabalho se caracterizam, é claro, por possuirem CPUs completas e grande capacidade de memória. Esta característica as definem como sendo máquinas multicomputadoras **fracamente acopladas**, onde a conexão entre os processadores, apesar da alta tecnologia, ainda é considerada relativamente lenta. Um dos principais motivos é o fato de que a comunicação entre os elementos de processamento se dá através da troca de mensagens, o que além de poder causar um tráfego intenso de mensagens na rede de interconexão, aumenta a complexidade dos algoritmos. Entretanto, vale lembrar a grande vantagem deste esquema, que são o custo e a flexibilidade.

Esse tipo de máquina se opõe às **fortemente acopladas** onde ou existe o compartilhamento de memória ou os componentes são muito pequenos, próximos e se comunicam freqüentemente, mas a taxas e volume de transmissão muito menores. A vantagem das máquinas onde há o compartilhamento de memória é a facilidade de programação, que herda todos os conceitos já bastante estudados e conhecidos de controle de concorrência.

Por este motivo, existe muito estudo em cima de esquemas que combinem as vantagens da memória compartilhada com a memória distribuída. O principal deles é a "emulação" por *software* de memória compartilhada em cima de um *hardware* de memória distribuída. Exemplos desta abordagem são as linguagens de programação Linda [CAR 94, CHA 96] e Orca [BAL 88 *apud* TAN 92].

As características de *hardware* citadas ainda estão intimamente ligadas a um outro conceito, só que de *software*, que é o de **tamanho do grão**. Quando o sistema é fracamente acoplado, como nas redes de estações de trabalho, a quantidade de elementos de processamento é mais reduzida e a unidade de paralelismo é maior: no nível de processos ou de sub-processos. Diz-se que sistemas deste tipo são de **granularidade grossa**, em oposição aos que possuem, por exemplo, paralelismo ao nível de instrução, que são os de **granularidade fina**. O presente trabalho se preocupa em demonstrar e analisar o comportamento de sistemas de granularidade grossa, onde

o paralelismo está no nível de processos paralelos que correspondem a um mesmo programa.

2.2 Paradigmas de Programação Paralela

Em se tratando de máquinas MIMD, o *software* paralelo herda suas características e, portanto, a subdivisão do *hardware*. Seguindo este princípio, temos que a programação paralela também é subdividida em dois grandes grupos: baseada em memória compartilhada e memória distribuída [KIT 95a].

O primeiro grupo possui a vantagem de ser mais acessível, do ponto de vista do programador. Além disso, todo o estudo existente em programação concorrente pôde ser reaproveitado nestes sistemas onde os vários processadores têm acesso a uma memória comum, o que permite que os diversos processos que compõem um programa compartilhem o mesmo espaço de endereçamento. Entretanto, os problemas com este tipo de sistema estão no nível do *hardware*, onde a principal causa é o fator de escalabilidade, que é muito fraco.

Por este motivo, uma das abordagem que mais tem sido foco de estudos é a simulação de memória compartilhada sobre arquiteturas com memória distribuída. Desta forma, pretende-se juntar o útil ao agradável, facilitando o trabalho tanto para os programadores quanto para os fabricantes de *hardware*.

Não se deve esquecer, porém, que o segundo grupo, que abrange as máquinas com memória distribuída, sempre necessitará de comunicação via **troca de mensagens** entre os processadores/processos, ainda que isso aconteça apenas nas camadas inferiores de *software*. Este tem sido um grande gargalo no desenvolvimento de programas paralelos, visto que este tipo de paradigma de programação não é nada trivial. Daí surge a necessidade de ambientes e ferramentas de programação que sejam amigáveis o suficiente para que atraiam o programador e para que finalmente se alcance o tão esperado *boom* da programação paralela/distribuída.

2.3 Bibliotecas de Programação Paralela

O objetivo desta seção é abordar duas das principais alternativas em termos de bibliotecas para a programação paralela. Visto que o foco deste trabalho se encontra nos ambientes com memória distribuída, principalmente *clusters* de estações de trabalho, serão abordadas apenas aquelas que se utilizam do paradigma baseado em troca de mensagens e que são as mais utilizadas neste tipo de configuração: **PVM** - *Parallel Virtual Machine* [GEI 94] e **MPI** - *Message Passing Interface* [MAL 96].

2.3.1 PVM

A biblioteca PVM é uma infraestrutura de *software* que emula um sistema com memória distribuída num ambiente de rede heterogêneo, que pode ser composto por uma grande variedade de máquinas diferentes (de PC's a supercomputadores). O PVM permite que se crie uma **máquina virtual** composta de um número praticamente ilimitado de *hosts* heterogêneos.

Diversos usuários podem configurar diferentes máquinas virtuais ao mesmo tempo (sobrepostas) e cada usuário pode executar diversas aplicações PVM simultaneamente. O PVM fornece funções para a criação de processos (*tasks*) na máquina virtual e permite que estes se comuniquem e se sincronizem uns com os outros. Um processo ou *task* é definido como uma unidade computacional em PVM e é análogo a um processo Unix (em geral **é** um processo Unix) [SUN 94].

As aplicações, escritas em C ou Fortran, podem ser paralelizadas através das primitivas para troca de mensagens que são comuns na maioria dos sistemas com memória distribuída. Com a troca de mensagens implementada através de primitivas do tipo *send/receive*, os vários processos que compõem a aplicação podem cooperar para a resolução do problema em paralelo.

O modelo computacional do PVM assume que qualquer processo pode enviar uma mensagem para qualquer outro processo PVM e que não há limite para o tamanho ou para o número destas mensagens. O modelo de comunicação do PVM fornece envio (*send*) e recepção (*receive*) de mensagens assíncronos e ainda recepção

síncrona. Note-se que o envio de mensagens é de uma certa forma bloqueante, já que o processo emissor espera até que o *buffer* de envio esteja livre para reutilização. Observa-se ainda que a recepção síncrona (ou bloqueante) difere da assíncrona (não-bloqueante) por colocar o processo receptor em estado de espera até que a mensagem que lhe é destinada chegue ao *buffer* de recepção.

Além das primitivas de comunicação mencionadas, a biblioteca PVM ainda fornece funções para:

- inicialização e término de processos PVM;
- adição e remoção de *hosts* da máquina virtual;
- sincronização e envio de sinais entre processos PVM;
- obtenção de informações sobre a configuração da máquina virtual e processos ativos;
- empacotamento e desempacotamento de dados;
- difusão de mensagens (*multicast* e *broadcast*);
- criação dinâmica de grupos de processos.

2.3.2 MPI

O projeto MPI - *Message Passing Interface*, consiste numa proposta conjunta, entre pesquisadores e fabricantes, de padronização para a troca de mensagens em computadores MIMD. O principal objetivo é a construção de um ambiente em que as aplicações, bibliotecas de *software* e ferramentas sejam portadas de forma transparente entre diferentes máquinas [WAL 94].

O MPI implementa conceitos já difundidos e aplicados anteriormente em outras bibliotecas baseadas em troca de mensagens. Os principais são a comunicação ponto-a-ponto e coletiva, grupos de processos, contexto de mensagens, tipos de dados e topologias virtuais [MAL 96].

A **comunicação** entre processos (entidades únicas que realizam alguma tarefa computacional) é a única forma pela qual um processo pode acessar dados contidos na memória local de um outro. A comunicação ponto-a-ponto, que envolve apenas dois processos (emissor e receptor) é suportada por um conjunto de rotinas de envio e recepção (*send/receive*) que podem ser bloqueantes ou não-bloqueantes. A comunicação coletiva inclui uma boa variedade de tipos encontrados em outras implementações (todos bloqueantes): barreiras, *broadcast, scatter and gather, reduce and scan, gather-to-all* [MAL 96].

Grupos de processos podem ser definidos de forma estática e tipicamente são compostos por aqueles que têm uma tarefa em comum. O **contexto de mensagens** é aplicado com o objetivo de impedir que mensagens ambíguas, enviadas principalmente pelo usuário ou pelo sistema, possam ser confundidas e recebidas no momento errado por um dado processo.

Topologias virtuais podem ser implementadas com o objetivo de que haja uma correlação entre o algoritmo e a topologia de comunicação entre os processos. Dois tipos de topologia podem ser utilizados: a cartesiana (definida por um sistema de coordenadas) e a de grafo (definida pelos vizinhos mais próximos de cada processo).

Os **comunicadores** são objetos usados para definir o escopo de uma operação de comunicação, ou seja, o contexto utilizado, topologia virtual e o grupo ou grupos envolvidos. Dependendo do caso, por exemplo a comunicação ponto-a-ponto entre processos do mesmo grupo, apenas um grupo deverá ser especificado. No caso destes processos pertencerem a grupos diferentes, os dois deverão ser especificados. Os comunicadores são do tipo *opaque objects* e, desta forma, sua utilização fica limitada à passagem de argumentos para as rotinas MPI e a expressões lógicas. Todos os processos envolvidos numa comunicação devem fornecer o mesmo comunicador como argumento nas chamadas às rotinas MPI correspondentes.

Entre os conceitos não implementados em MPI, onde alguns estão sob análise para implementações futuras, estão a gerência de processos, a transferência remota de blocos de memória, as mensagens ativas, os processos leves (*threads*) e a memória virtualmente compartilhada [KIT 95a].

Apesar de possuir grandes vantagens e cada vez mais adeptos, a biblioteca MPI não foi a escolhida como o objetivo deste trabalho. Ao invés disto, optou-se pelo PVM, que além de ainda se manter na posição de "padrão de fato" entre os programadores de aplicações paralelas, possui diversas ferramentas de monitoração já implementadas, dentre as quais uma foi escolhida para utilização neste trabalho.

2.4 Monitoração para Avaliação de Desempenho de Programas Paralelos

Existem diversas técnicas para avaliação de desempenho de programas paralelos, como apresentado em [MEN 93]. Estas técnicas podem ser classificadas em três tipos básicos: mensuração, modelos analíticos e simulação, como demonstra a taxonomia clássica da figura 2.2.

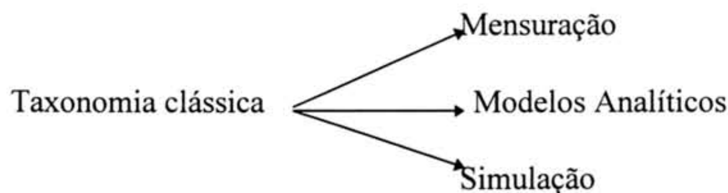


FIGURA 2.2 - Taxonomia clássica para avaliação de desempenho

A monitoração se insere dentre as técnicas de mensuração. Neste tipo de técnica, que é dita elementar ou direta, a principal característica é a **existência** do sistema em estudo e sua avaliação direta através de instrumentos. Estes instrumentos são utilizados na extração de parâmetros de desempenho e variam segundo o objeto em avaliação e a precisão desejada.

Além da monitoração, a mensuração ainda inclui *benchmarking* e programas sintéticos, entre outros. Um *benchmark* é um conjunto de programas, geralmente escrito em linguagem de alto nível, que é representativo de uma dada classe de programas. Através da execução de diversos *benchmarks*, o desempenho global de um sistema pode ser realisticamente estimado [MEN 93].

Um programa sintético é um programa artificial que pode ser construído a partir de um modelo de uma aplicação. Funciona como um programa real, em termos de demanda de recursos. Difere, no entanto, por não gerar um resultado útil. A utilização de um programa sintético freqüentemente tem como objetivo a quantificação do desempenho de *software* e *hardware* [MEN 93].

Neste trabalho, o conceito de monitoração é importante na medida em que, apesar de não ser seu objeto de estudo, é o seu principal ponto de partida e tem uma participação fundamental na construção da ferramenta proposta. Os conceitos que envolvem a monitoração, portanto, devem ser abordados para que se possa ter uma maior compreensão sobre o funcionamento da ferramenta. Salienta-se que a ferramenta proposta **não** implementará a monitoração, apenas fará uso da mesma.

O conceito de monitoração será adotado única e exclusivamente para descrever o *processo de coletar informações sobre a execução de um programa*, sem incluir a etapa da visualização destas informações. Esta será, portanto, considerada uma etapa à parte, um passo a mais para a compreensão do programa. Entretanto, cabe salientar que estas duas etapas são complementares, e que os dados que serão posteriormente visualizados dependem em grande parte da forma como foram monitorados. Note-se, inclusive, que o conceito de monitoração, ao contrário do que é considerado neste trabalho, pode incluir a apresentação destas informações ao usuário, como sugere, por exemplo, [JOY 87].

A monitoração serve para dar suporte às ferramentas de depuração, que se utilizam das informações geradas por ela. Como definiu [SNO 88], *a monitoração é o primeiro passo na compreensão de um processo computacional para o qual fornece uma indicação de **o quê** aconteceu, servindo, então, como um pré-requisito para que se deduza **o porquê** aconteceu.*

As informações geradas pela monitoração podem ser utilizadas pelas ferramentas de visualização de duas maneiras: em tempo de execução (*on-line*) ou ao término da execução do programa (*post-mortem*). As ferramentas de análise que se utilizam da primeira abordagem requerem maiores cuidados no sentido de que não sejam introduzidos pela ferramenta de depuração muitos atrasos no momento da execução original do programa. Este cuidado deve ser redobrado pelo fato de que a visualização estará sendo executada de forma concorrente com a monitoração. Isto pode ocasionar sérios problemas de intromissão que podem levar a ferramenta a causar distorções no funcionamento da aplicação em análise.

TABELA 2.1 - Classificação das ferramentas de monitoração

Tipo de Monitoração	Características
Hardware	• O monitor é um dispositivo à parte. • Restringe os pontos de observação. • Não influi no sistema monitorado. • Gera informações limitadas e de difícil compreensão.
Software	• Compartilhamento de recursos entre o sistema e a ferramenta. • Pode influir no sistema monitorado.
Híbrida	• Dispositivo de *hardware* + ferramenta de *software*. • Interferência mínima no sistema monitorado.

A segunda abordagem também corre o risco de interferir no funcionamento da aplicação, entretanto este risco se torna menor à medida em que não existe uma ferramenta de visualização executando concorrentemente. Entretanto, em ambas as abordagens este risco deve ser considerado e todo o esforço deve ser feito no sentido de evitar este tipo de mal funcionamento, que pode acabar invalidando a ferramenta de monitoração e, conseqüentemente, a de visualização também.

Existem diversas formas de se monitorar uma aplicação. Baseado nisso, foram identificados três tipos básicos de ferramentas de monitoração, apresentados em [HAB 90] que são brevemente descritos na tabela 2.1 acima.

2.5 Monitoração por *software*

A monitoração por *software* é a mais flexível e portável visto que não necessita de nenhum mecanismo de *hardware* específico. Por este motivo, será abordada com mais detalhe nesta seção. Cabe lembrar, entretanto, que esta abordagem possui algumas desvantagens, como por exemplo o grau de intromissão na execucão do programa, apesar deste fator ser considerado aceitável na maioria dos sistemas. Mais detalhes sobre este e os outros tipos de monitoração podem ser encontrados em [RIE 91].

2.5.1 Detecção de um evento

Para que determinados eventos sejam detectados, são inseridas **sondas de *software*** no código. Estas sondas devem disparar as rotinas responsáveis pela geração do histórico de execução, que é um arquivo contendo todos os eventos da aplicação que foram monitorados. Este arquivo também é conhecido como *trace file*.

A forma em que estes eventos são coletados pode variar. Isto é uma opção de implementação que pode influir bastante na eficiência da ferramenta de monitoração. Uma das opções existentes é de, durante a execução, armazenar os eventos em um *buffer* que é descarregado de tempos em tempos, gravando-se estes eventos em um arquivo. Esta opção é ideal para a análise *post-mortem* dos eventos. Outra opção, indicada para análises *on-line*, é o redirecionamento imediato do evento coletado do *buffer* da ferramenta de monitoração para o da ferramenta de visualização, que repassa, então, a informação para a tela.

2.5.2 Instrumentação do programa

O processo de inserir sondas no código do programa é denominado instrumentação de código. As abordagens mais comuns são [RIE 91]:

- **Instrumentação do código fonte**: neste caso as sondas são inseridas no próprio código da aplicação. Por este motivo, podem ser colocadas em qualquer ponto o que faz com que o monitor tenha acesso a qualquer evento da aplicação. A inserção da sonda pode ser manual ou automática. O processo manual, apesar de mais flexível, pode tornar-se muito trabalhoso, o que faz com que sejam desenvolvidas ferramentas para automatização da instrumentação.
- **Biblioteca instrumentada**: consiste num conjunto de primitivas, como as de comunicação, por exemplo, que já contém as sondas de monitoração embutidas na sua própria implementação. Estas sondas se encarregarão de gerar o histórico durante a execução da aplicação.
- **Instrumentação do código objeto**: para este tipo de instrumentação se faz necessário o uso de um compilador que instrumente o código. Este compilador insere as sondas em tempo de compilação, o que torna o processo transparente ao usuário.
- ***Kernel* instrumentado**: neste caso, as sondas (ou *hooks*, neste contexto) são inseridas diretamente no núcleo do sistema

operacional, o que faz com que a instrumentação seja completamente transparente ao usuário. Cada vez que a aplicação fizer uma chamada ao sistema, como um *send* ou *receive*, a sonda é executada e o evento detectado. Cabe salientar que esta abordagem requer modificações no sistema operacional, o que a torna bastante complexa e pouco flexível.

2.5.3 Formato de um arquivo de *trace*

O formato do arquivo de *trace* é um aspecto de vital importância no projeto de uma ferramenta de monitoração, assim como no projeto da ferramenta de visualização que dele se utilizará. É ele que define o tipo de informação que será coletada, o nível de portabilidade da ferramenta de monitoração e o porte da aplicação monitorada, já que a quantidade de informação que poderá ser coletada depende do nível de compactação que o arquivo de *trace* poderá atingir.

O tipo de informação coletada está relacionado aos tipos de eventos monitorados, seu *timestamp* (tempo em que ocorreram), processo e processador onde ocorreram e outras informações úteis. Este formato pode ser fixo, como nas ferramentas PICL [WOR 92], Upshot [FOS 95] e TAPE/PVM [MAI 94] ou ser um meta-formato, como o SDDF [AYD 94].

O novo formato para arquivos de *trace* da ferramenta PICL [WOR 92] possui um conjunto de eventos fixos, mas também suporta eventos definidos pelo usuário. A ferramenta PICL não possui uma larga utilização, mas é uma bilioteca instrumentada que gera automaticamente *traces* dos programas que utilizam suas rotinas. O seu formato é o utilizado na ferramenta ParaGraph [HEA 91] uma das mais utilizadas para avaliação de desempenho de programas paralelos, sendo uma das pioneiras a implementar uma grande variedade de análises diversas. Daí o formato de *trace* PICL ser tão utilizado, mais até que a própria biblioteca. Isto se explica pelo fato de muitas ferramentas de monitoração para outras bibliotecas incluirem programas de conversão dos seus prórios formatos para o PICL, com o objetivo de possibilitar sua visualização através do ParaGraph.

O novo formato PICL, que pretende ser um padrão apesar de ir de encontro à tendência atual que é a de se utilizar meta-formatos, tenta ser genérico o suficiente para atender às necessidades de outra ferramentas e bibliotecas similares. A tabela 1.1 apresenta a estrutura básica de um registro do arquivo de *trace*.

O formato de *trace* utilizado na ferramenta Upshot não difere muito do formato apresentado acima. Entretanto, inclui, no início do arquivo, uma série de pseudo-eventos que fornecem outras informações sobre a aplicação. Este pseudo-eventos são identificados através de tipos negativos. A tabela 2.2 apresenta estas informações. Este tipo de abordagem favorece enormemente a apresentação posterior dos resultados e é a abordagem que foi levada a fundo neste trabalho, ou seja, a extração de um maior número de informações do arquivo de *trace* para facilitar a visualização. Salienta-se porém que a ferramenta idealizada neste trabalho não coleta informações já presentes no *trace* e sim **gera** estas novas informações.

TABELA 2.2 - Estrutura básica de um arquivo de *trace* PICL

Nome do campo	**Significado**
tipo do registro	tipo de informação no registro de *trace*
tipo do evento	tipo de evento ao qual está associado
timestamp	quando a informação ficou válida
id do processador	processador ao qual está associado
id do processo	processo ao qual está associado
número de campos	número de outros campos ao qual está associado
descritor de dados	formato de outros campos
dados	outros campos

O formato do TAPE/PVM será abordado mais adiante e com mais profundidade, já que foi o escolhido para este trabalho. Sua escolha deveu-se a três fatores determinantes: a geração de *traces* para aplicações PVM, a existência de uma biblioteca com rotinas para a leitura do *trace* e a possibilidade de transformação de seu formato para o do PICL que, como já mencionado, permite a utilização do ParaGraph.

TABELA 2.3 - Pseudo-eventos do Upshot

Tipo	Processado	Processo	Dado (inteiro)	Ciclo	Timesta	Dado (string)
-1						criador e data
-2			# eventos			
-3			# processadores			
-4			# processos			
-5			# tipos de eventos			
-6					início	
-7					fim	
-8			# ciclos			
-9			tipo de evento			descrição
-10			tipo de evento			string de

Já o SDDF (*Self-Defining Data Format*), que foi projetado para o ambiente PABLO [REE 91, *apud* AYD 94], é uma linguagem de descrição de *trace* que permite a descrição tanto da estrutura dos registros quanto de suas instâncias. Esta abordagem, que o caracteriza como um meta-formato, tem a vantagem da flexibilidade e extensibilidade, possibilitando que se tenha um conjunto diversificado de eventos e simplificando a adição de novos eventos.

Entretanto, a desvantagem deste método recai no mesmo tipo de desvantagem da abordagem "meta" em geral. Esta é a de se ter que **aprender** uma nova linguagem e **programar** neste novo método. Isto requer disponibilidade por parte dos usuários para uma tarefa que, muitas vezes, é considerada supérflua por eles (apesar de **não** o ser) que é a análise de desempenho.

Um outro aspecto que deve ser levado em conta no projeto de um formato de *trace* é sua codificação. A quantidade de eventos gerada durante a monitoração pode tornar-se proibitivamente grande, o que faz com que seja necessária a compactação destes dados obtidos. Entretanto, esta compactação pode tornar complexa a tarefa de interpretação dos dados e diminuir o grau de portabilidade do *trace*. Duas abordagens se fazem presentes neste sentido: representação textual e representação binária.

trace. Duas abordagens se fazem presentes neste sentido: representação textual e representação binária.

A representação textual consiste na seqüência de caracteres ASCII que são legíveis ao ser humano e altamente portáveis. Entretanto podem ser um problema na medida em que cresce a quantidade de informação que deverá ser armazenada. Utilizam este tipo de codificação todos os exemplos abordados acima, nesta seção.

Já a representação binária atende o problema da compactação, mas recai no da portabilidade, já que pode diferir de máquina para máquina. O SDDF também implementa esta abordagem, possibilitando a utilização de qualquer uma das duas, o que vai depender do problema em questão. Além disso, possui programa de conversão de uma codificação para outra.

Todos os formatos aqui apresentados foram considerados durante a escolha do formato alvo de estudo para este trabalho. Como já mencionado, o TAPE/PVM foi o escolhido e será melhor detalhado adiante.

3 Características das ferramentas de depuração e análise

Através da observação de diversas características das ferramentas de análise de programas paralelos que estão em evidência no momento, apresentamos uma classificação de seus diferentes aspectos. O objetivo é demonstrar as opções existentes e situar o sistema desenvolvido neste contexto.

Em primeiro lugar, verificou-se que as ferramentas de auxílio à programação dividem-se em dois grandes grupos em relação ao objetivo a que se propõem: estão voltadas à depuração ou à análise de desempenho dos programas paralelos.

Uma outra característica muito importante diz respeito ao tipo de execução da ferramenta, que é o segundo aspecto apresentado. Este descreve o nível de compartilhamento de recursos entre a ferramenta e a aplicação, especificando se a ferramenta vai ser executada total ou parcialmente em tempo de execução da aplicação.

O terceiro aspecto apresenta as duas formas básicas de apresentação dos resultados: textual e gráfica. Cabe salientar que a presença de um não anula a do outro.

Já o quarto e último aspecto analisado classifica o tipo de interação entre usuário e ferramenta, descrevendo seu modo de utilização. Um resumo das características analisadas pode ser encontrado na tabela 3.1 e uma descrição mais detalhada nas seções subseqüentes.

Com o surgimento cada vez maior de novas alternativas ainda não se chegou a um consenso de qual seria o melhor tipo de ferramenta nem qual a melhor abordagem, ou seja, não existe um padrão absoluto a ser seguido. Existe sim uma tendência à apresentação de vários aspectos de comportamento da aplicação ao mesmo tempo, através da utilização de múltiplas janelas. Desta forma, a comunhão entre as características apresentadas passa a ser simplesmente uma definição realizada em tempo de projeto da ferramenta.

TABELA 3.1 - Características das ferramentas de depuração e análise

Classificação	Tipos	Descrição
Quanto ao objetivo.	Correção	Ferramentas voltadas à depuração que utilizam técnicas semelhantes às da depuração de programas seqüenciais.
	Desempenho	Ferramentas com facilidades (geralmente gráficas) para fornecer os parâmetros de desempenho do programa.
Quanto à execução.	On-line	Monitoração e apresentação de resultados em tempo de execução da aplicação.
	Post-mortem	Apenas a monitoração é feita em tempo de execução da aplicação.
Quanto à apresentação dos resultados.	Textual	Resultados na forma de texto; principalmente para demons- tração de código.
	Gráfica	Resultados em janelas gráficas; principalmente para parâmetros de desempenho.
Quanto à utilização.	Meta-tool	Usuário desenvolve sua forma de visualizar o comporta- mento da aplicação.
	Toolkit	Usuário utiliza um conjunto de janelas prontas.

3.1 Classificação Quanto ao Objetivo

Esta característica da ferramenta diz respeito à sua finalidade, ou seja, para quê será utilizada. Se uma ferramenta de depuração foi construída visando auxiliar o programador a encontrar erros de programação em sua aplicação, então a ferramenta é voltada à correção do programa. Caso ela tenha sido construída visando auxiliar o programador a aumentar a eficiência da aplicação, então é voltada ao desempenho.

Uma ferramenta voltada a correção deverá fornecer facilidades que para os programas seqüenciais são extremamente comuns, mas que se tornam bastante complexas em se tratando de programação paralela. Um dos problemas é a apresentação do código do programa fonte. Se na programação seqüencial temos apenas um evento acontecendo de cada vez, na paralela podemos ter uma grande quantidade de processos executando ao mesmo tempo, o que pode dificultar bastante na hora de visualizar de forma textual o ponto do programa em que cada processo se encontra.

Além disso, uma ferramenta deste tipo deve fornecer uma visão geral do sistema, para uma melhor compreensão deste como um todo. Esta visão geral também pode auxiliar na detecção de um erro muito comum em programas paralelos, que é o *deadlock*. Outra facilidade importante e complexa é a possibilidade de inclusão de pontos de parada e de execução passo-a-passo. Maiores detalhes sobre este tipo de ferramenta podem ser encontrados em [IBÁ 95], [LEB 87], [LEU 93], [SAL 92], [STO 88], [MCD 89], [NET 92] e [STR 95].

Já uma ferramenta voltada à análise e melhoria do desempenho de programas paralelos deve fornecer parâmetros confiáveis que demonstrem seu comportamento sob diferentes aspectos. Neste sentido, diversas características da

aplicação em questão devem ser observadas durante sua execução e apresentadas de uma maneira o mais clara possível ao programador. A maneira mais natural de se fazer isso e que tem sido amplamente utilizada é a apresentação das informações através de diversas **janelas de visualização**. Cada uma destas janelas se preocupam, então, em demonstrar uma característica ou parâmetro diferente para a análise de desempenho. Facilidades como pontos de parada e execução passo-a-passo também são bastante utilizadas e necessárias neste tipo de ferramenta.

O presente trabalho visa o desenvolvimento e, principalmente, a experimentação de novas janelas de visualização com o objetivo de atender à segunda abordagem, que é a da análise de desempenho. Entretanto, algumas das janelas idealizadas podem ser aplicadas no desenvolvimento de ferramentas voltadas à correção, já que visam a demonstração do sistema como um todo. Maiores informações sobre esta abordagem podem ser encontradas em [GEI 96], [HEA 91], [HOL 91], [KAR 94], [MIL 94], [PER 95], [STA 96], [SAL 92] e [TOP 95].

3.2 Classificação Quanto ao Modo de Execução

Esta característica diz respeito à execução da ferramenta como um todo: se ela se dará em conjunto com a execução da aplicação ou após. Quando a ferramenta executa conjuntamente com a aplicação se diz que ela é do tipo *on-line*. Caso contrário, se apenas uma pequena parte executa ao mesmo tempo, ela é dita *post-mortem*.

O projeto de uma ferramenta *on-line* deve levar em conta um fator de risco que, principalmente para este tipo de ferramenta, pode ser fatal em relação à usabilidade da mesma. Este fator de risco é o **grau de intromissão** na execução da aplicação, causado pelo compartilhamento de recursos. Esta intromissão pode causar

uma distorção no funcionamento real da aplicação, invalidando, assim, a utilização da ferramenta para qualquer que seja o seu fim.

Já as ferramentas do tipo *post-mortem* são construídas justamente com o objetivo de diminuir ao máximo este grau de intromissão. Nelas, somente o que for indispensável para a observação e coleta de informações sobre o comportamento da aplicação executará juntamente com ela.

Neste caso, também são válidas as ferramentas híbridas, onde as duas abordagens podem coexistir. Assim, o usuário poderá optar por uma das duas, escolhendo a que mais lhe convir. Entretanto, aqui cabe uma ressalva às ferramentas *on-line* que, durante sua execução em conjunto com a aplicação, gravam um histórico de execução que pode depois ser usado de maneira *post-mortem*. Desta forma, as informações que serão demonstradas terão sofrido a mesma intromissão que as *on-line*.

Em relação ao objetivo da ferramenta (correção ou desempenho), qualquer combinação relacionada à sua forma de execução é possível e é objeto de estudo. As ferramentas voltadas à correção podem ser *on-line* ou *post-mortem*. O primeiro tipo é mais comum, principalmente se for adotada a metodologia para depuração de programas seqüenciais, como em [ADA 86]. Entretanto, volta-se a salientar o problema da intromissão, que pode fazer com que a aplicação só apresente erro durante a depuração, ou vice-versa. Da mesma forma, ferramentas voltadas à correção também podem executar de forma *post-mortem*, através da **reexecução** da aplicação baseada no que foi observado durante sua execução original. Para isto, devem ser levados em conta principalmente as características de causalidade e precedência de eventos dos programas paralelos. Isto têm sido objeto de estudo desde o célebre artigo de Lamport [LAM 78], que é a base de muitos estudos desenvolvidos a este respeito [LEB 87], [HEL 91], [NET 92].

Da mesma forma, ferramentas voltadas à análise de desempenho podem vir a ser *on-line* ou *post-mortem*. Neste caso, as duas formas são bastante comuns, devendo-se levar em conta as características destes dois tipos, já descritas acima. O presente trabalho se inclui no escopo das ferramentas do tipo *post-mortem*, como será melhor detalhado adiante.

3.3 Classificação Quanto à Apresentação dos Resultados

Esta divisão diz respeito à forma em que as informações são apresentadas ao usuário, que pode ser textual ou gráfica. Evidentemente que uma ferramenta com várias janelas de visualização pode apresentar as duas formas de apresentação.

A forma textual é comumente utilizada para demonstrar o código dos programas em ferramentas voltadas à depuração. Também é utilizada em ferramentas de avaliação de desempenho com o objetivo de demonstrar no código o evento responsável por determinado acontecimento.

Da mesma forma, a representação gráfica pode ser utilizada em ambos os tipos de ferramentas. Entretanto, é mais amplamente utilizada em ferramentas de avaliação de desempenho, pois, em geral, possuem um maior número de janelas de visualização e de diferentes aspectos de comportamento da aplicação a serem demonstrados. A grande maioria destas janelas representa graficamente alguma característica da aplicação.

Neste trabalho, serão sugeridas novas representações gráficas para algumas características de comportamento de programas paralelos. Algumas estão baseadas em janelas existentes e até já amplamente utilizadas e outras possuem características originais.

3.4 Classificação Quanto ao Modo de Utilização

Esta divisão, apresentada inicialmente em [CAS 92], diz respeito à utilização da ferramenta. Define se o usuário terá ou não participação na construção do tipo de visualização oferecido. No primeiro caso, a ferramenta será denominada *meta-tool* e no segundo, *toolkit*.

Se a ferramenta for uma *meta-tool*, o usuário participará da construção do tipo de visualização ou mais especificamente da animação em si. Nesta abordagem, a ferramenta oferece facilidades que mascaram detalhes de mais baixo nível de implementação, possibilitando a especificação de uma animação a partir de alguma das diferentes técnicas existentes, como por exemplo a de [STA 93]. Para utilizar uma *meta-tool* algum tempo deverá ser gasto pelo usuário para que aprenda a utilizá-la. Em contrapartida, terá uma animação criada por ele próprio, sendo, portanto, de mais fácil compreensão.

Já uma ferramenta do tipo *toolkit* oferece uma grande variedade de opções, através de janelas que produzem animações padronizadas. Neste caso, o usuário não tem nenhuma participação na construção das visualizações. Por um lado, isto facilita na medida em que não precisará aprender a programar a animação, por outro lado, perderá em flexilidade.

Este trabalho tem como objetivo a segunda abordagem, ou seja, a construção de uma coleção de janelas prontas que demonstrem o comportamento de programas paralelos segundo determinadas características. Entretanto, estas janelas estarão de certa forma "personalizadas", já que trarão embutidas algumas informações exclusivamente ligadas à aplicação que estiver sendo analisada.

3.5 Análise de Ferramentas para PVM e MPI

Esta seção tem como objetivo traçar um paralelo entre as bibliotecas de comunicação para ambientes heterogêneos mais conhecidas: PVM e MPI. Entretanto, este paralelo será feito considerando apenas o *software* disponível para cada biblioteca em se tratando de ferramentas para avaliação de desempenho, que é o escopo deste trabalho.

3.5.1 Ferramentas para PVM

Por ter sido uma das bibliotecas pioneiras para ambientes heterogêneos, o PVM não contou com um projeto muito preocupado em oferecer grandes facilidades para depuração e avaliação de desempenho. Somente a partir da versão 3.3 é que passou a ser fornecido o XPVM [GEI 96], que é uma *interface* gráfica para monitoração e avaliação de desempenho distribuída juntamente com o PVM. Além disso, foi criada uma *interface* padrão para permitir que depuradores paralelos existentes possam se associar e depurar aplicações PVM.

O XPVM conta com uma instrumentação automática do código do usuário em tempo de compilação. Durante a execução conjunta (*on-line*) entre aplicação e XPVM, os eventos vão sendo gravados em um arquivo de *trace* que pode ser utilizado depois para uma análise *post-mortem*. Salienta-se que a animação nas várias janelas do PVM são feitas *on-line* e que o mesmo arquivo gravado durante esta execução é que é utilizado posteriormente para a análise *post-mortem*, o que significa que este tipo de análise pode conter o mesmo nível de intromissão da outra. Esta desvantagem e mais o fato de que esta ferramenta possui um conjunto padrão fixo de janelas de visualização (*toolkit*) é que motiva a construção não só de ferramentas de monitoração mais isentas quanto de ferramentas de visualização mais completas.

Dois exemplos de ferramentas de monitoração que procuram evitar a intromissão ao máximo possível, deixando a análise completamente à parte da execução da aplicação são o PGPVM [TOP 95] e o TAPE/PVM [MAI 95]. Estas duas ferramentas realizam monitoração por *software*, o que significa que instrumentam automaticamente o programa, que deve ser recompilado e ligado para que o executável instrumentado seja gerado.

O PGPVM é simplesmente uma ferramenta de monitoração cujos arquivos de *trace* gerados devem ser convertidos para o formato PICL para poderem ser analisados pelo ParaGraph. O TAPE/PVM, apesar de também ser uma ferramenta de monitoração e também poder ser convertido para visualização pelo ParaGraph, foi desenvolvido com uma visão mais ampla que previu sua utilização para o desenvolvimento de novas ferramentas. Além da instrumentação e da conversão para o formato PICL, o TAPE/PVM possui uma biblioteca específica para a leitura dos *traces* gerados. Entre as rotinas disponíveis estão a abertura do arquivo e leitura evento-a-evento, além da definição de tipos para cada um.

Além das duas *interfaces* gráficas já citadas para visualização de programas PVM, o XPVM e o ParaGraph (graças às ferramentas específicas também já citadas), pode-se acrescentar ainda o PVaniM 2.0 [TOP 96], dos mesmos autores do PGPVM. Esta ferramenta permite visualização *on-line* e *post-mortem* de aplicações com comunicação intensiva entre os processos, utilizando técnicas de redução de perturbação e possibilitando a análise através de um conjunto de janelas prontas, no mesmo estilo do ParaGraph.

3.5.2 Ferramentas para MPI

Desde a concepção do padrão MPI houve uma preocupação com o fornecimento de facilidades para a construção de ferramentas de avaliação de desempenho. Ao invés do fornecimento de uma ferramenta padrão juntamente com a

biblioteca, foi definida uma *interface* para a construção de ferramentas de monitoração, a *MPI Profile Interface* [KAR 94].

A idéia consiste em utilizar o *linker* para substituir a função MPI, chamada pela aplicação, pela função escrita pelo usuário, que funciona como uma espécie de "capa" para a verdadeira. Assim, o usuário pode escrever rotinas de monitoração que, após realizarem o trabalho necessário, chamam as rotinas MPI verdadeiras.

Entretanto, não é necessário que o usuário escreva este tipo de rotina para todas as chamadas ao MPI. Na verdade, ele escreve somente as que quiser e o *linker* verifica a sua existência ou não. Caso exista uma definição feita pelo usuário, o *linker* a utiliza, caso contrário ele chama a rotina MPI original.

Uma das maneiras encontradas para a implementação desta abordagem foi a duplicação completa da biblioteca MPI, uma delas com o prefixo MPI e a outra com o prefixo PMPI. A figura 2.3 apresenta a resolução da ordem de precedência utilizada pelo *linker*. A primeira rotina da aplicação possui uma versão para monitoração e a segunda não.

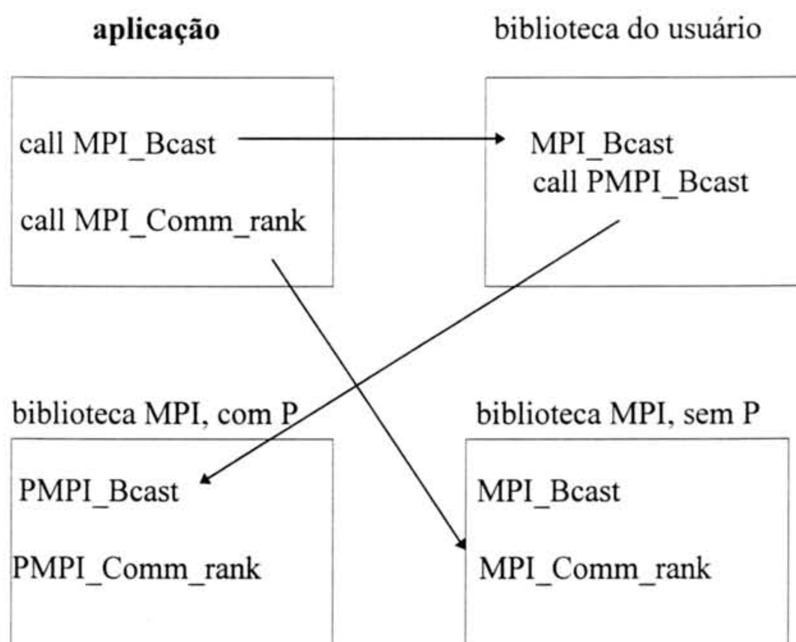


FIGURA 3.1 - Bibliotecas utilizadas pelo *linker* do MPI

Além de oferecer esta abordagem bem flexível, que permite a escrita de rotinas de monitoração por parte do usuário, o MPI já oferece três ferramentas de monitoração e avaliação de desempenho juntamente com sua própria implementação. Duas delas se utilizam ainda da biblioteca MPE (*Multi-Processing Environment*) que provê facilidades para gravação de eventos e gráficos simples.

A primeira é uma biblioteca para o cálculo do tempo gasto em cada rotina MPI. Cada rotina desta biblioteca faz uma chamada à PMPI_Wtime antes e depois de sua própria chamada. O tempo é contabilizado e cada processo mantém a sua cópia dos tempos gastos, que depois podem ser combinados para análise ou não.

A segunda é a criação de um arquivo de *trace* (ou *logfile*) e sua utilização juntamente com o Teeshot [KAR 94], que consiste num conjunto de janelas de visualização que demonstram diversos aspectos de desempenho do programa.

A terceira é a animação em tempo real, possibilitada pela biblioteca MPE, que permite que vários processos possam compartilhar uma mesma janela gráfica. Assim, é possível criar animações do programa com a participação de todos os processos, como é demonstrado em [KAR 94].

A abordagem MPI mostra-se bastante flexível em termos de facilidades de construção de novas ferramentas. Ela permite que se criem tanto ferramentas *on-line* quanto *post-mortem*, textuais ou gráficas, *meta-tools* ou *toolkits*. Apesar desta *interface* ser voltada especialmente à avaliação de desempenho, algumas de suas características podem ser aplicadas à construção de ferramentas para correção de programas.

O ambiente Annai [CLÉ 96] é um exemplo de um ambiente integrado que oferece ferramentas para paralelização, depuração e avaliação de desempenho, todos compartilhando a mesma *interface*. O ambiente é utilizado para programação em HPF (*High Performance Fortran*) e MPI.

As ferramentas do Annai são: PST (*Parallelization Support Tool*), PDT (*Parallel Debugging Tool*) e PMA (*Performance Monitor and Analyzer*). A PST estende a definição corrente da linguagem HPF fornecendo construções e suporte em tempo de execução para a paralelização de computações irregulares [MÜL 95 *apud* CLÉ 96]. A PDT é um depurador convencional de código fonte que pode suportar diferentes níveis de abstração, como os de dados paralelos e mensagens [CLÉ 95 *apud* CLÉ 96]. A PMA, enfim, explora informações colhidas em tempo de execução sobre determinadas regiões do código fonte onde a intrumentação foi previamente inserida de forma interativa [CLÉ 96].

3.5.3 PVM x MPI

Considerando que as bibliotecas MPI e PVM são as mais utilizadas para ambientes heterogêneos, principalmente *clusters* de estações de trabalho, nota-se que pouco já foi feito em termos de ferramentas de visualização. O PVM é o padrão de fato atualmente, por ser mais popular entre programadores de ambientes paralelos. No entanto, nota-se esta carência de ferramentas mais poderosas de auxílio à programação. Mesmo as existentes muitas vezes não são utilizadas, sendo preferidos os *printf's*, que não necessitam de nenhum aprendizado prévio.

O MPI, que por si só já pretende ser um padrão para a programação por troca de mensagens, tem recebido cada vez mais adesões e provavelmente terá uma maior popularidade num futuro muito próximo. Também está carecendo de ferramentas de auxílio à programação, mas só a preocupação que se teve no projeto

em deixar o caminho aberto para novas implementações já fornece boas perspectivas em relação ao futuro.

Entretanto, o próprio fato de o PVM ainda ser a biblioteca mais popular, motivou o desenvolvimento do presente trabalho voltado à sua plataforma. Além disso, há ainda o fato de o objetivo não ser o de construir uma nova ferramenta de monitoração e sim utilizar *traces* gerados por uma já existente. Neste caso, o PVM também se mostrou uma ótima opção, já que trabalhos deste tipo já existem, como mencionado acima. A ferramenta escolhida para tal foi o TAPE/PVM, que será detalhada adiante.

4 Descrição do *Trace File Preprocessor System* - TFPS

O objetivo deste capítulo é fornecer uma descrição do **TFPS**, de *Trace File Preprocessor System*, desenvolvido neste trabalho com o intuito de validar a utilização prévia do arquivo de *trace* para auxiliar na visualização dos programas paralelos. Sua implementação é composta de duas ferramentas: o pré-processador (**TFP**) e o conjunto de janelas de visualização (**TFP View**).

Em primeiro lugar será apresentada uma visão geral do sistema, demonstrando a integração das duas ferramentas mencionadas acima com a ferramenta utilizada para a monitoração dos processos PVM.

Em seguida, será apresentada uma descrição do objetivo final da ferramenta que são as janelas de visualização, concebidas a partir do mecanismo de pré-processamento de *traces* proposto neste trabalho.

4.1 Visão Geral

O **TFPS** é baseado no esquema de análise *post-mortem*, que normalmente é dividido em dois passos: monitoração e animação. A monitoração fornece um histórico da execução do programa, o arquivo de *trace*, para que a ferramenta de animação possa apresentar estes resultados (visualização).

O sistema apresentado em [STR 96] inclui um passo a mais no esquema de análise *post-mortem*, interpondo o pré-processador entre a monitoração e a visualização. O objetivo é captar do arquivo de *trace* as informações que servirão para a montagem das janelas de visualização.

A figura 4.1 ilustra este esquema, onde a monitoração e a correspondente geração do arquivo de *trace* é realizada pela ferramenta TAPE/PVM [MAI 95], o pré-processamento e a geração do arquivo **Info** são realizados pela ferramenta **TFP** e a visualização faz parte da ferramenta **TFP View**.

O TAPE/PVM é um *software* de monitoração para aplicações que se utilizam do PVM para expressar o paralelismo de forma explícita. Constitui-se num conjunto de ferramentas, entre elas um pré-processador para o arquivo fonte que realiza a instrumentação do código. Desta forma, são inseridas as sondas nas chamadas às rotinas PVM que correspondem aos eventos que se pretende monitorar. Estes eventos podem ser selecionados por tipo de evento e por módulos do programa fonte, ou seja, os eventos podem ser selecionados e "filtrados" antes que se inicie a fase de monitoração.

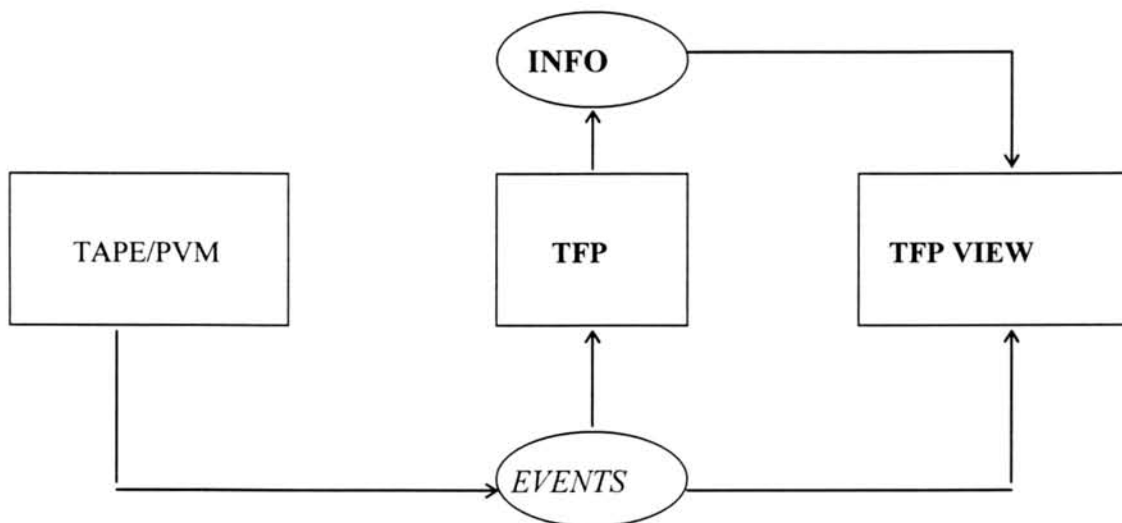


FIGURA 4.1 - Descrição do TFPS

Após ser instrumentado, o programa é compilado e ligado à biblioteca TAPE. Feito isso, pode-se iniciar a execução da aplicação, que será monitorada ao mesmo tempo. Durante a monitoração, os eventos de cada processo vão sendo armazenados localmente e descarregados em arquivos locais, que ao final são automaticamente colocados em um só. O TAPE/PVM utiliza um método não-intrusivo e estatístico para estimar um tempo global para os diversos processos [MAI 95].

Cada evento gravado no arquivo de *trace* contém duas partes. A primeira é o cabeçalho, cujo formato é sempre o mesmo, independente do tipo de evento. A segunda é variável e seu conteúdo dependerá do tipo de evento. Para cada

tipo de evento haverá um número fixo de campos, sendo que o cabeçalho contém sete campos e a parte variável pode conter de zero a seis campos. Os campos do cabeçalho podem ser visualizados na tabela 4.1.

TABELA 4.1 - Cabeçalho de um evento TAPE/PVM

Campo	Significado
alpha	demora, em microssegundos, para gerar o evento
type	tipo do evento
task	identificador do processo PVM
file	identificador do arquivo fonte em PVM que gerou o evento
line	número da linha no arquivo fonte que gerou o evento
date_s	tempo global do evento em segundos
date_us	e em microssegundos

TABELA 4.2 - Exemplo da parte variável dos eventos TAPE/PVM

Nome do evento	Campos
TapeBcastEvent	**ret group delta_s delta_us msgtag bytes**
TapeEnrollEvent[1]	**hi_name hi_arch hi_speed**
TapeExitEvent	
TapeJoinGroupEvent	ret group
TapeLvgroupEvent	ret group
TapeMcastEvent	**ret TaskList delta_s delta_us msgtag bytes ntask**
TapeMytidEvent	ret
TapeRecvEvent	ret tid delta_s delta_us msgtag bytes arrived
TapeSendEvent	**ret tid delta_s delta_us msgtag bytes**
TapeSpawnEvent	ret flag ntask Tasklist

[1]Este é um evento especial que não possui correspondente chamada na biblioteca PVM. Ele é o primeiro evento que ocorre em cada processo e seus campos representam o nome da máquina (*host*), sua arquitetura e velocidade.

O tipo de informação armazenada pelo cabeçalho é de vital importância para a animação e é necessária a todos os eventos. Já a parte variável contém informações relativas ao tipo de evento, portanto, cada evento possível terá seu próprio conjunto de campos. Os eventos possíveis de serem monitorados correspondem a todas as rotinas da biblioteca PVM. A tabela 4.2 acima apresenta um exemplo com alguns desses eventos, salientando os que são analisados pelo pré-processador aqui desenvolvido (em negrito).

Ao final da execução/monitoração do programa do usuário obtêm-se o arquivo *events* gerado pelo TAPE/PVM. Este arquivo mantém agrupados os eventos de um mesmo processo, ou seja, não é feita uma ordenação automática dos eventos por tempo global de execução de cada um, que é dado pelos campos *date_s* e *date_us* do cabeçalho. Entretanto, isto não é considerado como uma desvantagem, já que esta característica favoreceu a construção do algoritmo do pré-processador **TFP**, em particular.

Analisando-se o tipo de informação armazenada chega-se a conlusão de que o conjunto é muito poderoso e que utilizá-lo apenas para a visualização dos resultados pode ser considerado um desperdício. É aí que entra o processador **TFP**, que é o reponsável pela análise destas informações, processando-as de forma que se crie uma espécie de organização. O **TFP** gera o arquivo **Info**, que posteriormente é utilizado pela ferramenta de visualização **TFP View**.

A **TFP View** é um conjunto de quatro janelas de visualização que buscam, em conjunto, oferecer uma visão ampla do programa paralelo e uma forma de identificar alguns problemas de desempenho da aplicação. Ao mesmo tempo, salienta-se que o objetivo não é o de cobrir toda a grande variedade deste tipo problema que pode ocorrer. O objetivo é sim o de experimentar o poder do pré-processamento do arquivo de *trace* na reformulação de janelas já implementadas em outras ferramentas e na criação de novas janelas, possibilitadas justamente por este pré-processamento.

A próxima seção descreve o projeto e as características das janelas de visualização criadas para a **TFP View**. Os próximos capítulos descreverão com mais detalhes a implementação dos protótipos para o **TFP** e para a ferramenta **TFP View**.

4.2 Projeto das Janelas de Visualização

As janelas de visualização, que fazem parte do bloco de visualização (fig. 4.1) foram idealizadas considerando as vantagens trazidas pelo pré-processamento. Assim, algumas foram concebidas com base em outras ferramentas de análise, mas com alterações, e as outras foram construídas exclusivamente a partir de informações coletadas no arquivo de *trace*. A seguir são descritas as janelas implementadas: Diagrama Espaço-tempo, Diagrama Kiviat, Diagrama de Mapeamento de Processos e Grafo de Processos.

4.2.1 Diagrama Espaço-tempo

A primeira janela é bastante encontrada em ferramentas de avaliação de desempenho, mas que neste trabalho sofreu algumas alterações. É o Diagrama Espaço-tempo, que consiste numa representação bidimensional do sistema paralelo através do tempo, onde um eixo representa os processos e o outro, o tempo decorrido. Neste gráfico, cada processo é representado por uma linha horizontal, que demonstra seu estado. A comunicação entre eles é reproduzida por uma linha que liga os dois processos comunicantes. Este tipo de diagrama está presente na maioria das ferramentas de análise, o que comprova sua grande utilidade no sentido de demonstrar o comportamento do programa paralelo, principalmente quanto à interação entre os processos.

As alterações implementadas a partir das informações obtidas com o pré-processador dizem respeito à ordem em que os processos estão dispostos no eixo x. Na maioria das ferramentas esta ordem não tem muita importância e, em geral, nem aparece especificada na documentação. Aqui, a ordem dos processos aparece como uma informação a mais, que antes da animação já pode ser interpretada.

Esta ordenação dos processos consiste em, num primeiro nível, agrupar os processos por *host* e apresentar estes grupos ordenadamente no eixo x. O grupo relativo ao *host* que mais colocou mensagens na rede de comunicação é o primeiro de

cima para baixo. Num segundo nível, os processos são ordenados internamente a cada grupo por quantidade de mensagens enviadas entre os processos componentes da aplicação.

A figura 4.2 mostra um exemplo de diagrama espaço-tempo ordenado. Os processos estão distribuídos em quatro *hosts* que são ordenados por maior número de mensagens colocadas na rede. Dentro de cada grupo os processos também estão ordenados por quantidade de mensagens enviadas.

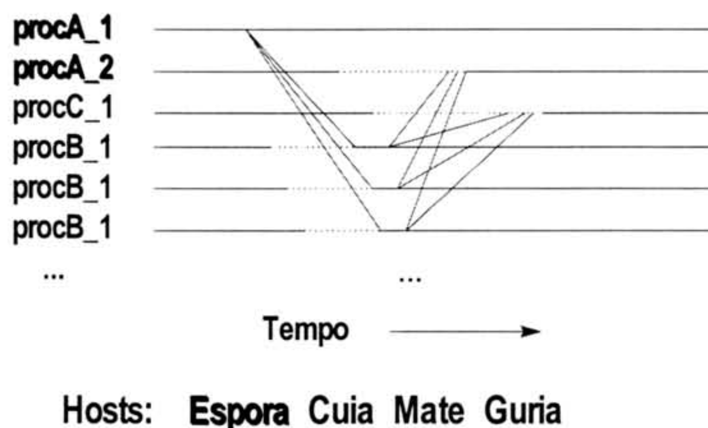


FIGURA 4.2 - Exemplo de Diagrama Espaço-tempo ordenado

A vantagem da ordenação dos processos no eixo x está, principalmente, na inserção de mais uma informação ao programador sobre o comportamento da aplicação sem o acréscimo de poluição visual. Assim, o que ocorre é um reaproveitamento de um espaço da janela com informações que unem aspectos físicos e lógicos da comunicação global da aplicação. Físicos por que apresentam, no nível mais alto, informações sobre os *hosts* que formam a rede utilizada pela aplicação, e lógicos porque ao mesmo tempo demonstram a intercomunicação dos processos que a compõem. Note-se, também, que este tipo de organização permite identificar o mapeamento de processos na rede de *hosts*.

UFRGS
INSTITUTO DE INFORMÁTICA

4.2.2 Diagrama Kiviat

A segunda janela também foi idealizada através de um estudo de ferramentas que a utilizam, como o ParaGraph [HEA 91]. O Diagrama Kiviat possui diversas aplicações e tem sido constantemente utilizado para a avaliação de desempenho de sistemas paralelos. Consiste numa descrição geométrica do sistema, onde a unidade de informação é o processador (ou *host*). Nesta descrição, cada *host* é representado por um raio de um mesmo círculo que, normalmente, representa a utilização do *host* num determinado momento. A união destes pontos determinam os vértices do polígono que descreve um sistema, o que geralmente auxilia no balanceamento de carga.

Entretanto, o Kiviat aqui apresentado **não** demonstra o balanceamento de carga dos *hosts* que compõem a rede e sim o **percentual de mensagens colocadas na rede por *host* em relação ao total da aplicação**. Assim, durante a animação, a figura geométrica apresenta os *hosts* que mais estão utilizando a rede de comunicação.

Além disso, a ordem em que as máquinas estão dispostas no círculo também é previamente processada. Ela corresponde à mesma ordem em que os grupos de processos estão dispostos no Diagrama Espaço-tempo, ou seja, a dos *hosts* que mais enviam mensagens. Assim, o sentido horário indica os *hosts* que mais utilizam a rede, o que acentua o desbalanceamento, se este estiver ocorrendo. A figura 4.3 mostra um exemplo de diagrama Kiviat de utilização da rede, que pode ser utilizado em conjunto com o Diagrama Espaço-tempo apresentado anteriormente.

Esta abordagem para o diagrama Kiviat é vantajosa principalmente em se tratando de aplicações com um número razoável de processos, mas que serão executadas em redes ou arquiteturas com um número menor de processadores. Neste caso, o melhor desempenho é alcançado se os processos que se comunicam com maior freqüência forem mapeados para a mesma máquina, evitando sobrecarga na rede de comunicação. Neste sentido, o diagrama Kiviat aqui apresentado auxilia na detecção dos *hosts* que são os "gargalos" do sistema, ou seja, os responsáveis pela maior carga de comunicação da rede.

O funcionamento do Kiviat durante a animação é de análise do percentual de mensagens enviadas aos outros *hosts* a cada momento, em relação ao total de mensagens colocadas na rede pela aplicação. Assim, a figura tende a crescer e, ao final, mostra o percentual de cada *host* em relação a esse total. O caso de a figura ser homogênea indica uma utilização também homogênea da rede de comunicação pelos *hosts*.

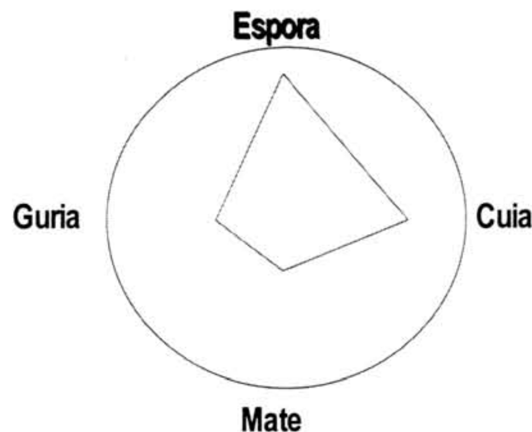


FIGURA 4.3 - Exemplo de Kiviat de utilização da rede

4.2.3 Diagrama de Mapeamento de Processos

O terceiro tipo de janela não é encontrado nas ferramentas mais conhecidas de avaliação de desempenho de programas paralelos. Isto porque a maioria das ferramentas assume uma correspondência fiel entre a arquitetura paralela e o programa paralelo. Entretanto, a maioria das ferramentas de programação, como o PVM, são independentes da arquitetura e não obrigam um compromisso fiel entre os aspectos físicos e lógicos do programa.

Este diagrama leva em consideração esta observação e fornece o **mapeamento de processos** nos diversos *hosts* que compõem a **máquina virtual** do PVM, considerando, inclusive, a hipótese de se ter mais de um processo por *host*.

O diagrama é composto por conjuntos de processos agrupados por *host* que, por sua vez, estão dispostos num grande círculo. Cada processo é um nodo cuja cor representa seu estado durante a animação (**enviando, recebendo, ocupado, inativo**). Esta representação dos estados do processo é bastante difundida em ferramentas deste tipo. A figura 4.4 apresenta um exemplo do Diagrama de Mapeamento de Processos que pode ser utilizado em conjunto com os diagramas anteriormente apresentados.

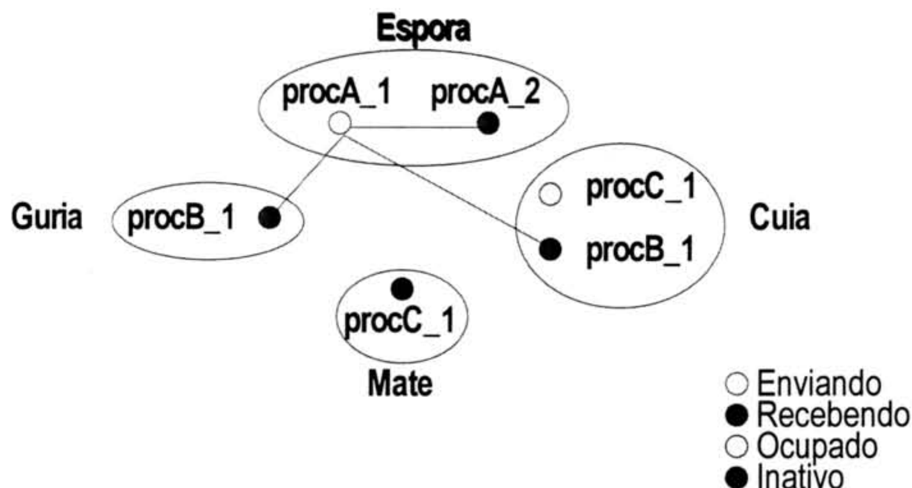


FIGURA 4.4 - Exemplo do Diagrama de Mapeamento de Processos

Este diagrama mostra de uma maneira direta e clara a máquina virtual montada com o PVM e a localização de cada processo durante a execução da aplicação. A animação demonstra a interação e o estado de cada processo a cada momento, o que é comum nas ferramentas de análise de desempenho. Entretanto, a maioria das ferramentas faz esta demonstração através dos *hosts* (fisicamente) e não dos processos (logicamente).

4.2.4 Grafo de Processos

Por último, foi projetada a janela Grafo de Processos que tem como objetivo principal levar ao extremo a representação lógica dos processos que

compõem a aplicação paralela. Assim, o grafo é desprovido de qualquer informação que ligue os processos aos *hosts* em que foram executados.

Segundo [NOR 93], existem duas classes de modelos computacionais que incorporam a comunicação entre processos de diferentes maneiras e podem ser representados através de grafos: os modelos baseados em **tarefas** e os baseados em **processos**. Os baseados em tarefas são geralmente utilizados por pesquisadores interessados em problemas de escalonamento e tendem a ter base em multiprocessadores. Já os baseados em processos são geralmente utilizados para mapeamento explícito de programas paralelos e tendem a ter base em sistemas distribuídos.

Os modelos baseados em **tarefas** consistem em arranjá-las em grafos acíclicos orientados. Neste modelo, um arco entre um par de tarefas corresponde tanto a uma relação de precedência quanto a um evento de comunicação associado a ele.

Os modelos baseados em **processos** consistem em arranjá-los em grafos não dirigidos. Um arco, neste modelo, corresponde ao volume de comunicação entre os processos, além do próprio evento de comunicação associado.

Visto que a representação da relação de precedência não é tão essencial na representação de sistemas distribuídos quanto o é em sistemas de memória compartilhada, optou-se, neste trabalho, pela representação dos programas paralelos através de grafos não dirigidos.

[SIL 94] apresenta um trabalho de comparação e implementação de diferentes tipos de grafos, através de diversos algoritmos encontrados na literatura para tal. Entre os algoritmos para desenho de grafos não-orientados, foram estudados o *Simulated Annealing* e o *Random Search*. Segundo a autora, apesar de ambos os algoritmos gerarem figuras satisfatórias, não se mostraram adequados a ferramentas interativas. Esse problema decorre do fato de consumirem muito tempo na geração das figuras e necessitarem de um grande número de parâmetros, que devem ser iniciados a cada execução.

Justamente este problema motivou os autores de [FRU 91], que salientam que seus objetivos com o algoritmo *Force-directed Placement* foram justamente a velocidade e a simplicidade, o que por sua vez motivou sua adoção neste trabalho. Os critérios de estética adotados pelo algoritmo são os seguintes:

- geração de um grafo não-orientado;
- distribuição uniforme de vértices na janela de visualização;
- tamanho uniforme de arcos e
- rapidez e simplicidade na implementação em relação a outros algoritmos com o mesmo propósito.

Este algoritmo, então, é utilizado para gerar um grafo onde, da mesma forma que no Diagrama de Mapeamento de Processos, cada vértice corresponde a um processo e os arcos à sua comunicação. Além disso, o estado de cada processo, durante a animação, também é indicado por sua cor, obedecendo à mesma convenção do diagrama anterior.

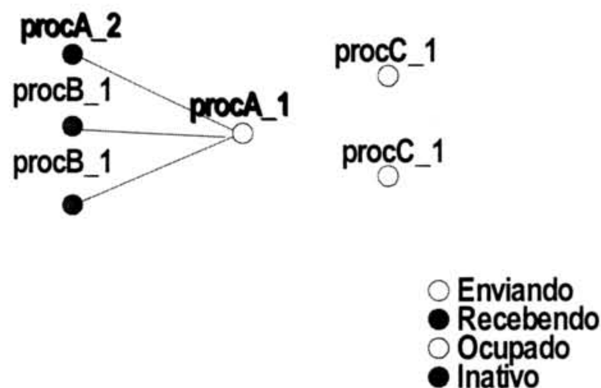


FIGURA 4.5 - Exemplo do Grafo de Processos

Através dele, pretende-se obter uma figura que represente logicamente o sistema, independentemente da configuração da rede ou arquitetura. Assim, cada aplicação terá uma figura correspondente, de acordo com a aproximação dos

processos que mais se comunicam e da distribuição uniforme destes processos na janela de visualização.

Acima é apresentado o que se pretende em termos visuais através desse grafo (figura 4.5). A figura contém os mesmos processos apresentados nas anteriores, mantendo as características de padronização do conjunto de janelas. Entretanto, salienta-se que esta figura é apenas um exemplo do objetivo do grafo e da completa desvinculação dos processos dos *hosts* em que executaram, não tendo qualquer compromisso com o algoritmo de geração do grafo. Sua apresentação aqui é meramente para que se possa idealizar visualmente o conjunto completo de janelas projetadas para a ferramenta. Maiores detalhes serão vistos mais adiante neste texto.

5 Características de Implementação do Pré-processador TFP

Neste capítulo são descritas as estruturas de dados utilizadas pelo **TFP** para o armazenamento das informações lidas do arquivo de *trace*. Além disso, também são apresentados o processo de leitura destes dados necessários e a conseqüente geração das informações que farão parte do arquivo **Info**.

5.1 Estruturas de Dados

As principais estruturas criadas e que armazenam as informações lidas do arquivo de *trace* podem ser visualizadas nas figuras 5.1 e 5.2. A primeira é a **lista de adjacências** que armazena o grafo. A lista principal contém os nós, que são todos os processos que fazem parte da aplicação paralela e a secundária contém suas adjacências, que são os processos para os quais cada um enviou uma ou mais mensagens. A lista principal é um vetor limitado pelo número máximo de processos que o sistema pode conter.

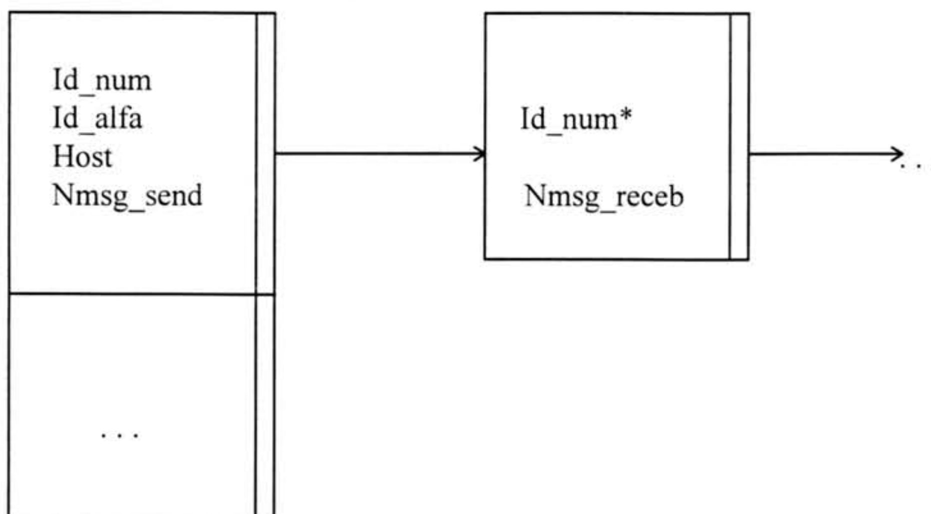


FIGURA 5.1 - Lista de adjacências do TFP

Onde:

- **Id_num**: identificador numérico do processo PVM extraído do *trace*.
- **Id_alfa**: nome alfanumérico do processo. Constitui-se de um nome extraído do nome de seu arquivo fonte e de um número correspondente à sua ordem de criação no *host*. Esta informação não é extraída do *trace*.
- **Host**: nome do *host* da máquina virtual onde o processo executou.
- **Nmsg_send**: total de mensagens enviadas a processos de outros *hosts*. Informação calculada após a leitura do *trace*.
- **Id_num***: identificador numérico do processo PVM que foi destino de uma ou mais mensagens enviadas pelo processo da lista principal.
- **Nmsg_receb**: número de mensagens que o processo destino (Id_num*) recebeu do processo emissor (Id_num).

A segunda é a **lista de *hosts*** que contém o mapeamento dos processos nos *hosts* em que estes executaram. Consiste em uma lista principal e uma secundária associada a cada posição da principal. Esta é também um vetor onde cada posição representa um *host* diferente da máquina virtual montada pelo PVM. Já a secundária contém os processos que nele executaram.

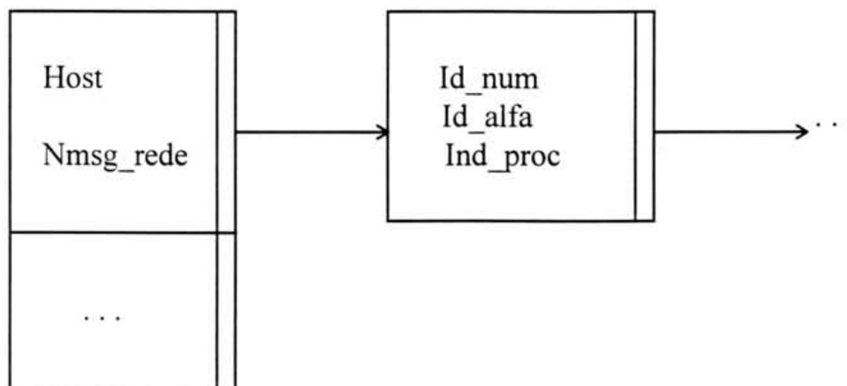


FIGURA 5.2 - Lista de *hosts* do TFP

Onde:

- **Host**: nome do *host* pertencente à máquina virtual. Informação extraída do *trace*.
- **Nmsg_rede**: número de mensagens colocadas na rede pelo *host*. Esta informação é calculada posteriormente à leitura do *trace*.
- **Id_num**: identificador numérico do processo PVM. Informação extraída do *trace*.
- **Id_alfa**: identificador alfanumérico para o processo. Constitui-se de um nome extraído do nome de seu arquivo fonte e de um número correspondente à sua ordem de criação no *host*.
- **Ind_proc**: índice do processo no vetor principal da lista de adjacências. Serve para que posteriormente seja possível acessar as informações do processo nessa estrutura de dados.

5.2 Leitura do arquivo de *Trace*

A leitura das informações é feita em apenas uma passagem pelo arquivo de *trace*. Este passo foi facilitado pelo biblioteca ***tapereader***, fornecida com o TAPE/PVM. O arquivo é lido de forma seqüencial e cada vez que um evento de interesse (ver tabela 4.2) ocorre, as devidas informações são armazenadas nas estruturas de dados correspondentes.

Durante a leitura seqüencial do *trace* os seguintes eventos vão sendo detectados com as respectivas ações sendo tomadas:

- ***enroll***: é o primeiro evento gravado para cada processo da aplicação paralela. Como todos os eventos de um mesmo processo estão agrupados de forma seqüencial no arquivo de *trace* é possível assumir que a partir do *enroll* todos os eventos seguintes corresponderão a um determinado processo. Assim, quando um evento deste tipo é encontrado, as duas estruturas de dados devem

ser atualizadas. A **lista de adjacências** com o novo processo sendo colocado na lista principal e a **lista de *hosts*** com este mesmo processo sendo anexado ao *host* correspondente.

- ***exit***: este evento marca o final dos processos PVM e é gravado no arquivo de *trace* com uma informação importante ao **TFP**. Esta informação, que também está presente em outros eventos, é o identificador do programa fonte no arquivo DB, gerado pelo TAPE/PVM. Este arquivo é criado em tempo de compilação e contém os nomes dos fontes da aplicação com um identificador numérico associado a cada um. A partir desta informação é construído o **Id_alfa**, que receberá ainda um número correspondente à ordem de criação do processo PVM em relação aos outros que executaram no mesmo *host*.

- ***send***, ***mcast*** e ***bcast***: estes são os eventos responsáveis pelo envio de mensagens e, portanto, recebem um tratamento semelhante. Na detecção de qualquer destes eventos a **lista de adjacências** deve ser atualizada, ou, mais especificamente, as próprias adjacências do processo que está sendo analisado. Se o evento for um *send*, o processo destino é incluído nas adjacências. Caso o evento seja um *mcast* ou um *bcast* o TAPE/PVM guarda numa estrutura chamada *TaskList* o nome de todos os processos destinos da mensagem. Esta lista, então, é adicionada às adjacências do processo atual, o emissor. No caso de um ou mais processos já estarem presentes nas adjacências, apenas a variável **Nmsg_receb** é incrementada.

Cabe salientar que, para simplificar o algoritmo e ao mesmo tempo torná-lo mais rápido, os eventos de recepção (*recv*) não são analisados. Assim, todas as mensagens enviadas são contadas como recebidas. Como a princípio esta ferramenta tem como objetivo a análise de desempenho, assume-se que o algoritmo não tenha falhas de funcionamento, o que implica que todas as mensagens enviadas serão efetivamente recebidas por algum processo da aplicação. Entretanto, nada impede que o algoritmo seja alterado neste sentido de maneira fácil e sem que se afete os demais módulos do sistema **TFP**.

5.3 Processamento das Informações

Após a leitura do arquivo de *trace* e armazenamento das informações necessárias tem início a fase de processamento, onde os dados que serão gravados no arquivo de saída do pré-processador são obtidos. As três primeiras janelas, Diagrama Espaço-tempo, Diagrama Kiviat e Mapeamento de Processos, compartilham informações, portanto o esforço em calcular estas informações compartilhadas não é repetido. Já a quarta janela, o Grafo de Processos, é um caso à parte, pois necessita da execução do algoritmo gerador.

Como primeira parte deste processamento tem-se a numeração dos identificadores alfanuméricos de cada processo (**Id_alfa**). Como já descrito, durante a leitura do *trace* o processo apenas recebe um nome, que corresponde ao do seu arquivo fonte. Durante o processamento, um número é concatenado a este nome. Este número, por sua vez, é retirado da ordem de criação do processo na máquina em que executou. Esta ordem de criação corresponde à mesma ordem crescente em que são criados os identificadores numéricos dos processos PVM.

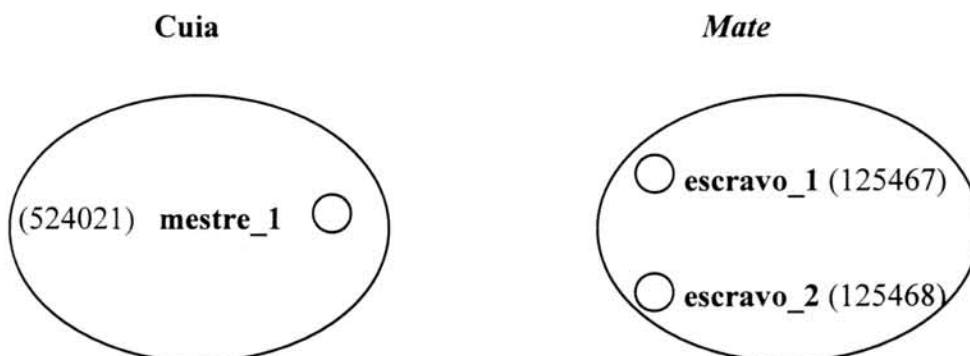


FIGURA 5.3 - Exemplo de criação dos nomes alfanuméricos

A figura 5.3 acima exemplifica este procedimento. No exemplo, existe um processo **mestre** e dois processos **escravos**, cujos arquivos fontes são, respectivamente, **mestre.c** e **escravo.c**. O processo mestre está executando num *host* diferente do que os dois escravos. A figura apresenta um Diagrama de Mapeamento de

Processos com o fictício identificador numérico de cada processo entre parênteses e o correspondente identificador alfanumérico em negrito.

Um outro procedimento importante é o cálculo do número de mensagens que tanto os *hosts* quanto os processos colocaram na rede. Este cálculo é realizado na ordem em que os *hosts* aparecem na **lista de *hosts***. A partir daí, são consultados todos processos que executaram em cada um deles. A informação sobre a quantidade de mensagens é retirada da **lista de adjacências** e corresponde à variável **Nmsg_receb** (ver figura 5.6). Desta forma, as duas estruturas de dados principais são utilizadas em conjunto. O algoritmo é, basicamente, o seguinte:

```
para cada host da lista de hosts  fazer:

    para cada processo do host fazer:

      acessar sua posição na lista de adjacências através
      do seu índice (Ind_proc - ver figura 5.2)

      para cada adjacência fazer:

         verificar se adjacência (receptor) e processo
         principal (emissor) executaram no mesmo host

         se não executaram, contabilizar Nmsg_receb como
         mensagem colocada na rede tanto para o processo
         quanto para o host em que este executou
```

Após executado o algoritmo de cálculo do número de mensagens, tem início a ordenação primeiro dos processos de cada *host* e posteriormente dos próprios *hosts* da máquina virtual.

Os processos que estão ligados a cada *host* da **lista de *hosts*** são ordenados de forma bastante simples. Eles vão sendo colocados ordenadamente numa lista auxiliar ao mesmo tempo em que vão sendo retirados da lista de processos do *host*. A ordem é dada pelo maior número de mensagens colocadas na rede, informação esta retirada da **lista de adjacências**, acessada mais uma vez através do índice do processo nesta lista (**Ind_proc**). Só que o interesse agora é na informação armazenada na variável **Nmsg_send**. No final, a lista ordenada auxiliar é ligada ao *host* a cujos processos são pertencentes.

A ordenação dos *hosts* é feita de maneira diferente, já que pertencem a uma estrutura não encadeada. Na verdade, é criada uma outra estrutura auxiliar e dinâmica que contém índices para a **lista de *hosts***, sendo estes, por sua vez, ordenados a partir da informação que está contida na variável **Nmsg_rede**. Esta, como já mencionado, contém o número de mensagens colocadas na rede pelo *host* em questão.

Assim, pode-se montar uma espécie de tabela onde a primeira coluna contém os nomes dos *hosts* ordenados por número de mensagens colocadas na rede e a segunda, contém o conjunto de processos que executaram em cada um destes *hosts* e que também estão ordenados pelo mesmo tipo de informação que a primeira coluna. Esta "**tabela**" será gravada no arquivo de saída, como será visto a seguir, e suas informações serão utilizadas para a montagem das seguintes janelas de visualização:

- **Diagrama Espaço-tempo**: utiliza a tabela para pegar os nomes dos processos que serão colocados no eixo x. Os nomes são retirados linha a linha, de forma que será mantida a ordem dos *hosts* e a dos processos. Os nomes dos *hosts* também são retirados para que façam parte da legenda colocada na linha inferior da janela.
- **Diagrama Kiviat:** apresenta os *hosts* ordenadamente ao redor do círculo principal. Esta ordem é retirada da primeira coluna desta tabela, que contém os nomes dos *hosts* da máquina virtual ordenados por número de mensagens colocadas na rede.
- **Diagrama de Mapeamento de Processos**: cada conjunto de processos neste diagrama representa um *host* diferente da máquina virtual e corresponde a uma linha desta tabela, que contém o nome do *host* e os processos que nela executaram. A ordem aqui só é importante para os nomes dos *hosts* já que aparecerão na janela na mesma ordem da legenda do Espaço-tempo e do círculo do Kiviat.

Além do processamento já citado, é feito ainda o cálculo do total de mensagens na rede enviadas pelos *hosts*. Isto servirá para que se calcule, em tempo de animação, o percentual de mensagens que cada *host* já colocou na rede em relação ao total (em cada evento do tipo *send*). Este cálculo do total é um simples somatório onde **Nmsg_rede** é acumulado para todos os *hosts* da **lista de *hosts***. Isto é facilitado pelo

fato de todo o processamento para o cálculo do valor da variável **Nmsg_rede** já ter sido feito anteriormente.

5.4 Algoritmo gerador do Grafo de Processos

O protótipo implementado utiliza a ferramenta Nature [FRU 92] para a geração do grafo de processos. Esta, por sua vez, consiste na implementação do algoritmo *Force-directed Placement* de [FRU 91] que faz parte do projeto da ferramenta **TFP View**.

No protótipo, existe apenas a visualização do grafo que corresponde ao programa. Esta visualização é possibilitada através da ferramenta XGraphDrawer, fornecida juntamente com a ferramenta Nature. A visualização será abordada no próximo capítulo.

Entretanto, para que o grafo possa ser criado é necessário que se forneça à ferramenta Nature uma lista de adjacências. Esta lista, já abordada neste capítulo, é construída pelo **TFP** e consiste de processos representando os vértices e da comunicação entre eles representando os arcos.

```
a:q b c d e f g h;
b:q a c h i;
c:q b d i j;
d:q c e j k;
e:q d f k l;
f:q e g l m;
g:q f h m n;
h:q g i n o;
i:q h j o p;
j:q i k p;
k:q j l p;
l:q k m p;
m:q l n p;
n:q m o p;
o:q n p;
p:q o;
q:p o n m l k j i h g f e d c b a;
```

FIGURA 5.4 - Arquivo .g para ferramenta Nature

O protótipo do **TFP** gera um arquivo **.g** com esta lista. Este arquivo é utilizado pela ferramenta Nature para gerar um arquivo **.nat** com as coordenadas para a ferramenta XGraphDrawer. O arquivo **.g** pode ser visualizado na figura 5.4 e o **.nat** na figura 5.5.

```
initcoords1000 1000
wipe
point a 173 -2330
point q 186 -3440
point b 108 -2990
point c 310 -3200
point d 369 -2400
point e 300 -2120
point f 116 -2150
point g -17 -2980
pointh -10 -3930
point i 166 -4470
point j 322 -4500
point k 390 -4040
point l 276 -3720
pointm l 21 -4040
point n 34 -4500
point o 95 -4490
point p 223 -4490
a : h g f e d c b q ;
q : a b c d e f g h i j k l m n o p ;
b : i h c a q ;
c : j i d b q ;
d : k j e c q ;
e : l k f d q ;
f : m l g e q ;
g : n m h f q ;
h : o n i g q ;
i : p o j h q ;
j : p k i q ;
k : p l j q ;
l : p m k q ;
m : p n l q ;
n : p o m q ;
o : p n q ;
p : o q ;
vector 450 450 450 -450
vector 450 -450 -450 -450
vector -450 -450 -450 450
vector -450 450 450 450
```

FIGURA 5.5 - Arquivo .nat para ferramenta XGraphDrawer

5.5 Formato do arquivo de saída

O arquivo de saída **Info** contém todas as informações necessárias para que a ferramenta de visualização, a **TFP View**, possa montar as janelas projetadas. Devido ao compartilhamento de informações, já demonstrado na seção 5.3, o arquivo fica de certa forma "compactado", ou seja, as informações não são repetidas para cada tipo de janela.

As primeiras informações presentes no arquivo **Info** são, respectivamente, o número de processos e o número de *hosts* utilizados durante a execução da aplicação. Estas informações serão úteis para que a ferramenta **TFP View** obtenha acesso imediato à quantidade de processos e de *hosts* que terão que ser demonstrados na tela, de acordo com cada janela.

A seguir, aparecem as informações relativas à máquina virtual e à aplicação que foram processadas pelo **TFP**. A primeira coluna de cada linha corresponde ao nome dos *hosts* em que os processos, que aparecem logo a seguir, executaram. Toda esta primeira coluna aparece já na ordem decrescente em que os *hosts* colocaram mensagens na rede.

Os nomes dos processos aparecem em duas versões: em primeiro lugar os seus identificadores alfanuméricos e, em segundo, seus identificadores numéricos. Os alfanuméricos são os que aparecem na janela de visualização e os numéricos servem para guiar a animação, já que no arquivo de *trace* eles aparecem descritos desta maneira. Os nomes dos processos, em cada linha, também já estão ordenados de forma decrescente por número de mensagens colocadas na rede em relação aos outros processos do mesmo *host*.

A figura 5.6 demonstra um trecho deste arquivo. Para os exemplos tanto do **TFP** quanto do conjunto de janelas da **TFP View**, apresentado no próximo capítulo, foi utilizada a ferramenta *ANDES* [KIT 94]. Trata-se de uma ferramenta para avaliação de programas paralelos onde a técnica utilizada é baseada em programas sintéticos [KIT 95b].

avaliação de programas paralelos onde a técnica utilizada é baseada em programas sintéticos [KIT 95b].

Os testes realizados foram feitos com programas do tipo SPMD (*Single Program Multiple Data*) fazendo com que a ferramenta TAPE/PVM capturasse apenas um nome de processo em execução. Isto fez com que todos os processos ficassem com o mesmo nome alfanumérico, o que para a demonstração aqui pretendida não chega a causar prejuízos.

```
#TFP info
17 5
#Espaco-tempo&Kiviat&Mepeamento de Processos
geronimo andes-synth_2 262157 andes-synth_3 262158 andes-synth_1 262156 andes-synth_4 262159
nsw4 andes-synth_2 1310729 andes-synth_1 1310728 andes-synth_3 1310730 andes-synth_4 1310731
nsw1 andes-synth_1 524297 andes-synth_2 524298 andes-synth_3 524299
nsw2 andes-synth_1 786441 andes-synth_2 786442 andes-synth_3 786443
nsw3 andes-synth_1 1048585 andes-synth_2 1048586 andes-synth_3 1048587
#...
```

FIGURA 5.6 - Trecho de um arquivo Info do TFP

O exemplo acima demonstra uma aplicação constituída por dezessete processos **andes_synth**[2] que executou em cinco *hosts* diferentes. Os *hosts*, em ordem decrescente de mensagens colocadas na rede são os seguintes: **geronimo**, **nsw4**, **nsw1** **nsw2** e **nsw3**. Salienta-se que a relação entre eles é de **maior ou igual** e não simplesmente de "maioridade". Da mesma forma, isto acontece com a relação de ordem existente entre os processos.

[2]O processo **andes-synth** é o núcleo da ferramenta *ANDES* e executa o programa sintético. Na verdade, os processos que executam em conjunto com ele são denominados **andes-worker.**

6 Montagem das Janelas de Visualização

Neste capítulo é descrita a implementação do protótipo feito para a ferramenta **TFP View**. Para agilizar o desenvolvimento deste protótipo foi definido que seria construído como parte do ParaGraph [HEA 91]. Este, permite a construção de novas janelas através de chamadas a rotinas do usuário, que são fornecidas como *stubs*. Após editados, os *stubs*, que agora são rotinas que contém a implementação do usuário para suas próprias janelas, são compilados e ligados ao ParaGraph, que terá botões especiais para a ativação destas janelas (figura 6.1).

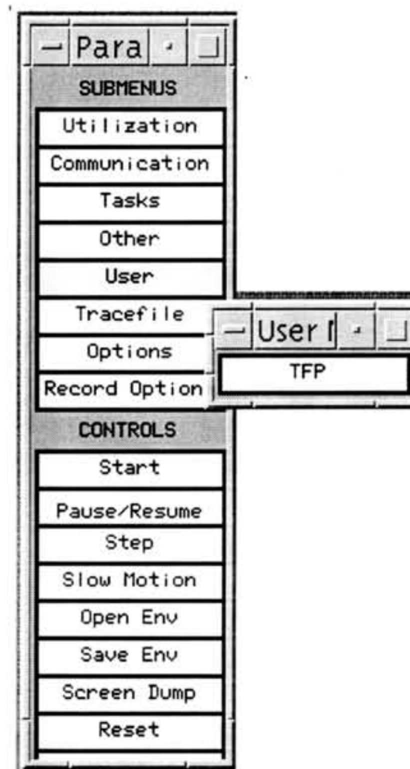


FIGURA 6.1 - ParaGraph com botões para ativação do TFP View

6.1 Janelas do Protótipo TFP View

A montagem das janelas pela ferramenta **TFP View** se dá a partir da leitura do arquivo **Info**, gerado pelo **TFP**. Para a janela Espaço-tempo, aqui chamada

Space-time Diagram para compatibilizar com a língua de origem do ParaGraph, são retirados deste arquivo os nomes dos *hosts* presentes na legenda e os nomes dos processos presentes no eixo x do gráfico (figura 6.2). A ordem dos processos no eixo x, como definida no capítulo 4, também é retirada do arquivo **Info**.

Como já mencionado, para os exemplos do conjunto de janelas da **TFP View** abaixo foi utilizada a ferramenta *ANDES* [KIT 94], que é uma ferramenta para avaliação de programas paralelos onde a técnica utilizada é baseada em programas sintéticos [KIT 95b]. Os exemplos a seguir partiram da mesma execução que gerou o arquivo **Info** do capítulo anterior.

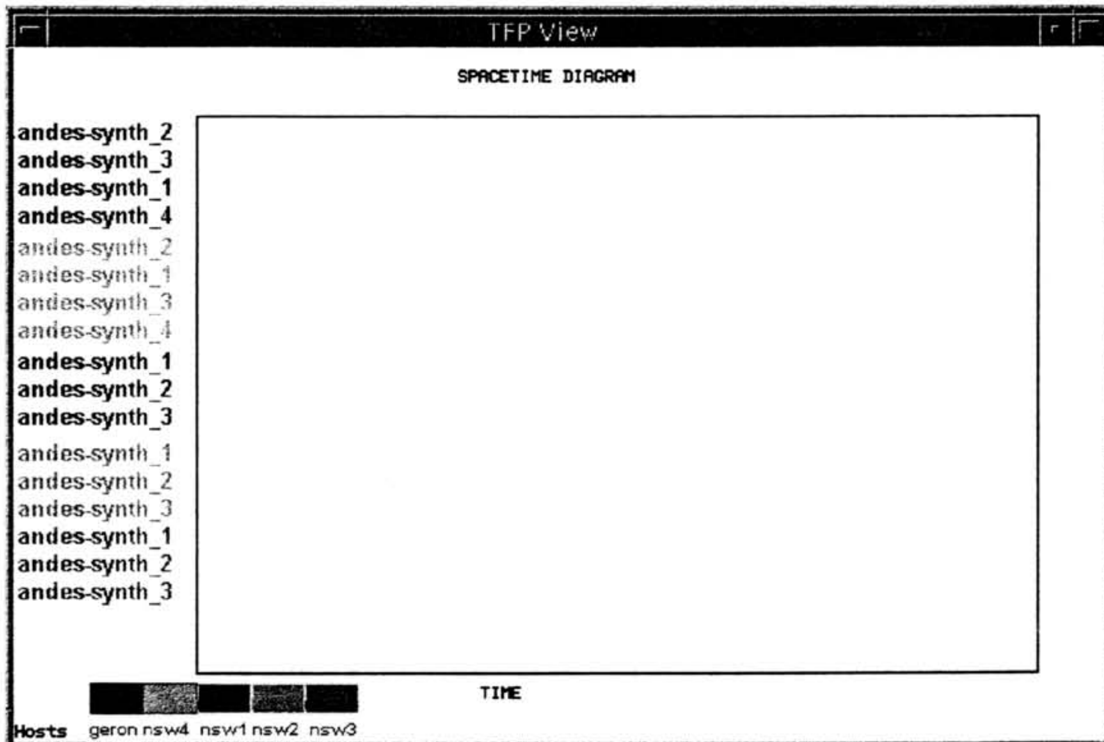


FIGURA 6.2 - Space-time Diagram do TFP View

A janela do Diagrama Kiviat foi renomeada para *Kiviat Diagram* no protótipo. Para esta janela, são utilizados, do arquivo **Info**, os nomes dos *hosts*, que já foram gravados em ordem decrescente de número de mensagens colocadas na rede.

Esta informação já foi retirada do arquivo **Info** para uso na janela anterior, portanto não há necessidade de nova leitura para este fim.

Para a animação deste diagrama é acessada a informação do número total de mensagens colocadas na rede por todos os *hosts*, que também está presente neste arquivo. Esta informação, como já descrito, servirá para que se calcule o percentual de cada *host* em relação ao total. A figura 6.3 demonstra o estado inicial do *Kiviat Diagram* do protótipo implementado.

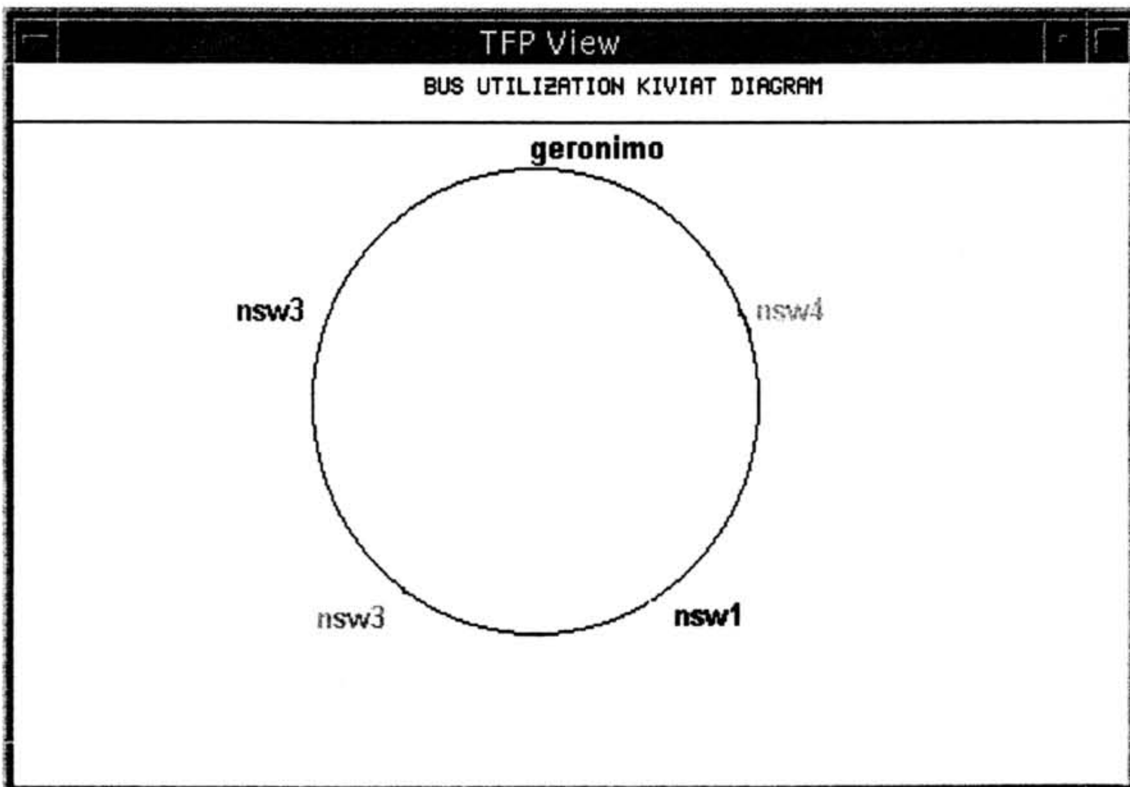


FIGURA 6.3 - Kiviat Diagram do TFP View

O Diagrama de Mapeamento de Processos, chamado de *Processes Mapping Diagram* neste protótipo, não necessita de nova leitura do arquivo **Info**. As informações necessárias à sua montagem correspondem àquelas utilizadas na *Space-time*, que já foram lidas. A figura 6.4 ilustra a montagem inicial deste diagrama.

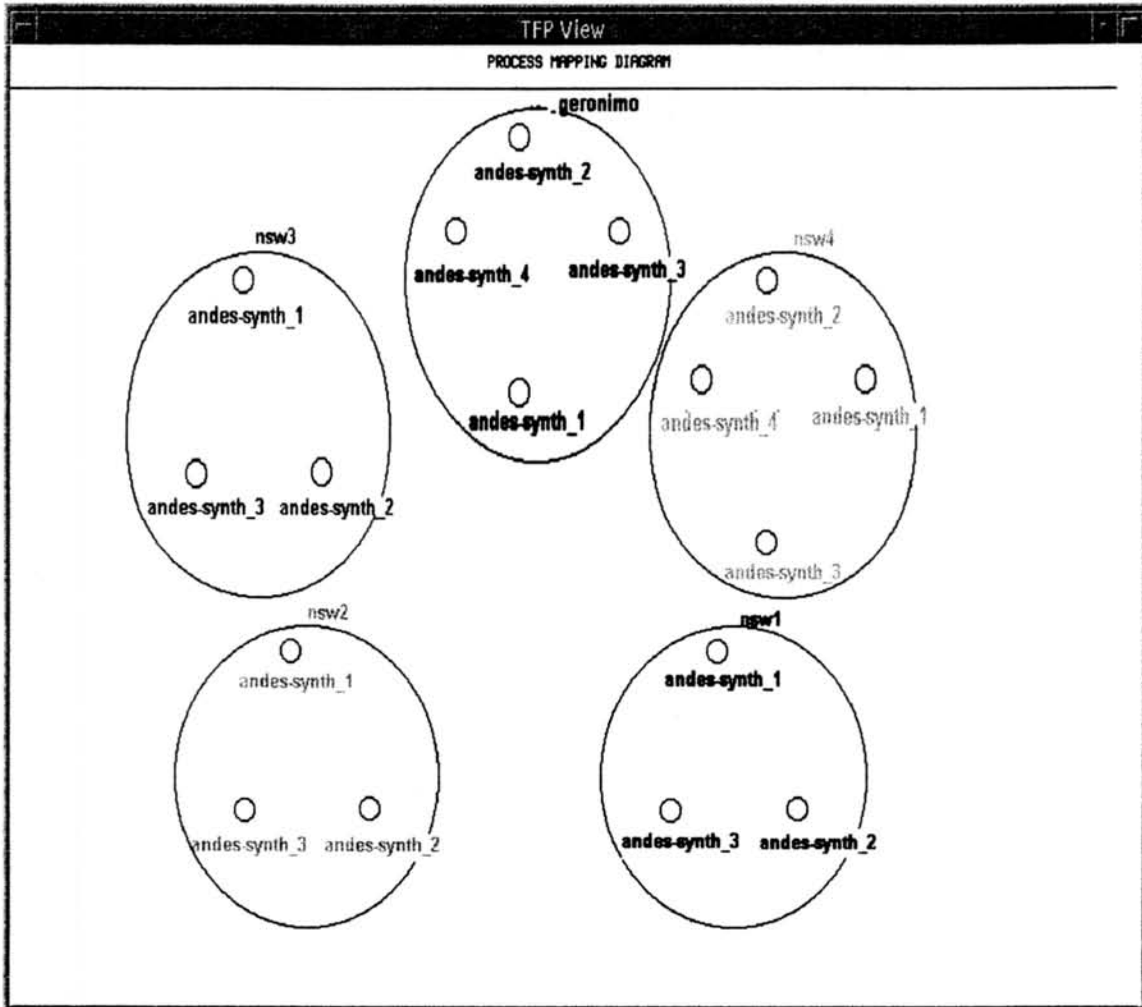


FIGURA 6.4 - Processes Mapping Diagram do TFP View

Já o Grafo de Processos, como mencionado anteriormente, não faz parte do protótipo da ferramenta **TFP View**. Para sua validação, foi utilizada a ferramenta **Nature** [FRU 92] que possibilita que um grafo seja gerado a partir de uma lista de adjacências. O **TFP** gera esta lista num arquivo **.g** que é a entrada da ferramenta Nature. Ela então gera o grafo através de suas coordenadas numa janela de tamanho que pode ser definido pelo usuário. Estas coordenadas, por sua vez, são gravadas num arquivo **.nat** e tanto podem ser visualizadas pela ferramenta **XGraphDrawer** (figura 6.5), quanto podem ser convertidas para o formato *PostScript* pela ferramenta **nat2ps**, ambas fornecidas juntamente com o pacote.

Infelizmente, os nomes dos processos não podem ser incluídos na figura gerada pela Nature/XGraphDrawer, mas apenas algumas letras para representar cada vértice. Desta forma, foi acrescentada ao arquivo **Info** uma legenda que identifica cada processo no grafo por sua letra correspondente (figura 6.6).

Obviamente que esta legenda deixa a visualização um pouco mais complexa, por isso torna-se necessário salientar que ela existe única e exclusivamente devido ao uso da ferramenta Nature, devendo ser completamente abolida no momento da implementação não do protótipo, mas da própria TFP View.

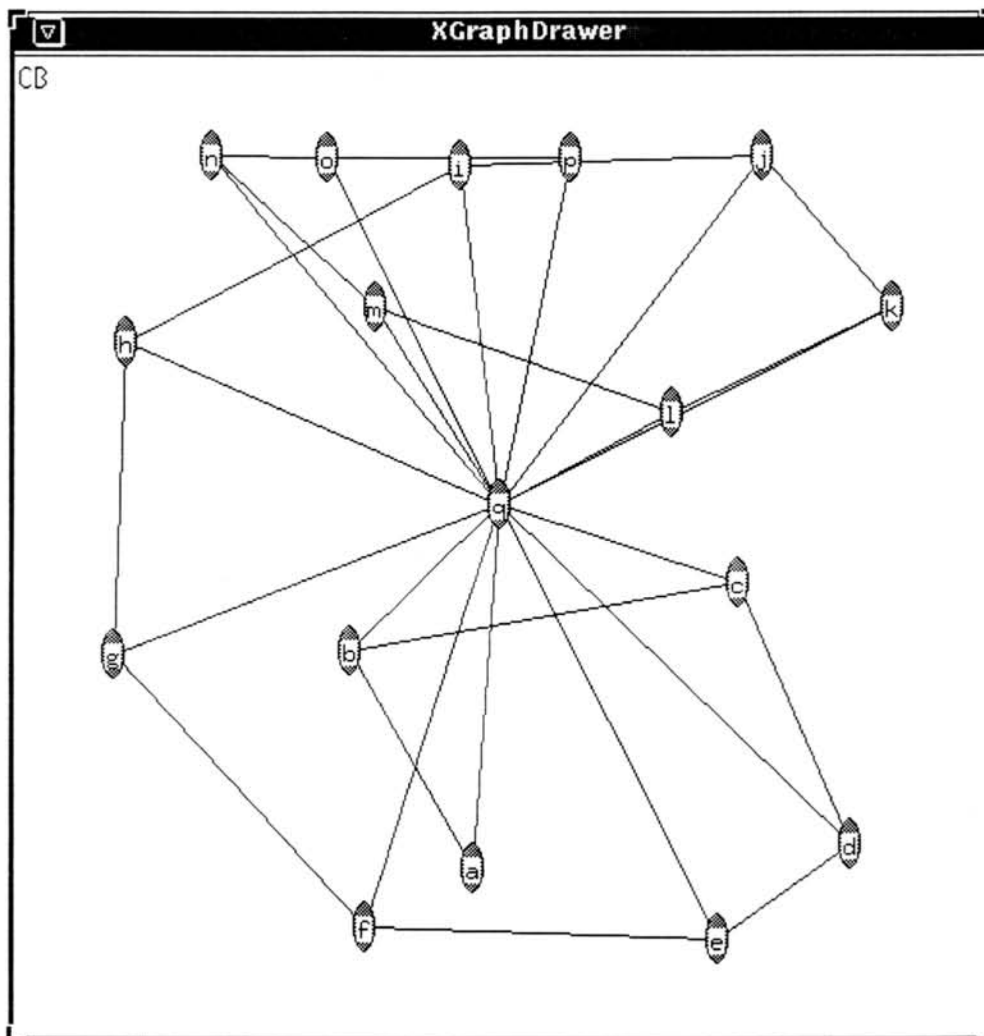


FIGURA 6.5 - Grafo gerado pela ferramenta Nature

```
#Grafo de Processos (legenda)
a andes-synth_4 nsw4
b andes-synth_3 nsw3
c andes-synth_3 nsw2
d andes-synth_3 nsw1
e andes-synth_4 geronimo
f andes-synth_3 nsw4
g andes-synth_2 nsw3
h andes-synth_2 nsw2
i andes-synth_2 nsw1
j andes-synth_3 geronimo
k andes-synth_2 nsw4
l andes-synth_1 nsw3
m andes-synth_1 nsw2
n andes-synth_1 nsw1
o andes-synth_2 geronimo
p andes-synth_1 nsw4
q andes-synth_1 geronimo
```

FIGURA 6.6 - Legenda para o Grafo de Processos

A figura 6.5 apresenta um grafo gerado a partir de um algoritmo sistólico denominado **diamante** executado com a ferramenta *ANDES*. O objetivo aqui é o de demonstrar de uma maneira geral o programa em análise. Um destaque especial é dado ao processo **q**. Pode-se observar que foi o primeiro a ser criado na máquina **gerônimo** e que possui comunicação com todos os outros, principalmente pela sua disposição centralizada na janela de visualização. Pode-se concluir que este é o processo principal, ou **andes-synth**, enquanto os outros são os "trabalhadores" , ou **andes-worker**. Num programa não SPMD, isto se tornaria mais claro, pois os processos teriam seus nomes diferenciados.

6.2 Comparação com outras ferramentas

Num primeiro momento esta comparação será feita considerando-se o ParaGraph [HEA 91], pois foi a ferramenta tomada como base para este trabalho.

Salienta-se que no momento da concepção deste trabalho o ParaGraph, ou PG, era uma das poucas opções existentes em termos de ferramenta de visualização de programas paralelos. Desde lá, muito trabalho tem sido realizado, trabalho esse que foi acompanhado durante esta pesquisa, o que pode ser comprovado ao longo deste texto.

Justifica-se a utilização do PG como alvo de uma comparação mais direta por três motivos. O primeiro, já mencionado acima, foi o estudo feito sobre sua estrutura, no início do trabalho, que serviu como base para o mesmo. Daí também decorre o segundo motivo, a possibilidade de se comprovar a evolução alcançada com o desenvolvimento do esquema de visualização realizado neste trabalho. O terceiro, de ordem prática, é a possibilidade de utilização de PVM para a demonstração.

Cabe salientar que o PG, apesar de ser uma ferramenta pioneira, não é, de forma alguma, obsoleta. Ainda é utilizado e bastante citado na literatura atual sobre o assunto. Além de ter sido base para este trabalho, muitos outros pesquisadores também o utilizaram como tal.

Após a comparação com o PVM também serão abordadas algumas outras ferramentas, principalmente no que diz respeito aos tipos de janelas de visualização que oferecem. As escolhidas foram a XPVM e a PVaniM.

6.2.1 TFP View x ParaGraph

A primeira comparação será feita considerando-se a janela *Space-time* de ambas as ferramentas. A da TFP View pode ser visualizada na figura 6.2, enquanto que a do PG corresponde à figura 6.7.

As duas figuras foram obtidas tendo-se executado o mesmo programa paralelo já mencionado. Elas demonstram apenas a janela inicial, e não um *snapshot* da animação, para salientar a principal característica do trabalho aqui desenvolvido, que é a inserção de informações na janela de visualização que podem ser interpretadas à primeira vista.

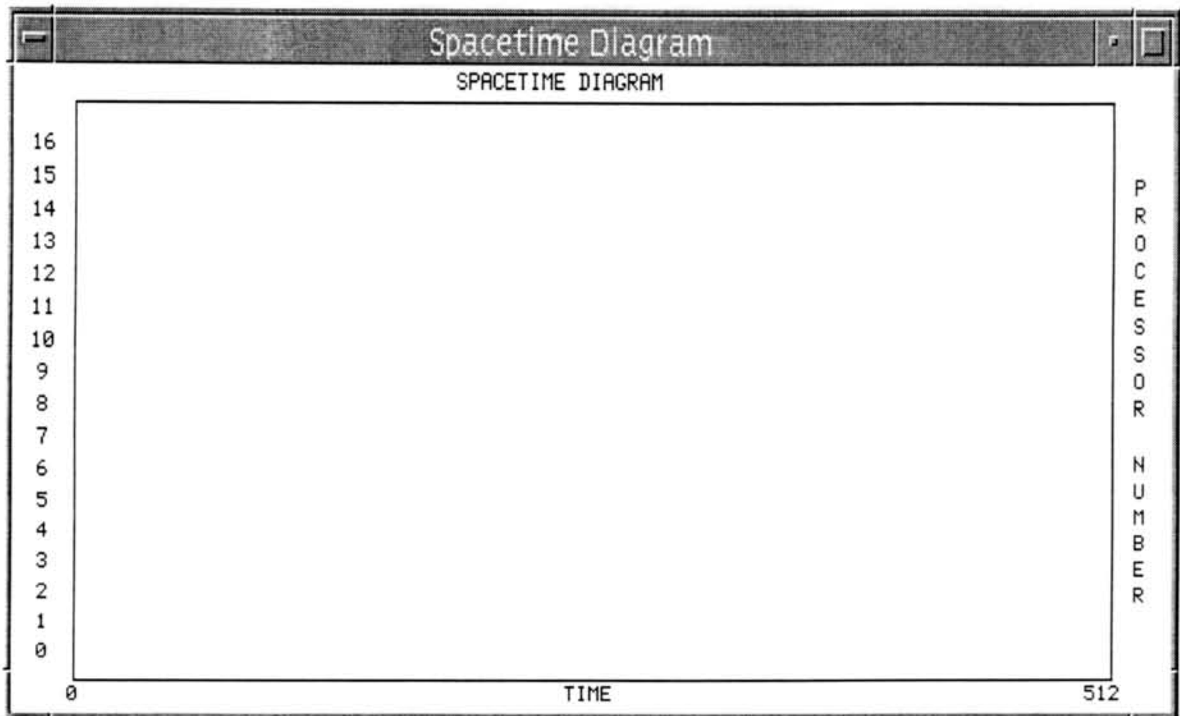


FIGURA 6.7 - Space-time do ParaGraph

Note-se que em ambas temos os processos representados no eixo x e o tempo no eixo y. Entretanto, a janela do PG apresenta os processos numerados, sem nenhuma referência ao programa paralelo em análise. Qualquer programa que tenha dezessete processos (ou processadores, que em verdade é a unidade básica do PG) terá o mesmo diagrama.

Já o gerado pelo TFP possui os nomes dos processos participantes. Estes nomes referenciam os dos processos do programa paralelo e juntamente com sua ordem de criação em cada máquina/processador em que executaram. Além disso, estão ordenados por quantidade de mensagens colocadas na rede de comunicação. Esta ordem se dá tanto dentro de cada máquina quanto entre máquinas. Todas estas informações foram **adicionadas**, em relação ao PG, **sem** o acréscimo de poluição visual.

Salienta-se que sem pré-processamento do arquivo de *trace* seria quase impossível a obtenção destas informações. Ainda que fosse possível obter-se os identificadores alfanuméricos, a ordenação seria totalmente inviável. Ainda assim, a

obtenção dos identificadores poderia causar um aumento da intromissão por parte da ferramenta de monitoração.

Outro detalhe importante diz respeito à identificação das máquinas onde os processos executaram. Esta identificação se dá através das cores dos nomes dos processos com a legenda na parte inferior da janela. Esta identificação não existe no PG, o que é perfeitamente explicável, já que o a única unidade de informação do PG são os processadores e é a conversão do formato TAPE/PVM para o PICL que transforma os processos PVM em processadores no PG.

As janelas Kiviat das ferramentas em análise não demonstram o mesmo parâmetro de desempenho. Enquanto a do PG (figura 6.8) pode demonstrar a utilização e o *overhead* dos processadores (neste caso, dos processos), a da TFP View (figura 6.3) demonstra a utilização da rede de comunicação que interliga os *hosts*. Ambos os parâmetros são importantes numa avaliação de desempenho, portanto este não será o critério de comparação.

A figura 6.3 e a 6.8 não apresentam a mesma unidade de informação. Como já mencionado, a conversão do TAPE/PVM faz com que sejam apresentados sempre os processos no PG e não os processadores. Assim, a primeira figura, que é a do Kiviat do TFP View, apresenta os *hosts* que foram utilizados na máquina virtual do PVM. Já a segunda, que é a do Kiviat do PG, apresenta os **processos** do programa paralelo.

A apresentação das duas janelas nesta seção de comparação vale simplesmente para uma comparação estética, pois enquanto a TFP View mostra os *hosts* com seus nomes e numa ordem pré-determinada, o PG simplesmente os numera sem relacioná-los à execução da aplicação paralela.

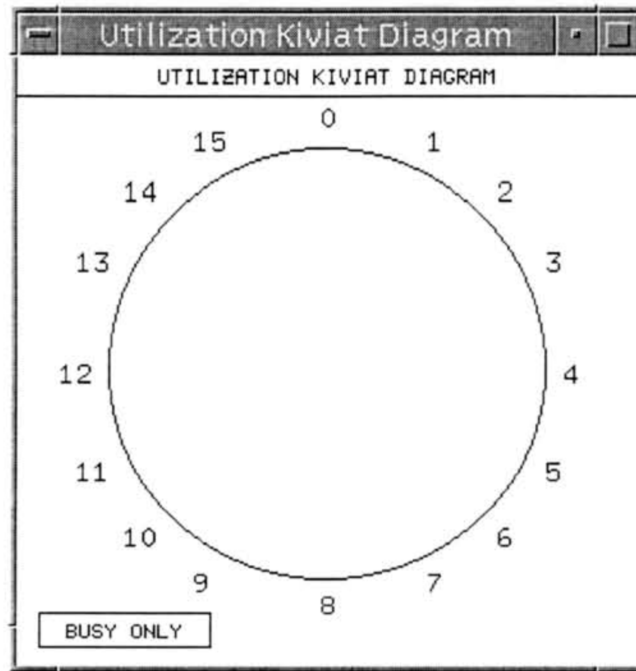


FIGURA 6.8 - Kiviat do ParaGraph

Da mesma forma que na comparação anterior, somente a demonstração da configuração inicial das janelas foi utilizada para evidenciar o trabalho do TFPS.

Já a janela da figura 6.4, a do *Processes Mapping Diagram*, não encontra similar dentre as inúmeras janelas do PG. Isso, é claro, pelo fato de o PG considerar como unidade de informação somente os processadores, sem distinção entre a estrutura lógica e a física que um programa paralelo pode ter. Esta característica de distinção não é considerada pelo PG mas é marcante quando se trabalha com *clusters* de estações de trabalho, onde o número de estações disponíveis pode ser inferior ao número de processos, o que obriga a um balanceamento de carga.

A visualização da máquina virtual do PVM juntamente com a localização dos processos pode ser muito interessante para a própria compreensão do comportamento do programa e vir a auxiliar num eventual balanceamento de carga "manual" que por ventura o programador necessite fazer (o PVM não realiza balanceamento de carga no sentido de fazer uma distribuição "inteligente" dos processos entre os processadores).

Além disso, pode ser de muita ajuda em sistemas de depuração de programas paralelos como uma forma de visualizar a aplicação como um todo. Este tipo de visualização pode servir como o nível de abstração mais alto, a partir dos qual se poderia ter acesso aos inferiores.

O Diagrama Grafo de Processos, apresentado com o auxílio da ferramenta Nature na figura 6.5, foi idealizado a partir da janela Animation do PG, cuja janela inicial na modalidade RING pode ser visualizada na figura 6.9. Esta janela produz uma animação onde a cor de cada processo indica seu estado e linhas entre eles indicam a comunicação.

Ela possui uma outra modalidade, a USER, onde o usuário pode movimentar os processos para qualquer outra posição dentro da janela com a opção de salvar cada nova configuração. A grande questão nesta modalidade é: movimentar para onde?

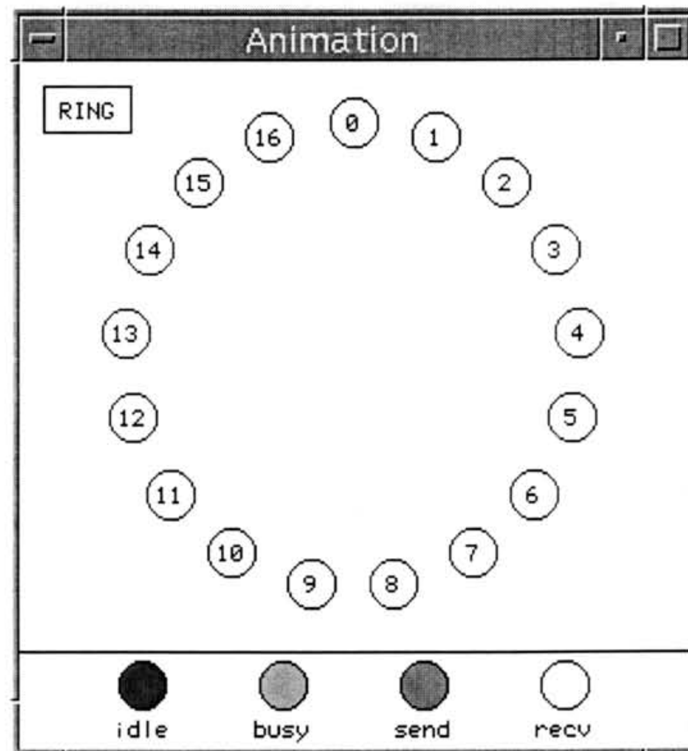


FIGURA 6.9 - Animation do ParaGraph (janela inicial)

Neste sentido foi idealizado o Grafo de Processos que gera automaticamente a configuração mais adequada para cada tipo de programa, de acordo com suas vizinhanças. Estas, no caso, estão relacionadas a quais processos cada um se comunica.

Como a ferramenta Nature gera o grafo com os arcos desenhados, a comparação da janela Animation em suas duas modalidades (figuras 6.10 e 6.11) e a do Grafo de Processos (figura 6.5) também é feita desta forma. As imagens da janela Animation foram capturadas após a animação do arquivo de *trace* da mesma aplicação responsável pela geração do grafo.

A partir desta experiência é fácil perceber que o grafo gerado a partir da lista de adjacências fornecida pelo TFP é de mais fácil visualização, permitindo uma identificação imediata dos processos que comunicam entre si.

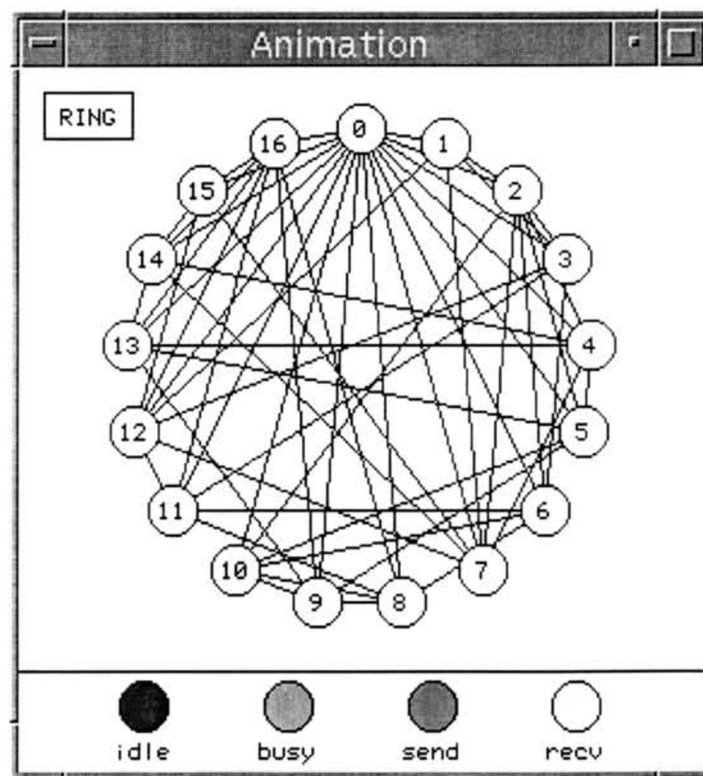


FIGURA 6.10 - Animation do ParaGraph (RING)

Na janela Animation do tipo USER, demonstrada na figura 6.11, alguns processos foram reposicionados aleatoriamente, apenas para efeito ilustrativo. A tarefa de organizá-los manualmente requer um conhecimento prévio da aplicação ou algum tempo de análise da Animation-RING para que se identifique as ligações entre os processadores. Ainda assim, seria necessário uma observação mais apurada e em tempo de animação para a identificação dos processos que têm comunicação mais freqüente entre si.

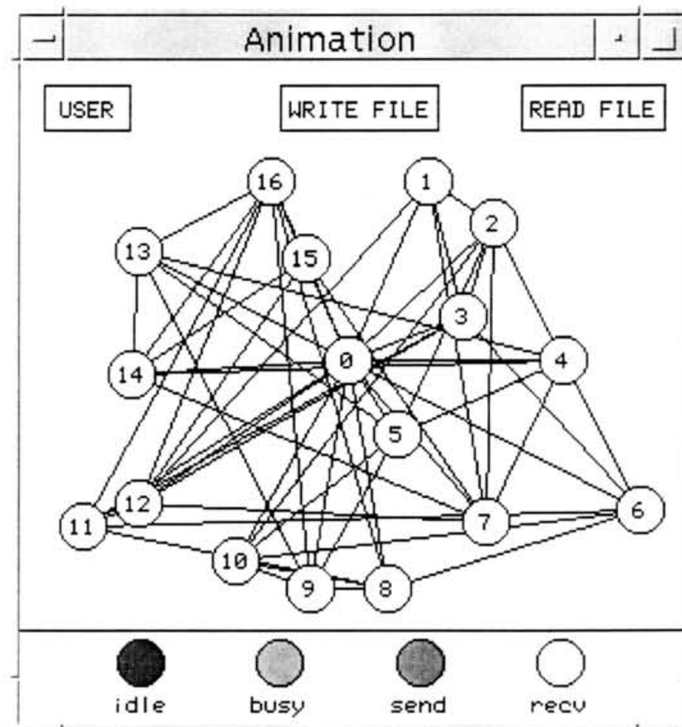


FIGURA 6.11 - Animation do ParaGraph (USER)

6.2.2 XPVM e PVaniM

Nesta seção a análise será realizada na mesma linha da anterior, ou seja, serão utilizadas para comparação ferramentas para PVM, onde a presença ou não de janelas similares às do TFP View será o foco principal. As duas ferramentas escolhidas foram XPVM [GEI 96] e PVaniM [STA 96], sendo que a primeira é a ferramenta padrão distribuída juntamente com o PVM.

O XPVM é um console e monitor gráfico para o PVM. Consiste de um conjunto de cinco janelas gráficas diferenciadas, onde cada uma demonstra um aspecto da aplicação paralela. Como mencionado anteriormente neste texto, o XPVM é uma ferramenta de monitoração *on-line*, ainda que seja possível a gravação de um arquivo de *trace* e sua posterior utilização para animação *post-mortem*. Entretanto, esta gravação terá sofrido a mesma intromissão que a execução *on-line*, caracterizando a principal desvantagem do XPVM.

As cinco janelas fornecidas pelo XPVM são:

- *Network*: demonstra a máquina virtual do PVM com um ícone para cada *host*. O estado dos *hosts* durante a execução vai sendo indicado por sua cor.
- *Space-time*: é um diagrama espaço-tempo onde cada processo é representado por uma barra horizontal e a cor desta barra indica o estado de cada processo.
- *Utilization*: é um resumo da anterior, demonstrado a cada instante durante a animação (número de processos em computação, em *overhead* e em espera).
- *Task-output*: mostra a saída textual de cada processo.
- *Call-trace*: demonstra textualmente a última chamada a uma rotina PVM de cada processo em execução.

Dentre estas cinco janelas, duas são de especial interesse para efeitos de comparação: *Network* e *Space-time*. Estas duas janelas estão presentes na figura 6.12. Esta, foi capturada após a execução do programa exemplo **master-slave** distribuído juntamente com o PVM.

Estas duas janelas do XPVM, que são as mais significativas, possuem características importantes e em comum com o TFP View. A primeira delas é a possibilidade de se visualizar a máquina virtual do PVM de uma maneira clara e simples. O XPVM o faz através da janela *Network* e o TFP View através da *Processes Mapping*.

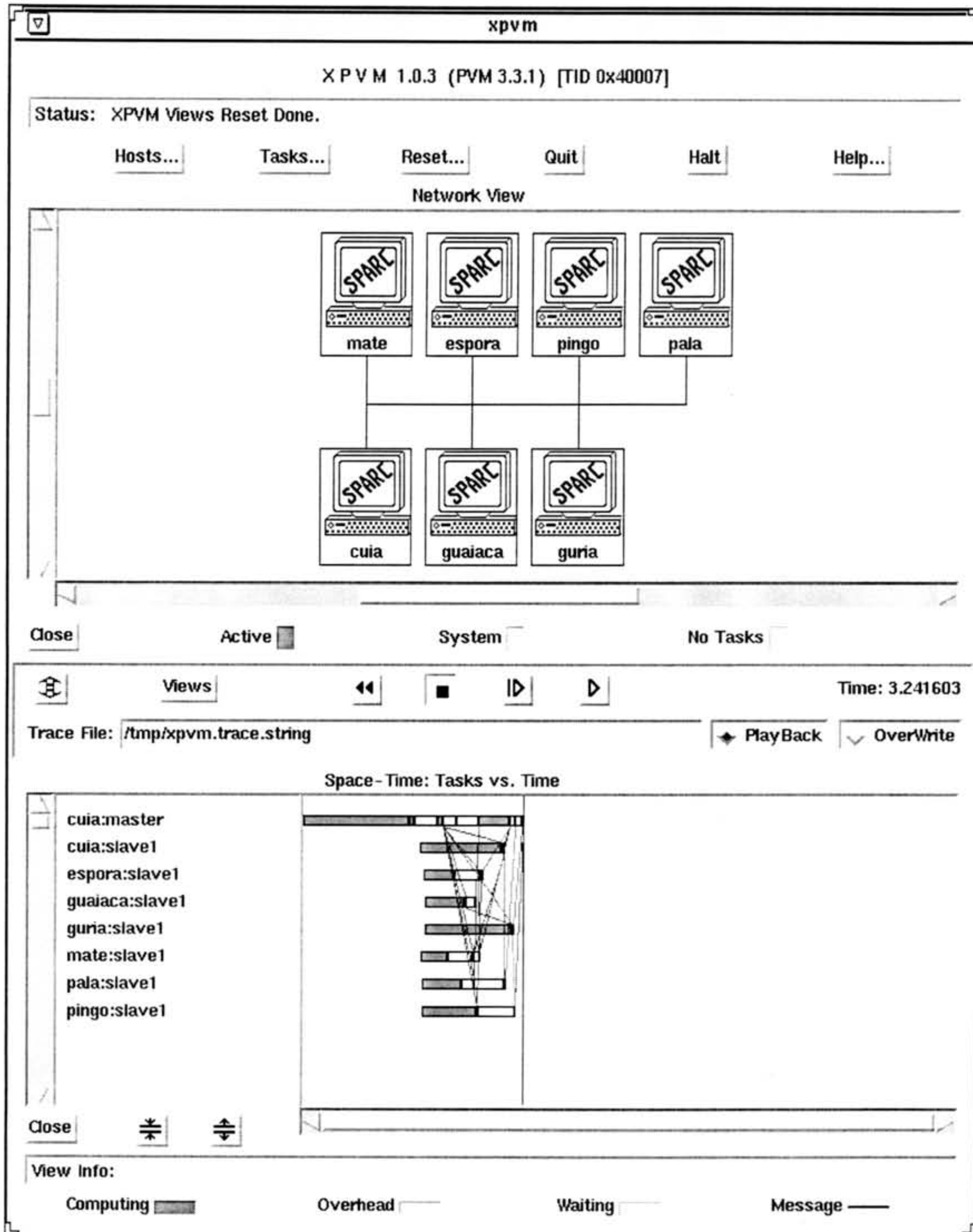


FIGURA 6.12 - Janelas do XPVM

Entretanto, as diferenças também não são poucas. Primeiramente, porque a do XPVM tem o principal objetivo de servir como **console gráfico** para o PVM e não para avaliação de desempenho. Por este motivo, ela demonstra toda a máquina virtual e não apenas a máquina que está sendo usada pela aplicação. Já a TFP View tem um outro objetivo que é o de ser utilizado para avaliação de desempenho de uma aplicação específica, não monstrando, portanto, os *hosts* que não foram utilizados por ela.

Obviamente que a principal diferença é o mapeamento dos processos nas máquinas em que foram executados pela TFP View. Apesar disso, o XPVM, com a ajuda de sua outra janela, a *Space-time*, também possibilita que se conheça essa informação. Isto porque esta janela apresenta cada processo com o nome de seu executável juntamente com seu *host*.

Em relação ao tipo de diagrama espaço tempo encontrado nas duas ferramentas, a diferença mais marcante é a ordenação dos processos, já definida anteriormente, realizada pela TFP View. Já o XPVM não tem esta ordem documentada, mas aparentemente ela é dada pela ordem alfabética dos nomes dos *hosts* e dos processos. Esta é uma vantagem da TFP View em relação ao XPVM, visto que o espaço destinado aos nomes dos processos na janela de visualização é melhor aproveitado.

A mesma vantagem existe com relação à janela similar encontrada no PVaniM, a *Causality View* (figura 6.13). Esta faz parte do grupo de janelas *post-mortem* desta ferramenta, que também tem um outro grupo de janelas para visualização *on-line*.

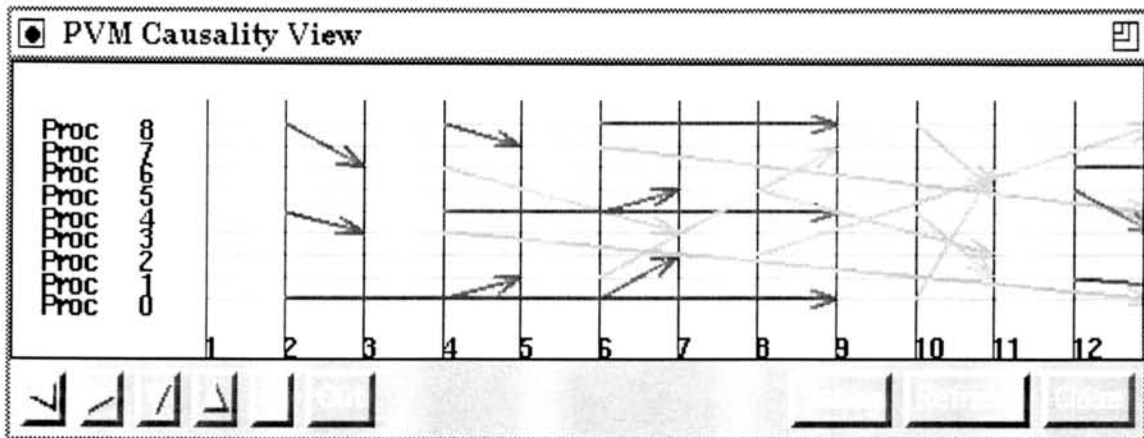


FIGURA 6.13 - Causality View do PVaniM

Esta janela consiste num sofisticado diagrama espaço tempo onde é possível visualizar tanto o volume da mensagem, quanto outras informações tais como emissor/receptor e tipo. Claro que esta se constitui numa grande vantagem em relação ao TFP View. Apesar disso, o eixo x dessa janela não é aproveitado, sendo os processos numerados e dispostos neste eixo de acordo com esta numeração.

Outra janela importante, apesar de não se mostrar adequada para uma comparação é a *Message Passing* (figura 6.14), que faz parte do grupo *post-mortem* do PVaniM. Esta é interessante na medida em que é uma espécie de sofisticação da janela *Animation* do PG, que também serviu de base para a criação da janela Grafo de Processos do TFP View.

A *Message Passing* é uma janela onde os processos estão dispostos de forma circular e as mensagens são pequenos círculos que "passeiam" do emissor ao receptor sempre passando pelo centro. A qualquer momento pode-se ter informações sobre a mensagem apenas clicando-se sobre ela. Este tipo de concepção para a mensagem pode ser aplicado tanto na janela *Processes Mapping* quanto na Grafo de Processos.

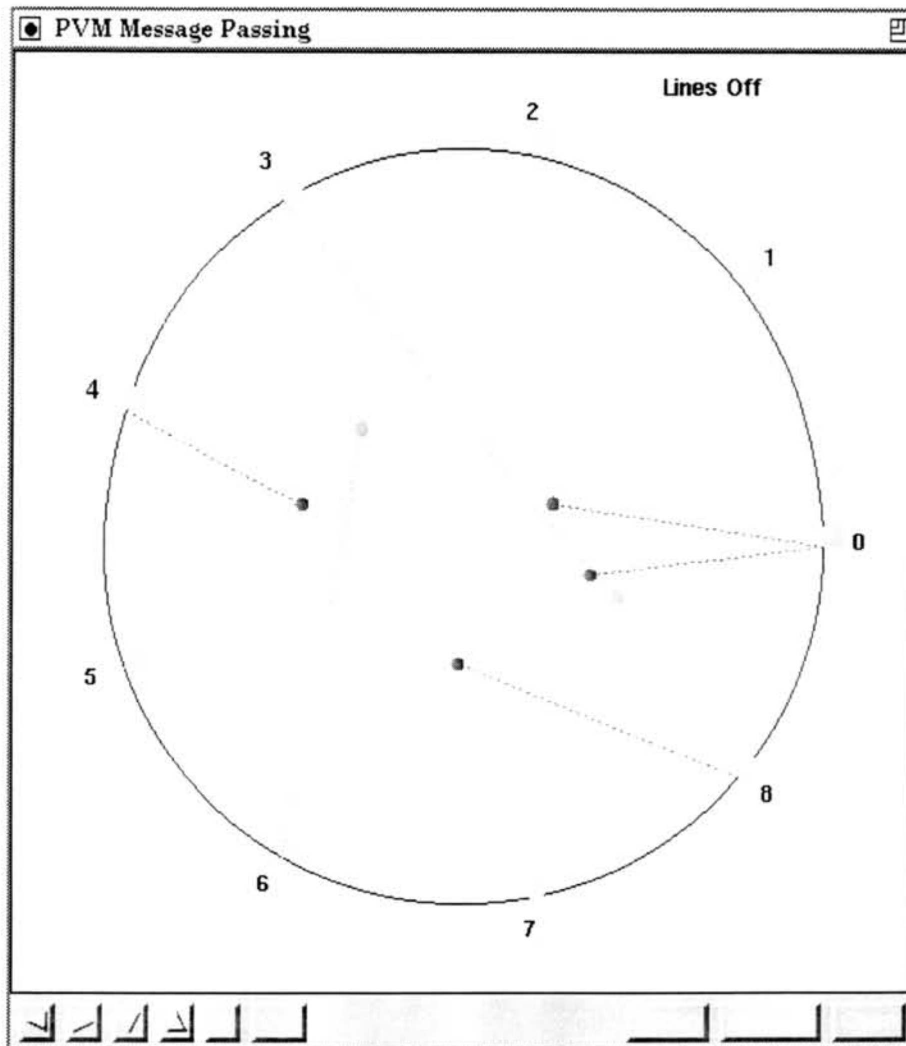


FIGURA 6.14 - Message Passing View do PVaniM

Dentre as janelas do tipo *on-line* do PVaniM, duas são de especial interesse: a *Host List* (figura 6.15) e a *Messages Sent/bytes sent* (figura 6.16). A primeira, consiste justamente numa espécie de mapeamento, feito no TFP View através da *Processes Mapping*. A diferença é que no PVaniM a janela é textual e apresenta os processos representados por seus índices definidos pelo usuário em um vetor que é passado para o sistema pela aplicação. Já a do TFP View é uma janela gráfica que contém os nomes alfanuméricos dos processos e, além disso, não é estática, apresentando uma animação dos estados dos processos e da troca de mensagens.

Host List

lenox	4
homepark	7
fulton	0,8
smyrna	3
decatur	1,2
oakmont	5,6

FIGURA 6.15 - HostList do PVaniM

A segunda, a *Messages Sent/bytes sent*, participa desta análise por expressar uma preocupação comum entre as duas ferramentas. Esta preocupação consiste em apresentar ao usuário informações sobre o volume de mensagens trafegando entre os processos. Esta janela apresenta estatísticas a respeito da quantidade de mensagens/*bytes* que são trocadas a cada intervalo de tempo. Já a TFP View demonstra este tipo de informação através das ordenações de *hosts* e processos que realiza e através da janela *Kiviat*, que demonstra o tráfego de mensagens na rede de interconexão.

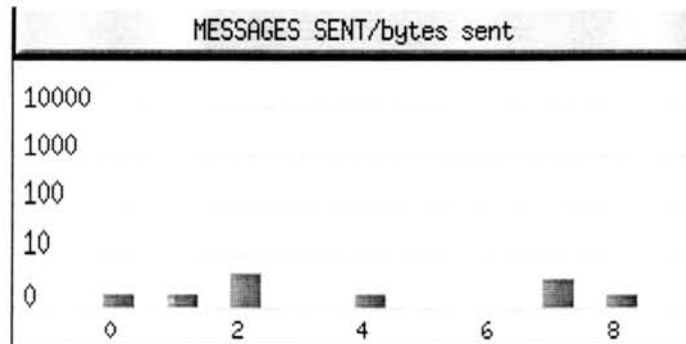


FIGURA 6.16 - Messages Sent/bytes sent do PVaniM

TABELA 6.1 - Tabela comparativa entre as janelas analisadas

Janelas	TFPS	ParaGraph	XPVM	PVaniM
Space-time	Ids. alfanum. Ordem eixo x. Mapeamento.	Ids. numéricos. - -	Ids. alfanum. - Mapeamento.	Ids. numéricos. - -
Kiviat	Utillização da rede. Ordem no círculo.	Utilização do processador. -	Não tem.	Não tem.
Processes Mapping	Localiz. gráfica dos processos.	Não tem.	Localiz. textual dos processos.	Localiz. textual dos processos.
Grafo de Processos[3]	Aproximação dos vizinhos.	Círculo ou usuário.	Não tem.	Círculo.

Vale lembrar que apenas um pequeno conjunto de janelas das ferramentas utilizadas para comparação foi abordado. Todas elas consistem de *toolkits* prontos para serem utilizados e possuem um conjunto de janelas que se complementam e são bastante abrangentes com relação aos parâmetros de desempenho que demonstram.

Para finalizar, apresenta-se a tabela 6.1 com uma comparação resumida entre as quatro ferramentas analisadas. Esta tabela leva em consideração apenas os tipos de janelas implementadas pelo **TFPS**.

[3]Título em português por não participar do protótipo implementado que funciona em conjunto com o ParaGraph.

7 Conclusão

Através de um estudo das ferramentas de avaliação de desempenho de programas paralelos mais importantes e utilizadas hoje em dia foi montada uma classificação levando em consideração os aspectos de finalidade e funcionalidade destas ferramentas. Esta classificação tornou possível a identificação da ferramenta apresentada no trabalho e é genérica o suficiente para que se possa utilizá-la em outros trabalhos deste tipo.

Observou-se, ainda, algumas características que tornam estas ferramentas alvo de críticas. A principal delas é a falta de janelas de visualização que demonstrem, de uma maneira clara e direta, a estrutura lógica da aplicação em análise. Isto se torna necessário principalmente para a análise de aplicações PVM, que são executadas a partir da *máquina virtual* criada pelo PVM e não têm qualquer compromisso com um tipo específico de arquitetura, podendo inclusive ser composta de máquinas heterogêneas.

Além disso, faltam também mecanismos que, além de demonstrarem o mapeamento de processos na máquina virtual, possibilitem e facilitem a alteração deste mapeamento em função de um possível aumento de desempenho da aplicação. Isto se faz necessário à medida em que o PVM não obriga o mesmo número de *hosts* e processos, permitindo mais de um processo por máquina. Ao mesmo tempo, não há uma política de balanceamento de carga inteligente, o que fica a cargo do programador da aplicação. Com isso, uma maneira de distribuição de processos através da máquina virtual poderia ser realizada a partir da colocação, em um mesmo *host*, de processos que possuem uma grande interação, evitando um alto tráfego de mensagens na rede de comunicação.

A partir destas observações, verificou-se que muitas das informações necessárias para se obter os tipos de visualizações requeridas teriam que ser obtidas **antes** da montagem das janelas e posterior animação das mesmas. Assim, chegou-se ao arquivo de *trace* que, por conter um histórico completo da execução do programa paralelo, contém também todas as informações necessárias para suprir a montagem e animação das janelas de visualização.

O mecanismo de pré-processamento do arquivo de *trace* empregado no **TFPS** possibilitou o aperfeiçoamento e criação de janelas de visualização que atendem aos requisitos e solucionam os problemas de análise de desempenho de programas PVM descritos acima.

O pré-processador se interpõem entre a fase de monitoração e a de animação, sendo um passo a mais na análise *post-mortem* de programas paralelos. Com isso, o pré-processador não influi nem no tempo de execução da aplicação, nem no tempo de execução da animação, ou seja, não há o indesejável compartilhamento de recursos que pode ser fatal em ferramentas de análise de desempenho de programas paralelos.

Foram aperfeiçoadas duas janelas já utilizadas em ferramentas de análise: Diagrama Espaço-tempo e Diagrama Kiviat, que sofreram alterações que possibilitam uma análise mais intuitiva e direta da aplicação paralela. Além disso, foram criadas duas novas janelas em função das informações obtidas através do pré-processamento: o Diagrama de Mapeamento de Processos e o Grafo de Processos, que visam demonstrar os aspectos lógicos da aplicação e relacioná-los aos aspectos físicos.

O pré-processador **TFP** foi implementado em linguagem C, com o auxílio da biblioteca de leitura de *traces* do TAPE/PVM, que é a ferramenta de monitoração que deve ser utilizada em conjunto com a aplicação PVM. Sua implementação é relativamente simples considerando-se o nível das informações que podem ser obtidas. Isto realça a vantajosa relação custo-benefício que o esquema de pré-processamento aqui apresentado pode oferecer aos sistemas de visualização de programas paralelos.

O protótipo da ferramenta de visualização **TFP View** foi implementado em ambiente de estações de trabalho Sun, linguagem C e interface baseada no sistema X Windows, desenvolvida com a biblioteca *Xlib*.

Salienta-se que este protótipo foi construído em conjunto com a ferramenta ParaGraph, que possibilita este tipo de desenvolvimento. Nada impede

entretanto que, no futuro, uma nova implementação independente seja desenvolvida. Isto não acarretaria em nenhuma mudança no pré-processador, que é inteiramente independente da ferramenta de visualização. Esta flexibilidade em relação às duas ferramentas complementares é apontada como uma outra vantagem do esquema aqui apresentado.

Bibliografia

[AYD 94] AYDT, Ruth A. **The Pablo Self-Defining Data Format.** Urbana, Illinois: Department of Computer Science, University of Illinois, 1994

[BAL 88] BAL, H. E. e TANENBAUM, A. S. Distributing Programming with Shared Data. **Computer Languages**, Oxford, v. 17, n. 3, p. 82-91, July 1992. Trabalho apresentado na International Conference on Computer Languages, 1988, Miami

[CAS 92] CASAVANT, Thomas L.; KOHL, James Arthur e PAPELIS, Yannis. Practical Use of Visualization for Parallel Systems. In: **Parallel Computing**: From Principles to Practice.[S.l]: Elsevier, 1992. p. 1-15.

[CAR 94] CARRIERO, N. et al. The Linda Alternative to Message Passing Systems. **Parallel Computing**, [S. l.], n. 20, 1994

[CHA 96] CHARÃO, Andrea S. e COSTA, Celso M. Yali - Uma Extensão do Modelo Linda com Suporte a Operações Globais. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES E PROCESSAMENTO DE ALTO DESEMPENHO, 8., 1996, Recife, PE. **Anais...** Recife: Departamento de Informática/UFPE, 1996. p. 173-182.

[CLÉ 95] CLÉMENÇON, Christian; FRITSCHER, Josef e RÜHL, Roland. Visualization, Execution Control and Replay of Massively Parallel

Programs within Annai's Debugging Tool. In: HIGH PERFORMANCE COMPUTING SYMPOSIUM, 1995. **Proceedings...** Montreal: [s.n], 1995. p. 393-404.

[CLÉ 96] CLÉMENÇON, Christian et al. **Annai Scalable Run-time Support for Interactive Debugging and Performance Analysis of Large Scale Parallel Programs.** Disponível por www em http://www.cscs.ch (abril 1996). (TR-96-04).

[FLY 74] FLYNN, M. J. Some Computer Organizations and Their Effectiveness. **IEEE Transactions on Computers**. New York, v. C-21, p. 948-960, Sept. 1972.

[FOS 95] FOSTER, Ian. **Design and Building Parallel Programs.** Disponível por www em http://www.mcs.anl.gov/80/dbpp (nov. 1995).

[FRU 91] FRUCHTERMAN, Thomas M. J. ; REINGOLD, E. M. Graph Drawing by Forced Directed Placement. **Soft. Practice and Experience**, London, v. 21, n. 11, p. 1129-1164, Nov. 1991.

[FRU 92] FRUCHTERMAN, Thomas M. J. **Nature Tool.** Disponível por ftp em emr.cs.uiuc.edu:/pub/grph-drawing/NatureDraw20Feb92.tar.Z, (feb. 1992).

[GEI 90] GEIST, G. A. et al. **A User's Guide to PICL**: a portable instrumented communication library. Oak Ridge: Oak Ridge National Laboratory, 1990. 25 p.

[GEI 94] GEIST, A. et al. **PVM 3 User's Guide and Reference Manual**. Oak Ridge: Oak Ridge National Laboratory, 1994. (TN 37831-6367).

[GEI 96] GEIST, G. A.; KOHL, James; PAPADOPOULOS, Philip. **Visualization, Debugging, and Performance in PVM.** Disponível por ftp em netlib2.cs.utk.edu/pvm3/xpvm (nov. 1996).

[HAB 90] HABAN, Dieter; WYBRANIETZ, Dieter. A Hybrid Monitor for Behavior and Performance of Distributed Systems. **IEEE Transactions on Software Engineering**, New York, vol. 16, n. 2, p. 197-211, Feb. 1990.

[HEA 91] HEATH, Michael T.; ETHERIDGE, Jennifer A. Visualizing the Performance of Parallel Programs. **IEEE Software**, New York, vol. 8, n. 5, p. 29-39, May 1991.

[HEL 91] HELMBOLD, David P.; MCDOWEL, Charles E. Computing Reacheable States of Parallel Programs. **ACM SIGPLAN Notices**, [S.l], v. 26, n. 12, p. 76-84, Dec. 1991.

[HOL 91] HOLLINGSWORTH, Jeffrey K.; IRVIN, Bruce R.; MILLER, Barton P. **The Integration of Application and System Metrics in a Parallel Program Performance Tool**. Disponível por ftp em ftp.cs.wisc.edu (jan. 1991).

[IBÁ 95] IBÁÑEZ, María Blanca. Plataforma Básica de Depurador para Programas Paralelos sobre PVM. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 15., e CONFERÊNCIA LATINO AMERICANA DE INFORMÁTICA,

21., 1995, Canela, RS. **Anais...** Porto Alegre: Instituto de Informática/UFRGS, 1995. p. 917-128.

[JOY 87] JOYCE, Jeffrey et al. Monitoring Distributed Systems. **ACM Transactions on Computers Systems**, [S.l.], v. 5, n. 2, p. 121-150, May 1987.

[KAR 94] KARRELS, Ed; LUSK, Ewing. **Performance Analysis of MPI Programs**. Disponível por ftp em URL: ftp://info.mcs.anl.gov /pub/mpi/misc/heath.ps, (nov. 1994).

[KIT 94] KITAJIMA, João Paulo F. et al. A Method and a Tool for Performance Evaluation. A Case Study: Evaluating Mapping Strategies. In: CUG - CRAY USERS GROUP - MEETING, 1994, **Proceedings...** Tours: [s.n.], 1994.

[KIT 95a] KITAJIMA, João Paulo F. **Programação Paralela Utilizando Mensagens**. Porto Alegre: Instituto de Informática/UFRGS, 1995. Curso apresentado na XIV Jornada de Atualização em Informática, no XV Congresso da Sociedade Brasileira de Computação, 1995, Canela, RS

[KIT 95b] KITAJIMA, João Paulo F. ANDES: a Tool for Evaluating Parallel Systems. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES E PROCESSAMENTO DE ALTO DESEMPENHO, 7., 1995, Canela, RS. **Anais...** Porto Alegre: Instituto de Informática/UFRGS, 1995. p. 367-381.

[LAM 78] LAMPORT, Leslie. Time, Clocks and Ordering of Events in a Distributed System. **Communications of the ACM**, [S. l..], v. 21, n. 7, p. 558-565, July 1978.

[LEB 87] LEBLANC, Thomas J.; MELLOR-CRUMEY, John. Debugging Parallel Programs with Instant Replay. **IEEE Transactions on Computers**, New York, v. C-36, n.4, p.471-482, Apr. 1987.

[LEU 93] LEU, Eric; SHIPER, Andre. ParaRex: A Programming Environment Integrating Execution Replay and Visualization. In: **Environments and Tools for Parallel and Scientific Computing.** Amsterdam: North-Holland, 1993.

[MAI 95] MAILLET, Eric. **TAPE/PVM an Efficient Performance Monitor for PVM Applications**: user guide. Grenoble: LMC-IMAG, 1995. 19 p. (Technical Report).

[MAL 96] MALARD, Joël. **MPI: A Message-Passing Interface Standard - History, Overview and Current State**. Disponível por www em http://www.epcc.ed.ac.uk/epcc-tec (feb. 1996).

[MEN 93] MENNA BARRETO, Ricardo. **Avaliação de Desempenho de Programas Paralelos.** Porto Alegre: Instituto de Informática/UFRGS, 1993. 182 p. Dissertação de Mestrado.

[MCD 89] MCDOWEL, Charles E.; HELMBOLD, David P. Debugging Concurrent Programs. **ACM Computing Surveys**, [S. l.], v. 21, n. 4, p. 593-622, Dec. 1989.

[MIL 94] MILLER, Barton P.; HOLLINGSWORTH, Jeffrey K.; CALLAGHAN, Mark D. **The Paradyn Parallel Performance Tools and PVM**. Disponível por ftp em ftp.cs.wisc.edu/tech-reports (aug. 1994). 10 p.

[MÜL 95] MÜLLER, Andreas; RÜHL, Roland. Extending High Performance Fortran for the Support of Unstructured Computations. In: INT. CONFERENCE ON SUPERCOMPUTING, 9., 1995. **Proceedings...** Barcelona: ACM Press, 1995. p. 127-136.

[NET 92] NETZER, Robert H. B.; MILLER, Barton P. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In: SUPERCOMPUTING, 1992. **Proceedings...** Washington, DC: IEEE Computer Society, 1992. p. 502-511.

[NOR 93] NORMAN, Michael G.; THANISH, Peter. Models of Machines and Computation for Mapping in Multicomputers. **ACM Computing Surveys**, [S. l.], v. 25, n. 3, p. 263-302, Sept. 1993.

[PER 92] PERUMALLA, Kalyan. **DBPVM: A Motif-based Graphical Interface to PVM**. Disponível por ftp em ftp.cc.gatech.edu /pub/software/dbpvm/ dbpvm3 (march 1995).

[REE 91] REED, D. A. et al. Scalable Performance Environment for Parallel Systems. In: DISTRIBUTED MEMORY COMPUTING CONFERENCE, 6., 1991. **Proceedings...** [S. l.]: IEEE Computer Society Press, 1991.

[RIE 91] RIEK, Maurice Van. **A General Approach to the Monitoring of Distributed Memory Machines**: A Survey. Lyon: Institut IMAG, 1991.

[SAL 92] SALTZ, Joel ; DAREMA, Frederica (chairs). **Software Tools**. In: WORKSHOP ON SYSTEMS AND TOOLS FOR HIGH PERFORMANCE COMPUTING ENVIRONMENTS, 1991. **Proceedings...** Pasadena: [s. n.], 1992.

[SIL 94] SILVA, Maria Inês Vale. **Desenho Automático de Diagramas**. Campinas: DCC-IMECC/Unicamp, 1994, 104 p. Dissertação de Mestrado.

[SNO 88] SNODGRASS, Richard. A Relational Approach to Monitoring Complex Systems. **ACM Transactions on Computers Systems**, [S. l.], v. 6, n. 2, p. 157-196, May 1988.

[STA 93] STASKO, John T.; KRAEMER, Eileen. A Methodology for Building Aplicattion-specific Visualizations of Parallel Programs. **Journal of Parallel and Distributed Computing**, [S. l.], v. 18, n. 2, p. 258-264, June 1993.

[STA 96] STASKO, John. **PVaniM 2.0**: Online and Postmortem Visualization Support for PVM. Disponível por www em http://www.gatech.edu /gvu/softviz/parviz/ pvanim (nov. 1996).

[STO 88] STONE, Janice M. Debugging Concurrent Processes: a Case Study. **ACM SIGPLAN Notices**, [S. l.], v. 23, n. 7, p. 145-153, July 1988.

[STR 95] STRINGHINI, Denise. **Um Estudo sobre Depuração Paralela Orientada a Eventos.** Porto Alegre: CPGCC da UFRGS, 1995. (TI-454).

[STR 96] STRINGHINI, Denise; NAVAUX, Philippe O. A. **Pré-processamento de Traces para Auxiliar a Visualização de Programas Paralelos**. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES E PROCESSAMENTO DE ALTO DESEMPENHO, 8., 1996, Recife, PE. **Anais...** Recife: Departamento de Informática/UFPE, 1996. p. 129-138.

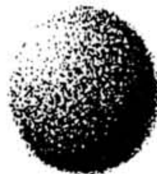
[SUN 94] SUNDERAM, V. S. et al. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. **Parallel Computing**, [S. l.], v. 20, n. 4, p. 531-545, Apr. 1994.

[TAN 92] TANENBAUM, Andrew S. **Organização Estruturada de Computadores**. 3. ed. Rio de Janeiro: Prentice Hall do Brasil, 1992

[TOP 95] TOPOL, Brad; SUNDERAM, Vaidy; ALUND, Anders. **PGPVM Performance Visualization Support for PVM**. Disponível por ftp em mathcs.emory.edu/ftp/pub/topol (feb. 1995).

[WAL 94] WALKER, David W. The Design of Standard Message Passing Interface for Distributed Memory Concurrent Computer. **Parallel Computing**, [S. l.], v. 20, n. 4, p. 657-673, Sept. 1994.

[WOR 92] WORLEY, Patrick H. **A New PICL Trace File Format**. Oak Ridge: Oak Ridge National Laboratory, 1992. 29 p.

Informática  *UFRGS*

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TFPS - Um Sistema de Pré-processamento de Traces para Auxiliar na Visualização de Programas Paralelos.

por

Denise Stringhini

Dissertação apresentada aos Senhores:

Prof. Dr. João Paulo Fumio Whitaker Kitajima (UFMG)

Prof. Dr. João Cesar Netto

Prof. Dr. Cláudio Fernando Resin Geyer

Vista e permitida a impressão.
Porto Alegre, 18/06/97.

Prof. Dr. Philippe O. A. Navaux,
Orientador.

INSTITUTO DE INFORMÁTICA
UFRGS
BIBLIOTECA