- **Centro Universitário de Brasília.**
- **Faculdade de Ciências Exatas e Tecnologia.**

PROJETO FINAL DE GRADUAÇÃO DO CURSO DE ENGENHARIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE UMA APLICAÇÃO DISTRIBUÍDA EM UM AMBIENTE CORPORATIVO UTILIZANDO PVM

ROBSON HIDEMITSU NAKAMURA

R.A.: 2001633/2

BRASÍLIA, 20 DE JUNHO DE 2005.

Centro Universitário de Brasília.

Faculdade de Ciências Exatas e Tecnologia.

PROJETO FINAL DE GRADUAÇÃO DO CURSO DE ENGENHARIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE UMA APLICAÇÃO DISTRIBUÍDA EM UM AMBIENTE CORPORATIVO UTILIZANDO PVM

AUTOR: ROBSON HIDEMITSU NAKAMURA
R.A.: 2001633/2

ORIENTADA POR: GUSTAVO GOIS

BRASÍLIA, 20 DE JUNHO DE 2005.

AGRADECIMENTOS

Agradeço primeiramente a Deus pelo dom da vida e pela tentativa de obter o dom da Perseverança.

Aos meus pais que sempre estiveram ao meu lado, me fazendo encontrar soluções diferentes.

E principalmente ao meu orientador Gustavo Góis que me ajudou muito na hora da monografia.

Agradeço também, aos meus amigos pela força e empréstimo de máquinas e de algumas idéias.

E por último e o mais importante a minha namorada Débora Alves Soares que sempre me apoio e ajudou nas horas mais difíceis.

Sumário

Sumário V
Lista de Figuras VII
Lista de Acrônimos IX
Resumo XI
Abstract XII
CAPÍTULO 1 1
Introdução 1
1.1 Estrutura de tópicos 1
1.2 Motivação 1
CAPÍTULO 2 3
Conceitos importantes 3
2.1 Granulosidade ou Nível de Paralelismo 3
2.2 Speedup e Eficiência 4
2.3 Arquitetura Paralela 5
2.3.1 Taxonomia de Flynn 5
2.3.2 Taxonomia de Duncan 10
2.4 Suporte à programação paralela 13
2.5 Topologias de comunicação 15
CAPÍTULO 3 18
Programação Concorrente 18
3.1 Projeto de um Algoritmo Paralelo 19
3.2 Desenvolvimento de Algoritmos Paralelos 22
3.3 Estilos de Paralelismo 22
3.3.1 Paralelismo geométrico 23
3.3.2 Paralelismo Processor Farm 23
3.3.3 Paralelismo Pipeline 24
3.4 Ativação de um processo paralelo 25
3.4.1 Fork/Join 26
3.4.2 Cobegin/Coend 27
3.4.3 Co-rotinas 28
3.4.4 DoAll 29
CAPÍTULO 4 31
Ambiente paralelo virtual 31
4.1 Modelo PVM 33
4.2 Componentes do PVM 34
4.2.1 As Bibliotecas de Programação 36
4.3 Tratamento das mensagens no PVM 38
4.4 Protocolo de comunicação 39
4.5 Tolerância a falha 41
4.6 Limitações do PVM 41
4.7 Exemplo de algoritmo usando PVM 43
4.7.1 Programa mestre usando PVM 43
4.7.2 Programa escravo 44
4.8 Outros ambientes de passagem de mensagem 44
4.8.1 P4 44

4.8.2 MPI 45
CAPÍTULO 5 48
Montagem do projeto 48
5.1 Ferramentas utilizadas no projeto 48
5.1.1 Hardware 48
5.1.2 Software 52
5.1.3 Ferramenta de tentativas frustradas 53
5.2 Montagem do projeto 55
5.2.1 Parte Física 55
5.2.2 Parte lógica 58
5.3 Programa integral 61
5.4 Desenvolvimento de algoritmo 62
5.4.1 Estilo de paralelismo escolhido 64
5.4.2 Balanceamento de cargas 65
5.4.3 Autonomia do escravo 67
5.4.4 Topologia da rede 68
5.5 Conhecimento obtido 68
5.6 Resultados obtidos 69
CAPÍTULO 6 71
Considerações Finais 71
6.1 Conclusão 71
6.2 Dificuldades do projeto 72
6.3 Sugestão para Futuros Trabalhos 72
Referências Bibliográficas 73
Anexo A Programa mestre com balanceamento de cargas 75
Anexo B Programa escravo 82

Lista de Figuras

Figura 2–1 SISD Fluxo único de dados e fluxo único de instruções 6

Figura 2–2 SIMD Fluxo Único de Instruções/Fluxo Múltiplo de Dados 7

Figura 2–3 MISD 9

Figura 2–4 MIMD 9

Figura 2–5 Classificação de Duncan 10

Figura 2–6 Hipercubo 12

Figura 2–7 .(a) Memória Centralizada,(b) Memória Distribuída 12

Figura 2–8 Topologias de redes (a) Topologia estática, (b) Topologia dinânica 16

Figura 3–1 Organização para execução concorrente 21

Figura 3–2 Distribuição das tarefas no tempo 21

Figura 3–3 Paralelismo Geométrico 23

Figura 3–4 Paralelismo Processor Farm 24

Figura 3–5 Paralelismo Pipeline 25

Figura 3–6 Fork/Join 26

Figura 3–7 Cobegin/Coend 28

Figura 3–8 Co-rotinas 29

Figura 3–9 Comparação entre seqüencial e DoAll 29

Figura 4–1 Componetes do PVM 35

Figura 4–2 Databuf 40

Figura 5–1 Esquema do Switch KVM 50

Figura 5–2 Switch KVM TK-400 da TRENDnet 50

Figura 5–3 Switch Mecânico 56

Figura 5–4 Switch KVM desmontado 57

Figura 5–5 KVM com teclado e mouse 57

Figura 5–6 Switch KVM Montado 57
Figura 5–7 Switch de Rede desmontado 58
Figura 5–8 Switch de Rede montado 58
Figura 5–9 Integral por área de trapézios 62
Figura 5–10 Detalhes da técnica do trapézio 62
Figura 5–11 Fluxo de processos 64
Figura 5–12 Topologia da rede 68
Figura 5–13 Erros na contabilização 69

Lista de Acrônimos

ANSI	American National Standards Institute
BSD	Berkeley Software Distribution
CC	Comando do compilador da linguagem C para Linux
CRC	Cyclic Redundancy Check (Contrôle de redondance cyclique)
DNS	Domain Name Server – Sevidor de resolução de nomes
GCC	Comando do compilador da linguagem C para Linux
GNU	*General Public Licence*
HD	Hard disk – Disco Rigido
IEEE	Institute of Electrical and Electronics Engineers
KVM	Keybord, Video, Mouse
MIMD	*Multiple Instruction stream/ Multiple Data stream* (Fluxo múltiplo de instruções , fluxo múltiplo de dados)
MISD	Multiple Instruction stream / Single Data stream (Fluxo mútiplo de instruções, fluxo único de dados)
MMX	*Multi Media Extensions*
MPI	*Message Passing Interface* (Interface de passagem de mensagem)
MPMD	*Multiple Program – Multiple Data* Múltiplos programas e múltiplos dados.
NFS	Network File System – Sistema de arquivo de rede
PVM	Parallel Virtual Machine – Máquina Virtual Paralela
PC	Computador Pessoal
PPS	Pacotes por segundo
PVMd	Componente do PVM. Processo que roda no sistema operacional.
RJ-45	Conector padrão de rede
RPM	Pacote de instalação de programa do sistema UNIX
RSH	Remote Shell Host
RSHsvc	Remote Shell Host service
SIMD	*Single Instruction Stream / Multiple Data stream* (Fluxo único de instruções, fluxo múltiplos de dados)
SISD	*Single Instruction stream/ Single Data stream* (Fluxo único de instruções, fluxo único de dados)
SPMD	*Single Program - Multiple Date* (Programa único e múltiplos dados)
TCP	*Transmission Control Protocol*

UDP	*User Datagram Protocol*
VNC	Virtual Network Connection
WWW	World Wide Web
XDR	External Data Representation Standard

Resumo

Há duas ações pelas quais se pode elevar o poder de processamento dos dados de uma empresa. A primeira é comprar um supercomputador multiprocessado, o que implicará gastos consideráveis com *hardware*, no desenvolvimento de aplicativo para controlar esses processadores. A segunda forma seria a construção de um *cluster* de *workstation*, usando algum ambiente de troca de mensagens. A segunda opção é bem mais acessível do que a primeira.

Este projeto tem como prioridade valorizar a *performance* da aplicação paralela que será projetada, bem como desenvolver um algoritmo que seja capaz de balancear as cargas entre as diversas máquinas existentes no projeto. Dar autonomia para as máquinas escravos bem com gerencias os recursos dessas maquinas como memória, espaço em disco.

Como exemplo para testar a máquina virtual, será utilizado um pequeno programa de cálculo de integral, utilizando a técnica de áreas de trapézios. O balanceamento de carga será dividido proporcionalmente às velocidades do processador de cada máquina.

Palavras-Chaves: PVM, Maquina Virtual Paralela, Balanceamento de cargas, Autonomia, Programação Paralela, Rede

Abstract

There are two ways of achieving a high level of information processing in a big company. The first one is to buy a multi-processed supercomputer, which would mean spending enormous amounts of money on hardware and the applicatory development of controlling these processors.

The second way is the construction of cluster of Workstation using some environment of message exchanging. This second option is more accessible than the first.

The priority of this project is to value the performance of the parallel application that will be projected as well as to develop an algorithm that is able to balance the loads between the different machines existing in the project.

As an example to test the virtual machine, a small program to calculate of integral will be used, applying the technique of area trapezes. In addition, the load balancing will be divided proportionally to the speed processors of each machine.

Keywords: PVM, Parallel Virtual Machine, Load Balance , Autonomy, Parallel Programming, Network

CAPÍTULO 1

Introdução

1.1 Estrutura de tópicos

O projeto consiste em montar e desenvolver um aplicativo paralelo usando pacote de passagem de mensagem PVM, bem como montar um ambiente com quatro máquinas para testar esse aplicativo.

Para o desenvolvimento de aplicação paralela é necessário um grande embasamento teórico sobre elementos de paralelismo e arquiteturas paralelas, que será discutido no capítulo 2.

No capítulo 3 será mostrado como montar e desenvolver um algoritmo para aplicações paralelas e quais os estilos de paralelização de um algoritmo e as estruturas básicas para comunicação entre os processos paralelos.

No capítulo seguinte, só se tratará do PVM: o que é, como funciona, as suas características, protocolo de comunicação, componentes e exemplo de algoritmo usando esse pacote de mensagem.

No capítulo 5, serão mostradas quais foram as ferramentas utilizadas para montar o projeto e relatar como foi montado, quais as principais configurações do sistema operacional e como foi feito o algoritmo paralelo. E os resultados do trabalho

E por ultimo no capitulo 6 esta a conclusão do trabalho, dificuldades do projeto e sugestões para trabalhos futuros.

1.2 Motivação

Um dos principais motivos para a escolha desse tema foi a grande necessidade de processamento exigido de diversas aplicações como calculo numéricos, calculo de

fractais, Cálculos de exploração de petróleo que uma grande empresa necessita, e ao mesmo tempo, os computadores ociosos dos seus próprios funcionários nos horários fora do expediente normal de trabalho.

E outro motivo foi a multidisciplinaridade de que o projeto necessita. O projeto utiliza diversas áreas como arquitetura de computadores, rede de computadores, cálculo numérico, linguagem de programação, entre outras; e todas essas áreas ligadas a informática.

CAPÍTULO 2

Conceitos importantes

Para se ter um maior embasamento teórico do que será falado sobre o projeto, há de se ter algum conceito de paralelismo, arquiteturas paralelas, modo de programação, elementos paralelos e demais conceitos muito importantes.

2.1 Granulosidade ou Nível de Paralelismo

Granulosidade de um sistema paralelo corresponde ao tamanho das unidades de trabalho submetidas aos processadores. Isso acaba influenciando na determinação do porte e da quantidade de processadores, uma vez que existe uma relação entre esses dois fatores. Em uma programa seqüencial, a unidade de paralelismo é todo o programa.

A granulosidade pode ser dividida em três níveis:

Granulosidade grossa entende-se com um paralelismo de alto nível, ou seja, poucos processos, mas grandes e complexos, e tem-se como unidade de paralelismo processos e programas.

Já a **granulosidade fina** é caracterizada pelo paralelismo de baixo nível, onde há um grande número de processos pequenos e simples e cujas unidades de paralelismo são instruções e operações, havendo centenas e milhares de processos.

Por último tem-se a **granulosidade média**, que é constituída de vários processos médios e cujas unidades de paralelismo são processos, instruções e *loops*, havendo dezenas de processos, que são caracterizados pela execução de trechos ou sub-rotinas de programas, de forma que o paralelismo é atingido entre os blocos ou sub-rotinas dos programas.

2.2 Speedup e Eficiência

O principal objetivo da computação paralela é diminuir o tempo de execução dos processos envolvidos. Diferentes métricas podem ser utilizadas para determinar se a utilização do processamento paralelo está sendo vantajosa ou não e quantificar o desempenho alcançado. O *speedup* é uma das métricas mais utilizadas para atingir esse objetivo, sendo definido como o aumento de velocidade em um computador paralelo com p elementos de processamento, em relação a um computador seqüencial, como a equação a seguir: (Tanembaum, 1995)

$$Sp = Ts / Tp$$

Onde:

Sp = *speedup* observado em um computador com p elementos de processamentos;

Ts = tempo de execução do programa em um único elemento de processamento;

Tp = tempo de execução do mesmo programa em uma computador paralelo, com p elementos de processamento.

Nos casos em que se atinge uma distribuição perfeita dos processos a serem executados em paralelo, obtém-se o valor máximo de *speedup*, ou seja, Sp = p (nesse caso, Tp = Ts/p). Na prática, diversos fatores influenciam no valor do *speedup* alcançado, principalmente para torná-lo menor. Alguns fatores que diminuem o *speedup* são: sobrecarga de comunicação e de sincronismo, balanceamento de carga e algoritmo ineficiente para explorar o paralelismo do problema que se deseja solucionar.

A eficiência representa a porcentagem dos elementos de processamento que estão sendo efetivamente utilizados de acordo com a equação a seguir.

E = Sp/p * 100

Onde:

Sp é speedup;

p = número de elementos de processamento.

2.3 Arquitetura Paralela

Há várias formas de se referenciar arquiteturas paralelas ou de multiprocessadores. Toda arquitetura paralela tem como principal objetivo aumentar o poder de processamento, aumentando o número de elementos de processamento.

Três características devem ser consideradas em uma arquitetura paralela:

- A granulosidade entre os elementos de processamento;
- A topologia que se refere ao padrão das conexões existentes entre os elementos de processamento;
- O controle de distribuição que está relacionado à alocação, à sincronização e à interação das tarefas dos elementos de processamento.

Na literatura existem duas taxonomias: a taxonomia de Flynn, que é a mais conhecida, e a taxonomia de Duncan, que abrange a maioria das arquiteturas paralelas.

2.3.1 Taxonomia de Flynn

Foi proposta em 1966 e é bastante utilizada e respeitada. A classificação é baseada no fluxo dos dados propostos por Flynn. Esta classificação considera o processo computacional como a execução de uma seqüência de instruções sobre um conjunto de dados. Aqui o modelo de Von Neumann, que corresponde à execução seqüencial de instruções, é visto como um fluxo único de instruções, controlando um fluxo único de dados (*Single Stream of Instructions/Single Strenm of Data* – SISD). A introdução de fluxos múltiplos de dados ou de fluxos múltiplos de instruções é que faz surgir o paralelismo. A combinação de fluxos de instruções únicos e múltiplos com

fluxos de dados únicos e múltiplos produz as quatros categorias de classificação de Flynn (1972).

SISD – Single Instruction Stream/Single Data Stream (Fluxo Único de Instruções/Fluxo Único de Dados)

Esta organização representa a maioria dos computadores seqüenciais disponíveis atualmente (Princípio de Vonn Neumann). As instruções são executadas seqüencialmente. A Figura 2–1 apresenta um esquema deste tipo de arquitetura.

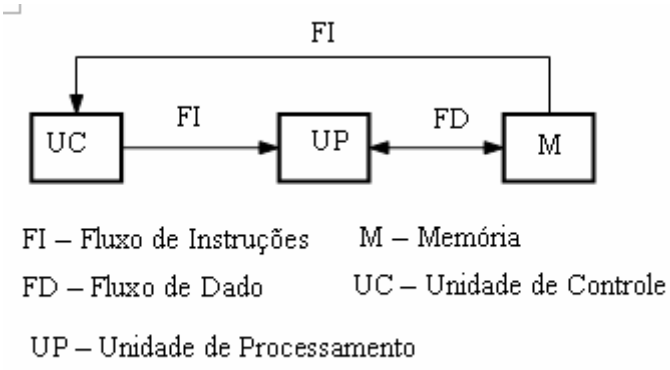

Figura 2–1 SISD Fluxo único de dados e fluxo único de instruções

Nessa categoria a execução seqüencial, pode ser otimizada pelo uso de *pipelines*[1], no entanto utiliza uma única unidade de controle.

1 É a execução de uma tarefa através da sua divisão num conjunto de subtarefas, empregando concorrência temporal. Na medida que conclui a execução num estágio e que o resultado segue para a próxima etapa, o estágio é carregado com a próxima tarefa.

SIMD – Single Instruction Stream/Multiple Data Stream (Fluxo Único de Instruções/Fluxo Múltiplo de Dados)

Há vários elementos processados (escravos) que estão sendo supervisionados por uma única unidade de controle (mestre). Todos os elementos processados recebem a mesma instrução para operar, mas recebem diferentes faixas de dados para processar. De acordo com a Figura 2–2. Normalmente os processadores vetoriais usam esse tipo de arquitetura paralela. Como é fácil de concluir, um computador com essa arquitetura é capaz de operar um vetor de dados por vez. Daí vem o nome de computadores vetoriais, ou *Array Processor* (Calônego, 1997).

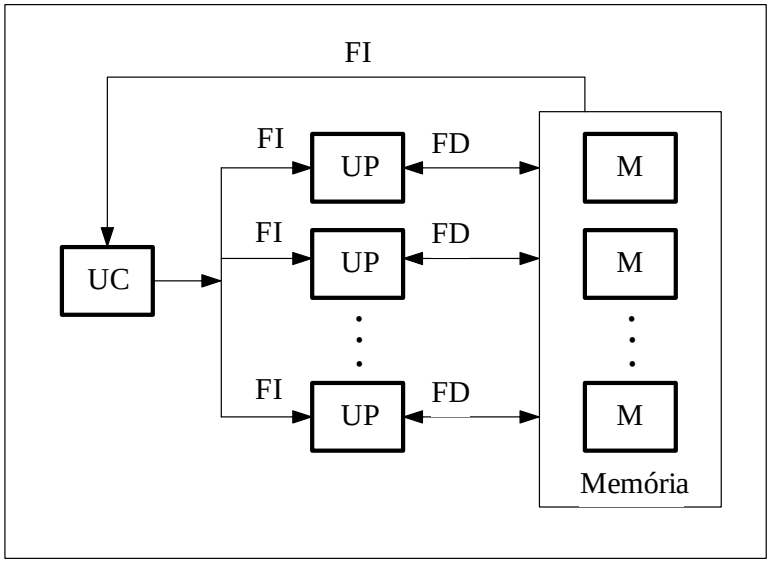

Figura 2–2 SIMD Fluxo Único de Instruções/Fluxo Múltiplo de Dados

O programa de arquitetura paralela é armazenado em uma memória de controle, que é também o repositório inicial para os dados. Os dados que serão processados pelos elementos de processamentos devem ser distribuídos a diversos elementos de memória.

A forma de conexão entre cada elemento de processamento e cada elemento de memória caracteriza a arquitetura SIMD e o modo pelo quais os elementos de processamento trocam informação entre si. Se cada elemento de processamento tem

acesso exclusivamente a um elemento de memória, caracterizando-se a arquitetura SIMD de memória local, neste caso, uma rede de interconexão interprocessamentos. Por outro lado, se todos os elementos de processamento estão conectados a todos os elementos de memória através de uma rede de interconexão, então a troca de informações é realizada por acesso a posições de memória compartilhadas, caracterizando assim um sistema SIMD de memória global.

Essa arquitetura é muito usada quando um mesmo programa deve ser executado sobre uma grande demanda de dados, como é o caso da prospecção de petróleo. Nota-se que nessa arquitetura não há problemas com a sincronização das tarefas, pois existe um único programa em execução.

MISD – Multiple Instruction Stream/Single Data Stream (Fluxo Múltiplo de Instruções/Fluxo Único de Dados)

Existem *n* unidades de processamento, cada uma recebendo diferentes instruções sobre um mesmo conjunto de dados. Alguns autores classificam essa arquitetura pouco significativa e alguns chegam a nem considerá-la. Em alguns casos, os “macropipelines” são encaixados nesta categoria, uma vez que a saída de uma unidade de processamento serve de entrada para outra unidade de processamento.

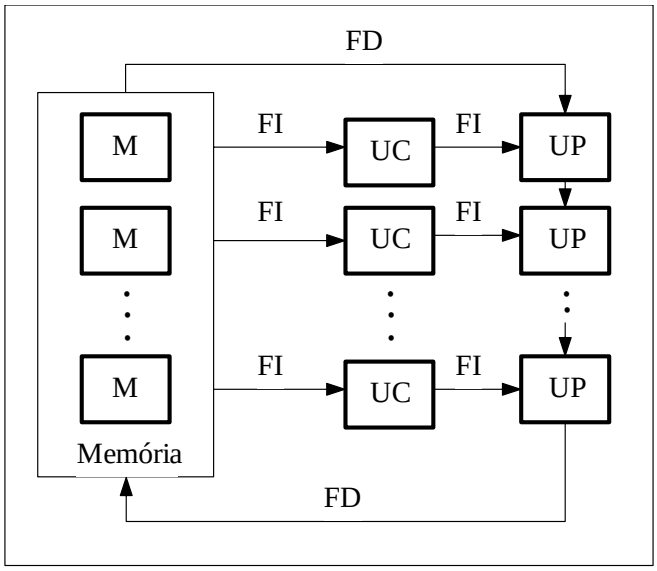

Figura 2–3 MISD

MIMD – Multiple Instruction Stream/Multiple Data Stream (Fluxo Múltiplo de Instruções/Fluxo Múltiplo de Dados)

A maioria dos sistemas multiprocessados está incluída nesta categoria. Nesta classe cada elemento de processamento é controlado pela sua própria unidade de controle, executando instruções independentemente sobre diferentes fluxos de dados. Essa arquitetura apresenta uma grande flexibilidade para desenvolvimento de algoritmos paralelos.

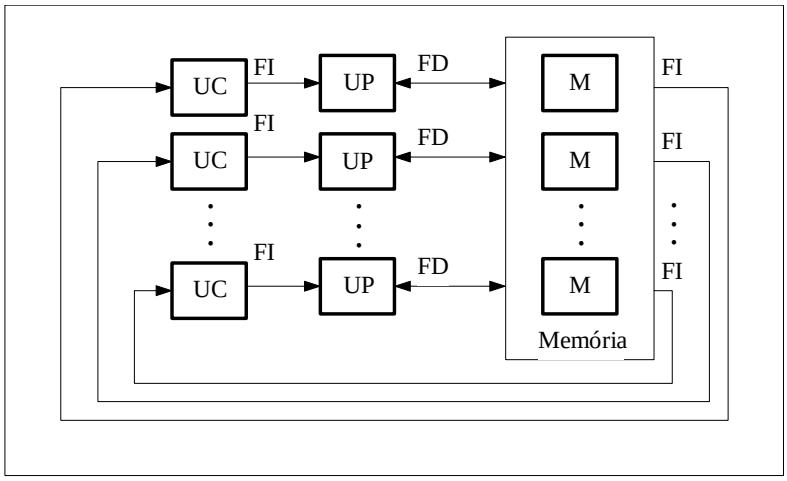

Figura 2–4 MIMD

2.3.2 Taxonomia de Duncan

Uma classificação mais refinada com o propósito de acomodar as inovações arquiteturais mais recentes, e que não se encaixa na de Flynn, foi proposta dor Duncan. Essa nova taxonomia conserva a taxonomia de Flynn e divide as arquiteturas em síncronas e assíncronas, possibilitando a inclusão de arquiteturas mais recentes que não são absorvidas pela taxonomia de Flynn. A Figura 2–5 apresenta uma visão geral da classificação de Duncan (Alamas – Duncan, 1990).

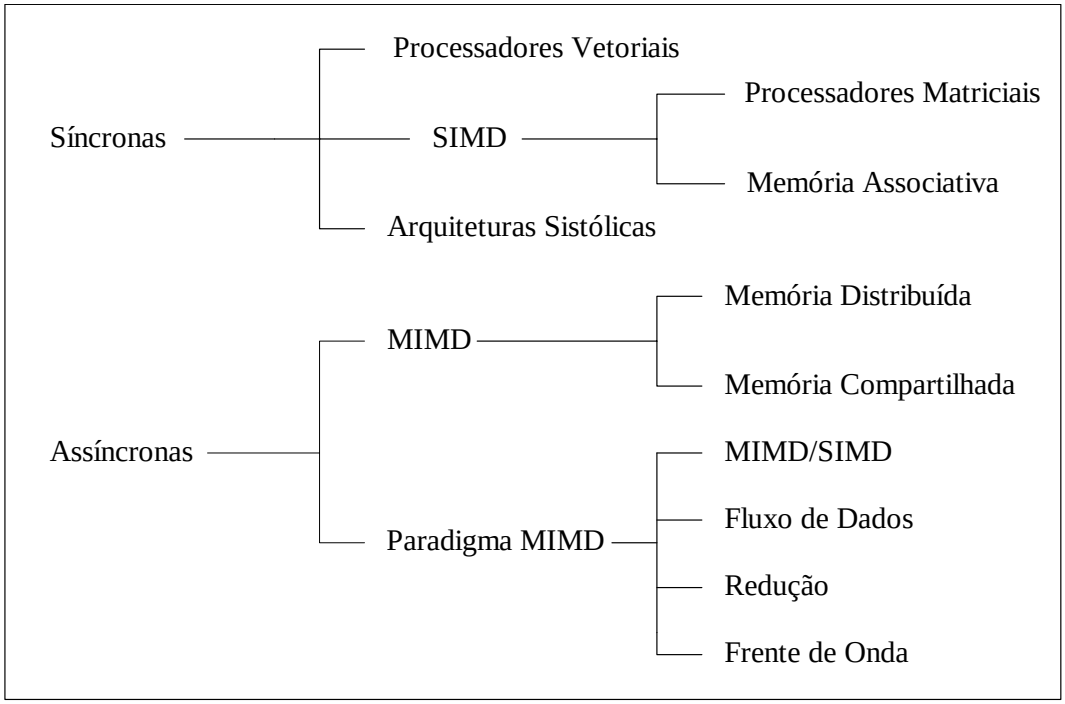

Figura 2–5 Classificação de Duncan

2.3.2.1 Arquiteturas síncronas

Arquiteturas síncronas são aquelas que coordenam as operações concorrentes com base em sinais de relógio global em unidades de controle centralizado. Têm um *clock* para controlar os processos.

O paralelismo vetorial é caracterizado pela execução da mesma operação sobre todos os elementos de vetor de uma só vez, de forma a otimizar as operações que são efetuadas. Esses processadores fazem uso de *pipeline* (Duncan, 1990).

Assim como os processadores vetoriais, os processadores matriciais também se encaixam, na classificação de Duncan, dentro das arquiteturas síncronas. Por fazer parte das máquinas SIMD, operam com múltiplos processadores que trabalham de forma paralela e síncrona.

Outro tipo de arquitetura presente na categoria síncrona são os arranjos sistólicos. Nesse algoritmo para aplicações específicas, dados fluem da memória para a rede de processadores e voltam para a memória sincronamente. As informações fluem sincronamente como se fosse uma pulsação sangüínea.

2.3.2.2 Arquiteturas Assíncronas

As arquiteturas assíncronas, que constituem o outro grande grupo em que é dividida a classificação de Duncan, caracterizam-se pelo controle descentralizado do *hardware* e cada instrução é executada em elementos de processamento diferentes. Essa categoria é composta por máquinas MIMD, convencionais ou não.

Arquitetura MIMD são computadores assíncronos, caracterizam-se pelo controle do *hardware* descentralizado. Nessa arquitetura a memória pode ser centralizada ou distribuída.

Nas arquiteturas de memórias distribuídas Figura 2–7 (b), cada processador possui sua memória local e os nós são interconectados através de redes de interconexão. Dessa forma, os nós se comunicam por meio de **passagens explícitas de mensagens**.

Diversas topologias de interconexão de rede podem ser utilizadas, destacando-se entre elas as arquiteturas de topologias em anel, árvore, hipercubo (Figura 2–6) e reconfigurável.

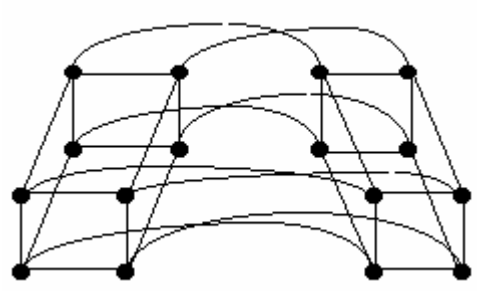

Figura 2–6 Hipercubo

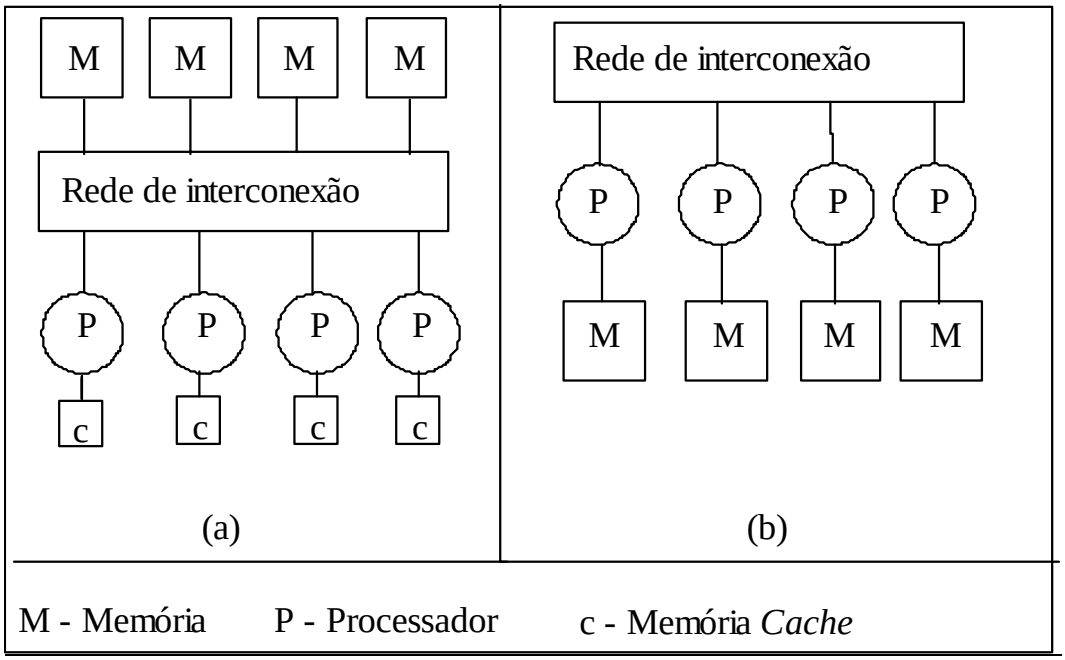

Figura 2–7 .(a) Memória Centralizada,(b) Memória Distribuída

As arquiteturas de memória compartilhada Figura 2–7 (a) , por sua vez, prove uma única memória que é compartilhada pelos processadores. Essas arquiteturas são também chamadas de multiprocessadores e não possuem problema no que diz respeito à troca de mensagens, forma pela qual são implementadas a comunicação e o sincronismo dos processos. Várias topologias de interconexão também são para esse tipo de arquitetura: *crossbar*, barramento e múltiplos estágios.

Como demais arquiteturas, enquadram-se as arquiteturas híbridas como fluxo de dados, redução e de frente de onda.

Arquiteturas híbridas são aquelas em que partes de arquiteturas SIMD são controladas por arquiteturas MIMD.

As arquiteturas *dataflow* (ou fluxo de dados) compõem-se da execução de seqüenciais de instruções baseadas na dependência de dados, que permitem a essas arquiteturas explorar concorrência em nível de tarefas, rotinas e até instruções.

Arquiteturas de redução, também conhecidas como dirigidas, apresentam um paradigma a partir do qual instruções são preparadas para executar quando os resultados são solicitados pelo operado ou por outras instruções. Redução implica a troca de parte do código-fonte pelo seu significado.

Finalmente, entre as arquiteturas que compõem a taxonomia de Duncan, têm-se as arquiteturas de frente de onda, que combinam *pipeline* de dados sistólicos com o paradigma de execução de *dataflow* assíncrono.

2.4 Suporte à programação paralela

Vários aspectos devem ser levados em conta na hora de escolher qual o tipo de ferramenta que será usado para a construção de um programa fonte que representa o algoritmo paralelo. Entre eles, o tipo de aplicação, tipo de usuário que utilizará a ferramenta e a arquitetura que executará os códigos gerados. Com base no tipo de aplicação e na arquitetura, define-se a granulação desejada que deve ser considerada também no processo de escolha. E, de acordo com Alamasi (1994), as coisas mais importantes na hora de se escolher a ferramenta são o tempo de trabalho do programador (tempo de programar mais o tempo de aprender a ferramenta) e o desempenho obtido com a paralelização do código.

Existe três ferramentas para ajudar montar um algoritmo paralelo:

Compiladores paralelizadores, que caracterizam o ambiente de paralelização automático, são responsáveis por gerar automaticamente versões paralelas de programas não paralelos. Tais compiladores exigem o mínimo de trabalho do usuário, mas o

desempenho obtido geralmente é modesto. Além disso, nem sempre esses compiladores são disponíveis, principalmente para sistema de memória distribuída. (GPACP)

Extensões paralelas são bibliotecas que contêm um conjunto de instruções que complementam linguagens seriais já existentes. Esses ambientes requerem algum trabalho do programador, mas ainda evitam a necessidade de aprendizagem de uma nova linguagem ou a completa reescrita do código-fonte (no caso onde se paraleliza um programa já implementado). Geralmente, o desempenho obtido é superior ao obtido pelos compiladores paralizadores. Existem extensões para arquiteturas de memória distribuída (MPI[2], PVM[3], P4[4]). Dentro das extensões paralelas, é interessante citar os ambientes de passagem de mensagens, que são ambientes de programação paralela para memória distribuída portáteis, que permitem o transporte de programas paralelos entre diferentes arquiteturas (e sistemas distribuídos) de maneira transparente.

O terceiro tipo de ferramenta, **linguagem concorrente**, relaciona as linguagens criadas especialmente para processamento concorrente, o que implica tempo de aprendizagem de uma linguagem totalmente nova e reescrita total do código-fonte. Essas ferramentas tendem a fornecer melhores desempenhos, de maneira a que se

---

[2] MPI Message Passing Interface - é uma biblioteca com funções para troca de mensagens, responsável pela comunicação e sincronização de processos

[3] PVM Parallel Virtual Machine é um conjunto integrado de bibliotecas e de ferramentas de software, cuja finalidade é emular um sistema computacional concorrente heterogêneo, flexível e de propósito geral. Será explicado mais detalhadamente no capítulo 4.

[4] P4 é uma biblioteca de macros e subrotinas desenvolvidas pelo *Argonne National Laboratory*, para a programação em uma variedade de máquinas paralelas. *P4* suporta o modelo de programação baseados em memória compartilhada (por meio de monitores) e distribuída (pela troca de mensagens).

compense a sobrecarga sobre o programador. Outra vantagem dessa linguagem é que geralmente ela possibilita a construção de códigos bem estruturados, tornando fácil a identificação dos processos que estão executando em paralelo e a comunicação entre eles. Exemplos dessas linguagens são: Occam e Ada. (MORSELLI)

De maneira sucinta, a escolha de uma ferramenta para programação paralela deve ser feita de acordo com os objetivos do programador. Compiladores paralelizadores oferecem desempenho ruim, porém com sobrecarga nula sobre o programador. Por outro lado, linguagens concorrentes oferecem melhor desempenho, mas oferecem uma sobrecarga considerável sobre o programador. Em um patamar intermediário de desempenho e sobrecarga sobre o programador, situam-se as extensões paralelas.

2.5 Topologias de comunicação

Uma topologia de comunicação trata da maneira como se estrutura a ligação entre os vários processadores e entre os processadores e a memória (rede de comunicação). Esse é um aspecto muito importante de uma arquitetura paralela, visto que a comunicação entre os processos influencia diretamente o desempenho de um programa paralelo.

Alguns aspectos devem ser considerados quando se analisa o desempenho de uma rede de comunicação quais sejam:

Latência – tempo de trânsito de uma mensagem pela rede de comunicação;

Bandwidth – quantidade de tráfego de mensagens que a rede de comunicação suporta;

Conectividade – quantidade de vizinhos que cada processador possui;

Confiabilidade – conseguida, por exemplo, através de caminhos redundantes.

Topologias de comunicação podem ser organizadas de duas maneiras: estaticamente Figura 2–8(a)ou dinamicamente Figura 2–8(b).

Topologias estáticas se caracterizam por conectar os elementos de processamento diretamente entre si, enquanto topologias dinâmicas utilizam-se de chaves na ligação entre os elementos de processamento. Enquanto que em topologias estáticas há necessidade de determinar explicitamente o caminho a ser seguido por uma mensagem trocada entre dois processadores (bibliotecas de comunicação, em nível de *software*, podem esconder este roteamento), em topologias dinâmicas somente é necessário o envio de uma mensagem para uma chave, que ela responsabiliza-se por rotear a mensagem até o seu destino. Por exemplo, a seguir uma mensagem trocada entre os processadores P1 e P4 necessitaria ser explicitamente transmitida para P2 para então alcançar P4. Este procedimento caracteriza uma topologia estática. Já no caso dinâmico, apenas se envia a mensagem para a rede (que no caso se trata de uma estrutura de chaves), que será transmitida automaticamente.

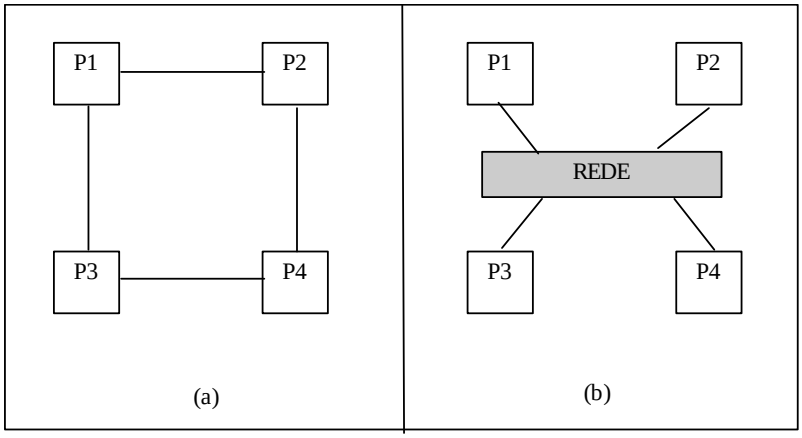

Figura 2–8 Topologias de redes (a) Topologia estática, (b) Topologia dinânica

Topologias estáticas não são reconfiguráveis, de maneira que a sua estrutura de conexão de processadores não pode ser modificada após a sua construção. Apresentam menor custo e maior simplicidade para o projeto, visto que a conexão dos elementos é regular, mas essas regularidades implicam a utilização destas topologias com eficiência

no projeto de computadores de propósito específico. Porém, é difícil a construção de um multiprocessador de propósito geral que se baseie numa topologia estática, visto que, para aplicações com padrões de comunicação variáveis, é interessante a utilização de topologias que adaptem a sua estrutura de conexão com a carga de trabalho nos vários processadores. Nesses casos, é mais útil o uso de topologias dinâmicas ou topologias estáticas que, em nível de *software*, possibilite padrões de comunicação dinâmicos.

Apesar de topologias dinâmicas apresentarem, em geral, melhor desempenho, são mais caras e de mais difícil construção. E também, geralmente, impõem restrições a futuras ampliações, isto é, o acréscimo de novos elementos de processamento à topologia pode diminuir drasticamente o desempenho.

Topologias estáticas são muito utilizadas para computadores SIMD e MIMD com memória distribuída. Exemplos para sua organização são: linear, anel, estrela, árvore, hipercubo, entre outros. Por outro lado, topologias dinâmicas são muito utilizadas em arquiteturas MIMD com memória centralizada, em virtude de possibilitar sua utilização são elas: barramento[5], multifásicas, entre outras.

Dentre as arquiteturas em geral, o modelo MIMD tem-se destacado (principalmente o modelo de memória distribuída) devido à sua flexibilidade e por representar uma boa opção para o desenvolvimento de algoritmos paralelos de granulações média e grossa.

---

[5] Esta topologia é caracterizada por uma linha única de dados (o fluxo é serial), finalizada por dois terminadores (casamento de impedância), na qual atrela-se cada nó de tal forma que toda mensagem enviada passa por todas as estações, sendo reconhecida somente por aquela que está cumprindo o papel de destinatário.

CAPÍTULO 3

Programação Concorrente

Um programa seqüencial é composto por um conjunto de instruções que são executadas seqüencialmente, sendo que a execução dessas instruções é denominada **processo**.(Andrews, 1993).

A programação concorrente consiste vários programas seqüência que estão sendo executado ao mesmo tempo, de forma a produzir um resultado particular mais rapidamente (Snow, 1992). Mas isso não se refere a múltiplos elementos de processamento e nem a paralelismo.

A existência de um único elemento de processamento implica que apenas uma tarefa seja executada a cada instante de tempo, tendo-se um pseudoparalelismo, ou seja, a ilusão que se tem é de que as tarefas estão sendo executadas em paralelo, mas na verdade o processador está sendo compartilhado com diversas tarefas que estão sendo executadas de forma intercalada, obedecendo a uma fila de processos. Há um possível ganho de velocidade, dependendo do tipo de tarefa que se intercala.

Um programa seqüencial é constituído basicamente de um conjunto de construções já bem dominadas pelo programador em geral, como, por exemplo, atribuições, comandos de decisão (if... then... else), laços (for... do), entre outros. Um programa concorrente, além dessas primitivas básicas, necessita de novas construções que lhe permitam tratar aspectos decorrentes da execução paralela dos vários processos.

Segundo Almasi Alamas, para a execução de programas paralelos, deve haver meios de:

- Definir um conjunto de tarefas a ser executadas paralelamente;
- Ativar e encerrar a execução dessas tarefas;
- Coordenar e especificar a interação entre essas tarefas.

A fase de definição da organização das tarefas paralelas é de extrema importância, pois o ganho de desempenho adquirido paralelização depende fortemente da melhor configuração das tarefas a serem executadas concorrentemente.

Definido o algoritmo, é necessário um conjunto de ferramentas para que o programador possa representar a concorrência, definindo quais partes do código serão executadas seqüencialmente e quais serão paralelas.

Além disso, processos cooperando para a resolução de determinado problema devem comunicar-se e sincronizar-se, a fim de que haja interação entre eles.

3.1 Projeto de um Algoritmo Paralelo

De acordo com Tanenbaum (1995), existem três maneiras de se construir um algoritmo paralelo:

- Detectar e explorar algum paralelismo inerente a um algoritmo seqüencial existente: essa abordagem é muito utilizada, apesar de apresentar baixo *speedup*, visto que não há necessidade de nova análise do algoritmo;
- Criar um algoritmo paralelo novo: possibilita melhor desempenho, necessitando, porém, de reestruturação completa do algoritmo;
- Adaptar outro algoritmo paralelo que resolva problema similar: nesse caso, tem-se bom desempenho, exigindo menor trabalho do programador em relação à construção completa do algoritmo.

Três aspectos importantes devem ser considerados quando se projeta um algoritmo paralelo. Primeiro, o processo de escolha da abordagem a ser seguida, entre as três citadas acima, deve ser feito de maneira cuidadosa, para que se consiga a melhor eficiência, levando em conta o tempo de escrita do algoritmo e o desempenho obtido.

Além disso, deve-se pesar o custo da comunicação entre processos em relação ao tempo de execução efetiva, visto que operações de comunicação geram uma sobrecarga que pode degradar o desempenho, tornando o algoritmo menos eficiente que o seqüencial.

Por último, deve-se considerar a arquitetura na qual se executará o algoritmo, visto que a sua eficiência pode variar de maneira drástica de acordo com o tipo de arquitetura.

Um exemplo de algoritmo paralelo:

Seja a expressão (a * b + c * d ^ 2) + (g + f * h). Uma possível organização da execução de cada uma das operações, respeitando-se a precedência, é mostrada na Figura 3.1 (a), onde cada tarefa (A, B, C, ..., G) executa uma operação matemática entre dois números. Pode-se organizar a execução de cada tarefa pelo diagrama apresentado na Figura 3–1(b). O sinal || representa tarefas sendo executadas em paralelo.

Supondo-se que se possuam três processadores (P1, P2 e P3), pode-se representar a distribuição das tarefas entre eles através de um diagrama *processadores x tempo*, apresentado na Figura 3–1(b). Para a construção do diagrama, deve-se respeitar a dependência entre as diversas tarefas. Por exemplo, a tarefa D necessita de dados das tarefas A e B e, portanto, necessita ser iniciada posteriormente ao término delas.

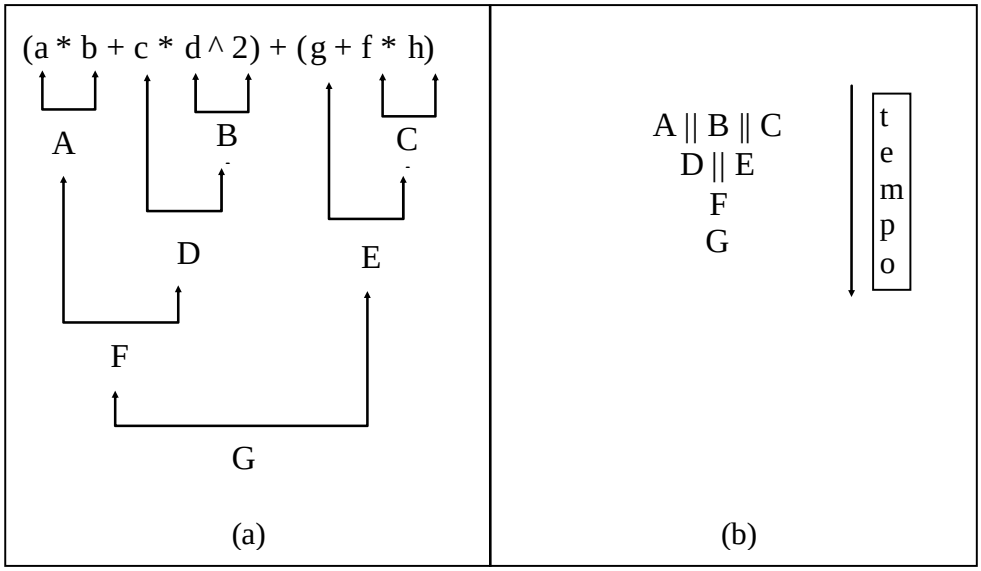

Figura 3–1 Organização para execução concorrente

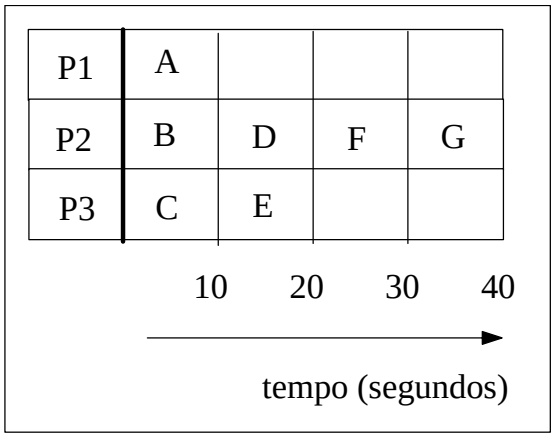

Figura 3–2 Distribuição das tarefas no tempo

Considerando os tempos de execução de todas as tarefas igual a 10 segundos (tA, tB, tC ..., tG = 10 segundos), têm-se:

Tempo seqüencial (tseq) = 70 segundos

Tempo paralelo (tpar) = 40 segundos

$$\text{Speedup} = \frac{70}{40} = 1{,}75$$

Como o paralelismo há um aumento de 75% da velocidade.

Se o Speedup fosse de 3 o programa paralelo estava perfeitamente paralelizado.

3.2 Desenvolvimento de Algoritmos Paralelos

No desenvolvimento de algoritmos paralelos, várias etapas, não utilizadas na programação seqüencial, devem ser seguidas. Em resumo, pode-se dividir o processo de desenvolvimento de algoritmos paralelos em 4 etapas:

1) Identificação do paralelismo inerente ao problema, que consiste no estudo do problema, a fim de se determinar possíveis eventos paralelizáveis;
2) Organização do trabalho, englobando a escolha do estilo de paralelismo que será utilizado (os estilos de paralelismo são apresentados na próxima seção) e a divisão de tarefas entre os diversos processadores a fim de maximizar o ganho de desempenho;
3) Desenvolvimento do algoritmo, utilizando ferramentas para expressar o paralelismo e para comunicação e sincronismo entre os processos;
4) Implementação por meio da utilização de um método de programação adequado. Esta etapa engloba a escolha de uma linguagem de programação adequada, tratamento de possíveis *deadlocks* e mapeamento de tarefas.

3.3 Estilos de Paralelismo

Um algoritmo paralelo pode ser organizado de várias maneiras, de acordo com o tipo de plataforma onde será executado. Genericamente, os modelos computacionais paralelos são divididos em dois tipos: memória compartilhada e troca de mensagens.

O modelo paralelo baseado em memória compartilhada divide-se de acordo com o tipo de acesso à memória. No modo síncrono (ou *PRAM – Parallel Random Acess Memory* – Acesso Aleatório Paralelo à Memória), os processadores atuam sincronamente no acesso à memória, enquanto no modo assíncrono esse sincronismo de acesso não existe.

O modelo paralelo, baseado em troca de mensagens, é dividido em três estilos, que são: paralelismo geométrico, paralelismo algorítmico ou processor farming.

3.3.1 Paralelismo geométrico

Também conhecido como paralelismo de resultados ou paralelismo de dados (Data Parallelism), caracteriza-se pela divisão do conjunto de dados a ser trabalhados igualmente entre todos os processadores. Dessa maneira, cada processador executa uma cópia do programa completo, porém em um subconjunto de dados. Este é considerado o modo mais fácil de desenvolvimento de algoritmos paralelos e é intensamente utilizado em computadores massivamente paralelos.

Considere, por exemplo, que se deseja construir uma parede de tijolos, onde cada um dos pedreiros é um processador. No paralelismo geométrico, o muro é dividido em partes iguais, e cada uma dessas partes é distribuída para um pedreiro (T1, T2, T3 e T4), como mostra a figura 3.3.

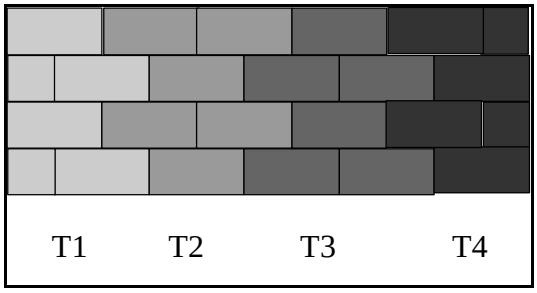

Figura 3–3 Paralelismo Geométrico

3.3.2 Paralelismo Processor Farm

Caracteriza-se pela existência de um processador “mestre” que supervisiona um grupo de processadores “escravos”, cada um processando assincronamente tarefas submetidas a ele pelo processador mestre. Este modelo também é conhecido como paralelismo pela pauta, visto que é definido um conjunto de tarefas (pauta) e a partir daí são distribuídas as tarefas pelo processador mestre.

No exemplo da parede de tijolos, uma construção baseada em *processor farm* (figura 3.4) pode ser organizada criando-se três tarefas básicas: pegar o cimento, pegar o tijolo e colocar no próximo lugar disponível. Essas três tarefas são submetidas a cada um dos pedreiros pelo mestre de obras (processador mestre), que, ao colocar o tijolo no lugar apropriado, está disponível para novas tarefas ordenadas pelo mestre de obras.

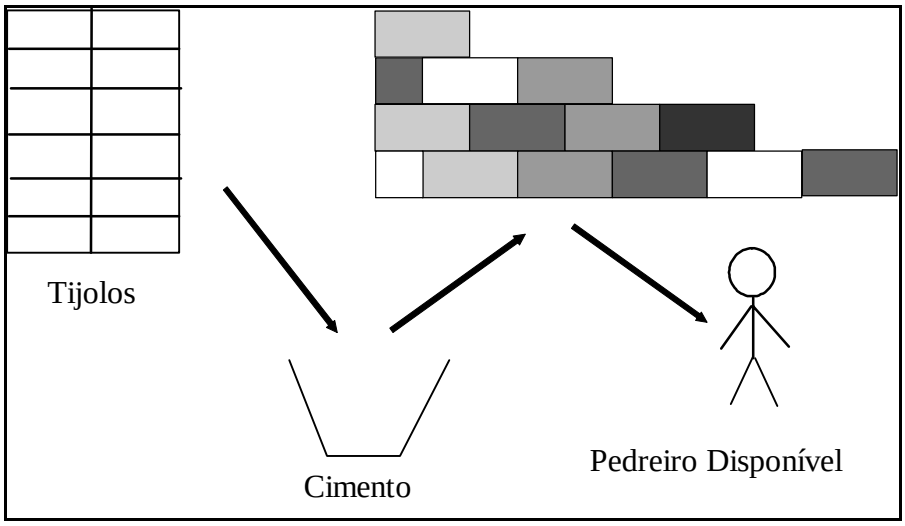

Figura 3–4 Paralelismo Processor Farm

O estilo *processor farm* possui várias vantagens, como, por exemplo: facilidade de ampliação do sistema, o que pode ser conseguido com o aumento de trabalhadores; facilidade de programação; balanceamento de carga mais natural, visto que as tarefas vão sendo submetidas aos processadores de acordo com a disponibilidade. Como desvantagens, podem ser citadas a sobrecarga de comunicação e a possibilidade de gargalo no processador mestre. Ou pode ter um gargalo em um escravo mais lento que não acompanhe os outros escravos.

3.3.3 Paralelismo Pipeline

Também conhecido por paralelismo especialista ou algorítmico, caracteriza-se pela divisão de uma aplicação em várias tarefas específicas, que são distribuídas aos processadores de forma *pipeline*. Nesse caso, quando apenas uma unidade de dados atravessa o *pipeline* não há paralelismo.

Um mecanismo *pipeline* para a construção de uma parede de tijolos é a divisão da parede a ser construída em várias camadas horizontais, de maneira a que cada pedreiro seja responsável pela construção de uma camada. Dessa maneira, o início das camadas superiores depende do término das primeiras camadas, como mostra a figura 3.5.

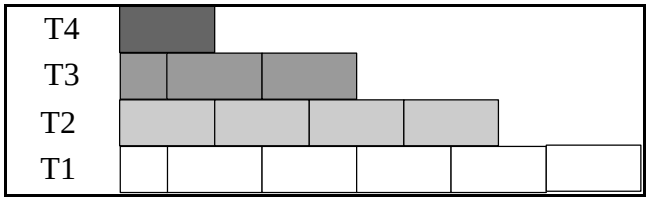

Figura 3–5 Paralelismo Pipeline

Muitas vezes, este é o estilo mais natural para desenvolver paralelismo a partir de um programa seqüencial, sendo até possível a sua detecção por compiladores e analisadores de código-fonte. Porém, algumas desvantagens que podem ser citadas são: pouca flexibilidade, visto que modificações no algoritmo podem requerer mudanças drásticas na rede de processadores; balanceamento de carga deve ser feito de maneira cuidadosa, a fim de que subtarefas lentas não tornem a execução de todo o pipeline lenta; a relação entre comunicação e execução deve ser baixa, a fim de que a sobrecarga de comunicação não torne o algoritmo ineficiente.

Durante o desenvolvimento de algoritmo paralelo, deve-se procurar escolher o estilo de paralelismo mais natural ao problema. Desenvolvido o algoritmo utilizando o método de programação mais natural ao estilo, podem-se conseguir resultados não satisfatórios em relação ao desempenho. Deve-se então transpor o algoritmo para um estilo de paralelismo mais eficiente. Podem-se utilizar métodos definidos em outras literaturas para facilitar essa transição entre estilos de paralelismo.

3.4 Ativação de um processo paralelo

Várias construções para a ativação e término de processos concorrentes são discutidas na literatura, apresentando características e finalidades distintas (Alamas,

1994; Andrews, 1983; Tanembaum, 1995; Snow, 1992). Dentre as especificações para ativação de processos concorrentes, *Fork/Join* foi uma das primeiras notações a ser criadas para programação concorrente. Outras notações como *Cobegin/Coend*, Co-Rotina e *DoAll* foram propostas, no entanto distinguem bem os conceitos de ativação e sincronização de processos.

3.4.1 Fork/Join

O comando *fork* implica a execução de um determinado conjunto de instruções (processo filho) a ser iniciado em paralelo ao conjunto que o executa (processo pai). Quando um comando *join* é por sua vez executado, ocorre a sincronização dos processos pais com os filhos gerados.

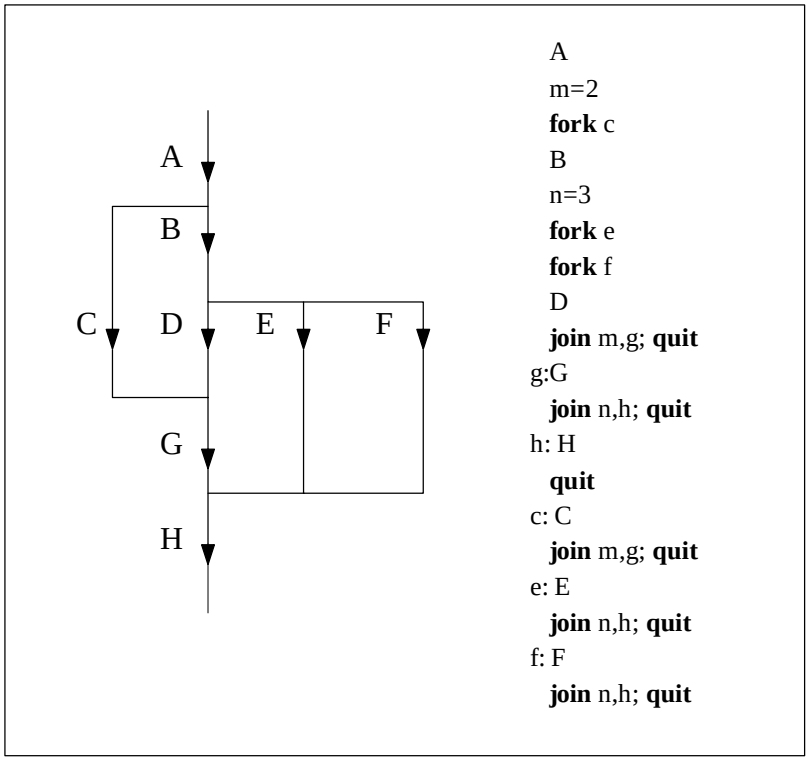

Figura 3–6 Fork/Join

As sintaxes dos comandos *fork* e *join* são:

Fork endereço – Faz que o processo que esteja localizado em endereço seja executado concorrentemente ao processo em execução;

Join var, end1, end2 – Onde var é decrescido de uma unidade antes de seu valor ser verificado. Caso seu valor seja zero, o processo no endereço end1 é executado, caso contrário, a execução é desviada para o endereço end2, que normalmente é utilizado para finalizar o processo mediante instrução *quit*. *Fork/join* têm como vantagem ser mais flexíveis que as outras notações, pois permitem a representação de qualquer execução paralela/seqüencial. Falta, entretanto, estruturação, fator que dificulta o entendimento do código.

3.4.2 Cobegin/Coend

O comando *cobegin* é uma estrutura que indica a execução concorrente de um conjunto de instruções em que o fim é indicado pelo comando *Coend*. Esses comandos são conhecidos também por *parbegin* e *parend*. A estrutura *cobegin*/*coend* é menos flexível que a *fork*/*join*, por exemplo, pelo fato de os processos ativados terem de terminar para que outros processos possam ser executados.

A sintaxe desse comando é:

cobegin

S1//S2//S3//...Sn

coend

onde S1, S2,... Sn são processos que serão executados em paralelo.

O processo pai é bloqueado até que S1, S2, S3,... Sn tenha terminado.

Quando o cobegin é executado é criado um processo para executar cada um dos comandos entre o cobegin e o coend. Baseado na linguagem Algol, os comandos entre o cobegin/coend podem ser simples ou compostos. Comandos compostos são delimitados por begin/end. Quando todos estes processos terminarem, o coend é executado, e o processo pai, o que executou o cobegin, recomeça a execução. Assim,

todos os processos entre o cobegin e o coend são executados concorrentemente. A Figura 3–7 a seguir ilustra a construção cobegin/coend.

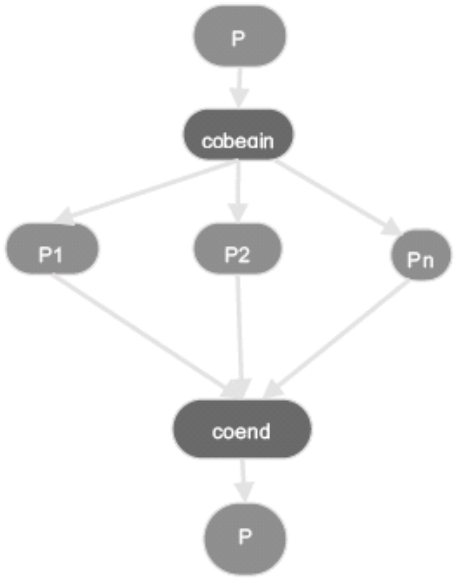

Figura 3–7 Cobegin/Coend

3.4.3 Co-rotinas

As co-rotinas são como sub-rotinas, mas permitem a transferência de controle de forma não hierárquica. Cada co-rotina pode ser vista com a implementação de um processo que pode ser executado intercaladamente. A seguir alguns comandos:

Resume – utilizado para transferência de controle de uma co-rotina para outra;

Call – utilizado para iniciar uma co-rotina;

Return – usado para transferir o controle para o programa chamado.

Uma sub-rotina, ativada por uma chamada call sub-rotina, ao executar o comando return retorna o controle ao módulo de programa que ativou e termina sua execução. Além disso, todas as vezes que ela é ativada, será executada desde o início. As co-rotinas transferem controle entre si de maneira livre, pelo comando resume co-

rotina. Sempre existe uma única co-rotina ativa em cada instante. As co-rotinas são utilizadas para ativação de processos paralelos.

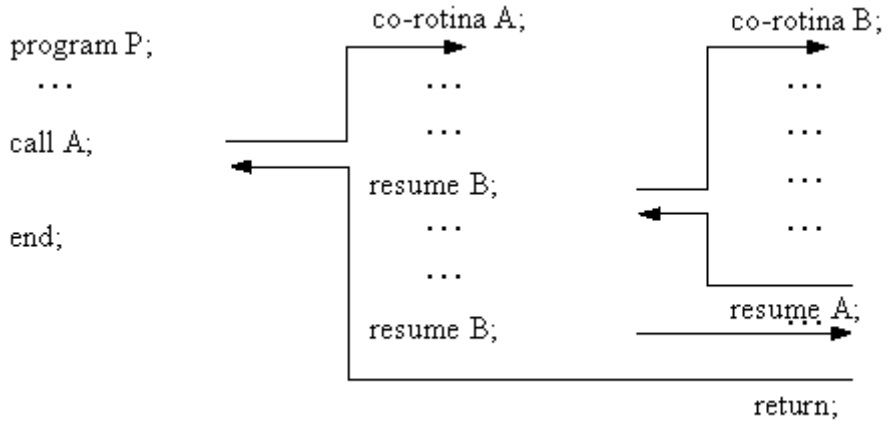

Figura 3–8 Co-rotinas

3.4.4 DoAll

Este comando pode ser visto como um comando cobegin/coend em que as instruções executadas em paralelo são as diversas instâncias de um bloco de comandos dentro de um comando de *loop*. Alguns comandos de função semelhante são: *forall*, *pardo* e *doacross*.

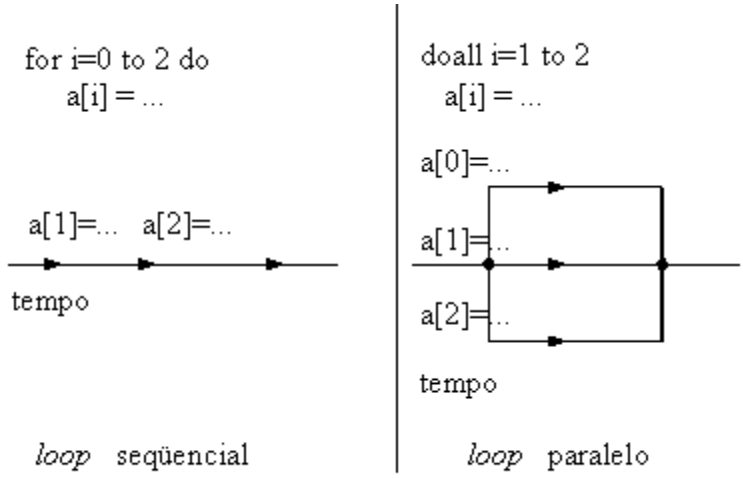

Figura 3–9 Comparação entre seqüencial e DoAll

A escolha do método de ativação de processos concorrentes deve ser feita de acordo com os objetivos do programador. Co-rotinas são utilizadas para ativação de processos concorrentes, *fork/join* e *cobegin/coend* para a ativação de processos paralelos e *doall* para a ativação paralela de instâncias de *loops*. Em relação ao contraste flexibilidade/estruturação, *fork/join* oferece um mecanismo flexível, porém desestruturado, enquanto *cobegin/coend* e *doall* apresentam maior estruturação, o que diminui a flexibilidade.

CAPÍTULO 4

Ambiente paralelo virtual

O modelo computacional MIMD tem-se destacado dentro da computação paralela devido à sua flexibilidade e ao seu potencial para execução de programas paralelos de complexidade média e alta. Entre as plataformas MIMD destacam-se as memórias distribuídas, que podem ser computadores paralelos ou máquinas virtuais.

O ambiente PVM (*Parallel Virtual Machine*) tem tido grande destaque na literatura, não só pela flexibilidade, mas também pelo fato de constituir um tipo de solução para os problemas de portabilidade de programas paralelos entre sistemas diferentes.

O PVM é um ambiente paralelo virtual que alcançou grande aceitação nos mais variados setores (acadêmico, industrial, comercia, etc.), podendo ser atualmente descrito como um padrão “de fato” entre outros ambientes paralelos virtuais.

O *Parallel Virtual Machine* é um conjunto integrado de bibliotecas de ferramentas de *software*, cuja finalidade é emular um sistema computacional concorrente heterogêneo, flexível e para qualquer finalidade (CSMORNL, 2005).

O projeto PVM teve início em 1989 no *Oak Ridge National Laboratory* – ORNL. A versão 1.0 (protótipo) foi implementada por *Vaidy Sunderam* e *Al Geist* (ORNL), sendo direcionada ao uso em laboratório. A partir da versão 2 (1991), houve a participação de outras instituições (como University of Tennessee, Carnegie Mellon University, entre outras), quando começou a ser utilizado em muitas aplicações científicas. A versão 2 deu início à distribuição gratuita do PVM. Depois de várias revisões (PVM 2.1 – 2.4), o PVM foi completamente reescrito, gerando a versão 3.0 (em fevereiro de 1993) (CSMORNL, 2005).

Várias mudanças foram feitas na versão 3.0, com objetivo de retirar erros de programação e ajustar pequenos detalhes, como oferecer interface com o usuário melhor e aumentar o desempenho de certas comunicações (como em multiprocessadores). A versão disponível mais recente é o PVM 3.4. Já foram realizadas várias revisões nessa versão, tendo como objetivo a retirada de erros e a inclusão de novas arquiteturas (PVM e-Book, 2005).

Diferentemente dos demais ambientes portáteis, o PVM nasceu com o objetivo de permitir que um grupo de computadores fosse conectado, permitindo diferentes arquiteturas de maneira a formar uma máquina paralela virtual. O sistema PVM permite que sejam escritas aplicações nas linguagens Fortran, C e C++, sendo que a escolha desse conjunto de linguagem se deve ao fato de que a maioria das aplicações possíveis de ser paralizáveis está escrita nessas linguagens.

Existem inúmeras vantagens em utilizar-se o PVM:

- Possibilidade efetiva de utilização de computação paralela, ainda que em arquiteturas não paralelas;
- Redução de tempo total de execução do programa;
- Capacidade de paralelização escalável e dinâmica;
- *Software* de domínio público;
- Grande difusão e aceitação;
- Flexibilidade;
- Variedade de arquiteturas e redes de trabalhos;
- Recurso computacional facilmente expansível;
- Independência das aplicações;
- Fácil atualização;

- Facilidade de programação graças a bibliotecas para linguagens universalmente usadas (Fortran e C).

O PVM está atualmente disponível para diversas plataformas de acordo com a Tabela 4-1

Tabela 4-1 Plataforma

Alliant FX/8
DEC Alpha
Sequent Balance
Bbn Butterfly TC2000
80386/486/Pentium com Unix (Linux ou BSD)
Thinking Machines CM2 CM5
Convex C-series
C-90, Ymp, Cray-2, Cray S-MP
HP-9000 modelo 300, Hp-9000 PA-RISC
Intel iPSC/860, Intel iPSC/2 386 host
Intel Paragon
DECstation 3100, 5100
IBM/RS6000, IBM RT
Silicon Graphics
Sun 3, Sun 4, SPARCstation, Sparc multiprocessor
DEV Micro VAX
80386/486/Pentium com Win32

4.1 Modelo PVM

Com o PVM, uma coleção de computadores heterogêneos (seqüenciais, paralelos e vetoriais) desempenha as funções de um computador com memória distribuída e alto desempenho. O PVM oferece funções para inicialização, comunicação

e sincronização de tarefas na máquina virtual. Uma tarefa é uma unidade computacional em PVM, análoga aos processos Unix. (PVM e-Book, 2005; Henrique, 2005).

O modelo do PVM é baseado na noção de que uma aplicação consiste de diversas tarefas. Cada tarefa é responsável pela execução de uma parte da aplicação. Uma aplicação pode ser paralelizada por dois métodos: o paralelismo funcional e o paralelismo de dados. No paralelismo funcional, a aplicação é dividida através das suas funções, isto é, cada tarefa desempenha um serviço diferente. O paralelismos de dados refere-se ao paradigma SPMD (Single Program – Multiple Data) descrito a seguir.

Os programas executados nos diversos processadores não devem, necessariamente, ser distintos, podendo utilizar o paradigma SPMD (Single Program – Multiple Data). Esse paradigma implica que o mesmo código-fonte seja distribuído pelos processadores e cada processador deve executar de maneira independente, o que implica a execução de diferentes partes desse programa, em cada um dos processadores. Quando se distribui códigos-fontes distintos para os processadores, utiliza-se o paradigma MPMD (Multiple Program – Multiple Data).

4.2 Componentes do PVM

O PVM possui dois componentes básicos, o *daemon*[6] (PVMd) e a biblioteca de rotinas com a interface PVM (Libpvm) como na Figura 4–1.

[6] *Daemon Um processo de segundo plano do sistema operacional que normalmente possui níveis de segurança de nível de root. Um daemon geralmente fica aguardando em segundo plano até que algo dispara sua atividade, como uma data ou hora específica, um intervalo de tempo, a chegada de um e-mail, etc.*

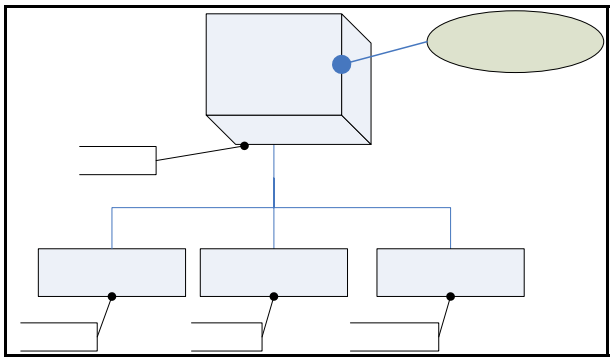

Figura 4–1 Componetes do PVM

O PVMd é um processo Unix, pertencente ao usuário, que é executado em cada *host* que compõe a máquina virtual, atuando com gerenciador da máquina e roteador de mensagens. É, portanto, o responsável pela aplicação PVM (Krug, 1999). O esquema de controle dos *daemons* baseia-se no modelo mestre-escravo, em que cada máquina deve possuir a sua versão do *daemon*, de acordo com a compilação para aquela arquitetura. Durante a maioria das operações, os PVMds não possuem diferença, isso ocorre apenas quando há necessidade de operações de gerenciamento, casos em que somente o mestre pode executar. (PVM e-Book, 2005)

O PVMd foi projetado para ser instalado por qualquer usuário com um login válido na máquina ou no domínio da rede. Quando um usuário deseja executar uma aplicação no PVM, ele deve primeiro criar a máquina virtual, iniciando-a com o processo PVMd. A aplicação poderá ser então executada a partir de qualquer máquina pertencente à máquina virtual. Pode haver mais de uma máquina utilizando os mesmo equipamentos simultaneamente, sendo que uma aplicação não interfere nas demais.

As estruturas de dados mais importantes no PVMd são as tabelas de *host* e de tarefas, que descrevem a configuração da máquina virtual e determinam as tarefas que estão sendo executadas.

O PVMd é executado em cada host da máquina virtual e eles são configurados para trabalhar juntos. O termo máquina virtual é utilizado para designar um computador

lógico com memória distribuída e o termo host é para designar um dos computadores que forma a máquina virtual. O PVMd não gera carga de processamento, ele atua exclusivamente como um roteador e controlador de mensagens, agindo como um ponto de contato entre cada *host*.

O primeiro PVMd (iniciado pelo usuário) é designado como mestre (*master*), enquanto os outros (iniciados pelos mestres) são chamados escravos (*slaves*). Durante a maioria das operações os PVMds não possuem diferença. Apenas quando há necessidade de operações de gerenciamento, como criar novos PVMds escravos e adicioná-los à máquina virtual, é que essa diferença aparece, pois somente o PVMd mestre pode fazer isto.

A segunda parte do sistema é uma biblioteca de rotinas da interface PVM. Ela contém um conjunto completo de primitivas que são necessárias para a cooperação entre as tarefas de uma aplicação.

4.2.1 As Bibliotecas de Programação

Todas as aplicações PVM devem ser ligadas com a biblioteca do PVM para permitir que elas se comuniquem com as outras entidades do sistema PVM. A biblioteca base (libpvm3.a) é escrita em C e suporta diretamente aplicações C e C++. A biblioteca Fortran (libfpvm3.a) consiste de uma série de funções para converter seqüências de chamadas Fortran para C (Krug, 1999).

Aplicações escritas em C devem ser ligadas pelo menos com a biblioteca PVM, libpvm3.a. Aplicações Fortran devem ser ligadas tanto com libpvm3.a quanto com

libfpvm3.a. Em alguns sistemas operacionais, programas PVM devem ser ligados também com outras bibliotecas fornecidas pelo fabricante (contendo, por exemplo, funções para *socket* [7]e XDR[8]).

Programas que usam funções de grupo devem também ser ligados com libgpvm3.a. Para utilizar alguma destas bibliotecas é necessário indicar explicitamente, na linha em que o compilador Unix (cc ou gcc) é chamado, a biblioteca que será utilizada. Por exemplo, para utilizar a biblioteca libpvm3.a, usa-se a opção -lpvm3. Para utilizar a biblioteca libgpvm3.a, usa-se a opção -lgpvm3.

Ou seja, para utilizar a biblioteca libx.a usa-se a opção -lx.

4.2.1.1 Funções da Biblioteca PVM

As sub-rotinas da biblioteca PVM podem, a grosso modo, ser divididas em cinco classes (Krug, 1999):

- Troca de mensagens;
- Controle de tarefas;
- Funções da biblioteca de grupo;
- Controle da máquina virtual;
- Funções diversas.

---

[7] Socket. É pequenos programas conectores utilizado entre aplicações em rede.

[8] XDR *eXternal Data Representation (Representação externa padrão)* Padrão para a representação de dados independente da implementação. Para se usar o XDR, o remetente traduz a representação dos dados na máquina local para a representação do padrão externo, enquanto a máquina destinatária faz o processo inverso.

4.3 Tratamento das mensagens no PVM

O PVM oferece rotinas para o “empacotamento” e o envio de mensagens entre tarefas. O modelo permite a qualquer tarefa enviar uma mensagem para outra, sem limites para o tratamento ou número dessas mensagens.

Enviar uma mensagem no PVM consiste em três passos:

- Um *buffer* [9]deve ser criado para que as mensagens enviadas sejam depositadas temporariamente;
- Mensagens devem ser “empacotadas” dentro do *buffer*;
- As mensagens inteiras (o conteúdo do *buffer*) são enviadas para outra tarefa ou grupo de tarefas.

O recebimento de mensagens, entretanto, pode ser feito por uma função bloqueante ou não-bloqueante. A mensagem é então “desempacotada”, retirando do *buffer* os dados enviados. As rotinas que manipulam o recebimento das mensagens podem ser instruídas para aceitar:

- Quaisquer mensagens;
- Quaisquer mensagens de uma tarefa específica;
- Quaisquer mensagens com um identificador específico (número de mensagem);
- Somente mensagens de uma tarefa específica com um identificador específico.

[9] Buffer Uma área de armazenamento temporária na RAM utilizada por informações que estão esperando o processamento pela CPU.

4.4 Protocolo de comunicação

O processo de envio de uma mensagem no PVM envolve três fases principais: a criação do *buffer* de envio, a preparação da mensagem e o envio efetivo da mensagem para a outra tarefa. A tarefa receptora recebe a mensagem através de uma função que retira a mensagem do *buffer*, residente no host receptor da mensagem. As funções para recebimento de mensagens podem aceitar quaisquer mensagens, qualquer mensagem de um *host* específico, qualquer mensagem com um identificador específico ou apenas mensagens de um *host* e identificador específicos. (PVM e-Book, 2005)

As formas de comunicação existentes no PVM são o *send* bloqueante assíncrono, *receive* bloqueante assíncrono e *receive* não bloqueante. O *send* bloqueante assíncrono retorna tão logo o *buffer* de transmissão esteja disponível para ser utilizado novamente pela aplicação, não dependendo da execução de um *receive* para poder retornar. O *send* bloqueia o processo quando o tamanho da mensagem exceder o tamanho do *buffer* de envio e precisar ser dividida. Nesse caso, é necessário que o *host* receptor execute um *receive* para liberar o *buffer*, permitindo assim a continuidade do envio da mensagem.

O *receive* bloqueante retorna apenas quando existem dados no *buffer* de recepção. A versão não bloqueante dessa função permite apenas a verificação desse *buffer*, retornando um código que indica se existem ou não mensagens no *buffer*. Além da comunicação ponto-a-ponto, o PVM disponibiliza para a aplicação as formas broadcasting (envio de uma mensagem para um grupo de tarefas definido pelo usuário) e multicasting (envio de uma mensagem para um conjunto de tarefas) para envio de mensagens. A comunicação disponibilizada pelo PVM é implementada pelos protocolos TCP (Transmission Control Protocol) e UDP (User Datagram Protocol), utilizando como ponto de acesso a interface de *sockets*.

O protocolo TCP é utilizado para a comunicação entre as tarefas e o PVM *daemon*, devido à confiabilidade que esse protocolo possui. Já a comunicação entre os PVM *daemon* é implementada utilizando o protocolo UDP.

O padrão XDR é utilizado quando existe comunicação em ambientes heterogêneos. Essa utilização é muitas vezes necessária e gera tempos adicionais na transmissão da mensagem, devido às sobrecargas impostas para a codificação da mensagem pelo processo emissor e a decodificação da mensagem pelo processo receptor. É por esse motivo que sistema usando MPI tem melhor *performance* do que PVM.

As mensagens podem variar o seu tamanho dinamicamente, sem a necessidade de que seja conhecido de antemão o seu tamanho máximo. Todas as funções de preparação da mensagem utilizam blocos de memória (ou, de acordo com a terminologia PVM, fragmentos) de tamanho relacionado com a mensagem que será enviada. Esses fragmentos são chamados de *databufs* (figura 4.2) e são organizados por descritores de fragmentos (*struct frag*).

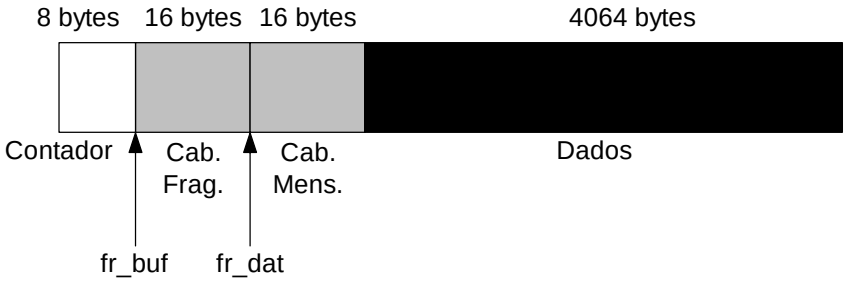

Figura 4–2 Databuf

Os *databufs* são compostos de um cabeçalho de 32 *bytes* de tamanho, um contador de referência (8 *bytes*) e uma porção de tamanho variável de dados. O tamanho de cada *databuf* é configurado em tempo de compilação, podendo ser alterado pela aplicação através da função pvm_setopt(). Para a maioria das plataformas, o valor inicial é 4096 *bytes* (dados e cabeçalhos), mais 8 *bytes* para o contador de referências.

Logo, as mensagens trocadas entre as tarefas PVM são fragmentadas em mensagens de tamanho máximo igual ao tamanho máximo de um *databuf*, menos o tamanho do cabeçalho (32 *bytes*). Após essa fragmentação, os dados são enviados pela rede, através dos protocolos TCP ou UDP. Por exemplo, se uma tarefa deseja enviar 4096 *bytes* de dados para uma outra tarefa, o PVM fragmentará essa mensagem em dois *databufs*: um com 4064 *bytes* de dados e um outro com 32 *bytes* de dados. Esse processo de fragmentação é transparente ao usuário.

4.5 Tolerância a falha

A terceira versão do PVM foi projetada para resistir à maioria das falhas envolvendo *host* e redes. Ele não recupera automaticamente uma aplicação depois de algum erro, porém fornece os recursos necessários para que o usuário construa aplicações tolerantes a falhas (pelo menos à maioria delas).

Quando um *host* escravo perde a comunicação com o mestre, ele mesmo (o escravo) provoca sua saída da máquina virtual, eliminando também todas as tarefas e operações pendentes nos salvadores de contexto, permitindo que a máquina virtual continue operando.

Por outro lado, quando o *host* mestre perde a comunicação com o *host* escravo, ele é retirado da máquina virtual pelo mestre.

Quando o *host* mestre for “perdido”, toda a máquina virtual é finalizada.

4.6 Limitações do PVM

O PVM foi projetado para, sempre que possível, não impor nenhum tipo de limitação de acesso aos seus recursos. Normalmente os limites são impostos pelo *hardware* e/ou pelo sistema operacional utilizado. Sistemas multiusuários afetam dinamicamente os limites do sistema.

A quantidade de tarefas que cada PVM pode gerenciar e a quantidade de memória disponível para tais processos são os dois grandes fatores de limitação do PVMd. O número de tarefas PVM é limitado por dois fatores. O primeiro está relacionado com o número de processos concorrentes permitidos pelo sistema operacional, sendo raramente esgotado pelo PVM. Não faz sentido ter uma quantia demasiadamente grande de processos em um único *host*. O segundo fator é o número de descritores de arquivos permitidos ao PVMd, ou seja, quantos arquivos um processo pode manter aberto ao mesmo tempo. É bom lembrar que cada mecanismo de comunicação (*socket*) representa um descritor utilizado, assim como cada tarefa filha do PVMd (a grande maioria das tarefas PVM são filhas do *daemon*). O PVMd pode tornar-se o "gargalo do sistema" se todas as tarefas tentarem comunicar-se através dele. O PVMd aloca memória dinamicamente para armazenar as mensagens roteadas por ele.

Enquanto a tarefa de destino não aceita a sua mensagem, os pacotes são armazenados em filas no PVMd, sendo que não há controle algum de fluxo para tais pacotes, ou seja, o PVMd irá aceitar qualquer pacote que chegue para ele, até não conseguir mais alocar memória para armazená-los.

As tarefas PVM também possuem um limite no número de conexões diretas que elas podem fazer com outras tarefas, embora esse problema não exista se a comunicação for feita através do PVMd.

A maior mensagem PVM possível para uma tarefa é limitada pela quantia de memória disponível para a tarefa. Quando a mensagem é enviada via PVMd, ele aloca memória para poder roteá-la, o que diminui o tamanho considerado disponível.

Se muitas tarefas enviam mensagens, ao mesmo tempo, para a mesma tarefa destino, tanto o PVMd quanto a tarefa destino ficam sobrecarregados, tentando armazenar as mensagens. Esses problemas devem ser considerados, projetando-se

aplicações para usar mensagens pequenas, eliminar gargalos e gerenciar as mensagens na mesma ordem que elas foram geradas.

4.7 Exemplo de algoritmo usando PVM

Um exemplo do programa que o pacote PVM disponibiliza na pasta /usr/share/pvm3/examplo/. É um pequeno programa mestre que envia uma mensagem para um escravo e esse escravo retorna o seu próprio nome.

4.7.1 Programa mestre usando PVM

Esse programa mestre envia a mensagem para o escravo e recebe mensagem do escravo e exibe o resultado na tela. O código desse programa é hello.c

```
#include “pvm3.h” /* Arquivo de cabeçalho para o PVM */
int main()
{
int cc; /* Resultado de chamadas a funções do PVM */
tid; /* Identificador de processo */
char buf[100]; /* String recebido na mensagem */
/* Imprime uma mensagem com o número do próprio processo */
printf("i'm t%x\n", pvm_mytid());
/* Inicia a execução de 1 programa hello_other */
cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);
/* Se conseguiu iniciar 1 programa hello_other, então ... */
if (cc == 1) {/* ... espera por uma mensagem. */
cc = pvm_recv(-1, -1);
/* Descobre o número do processo que enviou a mensagem. */
pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
/* Desempacota a mensagem. */
pvm_upkstr(buf);
/* Imprime a mensagem recebida. */
printf("from t%x: %s\n", tid, buf);
}
else /* Caso contrário, imprime uma mensagem de erro. */
printf("can't start hello_other\n");
/* Encerra a execução do programa. */
pvm_exit();
exit(0);
}
```

4.7.2 Programa escravo

Esse programa escravo recebe a mensagem do programa mestre, extrai o nome da máquina e envia a mensagem para o mestre com o nome do host.

```
#include "pvm3.h" /* Arquivo de cabeçalho para o PVM */
int main()
{
int ptid; /* Identificador do processo pai */
char buf[100]; /* String recebido na mensagem */
/* Identifica o processo pai. */
ptid = pvm_parent();
/* Cria um string com o nome da máquina atual. */
strcpy(buf, "hello, world from ");
gethostname(buf + strlen(buf), 64);
/* Prepara para enviar uma mensagem. */
pvm_initsend(PvmDataDefault);
/* Empacota o string na mensagem. */
pvm_pkstr(buf);
/* Envia a mensagem de volta ao processo pai. */
pvm_send(ptid, 1);
/* Encerra a execução do programa. */
pvm_exit();
exit(0);
}
```

4.8 Outros ambientes de passagem de mensagem

Será falado brevemente sobre outros ambientes de mensagem mais utilizado com MPI, P4.

4.8.1 P4

O sistema P4 desenvolvido no *Argonne National Laboratory* foi a primeira tentativa de desenvolvimento de uma plataforma portátil. Seu projeto começou a ser elaborado em 1984 (chamado na época de Monmacs) e a partir da sua terceira geração (em 1989) ele foi reescrito com o objetivo de produzir um sistema mais robusto para as necessidades recentes de passagem de mensagens entre máquinas heterogêneas. (BUTLER. 1994)

A principal característica do P4 é a possibilidade de ser utilizado por múltiplos modelos de computação paralela, podendo ser empregado em arquiteturas MIMD com memória distribuída, com memória compartilhada e em *clusters* onde há uma coleção de máquinas com ambos os tipos de memória.

Para as arquiteturas MIMD com memória compartilhada ele fornece o modelo de monitores para coordenar o acesso aos dados compartilhados.

Para as arquiteturas MIMD com memória distribuída o sistema P4 fornece funções para enviá-la e receber mensagens. Essas operações são bloqueantes, sendo isso uma grande restrição do P4.

As mensagens podem ser enviadas para plataformas heterogêneas. XDR é usado para traduzir as mensagens entre máquinas com diferentes formatos de dados. Devido à necessidade de desempenho, essa tradução é feita somente quando é absolutamente necessária.

4.8.2 MPI

Message Passing Interface consiste em uma tentativa de padronização, independente de plataforma paralela, para ambientes de programação via troca de mensagens. O MPI surgiu da necessidade de se resolver alguns problemas relacionados às plataformas de portabilidade, tais como o grande número de plataformas existentes ocasiona restrições em relação à real portabilidade de programas e ao mau aproveitamento de características de algumas arquiteturas paralelas.

4.8.2.1 Especificação do MPI

O M.P.I. define um conjunto de 129 rotinas, que oferecem os seguintes serviços: (WALKER 1994)

Comunicação Ponto-a-Ponto e Coletiva: O M.P.I. implementa diversos tipos de comunicação.

Suporte para Grupos de Processos: O MPI relaciona os processos em grupos, e esses processos são identificados pela sua classificação dentro desse grupo, a partir de zero. Essa classificação dentro do grupo é denominada *rank*. O M.P.I. apresenta primitivas de criação e destruição de grupos de processos. Então, um processo no MPI é identificado por um grupo e por um *rank* dentro deste grupo.

Suporte para Contextos de Comunicação: Contextos podem ser definidos como escopos que relacionam um determinado grupo de processos. Esses tipos de instâncias são implementadas com o intuito de garantir que não existam mensagens que sejam recebidas ambiguamente por grupos de processos não relacionados. Então, um grupo de processos ligados por um contexto não consegue comunicar-se com um grupo que esteja definido em outro contexto. Esse tipo de estrutura não é visível nem controlável pelo usuário, e o seu gerenciamento fica a cargo do sistema.

Suporte para Topologias: O MPI fornece primitivas que permitem ao programador definir a estrutura topológica que descreve o relacionamento entre processos. Como exemplo de uma topologia, pode-se citar uma malha, onde cada ponto de intersecção na malha corresponde a um processo.

4.8.2.2 Características do MPI

O padrão M.P.I. apresenta as seguintes características: (WALKER 1994) (SNIR 1996)

Eficiência - foi cuidadosamente projetado para executar eficiente em máquinas diferentes. Especifica somente o funcionamento lógico das operações. Deixa em aberto a implementação. Os desenvolvedores otimizam o código usando características específicas de cada máquina.

Facilidade - Define uma interface não muito diferente dos padrões P.V.M., Express, P4, etc, e acrescenta algumas extensões que permitem maior flexibilidade.

Portabilidade - É compatível com sistemas de memória distribuída, shared-memory, NOWs (network of workstations) e uma combinação deles.

Transparência - Permite que um programa seja executado em sistemas heterogêneos sem mudanças significativas.

Segurança - Provê uma interface de comunicação confiável. O usuário não precisa se preocupar com falhas na comunicação.

Escalabilidade - O M.P.I. permite crescimento em escala sob diversas formas, por exemplo, uma aplicação pode criar subgrupos de processos que permitem operações de comunicação coletiva para melhorar o alcance dos processos.

CAPÍTULO 5

Montagem do projeto

Neste capítulo serão descritas as ferramentas para a montagem da máquina virtual bem como descrever com foi montado esse ambiente

5.1 Ferramentas utilizadas no projeto

Para implementar a máquina virtual do projeto, foram necessários quatro computadores pessoais, placas de redes, cabos, *switch* de rede, *switch* KVM, *software* de domínio público (Linux e Pacote PVM).

5.1.1 Hardware

Para montar a rede e a estrutura para programar foram adquiridos, além dos computadores, alguns *hardwares* para comunicação entre as máquinas e equipamentos para facilitar no manuseio e instalação de sistema operacional, como *switch* de rede, cabos, RJ-45, *switch* KVM.

5.1.1.1 Switch KVM

O *switch* KVM ou *Server Switch* permite comandar diversos computadores com apenas um conjunto de teclado, monitor e mouse. A alternância entre os PCs pode ser feita com um simples toque de um botão no equipamento ou com o próprio teclado, eliminando problemas com mudança de cabos e economizando espaço e energia. Com ele pode-se controlar todos os computadores da PVM a partir de um único conjunto de teclado, monitor e mouse.

As características do Server Switch são:

- Chaveador KVM 4 Portas em tamanho compacto;
- Suporte a todos os tipos de mouse;

- Suporta DOS, Windows 3.x/95/98/ME/NT/2000/XP, Netware, Unix, Linux e mais;
- Facilidade Hot-Plug, adiciona ou remove PCs conectados para manutenção sem desligar o chaveador KVM ou outros PCs;
- Suporta Vídeo de Alta Qualidade, até 1920 x 1440 com banda passante de 200MHz;
- Não requer *software*, seleção simples de PC via Push Button e Hot-Keys;
- Modo Auto-Scan para monitoração de PCs;
- *Status* do teclado restaurado ao se chavear entre PCs;
- *Led* indicador para fácil monitoração do *status*;
- Som de *bip* para confirmação de troca de porta.

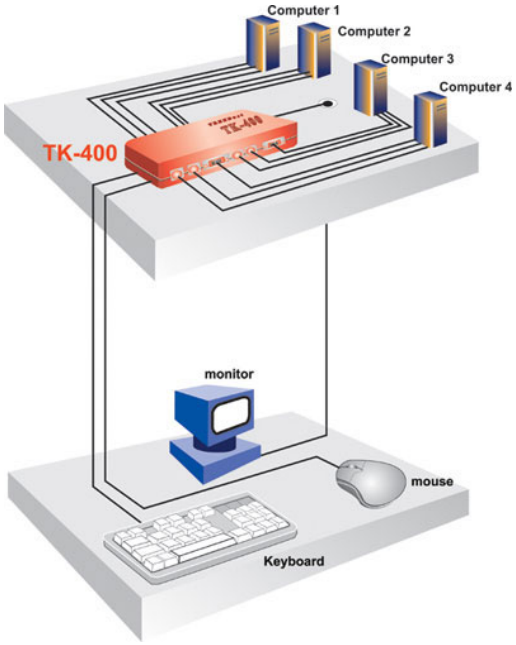

Figura 5–1 Esquema do Switch KVM

Figura 5–2 Switch KVM TK-400 da TRENDnet

5.1.1.2 Switch

O *switch* é um aparelho muito semelhante ao *hub*, mas tem uma grande diferença: os dados vindos do computador de origem somente são repassados ao computador de destino. Isso porque os *switchs* criam uma espécie de canal de comunicação exclusiva entre a origem e o destino. Dessa forma, a rede não fica "presa" a um único computador no envio de informações. Isso **aumenta o desempenho** da rede, já que a comunicação está sempre disponível, exceto quando dois ou mais computadores tentam enviar dados simultaneamente à mesma máquina. Essa característica também diminui a ocorrência de erros (colisões de pacotes, por exemplo).

Assim como no *hub*, é possível ter várias portas em um *switch* e a quantidade varia da mesma forma. O *hub* está cada vez mais em desuso. Isso porque existe um dispositivo chamado *hub switch*, que possui preço parecido com o de um *hub* convencional. Trata-se de um tipo de *switch* econômico, geralmente usado para redes com até 24 computadores.

Característica do Switch D-Link DES 1008D

O *switch* DES-1008D é um *switch* de alto rendimento e grande versatilidade. É desenhado para reforçar o rendimento do Soho e a pequenas empresas, proporcionando flexibilidade e manejo a 10/100Mbps. Possui ainda 8 portas *autodetect*, o que permite aos grupos de trabalho aumentar o rendimento em rede.

Este *switch* possui 8 portas com suporte para padrão Nway. As portas têm a capacidade de negociar as velocidades de rede entre 10BASE-T e 100BASE-TX, como também o modo de operação em Half ou Full Duplex.

A arquitetura de *Parallel Switching* para o modo de operação *Store & Forward*, permite a transferência de dados de forma direta entre as diferentes portas, com Full Error Checking, eliminando o tráfego da rede o envio de pacotes incompletos, fragmentados ou com erros de CRC, assegurando desta forma a integridade dos dados.

Portas	8 (10/100Base-TX)
Padrões	IEEE 802.3 10Base-T Ethernet Repeater, IEEE 802u 100Base-TX class II Fast Ethernet repeater e ANSI/IEEE Std 802.3 Nway autonegociação
Conectores	RJ-45
Transferência	10/100Mbps Full Duplex, autodetect
Método de acesso	CSMA/CD
Método de transmissão	Store & forward
Topologia	Estrela
Filtering Address Table	8K por dispositivo
Packet Filtering/Forwarding Rates	148.800pps por porta (em full duplex)
Leds indicadores	Por porta: link/activity, velocidade 100Mbps, Full duplex collision. Por switch : Power
Consumo	8W Máximo Modelo Rev. C2 2W Máximo Modelo Rev. D1

5.1.1.3 Computadores

Optou-se no projeto por nomes fáceis para facilitar na hora de programar.

A tabela a seguir dar alguns detalhes das máquinas usadas no projeto.

Descrição	Mestre	Escravo	Casa	Slaver
Processador	700 Mhz	400 Mhz	1100 Mhz	1250 Mhz
Memória	256 MB	128 MB	256 MB	256 MB
HD	8 GB	5GB	20GB	40GB
Placa de red	Onboard	Off board	Onboard	Onboard
Placa de vídeo	Onboard	Off board	Onboard	Onboard

Ponto importante é que todas as máquinas na rede tenham CD-Rom para instalação do sistema operacional e precisam de placa de rede para poder se comunicar com outras máquinas.

5.1.2 Software

Neste projeto usou-se como sistema operacional o Linux Red Hat 9, a linguagem C como padrão e o pacote PVM.

5.1.2.1 Linux Red Hat 9

O Unix é um dos sistemas operacionais mais populares disponíveis hoje em dia. Um dos motivos desta popularidade é o grande número de arquiteturas que rodam alguma versão de Unix. Ele foi originalmente desenvolvido na década de 1970, nos laboratórios Bell, como um sistema multitarefa para minicomputadores e computadores de grande porte.

O sistema operacional Linux é uma implementação independente da especificação para sistemas operacionais Posix, com extensões System V e BSD. O Linux é distribuído gratuitamente nos termos da Licença Pública GNU (GNU General Public Licence), sendo que não existe código de propriedade em seu pacote de distribuição. Ele funciona no padrão IBM PCs e é compatível com processador I386, mas existem também implementações para outras arquiteturas de *hardware*.

Hoje em dia o Linux é um clone completo do Unix, capaz de rodar *x* Windows, TCP/IP, Emacs, UUCP, mail e muitos outros *softwares* consagrados. Quase todos os programas de domínio público têm sido portados para Linux. *softwares* comercias também têm recebido versões para Linux.

5.1.2.2 Linguagem C

A linguagem C foi criada por Dennis M. Ritchie e Ken Thompson no laboratório Bell em 1972, baseada na linguagem B de Thompson, que era uma evolução da antiga BCPL.

O C é uma linguagem de alto nível com uma sintaxe bastante estruturada e flexível, tornando sua programação bastante simplificada. Programas em C são compilados, gerando programas executáveis.

O C compartilha recursos tanto de alto quanto de baixo nível, pois permite acesso e programação direta do microprocessador. Com isto, rotinas cuja dependência do tempo é crítica podem ser facilmente implementadas, usando instruções em Assembly.

No início da década de 1980, a linguagem C é padronizada pelo American National Standard Institute: surge o Ansi C.

5.1.3 Ferramenta de tentativas frustradas

Aqui estão algumas ferramentas utilizadas que não funcionaram na implementação do projeto e esse é o principal motivo do completo atraso do projeto.

5.1.3.1 Sistema operacional

No início do projeto tentou-se usar como sistema operacional Microsoft Windows 2000 server para o Mestre e Windows 2000 Professional para as máquinas-nó. No entanto percebeu-se que não há como iniciar nem utilizar o pacote PVM para o

Windows 2000, somente Windows NT, mas não havia Licença do Windows NT Server nem do Windows NT Workstation disponível. Esse foi um dos principais motivos do abandono do sistema operacional Windows. Para o Windows 2000 conseguiu-se a licença do trabalho e outra ao adquirir a máquina. Não haveria problema algum com licenças para Windows 2000.

Na tentativa de funcionar o PVM no sistema operacional Windows foram utilizadas várias ferramentas para funcionar o serviço RSH, mas nenhuma funciona de forma apropriada. É muito difícil configurar esse serviço no Windows, já que ele não é nativo. Existem várias ferramentas que prometem fazer a mesma coisa semelhante que o serviço RSHsvc do UNIX, porém nenhuma é compatível para rodar perfeitamente para o funcionamento da máquina virtual para rodar o processo *daemon* PVMd

5.1.3.2 Compilador

Outra tentativa de rodar programas do PVM foi compilar o programa *daemon* PVMd para Windows. Faltavam inúmeras bibliotecas e arquivos *.h*, que são típicos do sistema operacional Linux como XDR, outros arquivos que só funcionam com Windows NT.

Nessas tentativas de compilar os programas PVM no sistema operacional Windows foram usados dois compiladores. O primeiro foi o DEVC++, que é um *software* gratuito e compatível com Windows e de fácil manipulação, mas não se conseguiu por falta de arquivo de cabeçalho. Foram copiados os arquivos de cabeçalho que faltavam do Linux para o Windows e não deu certo. Muitos desses arquivos são de uso exclusivo do sistema Unix. O mesmo erro aconteceu com outro compilador próprio da Microsoft, o Visual C++ 6.0. Foi tentado de várias formas e sempre faltavam os arquivos de cabeçalho e, quando se encontravam esses arquivos, eram de uso do sistema Unix e similares ou do Windows NT. Em quase todos os arquivos de cabeçalho de que

se necessitava havia uma instrução verificando se é Windows NT ou se é do Unix e seus compatíveis.

Foram desperdiçados mais de 2 meses do início do projeto para montar o PVM no sistema operacional Windows.

Outra tentativa que foi a utilização do Linux Mandrake, apesar de ser uma distribuição do sistema operacional Unix, não tem o pacote PVM para ser instalado pelo CD e por isso optou-se por Linux Red Hat 9, que já tem essa facilidade.

A instalação do PVM no Mandrake poderia ser feita manualmente baixando o pacote de instalação do site http://www.netlib.org/pvm3/.

5.2 Montagem do projeto

No início do projeto previa-se utilizar os computadores da faculdade como projeto-piloto de um projeto bem maior mas por questão de praticidade foi optado a montagem do projeto em casa.

5.2.1 Parte Física

Primeiro precisou-se adquirir uma máquina para começar o projeto. De início foi comprado usado um computador constando somente um gabinete, placa-mãe, HD e memória de 128Mb, com teclado e *mouse*.

Na busca dos outros monitor e teclado para os computadores, já que se pretendia fazer o projeto de uma rede completa de computadores. O projeto iria ocupar muito espaço e energia elétrica.

Poderia usar duas opções comprar um switch KVM ou controlar os computadores remotos usando algum programa de acesso remoto com VNC, Teminal Service e outro. Mas com esse programa geram uma carga a mais de processamento e uma pequena sobrecarga indesejada no *switch* de rede. A melhor opção é switch KVM

pelo fato de que não gera nenhum processamento a mais nas máquina para o controle remoto e não gera nenhuma trafego adicional sobre o switch de rede.

Definiu-se que seriam somente quatro máquinas no projeto e com isso seria necessário somente um *switch* KVM de quatro portas. Inicialmente, procurou-se um *switch* KVM mecânico de chave, mas depois de procurar em vários lugares diferentes e fóruns, foi constatado que esse *switch* mecânico não era muito confiável e tinha algumas deficiências, como:

- Não pode trocar rapidamente de uma máquina para outra;
- Não funciona para altas resoluções;
- Tem algumas falhas ao ponto de ter de reiniciar todas as máquinas para voltar ao normal.

Figura 5–3 Switch Mecânico

Depois de procurar, a melhor opção em custo-benefício foi o *switch* KVM da TRENDnet TK-400, pois era o mais barato à época e na compra do equipamento vêm todos os cabos e não necessita de alimentação externa. Para montar esse equipamento é só conectar os cabos na saída de monitor, teclado, mouse do computador no *switch* KVM. Para troca de uma máquina para outra, basta dar um comando no próprio *switch* e no teclado.

Figura 5–4 Switch KVM desmontado

Figura 5–5 KVM com teclado e mouse

Figura 5–6 Switch KVM Montado

Na compra de *switch* de rede inicialmente pensou-se somente em um *hub*; mas, como o projeto necessita de um equipamento que valoriza o desempenho, o *switch* era bem mais indicado, já que ele não trabalha com sistema de *broadcast*. A melhor escolha

foi o modelo D-link DES1008D, porque consta em alguns fóruns que tem um ótimo desempenho e a marca D-Link é de extrema confiança.

A montagem dos cabos de rede foi feita toda de modo manual com clipador e RJ45.

Figura 5–7 Switch de Rede desmontado

Figura 5–8 Switch de Rede montado

5.2.2 Parte lógica

A instalação do Red Hat é igual como qualquer outro sistema operacional. Há de se tomar o maior cuidado para não instalar serviços desnecessários. Não foi verificado se todos os serviços instalados em cada máquina são extremamente necessários para rodar o PVM, mas o mais importante é que não se pode esquecer de selecionar o pacote PVM na hora de instalar o Linux. O pacote fica na parte de aplicações > Engenharia e Ciência. É extremamente fácil achar manuais sobre instalação do Linux Red Hat.

5.2.2.1 Arquivos de configuração

Aqui está a listagem de alguns arquivos que precisam ser configurados para que o PVM funcione corretamente e que facilitam muito na hora de programar e

distribuir os arquivos para os escravo poderem executar. O programa escravo tem de estar na própria máquina escrava no diretório */usr/share/pvm3/bin/LINIXI386*.

- /etc/hosts

Este faz o relacionamento entre um nome de computador e endereço IP local. Recomendado para IPs constantemente acessados e para colocação de endereços de *virtual hosts* (quando deseja referir pelo nome em vez de IP). A inclusão de um computador neste arquivo dispensa a consulta de um **servidor de nomes (DNS)** para obter um endereço IP, sendo muito útil para máquinas que são acessadas freqüentemente.

- /etc/pam.d/rsh – Arquivo para configuração do serviço RSH

Este arquivo contém as configurações de permissões de execução e permissões de auditorias que o serviço RSHsvc irá requisitar na hora de um *host* remoto requisitar a execução de algum processo local. A configuração foi para não fazer auditoria e não requisitar nenhum tipo de permissão, para evitar possíveis transtornos na elaboração do projeto.

- /etc/hosts.equiv – Arquivo de configuração do servidor NFS[10]

Esse arquivo serve para configurar o servidor NFS. Colocando essas linhas de comando, fala para o computador que todos os computadores dessa listagem são confiáveis e podem ser usados como se fossem um diretório comum entre eles. O diretório comum é */home* da máquina mestre.

[10] NFS Network File System. Sistema de arquivo de rede para sistema operacional Unix que possibilite que todas as máquinas da rede tenham como um diretório padrão para troca de arquivos em comum.

- /etc/exports – Arquivo de configuração do servidor NFS

Esse arquivo é para configurar qual o diretório em comum entre todas as máquinas, quem são os *hosts* que poderão utilizar esse diretório em comum e que tipo de permissão esses *hosts* terão. Pode utilizar permissão de somente leitura, ou de leitura e escrita. A configuração tem de ser feita no servidor NFS, que no caso é o *host* mestre.

Então, inserindo essa linha:

/home/ casa(rw,sync, no_root_squash)

/home/ escravo(rw,sync, no_root_squash)

/home/ slaver(rw,sync, no_root_squash)

- /etc/fstab – Arquivo de configuração do cliente NFS

Esse arquivo fala que toda vez que o computador inicia irá montar uma conexão no servidor NFS, que é o *host* de nome mestre. Essa é a linha que deve ser adicionada no arquivo */etc/fstab*.

mestre:/home /home nfs

Por vezes essa configuração não funciona, então pode-se digitar o comando para montar a partição NFS:

M*ount mestre:/home /home*

- /root/.bashrc – Arquivo de configuração de inicialização

Esse é o arquivo de *login* do sistema operacional Linux. Qualquer comando que se coloque aqui será executado quando o usuário *root* "logar" na máquina. É nele que se configuram as variáveis do sistema, as variáveis do PVM.

As seguintes linhas de comando devem ser inseridas para configurar as variáveis do PVM:

export PVM_ROOT=/usr/share/pvm3

export PVM_ARCH=LINUXI386

export PVM_TMP=$PVM_ROOT/TMP

Outra configuração do arquivo *.basrc* é para que o PVM inicie automaticamente. Para fazer isso, basta colocar essa linha de comando:

deletehost hostname

Para que a configuração funcione, o PVM mestre já tem de estar iniciado. Essa configuração é para um nó sair da máquina virtual quando o usuário começa a utilizar a máquina.

- /root/.bash_logout – Arquivo de configuração logout

Esse arquivo serve para configura o que será executado quando o usuário efetua logout.

Para adicionar uma máquina no PVM para poder calcular é necessário colocar essa linha de execução nesse arquivo.

Rsh adcionahost hostname

5.3 Programa integral

O programa integral consiste calcular a integral de uma função dependente somente de uma variável usado a técnica de áreas de trapézios.

Como exemplo de função que irá roda $\int_0^{40} sen2x + \cos x^3 dx$. Mas no entanto pode-se calcular a integral de qualquer função de uma variável. E a faixa de integração é de 0 a 40. Da mesma forma que a função pode ser mudada esses intervalos de acordo com as necessidades.

Para calcular a integral da função foi utilizada a técnica de áreas de trapézios de acordo com a Figura 5–9. A figura é só um ilustrativo de como são os trapézios.

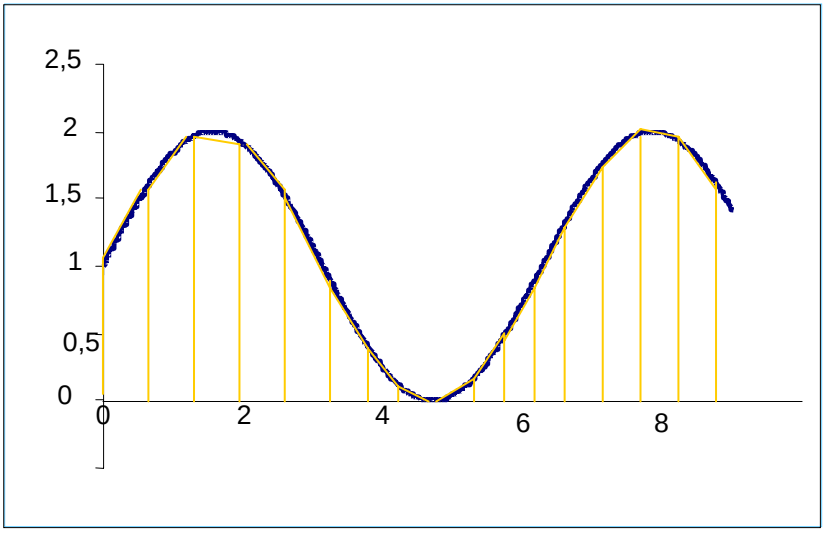

Figura 5–9 Integral por área de trapézios

A técnica de áreas de trapézio precisa de um delta que é a altura do trapézio Figura 5–10. No programa integral ele é de 0,0000001

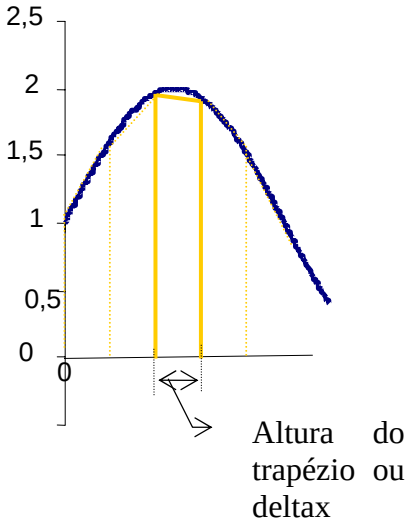

Figura 5–10 Detalhes da técnica do trapézio

5.4 Desenvolvimento de algoritmo

A arquitetura paralela do projeto é SIMD. Como foi falado na seção 2.3.1, essa arquitetura trabalha com a idéia de mestre e escravo e cada escravo recebe o mesmo programa, mas cada um recebe faixa distinta de trabalho. No projeto todas as máquinas

recebem o programa escravo que está no anexo B, inclusive o mestre, pois está configurado para trabalhar para processar junto com os escravos.

No projeto foi utilizado o sistema de memória distribuída, pois cada nó tem a sua própria faixa de dados para executar a integral. Quando o escravo retorna com a resposta da área de integração, o mestre manipula e guarda a resposta na memória.

A ordem de execução do programa é a seguinte da parte mestre:

1) Mestre inicia o PVMd, que é o programa base;
2) Os nós são adicionados na máquina virtual;
3) Inicia o programa mestre;
5) Programa mestre recolhe informações dos escravos como Tid e nome do *host*;
6) Envia o programa escravo para todos os nós;
7) Verifica qual é o escravo para enviar a faixa de integração correta;
8) Empacota a mensagem;
9) Envia a mensagem;
10) Espera pela resposta do escravo
11) Recebe a mensagem proveniente do escravo;
12) Desempacota a mensagem proveniente do escravo;
13) Faz o somatório das áreas de integração;
14) Mostra o resultado.

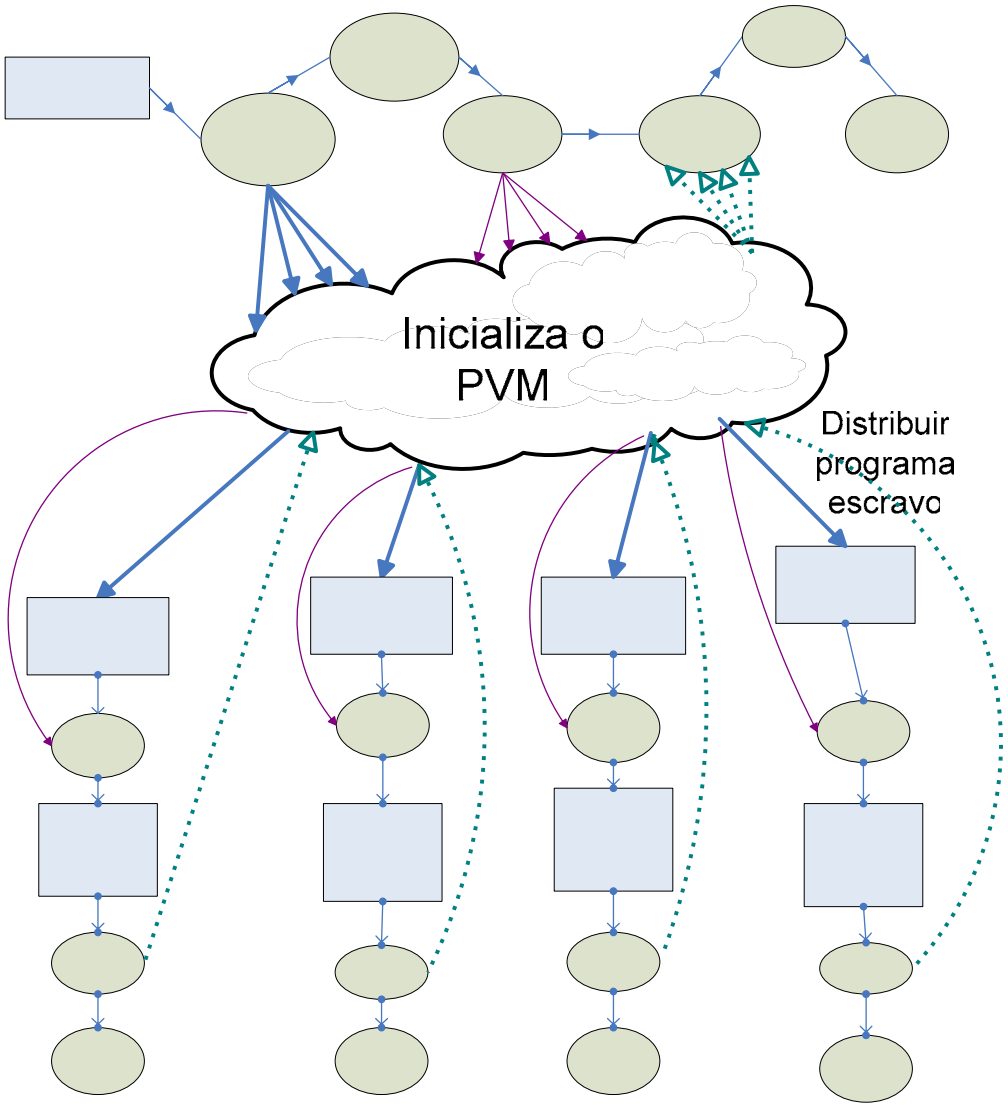

Figura 5–11 Fluxo de processos

5.4.1 Estilo de paralelismo escolhido

Neste projeto resolveram-se mesclar dois estilos para programação paralela o *processor farm*, e paralelismo geométrico.

Como foi dito na sessão 3.3.1, é um paralelismo geométrico mais fácil de se implementar. E que melhor se adapta ao programa de integral. Pois cada escravo fica responsável por uma faixa de integração.

No entanto não foi somente o estilo de paralelismo geométrico utilizado no projeto. Outro estilo de paralelismo usado é o *processor farm*, que é descrito na seção 3.3.2. Esse estilo utiliza a técnica mestre e escravo. No programa de integral, qualquer

máquina pode ser mestre, basta executar o programa mestre, que se encarrega de encaminhar os intervalos de integração para os nós escravos. O mestre pode ser escravo também, mas como padronização o *host* de nome mestre será quem executar o programa mestre.

Essa é a forma mais adequada para o programa integral, pois, se todo o processo fosse geométrico, não haveria como teoricamente calcular o somatório das áreas da integral; mas, se fosse puro *processor farm*, cada máquina da rede faria uma operação matemática e isso não faria sentido, pois haveria uma enorme quantidade de tráfego na rede e a granularidade ficaria muito fina, havendo uma alta taxa de transferência de dados entre o mestre e o escravo. O retardamento da rede aumentaria muito o tempo de execução do cálculo.

5.4.2 Balanceamento de cargas

No exemplo de integral do projeto, todos os nós receberão o mesmo programa escravo para fazer o cálculo paralelo, mas com uma peculiaridade: cada nó receberá uma faixa de cálculo de integral proporcional à velocidade do processador. No projeto optou-se por colocar como padrão de distribuição de faixa de integração a velocidade do processador. No entanto pode ser implementado qualquer outro algoritmo de divisão de tarefas.

As tarefas são previamente divididas em:

Host	Processador (MHz)	Intervalos de integração
Casa	1100	13,52112676056... a 25,91549295774…
Mestre	700	0 a 7,887323943661…
Escravo	400	7,887323943661… a 13,52112676056...
Slaver	1250	25,91549295774… a 40

Com a divisão entre parte proporcional e velocidade do processador, caracteriza-se o balanceamento de carga. A biblioteca PVM não faz balanceamento de carga automático, somente disponibiliza as ferramentas para que o programador faça esse balanceamento de cargas.

O balanceamento de carga pode ser tanto para dois computadores como para grandes quantidades. No entanto não há como testar o que acontece com o balanceamento de carga quando o número de escravos for muito grande.

Como foi dito na seção 4.6, uma das limitações do PVM é que a quantidade de nós dependerá da quantidade de conexões simultâneas que o sistema operacional suportar.

No projeto tentou-se usar uma nova abordagem para balanceamento de carga mas não houve uma melhoria, pois foi distribuído inversamente proporcional ao tempo de execução do programa integral serial dá seguinte forma:

Computador	Processador (MHz)	Tempo de execução serial Te	Velocidade relativas (Vr)
Escravo	400	929.06s	23376
Mestre	700	492.64s	44084
Casa	1100	397.74s	54602
Slaver	1250	352.31s	61643
		Ste=2171,75s	

A velocidade relativa é dada pelo somatório de todos os tempos de execução serial de todos os nós dividido pelos seus respectivos tempos de execução.

Vr = (Ste/Te)10000

Onde Vr é velocidade relativa

Ste é o somatório de tempo de execução

Te é o tempo de execução serial.

Noto-se como o comando top do UNIX que a máquina mestre estava divida entre o processamento do programa mestre e do programa escravo. E percebi-se que

metade do processamento esta reservado para o programa mestre e outra metade para o programa escravo. Então é mais lógico dar um intervalo menor de integração para a máquina mestre calcular.

Computador	Processador (MHz)	Velocidade relativas
Escravo	400	400
Mestre	700	350
Casa	1100	1100
Slaver	1250	1250

5.4.3 Autonomia do escravo

Para dar uma ligeira autonomia para os nós escravos da máquina virtual foi feito um pequeno programa de configuração automática do PVM, que é adicionar e retirar o nó escravo para a máquina virtual usando *script* de *login* e *logoff*.

O código do programa de adição e remoção de uma máquina da máquina virtual é:

```
//programa para adicionar a maquina qualquer máquina pvm.
       #include "pvm3.h"
       #include <stdio.h>
       #include <stdlib.h>
       int main (int argc, char *argv[])
       {
       int info, infos[1];

       if (argc != 2)
       {    printf("Digite o nome do host");
            exit(1);
       }

       static char *hosts[] = {""};
       *hosts = argv[1];
       info                 = pvm_addhosts(hosts,1,infos);
       if (info < 1)
            {
            printf("erro, %d\n",info );
            }
       pvm_exit();
       //return();
       exit(1);
}}
```

```
//programa para remover qualquer maquina do pvm
       #include "pvm3.h"
       #include <stdio.h>
       #include <stdlib.h>
       int main (int argc, char *argv[])
       {
       int info, infos[1];
       if (argc != 2)
       {    printf("Digite o nome do host");
            exit(1);
       }
       static char *hosts[] = {""};
       *hosts = argv[1];
       info                 = pvm_delhosts(hosts,1,infos);
       if (info < 1)
            {
                 printf("erro, %d\n",info );
            }
       pvm_exit();
       //return();
       exit(1);
}
```

O escravo está habilitado para entrar e sair quando quiser, mas enquanto a máquina virtual estiver processando não há como um escravo sair, pois não está implementada a parte de tolerância a falha. Como foi dito na seção 4.5, o PVM não implementa por padrão a tolerância a falha, mas há funções de controle de processo para se fazer isso.

5.4.4 Topologia da rede

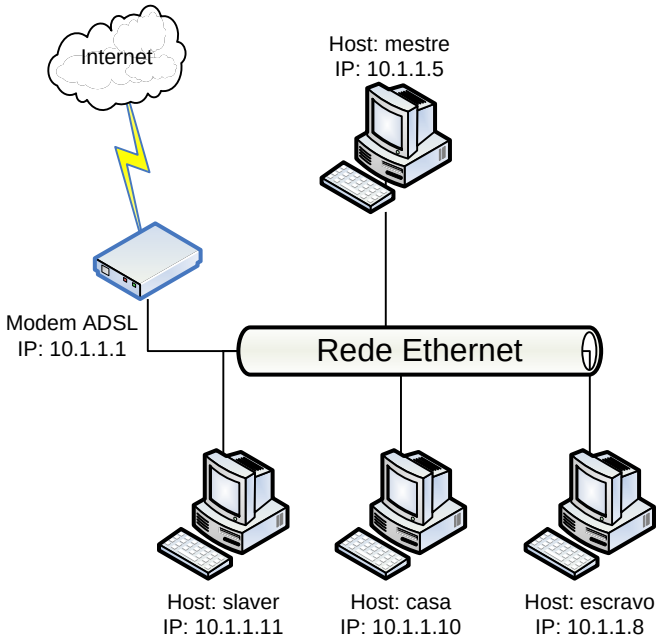

Figura 5–12 Topologia da rede

No projeto foi utilizada a distribuição de rede 10.0.0.0/8, pois é a faixa de IP com que o *modem* trabalha. O *modem* serve somente para acesso à Internet. Ele só tem essa funcionalidade no projeto.

5.5 Conhecimento obtido

Há uma pequena falha na paralelização do algoritmo paralelo que é a perda de algumas áreas de integração.

No programa seqüência o erro dessa imprecisão é aceitável pelo falto dela ocorre somente uma única vez. O erro do programa é não contabilizar com a área de integração entre a ultima interação e o final da integração.

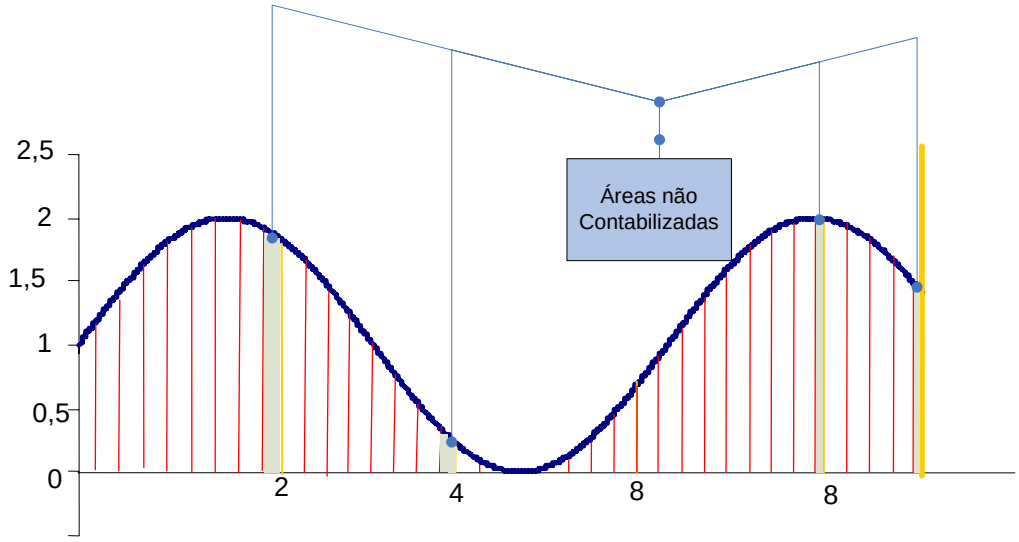

Figura 5–13 Erros na contabilização

No programa integral paralelo o erro ocorre *n* vez sendo que *n* é o número de escravos.

O erro é muito difícil de calcular pois vária muito de um intervalo para outro. A altura de trapézio que não se calcula varia de 0 a (deltax – deltax/10).

Onde deltax é a altura do trapézio do programa.

5.6 Resultados obtidos

Cada máquina do PVM fez o cálculo da integral de forma serial, sem dividir as tarefas. O resultado foi:

Tabela 6.1. Resultado do cálculo de modo serial

	Velocidade do processado MHz	Tempo de execução serial (s)

Slaver	1250	352.31
Casa	1100	397.74
Mestre	700	492.64
Escravo	400	929.06

O cálculo se faz de modo paralelo sem balanceamento de carga. O tempo de execução é 261,52s.

Quando há balanceamento dividindo a velocidade proporcionalmente entre as velocidades de processamento de cada máquina, o tempo é de 198,98s. Uma melhora de 31% com relação a execução do calculo sem balanceamento de carga.

Há uma melhoria considerável com o balanceamento de cargas, mas não há melhoria quando se usa um balanceamento de carga usando como base o tempo de execução do programa *integral_serial*.

Um resultado surpreendente com o tempo de execução de 141s quando se da um menor intervalo de integração para a máquina mestre. Tem uma melhoria de 85%.

E por ultimo, os erros toleráveis na programação seqüência torna-se erros graves quando se projeta um programa paralelo usando a abordagem de paralelização de algoritmo seqüencial existente.

CAPÍTULO 6

Considerações Finais

6.1 Conclusão

Com o balanceamento de cargas, a velocidade total tem uma melhoria de aproximadamente 85%. Não há como calcular a eficiência nem o *speedup* do processamento de acordo com a seção 2.3, pois os elementos processados têm velocidades diferentes.

Outra conclusão é que não se pode colocar qualquer máquina para processar sem um balanceamento de carga adequado pois, o tempo total de execução está sendo determinado pelo elemento de processador mais lento. A máquina mais lenta esta provocando um gargalo na execução do programa.

Em uma rede com número muito grande de máquinas e com grande variedade de processadores, é extremamente importante utilizar algum tipo de algoritmo de balanceamento de cargas, pois tudo pode ficar mais lento por conta de um nó lento.

Outro ponto importante: para se desenvolver algoritmos paralelos demanda muito tempo, como:

- Tempo de os programadores aprenderem o programa paralelo;
- Montagem do ambiente paralelo;
- Desenvolvimento do algoritmo;
- A própria execução do programa;

Há de se levar em conta todos esses pontos. Caso isso supere o tempo de montar um programa seqüencial, não vale a pena montar algoritmos paralelos.

Se um programa a precisão for mais importante que a velocidade de execução é recomendada à elaboração do programa de integral de forma seqüencial mas se a

velocidade é mais importante o programa integral pode ser paralelizado sem maiores danos.

6.2 Dificuldades do projeto

A primeira dificuldade já foi dita na seção 5.3 em ferramentas frustradas. Outra dificuldade for a utilizar o sistema operacional Linux em particular a compilação e programação do programa de integral, montagem do sistema de arquivos NFS, instalar e configurar interface de rede, iniciar e finalizar processos do Linux.

Relembrar a programação em C; relembrar conceito de cálculos numéricos; não ter conhecimento de programação paralela, pois foi preciso aprender ao longo do projeto; e informações desencontradas na Internet.

6.3 Sugestão para Futuros Trabalhos

1. Balanceamento de carga entre os nós de forma mais dinâmica. Quando um nó sai e entra no PVM, o sistema recalcula as cargas e redistribui entre os elementos de processamento.
2. Implementar a falha de tolerância e desenvolver um aplicativo que retire a máquina no meio do processamento sem prejuízo para os outros nós;
3. Implementação de balanceamento de cargas em redes maiores.

Referências Bibliográficas

ADMP Monografia Análise de desempenho em Passagem de mensagem.

ALMASI, G. S., Gottlieb A. *Highly Parallel Computing*. 2ª ed., The Benjamin Cummings Publishing Company, Inc., 1994.

ANDREWS, G. R., Schineider, F. B., *Concepts and Notations for Concurrent Programming*. ACM Computing Survey, 1983.

Browne, S. *Technology Evaluation and Notice #1: ScaLAPACK, PARPACK, and PVM*. *http://apolo.wes.army.mel/pet/CEWES/Reports/ten1.html*. Julho/1996.

BUTLER, R. M., Lusk, E. L., "Monitors, messages and clusters: The P4 parallel programming system", *Parallel Computing*,1994

CALÔNEGO JR., N. *Uma abordagem orientada a objeto de uma ferramenta de auxílio à programação paralela*. Tese (Doutorado), Outubro 1997

COULOURIS, G. F. *Distributed systems: concepts and design.* 3nd. ed. London: Addison-Wesley, 2001.

CSM.ORNL. *Site: http://www.csm.ornl.gov/pvm/pvm_home.html*. Acessado em março de 2005.

DUNCAN, R.. *A Survey of Parallel Computer Architectures*. IEEE Computer, pp.5-16, Fevereiro, 1990.

FLYNN, M. J. *Some Computer Organizations and Their Effectiveness*. IEEE Transactions on Computers, 1972.

GPACP. *Site: http://black.rc.unesp.br/gpacp/ link A-01* – Computação Paralela | ICMC – USP – São Carlos. Acessado em março de 2005.

HENRIQUE, Sergio – *Uma Introdução ao PVM. Site: http://www.dep.fem.unicamp.br/~sergio/pvm_intro.html*. Acessado em janeiro de 2005.

HWANG, K.; Briggs, F. A. *Computer Architecture and Parallel Processing*. McGraw-Hill International Editions, 1984.

KIRNER, C., *Arquiteturas de sistemas avançados de computação*. Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho, 1991.

KRUG, R. C.; Teodorowitsch R. Ambiente de Programação Paralela *Site: http://www.ulbra.tche.br/~roland/pub/rel-fin.pdf.* Acessado em março 2005.

MORSELLI ,*J C de Moraes Pesquisa:* Processamento Paralelo site: *http://www.inf.pucpcaldas.br/~morselli/* Acessado em Março de 2005

PVM: Parallel Virtual Machine. *A Users' Guide and Tutorial for Networked Parallel Computingsite*. Site: http://www.netlib.org/pvm3/book/pvm-book.html Acessado em Março de 2005.

QUINN, M.J. *Designing Efficient Algorithms for Parallel Computers*. McGraw Hill, 1987.

SNIR, M., et al, *MPI: The Complete Reference, The MIT Press*, 1996.

SNOW, C.R. *Concurrent Programming*. Cambridge University Press, 1992.

TANENBAUM, A. S. *Distributed Operating Systems*. Prentice Hall, 1995.

WALKER, D. W., *The design of a standard message passing interface for distributed memory concurrent computers, Parallel Computing*, 1994.

Anexo A Programa mestre com balanceamento de cargas

Código forte do programa paralelo com balanceamento de cargas

```
////////////////////////////////////////////////////////////////////////
// Centro Universitário de Brasília – UniCeub
// Projeto Final:
////////////////////////////////////////////////////////////////////////
// Orientador: Gustavo Gois
// Aluno: Robson Nakamura
////////////////////////////////////////////////////////////////////////
// Arquivo: pvm_mestre_load_balance.c
// Descrição: cálculo da integral através de algoritmo de processamento
//                 paralelo usando troca de mensagens
//                 Este é o processo master, o qual
//                 é responsável por criar e gerenciar os processos escravos
//                 e fazer o balanceamento de cargas entre os escravos
////////////////////////////////////////////////////////////////////////
#include "integral_master.h"
void main ()
{
        int nTid;
        pvm_config(&num_host, &narch, &hostp);
        struct timeb tmbInicio, tmbFim;
        int i;
        double dAreaTotal = 0.0;
        double dInicio, dFim;
```

```
        double dAreaEscravo[num_host];
        // Inicio do processamento
        ftime(&tmbInicio);
        // (pvm) Numero do processo (tarefa) pvm corrente
        nTid = pvm_mytid();
        printf("\n numeros escravos de %i",num_host);
        /*struct pvmhostinfo {
        int hi_tid;
        char *hi_name;
        char *hi_arch;
        int hi_speed;
        };*/
    //partindo do pressuposto de que todos os nós estão adicionados no PVM
    ////////////////////////////////////////////////////////////////////
    //enviar programa escravo para os escravos
    ////////////////////////////////////////////////////////////////////
    int nTarefas;
    for (i = 0 ; i < num_host; i++)
    {
        nTarefas = pvm_spawn(PROGRAMA_ESCRAVO, NULL,
PvmTaskHost, hostp[i].hi_name,1,&vecTids[i]);
    if (nTarefas = = 1)
    {
        printf ("\n números de tarefas %d, vetor %d ",nTarefas,vecTids[i]);
        }else printf("\nErro na inicialização do escravo %s ",hostp[i].hi_name);
```

```
        pvm_exit();
        exit(1);
}
////////////////////////////////////////////////////////////////
//calcular o total de velocidade para o balanceamento de carga///
////////////////////////////////////////////////////////////////
for (i = 0; i < num_host; i++)
{
        speed[i] = (double)hostp[i].hi_speed;
        printf("\n %f\n" ,speed[i]);
        SomaSpeed += speed[i];
}
printf("\nSoma total da velocidade é de %d \n",SomaSpeed);
////////////////////////////////////////////////////////////////
//printf("\n %f teste",(INT_FINAL – INT_INICIAL));
enviar_mensagem_escravos();
receber_mensagem_escravos(dAreaEscravo);
ftime(&tmbFim);
        mostrar_resultados(tmbInicio, tmbFim, dAreaEscravo);
// (pvm) Finalizar este processo (tarefa) pvm
        pvm_exit();
        return;
}
////////////////////////////////////////////////////////////////
//////////funções
```

```
//////////////////////////////////////////////////////////////////
int enviar_mensagem_escravos()
{       double dIntFinal, dIntInicial, dDelta_x;
        int nEscravo; //controlador de número de escravos
        // Dividindo o intervalo de integração pelo número de escravos
// Distribuindo os intervalos para os escravos
        dIntFinal = INT_INICIAL;
        for (nEscravo = 0; nEscravo < num_host; nEscravo++)
{
                // Determinando o fim do intervalo parcial
                //str = hostp[nEscravo].hi_name;
                dDelta_x = DELTA_X;
                dIntInicial = dIntFinal;
                dIntFinal = dIntFinal +( (speed[nEscravo] /SomaSpeed) *
(INT_FINAL – INT_INICIAL));
                printf ("Área inicial %f, Área final %f para o host %s\n
\n",dIntInicial , dIntFinal, hostp[nEscravo].hi_name);
                if (dIntFinal > INT_FINAL)
                {dIntFinal = INT_FINAL;}
                pvm_initsend (PvmDataDefault); // (pvm) Limpar o buffer de
transmissão
// (pvm) Empacotar N_Dados do tipo double no buffer de transmissão
                pvm_pkdouble (&dIntInicial, N_DADOS, 1);
                pvm_pkdouble (&dIntFinal, N_DADOS, 1);
                pvm_pkdouble (&dDelta_x, N_DADOS, 1);
```

```
// (pvm) Enviar a mensagem corrente para o escravo (tarefa) em questão
                pvm_send (vecTids[nEscravo], TAG_MASTER_SEND);
                }
                if (dIntFinal != (INT_FINAL – INT_INICIAL))
        {
                printf("Erro no balanceamento de cargas %f",dIntFinal);
                exit;
                pvm_exit();
        };
                return(1);
        } //final da função
        void receber_mensagem_escravos(double *dAreaEscravo)
        {
                int i, j, nBuffId, nBuffSize, nMsgTag, nTidEscravo;
                double dArea;
                for (i = 0; i < num_host; i++)
                {
        // (pvm) Loop (não bloqueante) de verificação da chegada de mensagens das
tarefas escravas
                do
                {
                        nBuffId = pvm_probe (-1, TAG_SLAVE_SEND);
                } while (!nBuffId);
                // (pvm) Obtendo informações sobre o buffer recebido
                pvm_bufinfo (nBuffId, &nBuffSize, &nMsgTag, &nTidEscravo);
```

```
                    // Encontrando a posição do Tid da tarefa escrava no vetor de
Tid's
                    for (j = 0; vecTids[j] != nTidEscravo; j++);
                    // (pvm) Recebendo as informações da tarefa escrava
                    pvm_recv(vecTids[j], TAG_SLAVE_SEND);
                    // (pvm) Extraindo um número double do buffer recebido e o
colocando no vetor de áreas
                    pvm_upkdouble (&dArea, N_DADOS, 1);
                    // Armazenar a área do escravo em questão
                    dAreaEscravo[j] = dArea;
               }
               return;
          }
          void mostrar_resultados(struct timeb tmbInicio, struct timeb tmbFim, double
*dAreaEscravo)
          {    // Mostrar resultados na tela
               int i;
               double dAreaTotal = 0.0;
               double dInicio, dFim;
               dInicio = (double) tmbInicio.time + (double) tmbInicio.millitm / 1000;
               dFim = (double) tmbFim.time + (double) tmbFim.millitm / 1000;
               printf("**************************************************\
n");
               printf("********** CÁLCULO DA INTEGRAL PELO MÉTODO
PVM\n");
```

```
                printf("*****************************************************\n\
n");
                // Imprimir informações dos escravos
                for (i = 0; i < NUM_ESCRAVOS; i++)
                {
                        // Calcular a área total
                        dAreaTotal += dAreaEscravo[i];
                        printf("O ESCRAVO%s CALCULOU A AREA: %6.5f \n",
hostp[i].hi_name, dAreaEscravo[i]);
        }
                // Imprimir informações do master
                printf("\nO MASTER CONTABILIZOU UMA AREA TOTAL DE:
%6.5f\n", dAreaTotal);
                printf("TEMPO TOTAL DE EXECUÇÃO: %6.2f SEGUNDO(S)\n\n",
dFim – dInicio);
                printf("*****************************************************\
n");
                printf("*****************************************************\n\
n");
                return;
        } //fim da função de envio
```

Anexo B Programa escravo

O programa escravo foi quase que uma copia do programa escravo do projeto final de conclusão do curso de três alunos da Universidade católica de Brasília.

Todos os créditos do programa escravo devem a eles.

```
//////////////////////////////////////////////////////////////////////////
// Centro Universitário de Brasília – UniCeub
// Projeto Final:
//////////////////////////////////////////////////////////////////////////
// Orientador: Gustavo Gois
// Aluno: Robson Nakamura
//////////////////////////////////////////////////////////////////////////
// Arquivo: pvm_mestre_load_balance.c
// Descrição: cálculo da integral através de algoritmo de processamento
// Foi usado quase que uma cópia do programa escravo do projeto final
// Projeto Final: Analise de Desempenho de Passagem de Mensagens.
// Descricao:  implementacao do escravo na arquitetura PVM (Parallel
//             Virtual Machine). Responsavel pelo calculo da integral,
//             utilizando os limites de integracao e a variacao delta_x
//             passados pelo processo master.
//////////////////////////////////////////////////////////////////////////
#include "integral_slave.h"


void main ()
{
        int nTid;
```

```
time_t tmInicio, tmFim;

// Inicio do processamento
tmInicio = time(NULL);

// (pvm) Numero do processo (tarefa) pvm corrente
nTid = pvm_mytid();

// (pvm) Numero do processo parente (master).
nParentId = pvm_parent();

// Recebendo mensagens (parametros) do processo master.
if (receber_mensagens_master() == 0) {exit(0);}

// Calcular a integral (area dos trapezios).
dArea = calcula_integral(dInicio, dFim, dDelta_x);

//  Enviar mensagens (resultados) para o processo master.
enviar_mensagens_master();

// Fim do processamento
tmFim = time(NULL);

// (pvm) Finilizar este processo (tarefa) pvm
pvm_exit();
```

```
        return;
    }

    int receber_mensagens_master()
    {
        // (pvm) Espera (bloqueante) ate que uma mensagem do master (nParentId) chegue
        pvm_recv (nParentId, TAG_MASTER_SEND);

        // (pvm) Extrai um numero double do buffer de recepcao e o coloca no endereco apontado
        pvm_upkdouble (&dInicio, N_DADOS, 1);
        pvm_upkdouble (&dFim, N_DADOS, 1);
        pvm_upkdouble (&dDelta_x, N_DADOS, 1);

        return(1);
    }

    void enviar_mensagens_master ()
    {
        // (pvm) Limpar o buffer de transmissao
        pvm_initsend (PvmDataDefault);

        // (pvm) Empacotando os dados (area calculada) no buffer de transmissao
```

```
            pvm_pkdouble (&dArea, N_DADOS, 1);


            // (pvm) Enviando os dados (area calculada) para o processo master
(nParentId)
            pvm_send(nParentId, TAG_SLAVE_SEND);
        }
```