LA FEDERAL DE ENGENHARIA DE ITAJUBA

Uma Proposta de Paralelismo Aplicada à Análise de Contingência

CARLOS BERNARDES ROSA JUNIOR

ITAJUBA-MG
Fevereiro de 2001

Escola Federal de Engenharia de Itajubá

Uma Proposta de Paralelismo Aplicada à Análise de Contingência

Carlos Bernardes Rosa Junior

EFEI

Uma Proposta de Paralelismo Aplicada à Análise de Contingência

Carlos Bernardes Rosa Junior

Orientador: Prof. Doutor Otávio Augusto Salgado Carpinteiro

Co-orientador : Prof. Doutor Antônio Carlos Zambroni de Souza

Dissertação apresentada à Escola Federal de Engenharia de Itajubá, como parte dos requisitos para a obtenção do título de mestre em Ciências – Área: Engenharia Elétrica.

EFEI – Fevereiro/2001

À Aline e Ana Laura, pelo apoio e acima de tudo pela compreensão.

Agradecimentos

Em uma certa ordem cronológica, gostaria de agradecer aquelas pessoas que, passando pela minha vida, tornaram possível este trabalho. A todos eles, de coração, o meu muito obrigado.

A Deus, cheio de bondade e equilíbrio, que me deu saúde e força de vontade para vencer todos os obstáculos do caminho.

Ao meu pai Carlos que, com suas sábias palavras, sempre me conduziu para o caminho correto. Hoje não sou engenheiro como você sonhava, mas posso lhe dar um prazer maior: sou mestre na arte que você será, sempre, o meu doutor.

À minha mãe Sueli, por toda a dedicação, carinho e amor que sempre transbordaram de seu coração, iluminando a todos nós.

À minha irmã Raquel pela compreensão e por todo o carinho.

À Profª. Vânia Hostalácio, minha primeira professora, pelo carinho.

À Profª. Iêda Canto, responsável por minha alfabetização.

À Profª Ivonete Coelho, que sempre chamava a minha atenção, no nível médio, sobre a importância da educação.

Ao meu treinador de Voley-ball Rubens Paiva Cardoso (Pavão), que sempre dizia: "meu atleta tem que ser bom de bola e bom de escola".

À minha querida tia Carlinda, uma segunda mãe.

Ao Prof. José Carlos Veloso, amigo e mestre, grande incentivador de minha graduação.

Ao Prof. José Ivo, amigo e eterno professor, pela sua presença e tão sábias respostas às tantas perguntas que sempre faço.

Ao Prof. Ronan Vaz de Campos, mestre e conselheiro que sempre me deu grandes oportunidades. Obrigado pela confiança.

Aos meus sogros Anair e Magda, pai e mãe. Seria impossível a realização deste trabalho sem o apoio incansável dos dois.

Ao Prof. Marco Antonio de Sousa Leão, amigo particular e diretor da Faculdade de Filosofia, que, acreditando em mim, abriu as portas para que eu pudesse fazer este trabalho.

Ao Prof. Msc. Alexandre Pimenta e Profª. Msc. Joice Lee Otsuka que tanto se empenharam para que tudo desse tão certo.

Ao companheiro Chrystian, pela disposição em me ajudar.

Ao Prof. Dr. Germano Lambert Torres, pela atenção dispensada sempre a tempo e a hora.

Ao amigo, Professor, Orientador e companheiro Otávio Augusto, por sempre acreditar em meu trabalho, e ainda ter tempo para me ouvir.

Ao Prof. Dr. Antônio Carlos Zambroni, meu co-orientador, pela paciência e, acima de tudo, pela credibilidade em mim depositada.

Ao Prof. Dr. Luiz Edval, pela amizade e disposição em me ajudar sempre que foi necessário.

Ao Prof. Dr. Robson da EFEI, pela atenção dispensada nos momentos em que precisei.

Às bibliotecárias da EFEI, Cristiane, Jaqueline, Margareth, Monalisa e Shirley, conterrâneas e amigas que não mediram esforços para me ajudar.

À amiga Cristina da PPG, sempre atenta e pronta para dar aquela força.

Ao companheiro Ahmed Esmin, pelas valiosas trocas de idéias.

Ao jovem Ferdinando pelo apoio no desenvolvimento da página da internet.

À Profa. Lucimar Torres pelas correções do texto.

Resumo

Em um Sistema Elétrico de Potência, um ponto de operação qualquer está sujeito a sofrer uma série de possíveis perturbações, danificando equipamentos e causando deficiências na distribuição de energia aos pontos ou regiões do sistema afetados pela perturbação.

Um estudo de contingências consiste numa avaliação da segurança da operação de um determinado estado permanente, dando capacidade ao sistema de suportar contingências não planejadas sem haver perdas parciais ou totais de carga ou transgredir as restrições de operação. Para isso, é então necessária a utilização de uma ferramenta eficaz e rápida.

Dentro deste contexto, este trabalho descreve a transformação de uma aplicação serial de análise de contingência em uma aplicação paralela utilizando computação paralela distribuída com PVM.

PVM (*Parallel Virtual Machine*) é um ambiente de passagem de mensagens muito utilizado pela comunidade científica mundial, que permite a criação de uma máquina paralela virtual, utilizando-se uma rede de computadores.

Foram realizados diversos estudos com a aplicação paralela e a serial. Os resultados obtidos mostram que uma ferramenta paralela de Análise de Contingência é melhor que uma serial quando empregada em sistemas elétricos de maior porte.

Abstract

In an electric power system, any operation point may be disrupted by a series of possible disturbances, damaging equipments and causing deficiencies in the distribution of energy to the points or regions of the system affected by those disturbances.

A study of contingencies consists in an evaluation of the operating safety of a specified permanent state, giving to the system the ability to tolerate unplanned contingencies, without either partial or total charge losses, and without violations to the operating restrictions. To this aim, therefore, it is necessary the use of both an efficient and fast application software.

This work describes the conversion of a serial application software for contingency analysis into a parallel application software, making use of parallel and distributed computation with PVM.

PVM (Parallel Virtual Machine) is a message passing environment highly employed by the world scientific community, which allows the creation of a parallel virtual machine, by means of the use of a computer network.

Several studies with both the serial and parallel application software were performed. The results produced show that a parallel application software for contingency analysis is better than a serial one, whenever it be employed in large electric systems.

Conteúdo

Índice de Tabelas

Índice de Gráficos

Índice de Figuras

Capítulo 1

Introdução

Num sistema elétrico de potência (SEP) é muito importante a confiabilidade e a estabilidade de todo o sistema. Para isso, é então necessário que sejam feitas várias análises que vão desde o projeto até o controle do sistema já em operação.

Estas análises são realizadas no sentido de avaliar e monitorar o desempenho dos sistemas de potência na ocorrência de distúrbios como: variação brusca de carga, perturbações no sistema de transmissão ou até mesmo em condições normais de operação. São exemplos desses estudos: análise do fluxo de potência, estudos de curto-circuito, estudos de sobretensão à freqüência fundamental, estudos de estabilidade eletromecânica, análise de contingências, análise de sensibilidade de tensão, etc.

A utilização de ferramentas confiáveis e rápidas é, então, de extrema necessidade para essas análises, visando respostas corretas e em tempo hábil. Uma ferramenta para a análise de contingência tem alto esforço computacional devido à grande quantidade de informações a serem processadas.

São vários os métodos utilizados para a análise de contingência citados na literatura. Recentemente foi mostrado que a norma do vetor tangente identifica com antecedência a barra crítica, comportamento não observado em nenhum método previamente proposto na literatura.

Com o recente avanço tecnológico nas diversas áreas do conhecimento, buscam-se, cada vez mais, novos métodos, no sentido de reduzir custos e otimizar a obtenção de resultados. Assim, atentou-se para o fato de que a computação, com suas máquinas cada vez mais poderosas e plataformas de alto desempenho, torna cada vez mais viável a substituição, com vantagens, dos experimentos de bancada por simulações em modelos computacionais, sejam de experimentos do próprio homem ou até mesmo de simulações de fenômenos naturais. O primeiro problema com o qual se deparou foi o fato de que tais simulações requeriam uma enorme quantidade de cálculo e elevado número de iterações, o que, mesmo para a tecnologia atual, toma tempo proibitivo caso se queiram resultados rápidos e precisos.

Porém, a limitação física fundamental imposta pela velocidade da luz cria barreiras definindo limites para a velocidade final dos computadores.

A computação paralela, então, surgiu visando oferecer melhor desempenho para aplicações que exigiam muito de uma máquina serial. Através de um grupo de processadores que trabalham em conjunto, problemas que exigem muito processamento são resolvidos com maior facilidade, se comparados a uma máquina serial.

Um dos grandes problemas da computação paralela é o seu custo. Atualmente já existem propostas alternativas para se resolver este problema.

Com o desenvolvimento de processadores mais rápidos, redes de computadores com alto desempenho e grandes discos de armazenamento, pode-se usar, com vantagens, computadores autônomos, porém interligados, para simular uma máquina maciçamente paralela. A isso é dado o nome de sistemas distribuídos.

A computação paralela e sistemas distribuídos surgiram por motivos diferentes: a computação paralela para melhor desempenho de aplicações "pesadas" e os sistemas distribuídos surgiram para compartilhar recursos (como impressoras, discos, etc).

Para utilizar um sistema distribuído para processamento paralelo, foram desenvolvidas ferramentas de software que permitem a utilização do conceito de máquina paralela virtual. Exemplos dessas ferramentas são os ambientes de passagem de mensagens como o PVM (*Parallel Virtual Machine*).

O PVM é uma ferramenta amplamente discutida na literatura e muito utilizada pela comunidade científica mundial. Tem também a grande vantagem de ser um software livre e com código fonte aberto. Por possuir todos esses predicados e ainda uma versão para Windows®, o PVM foi a ferramenta escolhida para nosso trabalho.

Na transformação da ferramenta de análise de contingência serial em paralela utilizamos a versão PVMWIN 3.4. Na ocasião, era a versão mais atual do PVM para Windows®.

Este trabalho está organizado em sete capítulos. No capítulo 2, são discutidos os sistemas distribuídos, descrevendo suas principais características, modelos arquiteturais e os principais componentes.

O capítulo 3 apresenta uma revisão bibliográfica sobre computação paralela, abordando os conceitos, arquiteturas, mecanismos de softwares necessários. São também citadas as principais características, motivos para a utilização da computação paralela distribuída e comentários sobre ambientes de passagem de mensagens.

O capítulo 4 descreve mais detalhadamente o PVM. Parte das características de implementação do PVM também são citadas, além de um programa exemplo.

O capítulo 5 descreve o sistema elétrico de potência (SEP) e seus componentes. É também abordado o fluxo de carga, estado de segurança de um SEP e como funciona a análise de contingência, alvo de melhoramento deste trabalho.

No capítulo 6, é apresentada a implementação da aplicação paralela, análise dos resultados, gráficos e tabelas.

No capítulo 7, estão as devidas conclusões e uma análise final sobre todo o trabalho. Apresentam-se neste capítulo as contribuições, dificuldades e sugestões para trabalhos futuros.

Capítulo 2

Sistemas Distribuídos

Este capítulo tem como objetivo discutir os principais tópicos que envolvem os sistemas distribuídos

2.1 Considerações iniciais

Um Sistema Computacional Distribuído é um conjunto de computadores autônomos interligados por uma rede de comunicação e equipados com um sistema operacional distribuído. O sistema operacional distribuído é o responsável por coordenar as atividades desenvolvidas e compartilhar os recursos do sistema como: hardware, software e dados. Usuários de um sistema distribuído bem projetado devem ter a impressão de que eles estão utilizando um sistema operacional único e integrado, embora esse sistema esteja implementado em vários computadores, em diferentes locais [COU94].

O desenvolvimento dos sistemas operacionais distribuídos foi possível após o aparecimento de tecnologias de redes de computadores de maior desempenho e maior confiabilidade, no início da década de 70. Com o avanço tecnológico das redes de comunicação e com o crescente aumento da potência computacional dos computadores pessoais e das estações de trabalho, vários projetos foram desenvolvidos tornando os sistemas distribuídos mais eficazes e mais difundidos [SAN89a, SAN89b, TAN90, TAN92, MUL93, COU94].

A definição de sistemas distribuídos não é única. Mullender [MUL93] define sistema distribuído como sendo um conjunto de computadores realizando alguma tarefa conjuntamente, destacando três características básicas: múltiplos computadores, conexões dos computadores e o controle de concorrência. Ele enfatiza a potência computacional que os

sistemas distribuídos possuem a mais em relação aos centralizados, citando a tolerância a falhas como aspectos importantes.

Tanenbaum [TAN95,TAN97] descreve sistemas distribuídos como um caso especial de redes de computadores, pois permite ao usuário trabalhar com um conjunto de computadores autônomos, interligados por uma rede de comunicação, de modo transparente, ou seja, para o usuário não é visível a existência de múltiplos recursos. O sistema operacional é o responsável por escolher automaticamente qual é o melhor processador, transferir os arquivos necessários e enviar as respostas para o devido lugar. Segundo essa abordagem, o que caracteriza realmente um sistema distribuído é o software. É o sistema operacional distribuído, o elemento responsável por manter as características necessárias, utilizando como meio de comunicação a rede.

2.2 Caracterização de sistemas distribuídos

Observando-se a história dos sistemas distribuídos, nota-se que há propostas de vários modelos, variando suas características de software e de hardware [TAN95]. Portanto, para a definição dos principais componentes do sistema, como eles se relacionam e quais os seus propósitos, uma descrição arquitetural faz-se necessária, destacando-se como principais aspectos: o tipo de computadores utilizados, sua localização na rede de comunicação e a localização dos componentes de software, geralmente separados e em computadores distintos [COU88]. Para isso dá-se o nome de modelo arquitetural de sistemas distribuídos.

2.2.1 Modelos arquiteturais

Vários modelos arquiteturais foram propostos na década de 80, considerando-se os recursos disponíveis e a interligação apropriada para seus objetivos. Dentre os que se destacam temos: [TAM85, COU88, MUL93, COU94]

- **Estação de trabalho/servidor:** Consiste basicamente em estações de trabalho interligadas através de uma rede de comunicação. As estações apresentam diferentes configurações, possuindo mais ou menos recursos. Estações com uma melhor configuração (maior potência de processamento, mais memória, volume de memória disponível em disco, entre outros) geralmente desempenham a função de servidor. As estações com menos recursos

são utilizadas apenas para executar as aplicações dos usuários. Quando há a necessidade de outros recursos (por exemplo discos), esses são acessados através da rede.

As máquinas que fazem o papel de servidores podem ser dedicadas ou não, desempenhando apenas o trabalho de servidor ou compartilhando a estação com outros servidores e com as aplicações dos usuários. Esse modelo apresenta o inconveniente de oferecer aos usuários uma potência computacional que, ora pode estar ociosa, pois a aplicação pode não necessitar de todos os recursos que a estação oferece, ora pode ser insuficiente para o que se necessita [TAN95, MUL97, COU94].

- **Banco de processadores:** tem como objetivo otimizar a utilização da potência computacional disponível. Com a utilização de terminais, aumenta-se o tráfego na rede, o que é prejudicial para aplicações mais interativas. O modelo banco de processadores necessita de um servidor de processamento que será o responsável pela alocação e liberação de processadores e que geralmente são utilizados como servidores (de arquivos, impressão, etc) devido a sua alta utilização. O banco de processadores é normalmente um conjunto de computadores baratos compostos de placa principal, com processador e memória e de uma interface para a comunicação na rede [TAN95, COU88].

- **Modelo híbrido:** é a união do modelo estação de trabalho/servidor e do modelo banco de processadores, permitindo a alocação dinâmica para tarefas que necessitam de vários computadores concorrentemente sem perder, entretanto, a utilização das estações de trabalho onde são executadas as aplicações que não se adaptam aos terminais [TAN95, COU88].

- **Modelo integrado:** foi desenvolvido para interligar, através de uma rede de comunicação, minicomputadores com estações de trabalho e com terminais. Cada computador possui o software apropriado para desempenhar tanto as tarefas de servidor como também a função de executar as aplicações [COU88].

Analisando a literatura recente [MUL93] e [COU94], observa-se que a classificação proposta na década de 80 tende a não ser mais largamente empregada, onde alguns dos modelos apresentados anteriormente não têm sido nem ao menos citados. Podemos perceber ainda que a organização dos sistemas distribuídos atuais está voltada para o modelo

estação de trabalho/servidor, pelo fato de atender a maioria das necessidades dos usuários e pelos baixos custos que esse modelo apresenta.

2.2.2 Características essenciais

Os sistemas distribuídos devem ter algumas características essenciais ao seu perfeito funcionamento como: compartilhamento de recursos, abertura, concorrência, escalabilidade, tolerância a falhas e transparência [COU94].

- **Compartilhamento de recursos** – é uma necessidade que já vem desde os antigos sistemas centralizados e por isso é fácil compreender os seus benefícios. Como recurso compreendem-se os componentes de software e hardware que estão disponíveis para a utilização do usuário. É uma das características fundamentais de sistemas distribuídos e está diretamente relacionada com o modelo arquitetural do sistema, pois visa uma melhor utilização de seus componentes. A diminuição de custos, o compartilhamento de dados e o trabalho em grupo são alguns dos objetivos dessa característica.

- **Abertura** – é a característica que sintetiza a facilidade com que o sistema pode ser expandido, sempre que necessário, por possuir uma interface muito bem definida e divulgada para que os outros desenvolvedores possam criar produtos (hardware e software) que sejam utilizáveis pelo sistema. Um sistema pode ser aberto quando pode ser expandido no hardware mudando-se memórias, discos, interfaces de comunicação e outros; ou no software, incluindo-se novas características do sistema operacional, protocolos de comunicação e serviços de compartilhamento de recursos; ou fechado, quando o sistema não pode ser expandido como descrito acima. Historicamente, os sistemas sempre foram fechados, sendo que os sistemas baseados em UNIX são alguns dos poucos exemplos de sistemas abertos.

- **Concorrência** – existe quando X processos disputam a utilização de N recursos (sendo $X > N$). Quando $X=N$, os processos podem ser executados simultaneamente, ou seja, eles são executados paralelamente. Em sistemas distribuídos, existem vários computadores com um ou mais processadores, cada um executando o(s) seu(s) processo(s). Por essa razão, a execução paralela e a concorrente ocorrem naturalmente, trazendo o benefício de um desempenho maior e o problema de um gerenciamento mais

complicado, pois os acessos concorrentes e atualização de recursos compartilhados devem ser sincronizados.

- **Escalabilidade** – é a característica que permite a um sistema aumentar a sua capacidade computacional sem afetar a sua utilização (por exemplo, a perda de desempenho com o aumento do número de estações de trabalho). Na década de 80, muitas pesquisas foram desenvolvidas nessa linha, sendo um assunto bastante explorado até o momento. Algumas das técnicas que estão sendo usadas com sucesso para atingir uma maior escalabilidade são a replicação de dados, *cache* e a existência de múltiplos servidores que permitem que tarefas similares sejam feitas em paralelo, distribuindo assim a carga de trabalho entre os servidores.

- **Tolerância a falhas** – é necessária porque algumas vezes computadores falham e, quando isso ocorre, resultados incorretos podem ser apresentados e os dados podem se tornar inconsistentes. O projeto de mecanismos visando a tolerância a falhas do sistema é feito considerando-se dois aspectos fundamentais: a redundância de hardware (com o uso de componentes replicados) e a recuperação de software (com o projeto de programas de recuperação de falhas). Em sistemas distribuídos, a tolerância a falhas é mais facilmente atingida por existirem múltiplos elementos que podem suprir temporariamente a falta de um servidor que não esteja em operação.

- **Transparência** – é a característica que esconde do usuário e de sua aplicação que o sistema é composto por um conjunto separado de elementos, ou seja, o sistema é visto como um todo. A transparência é uma característica de muita importância no projeto de sistemas distribuídos.

2.2.3 Redes de comunicação

Os componentes de um sistema distribuído estão geralmente separados e para que possam compartilhar todos os recursos do sistema, faz-se necessária uma rede de comunicação que permita a sua interconexão. A rede de comunicação pode ser dedicada ou de propósito geral. A rede dedicada é muito empregada em computadores paralelos, para a comunicação entre os processadores, sendo caracterizada principalmente pela sua alta velocidade e confiabilidade. A rede de propósito geral refere-se às redes de computadores,

responsáveis pela interligação de máquinas que visam, principalmente, o compartilhamento de recursos, como discos, impressoras e outros.

Com o crescente aumento da velocidade de transferência das redes e com a constante redução dos custos envolvidos, as redes de computadores tornaram-se o meio de comunicação mais utilizado em sistemas distribuídos [COU94].

A interligação dos componentes de um sistema distribuído é um dos fatores críticos para que tais sistemas apresentem bom desempenho e alta confiabilidade. A interconexão utiliza uma variedade de dispositivos de hardware: meio de transmissão (cabo coaxial, cabo de par trançado, fibra óptica, satélites e outros), interfaces de comunicação, chaveadores de circuitos e componentes de software como os protocolos de comunicação. Alguns autores (por exemplo Tanenbaum [TAN97] e Coulouris [COU94]) rotulam esse conjunto de elementos de comunicação como um subsistema de comunicação.

2.2.3.1 Tecnologias de redes

As redes de longas distâncias (WANs – *Wide Area Networks*) foram as primeiras redes desenvolvidas e tinham como objetivos interligar computadores separados por longas distâncias.

As WANs são caracterizadas por baixas taxas de transferência (de 20 a 500 Kilobits/segundo) e baixa confiabilidade, exigindo protocolos de comunicação mais rigorosos no controle de qualidade das informações transmitidas. Uma WAN é composta por um conjunto de computadores conhecidos como PSEs (*Packet-switching Exchange*), cuja tarefa é receber pacotes de dados e enviar a outro PSE, estabelecendo uma rota entre a origem e o destino da informação. Os *host*s são computadores que recebem e enviam informações na rede, através dos PSEs [TAN97].

As operações de roteamento fazem com que as WANs apresentem alto tempo de latência (entre 0,1 e 0,5 segundos) que, aliado às baixas taxas de transmissão, não as faz recomendáveis para sistemas distribuídos [COU94].

O desenvolvimento da ISDN (*Integrated Services Digital Network*) e da B-ISDN (Broadband ISDN), redes digitais para a transferência de voz, dados e de imagens (no caso da B-ISDN), está mudando o padrão de WANs, apresentando taxas de transmissão extremamente elevadas. A adoção de técnicas de chaveamento ATM (*Asynchronous Transfer Mode*), em substituição aos PSEs, na rede B-ISDN, reduzirá o tempo de latência a até 1 milisegundo [TAN97, MUL93,COU94].

As redes locais (LANs – *Local Area Networks*) foram desenvolvidas para interligar computadores separados por pequenas distâncias (escritórios, indústrias, laboratórios, etc). As LANs são apropriadas para sistemas distribuídos por suas altas taxas de transferência (0.2 a 100 Megabits/seg) e por sua alta confiabilidade (com taxas de erro geralmente 1000 vezes menores que as WANs). As LANs normalmente são estruturas em barramento ou em anel e não utilizam-se de PSEs, pois não necessitam de roteamento. Os *hosts* são conectados na rede diretamente no canal de comunicação através de uma interface extremamente simples [COU94].

O software de comunicação é mais simples do que o software das WANs, permitindo protocolos menos complexos e consequentemente mais rápidos. O meio de transmissão geralmente varia entre o cabo de par trançado, o cabo coaxial e a fibra óptica, sendo que a fibra óptica tem a melhor velocidade de transmissão e menor taxa de erros.

As MANs (*Metropolitan Area Networks*) têm sido apontadas como uma rede de tamanho intermediário entre as LANs e as WANs. Uma MAN é baseada principalmente em fibras óticas para transmissão de vídeo, voz e outros dados sobre distâncias acima de 50 Km (normalmente dentro de uma cidade). Suas taxas de transferência são similares a ISDN, podendo usar também as técnicas de chaveamento ATM, além de possuir tempo de latência baixo (menor que 1 milisegundo). Suas características atendem às necessidades dos sistemas distribuídos modernos pois são rápidas e mais confiáveis que as WANs.

2.3 Comunicação entre os processos

Mecanismos para a comunicação entre processos (IPC – *Interprocess Communication*) foram introduzidos em alguns sistemas operacionais (por exemplo o BSD4.x, versão do UNIX desenvolvido pela University of California at Berkeley) para atender às necessidades de comunicação e sincronização entre os processos concorrentes. Um mecanismo IPC consiste, portanto, de um conjunto de regras (ou protocolos) de comunicação que permitem que um processo envie uma mensagem para um outro processo (ou para um grupo de processos), trocando informações e sincronizando as atividades que estão sendo executadas separadamente.

A comunicação entre processos envolve características fundamentais para algumas necessidades dos usuários de sistemas distribuídos. Dentre elas destacam-se: representação de dados externa, operações *send* e *receive*, comunicações síncronas e assíncronas.

A representação de dados externa ou XDR (*External Data Representation*) é um padrão de representação de dados desenvolvido pela SUN Microsystems, Inc., para a troca de mensagens entre clientes e servidores quando estes não são do mesmo tipo [STE90] [COU94]. Quando uma mensagem é enviada, ela é convertida em uma forma externa ao sistema de origem (padrão XDR por exemplo), transmitida pela rede de comunicação e, quando recebida, é transformada no formato reconhecido pela máquina destino.

Send e *receive* são duas operações básicas para a troca de mensagens. O *send* é usado quando um processo envia uma mensagem (solicitando um determinado serviço, por exemplo) a um outro processo, o qual está esperando mensagens através da operação *receive*.

Operações Síncronas e Assíncronas definem se os processos que estão envolvidos na comunicação estão sincronizados em todas as mensagens enviadas. Por exemplo, para que uma comunicação seja síncrona, o processo que envia a mensagem deve ser bloqueado até que o correspondente *receive* esteja apto a receber a mensagem. Em uma comunicação assíncrona, o(s) processo(s) não é (são) bloqueado(s) (o processo emissor, receptor ou ambos), sendo que a mensagem pode ser inserida em um *buffer* para seu armazenamento temporário.

Dois padrões de comunicação são geralmente usados em sistemas distribuídos: Comunicação cliente/servidor e comunicação de grupo.

2.3.1 Comunicação cliente/servidor

Esse padrão de comunicação é um dos mais adotados por apresentar uma estrutura simples, de bom desempenho e confiável. Um servidor é todo processo que oferece serviços para outros usuários do sistema (os clientes). Um cliente é todo processo que necessita de serviços disponíveis nos servidores. A comunicação cliente/servidor consiste, portanto, em uma mensagem enviada pelo cliente, solicitando um serviço do servidor que , por sua vez, executa o pedido e envia uma nova mensagem para o cliente com a resposta.

2.3.2 Comunicação de grupo

Existem situações onde a comunicação deve ser feita entre processo e um grupo de processos, ou seja, de 1 para N. Nesses casos, o modelo cliente/servidor não é a melhor opção. A comunicação de grupo permite que uma mensagem seja enviada para membros de um

grupo de processos, geralmente em diferentes computadores, com o objetivo de fornecer maior tolerância a falhas e maior disponibilidade dos recursos [COU94].

A comunicação de grupo é muito empregada nos sistemas distribuídos que possuem os seguintes objetivos:

- **A tolerância a falhas baseadas em serviços replicados:** Onde as requisições dos clientes são enviadas para vários servidores replicados que irão fazer exatamente a mesma operação. Caso um dos servidores interrompa o seu funcionamento, outro poderá atender os clientes.

- **Localização de objetos em serviços distribuídos:** A comunicação de grupo pode ser usada para localizar objetos, como arquivos, em um serviço distribuído de arquivos.

- **Melhor desempenho através de dados replicados:** A replicação é muito utilizada para permitir que os recursos/dados do sistema como discos, arquivos, impressoras, fiquem próximos dos usuários e com isso sejam obtidos com um menor tempo de espera.

- **Atualização múltipla:** Na comunicação de grupo, pode-se avisar a um grupo de usuários interessados sobre um determinado acontecimento, como por exemplo, a ocorrência de uma nova mensagem do departamento pessoal.

2.4 Chamada de procedimento remoto (RPC)

Utilizando o bem difundido conceito das chamadas de procedimento ou LPC (Local Procedure Call), a chamada de procedimento remoto ou RPC (*Remote Procedure Call*) é um mecanismo de comunicação de mais alto nível, onde o objetivo é esconder do usuário as técnicas utilizadas para realizar as trocas de mensagens entre os processos que estão espalhados pela rede [STE90].

RPC é muito utilizada para a comunicação cliente/servidor, onde clientes chamam procedimentos localizados nos servidores (responsáveis pela execução do procedimento).

Todos os processos de comunicação, como a localização do processo remoto, o empacotamento dos dados para a transmissão e a transmissão em si, são transparentes ao usuário final, graças aos procedimentos *stubs*, responsáveis por receber a chamada remota

feita pelo cliente, realizar a comunicação (verificando todos os detalhes necessários), esperar a reposta do servidor e devolver ao cliente os resultados obtidos [COU94].

2.5 Serviços disponíveis em um sistema distribuído

Os sistemas distribuídos são caracterizados pela existência de múltiplos serviços separados em computadores diferentes e implementados fora do núcleo (*Kernel*) do sistema operacional. Isso possibilita o projeto de sistemas operacionais distribuídos compactos, responsáveis apenas pelas atividades essenciais de todo sistema computacional e pela comunicação entre os processos [TAN95, MUL93, COU94].

Esta seção discute alguns dos principais serviços disponíveis nos sistemas distribuídos. Esses serviços são implementados através de servidores (processos que prestam serviços), podendo ser implementados em máquinas dedicadas ou compartilhando uma máquina com outros servidores.

Serviços de arquivos: É um componente essencial dos sistemas distribuídos, sendo responsável pelo armazenamento não volátil de informações. Outra finalidade é permitir o compartilhamento de arquivos, possibilitando o acesso remoto a partir de outras estações de trabalho, o que incrementa a mobilidade dos usuários. O serviço de arquivos é um dos mais requisitados, servindo também como instrumento de armazenamento para outros serviços como o de nomes, o de impressão e outros.

Serviço de nomes: É um dos componentes de um sistema distribuído responsável pela relação entre nomes textuais de recursos e seus respectivos endereços. O serviço de nomes consiste basicamente de um banco de dados onde um servidor (o de nomes) interpreta um nome, ou seja, traduz um nome para um identificador reconhecido e localizado pelos demais componentes do sistema. Outra finalidade do serviço de nomes é verificar os privilégios do usuário que está solicitando acesso a um determinado recurso.

Serviço de impressão: Oferece aos usuários as facilidades de impressão onde arquivos podem ser impressos em máquinas compartilhadas por vários usuários. Para a implementação do serviço de impressão, normalmente utiliza-se o serviço de arquivos.

Serviço de tempo: Esse serviço é responsável por manter todas as máquinas do sistema sincronizadas. Segundo Tanenbaum [TAN95], é impossível ter todos os processos exatamente sincronizados, devido principalmente à demora variável de propagação da rede, de acordo com a sua utilização. Algoritmos que diminuam esse problema tornando os relógios dos computadores quase sincronizados são apresentados em Coulouris [COU94].

Serviço de *boot*: Esse serviço é utilizado quando as máquinas são ligadas, fornecendo-lhes o sistema operacional necessário para seu funcionamento. Tanenbaum [TAN95] descreve ainda que o serviço de *boot* pode verificar todas as máquinas do sistema, para que seja efetuada a recuperação de falhas em uma eventual paralisação.

Os serviços também são comumente encontrados nas implementações de sistemas distribuídos, usados para fornecer um melhor ambiente de trabalho para seus usuários. A literatura destaca os seguintes serviços complementares: correio eletrônico, serviço de replicação, serviço de comunicação, etc.

2.6 Considerações finais

Nos últimos anos, os sistemas distribuídos tornaram-se muito populares e muitas são as linhas de pesquisa que têm sido desenvolvidas no sentido de melhorar cada vez mais o compartilhamento de recursos com transparência, desempenho e confiabilidade.

Os principais tópicos e características envolvidos com os sistemas computacionais distribuídos foram apresentados neste capítulo. A transparência é o requisito mais importante e o mais discutido na literatura.

Desempenho e confiabilidade são dependentes diretamente da tecnologia adotada para a rede de comunicação de dados. Com o desenvolvimento de novas tecnologias (como ATM e B-ISDN), a utilização de sistemas distribuídos crescerá significativamente.

Com o avanço tecnológico dos processadores e das redes de comunicação, os sistemas distribuídos começaram a ser utilizados por várias outras áreas da ciência da computação, as quais necessitam de uma maior potência computacional. A computação paralela é um exemplo disso, onde a utilização dos sistemas distribuídos surgiu como alternativa para melhorar a relação custo/benefício dos sistemas paralelos.

Capítulo 3

Computação Paralela

Este capítulo descreve algumas características fundamentais sobre computação paralela, abordando os seus tópicos mais relevantes. São apresentados conceitos básicos, classificações de arquiteturas paralelas, técnicas utilizadas em programação concorrente, suporte à obtenção de algoritmos paralelos e, finalizando o capítulo, são discutidas as principais características da computação paralela utilizando sistemas distribuídos.

3.1 Considerações iniciais

Computação paralela (ou processamento paralelo) consiste basicamente em um conjunto de elementos de processamento, que cooperam e comunicam-se entre si para solucionarem grandes problemas, de maneira mais rápida e eficiente do que se estivessem sendo processados serialmente [ALM94].

Por mais que arquiteturas seqüenciais de Von Neumann (tradicionalmente aceitas) tenham conseguido nos últimos anos um grande avanço tecnológico, permitindo maiores desempenhos, elas se mostram ineficientes em aplicações que necessitam de uma grande quantidade de cálculos e elevado número de iterações o que, mesmo para as arquiteturas seqüenciais atuais, toma tempo proibitivo caso se queiram resultados rápidos e precisos. A limitação física fundamental imposta pela velocidade da luz cria barreiras definindo limites para a velocidade final dos computadores.

Assim, surgiu, inicialmente na forma de um computador capaz de resolver equações diferenciais em paralelo, projetado por Vanevaur Bush em 1920, a idéia do paralelismo. No entanto, o marco fundamental foi o surgimento de máquinas para processamento paralelo como o ILLIAV IV, construído nos fins de 60 na Universidade de Illinois, composto por 64 processadores. Atualmente, pode-se contar com máquinas muito poderosas compostas de, até mesmo, milhares de processadores em um único computador.

Como exemplos de aplicações de processamento paralelo podemos citar :

- Previsão do tempo – Para prevermos o tempo é necessária a simulação de um futuro atmosférico, evoluindo do estado inicial até o momento em que se quer prever;
- Engenharia – Alta velocidade computacional é essencial à pesquisa da engenharia moderna (como exemplo podemos citar a Análise de contingências);
- Economia – Usado para previsão, baseando-se em teorias;
- Inteligência Artificial;
- Bases de dados extremamente grandes;
- Circulação de correntes marinhas;
- Aerodinâmica;
- Exploração sismológica;
- Engenharia Genética.

A razão principal para o surgimento da computação paralela foi, portanto, a capacidade de aumentar o processamento em uma única máquina. Entre as principais vantagens destacam-se:

- Alto desempenho para programas lentos e ou de alto esforço computacional;
- Solução mais natural para problemas intrinsecamente paralelos;
- Maior tolerância a falhas.

3.2 Conceitos básicos

A computação paralela apresenta inúmeras características não encontradas na computação seqüencial pois, além das inúmeras maneiras de se organizar os processadores, há também a necessidade de um gerenciamento complexo dos elementos de processamento, bem como a manutenção da coerência da informação que trafega pela máquina [NAV89].

3.2.1 Paralelismo e concorrência

A concorrência existe quando, em um determinado instante, dois ou mais processos começaram a sua execução, mas não terminaram [ALM94]. Por essa definição, concorrência pode ocorrer tanto em sistemas com um único processador, quanto em sistemas com múltiplos processadores.

Afirmar que processos estão sendo executados em paralelo implica na existência de mais de um processador, ou seja, paralelismo (ou paralelismo físico) ocorre quando há mais

de um processo sendo executado no mesmo intervalo de tempo e em sistemas com multiprocessadores.

Quando vários processos são executados em um único processador, sendo que somente um deles é executado a cada vez, tem-se um pseudo-paralelismo. O usuário tem a falsa impressão de que suas tarefas (*tasks*) são executadas em paralelo, mas, na realidade o processador é compartilhado entre os processos. Isto significa que, em um determinado instante, somente um processo é executado, enquanto que os outros que já foram iniciados antes, aguardam a liberação do processador para continuarem.

Entende-se por processo, um programa em execução. Um processo consiste em um programa executável, seus dados, a pilha e o ponteiro para a pilha, o contador de instruções, os registradores e todas as informações de que necessita para ser executado [TAN92].

3.2.2 Níveis de paralelismo

O nível do paralelismo ou granularidade indica o tamanho da tarefa executada pelo processador. Várias granularidades são sugeridas na literatura [ALM94, HWA84, KIR91, NAV89]. Resumidamente, podem ser classificadas em fina, média e grossa.

Uma granularidade fina indica que o paralelismo é feito ao nível de operações e geralmente implementada no hardware. Na granularidade média o paralelismo é atingido entre os blocos ou subrotinas do programa. Em granularidade grossa são paralelizados os processos.

A granularidade está diretamente relacionada ao hardware. A granularidade fina requer um maior número de processadores e também mais específicos. Na granularidade grossa tem-se um menor número de processadores, podendo ser mais genéricos.

3.2.3 *Speedup* e eficiência

Um dos fatores mais importantes da computação paralela é o aumento de velocidade de processamento obtida com o paralelismo. Para se calcular o aumento obtido, dois parâmetros são abordados na literatura: *speedup* e eficiência [ALM94, KIR91].

***Speedup*(Sp)** é utilizado para determinar o aumento de velocidade obtido durante a execução de um programa em um computador paralelo, em relação ao número de elementos de processamento.

$$Sp = t1 / tp$$

Onde **t1** é o tempo que o programa (utilizando o algoritmo paralelo) gastou ao ser executado em um processador e **tp** é o tempo com p processadores.

O caso ótimo é obtido quando Sp=p, ou seja, na medida em que se aumenta o número de processadores, aumenta-se diretamente a velocidade de processamento. Sistemas onde Sp=p são conhecidos como Sistemas Escalares [MCB94]. Fatores que influenciam a obtenção do caso ótimo estão relacionados principalmente com a comunicação entre os processos paralelos e a parte seqüencial (que não pode ser paralelizada) do algoritmo.

A **eficiência(Ep)** determina o quanto estão sendo utilizados os processadores, ou seja, quando Sp não é diretamente proporcional a p, há uma perda de desempenho.

$$Ep = Sp / p$$

Onde **p** é o número de processadores.

A variação de Ep é entre 0(zero) e 1(um), sendo que o valor 1 indica uma eficiência de 100 %.

3.2.4 *Pipeline*

O *pipeline* é obtido dividindo-se o processo em uma seqüência de sub-processos onde cada um será executado por um estágio de hardware específico, que trabalhará concorrentemente com os outros estágios do *pipeline* [NAV89].

Os microprocessadores atuais estão empregando as técnicas de *pipeline* para resolver os problemas de desempenho dos computadores seqüenciais. Esse nível de paralelismo com granularidade fina não implica que esses computadores sejam paralelos, visto que a máquina continua tendo apenas um elemento processador.

3.3 Arquiteturas paralelas

Por arquitetura paralela, entende-se a máquina capaz de executar mais de uma tarefa ao mesmo tempo, excluindo-se o paralelismo de baixo nível. Com o avanço do processamento

paralelo, observa-se que foram propostas inúmeras arquiteturas, cada uma apresentando características diferentes.

Para acompanhar o desenvolvimento das arquiteturas e para agrupar equipamentos com características comuns, foram propostas algumas classificações. A classificação de Flynn [FLY72], embora muito antiga, é amplamente adotada e baseia-se no fluxo de instruções e no fluxo de dados. Duncan [DUN90] apresenta uma classificação com o objetivo de acabar com a dificuldade de acomodar as novas arquiteturas dentro da classificação de Flynn.

3.3.1 Classificação de Flynn

A classificação de Flynn divide as arquiteturas em quatro categorias de máquinas. Cada arquitetura depende da quantidade de fluxo das instruções e dos dados. As categorias são : [FLY72, HWA84, NAV89, ALM94, DUN90, KIR91]

- **SISD** (*Single Instruction Single Data Stream*)
- **SIMD** (*Single Instruction Multiple Data Stream*)
- **MISD** (*Multiple Instruction Single Data Stream*)
- **MIMD** (*Multiple Instruction Multiple Data Stream*)

A categoria **SISD** possui apenas um fluxo de instruções e um fluxo de dados. Compreendem as máquinas de Von Neumann, largamente utilizadas. A execução é seqüencial embora uma possível execução *pipeline* possa existir dentro do processador. Pode haver mais de uma unidade funcional (como coprocessadores), mas mesmo assim todos estão subordinados a uma única unidade de controle.

A categoria **SIMD** apresenta um único fluxo de instruções atuando sobre vários fluxos de dados. Esta arquitetura possui várias unidades de processamento supervisionadas por uma única unidade de controle. Isso faz com que todos os elementos de processamento executem as mesmas instruções sobre dados diferentes. Processos matriciais são exemplos dessa categoria. A memória utilizada pode ser compartilhada, ou seja, comum a todas as unidades de processamento, ou distribuída, onde cada unidade de processamento possui o seu próprio espaço para endereçamento da memória.

A categoria **MISD** é caracterizada por apresentar múltiplos fluxos de instruções para um único fluxo de dados. O fluxo de dados passaria por todas as unidades de processamento, sendo que o resultado de uma seria a entrada para a próxima unidade. Não há, na literatura

disponível, exemplos de máquinas MISD. Contudo alguns autores [ALM94, NAV89] citam a possibilidade de encaixar o *pipeline* como representante desta categoria.

A categoria **MIMD** é composta por múltiplos fluxos de instruções e múltiplos fluxos de dados. Cada unidade de processamento atua sobre um conjunto de dados diferente e possui uma unidade de controle que a supervisiona. Podem ser fracamente ou fortemente acoplados, dependendo do grau de interação existente entre os processadores.

3.3.2 Classificação de Duncan

Duncan [DUN90] propõe uma classificação com o objetivo de encaixar as inovações arquiteturais dos últimos anos em uma taxonomia mais coerente. Para o desenvolvimento dessa classificação três aspectos básicos são considerados:

- Manter os elementos da classificação de Flynn, devido à ampla utilização;
- Incluir as novas arquiteturas criadas depois da classificação de Flynn (como as sistólicas e hipercúbicas) e que são difíceis de se enquadrar;
- Excluir arquiteturas que apresentam apenas paralelismo de baixo nível, como por exemplo as várias unidades funcionais e a execução decomposta do processador em estágios autônomos (*pipeline*).

A classificação é composta de dois grandes grupos:

1. **Arquiteturas síncronas**
   - 1.1. Processadores Vetoriais
   - 1.2. Processadores Matriciais (SIMD)
   - 1.3. Arquiteturas Sistólicas
2. **Arquiteturas assíncronas**
   - 2.1. Memória Compartilhada
   - 2.2. Memória Distribuída
   - 2.3. Arquiteturas híbridas

   } Arquiteturas MIMD

   - 2.4. Arquiteturas a fluxo de dados
   - 2.5. Arquiteturas de Redução
   - 2.6. Arquiteturas de frente de onda

   } Arquiteturas não convencionais

Arquiteturas síncronas: As operações concorrentes são coordenadas por uma unidade de controle central e por um relógio comum ao sistema. Pertencem a esta categoria:

- **Processadores vetoriais:** Possuem um hardware específico, o qual permite que sejam feitas seqüências de instruções idênticas sobre vetores de forma mais rápida do que uma seqüência de operações escalares [ALM94]. Os processadores vetoriais utilizam *pipeline* (paralelismo temporal) para agilizar a execução das operações.

- **Arquiteturas SIMD:** Como já descrito anteriormente, as máquinas SIMD são compostas por um único fluxo de instruções atuando sobre diferentes dados. Os processadores matriciais são exemplos dessa categoria síncrona, onde a unidade de controle supervisiona e sincroniza as unidades de processamento.

- **Arquiteturas sistólicas:** Essas arquiteturas possuem vários processadores enfileirados (*pipeline*), sendo que a informação trafega por vários processadores antes de retornar à memória. Segundo Almasi [ALM94], o nome sistólicas vem do fato das informações pulsarem sincronamente entre os processadores como o sangue pulsando no coração. Apresentando alto grau de paralelismo (com granularidade baixa), arquiteturas sistólicas apresentam alto desempenho e são empregadas em problemas específicos (como processamento de sinais) [DUN90].

Arquiteturas assíncronas: São caracterizadas principalmente por um controle de hardware descentralizado, onde cada unidade de processamento executa diferentes instruções sobre diferentes dados. Essa é a definição da categoria MIMD, na classificação de Flynn, ou seja, as arquiteturas assíncronas são compostas basicamente de máquinas MIMD, convencionais ou não.

- **Arquiteturas MIMD convencionais:** nessas arquiteturas, a comunicação e o sincronismo necessários são feitos de acordo com a organização da memória, que podem ser compartilhada (comum a todos os processadores) ou distribuída (cada processador possui sua própria memória, não a dividindo com outro processador).

As arquiteturas MIMD com memória compartilhada possuem uma única memória, à qual todos os processadores têm acesso. A memória não necessita estar agrupada fisicamente, podendo ser constituída por várias unidades independentes, mas agrupadas logicamente

(endereçamento de memória comum a todos os processadores). O simples acesso aos dados permite a comunicação e o sincronismo, embora sejam necessários critérios adicionais para que acessos concorrentes não tornem as informações inconsistentes.

Essas máquinas também são chamadas de multiprocessadores e são fortemente acopladas devido à grande troca de informação existente entre os elementos de processamento. A rede de conexão deve apresentar bom desempenho e alta confiabilidade, o que torna o seu custo mais elevado. Essa arquitetura deve ser empregada em situações onde o nível de paralelismo é mais fino necessitando, assim, maior troca de informações.

Nas arquiteturas MIMD com memória distribuída, cada processador possui a sua própria memória, não tendo acesso à memória dos outros processadores. A comunicação e o sincronismo ocorrem por troca de mensagens através da rede de conexão. Problemas de inconsistência devidos ao acesso concorrente das informações, desaparecem.

- **Arquiteturas MIMD não convencionais:** As arquiteturas não convencionais agrupam todos os sistemas que não se encaixam em nenhuma das categorias anteriores. Máquinas híbridas (MIMD/SIMD), a fluxo de dados, de redução e de frente de onda são exemplos dessas arquiteturas. Elas apresentam aspectos MIMD, pois são assíncronas e com múltiplos fluxos de instruções e de dados, mas possuem características próprias, impedindo que sejam classificadas apenas como MIMD [DUN90].

As arquiteturas híbridas são compostas basicamente por uma arquitetura MIMD que possui algumas partes controladas por arquiteturas SIMD.

As arquiteturas a fluxo de dados (*dataflow*) possuem como característica fundamental o modelo de execução, no qual as instruções tornam-se habilitadas para a execução somente quando seus operandos estiverem disponíveis. Com isso a seqüência de instruções é baseada na dependência dos dados, permitindo a essas arquiteturas explorarem a concorrência ao nível de tarefas, rotinas e instruções [DUN90].

As arquiteturas de redução são também conhecidas como arquiteturas dirigidas por demanda, visto que os processadores somente executam os comandos quando as expressões necessitam do seu cálculo. O conceito de redução está na substituição de porções do código fonte original de um programa pelo seu significado [KIR91]. Estruturas em árvore podem ser utilizadas em arquiteturas de redução, onde as folhas possuem os itens do programa mapeado e os outros nós realizam os procedimentos necessários.

As arquiteturas de frente de onda combinam as estruturas *pipeline* sistólicas com a execução assíncrona a fluxo de dados. O objetivo é conservar as vantagens das arquiteturas

sistólicas e eliminar algumas das suas desvantagens. Ambas as arquiteturas, sistólicas e de frente de onda, possuem uma rede de conexão regular, interligando os processadores. Porém, as arquiteturas de frente de onda não possuem um relógio global com atrasos explícitos, usados para manter a sincronização das estruturas *pipeline*. Protocolos assíncronos são usados para coordenar os movimentos dos dados entre os processadores, fazendo com que as frentes de ondas computacionais passem pela matriz de processadores sem causar interferência [KIR91, DUN90].

3.4 Programação concorrente

A programação seqüencial utilizada nas arquiteturas de Von Neumann possui basicamente: instruções executadas seqüencialmente, desvios condicionais e incondicionais, estruturas de repetição (laços) e subrotinas. Para determinadas aplicações esses recursos implicam em processamento lento e em sub-utilização dos recursos de hardware.

A programação concorrente tem por objetivo otimizar o desempenho dessas aplicações, explorando a concorrência em arquiteturas monoprocessadoras ou multiprocessadoras, permitindo assim uma melhor utilização dos recursos de hardware disponíveis.

A idéia básica da programação concorrente, portanto, é que determinadas aplicações sejam divididas em partes menores e que cada parcela resolva uma porção do problema. Para essa divisão, há a necessidade de certos recursos adicionais que são responsáveis, por exemplo, pela ativação e finalização dos processos concorrentes, e que não estão disponíveis na programação seqüencial.

Segundo Almasi [ALM94], três fatores são essenciais para a execução paralela:

- Definir um conjunto de subtarefas para serem executadas em paralelo;
- Capacidade de iniciar e finalizar a execução das subtarefas;
- Capacidade de coordenar e especificar a interação entre as subtarefas enquanto estiverem executando.

Para que os processos concorrentes sejam ativados, faz-se necessário um conjunto de ferramentas que determine exatamente que porção do código será paralela e qual será seqüencial.

A coordenação e especificação da interação entre os processos concorrentes são feitas pela comunicação e pela sincronização entre eles. A comunicação permite que a execução de um processo interfira na execução de outro e pode ser feita ou pelo uso de variáveis

compartilhadas (memória compartilhada) ou pelo uso de passagem de mensagens (memória distribuída).

A sincronização se faz necessária para que o acesso simultâneo não torne os dados que são compartilhados inconsistentes e para que seja possível o controle na seqüência da execução paralela.

3.4.1 Comunicação e sincronização

Observa-se da literatura disponível [HWA84, NAV89, KIR91, ALM94], que os mecanismos de coordenação (comunicação e sincronização) foram desenvolvidos para atuar em um dos dois grandes grupos: arquiteturas com memória compartilhada ou nas arquiteturas com memória distribuída.

Para coordenação de processos utilizando memória compartilhada, foram criados alguns mecanismos como *busy-waiting*, semáforos e monitores, os quais utilizam variáveis de memória para o controle necessário.

Para a coordenação de processos em arquiteturas com memória distribuída, são utilizados mecanismos como a comunicação ponto-a-ponto, *rendezvous* e chamada de procedimento remoto (RPC).

Comunicação e sincronização com memória compartilhada. A sincronização de processos concorrentes, utilizando memória compartilhada, tem dois objetivos:

1. Controlar a seqüência onde, para determinados trechos do programa, é estabelecida a ordem de execução dos processos.
2. Exclusão mútua, ou seja, estabelecer um controle de acesso a determinadas áreas do programa (chamadas regiões críticas), para que dois ou mais processos não tenham acesso simultaneamente, por exemplo, às variáveis de memória, tornando-as potencialmente inconsistentes.

- **Busy-waiting** é uma das possíveis formas de sincronização na qual, para o processo ter acesso à região crítica, deve primeiro passar por um protocolo de entrada. O protocolo assegura que somente um processo utilize a região crítica de cada vez. Caso o processo não tenha acesso, ele permanece em uma estrutura de repetição tentando obter permissão para prosseguir. Várias abordagens foram desenvolvidas (como por exemplo instruções *test and*

set, soluções de Peterson, entre outras) e são bastante discutidas na literatura [TAN92, KIR91, ALM94, HWA84].

Busy-waiting apresenta as desvantagens de manter o processo ocupado enquanto espera, utilizando para isso tempo de processamento, além de possibilitar a entrada de mais de um processo ao mesmo tempo na região crítica, caso não se utilize protocolos mais complexos que garantam a exclusão mútua.

- **Semáforo** é um mecanismo de sincronização genérico composto por duas operações atômicas (indivisíveis); *Down* (ou p) e *Up* (ou v). Essas operações atuam sobre uma variável compartilhada S inteira e não negativa, tanto para permitir a exclusão mútua (quando S é iniciado com o valor 1), quanto permitir para o controle o acesso (quando S é iniciado com um valor maior que 1). *Down* é o protocolo de entrada na região crítica, verificando o valor de S e bloqueando, quando S for 0(zero) e decrementando S de uma unidade quando S for maior que 0(zero). *Up* é o responsável por liberar a região crítica quando o processo a estiver deixando.

- **Monitor** é uma estrutura de dados compartilhada e um conjunto de funções que tem acesso a essa estrutura. Seu objetivo é controlar o acesso a essa estrutura pelos processos concorrentes. Basicamente, um monitor consiste de algumas variáveis permanentes que armazenam o estado do recurso compartilhado e de alguns procedimentos, responsáveis pelas operações no recurso.

Para se ter acesso ao monitor, deve ser especificado o seu identificador (seu nome), seguido do nome do procedimento desejado. Ao chamar o monitor, o processo poderá passar por uma fila de entrada (caso o monitor esteja ocupado) e quando estiver executando a operação, poderá permanecer algum tempo nas filas de condição, caso precise de alguma condição para prosseguir.

Monitor é uma estrutura utilizada para sincronização, bem elaborada, de alto nível e que protege e disciplina o uso de recursos compartilhados, os quais são executados em exclusão mútua.

Comunicação e sincronização com memória distribuída. Quando a passagem de mensagens é usada para a comunicação e sincronização, os processos enviam e recebem informações ao invés de compartilhar variáveis. A troca de mensagens é feita através das

primitivas *send/receive*, ou através de chamadas de procedimentos remotos (RPC) [HWA84, KIR91, ALM94].

A comunicação pode ser síncrona ou assíncrona. Na comunicação síncrona o comando *send* aguarda que o comando *receive* seja executado, permitindo assim que a mensagem seja enviada. Na comunicação assíncrona, o comando *send* não necessita esperar o comando *receive*, utilizando um *buffer* onde deposita temporariamente o conteúdo da mensagem para que o *receive*, quando pronto, a receba.

No mecanismo de comunicação ponto-a-ponto, dois processos concorrentes executam as primitivas *send/receive*. Esse mecanismo é caracterizado por apresentar comunicação síncrona e unidirecional. No mecanismo *rendezvous*, dois processos concorrentes executam as primitivas *send/receive* duas vezes, apresentando comunicação síncrona e bidirecional.

Na chamada de procedimento remoto(RPC), um processo pode solicitar a execução de um outro processo remotamente, usando a mesma sintaxe das chamadas de procedimentos locais. O processo que chamou fica bloqueado até que o procedimento remoto seja executado fazendo, portanto, uma comunicação síncrona. Normalmente, o procedimento remoto retorna valores, tornando a comunicação bidirecional.

3.5 Suporte para obtenção de algoritmos paralelos

O desenvolvimento de algoritmos paralelos requer ferramentas eficazes para ativar os mecanismos de iniciam/finalizam e coordenam processos concorrentes, descritos na seção anterior. Essas ferramentas podem ser implementadas de duas maneiras distintas: ou como linguagens de programação concorrente ou como ambientes de passagem de mensagens.

As linguagens de programação concorrente contêm os comandos específicos que os algoritmos paralelos necessitam. Os ambientes de passagem de mensagens são compostos por bibliotecas de comunicação que implementam os mecanismos necessários à execução paralela, agindo como extensões das linguagens seqüenciais existentes [ALM94].

3.5.1 Linguagens de programação concorrente

Observa-se na literatura disponível que as linguagens de programação podem ser classificadas de diversas formas. Andrews [AND83] classifica as linguagens em três grupos:

as orientadas a procedimentos (baseadas em variáveis compartilhadas), as orientadas a mensagens e operações (ambas baseadas em passagem de mensagens).

Almasi [ALM94] agrupa as linguagens em imperativas e declarativas. As imperativas são a grande maioria das linguagens disponíveis (como Fortran, C, Pascal, Visual Basic, Delphi, etc) caracterizadas principalmente pela seqüência de comandos, os quais são responsáveis pela manutenção dos dados manipulados pelo programa. As linguagens imperativas analisam "como" um determinado problema pode ser resolvido.

As linguagens declarativas são projetadas para enfatizar "o que" o problema quer resolver, tentando com isso deixar a aplicação menos voltada para o equipamento que está sendo utilizado e mais para o problema em si. Linguagens para programação funcional (como a fluxo de dados) e para lógica (como Lisp, Prolog e Prolog concorrente) são exemplos dessa categoria [ALM94].

Como a maioria das linguagens utilizadas são imperativas, o maior número de versões paralelas também são dessa categoria. O Fortran e a linguagem C possuem várias versões e/ou extensões paralelas (como Fortran90, PVM para C e Fortran, entre outras) visto que são utilizados principalmente no meio científico, onde são encontradas várias aplicações paralelizáveis. Essas versões paralelas têm, portanto, o objetivo de preservar o investimento feito no desenvolvimento das aplicações, proporcionando uma maneira mais rápida e mais barata de aumentar o desempenho dos algoritmos.

3.5.2 Ambientes de passagem de mensagem

Um ambiente de passagem de mensagem consiste basicamente de uma biblioteca de comunicação que, atuando como uma extensão das linguagens seqüenciais (como C e Fortran), permite a elaboração de aplicações paralelas.

Os ambientes de passagem de mensagens foram desenvolvidos inicialmente para máquinas com processamento maciçamente paralelo (*Massively Paralell Processing* –MPP) onde, devido à ausência de um padrão, cada fabricante desenvolveu seu próprio ambiente, sem se preocupar com a portabilidade do software gerado. Com o passar dos anos, muita experiência foi adquirida, pois os diferentes projetos de passagem de mensagens enfatizavam aspectos diferentes para o seu sistema. Exemplos desses sistemas são: nCube PSE, IBM EUI, Meiko CS System e Thinking Machines CMMD [MCB94].

Na atualidade, os ambientes de passagem de mensagens foram remodelados e/ou desenvolvidos com três grandes objetivos:

- Utilizar o potencial dos sistemas distribuídos para o desenvolvimento de aplicações paralelas;
- Permitir a união de plataformas heterogêneas;
- Permitir a portabilidade das aplicações paralelas desenvolvidas.

Baseados nesses objetivos principais, vários grupos de pesquisa desenvolveram ambientes de passagem de mensagens independentes da máquina a ser utilizada. Esses ambientes foram chamados de ambientes de passagem de mensagens com plataforma portátil e puderam ser implementados em várias plataformas de hardware e sistemas operacionais.

Diversas plataformas portáteis atuais apresentam, ainda, alguns dos problemas identificados nos ambientes de passagem de mensagens propostos inicialmente. A maioria das plataformas portáteis existentes possui apenas um subconjunto das características necessárias para os mais diversos equipamentos fabricados [MCB94], dificultando a tarefa de implementação em diferentes máquinas.

Devido a esses problemas foi criado o comitê MPI (1992) para definir um padrão para os ambientes de passagem de mensagens denominado MPI (Message Passing Interface). Um dos principais objetivos é a padronização para a maioria dos fabricantes de hardware e plataformas portáteis.

O comitê MPI reúne membros de aproximadamente 40 instituições e inclui quase todos os fabricantes de máquinas com MPP, universidades e laboratórios governamentais pertencentes à comunidade envolvida na computação paralela mundial [MCB94].

O MPI é um padrão de interface de passagem de mensagens para aplicações que utiliza computadores MIMD com memória distribuída. Ele não oferece nenhum suporte para tolerância a falhas e assume a existência de comunicações confiáveis. O MPI não é um ambiente completo para programação concorrente, visto que ele não implementa: I/O paralelos, depuração de programas concorrentes, canais virtuais para comunicação e outras características próprias de tais ambientes [WAL94].

O MPI possui grupos de processos e rotinas para gerenciamento dos grupos. Os grupos podem ser usados para duas funções distintas. Na primeira, os grupos especificam os processos envolvidos em uma operação de comunicação coletiva, como um *broadcasting*. Toda a comunicação ocorre dentro e através dos grupos. Na segunda, eles podem ser usados para introduzir o paralelismo dentro da aplicação, onde diferentes grupos realizam diferentes tarefas.

3.6 Computação paralela utilizando sistemas distribuídos

O motivo principal para a criação e desenvolvimento de sistemas distribuídos foi, inicialmente, a necessidade de se compartilharem recursos, normalmente de alto custo e separados fisicamente. A computação paralela, entretanto, teve como objetivo fundamental aumentar o desempenho observado na implementação de problemas específicos [ZAL91]. Embora duas áreas tenham surgido por razões diferentes, observou-se uma rápida convergência ao longo da última década, culminando atualmente com um forte inter-relacionamento caracterizado por muitos aspectos em comum.

Essa convergência deve-se principalmente ao avanço tecnológico e às linhas de pesquisa ocorridos a partir da década de 80. A computação paralela que empregava quase que na totalidade arquiteturas SIMD, começou também a utilizar (devido à sua versatilidade e alto desempenho) máquinas MIMD com memória distribuída. Essas máquinas passaram a contar com processadores de propósito geral, tornando-se possível a sua visualização como um conjunto de computadores autônomos (com unidade de controle, unidade de processamento e memória), interligados por uma rede de comunicação.

Devido às características em comum, vários trabalhos foram desenvolvidos com o objetivo de utilizar a computação paralela sobre sistemas distribuídos onde a idéia básica é ter um grupo de computadores interligados, funcionando como elementos de processamento de uma máquina paralela.

Para a realização da computação paralela sobre sistemas distribuídos são utilizados os ambientes de passagem de mensagens citados anteriormente. Esses ambientes são aperfeiçoados constantemente, permitindo assim a união de uma quantidade de computadores cada vez mais significativa.

Apesar de possuir um meio de comunicação mais lento, tornando-se um fator que degrada rapidamente o desempenho, tais sistemas têm sido utilizados com sucesso para paralelizar aplicações que possuem granularidade grossa e pouca necessidade de comunicação entre os processos [ZAL91].

3.6.1 Vantagens e problemas da computação paralela sobre sistemas distribuídos

Em uma arquitetura maciçamente paralela, todos os processadores são exatamente iguais em capacidade, recursos, software e velocidade de comunicação. Isso não acontece em uma rede. Os computadores disponíveis em uma rede podem ser de diferentes fabricantes com diferentes compiladores. Portanto, quando se pretende utilizar um conjunto de computadores interligados por uma rede, poderão existir vários tipos como por exemplo:

- Arquiteturas;
- Formato de dados;
- Potência computacional;
- Carga de trabalho em cada processador e na própria rede;
- Sistemas operacionais de versões diferentes e não compatíveis.

O conjunto de computadores disponíveis na rede pode incluir várias arquiteturas como: computadores CISC (386, 486, Pentium), computadores RISC (Sun SPARCstations, DECstations, entre outros), multiprocessadores com memória compartilhada, etc. A máquina paralela virtual pode ser composta por computadores paralelos, e mesmo que sejam empregados apenas computadores seriais, ainda assim haverá o problema de compatibilidade no arquivo executável gerado, sendo então necessário compilar a tarefa paralela em cada máquina que compõe a máquina paralela virtual.

Computadores diferentes podem ter formatos de dados diferentes. Num sistema distribuído, é extremamente importante que os computadores se "entendam", visto que há necessidade de passagem de mensagens entre os mesmos.

Em sistemas distribuídos, normalmente, existem vários usuários trabalhando concorrentemente. Isso quer dizer que as máquinas podem estar com cargas de trabalho diferentes, dificultando o paralelismo entre os processos de máquinas diferentes.

Apesar da aplicação de paralelismo em sistemas distribuídos possuir todos esses inconvenientes, existem também muitas vantagens como:

- O custo reduzido, visto que podemos utilizar os computadores já existentes na rede;
- Se houver mais de uma arquitetura, podemos atribuir cada tarefa à mais apropriada;

- Os recursos da máquina virtual podem ser aumentados sempre, visto que a troca de uma máquina antiga por uma de tecnologia mais atual oferece melhor desempenho;
- A computação distribuída pode facilitar o trabalho corporativo.

Um dos ambientes de passagem de mensagens mais utilizados é o PVM. Diferentemente dos outros ambientes, o PVM é uma metodologia integrada para o processamento concorrente, distribuído e paralelo. Como utilizamos o PVM em nossa pesquisa, faremos no próximo capítulo uma discussão mais completa deste ambiente.

Capítulo 4

PVM – Parallel Virtual Machine

Este capítulo descreve com detalhes as principais características do PVM, salientando seus aspectos mais importantes. No apêndice A discutiremos mais detalhes sobre a utilização, erros e problemas encontrados no PVM, além de apresentar com maiores detalhes a versão para Windows® .

4.1 Considerações iniciais

PVM (*Parallel Virtual Machine*) é um conjunto integrado de bibliotecas e de ferramentas de software, cuja finalidade é emular um sistema computacional concorrente heterogêneo ou não, flexível e de propósito geral [BEG94].

Diferentemente de outros ambientes portáteis de passagem de mensagens, o PVM nasceu com o objetivo de permitir que um grupo de computadores em rede, com arquiteturas diferentes ou não, possa trabalhar cooperativamente formando uma máquina paralela virtual [GEI94].

O projeto PVM teve início em 1989 no Oak Ridge National Laboratory – ORNL. A versão 1.0 foi implementada por Vaidy Sunderam e Al Geist (ORNL) sendo direcionada ao uso em laboratório. A partir da versão 2.0 (1991), houve a interação de outras instituições (como a University of Tennessee, Carnegie Mellon University, entre outras), quando começou a ser utilizado em muitas aplicações científicas. A versão 2.0 deu início à distribuição gratuita do PVM. A versão disponível, utilizada para o desenvolvimento deste trabalho é a versão 3.4 para Windows®. O PVM foi escolhido como ambiente de passagem de mensagens, basicamente, por três motivos: é um software gratuito, compatível com Windows® e devido a sua grande aceitação no meio científico.

O PVM permite que sejam criadas aplicações nas linguagens Fortran, C e C++. A escolha por esse conjunto de linguagens se deve ao fato de que a maioria das aplicações científicas estão ou são escritas nessas linguagens.

O PVM está disponível para uma grande variedade de plataformas atualmente.

4.2 O modelo PVM

Através do PVM, uma coleção de computadores (seriais, paralelos e vetoriais) desempenham as funções de um computador com memória distribuída e com alto desempenho. O PVM fornece as funções que permitem ao usuário iniciar, comunicar, sincronizar e finalizar tarefas (*tasks*) na máquina virtual. Uma tarefa é definida no PVM como uma unidade computacional.

O modelo computacional do PVM é, portanto, baseado na opção de que uma aplicação consiste de várias tarefas. Cada tarefa é responsável por uma parte da carga de trabalho da aplicação.

Uma aplicação pode ser paralelizada por dois métodos: 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, como por exemplo entrada, processamento e saída.

O paralelismo de dados refere-se ao paradigma SIMD descrito anteriormente. O PVM permite qualquer um dos métodos, como também um método híbrido.

4.3 Componentes

O sistema PVM é composto de duas partes [GEI94]. A primeira parte é o *daemon* (Pvmd) e a segunda é uma biblioteca de rotinas com a interface PVM (Libpvm ou Libfpvm3 para Fortran e ainda Libpvm3 para versões mais atuais do PVM). O Pvmd é executado em cada *host* que compõe a máquina virtual, atuando como gerenciador da máquina e roteador de mensagens entre os processos. O termo *host* é utilizado para designar um dos computadores que formam a máquina virtual.

A Libpvm contém um conjunto de primitivas que atuam como elo de ligação entre uma tarefa e a máquina virtual (Pvmd e as outras tarefas).

4.3.1 PVM *Daemon* (Pvmd)

O Pvmd (Pvmd ou Pvmd3) foi projetado para ser instalado por qualquer usuário com um *login* válido. Quando o usuário quer executar uma aplicação PVM, ele deve em primeiro lugar disparar o *daemon*, criando assim a máquina virtual.

O Pvmd é executado em cada *host* da máquina virtual e eles são configurados para trabalharem juntos. O Pvmd que faz parte da máquina virtual de um usuário, não tem acesso a outro Pvmd pertencente a outra máquina virtual. O Pvmd foi projetado para executar sob um *login* sem privilégios, reduzindo assim os riscos de uma máquina virtual interferir na execução de outras.

O Pvmd não faz processamentos. Ele faz o roteamento e controla as mensagens, agindo como um contato entre cada um dos *hosts*, autenticando mensagens, fazendo controle de processos e detectando falhas.

O primeiro Pvmd disparado é chamado mestre (*master*), enquanto os demais (que são disparados pelo mestre) 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 a diferença aparece. Só o Pvmd mestre pode fazer isso.

As estruturas de dados mais importantes no Pvmd são as tabelas de *hosts* e de tarefas, as quais descrevem a configuração da máquina virtual e determinam as tarefas que estão sendo executadas. Ligadas a essas estão as filas de pacotes e mensagens, e os salvadores de contexto (*wait contexts*). Esses últimos são procedimentos e estruturas de dados que manipulam o estado da informação quando ocorre o escalonamento entre os processos [BEG94].

4.3.2 Biblioteca de comunicação (libpvm)

A biblioteca Libpvm foi desenvolvida com o intuito de torná-la tão pequena quanto possível, visto que compartilha espaço de endereçamento com o código desenvolvido pelo usuário (a aplicação paralela).

As rotinas podem ser chamadas pelo usuário para efetuar passagem de mensagens, criar processos e finalizá-los, coordenar tarefas, etc.

4.3.3 Identificadores de tarefas (TID)

Um identificador de tarefa (TID – *Task Identifiers*) tem por objetivo fornecer um único código identificador para cada tarefa pertencente à máquina virtual. Os TIDs atuam sobre as tarefas de um só usuário, Pvmds (mestre ou escravo) e/ou grupo de tarefas. São formados por quatro campos dispostos dentro de um dado do tipo inteiro (32 bits), o qual é disponível em uma grande quantidade de máquinas.

O TID não é um número inteiro qualquer. Ele é formado segundo algumas características da máquina virtual (como por exemplo o número do *host*).

4.4 Tratamento das mensagens PVM

O PVM possui rotinas para o empacotamento, envio e desempacotamento de mensagens entre tarefas. O modelo assume que qualquer tarefa pode enviar uma mensagem para outra e que não há limites para o tamanho ou número de tais mensagens.

Sob a ótica do usuário PVM, enviar uma mensagem compreende, basicamente, três passos:

- Um *buffer* deve ser criado para que as mensagens enviadas sejam depositadas temporariamente;
- A mensagem deve ser "empacotada" dentro do *buffer*;
- A mensagem inteira (todo o conteúdo do *buffer*) é enviada para outra tarefa ou grupo de tarefas.

A mensagem é recebida por uma função bloqueante ou não e então "desempacotada", retirando do *buffer* (no *host* receptor da mensagem) os dados enviados. As rotinas que manipulam as mensagens recebidas podem aceitar:

- Quaisquer mensagens;
- Quaisquer mensagens de uma tarefa específica;
- Quaisquer mensagens com um TID específico;
- Somente mensagens de uma tarefa específica com um TID específico.

Para o usuário PVM, a comunicação empregada utiliza *send* bloqueante assíncrono, *receive* bloqueante assíncrono e *receive* não bloqueante. O *send* é chamado bloqueante porque retorna tão logo o *buffer* utilizado para enviar a mensagem esteja livre para ser novamente utilizado e é chamado de assíncrono porque não depende do receptor executar um *receive* para retornar. O *send* só bloqueia quando a mensagem exceder o tamanho do *buffer* e precisar ser dividida. Nesse caso o *send* fica bloqueado até o receptor executar um *receive* e liberar o *buffer* para continuar o envio da mensagem.

Um *receive* é não bloqueante quando ele retorna imediatamente após ter verificado o *buffer* no *host* receptor. Um *receive* bloqueante não retorna enquanto a mensagem esperada não for recebida e inserida no *buffer* [GEI94, BEG94, SUN94].

O PVM oferece comunicação ponto-a-ponto, *broadcasting* (para um grupo de tarefas) e *multicasting* (para um conjunto de tarefas).

4.4.1 Descritores de fragmentos de mensagem e *databufs*

O PVM possui mensagens dinâmicas, ou seja, elas podem aumentar seu tamanho conforme a necessidade, sem saber *a Priori* qual será o seu tamanho máximo. As funções de "empacotamento" utilizam blocos de memória (ou fragmentos) de acordo com o tamanho da mensagem a ser enviada. Esses blocos são chamados de *Databufs* e utilizam os descritores de fragmentos de mensagem (*struct frag*) para uni-los, formando assim uma mensagem.

No início de cada *databuf* é reservado um espaço para o cabeçalho de cada fragmento e da mensagem, bem como um contador (*refcount*) o qual tem por finalidade indicar quantas vezes o *databuf* está sendo referenciado. Isso tem como objetivo evitar a duplicação desnecessária dos dados. Quando o contador possui valor 0(zero), o *databuf* pode ser eliminado. Essa estratégia é bastante empregada no PVM.

O tamanho de cada *databuf* é configurado em tempo de compilação mas pode também ser alterado dinamicamente (em tempo de execução).

4.5 Pvmd – detalhes de implementação

Embora o código o Pvmd tenhas várias características importantes, são relacionadas nesta seção apenas as mais importantes.

4.5.1 Iniciar e finalizar o Pvmd

Quando o Pvmd é iniciado, ele é configurado ou como mestre ou como escravo, dependendo dos argumentos. Como a finalidade principal é o roteamento de mensagens, o primeiro passo tomado pelo Pvmd é criar mecanismos de comunicação entre os processos (*sockets*) e permitir a sua utilização. Isso possibilita a comunicação com as tarefas e os outros Pvmds. Num segundo momento, o Pvmd cria as estruturas de dados que irão conter as informações sobre os outros *hosts* e sobre as tarefas que estão sob sua supervisão.

Depois dos acertos citados acima, o Pvmd entra em *loop*, com a finalidade de aguardar o recebimento de mensagens (locais ou de outros Pvmds). Quando os pacotes são recebidos, ou eles são colocados em filas para posterior envio ou são recuperados pelo Pvmd.

Quando o Pvmd é finalizado, ele realiza duas ações: envia um sinal às tarefas a ele subordinadas com o sentido de finalizá-las (matá-las), e ainda informa a todos os outros Pvmds que ele está saindo da máquina virtual. Apesar dos outros Pvmds conseguirem descobrir sozinhos a ausência do Pvmd finalizado (através de uma verificação feita a cada período de tempo), isso agiliza e muito esse processo.

Caso o Pvmd mestre seja finalizado, todos os demais Pvmds serão "mortos".

4.5.2 *Shadow* Pvmd

O Pvmd *shadow* (Pvmd') é executado no *host* mestre e tem como finalidade iniciar todos os novos *hosts* escravos. O motivo principal para a existência do Pvmd' é liberar o Pvmd mestre para as outras atividades enquanto a inicialização dos Pvmds escravos está em andamento.

As atividades envolvidas na criação dos Pvmds escravos (como executar um interpretador de comandos (*Shell*) em uma máquina remota) podem bloquear o Pvmd mestre por um intervalo de tempo variado (podendo ser pequeno ou grande), e este deve estar apto a responder outras mensagens durante esta atividade.

O Pvmd' é chamado através do número 0(zero), ou seja, o Pvmd' é o Pvmd de número 0, e embora ele nunca envie e nem receba nenhuma mensagem de qualquer tarefa ou outro Pvmd.

O Pvmd' só é executado quando há a necessidade de se iniciar novos Pvmds escravos. Quando sua tarefa é finalizada, ele é eliminado da memória.

4.5.3 Tabela de *hosts*

A configuração atual da máquina virtual é uma das principais finalidades da tabela de *hosts*. Ela gerencia várias filas de pacotes e *buffers* de mensagens. Com ela o Pvmd tem condições de manipular conjuntos de *hosts* que são candidatos em potencial para iniciar as tarefas geradas pelo usuário.

As tabelas de *host* são mantidas sincronizadas em toda a máquina virtual, embora isto não possa ser verdade durante todo o tempo de execução. Quando ocorrem determinados problemas com um *host* e ele é retirado de operação, podendo demorar algum tempo para que os outros Pvmds notem a sua falta (através de mensagens de controle). Nesse intervalo de tempo, as tabelas de *hosts* não estão sincronizadas.

O Pvmd mestre é responsável pela atualização da tabela de *hosts*, usando para isso mensagens. Ele inclui ou retira qualquer posição da tabela.

4.5.4 Tabela de tarefas

A tabela de tarefas é formada por uma lista de *task structs*, as quais contêm informações sobre cada tarefa em execução. Essa tabela é necessária para que o Pvmd saiba como está o processamento das tarefas que estão sob sua supervisão.

As tarefas normalmente são iniciadas pela função pvmfspawn(), onde o Pvmd responsável pelo disparo da mesma é o pai do processo.

4.5.5 Salvadores de contexto

"Salvadores de contexto" (*wait contexts*) são utilizados para armazenar o estado da informação quando uma determinada operação no Pvmd deve ser interrompida. Quando um Pvmd necessita se interagir com outro Pvmd, este não pode ser bloqueado, no sentido de aguardar pela resposta. Obviamente que, se o Pvmd é responsável pelo roteamento das mensagens, ele não deve ficar inerte aos processos. Assim, o Pvmd salva todas as informações

pertinentes à operação que ele estava realizando, utilizando as estruturas de salvadores de contexto, e retorna ao *loop* aguardando por outras mensagens.

Quando a resposta do Pvmd remoto chega, o Pvmd lê a informação necessária do salvador de contexto e em seguida envia a resposta para a tarefa que solicitou a informação.

Os salvadores de contexto são codificados seqüencialmente e esse número é enviado no cabeçalho da mensagem com a requisição e retorna com a resposta. São previstas também operações compostas, ou seja, realizadas em mais um fase.

O custo computacional envolvido no armazenamento das informações com os salvadores de contexto não é muito alto (se comparado com o tempo que o Pvmd ficaria bloqueado, caso não existisse este recurso), visto que, para a maioria das operações, as únicas informações salvas são o TID e o tipo da operação.

4.5.6 Tolerância a falhas

O PVM (versão 3 ou superior) foi projetado para resistir à maioria das falhas envolvendo *hosts* e redes. O PVM fornece todos os recursos necessários para que o usuário construa aplicações tolerantes a falhas, apesar de recuperar automaticamente uma aplicação após um erro.

Se um *host* escravo perde a comunicação com o mestre, ele mesmo (o escravo) provoca sua saída da máquina virtual, eliminando todas as tarefas e operações pendentes.

Se o *host* mestre perde a comunicação com um *host* escravo, este é retirado da máquina virtual pelo mestre.

Se o *host* mestre é perdido, toda a máquina virtual é finalizada. Com este sentido não há nenhuma implementação no PVM.

4.5.7 Roteamento de pacotes e mensagens

Os pacotes recebidos e/ou enviados pelos Pvmds são manipulados por *buffers* de pacotes. Sua estrutura é análoga a dos fragmentos de mensagens descritos anteriormente, porém, eles também incluem informações necessárias à realização do protocolo Pvmd⇔Pvmd.

O Pvmd envia mensagens através da função *sendmessage*( ) a qual irá roteá-las para o destino. Quando enviada para outro Pvmd ou para tarefas, a mensagem é anexada aos *buffers*

de pacotes, que são inseridos em filas para posterior envio. Se a mensagem tem como destino o próprio Pvmd, ela é colocada diretamente no ponto de entrada das mensagens vindas da rede, economizando assim todo o custo computacional de fragmentação da mensagem em pacotes e o envio pela rede.

Depois de receber os pacotes, o Pvmd transforma-os em mensagens através de duas funções: uma analisa pacotes vindos de outros Pvmds e a outra analisa os pacotes vindos de tarefas locais.

4.6 Libpvm – detalhes de implementação

A Libpvm foi desenvolvida utilizando-se a linguagem C e permite que as aplicações dos usuários sejam desenvolvidas em C, C++ e Fortran. A versão da Libpvm para Fortran (Libfpvm3) é também escrita em C, porém, as funções que fazem a interface com o usuário foram desenvolvidas segundo o padrão de chamadas Fortran, permitindo dessa forma uma ligação entre as linguagens.

A Libpvm contém 82 funções disponíveis para o usuário [BEG94, GEI94]. Elas compreendem uma ligação entre os processos na máquina virtual. Simples e fáceis de usar, as funções da Libpvm tornaram o PVM fácil de ser usado e, portanto, tão bem aceito e utilizado.

4.7 Protocolos de comunicação

A comunicação realizada pelo PVM é baseada em TCP (*Transmission Control Protocol*), UDP (*User Datagrama Protocol*) e *sockets* do domínio UNIX, assumindo, portanto, que todos os *hosts* pertencentes à máquina virtual sejam capazes de conectarem-se através desses mecanismos de comunicação. Algumas máquinas multiprocessadoras, entretanto, não possuem *sockets* disponíveis em seus elementos de processamento, possuindo *hosts* que atuam como *front-end*, onde é possível executar tais mecanismos de comunicação.

O PVM procura evitar modificações no núcleo (*kernel*) do sistema operacional, executando seus protocolos como processos normais (Pvmd e tarefas). Isso faz com que o desempenho da passagem de mensagens seja reduzido. O trabalho de gerenciar a memória, executar o chaveamento de contextos e operações de cópia, quando feitas no espaço do usuário, tornam-se caras. O desempenho seria muito maior se o código estivesse no *Kernel* ou se a interface de rede estivesse disponível diretamente para os processos, ignorando o *Kernel*.

A comunicação no PVM é feita entre os Pvmds e as tarefas. Portanto, há três conexões a considerar: entre Pvmds, entre um Pvmd e sua(s) tarefa(s) e entre tarefas.

4.8 PVM em sistemas multiprocessadores

Desenvolvido inicialmente para executar em estações de trabalho com UNIX, o PVM foi também adaptado para as máquinas com multiprocessadores, devido à necessidade de seus usuários executarem aplicações específicas.

Uma das principais características do PVM é a sua portabilidade. Isso permite que as aplicações desenvolvidas para as estações de trabalho possam também ser executadas em computadores MPP (desde que recompilados).

Computadores com memória compartilhada podem ser utilizados para as aplicações com menor granularidade.

A máquina virtual esconde os detalhes de configuração do usuário. Os processadores físicos podem ser: uma rede de estações de trabalho ou nós de um computador com multiprocessadores. O usuário não sabe como e onde as tarefas são executadas. O PVM assume esta responsabilidade. Entretanto existe a possibilidade, dependendo da aplicação, de se especificar uma configuração desejada para determinadas tarefas.

4.9 Limitações

O PVM foi projetado para, sempre que possível, não impor nenhum tipo de limitação de acesso aos recursos. Normalmente os limites são do próprio *hardware* e sistema operacional utilizados.

4.9.1 Limitações no Pvmd

O número 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.

A quantidade de tarefas do PVM é limitada por dois fatores. O primeiro está relacionado com o número de processos concorrentes permitidos pelo sistema operacional, apesar de não fazer nenhum sentido um grande número de processos rodando 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 simultaneamente.

O Pvmd aloca memória dinamicamente para armazenar as mensagens roteadas por ele. Enquanto a tarefa destino não aceitar a sua mensagem, os pacotes são armazenados em filas no Pvmd, sendo que não há nenhum controle 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.

4.9.2 Limitações na Libpvm

As tarefas PVM também possuem limite no número de conexões diretas que elas podem fazer.

A maior mensagem possível para uma tarefa PVM é limitada pela quantidade de memória disponível para a tarefa designada. Quando a mensagem é enviada via Pvmd, este aloca memória para então roteá-la, diminuindo o tamanho da memória disponível previamente.

Se muitas tarefas enviam mensagens simultaneamente para a mesma tarefa destino, tanto o Pvmd quanto a tarefa destino ficarão sobrecarregados no sentido de armazenar as mensagens que chegam. Esses problemas devem ser analisados quando no projeto da aplicação, evitando mensagens longas e eliminam "gargalos" no processamento.

4.10 O PVM para Windows®

O PVM para Windows® (PVMWIN) fornece os recursos necessários para que programas concorrentes possam ser desenvolvidos e executados em computadores pessoais (PCs), com a utilização do sistema operacional Windows® e conectados por uma rede de comunicação, de modo análogo ao PVM.

O objetivo principal é, portanto, viabilizar a computação paralela para um maior número de usuários, os quais não têm acesso a uma plataforma UNIX e/ou computadores paralelos. São exemplos de usuários do PVMWIN:

- Cursos de programação concorrente na maioria das universidades, as quais não possuem laboratórios de ensino/pesquisa equipados adequadamente;

- Laboratórios e empresas de engenharia, automação, simulação, entre outros, os quais utilizam, em sua maioria, PCs com Windows® .

A escolha pelo *Microsoft Windows®* deve-se principalmente a sua popularidade e vasta utilização [KIN95]. A necessidade de um hardware com uma configuração simples, como por exemplo pouca necessidade de memória RAM, também contribuiu para a escolha desta plataforma.

O Windows® também facilitou a instalação dos protocolos de comunicação utilizados (como TCP/IP), pois eles, além de já serem fornecidos com o mesmo, também são fáceis de serer instalados, visto que eles fazem parte das rotinas de instalação. Maiores detalhes sobre o PVMWIN podem ser vistos no apêndice A.

4.11 PVM UNIX x PVMWIN

O UNIX e o Windows® possuem inúmeras diferenças, porém, nem todas são percebidas, ou seja, não afetam o PVMWIN. Isso se deve ao fato de que nem todos os recursos do UNIX são utilizados pelo PVM e muitos dos recursos utilizados no UNIX possuem um comando similar no Windows®.

Dentre as principais diferenças percebidas (necessárias ao PVM), destaca-se a ausência do RSH (*Remote Shell*). O RSH é o responsável pela comunicação entre os Pvmds.

O XPVM é um aplicativo desenvolvido sobre a Libpvm e com uma interface gráfica no padrão X-Windows . O XPVM permite uma melhor visualização de todos os recursos disponíveis no Pvmconsole, incluindo ainda monitoramento das tarefas, desempenho das máquinas envolvidas na máquina paralela virtual e o *status* de cada *host*, mostrando quais as mensagens estão sendo enviadas. Entretanto no site do PVM (http://www.epm.ornl.gov/pvm/pvm_home.html) podemos encontrar uma mensagem que fala sobre a implementação do XPVM para Windows®.

4.12 Considerações finais

O PVM apresenta características como simplicidade no uso de suas funções, robustez e portabilidade, destacando-se na literatura e na comunidade de computação paralela como um ambiente de passagem de mensagens amplamente discutido e utilizado. Entre alguns dos

usuários PVM estão a Ford, Boeing, Texaco, General Eletric, Siemens, Mobil Oil, Cray Research, Shell Oil, IBM, entre outros [GEI95].

Os principais componentes do PVM são o PVM Daemon (Pvmd) e a biblioteca de comunicação (Libpvm), responsáveis respectivamente pela composição da máquina virtual e pela interface com o usuário.

O principal problema do Pvmd é o *overhead* gerado em cada mensagem roteada por ele. Suas estruturas, apesar de serem robustas, impõem um atraso significativo na troca de mensagens.

A Libpvm com suas 82 funções disponíveis às aplicações dos usuários desempenha bem o seu papel. Ela possibilita o roteamento direto entre as tarefas reduzindo assim o *overhead* evitando a utilização do Pvmd. A única ressalva é que o número de ligações pode ser limitado pelo sistema operacional utilizado, impedindo que uma tarefa conecte-se a um grande número de tarefas simultaneamente.

O PVM tem também a vantagem de não impor limites aos recursos utilizados. Os recursos de hardware e software é que são o limite da aplicação. A aplicação paralela é que deve tratar os erros referentes a este problema.

4.13 Um programa exemplo de utilização da Libpvm em Fortran

Podemos tomar como exemplo um somatório, para exemplificarmos a proposta de paralelismo, que fica no intervalo, fechado, de 1 a 1000.

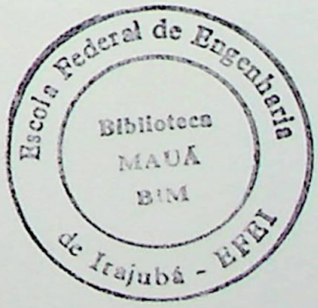
Programa Serial em Fortran

```
Program serial
integer  inicio, fim, resultado  ! definição das variáveis de tipo inteiro
fim=1000
do inicio=1, fim   ! início do loop que faz o somatório de 1 até 1000
        resultado = resultado + inicio
enddo   ! fim do loop
print *, resultado     ! imprimindo o resultado
pause
end
```

Programa paralelo em Fortran utilizando a biblioteca PVM (master)

```
Program master
Include 'fpvm3.h'    ! inclusão do header de abertura do PVM
Integer soma, Info, mytid, TID, vinicio, vfim
vinicio=500
vfim=1000

Call pvmfmytid(mytid)     ! Recebendo a identificação do processo
Call pvmfspawn("slave", 0,"*", 1, mytid, Info)   ! Disparando o processo
Call pvmfinitsend(PVMDEFAULT, Info)   ! Inicializando buffer
Call pvmfpack(INTEGER4, vinicio, 1, 1, Info)   ! empacotando vinicio
Call pvmfpack(INTEGER4, vfim, 1, 1, Info)   ! empacotando vfim
Call pvmfsend(mytid, 1, Info)   ! enviando vinicio e vfim
Do i=1, 499   ! somatório de 1 a 499
        Soma = soma + i
Enddo
Call pvmfrecv(-1,2,info)   ! recebendo a variável vfim já com o resultado final
Call pvmfpack(INTEGER4, vfim,1 , 1, Info)   ! desempacotando vfim
Soma = soma + vfim   ! resultado final
Print *, soma
Pause
Call pvmfexit(info)   ! finalizando o PVM
end
```



Programa paralelo em Fortran utilizando a biblioteca PVM (slave)

```
Program slave
Include 'fpvm3.h'
Integer soma, mytid, minha, Info, vinicio, vfim
Call pvmfmytid(mytid)   ! Recebendo a Identificação do processo
Call pvmfparent(minha)  ! Recebendo a identificação do processo mestre
Call pvmfrecv(-1, 1, Info)  ! Recebendo a mensagem da rotina master
Call pvmfunpack(INTEGER4, vinicio, 1, 1, Info) ! desempacotando variável
Call pvmfunpack(INTEGER4, vfim, 1, 1, Info) ! desempacotando variável
Do i=vinicio, vfim
        Soma = soma + i
Enddo
vfim=soma
Call pvmfinitsend(PVMDEFAULT, Info)  ! Inicializando buffer
Call pvmfpack(INTEGER4, vfim, 1, 1, Info) ! Empacotando variável
Call pvmfsend(minha, 2, Info) ! enviando variável vfim para Master
Call pvmfexit(Info) ! finalizando o processo slave
end
```

As rotinas acima fazem o mesmo cálculo do programa serial, porém paralelamente. A rotina master foi escrita tendo como base a utilização de duas máquinas. Isso explica o motivo pelo qual o somatório de 1 a 499 foi executado na rotina master.

Capítulo 5

O Sistema Elétrico de Potência

5.1 Introdução

Um sistema elétrico de potência (SEP) tem como objetivos: gerar energia elétrica em quantidades suficientes e nos locais mais apropriados, transmiti-la aos centros de carga em forma e qualidade apropriada [ELG76].

Um SEP adequadamente projetado e operado deve ter as seguintes características :

- Fornecer energia elétrica em todos os locais necessários;
- Como a carga alimentada é variável, o SEP deve estar apto a se adequar a esta demanda;
- Fornecer energia elétrica com qualidade. Três variáveis básicas determinam esta qualidade: freqüência constante, tensão constante e confiabilidade;
- Fornecer energia elétrica com custos mínimos, tanto econômicos, como ecológicos.

5.2 Análise do estado de operação do sistema de potência

Esta análise é realizada no sentido de avaliar e monitorar o desempenho dos sistemas de potência na ocorrência de distúrbios como: variação brusca de carga, perturbações no sistema de transmissão ou até mesmo em condições normais de operação. São exemplos destes estudos: análise do fluxo de potência, estudos de curto-circuito, estudos de sobretensão à freqüência fundamental, estudos de estabilidade eletromecânica, análise de contingências, análise de sensibilidade de tensão, etc.

Como este trabalho se propõe a paralelizar um processo computacional associado a fluxo de carga em redes elétricas, as próximas seções abordarão este tópico de uma forma resumida, porém suficiente para se apresentar o problema.

5.3 Fluxo de potência em regime permanente

O cálculo do fluxo de potência (*load flow*) em um SEP se fundamenta na determinação do estado de operação deste sistema, dada sua topologia, a uma certa condição de carga e de geração. Este estado de operação consiste de :

- Determinação das tensões e ângulos para todas as barras do sistema;
- Determinação dos fluxos de potência ativa e reativa através dos ramos (linhas de transmissão e transformadores) do sistema.
- Determinação da potência ativa e reativa geradas, consumidas e perdidas nos diversos elementos componentes do sistema elétrico.

Em termos gerais, o problema do fluxo de potência é não linear, necessitando de processos iterativos de cálculo numérico para a resolução do problema. A não linearidade das equações decorre de certas características da modelagem de alguns componentes do sistema.

O interesse da análise de fluxo de potência é obter uma solução do sistema operando em regime permanente senoidal. Sendo assim, a modelagem do sistema é estática, significando que as equações e inequações representativas da rede são algébricas e não diferenciais.

5.3.1 Suposições e aproximações

Em cálculos de fluxo de potência são feitas, usualmente, as seguintes aproximações:

- As cargas ativas e reativas nos barramentos do sistema são supostas constantes;
- Torna-se suficiente uma representação unifilar para a rede, pois admite-se que a mesma opere de maneira equilibrada em suas três fases;
- Os elementos passivos do sistema são representados com parâmetros concentrados.

5.3.2 Representação dos componentes

Abaixo segue a representação de alguns dos principais elementos que compõem um SEP.

- **Geradores**

Os geradores são elementos capazes de fornecer potência ativa e fornecer/absorver potência ativa e reativa. O gerador deve fornecer potência ativa e reativa ao barramento ao qual está conectado, além de controlar o módulo da tensão neste barramento.

- **Linhas de transmissão**

As linhas de transmissão são responsáveis pelo transporte da potência através dos sistemas. Os dados típicos da linha são a resistência, reatância e susceptância, sendo que este último chamado de *line charging* da linha, ou seja, o montante de potência reativa que a própria linha gera.

- **Transformadores**

Os transformadores são os responsáveis pela troca (elevação ou abaixamento) de níveis de tensão ao longo do sistema e seus dados típicos são sua reatância de dispersão, tapes, tensão e potência.

- **Cargas**

São representadas pelas potências ativas e reativas consumidas, supostas constantes.

5.3.3 Resumo

O estudo do fluxo de potência tem como objetivo calcular o estado operativo

da rede elétrica para dadas condições de carga, geração, topologia e determinadas restrições operacionais.

5.4 Análise de contingências em sistemas de energia elétrica

Todo SEP num ponto de operação qualquer, está sujeito a sofrer uma série de possíveis perturbações que podem provocar mudanças na configuração do sistema em questão, assim como também danos nos equipamentos e deficiências na distribuição de energia aos pontos ou regiões do sistema afetados pela perturbação [AST82].

Entende-se por contingência num SEP, a saída ou entrada no sistema de um ou mais componentes.

Um estudo de contingência consiste numa avaliação da segurança da operação de um determinado estado permanente, dando capacidade ao sistema de suportar contingências não planejadas sem haver perdas parciais ou totais de carga ou transgredir as restrições de operação. As restrições de operação estipulam as variações de freqüência e tensão dentro de um intervalo tolerável e preestabelecido para que os componentes não tenham seus limites de operação transgredidos.

5.4.1 Principais causas

Um SEP sofre contínuas variações em seu estado permanente de operação. As causas dessas mudanças podem ser as mais diversas:

- Alterações nas demandas das cargas;
- Reprogramação da geração;
- Desconexões de linhas de transmissão ou transformadores para manutenção ou por atuação do sistema de proteção;
- Perturbações que resultam da ação combinada das anteriores.

5.4.2 Estados de segurança

O grau desejado de segurança de um sistema é proveniente dos critérios adotados. Podemos defini-los em quatro estados:

- **Estado normal -** Onde a demanda e as restrições de operação estão satisfeitas e, no caso de uma contingência, o sistema continua operando, respeitando as restrições de carga e operação.

- **Estado de alerta** – Ainda são satisfeitas a demanda e restrições de operação, mas a próxima contingência levará o sistema para um estado de emergência.

- **Estado de emergência** – Neste estado algumas cargas não são atendidas e/ou os componentes estão sobrecarregados, ou a qualidade do serviço, em termos de tensão e freqüência, vem deteriorando.

- **Estado de restauração** – É o estado seguinte à emergência, no qual a deterioração das características do sistema foi detida, mas as condições ainda não são normais.

Os estados de operação e as transições entre eles podem ser visualizados na figura 5.1. As transições são causadas por contingências ou ações de controle.

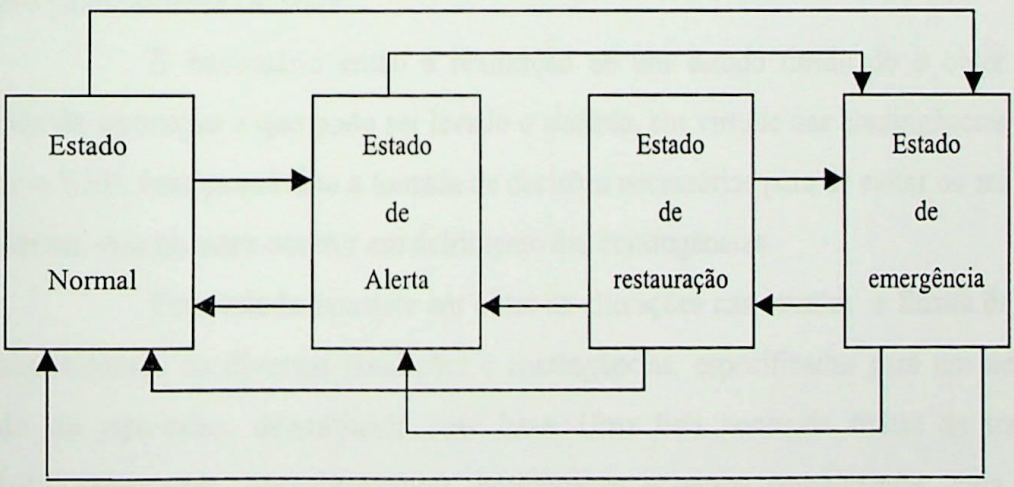


Fig. 5.1 – As transições para a direita são causadas por contingências e as para a esquerda por ações de controle.

5.4.3 Análise de contingência estática

O estudo de contingências visa principalmente determinar se um sistema a ser instalado (fase de planejamento) ou um sistema em funcionamento (fase de operação) oferece a segurança necessária que garanta o seu bom funcionamento, em virtude das perturbações já citadas.

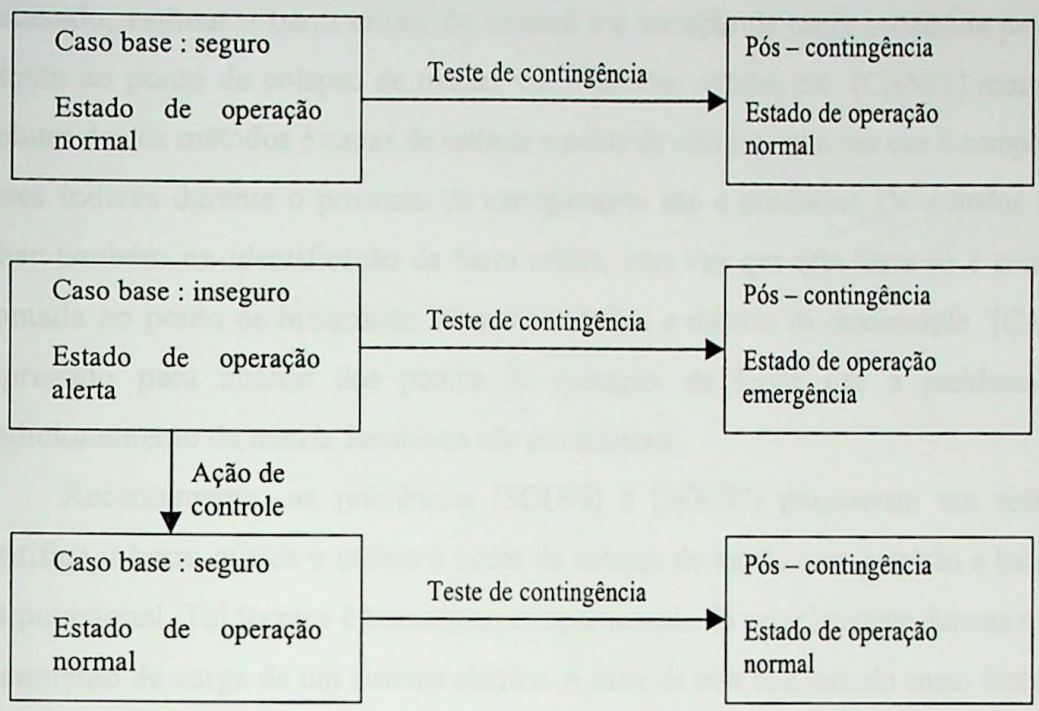
O estudo de contingências é então um estudo de segurança de um SEP, dentro do que se denomina como segurança em estado permanente, o que significa que o interesse do estudo está voltado para o período após o transitório provocado pela contingência, de forma a verificar se neste novo estado de operações (que pode ou não acontecer), existem modificações que afetem o sistema , ou parte dele, ou seus equipamentos; estas modificações podem ser sobrecargas, sobretensões, etc.

Uma forma muito eficaz para melhorar a segurança dos sistemas de potência é através da redundância de equipamentos, de maneira que, no caso de perda de um equipamento, outro similar possa entrar no lugar do mesmo, mantendo a configuração original. Como exemplo podemos citar linhas e transformadores em paralelo. Torna-se evidente no entanto que esta maneira de tratar o problema aumenta absurdamente o custo do projeto. A redundância citada, em alguns casos, é impossível tecnicamente, além de tornar o projeto praticamente inviável.

É necessária então a realização de um estudo destinado a obter os novos estados de operação a que pode ser levado o sistema, em virtude das contingências que possa sofrer o SEP. Isso possibilita a tomada de decisões necessárias para se evitar ou minimizar os problemas que possam ocorrer em detrimento das contingências.

Este estudo consiste em obter as alterações nas tensões e fluxos de potências correspondentes às diversas condições e contingências, especificadas para um determinado estado de operação, denominado caso base. Uma lista contendo linhas de transmissão, unidades geradoras, transformadores, barramentos e outros componentes, cuja saída, ou entrada, do sistema devem ser simuladas e para cada caso a solução do fluxo de carga é obtida.

O caso base é considerado seguro se, nos testes efetuados, o novo estado de operação for normal. No caso de ser detectado, após a contingência, um estado de emergência, o caso base não é mais seguro e ações de controle devem ser tomadas para levá-lo a um estado normal de operação.



Um estado de contingência, entretanto, não fornece nenhuma indicação direta das correções a serem empregadas. Apenas apresenta um diagnóstico da segurança da operação do sistema. A alternativa mais viável deverá ser posteriormente analisada.

Uma análise de contingência pode ser efetuada tanto no planejamento com o sistema desenergizado, ou em operação com o sistema já energizado. Qualquer que seja o caso, para que esta seja viável, é necessária uma solução rápida em termos computacionais.

5.4.4 Formulação matemática do problema

Diversos índices baseados em análise estática têm sido propostos na literatura. As referências [LOF92] e [BAR95] propõem avaliar cada ponto de operação através do cálculo do menor valor singular da matriz Jacobiana, enquanto [GAO92] e [MAR94] avaliam o ponto de operação através do cálculo do menor autovalor. A referência [PRA91] propõe a redução da matriz Jacobiana às derivadas parciais de potências ativa e reativa de cada barra em relação a seus respectivos ângulo de fase e módulo da tensão. A barra associada ao menor determinante reduzido seria crítica naquele ponto de operação. A referência [CAN93] testou

os métodos anteriormente citados com o objetivo de, a partir de um ponto de operação conhecido, estimar a barra crítica do sistema e a variação de carga necessária para levar o sistema ao ponto de colapso de tensão. Os resultados obtidos em [CAN95] mostram que nenhum destes métodos é capaz de estimar o ponto de colapso, uma vez que o comportamento destes índices durante o processo de carregamento não é previsível. Os métodos tendem a falhar também na identificação da barra crítica, uma vez que esta barra só é corretamente apontada no ponto de bifurcação. Naquele trabalho, o método da continuação [CAN93] foi empregado para análise dos pontos de operação, de forma que o problema de mal condicionamento da matriz Jacobiana não se manifesta.

Recentemente, as referências [SOU96] e [SOU97] propuseram um método que identifica a barra crítica e estima o ponto de colapso de tensão com precisão e baixo tempo computacional. Tal técnica é baseada no comportamento do vetor tangente durante o processo de aumento de carga de um sistema elétrico. A idéia de usar este método como ferramenta de análise de colapso de tensão foi estendida em [SOU98], onde uma análise sobre os efeitos das perdas elétricas sobre o fenômeno de colapso de tensão foi desenvolvida. É mostrado em [SOU98] que, muito embora o ponto de colapso de tensão possa estar associado a altas perdas elétricas, as ações de controle requeridas para reduzir as perdas do sistema podem não ser as mesmas ações recomendadas para melhorar a margem de carga. Por margem de carga, entende-se a variação de carga necessária para levar o sistema ao ponto de bifurcação.

Neste trabalho, o vetor tangente é usado uma vez mais como ferramenta de análise de colapso de tensão. Desta vez, a norma do vetor tangente será usada como medida relativa de segurança. Sabe-se que esta norma tende a um valor muito elevado no ponto de bifurcação, uma vez que pequenas variações de carga produzem grandes alterações nas variáveis de estado. Esta característica permite que se monitore a norma deste vetor como índice de severidade de segurança do sistema frente a mudanças topológicas. Para cada falta, calcula-se o novo ponto de equilíbrio e a norma do vetor tangente. As contingências associadas às maiores normas são consideradas críticas. Note que este tipo de análise de contingência não contempla o regime transitório associado a cada falta, onde as características dinâmicas do sistema devem ser consideradas [MAN95] e [JAR94]. Neste processamento são selecionadas as contingências mais severas.

A cada contingência, é necessário calcular de fluxo de carga. Como em um SEP pode haver muitos componentes envolvidos, o tempo pode ser muito elevado. Diante disso é que no

próximo capítulo será mostrado que técnicas de paralelismo aplicadas à análise de contingência podem ser extremamente úteis, fazendo com que o tempo seja reduzido.

Capítulo 6

Avaliação de desempenho da aplicação de análise de contingência estática utilizando paralelismo

São apresentados uma descrição das aplicações serial e paralela, os resultados alcançados bem como uma discussão sobre os mesmos.

6.1 Considerações iniciais

Manter elevados os níveis de confiabilidade nos sistemas de potência torna-se cada vez mais difícil e necessário, à medida que os sistemas crescem. Os centros de controle e supervisão desempenham o papel de vigia permanente do sistema, monitorando a segurança da operação. Para coordenar as decisões requeridas em cada situação, os centros dispõem de setores informatizados para análises mais rápidas e confiáveis. Disso dependem as decisões a serem tomadas sobre como alterar o ponto de operação, conforme necessário, através de ações de controle [DEC79].

A análise de contingências é uma das ferramentas utilizadas para manter o sistema elétrico de potência dentro dos padrões de confiabilidade necessários. A técnica empregada neste trabalho é baseada no comportamento do vetor tangente, devido à precisão na resposta e baixo esforço computacional [SOU98].

Dada a essa importância, é então necessário que se disponha de uma ferramenta eficaz para a realização dos cálculos, e que em razão da grandeza do problema deve ser a mais rápida possível.

Considerando-se que um dos objetivos principais da computação paralela é melhorar o desempenho das aplicações seqüenciais e que o PVM para Windows® permite o desenvolvimento e execução de aplicações paralelas sobre uma plataforma distribuída,

mostraremos que a utilização de técnicas de paralelismo em uma ferramenta de análise de contingência é viável, devido ao menor tempo de processamento, se comparada a uma aplicação serial com a mesma atribuição.

As avaliações foram feitas comparando os tempos da aplicação serial e paralela, levando-se em consideração os seguintes aspectos:

- O número de barras existentes no sistema elétrico que será processado;
- O número de máquinas utilizadas no processamento (no caso da aplicação paralela).

6.2 A aplicação serial utilizada

A aplicação serial utilizada para avaliar a proposta de paralelismo já foi amplamente testada. Como toda aplicação serial, não foi levada em consideração qualquer tipo de implementação no sentido de posteriormente ser paralelizada. Isso dificultou bastante a análise e implementação do código fonte com técnicas de paralelismo. A aplicação é composta basicamente das subrotinas apresentadas na figura 6.1.

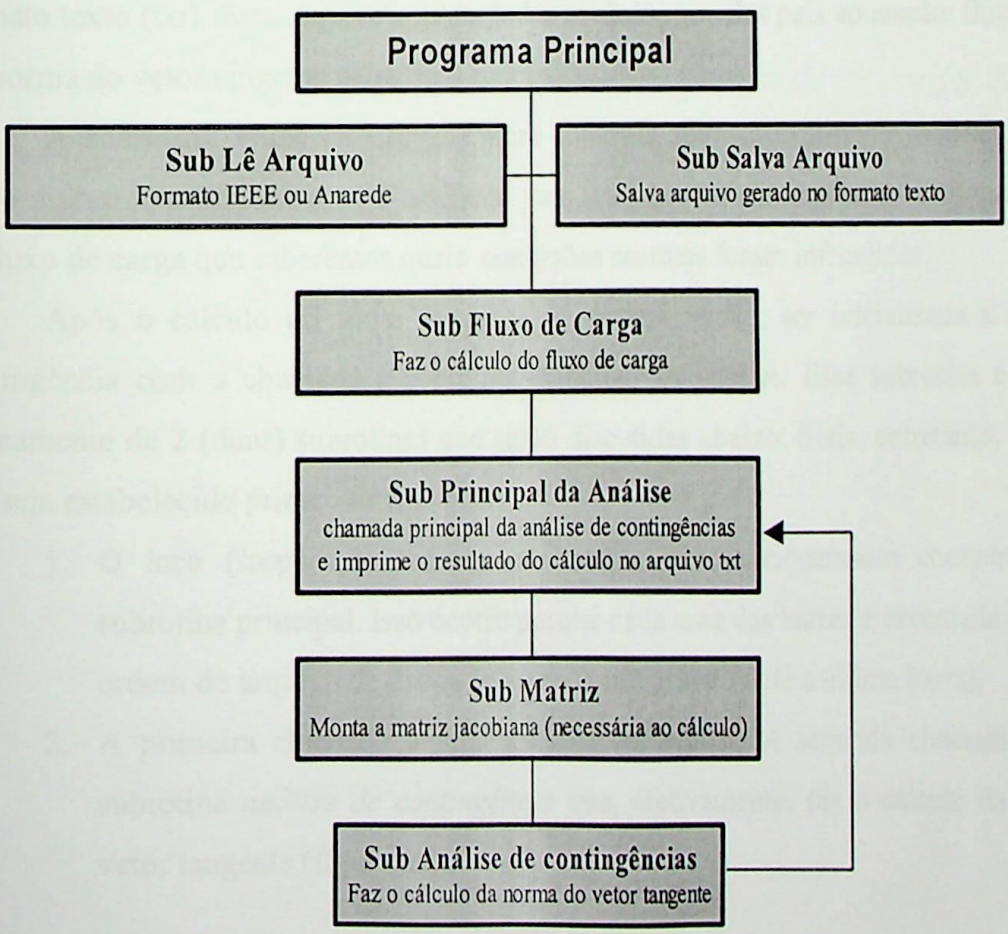


Figura 6.1 Subrotinas da Aplicação Serial

```
IF(IF2.EQ.24)THEN
        PRINT *, ' SISTEMA DE EQUACOES DIVERGE '
        AS=99.
ELSE
        CALL TTCONT
END IF
```

Figura 6.2 (chamada a rotina do cálculo da norma do vetor tangente *ttcont*)

O programa principal (*main.for*) é composto basicamente de rotinas que fazem as declarações das variáveis, chamada às rotinas externas do Fortran e gerenciamento da aplicação como um todo.

A subrotina *lê arquivo* é uma interface com o usuário, onde ele identifica o arquivo que contém os dados necessários para a análise. Esses arquivos de dados têm dois formatos: IEEE e Anarede. Distinguindo-se pelo formato em que são gerados, necessita-se, obviamente, de formatos de leitura diferentes.

A subrotina *Salva Arquivo* tem como função fazer a interface com o usuário, identificando o nome do arquivo que será gerado como relatório para todo o fluxo de carga e análise de contingência. Este arquivo é salvo na pasta onde está o arquivo executável e tem formato texto (txt). Este arquivo contém todos os dados gerados pela aplicação: fluxo de carga e a norma do vetor tangente entre duas barras.

A subrotina *Fluxo de Carga* é uma chamada feita externamente à principal e que, como discutido anteriormente, é essencial para a análise de contingência, visto que é através do fluxo de carga que saberemos quais condições normais foram infringidas.

Após o cálculo do fluxo de carga, já poderá, então, ser inicializada a análise de contingência com a chamada à subrotina principal da análise. Essa subrotina é composta basicamente de 2 (duas) subrotinas que serão discutidas abaixo. Seria, entretanto, necessário que seja estabelecido primeiramente que:

1. O laço (*loop*) que mantém o sistema em funcionamento encontra-se nesta subrotina principal. Isso ocorre porque cada uma das barras é executada seguindo a ordem do arquivo de dados de entrada (da primeira até a última barra).
2. A primeira chamada é feita à subrotina *matriz*, A segunda chamada é feita à subrotina *análise de contingência* que, efetivamente, faz o cálculo da norma do vetor tangente (figura 6.2).

3. Quando a última barra é executada, o processamento retorna para a rotina *main* que finaliza a gravação dos dados e do programa.

A subrotina *matriz* é responsável pela montagem da matriz jacobiana e deve ser executada a cada mudança de barra.

Chegamos então ao principal objeto de estudo do trabalho: A subrotina de análise de contingência. É nesta rotina que ocorre a análise de contingência estática. A cada barra executada na subrotina principal da análise, novos dados são gerados pela subrotina matriz para, então, ser feita a análise. Cada linha do arquivo de dados contém informações sobre um componente participante do sistema elétrico de potência (SEP). A cada cálculo, faz-se uma simulação da seguinte forma:

1. Retira-se o componente em questão do SEP (é como se ele fosse perdido);
2. O fluxo de carga é refeito fazendo-se o cálculo das variáveis mais importantes;
3. A norma do vetor tangente é, então, calculada.

Estes são os principais passos executados pelo programa serial e neste trabalho, mostraremos a viabilidade, com ganhos, de se paralelizar o cálculo da norma do vetor tangente.

6.3 Uma análise sobre como aplicar técnicas de paralelismo a aplicação serial de análise de contingência

As rotinas que fazem a montagem da matriz possuem maior esforço computacional que a subrotina de análise. Entretanto, o número de iterações da simulação para a análise de contingência pode ser muito grande. Os arquivos de dados disponíveis chegam a 1.916 (mil novecentas e dezesseis) barras (sistema sul sudeste). Portanto, a cada barra removida do SEP na simulação, deve acontecer a análise com as barras restantes. Isso implica em uma grande massa de dados envolvidos no processamento a cada iteração. Apesar deste conhecimento prévio, optamos por paralelizar a rotina que faz o cálculo da norma do vetor tangente, pelo fato de que a rotina foi escrita de uma forma mais desacoplada que as demais rotinas.

Diante dessa análise é que foi então implementada aplicação de análise de contingência estática paralela.

6.4 A aplicação paralela de análise de contingência utilizando PVM

A aplicação paralela da análise de contingência foi implementada sobre a aplicação serial já citada anteriormente. Tendo-se consciência de que um dos fatores que norteiam a eficiência de uma aplicação paralela é, principalmente, o algoritmo paralelo utilizado e não somente o PVM ou o hardware utilizado, foram adotados todos os cuidados necessários para a eficiência do algoritmo paralelo.

O primeiro passo foi a verificação das variáveis dependentes entre as rotinas. Isso se deve à necessidade da utilização de um ambiente de passagem de mensagens, pois cada variável dependente deve ser enviada pelo PVM. Como cada rotina usaria dados diferentes, seria então necessário verificar quais eram as variáveis e se havia a possibilidade de "enxugamento" das mesmas, diminuindo o tamanho das mensagens.

Algumas variáveis foram declaradas com tipos de dados diferentes (como inteiro, real, etc). Os vetores tiveram seu tamanho diminuído onde foi possível.

6.4.1 Empacotamento dos dados

As variáveis que precisavam ser enviadas por mensagens são manuseadas pela subrotina chamada ***empacota.***

```
C       EMPACOTANDO DADOS E ENVIANDO PARA AOS PROCESSOS FILHOS

SUBROUTINE EMPACOTA

INCLUDE 'PARAM.FI'
INCLUDE 'CONTL.FI'
INCLUDE 'ESTAD.FI'
INCLUDE 'JACOB.FI'
INCLUDE 'YBARR.FI'
INCLUDE 'DADOS.FI'
INCLUDE 'FPVM3.H'
INCLUDE 'PVM.FI'
INTEGER INFO, TID, ir, contp
CALL PVMFINITSEND(PVMDEFAULT,INFO)
CALL PVMFPACK(3,LD,1,1,INFO)
CALL PVMFPACK(3,NBUS,1,1,INFO)
...
CALL PVMFSEND(tid(contp+1), 1, INFO)
```

Figura 6.3 (Rotina empacota)

O primeiro passo é limpar uma parte específica da memória para o PVM empacotar as mensagens a serem enviadas. A função *pvmfinitsend*( ) (Libpvm) faz este trabalho.

Utilizando a função *pvmfpack*( ) (Libpvm), as variáveis necessárias ao processamento da tarefa escrava (*slave*) são "empacotadas".

Após empacotar todas as variáveis, a mensagem é transmitida para a tarefa específica, liberando-a para o processamento e consecutivamente enviar a resposta. A transmissão da mensagem é realizada pela função *pvmfsend*( ) (Libpvm).

A rotina *empacota*, na verdade, substituiu a chamada serial da análise de contingência. A cada iteração do arquivo de dados, uma mensagem era enviada para uma tarefa específica. Conforme discutido anteriormente, cada tarefa tem seu TID e é através dele que o PVM consegue saber qual é a tarefa e em qual *host* ela se encontra.

6.4.2 Inicializando os processos escravos (*slaves*)

Os processos escravos (*slaves*), utilizando-se a função *pvmfspawn*( ) (Libpvm), podem ser inicializados de duas formas:

```
do j=1, tamvet
        CALL PVMFSPAWN("CONTI_SLAVE",1,vetorhost(j),1,TID(j),INFO)
        if (info.lt.1) then
                pause
        endif
end do
```

Figura 6.4 (Rotina que dispara os processos slaves)

1. O processo é disparado em qualquer *host* participante da máquina virtual. Neste caso, o PVM seleciona em qual *host* disparar o processo. Isso implica até na possibilidade (não muito remota) da utilização da máquina onde o processo mestre(*master*) é executado.

2. O processo é disparado em uma máquina específica. Através de um parâmetro, é possível utilizar um vetor com os nomes dos *hosts* (vetorhost figura 6.4). Para isso, basta que o vetor contenha somente *hosts* participantes da máquina virtual.

Quando um processo é disparado, o vetor TID recebe o ID do processo. É essa a maneira como o TID está sempre atualizado, evitando que se faça uma passagem de mensagens a uma rotina que já não participa da máquina virtual.

Como o processamento na rotina *master* exige muito do processador, era então necessário evitar que no *host* principal se executasse processos *master e slaves* conjuntamente.

6.4.3 A rotina escrava da análise (*slave*)

A rotina escrava, chamada *conti_slave*, na verdade, é a rotina de análise de contingência serial. Obviamente que com a mesma idéia anterior, a rotina foi otimizada e ficou assim estruturada:

```
1       CALL PVMFMYTID(MYTID)
        CALL PVMFPARENT(MINHA)
        CALL PVMFRECV(-1,-1,INFO)
        CALL DESEMPAC
```

Figura 6.5 (Receive bloqueante da aplicação slave)

1. Uma função que recebe a mensagem *pvmfrecv*( ) (Libpvm). Esta função bloqueia o processo até receber uma mensagem a ela destinada. O processo escravo, quando inicializado, executa todas as linhas de instrução até chegar nesta função, onde a execução pára e aguarda pela mensagem.

```
SUBROUTINE DESEMPAC

INCLUDE 'PARAM.FI'
 INCLUDE 'CONTL.FI'
 INCLUDE 'ESTAD.FI'
 INCLUDE 'JACOB.FI'
 INCLUDE 'YBARR.FI'
 INCLUDE 'DADOS.FI'
INCLUDE 'FPVM3.H'
INTEGER INFO
CALL PVMFUNPACK(3,LD,1,1,INFO)
CALL PVMFUNPACK(3,NBUS,1,1,INFO)
...
```

Figura 6.5 (Rotina desempacota)

2. Uma função que desempacota os dados (figura 6.5) *desempac*( ). Esta função tem como finalidade desempacotar os dados que chegam pelo PVM. Em sua implementação, foi utilizada a função *pvmfunpack*( ) (Libpvm). A função *pvmfunpack* desempacota os dados provenientes das mensagens derivadas do processo *master*. No apêndice A, são dadas mais informações sobre esta e outras funções do PVM.

```
C       ########## ENVIANDO DADOS PARA ROTINA MASTER ##########

CALL PVMFINITSEND(PVMDEFAULT,INFO)
CALL PVMFPACK(3,NF2(0),2,1,INFO)
CALL PVMFPACK(3,NT2(0),2,1,INFO)
CALL PVMFPACK(4,HS(0),2,1,INFO)
CALL PVMFPACK(3,IBUS(0),2,1,INFO)
CALL PVMFSEND(minha,2,INFO)
```

Figura 6.6 (Instruções que enviam resposta para o processo master)

3. Foram também implementadas instruções para empacotar o resultado e transmitir a mensagem com os resultados da análise ao processo *master* (figura 6.6).
4. Após enviar a mensagem com os resultados, o processo *slave* tem a execução das instruções desviada novamente para a função de recebimento *pvmfrecv*( ) e aguarda uma nova mensagem.

```
      DO INTCONT1=1, intsobra
            CALL PVMFRECV(-1,-1,INFO)
            CALL PVMFUNPACK(3,NF2(0),2,1,INFO)
            CALL PVMFUNPACK(3,NT2(0),2,1,INFO)
            CALL PVMFUNPACK(4,HS(0),2,1,INFO)
            CALL PVMFUNPACK(3,IBUS(0),2,1,INFO)
            WRITE(2,296) nf2(ir),nt2(ir),HS(1),IBUS(ir)
296         FORMAT(2X,2(i4,2x),F9.5,1X, I4,/)
            PRINT *, 'LINHA ', IR
      ENDDO
```

Figura 6.7 (Recebendo os resultados da rotina slave na rotina master)

6.4.4 Recebendo os resultados da análise de contingência no processo *master*

O processo *master*, após disparar n tarefas, entra na rotina de recebimento das mensagens (figura 6.7). Esta rotina de instruções recebe as n respostas e desvia a execução da

aplicação para o laço executando as próximas n barras. Isso acontece até o momento em que todas as barras são executadas, finalizando a aplicação.

6.5 Testes e resultados obtidos

Os testes foram feitos em finais de semana onde não havia fluxo de dados na rede.

A configuração, importante para os testes, dos PCs e rede aplicados, é a seguinte:

- Pentium II 350 Mhz;
- 64 Mb de memória RAM;
- Placas de rede padrão Ethernet 10/100 Mbs;
- Hubs da marca 3com de 10/100 Mbs.

O tempo foi cronometrado pela função *timef*( ) (interna do Fortran) assim que a função *pvmfspawn*( ) (função da biblioteca Libpvmf) era chamada, inicializando as tarefas na máquina paralela virtual. Isso quer dizer que inclusive o disparo das tarefas faz parte do tempo total cronometrado, visto que a principal preocupação é com o tempo consumido no *overhead* de inicialização de qualquer chamada externa ao código fonte. A técnica empregada para fazer o disparo da tarefa uma única vez (diminuindo o *overhead*) foi manter as tarefas em *looping*. Normalmente, quando uma tarefa termina o processamento que lhe foi incumbido, ela é então exterminada da máquina em que foi disparada. Como queríamos otimizar o máximo possível o processamento, quando a tarefa atinge o fim dos cálculos, a execução é desviada para a função *pvmfrecv*( ) (Libpvmf), fazendo com que a tarefa esteja novamente preparada para receber uma nova mensagem e fazer um novo cálculo.

1. Cada medição foi feita no mínimo três vezes para cada caso estudado. Em todos os casos, as diferenças entre as mesmas eram infimamente pequenas, podendo ser desprezadas.
2. As medições da aplicação serial foram feitas todas na mesma máquina. Quando do processamento, o processador estava livre de qualquer outra atividade.
3. A aplicação paralela foi utilizada somente em finais de semana e feriados, onde o fluxo de dados pela rede era desprezível.
4. As tomadas de tempo foram feitas sempre na mesma máquina (onde "rodava" o processo *master*) para evitar problemas com variações de *clock*.

Programa Serial	
N.º de barras	Tempo (s)
14	12,53
57	25,27
118	54,76
340	1.530,17
1916	37.632,32

Tabela 6.1

Aplicação Paralela Utilizando PVM			
N.º de barras	N.º de máquinas	N.º de Processos	Tempo (s)
14	2	2	30,46
	5	**5**	**30,37**
	5	10	36,74
	10	10	37,64
57	**5**	**5**	**126,17**
	5	10	128,47
	10	10	127,31
118	**5**	**5**	**170,29**
	5	10	180,50
	10	10	180,97
340	5	5	1.498,65
	5	10	1.471,07
	5	15	1.461,13
	10	15	1.459,02
	10	**10**	**1.449,54**
1916	**5**	**5**	**32.262,88**
	10	10	32.837,21
	5	10	33.324,72

Tabela 6.2

Na tabela 6.2 pode-se encontrar os tempos obtidos para cada execução da aplicação paralela. Os tempos tomados variam de acordo com o número de máquinas que participam do processo e o número de linhas envolvidas no processamento.

Comparação entre a aplicação serial e a paralela					
N.º de barras	Serial (s)	Paralelo (s)	Média Aritmética(s)	Variação (s)	Variação %
14	12,53	30,37	21,45	17,84	242,38
57	25,27	126,17	75,72	100,90	499,29
118	54,76	170,29	112,53	115,53	310,98
340	1.530,17	1.449,54	1.489,86	-80,63	**94,73**
1916	37.632,32	32.808,27	35.220,30	-4.824,05	**87,18**

Tabela 6.3

Na tabela 6.3 e gráficos 6.1, 6.2, 6.3, 6.4, 6.5 e 6.6, pode-se verificar os tempos obtidos nas aplicações serial e paralela. A variação em segundos, negativa, implica em melhor tempo da aplicação paralela.

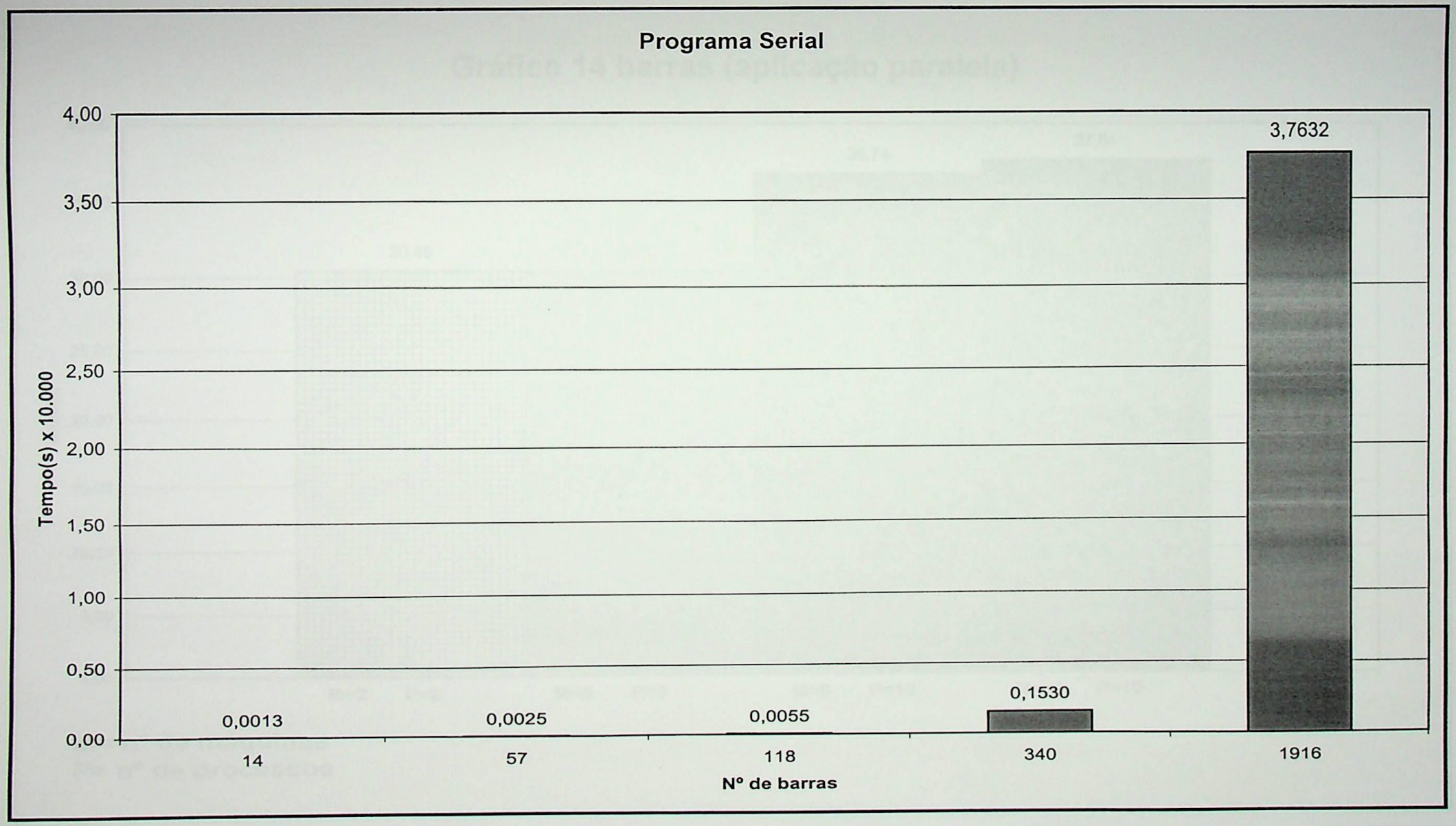

Programa Serial
4,00
3,50
3,00
2,50
2,00
1,50
1,00
0,50
0,00
Tempo(s) x 10.000
0,0013
0,0025
0,0055
0,1530
3,7632
14
57
118
340
1916
Nº de barras

Gráfico 6.1

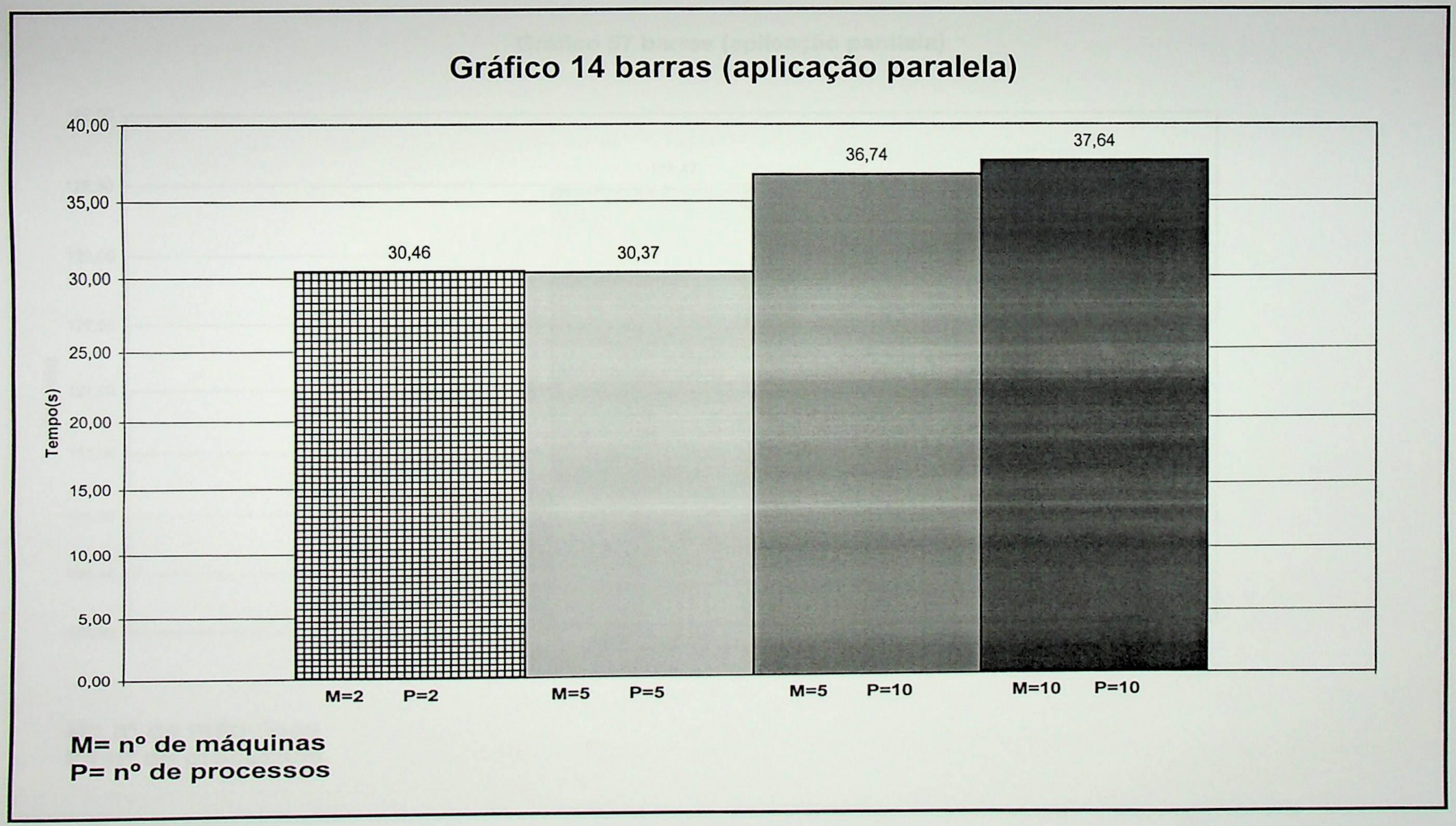


Gráfico 14 barras (aplicação paralela)
40,00
35,00
30,00
25,00
20,00
15,00
10,00
5,00
0,00
Tempo(s)
30,46
30,37
36,74
37,64
M=2 P=2
M=5 P=5
M=5 P=10
M=10 P=10
M= nº de máquinas
P= nº de processos

Gráfico 6.2

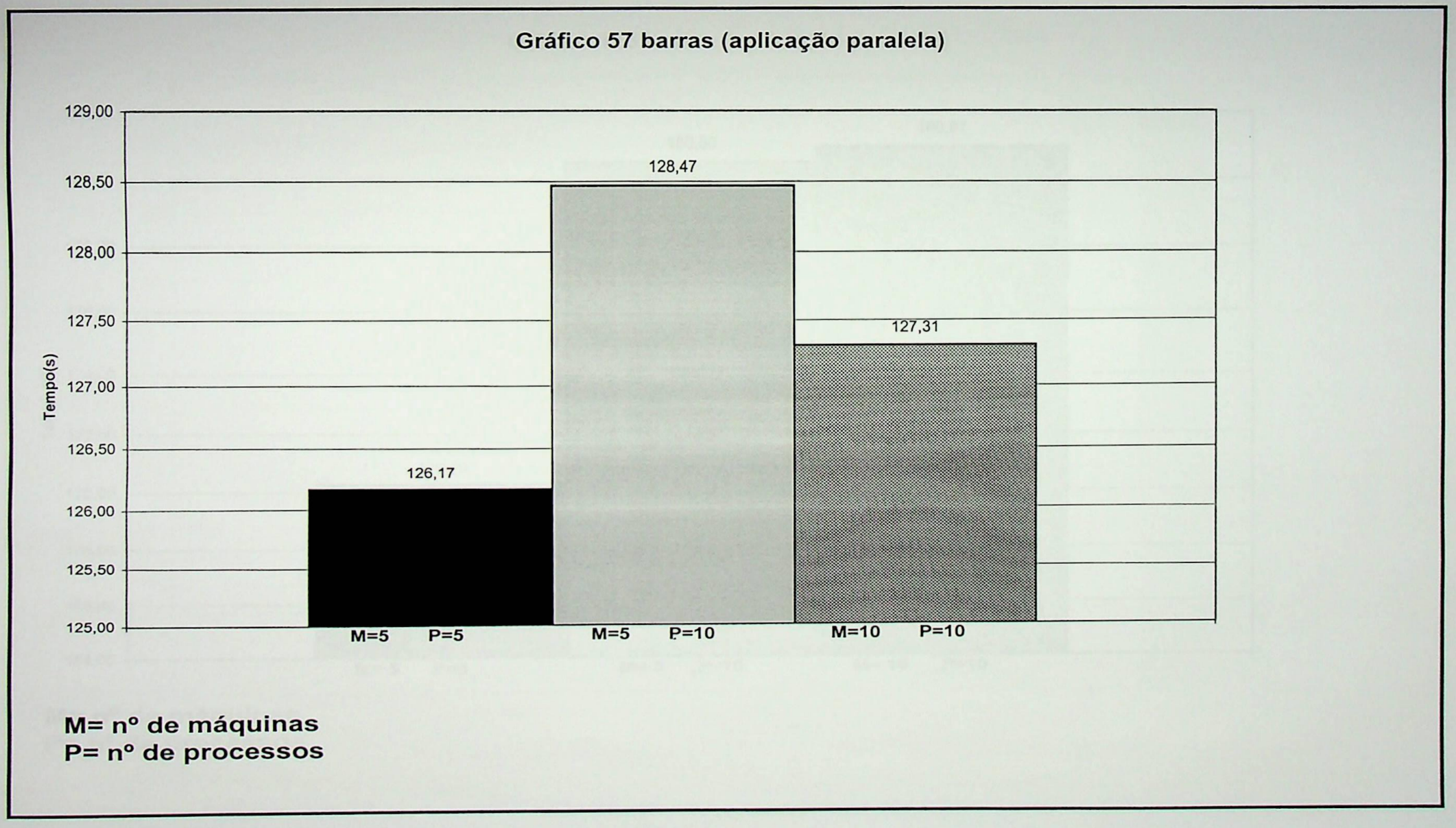


Gráfico 57 barras (aplicação paralela)
129,00
128,50
128,00
127,50
127,00
126,50
126,00
125,50
125,00
Tempo(s)
126,17
128,47
127,31
M=5 P=5
M=5 P=10
M=10 P=10
M= nº de máquinas
P= nº de processos

Gráfico 6.3

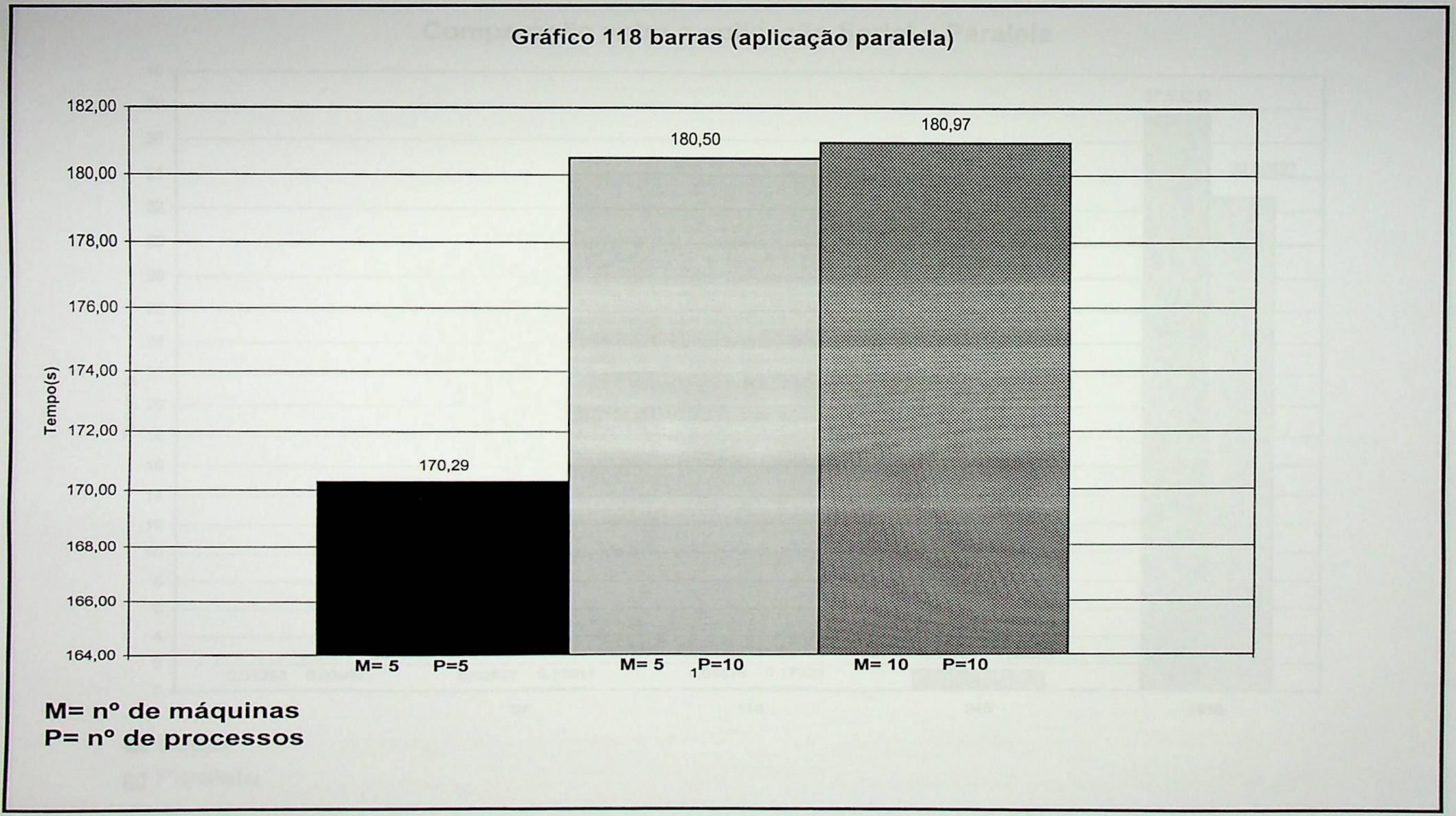


Gráfico 118 barras (aplicação paralela)
182,00
180,00
178,00
176,00
174,00
172,00
170,00
168,00
166,00
164,00
Tempo(s)
170,29
180,50
180,97
M= 5 P=5
M= 5 P=10
M= 10 P=10
M= nº de máquinas
P= nº de processos

Gráfico 6.4

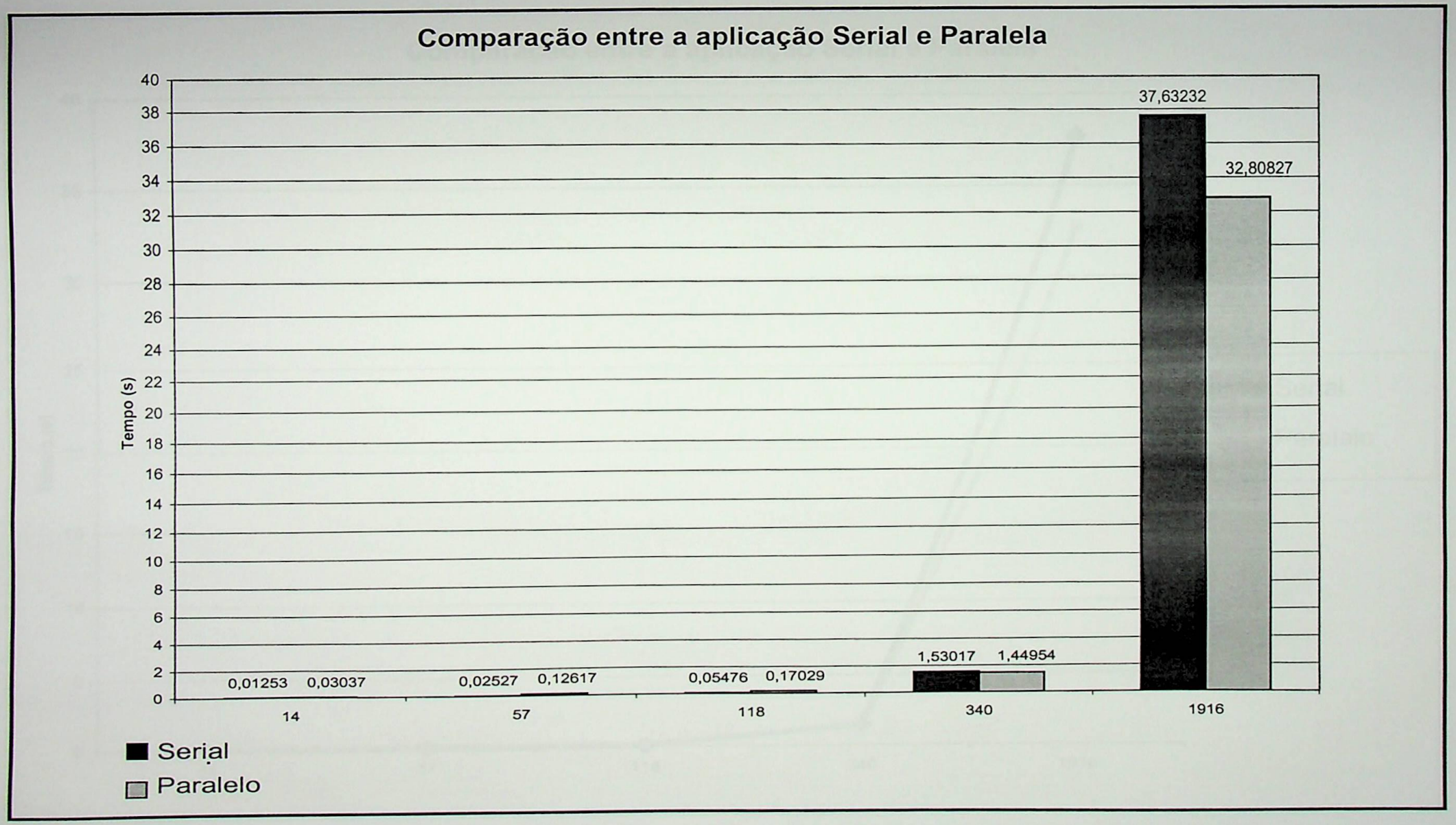

Comparação entre a aplicação Serial e Paralela
Tempo (s)
40
38
36
34
32
30
28
26
24
22
20
18
16
14
12
10
8
6
4
2
0
37,63232
32,80827
1,53017
1,44954
0,01253
0,03037
0,02527
0,12617
0,05476
0,17029
14
57
118
340
1916
Serial
Paralelo

Gráfico 6.7


Comparação entre a aplicação Serial e Paralela
Tempo (s)
40
35
30
25
20
15
10
5
0
14
57
118
340
1916
Serial
Paralelo

Gráfico 6.8

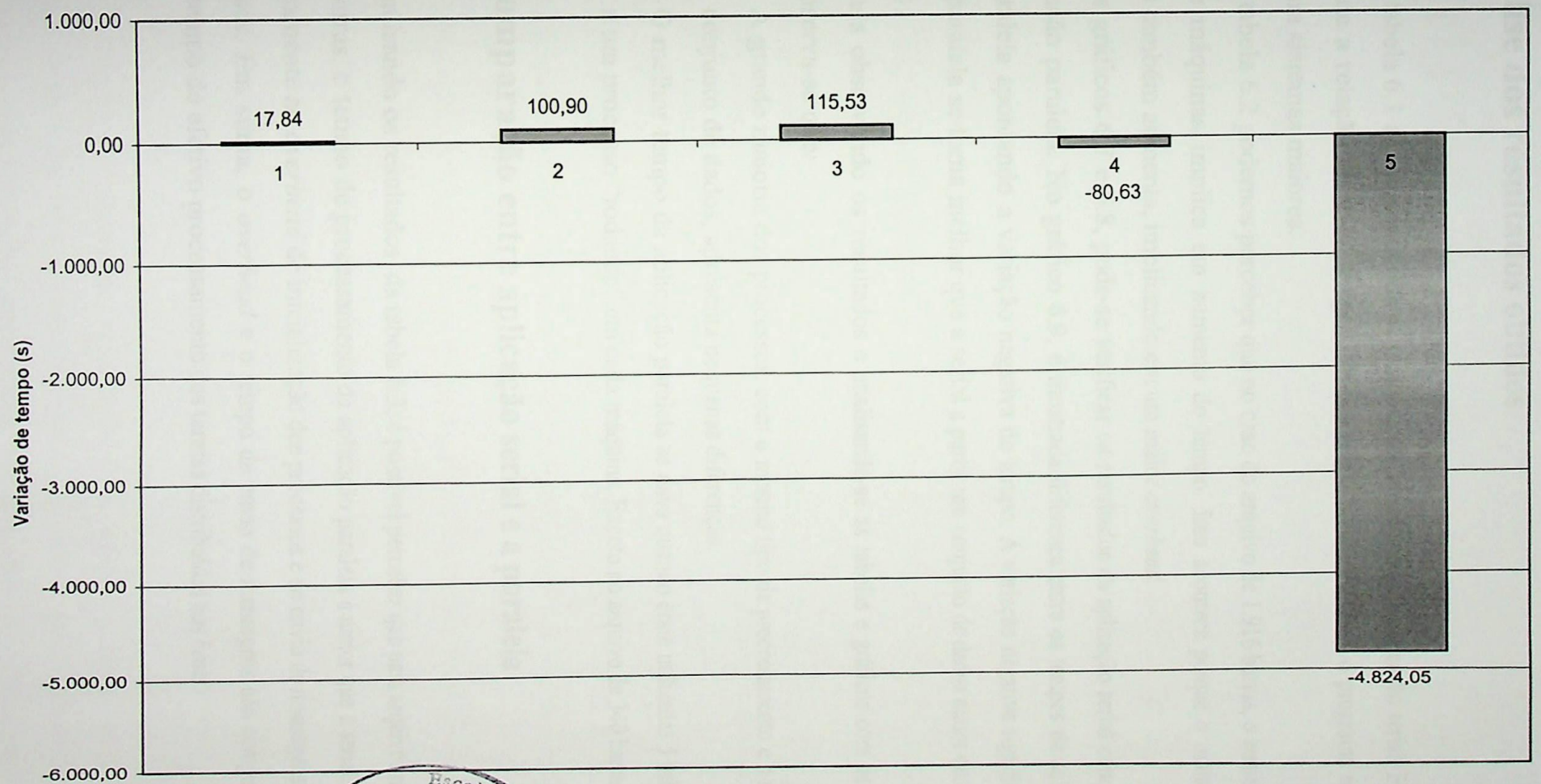


Gráfico 6.9

6.6 Análise dos resultados obtidos

Na tabela 6.1 são encontrados os tempos relacionados com a aplicação serial. Pode-se perceber que a relação entre os tempos obtidos não é linear. Portanto, o programa serial é inviável para sistemas maiores.

Na tabela 6.2 podemos perceber que no caso do arquivo de 1.916 barras, o aumento do número de máquinas implica em aumento do tempo. Isso acontece porque o número de mensagens também aumenta, implicando em um maior *overhead*.

Nos gráficos 6.7 e 6.8, pode-se verificar os resultados da aplicação serial comparada com aplicação paralela. No gráfico 6.9, é mostrada diferença entre os tempos da aplicação serial e paralela apontando a variação negativa do tempo. A variação negativa significa que aplicação paralela se torna melhor que a serial a partir um conjunto de dados maior ou igual a 340 linhas.

Ainda observando os resultados e analisando-se as tabelas e gráficos com os dados obtidos, observa-se que:

- A grande maioria dos processos, com o mesmo tipo de processamento e o mesmo conjunto de dados, apresenta pequenas diferenças.
- O melhor tempo da aplicação paralela se dava quando eram utilizadas 5 máquinas e um processo "rodando" em cada máquina. Exceto no arquivo de 340 barras.

6.6.1 Comparação entre aplicação serial e a paralela

Analisando os resultados da tabela 6.3, é possível perceber que para arquivos menores que 340 barras, o tempo de processamento da aplicação paralela é maior que a serial. Isso se deve basicamente ao *overhead* de inicialização dos processos e do envio de mensagens entre os processos. Em suma, o *overhead* e o tempo de envio de mensagens não compensam o ganho de tempo de efetivo processamento das tarefas distribuídas nos *hosts*.

Capítulo 7

Conclusões

Neste capítulo são apresentadas as conclusões deste trabalho. As contribuições principais e as dificuldades encontradas são discutidas, apresentando-se posteriormente sugestões para trabalhos futuros, que servirão como diretrizes para a continuidade do trabalho aqui iniciado.

7.1 Considerações iniciais

Neste trabalho, foi apresentada uma proposta de paralelismo aplicada à análise de contingência estática. A análise citada exige grande esforço computacional e, utilizando-se de técnicas de paralelismo, poder-se-ia reduzir o tempo de processamento.

Discutindo os principais aspectos das duas áreas onde este trabalho está inserido, sistemas distribuídos e computação paralela, foram apresentados os motivos iniciais para o surgimento dessas áreas até o momento onde surgiu uma mescla dessas áreas, a computação paralela distribuída, objeto de nossa aplicação.

A utilização de um ambiente de passagem de mensagens, PVM, foi discutido e longamente comentado no anexo A, enfatizando-se detalhes transparentes ao usuário e dando dicas de utilização das funções da biblioteca Libpvm.

Uma introdução ao sistema elétrico de potência e a análise de contingência também foram discutidas de forma introdutória, visto que o assunto é amplo e esse trabalho tem como finalidade apenas descrever como o paralelismo pode ser empregado no cálculo da análise de contingência.

Por último, a comparação entre a aplicação serial e a paralela na análise de contingência estática foi feita, demostrando que a utilização de técnicas de paralelismo no caso citado pode obter bons resultados quanto à redução do tempo de processamento.

7.2 Contribuições deste trabalho

Este trabalho apresenta várias contribuições. Entre as tantas contribuições, pode-se citar:

- Introduziu uma nova forma de resolução para o problema de análise de contingência;

- Permitiu que a computação paralela distribuída seja utilizada na análise de contingência estática com redução no tempo de processamento;

- Documentou a utilização do ambiente de passagem de mensagens PVM para Windows®. No anexo A são feitos comentários e dicas de utilização do PVM assim como são dados exemplos de utilização das funções básicas;

- Foi criada uma página na internet no endereço www.pvmwin.cjb.net onde pessoas interessadas pelo PVM podem tirar suas dúvidas ou ainda baixar arquivos com exemplos de utilização do PVM;

- Permitiu que usuários sem muitos conhecimentos de computação paralela distribuída possam utilizar o PVMWIN como ferramenta de estudos.

7.3 Dificuldades encontradas

7.3.1 Sobre o PVM

Dentre as muitas dificuldades encontradas e relacionadas abaixo, a utilização do PVM foi sem dúvida a maior. Nas poucas referências encontradas, quase em sua totalidade, as informações eram para o PVM de plataforma UNIX. Quando o trabalho foi iniciado, foram então notórias as diferenças entre o PVM UNIX e o PVM para Windows®. A ausência do RSH (*Remote Shell)* trouxe muitos horas perdidas de trabalho. Da instalação do PVM que traz alguns inconvenientes (todos citados no anexo A) à instalação do RSH, faltou material

destinado ao PVMWIN. Desses problemas nasceu idéia de ajudar a outros futuros usuários com a página na internet.

7.3.2 Em relação ao programa serial

- A aplicação serial não foi construída com o propósito de ser paralelizada. Variáveis declaradas como públicas, chamadas a procedimentos externos sem necessidade, dentre outros;

- As variáveis necessárias à passagem de mensagens eram muitas e de grande porte. Foi, então, necessário ajustar variáveis quanto a declaração de tipo e dimensão, no sentido de reduzir o tamanho das mensagens;

- A linguagem utilizada (Fortran) não é a ideal para a utilização com o PVM que, apesar de oferecer suporte para isso, possui limitações impostas pela conversação da função original em C para Fortran.

7.4 Trabalhos futuros

Para a continuidade deste trabalho, são feitas algumas sugestões:

- Escrever uma aplicação para a análise de contingência, utilizando-se de técnicas de paralelismo desde o princípio, ou seja, criar uma aplicação desde a sua base com o objetivo de que ela seja uma aplicação paralela;
- Implementar o paralelismo na matriz Jacobiana.

7.5 Considerações finais

Analisando-se os objetivos e a motivação inicialmente propostos para este trabalho, verifica-se que técnicas de paralelismo aplicadas à análise de contingência estática pode e deve ser usada, visto que os resultados deste trabalho demonstram que a utilização de paralelismo, para este caso, obtém redução do tempo de processamento em arquivos com

maior número de barras. Feita a análise dos resultados, foi possível perceber que arquivos com mais de 340 barras têm ganhos de velocidade se comparados com uma aplicação serial.

Com os melhoramentos propostos para o futuro, pode-se chegar a resultados bem melhores, aumentando o desempenho da aplicação paralela e consequentemente sendo viável a sua utilização para arquivos menores que 340 barras.

Referências Bibliográficas

[ALM94] ALMASI, G. S., Gottieb A., *High Parallel Computing,* 2ª ed., The Benjamin Cummings Publishing Company, Inc., 1994.

[AND83] ANDREWS, G. R., Schineider, F. B., "Concepts and Notations for Concurrent Programming", ACM *Computing Survey,* v. 15, nº 1, pp. 3-43, 1983.

[AST82] ASTORGA, O. A. M., "Análise de Fluxo de Carga e Contingências em Sistemas Elétricos de Potência: Uma Metodologia Eficiente", Dissertação de Mestrado, Escola Federal de Engenharia de Itajubá, Agosto, 1982.

[BAR95] BARQUIN, J., Gomes, T., "Estimating the Loading Limit Margin Taking Into Account Voltage Collapse Areas", IEEE/PES winter Meeting-Paper 95 WM 183-4 PWRS, New York, January 20 – February 2, 1995.

[BEG94] BEGUELIN, ª, et al., PVM: *Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, 1994.

[CAN93] CANIZARES, C. A., Alvarado, F. L., "Point of Collapse and Continuation Methods for Large ac/dc Systems", IEEE Transactions on Power Systems, v. 8, nº 1, pp. 1-8, February, 1993.

[CAN95] CANIZARES, C. A., et. al., "Comparision of Performance Indices for Detection of Proximity to Voltage Collapse", IEEE/PES Summer Meeting-Paper 95 SM 583-5 PWRS, Portland, July 23-27, 1995.

[COU88] COULOURIS, G., Dollimore, J., *Distributed Systems Concepts and Design*, Addison-Wesley Publishing Company, 1988.

[COU94] COULOURIS, G., et. al., *Distributed Systems Concepts and Design*, 2ª ed., Addison-Wesley Publishing Company, 1994.

[DUN90] DUNCAN, R., "A Survey of Parallel Computer Architectures", IEEE Computer, pp. 5-16, Fevereiro, 1990.

[ELG76] ELGERD, O. I., Introdução à Teoria de Sistemas de Energia Elétrica, São Paulo, McGraw-Hill, 1976.

[FLY72] FLYNN, M. J., "Some Computer Organizations and their Effectiveness", IEEE Transactions on Computers, v. C-21, pp. 948-960, 1972.

[GAO92] GAO, B., et. al., "Voltage Stability Evaluation Using Modal Analysis", IEEE Transactions on Power Systems, v. 7, nº 7, pp. 1529-1542, 1992.

[GEI94] GEIST, A., et. al., *PVM 3 User's Guide and Reference Manual*, Oak National Laboratory, Setembro, 1994.

[GEI95] GEIST, A., "PVM HOME PAGE, Recent PVM Highlights", página html, http://www.epm.ornl.gov/pvm/highlight.html.

[HWA84] HWANG, K., Briggs, F. A., *Computer Architecture and Parallel Processing*, McGraw-Hill International Editions, 1984.

[JAR94] JARDIM, J. L. A., "Advances in Power System Transient Stability Assessment Using Transient Energy Functions Methods", PhD Thesis, University of London, 1994.

[KIN95] KING, A., *Desvendando o Windows95*, Rio de Janeiro, Editora Campus, 1995.

[KIR91] KIRNER, C., "Arquitetura de Sistema Avançados de Computação", Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho, pp. 307-353, 1991.

[LOF92] LOF, A., et al., "Fast Calculation of Voltage Stability Index", IEEE Transactions on Power Systems, v. 7, nº 1, pp. 61-68, 1995.

[MAN95] MANSOUR, Y., et. al, "B. C. Hydro's on-Line Transient Stability Assessment (TSA) Model Developmente, Analysis and Post-Processing", IEEE Transactions on Power Systems, v.10, pp. 241-53, February 1995.

[MAR94] MARANNINO, P., et. al., "Voltage Collapse Proximity Indicators for Very Short Term Security Assesment", Proc. Bulk Power System Voltage Phenomena III – Voltage Stability and Security, ECC Inc., Davos, Switzerland, 1994.

[MCB94] MCBR, O. A., "Na Overview of Message Passing Environments", *Parallel Computing*, v. 20, pp. 417-444, 1994.

[MUL93] MULLENDER, Distributed Systems, 2ª ed., ACM Press Frontier Series, Addison-Wesley Publishing Company, 1993.

[NAV89] NAVAUX, P. O. A., "Introdução ao Processamento Paralelo". RBC-Revista Brasileira de Computação, v. 5, nº 2, pp. 31-43, outubro, 1989.

[PRA91] PRADA, R. B., et. al., "Voltage Stability: Phenomena Characterization Based on Reactive Control Effects and System Critical Areas Indentification", Proceedings of the Third SEPOPE meeting, Belo Horizonte, SP-14, 1991.

[RAM83a] RAMOS, D. S., Dias, E. M., Sistemas Elétricos de potência, vol. 1, Rio de Janeiro, 1983.

[RAM83b] RAMOS, D. S., Dias, E. M., Sistemas Elétricos de potência, vol. 2, Rio de Janeiro, 1983.

[SAN89a] SANTANA, R. H. C., "Performance Evaluation of Lan-Based File-Servers", PhD. Dissertation, University of Southampton, Outubro, 1989.

[SAN89b] SANTANA, M. J., "Na Advanced Filestore Architecture for a Multiple Lan Distributed Computing System", PhD. Dissertation, University of Southampton, outubro, 1989.

[SOU96] SOUZA, A. C. Z., et. al., "Critical Bus and Point of Collapse Determination Using Tangent Vectors", 28th North American Power Symposium, Cambridge, USA, , pp. 329-333, November 10-12, 1996.

[SOU97] SOUZA, A. C. Z., et. al., "New Techniques to Speed Up Voltage Collapse Computations Using Tangent Vectors", IEEE Transactions on Power Systems", v.12, nº 3, pp. 1380-1387, August 1997.

[SOU98] SOUZA, A. C. Z., "Tangent Vectors as a Tool for Voltage Collapse Analysis", VI SEPORE, Salvador, 24-29 maio 1998.

[STE90] STEVENS, W. R., *UNIX Networks Programming*, Prentice Hall International Inc., 1990.

[SUN94] SUNDERAM, V. S., et. al., "The PVM Concurrent Computing System: Evolution, Experiences and Trends", *Parallel Computing*, v. 20, pp. 531-545, 1994.

[TAN90] TANENBAUM, A. S., et. al., "Experience with the Amoeba Distributed Operating System", *Comunications of the ACM*, v. 33, nº 12, Dezembro, 1990.

[TAN92] TANENBAUM, A. S., *Modern Operating Systems*, 2ª ed., Prentice Hall International Inc., 1992.

[TAN95] TANENBAUM, A. S., Van Renesse, R., "Distributed Operating Systems", *ACM Computing Surveys*, v. 17, nº 4, Dezembro, 1995.

[TAN97] TANENBAUM, A. S., Redes de Computadores, 2ª ed., Rio de Janeiro, Editora Campus, 1997.

[WAL94] WALKER, D. W., "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers", *Parallel Computing*, v. 20, pp. 657-673, 1994.

[ZAL91] ZALUSKA, E. J., "Research Lines in Distributed Computing System and Concurrent Computation", Anais do Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software, pp. 132-155, 1991.

Bibliografia Complementar

[ALV94] ALVES, L., Comunicação de Dados, São Paulo, Makron Books, 1994.

[ARA95] ARAQUE, V. M. S., "Paralelização de Ajuste de Histórico de Produção em Rede de Estações Usando PVM", Dissertação de Mestrado, Universidade Estadual de Campinas, Dezembro, 1995.

[BAC90] BACH, M. J., *The Design of the UNIX Operating System*, Prentice Hall International Inc., 1990.

[CUE97] CUERVO, C. H. V., "Otimização do Cálculo de Parâmetros no Processo de Ajuste de Históricos de Produção usando PVM", Dissertação de Mestrado, Universidade Estadual de Campinas, março, 1997.

[DEC79] DECKMANN, S. M., "Equivalentes Estáticos para Sistemas de Energia Elétrica", Tese de Doutoramento, Universidade Estadual de Campinas, Dezembro, 1979.

[HEH87] HEHL, M. E., Linguagem Estruturada Fortran 77, São Paulo, McGraw-Hill, 1987.

[MON80] MONTICELLI, A. J., "Análise Estática de Contingências em Sistemas de Energia Elétrica", Universidade Estadual de Campinas, 1980.

[ROC82] ROCHA, M. C., "Redespacho Econômico Sob Contingência", Dissertação de Mestrado, Escola Federal de Engenharia de Itajubá, Setembro, 1982.

[SEB99] SEBESTA, R. W., Conceitos de Linguagens de Programação, 4ª edição, Porto Alegre, Editora Artes Médicas Sul Ltda, 2000.

[SIL91] SILVEIRA, J. L., Comunicação de Dados e Sistemas de teleprocessamento, São Paulo, McGraw-Hill, 1991.

[SIL98] SILBERSCHATZ, A., Galvin, P. B.,*Operating System Concepts*, Addison Wesley Longman, Inc., 1998.

[SOA95] SOARES, L. F. G., et. al., Redes de Computadores: das LANs, MANs e WANs às Redes ATM, Rio de Janeiro, Editora Campus, 1995.

[TED79] TEDESCHI, S. G. G., "Redespacho sob Contingência", Dissertação de Mestrado, Escola Federal de Engenharia de Itajubá, Dezembro, 1979.

Anexo A

PVM

Parallel Virtual Machine

Ao instalarmos o PVM versão 3.4 para Windows® (PVMWIN), tivemos vários problemas relacionados desde a instalação até o procedimento final que é o recebimento de mensagens com os devidos resultados das tarefas paralelas. Na busca de respostas para os nossos problemas, deparamo-nos com a falta de material especifico do PVMWIN.

Sem dúvida que, para outras plataformas, podemos considerar vasto o material de pesquisa, entretanto, ainda assim faltam maiores detalhes sobre vários pontos e que aqui neste anexo tentaremos ressaltar.

O PVM é composto basicamente de uma biblioteca que efetua a comunicação entre os processos paralelos. Basta para isso que o protocolo de comunicação seja IP.

Os detalhes que relataremos a partir de agora foram obtidos na versão citada acima e, portanto, pode ser que haja alguma diferença entre versões diferentes.

1. Instalação

1.1- PVM

O pacote PVMWIN pode ser baixado livremente no site http://www.epm.ornl.gov/pvm e a versão 3.4 tem aproximadamente 6 Mb.

O arquivo baixado instala e configura parcialmente o PVM. São necessários para a instalação os softwares no caso de se usar a linguagem C : Visual C++ 5.0 ou 6.0. No caso de utilizar-se da linguagem Fortran : Digital Fortran 5.0 ou 6.0. Esses softwares são utilizados para compilar as rotinas existentes e não compiladas para exemplos e rotinas usadas pelo próprio PVMWIN.

Abaixo são citadas algumas ocorrências que impedem o perfeito funcionamento do PVM.

- Na tela onde se pede o local (pasta) em que deve ser instalado o PVM, o default é C:*Arquivo de Programas\PVM3.4*. Como o nome da pasta indicada possui mais de 8 bytes, no modo MSDOS o nome da pasta seria então abreviado. Isso cria uma incompatibilidade entre o registro do Windows e a pasta onde se localiza o PVM. Sugestão: instale o PVMWIN em uma pasta cujo nome seja simples (você verá mais adiante que isso ajuda) e não ultrapasse 8 bytes. Exemplo: PVM. O mesmo pode ocorrer com a pasta temporária do PVMWIN.
- Caso o PVMWIN seja finalizado incorretamente, dentro da pasta temporária ficarão alguns arquivos que impedem a nova inicialização. Você deverá, sempre que o PVMWIN for finalizado incorretamente, apagar estes arquivos temporários que se encontram na pasta selecionada no momento da instalação. Exemplo: *C:\temp*

1.2- RSH

Para que exista a comunicação entre os processos, ainda é necessário o *Remote Shell* (RSH) em http://www.ataman.com/product_download.htm. O RSH é um software *shareware* que pode ser utilizado por até 15 dias. Após este prazo, sua licença espira e você deve então adquiri-lo.

Para instalá-lo basta digitar *arshd install start*. O próximo passo é inicializar o aplicativo regedit do Windows. Execute os seguintes passos:

- Selecione a chave HKEY_LOCAL_MACHINE
- Selecione SOFTWARE
- Selecione PVM
- Adicione Valor da seqüência *PVM_RSH*
- Modifique o valor para *nome da pasta onde instalou o RSH.* Exemplo: *C:\RSH*
- Vá ao painel de controle e inicialize o aplicativo Ataman RSHD Service (ele foi criado na instalação do RSH).
- Adicione um usuário (figura 1)
- Configure o *user name*
- Configure o *Host access List*. Este parâmetro contém a lista de endereços IP para o acesso do RSH. Exemplo: se você quer acessar vários endereços IP, uma maneira de fazer é usar um * , representando várias máquinas *192.168.0.** . O asterisco empregado permite a você acessar todas as máquinas que têm a combinação de IP acima citado.

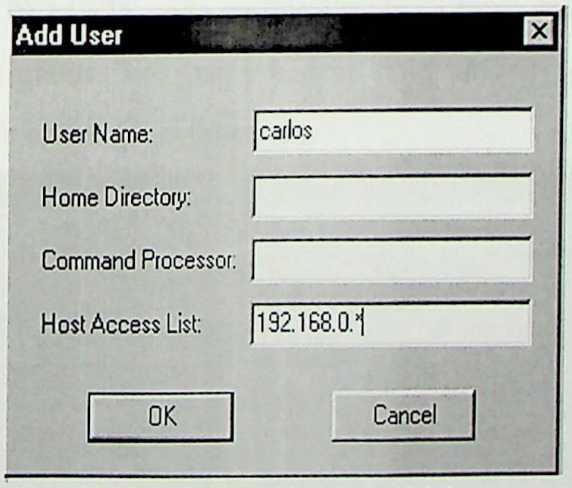


Figura 1 (adicionar usuário)

- Reinicialize a máquina

- Na inicialização da máquina, será aberta uma janela, onde você poderá ver o carregamento em memória do ARSHD. Basta, portanto, que você inicialize a máquina para que o RSH esteja residente em memória.
- Após instalar, em no mínimo duas máquinas, você pode testar se o RSH ficou OK. Digite o seguinte comando no *path* onde se localiza o RSH. *Rsh nomedamáquina –l username comando.* Exemplo : rsh violeta –l carlos dir . Este comando executará o comando dir na máquina chamada violeta. Faça o mesmo na outra máquina.
- Se você seguir todos os passos como proposto, dificilmente haverá problemas.

Problemas relacionados:

- Verifique se existe algum antivírus ou similar nas máquinas onde você vai instalar o RSH (obviamente em todas máquinas que farão parte da sua máquina virtual). Esses softwares citados podem impedir o perfeito funcionamento do RSH. Às vezes a máquina responde e não consegue executar um comando ou executa e não consegue responder. Na dúvida, desinstale estes softwares das máquinas. Um exemplo já comprovado foi a utilização do Norton antivírus. Ele efetivamente impede a execução de quaisquer comandos remotamente.
- Se em determinado momento não houver resposta entre as máquinas, finalize todos os aplicativos e reinicialize a máquina. É a única solução.

2- Console do PVMWIN

O console do PVMWIN pode ser inicializado no ícone criado em sua instalação. Esse console lhe permite efetuar comandos básicos do PVMWIN como: adicionar e deletar máquinas, monitorar e executar um programa.

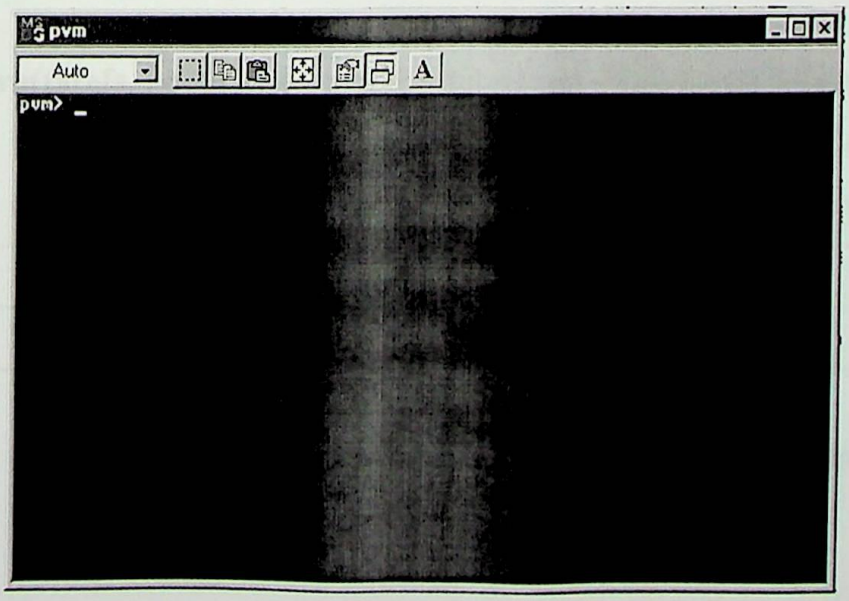


Figura 2 (console do PVMWIN)

Quando o console é executado, automaticamente você coloca a máquina local em disponibilidade para a máquina virtual. A máquina local será a máquina onde será disparada a tarefa master. Isso não quer dizer que não possa também executar tarefas *slaves*. Portanto a tarefa slave deve também ser atualizada localmente.

Algumas funções principais a serem utilizadas no console:

add – O comando add adiciona novas máquinas e tem o seguinte formato: *add "nomedamáquina dx=pathdoPVM\lib\win32\pvmd3.exe"* . Exemplo : add "violeta dx=c:\pvm\lib\win32\pvmd3.exe". Agora é possível compreender o quanto o caminho onde instalar o PVM deve ser simples. Isso evita que você tenha muito trabalho na hora de adicionar máquinas. Esse comando dispara o *daemon* da máquina (nomedamáquina) relacionada.

conf- O comando *conf* lhe permite verificar quais as máquinas estão ativas na configuração. Exemplo : *conf* . Serão listadas todas as máquinas ativas.

delete – Elimina a máquina escolhida da configuração. Exemplo: *delete violeta*

reset- Mata todas as tarefas ativas do PVM

halt – Finaliza o console e todas as aplicações em execução.

3- Funções principais do PVMWIN para Fortran

pvmfmytid(tid) – Essa função identifica o processo atual

pvmfspawn(tarefa, opção, onde, ntarefas, tid, nexec) – Essa função dispara os n processos.

- **tarefa** – variável caracter contendo o nome do arquivo slave executável
- **opção** – variável inteira. Como deverão ser disparados os processos.

0	**Iniciar em qualquer máquina**
1	**Será escolhida uma máquina**

- **onde -** variável caracter que indica onde deverá ser inicializado processo.
- **Ntarefas** – quantos processos devem ser inicializados.

- **Tid** – vetor inteiro de dimensão ntarefas que contém o id de cada processo inicializado.
- **Nexec** – variável inteira de retorno que indica o número de tarefas inicializadas. (deve ser igual a ntarefas)

Essa função, se executada uma vez, disparando o número de processos necessários de uma única vez, tem maior compromisso com a otimização do processo. Isso se deve ao *overhead* criado para cada execução da função.

pvmfinitsend(regra, info) – Essa função prepara a memória para empacotar e codificar uma nova mensagem.

- **regra** – variável inteira que especifica a regra de codificação da mensagem. O default é 0.
- **Info** – variável de retorno que identifica o sucesso da operação. Se info > 0=sucesso.

Pvmfpack(tipo, nomevar, nitem, posicao, info) – Essa função empacota os dados relativos a um determinado tipo de variável.

- **Tipo** – O tipo de dado a ser empacotado.

STRING	0	**REAL4**	4
BYTE1	1	**COMPLEX8**	5
INTEGER2	2	**REAL8**	6
INTEGER4	3	**COMPLEX16**	7

- **Nomevar** – nome da variável a ser empacotada. Se for um vetor e, se for o caso, ler apenas uma parte do vetor é então necessário identificar a posição inicial. Exemplo: vetor(5), o PVMWIN lerá o vetor a partir da posição 5.
- **Nitem** – esta variável identifica no caso de um vetor quantas posições serão lidas. Exemplo 1 : nomevar=vetor(5) e nitem=1 lê somente a posição 5 da variável vetor. Exemplo 2: nomevar=vetor e nitem=10, lê a variável vetor da primeira posição até a posição 10.
- **Posicao** – determina como deverão ser posicionados os dados a serem empacotados. Se a variável for um número complexo, *posicao=2,* do contrário *posicao=1.*
- **Info** – variável de retorno que identifica o sucesso da operação. Se info > 0=sucesso.

Pvmfsend(tid, rotulo, info) – Essa função envia a mensagem para o outro processo.

- **Tid** – variável inteira que identifica o processo ao qual deve ser enviada a mensagem. O conteúdo dessa variável dever ser igual ao vetor tid criado pela função pvmfspawn. Para cada pvmfsend executado deve existir um único processo *slave* aguardando. Isso quer dizer que o vetor tid será percorrido a cada pvmfsend executado.
- **Rótulo** – essa variável poderá rotular sua mensagem. Cada mensagem enviada pode, por exemplo, ter um rótulo específico
- **Info** – variável de retorno que identifica o sucesso da operação. Se info > 0=sucesso

Essa função somente pode ser executada tantas vezes quantas tarefas forem inicializadas em pvmfspawn.

pvmfparent(tid)- Essa função retorna o número da identificação do processo pai (master).

- **tid-** variável inteira contendo a identificação. Se tid < 0 existe um erro.

Pvmfrecv(tid, rotulo, bufid)- Essa função bloqueia o processo até receber uma mensagem do processo master.

- **Tid-** variável inteira que identifica de qual o processo a função deverá aguardar a mensagem. Se tid=-1, a função receberá uma mensagem de qualquer processo.
- **Rotulo-** rótulo que a função deverá utilizar para identificar a mensagem correta. Se rotulo=-1, a função receberá a mensagem com qualquer rótulo.
- **Bufid-** variável inteira de retorno que identifica o buffer de recebimento.

< 0	**Erro**
-2	**Argumento inválido**
-14	**Pvmd não responde (daemon não respondeu)**

Pvmfunpack(tipo, nomevar, nitem, posicao, info) – Essa função desempacota os dados relativos a um determinado tipo de variável. A essa função aplicam-se todos os parâmetros encontrados na função pvmfpack.

Uma atenção especial deve ser dada a essa função. A ordem em que as variáveis são empacotadas deve ser a mesma de desempacotamento. Não haverá nenhum erro em relação à

variável de retorno info, mas o conteúdo das variáveis pode ficar nulo ou até com valores incorretos.

Pvmfexit(info)- Essa função finaliza o processo da tarefa relacionada. Ao final da tarefa é sempre necessário usar essa função para que esse processo seja finalizado.

4 – Compilação do programa paralelo

O programa paralelo pode ser compilado no local do projeto onde ele se encontra. Porém é preciso ficar atento a alguns detalhes.

1. Copie o arquivo fpvm3.h e pvm3.h para a pasta *include* de seu compilador. Exemplo : no caso do Fortran da Microsoft seria : *c:\Arquivos de Programas\msdev\include*
2. Acrescente ao seu projeto as seguintes bibliotecas :
   PVMWIN- *libfpvm3.lib, libpvm3.lib*
   C++ *- wsock32.lib, gdi32.lib, winspool.lib, comdlg32.lib, advapi32.lib, shell32.lib, ole32.lib, oleaut32.lib, uuid.lib, oldnames.lib*
3. Essas bibliotecas são necessárias para a linkedição das rotinas do PVMWIN . Na verdade, o que acontece é uma conversão da rotina de Fortran para C. Daí a explicação do porquê de se utilizar as bibliotecas do C++. Essas bibliotecas deverão ficar também na mesma pasta do compilador utilizado, porém na pasta especifica de bibliotecas. Exemplo : *c:\Arquivos de Programas\msdev\lib*

Diante de todos os detalhes que pudemos ver, tivemos a iniciativa de criar uma página na internet que visa cooperar com a comunidade cientifica, no sentido de ampliar as fontes de pesquisa sobre o PVM que hoje já pode ser considerado um padrão em processamento paralelo distribuído.