UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE SISTEMAS DE ENERGIA ELÉTRICA

SOLUÇÃO CONCORRENTE DO PROBLEMA DO FLUXO DE POTÊNCIA ÓTIMO COM RESTRIÇÕES DE SEGURANÇA

Autor: Osvaldo Saavedra Méndez

Orientador: Alcir José Monticelli

Tese apresentada à Universidade Estadual de Campinas, como parte dos requisitos para a obtenção do título de DOUTOR EM ENGENHARIA ELÉTRICA.

Campinas, Junho de 1993

Este exemplar corresponde à redação final da tese defendida por Osvaldo S. Mendes e aprovada pela Comissão Julgadora em 30/06/93.
Alcir Monticelli
Orientador

UNICAMP
BIBLIOTECA CENTRAL

Nunca dês um nome a um rio:
Sempre é outro rio a passar.
Nada jamais continua,
Tudo vai recomeçar! ...

Mário Quintana

A minha família.

Agradecimentos

- Ao professor Alcir José Monticelli pela orientação;

- Ao professor Ariovaldo Verândio Garcia pela ajuda dispensada;

- Aos colegas do DSEE pelas discussões e apoio;

- A Fundação de Amparo à Pesquisa do estado de São Paulo - *FAPESP*, que financiou a aquisição do computador *NCUBE* no Projeto Temático de Equipe: Nova Geração de Centros de Controle de Sistemas de Energia Elétrica - No 90/3940-0.

- Ao CPqD-TELEBRÁS que facilitou o computador *PP*.

Este Trabalho contou com o apoio financeiro da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - *CAPES* e da Fundação de Amparo à Pesquisa do estado de São Paulo- *FAPESP*.

Resumo

Este trabalho apresenta um algoritmo concorrente para a solução do fluxo de potência ótimo com restrições de segurança em computadores paralelos. O método assíncrono sugerido é baseado em construções e estruturas típicas da programação concorrente, como é o caso do modelo produtor-consumidor utilizado no gerenciamento da troca de mensagens e na partilha de dados entre as tarefas e subtarefas nas quais o problema original é decomposto. Este modelo permite um gerenciamento simples e eficiente da execução das tarefas assíncronas, o que normalmente resulta em bom balanceamento de carga entre os processadores. Este estilo de programação acomoda uma ampla variedade de problemas relacionados com o fluxo de potência ótimo com restrições de segurança, ao mesmo tempo que facilita a implementação em ambientes físicos distintos (arquiteturas baseadas em memória compartilhada e distribuída). O novo método é implementado com sucesso em uma máquina paralela, tipo memória compartilhada e barramento comum, e em um computador de memória distribuída com arquitetura hipercúbica, de 64 processadores.

Conteúdo

1 Introdução **1**

2 Formulação do Problema **5**

2.1 Centros de Controle 5

2.2 Fluxo de Potência Ótimo 7

2.2.1 Variáveis de controle 8

2.2.2 Restrições operacionais 9

2.2.3 Funções objetivo 10

2.2.4 Modelos de Otimização 11

2.3 Fluxo de Potência Ótimo com Restrições de Segurança 14

2.3.1 Introdução 14

2.3.2 Algoritmo de Resolução 15

3 Análise de Segurança **18**

3.1 Análise de Contingências 18

3.1.1 Definição de Contingências 19

3.1.2 Seleção de Contingências 20

3.1.3 Avaliação de Contingências 20

3.2 Índices de Severidade para Potência Ativa 21

3.2.1 Índice Baseado no Nível de Carregamento de Potência Ativa 21

3.2.2 Gradiente do Índice de Desempenho 21

3.2.3 Índice Baseado na Sobrecarga 22

4 Processamento Paralelo **23**

4.1 Introdução 23

4.2 Computadores Paralelos 24

4.3 Classificação de Flynn para Arquiteturas de Computadores 27

4.4 Computadores Multiprocessadores 29

4.4.1 Arquiteturas de Memória Compartilhada 30

4.4.2 Arquiteturas de Memória Distribuída 32

4.5 Desempenho de Algoritmos Paralelos 34

4.5.1 Medidas de Qualidade de Algoritmos Paralelos 37

4.5.2 Portabilidade 38

5 Solução Concorrente do Fluxo de Potência Ótimo com Restrições de Segurança **39**

5.1 Decomposição do Problema 39

5.1.1 Granularidade e "Overheads" 40

5.1.2 O que será paralelizado 40

5.2 Modelos de Programação 43

5.2.1 Modelo de Programação Concorrente 43

5.2.2 Critério de Parada 46

5.2.3 Mapeando o Problema na Máquina Real 48

5.3 Máquina de Memória Compartilhada 48

5.3.1 Computador Utilizado. 48

5.3.2 Ferramentas de Programação Concorrente. 49

5.3.3 Implementação no Ambiente Físico 52

5.4 Máquina de Memória Distribuída 54

5.4.1 O Computador de Memória Distribuída. 54

5.4.2 Obtenção da Contingência a ser Analisada. 57

5.4.3 Envio de Restrições. 58

5.4.4 Estado do Sistema 59

5.4.5 Critério de Parada 60

5.4.6 Comentários 61

6 Resultados de Testes **62**

6.1 Introdução 62

6.2 Resultados Obtidos no Computador *PP* 65

6.3 Resultados Obtidos no Computador *NCUBE* 72

6.4 Recapitulação 77

7 Conclusões **80**

A Breve Revisão do Algoritmo de Programação Linear Utilizado. **86**

B Restrições de Ramos **91**

C Restrições de Contingências **92**

D Programas Fontes **94**

D.1 Programas Mestre e Escravo para o Computador de Memória Compartilhada (*PP*) 95

D.2 Programas Mestre e Escravo para o Computador de Memória Distribuída (*NCUBE*) 101

Capítulo 1

Introdução

Apesar do problema de fluxo de carga ótimo ter sido originalmente formulado há três décadas por Carpentier, somente agora começam a estar disponíveis algoritmos e equipamentos que permitem a sua utilização prática tanto no planejamento como na operação de sistemas de energia elétrica com grandes dimensões. No que se refere aos equipamentos, os grandes avanços tecnológicos no "hardware" computacional decorrentes do rápido desenvolvimento de processadores baseados em VLSI têm levado ao crescimento exponencial da capacidade de processamento e a uma queda relativa dos custos . Por outro lado houve também um grande aperfeiçoamento nos métodos de programação matemática (incluindo-se aí os métodos de decomposição), nos algoritmos de resolução e na própria modelagem dos sistemas elétricos; este último fator permitiu a formulação de problemas mais próximos da realidade das aplicações. Os centros de controle já estão se beneficiando desses desenvolvimentos, ganhando em velocidade de resposta e capacidade de manipular grandes volumes de dados [1].

Algumas das funções avançadas dos centros de controle tais como as relacionadas com a determinação de ações preventivas da mesma forma corretivas têm formulação matemática de problemas de otimização de grande porte, com dimensões e número de variáveis elevados. Sua resolução em máquinas uniprocessadoras normalmente exigem simplificações na modelagem, principalmente no que se refere às restrições de segurança (contingências). O advento de máquinas multiprocessadoras em conjunção com técnicas de decomposição matemática, passou a oferecer uma alternativa atraente para a implementação do fluxo de carga ótimo com restrições de segurança em centros de controle.

A disponibilidade de máquinas de processamento paralelo possibilita a aplicação destas novas tecnologias de computação para resolver problemas de sistemas de potência. As aplicações em tempo real precisam de respostas rápidas a problemas que envolvem muitas variáveis e restrições. Este objetivo ainda é um grande desafio, porém, a introdução de

processamento paralelo para a solução desses problemas oferece boas expectativas.

Utilização de Computação Paralela em Aplicações de Sistemas de Potência

No trabalho [22] é apresentada uma revisão do estado da arte da aplicação de computação paralela para a resolução de problemas de sistemas de potência. Estas aplicações são divididas em três grupos.

O primeiro grupo considera as aplicações que paralelizam a resolução de equações algébricas que têm a forma $Ax = b$, onde A é uma matriz esparsa. Este processo é muito requisitado na maioria dos problemas de sistemas de potência; o método seqüencial mais eficiente até hoje, é a utilização de fatoração triangular com substituições "forward-backward". As propostas paralelas existentes na bibliografia utilizam várias arquiteturas computacionais. Dentro deste grupo também são incluídas as soluções dos métodos de Newton, onde um conjunto de equações não lineares é resolvido por métodos iterativos. Algoritmos tipo Gauss-Seidel e suas variantes são citados na bibliografia como fontes de exploração de paralelismo; o principal obstáculo encontrado é o alto número de iterações para convergência.

O segundo grupo de aplicações envolve a resolução de equações algébricas e diferenciais. Dentro deste grupo está o problema de estabilidade transitória, o qual é definido por um conjunto de equações algebro-diferenciais não lineares. Várias propostas têm sido formuladas, utilizando-se máquinas baseadas em troca de mensagens e de memória compartilhada.

O terceiro grupo contempla aplicações variadas, tais como a determinação de autovalores e autovetores, cálculo de confiabilidade, e por fim, fluxo de potência ótimo com despacho corretivo. Neste âmbito, as primeiras iniciativas de aplicações paralelas foram apresentadas por Teixeira et alli [8] e Pinto et alli [9]. No primeiro trabalho são apresentados os resultados das implementações concorrentes do programa de avaliação de confiabilidade -multi-área, avaliação de custos de produção e confiabilidade e finalmente, fluxo de potência ótimo com restrições de segurança e redespacho corretivo. As implementações foram feitas em uma máquina de memória compartilhada. O segundo trabalho é uma extensão do primeiro, e apresenta algumas estratégias visando melhorias no desempenho para o algoritmo de fluxo de potência ótimo com restrições de segurança e redespacho corretivo.

Proposta deste Trabalho

A velocidade do desenvolvimento do "hardware" não tem sido acompanhada pelo "software" e aplicações, levando a ter soluções fortemente vinculadas ao tipo de arquitetura disponível, dificultando a migração para outras. Existe, também, uma tendência a paralelizar soluções seriais que, apesar de serem baratas do ponto de vista de desenvol-

vimento, nem sempre levam à melhor versão paralela. Precisa-se, então, de modelos de programação ou paradigmas que independam das particularidades das arquiteturas, que permitam que a aplicação "viaje" por distintos ambientes. Por outro lado, as primeiras aplicações paralelas de fluxo de potência ótimo têm sido fortemente influenciadas pelos algoritmos seqüenciais já existentes. As iniciativas têm sido paralelizar esse algoritmos em determinadas máquinas, visando reduzir custos de desenvolvimento[8].

A proposta deste trabalho é apresentar um algoritmo concorrente assíncrono para a resolução do cálculo de *fluxo de potência ótimo com restrições de segurança.* O desenvolvimento do algoritmo é realizado com o auxílio de modelos de programação, com o objetivo de tornar mais fácil a programação, de tal forma que a migração de um ambiente para outro seja facilitada. O método assíncrono sugerido é baseado em construções e estruturas típicas da programação concorrente, como é o caso do modelo produtor-consumidor utilizado no gerenciamento da troca de mensagens e na partilha de dados entre as tarefas e subtarefas nas quais o problema original é decomposto. Este modelo permite um gerenciamento simples e eficiente da execução das tarefas assíncronas, o que normalmente resulta em bom balanceamento de carga entre os processadores. O novo método foi implementado com sucesso em uma máquina paralela, tipo memória compartilhada e barramento comum e portado para um computador de memória distribuída com arquitetura hipercubo, de 64 processadores.

Organização do Trabalho

Este trabalho está organizado da seguinte maneira:

- No capítulo 2, é apresentado o problema de fluxo de potência ótimo em termos gerais e como está inserido dentro das funções de um centro de controle de energia elétrica. Mais para o fim do capítulo, o enfoque é particularizado para o problema de *fluxo de potência ótimo com restrições de segurança* e a metodologia escolhida para sua resolução.
- No capítulo 3, é feita um rápida revisão dos conceitos de análise de segurança e de alguns critérios de ordenação de contingências.
- No capítulo 4 são brevemente revisadas as arquiteturas de computadores paralelos e alguns conceitos de programação concorrente.
- No capítulo 5 é apresentada a proposta de solução concorrente para o problema de *fluxo de potência ótimo com restrições de segurança.*
- No capítulo 6 são apresentados e analisados os resultados das implementações do algoritmo proposto em um computador de memória compartilhada e em um computador de memória distribuída *NCUBE.*

- Por fim, no capítulo 7, são apresentadas as conclusões mais relevantes deste trabalho e propostas para trabalhos futuros.

Capítulo 2

Formulação do Problema

Neste capítulo tenta-se inicialmente localizar o problema a ser resolvido, dentro do contexto dos centros de controle de sistemas de energia elétrica. Aborda-se, em termos gerais, o problema de fluxo de potência ótimo, para logo em seguida particularizar em torno do problema específico a ser resolvido, a formulação adotada e a metodologia de resolução escolhida.

2.1 Centros de Controle

Os centros de controle de sistemas de energia elétrica são um complexo conjunto de "software" e "hardware" responsáveis pela monitoração "on-line", controle e otimização do sistema, com o fim de prevenir ou corrigir problemas operacionais mantendo a economia da operação do sistema.

Para definir as funções de um centro de controle, é necessária uma classificação formal dos níveis de segurança do sistema. A figura (2.1) apresenta uma classificação proposta por Stott, Alsaç e Monticelli em [2]. As setas indicam transições involuntárias entre os níveis 1 a 5 devido a perturbações, sejam elas decorrentes de contingências, alterações na geração ou na carga, ou devido a ações de controle.

Nível 1: Todas as cargas são atendidas e não há violações de limites operacionais. A ocorrência de uma contingência não causa violações.

Nível 2: Todas as cargas são atendidas e não há violações de limites operacionais. As violações decorrentes de uma contingência podem ser eliminadas com uma

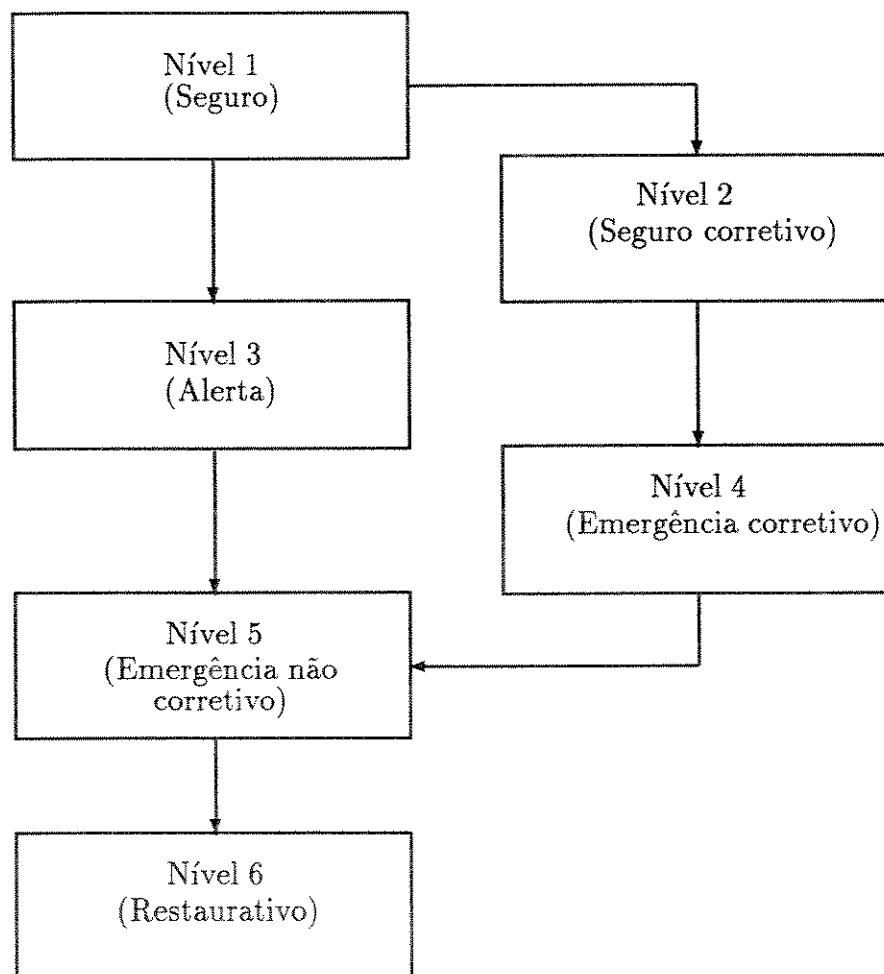

Figura 2.1: Níveis de segurança estática de um sistema de potência

apropriada ação de controle sem perda de carga.

Nível 3: Todas as cargas são atendidas e não há violações de limites operacionais. Porém, as violações decorrentes de uma contingência não podem ser eliminadas, sem perda de carga.

Nível 4: Todas as cargas são atendidas, porém há violações de limites operacionais. Essas violações podem ser corrigidas com uma apropriada ação de controle, sem perda de carga.

Nível 5: Todas as cargas são atendidas, porém há violações de limites operacionais. Essas violações não podem ser corrigidas sem perda de carga.

Nível 6: Não há violações de limites operacionais, porém há cargas que não estão sendo atendidas.

Os níveis 1 e 2 na figura (2.1) representam o estado de operação *normal* do sistema, no sentido que eles são estados operacionais aceitáveis. O nível 1 corresponde ao ideal em termos de segurança. O nível 2 representa uma alternativa mais econômica , baseada na remoção de violações não severas num intervalo especificado de tempo, sem a perda de carga.

A eliminação de violações no nível 4, envolve a realização de *despacho corretivo* ou *ações corretivas*, que levam o sistema ao nível 3. Quando o nível 3 é atingido, deve-se realizar um *despacho preventivo* que leve o sistema aos níveis 1 ou 2, dependendo dos objetivos de segurança. Por outro lado, se o sistema atinge o nível 5, haverá desligamentos de cargas, seja por ações automáticas locais, ou por ordem do centro de controle. Em alguns casos, a quantidade e a localização das ações de controle podem ser calculadas otimamente.

2.2 Fluxo de Potência Ótimo

O controle ótimo-seguro de um sistema de geração e transmissão de energia elétrica é uma tarefa extremamente complexa. A complexidade cresce com o tamanho do sistema, interconexão, tipos de carga, tipos de controle, etc.

O Fluxo de Potência Ótimo (FPO) é um nome genérico para uma série de funções de controle ótimo, que determinam de maneira *ótima* as estratégias de controle, levando em conta restrições de fluxos de potência na rede e limites operacionais do sistema.

O propósito de uma função FPO "on-line" é determinar a estratégia de controle que permita a operação em um nível desejado de segurança (normalmente 1,2 ou 3), enquanto é otimizada uma função objetivo, por exemplo custo de operação. Os níveis 1 e 2 levam em conta restrições de contingências.

Uma função "on-line" de FPO específica é projetada para:

- Operar em modo tempo real ou modo estudo.
- Determinar ações de controle de potência ativa, potência reativa ou ambas.
- Atingir um nível de segurança definido.
- Minimizar um objetivo operacional especificado.

O fluxo de potência ótimo é formulado matematicamente como um problema de otimização geral restrito:

$$Min \quad f(x,y) \tag{2.1}$$
$$g(x,y) = 0 \tag{2.2}$$
$$h(x,y) \leq 0 \tag{2.3}$$
$$x_{min} \leq x \leq x_{max} \tag{2.4}$$

onde x é o conjunto de variáveis controláveis no sistema, y é o conjunto de variáveis dependentes. A função objetivo (2.1) é escalar. As expressões (2.2) representam as equações de fluxo de potência, ocasionalmente aumentadas com algumas poucas restrições especiais de igualdade. As expressões (2.3) incluem limites operacionais e restrições de segurança. As expressões (2.4) representam os limites nas variáveis de controle.

2.2.1 Variáveis de controle

As variáveis de controle mais comuns podem ser classificadas de acordo com seu efeito primário, seja este sobre a potência ativa, reativa ou ambas.

As variáveis de controle primário para a potência ativa são:

- Geração de potência ativa.
- Transformadores defasadores.
- Intercâmbio de potência ativa entre áreas.
- Transferência em "link" de corrente contínua.
- Opções de compra e venda em "spot market".

Os principais controles de potência reativa são:

- Tensões nos geradores ou potências reativas.
- Reatores e capacitores shunt.
- Tap de transformadores em fase.

Controles de potência ativa e reativa:

- Transformadores com razões complexas variáveis
- Entrada e saída de operação de unidades geradoras.
- Redução ou corte de carga.
- Chaveamento de linhas.

2.2.2 Restrições operacionais

Além das restrições de balanço para fluxos, existem várias restrições de desigualdade tais como limites mínimo e máximo de geração, de transferência e de tensão. Algumas dessas restrições são resumidas a seguir:

Restrições em potência ativa:

- Limites de fluxos ativos em ramos.
- Reserva girante de potência ativa.
- Intercâmbio de potência ativa entre áreas.
- Limite de transferência de potência ativa de grupo de linhas.
- Abertura de ângulos de tensões de barras.

Restrições em potência reativa:

- Tensões de barras.
- Fluxos reativos em ramos.
- Reserva girante de potência reativa.
- Intercâmbio de potência reativa entre áreas.
- Limite de transferência de potência reativa de grupo de linhas.

Restrições em potência ativa e reativa:

- Fluxos de correntes em ramos e fluxos de MVA.
- Fluxos de MVA em grupos de linhas.

2.2.3 Funções objetivo

Vários objetivos são possíveis de se formular e são direcionados para diferentes aplicações. Alguns são mais adequados para planejamento, enquanto que outros são mais interessantes para tempo real. Algumas das funções objetivo mais comuns são descritas a seguir.

Mínimo custo de operação

Este objetivo considera a soma dos custos dos geradores controláveis, mais os custos de todas as transações de intercâmbio controlável. Todas as variáveis de controle do sistema são elegíveis para participar na minimização deste objetivo. Se apenas as potências ativas são controladas, o processo de cálculo é chamado de *Despacho de Potência Ativa com Restrições de Segurança.* As variáveis de controle sem custo direto, tais como tensões e tap de transformadores, são otimizadas visando minimizar as perdas ativas.

Um fator crítico na minimização de custo é a modelagem das curvas de custos dos geradores. A maioria dos métodos de otimização precisam de curvas de custo convexas; embora as curvas reais não satisfaçam isto, elas são aproximadas por uma curva convexa. Existem várias maneiras possíveis de descrever a curva de custo. Alguns exemplos são:

- Ajustamento de curvas.
- Aproximação exponencial.
- Funções quadráticas.
- Segmentação linear.

A seleção depende do algoritmo utilizado.

Mínima perda de transmissão de potência ativa

Os controles que podem ser direcionados para este objetivo são todos aqueles sem custo direto, isto é, todos exceto as gerações e intercâmbio de MW. Porém há controles tais como defasadores e fluxos em linhas de corrente contínua que não são utilizados para a minimização de perdas, pois eles são mais úteis para o controle de potência ativa. Logo, a

minimização de perdas é normalmente associada com o controle tensão/potência reativa e taps de transformadores.
Este é um objetivo de "ajuste fino" para a operação do sistema, que tende a diminuir a circulação de reativos e melhora o perfil de tensão. Em alguns sistemas, pode ser obtida uma considerável economia de geração de potência ativa.

Mínimo desvio de um ponto especificado

Este é um objetivo freqüentemente usado. Ele é usualmente definido como a soma dos quadrados ponderados dos desvios das variáveis de controle com relação a um ponto de referência dado, que pode ser o ponto inicial ou outro especificado. Este objetivo é adequado para despacho corretivo.

Mínimo número de ações de controle

Este objetivo é importante quando não é possível ou é indesejável reprogramar um número elevado de controles ao mesmo tempo. Ele é aplicável quando não estão disponíveis os meios automáticos em um centro de controle para manipular simultaneamente vários controles. Este objetivo é também adequado para o alívio em tempo real de violações de limites operacionais, visando manter o sistema próximo de um ponto de operação. Ao contrário do objetivo anterior (mínimo desvio), este não pode ser formulado ou resolvido rigorosamente em forma algébrica. A solução será tipicamente sub-ótima, porém é prática em controle em tempo real.

2.2.4 Modelos de Otimização

As diferenças entre os métodos são devidos não apenas ao processo de otimização, mas também ao modelo dos problemas [14].

Modelo Esparso

A formulação geral do problema pode ser escrita como segue:

$$Min \quad f(x,y) \tag{2.5}$$
$$g(x,y) = 0 \tag{2.6}$$
$$h(x,y) \leq 0 \tag{2.7}$$
$$x_{min} \leq x \leq x_{max} \tag{2.8}$$

que é chamada de *formulação esparsa*. As variáveis são divididas entre variáveis de controle (x) e variáveis dependentes (y). As restrições (2.6) e (2.7) são muito esparsas. A vantagem desta formulação é que o algoritmo de solução pode ser aplicado diretamente

à formulação do problema. A desvantagem é que o número de variáveis é elevado para sistemas de grande porte.

Modelo Compacto

Por várias razões, especialmente porque as não linearidades na resolução das equações de fluxo de potência ($g(x, y) = 0$) são suaves e porque as técnicas disponíveis para resolvê-las são muito eficientes, torna-se convidativa a formulação do problema apenas em termos das variáveis de controle:

$$Min \quad f(x, y[x]) \tag{2.9}$$
$$g(x, y[x]) = 0 \tag{2.10}$$
$$h(x, y[x]) \leq 0 \tag{2.11}$$
$$x_{min} \leq x \leq x_{max} \tag{2.12}$$

onde agora o processo de resolução deve ser combinado com um bloco de cálculo de fluxo de potência para a determinação das variáveis dependentes y.

Este modelo é chamado de *compacto* ou *não esparso* [5]. Também, as relações (2.9 - 2.12) são chamadas de modelo reduzido ou problema reduzido. A utilização da formulação não esparsa requer a aplicação de técnicas de relaxação. A idéia que está por trás das técnicas de relaxação na teoria de otimização é que normalmente poucas das restrições ficarão ativas na solução ótima. Uma eficiente técnica para identificação e geração dessas restrições pode acelerar o processo de resolução. Além da velocidade, as formulações não esparsas apresentam várias vantagens, entre elas:

- Manipulam restrições de segurança com facilidade.

- Fornecem um modelo reduzido válido para uma ampla região.

Uma típica seqüência de passos para resolver um problema de otimização com formulação não esparsa é apresentada na figura (2.2). Estas idéias serão retomadas mais adiante.

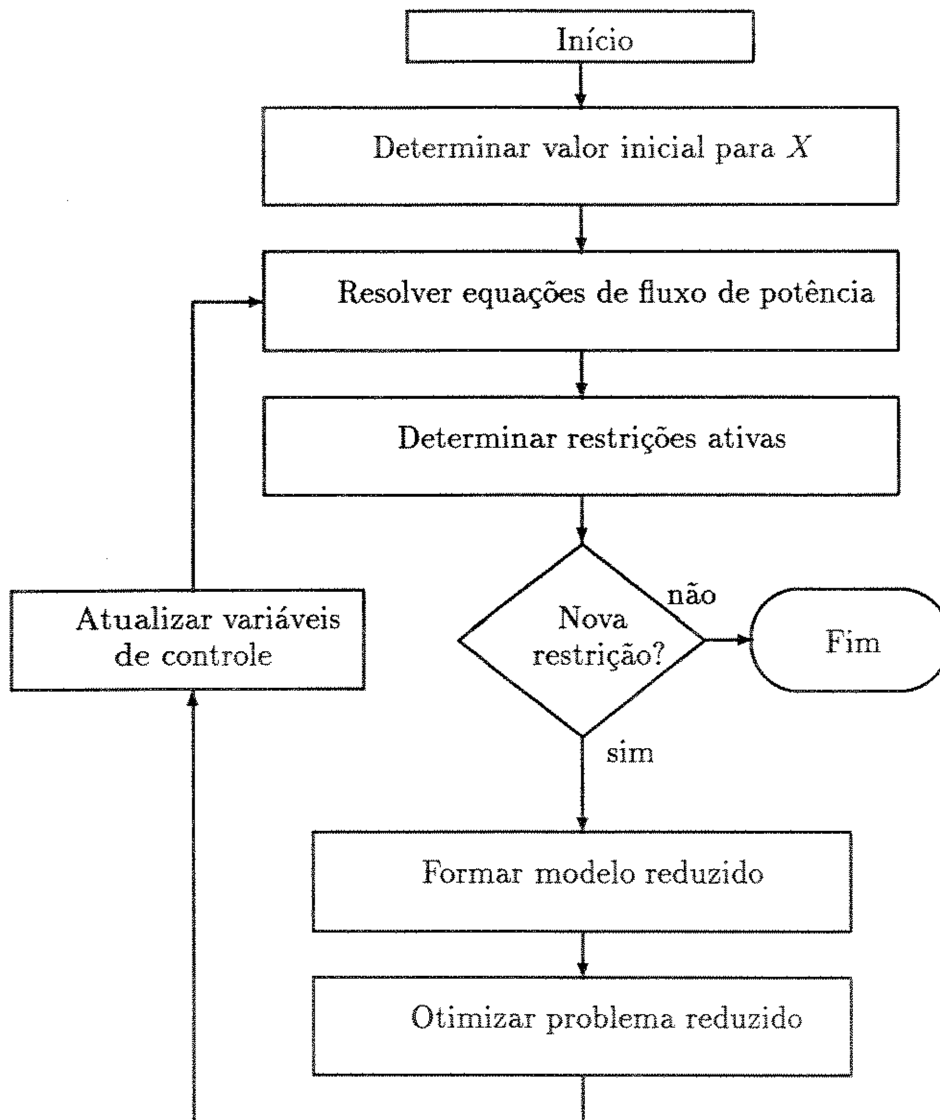

Figura 2.2: Seqüência típica de resolução de FPO com formulação compacta

2.3 Fluxo de Potência Ótimo com Restrições de Segurança

2.3.1 Introdução

O fluxo de potência ótimo com restrições de segurança (FPORS) objetiva determinar uma solução viável de mínimo custo (ou um outro objetivo como, por exemplo, mínimo desvio em relação a uma solução de mínimo custo), de tal forma que quando da ocorrência de uma contingência pertencente a uma lista especificada, o estado pós-contingência permanecerá viável. O problema FPORS está situado no nível 1 da figura (2.1). Este problema foi originalmente formulado em [3], sendo que extensões que tratam de remanejamento corretivo pós-contingência foram sugeridas em [7] e analisadas em [2].

As restrições de contingência são intrínsecas aos níveis 1 e 2 da figura (2.1). O número total de contingências a ser considerado no cálculo do FPORS é elevado. Dependendo do tamanho do sistema, cada caso de contingência pode envolver centenas ou milhares de inequações. Como uma típica lista de contingências considera um elevado número de casos, o número total de contingências pode atingir vários milhões. Felizmente, poucas dessas restrições estarão ativas na solução, o que torna convidativa a utilização de técnicas de relaxação.

Neste trabalho apresenta-se a solução concorrente do problema de fluxo de potência ótimo com restrições de segurança, com ênfase em potência ativa, utilizando programação linear dual baseada no método proposto por Stott e Marinho [4].

As técnicas de programação linear (PL) têm sido pesquisadas e desenvolvidas por três décadas, sendo que trabalhos pioneiros surgiram no final dos anos 50 nos E.E.U.U., e no início do anos 60, na Grã Bretanha.
As técnicas de PL têm várias vantagens com relação aos métodos de programação não linear [5], entre elas destacam-se:

- Flexibilidade para aplicações paralelas.

- Confiabilidade do processo de resolução PL.

- Facilidade de detecção de inviabilidades, propriedade de grande importância no processo de decomposição.

- O processo de resolução pode ser muito rápido utilizando um código orientado à aplicação.

O método de resolução considera o modelo compacto ou não esparso e combina programação linear dual com técnicas de relaxação, conjunção especialmente adequada quando

o objetivo a otimizar está relacionado com potência ativa. Logo, a formulação compacta (2.9-2.12) pode ser reescrita em uma forma linearizada do problema (modelo linearizado incremental) em termos das variáveis de controle:

$$Min \qquad f = c^t x \tag{2.13}$$

$$Ax \leq b \tag{2.14}$$

$$A_i x \leq b_i \qquad i = 1, \ldots.., nc \tag{2.15}$$

$$x_{min} \leq x \leq x_{max} \tag{2.16}$$

onde as variáveis de controle x representam as gerações de potência ativa controláveis; c é o vetor custo associado; a expressão (2.14) representa as restrições operacionais para o caso-base e (2.15) representa o mesmo tipo de restrições que as dadas por (2.14), para cada uma das nc configurações que resultam de cada contingência da lista. As restrições de igualdade (2.10) da formulação compacta correspondem aqui a um bloco de cálculo de fluxo de potência c.c. (modelo de corrente contínua), invocado pelo processo de otimização para a determinação das variáveis dependentes, neste caso, o estado do sistema (ângulos das tensões de barra). O algoritmo seqüencial do processo de resolução é dado na figura (2.3).

A seguir, é apresentada uma visão global do algoritmo de resolução do problema de FPORS. Uma revisão sumária do método de programação linear é apresentada no apêndice A e maiores detalhes do mesmo são encontrados em [4] e [5].

2.3.2 Algoritmo de Resolução

O algoritmo concorrente estudado neste trabalho é uma *extensão* do algoritmo de relaxação originalmente proposto em [4] para a solução seqüencial do fluxo de carga ótimo com restrições de segurança. Este algoritmo combina programação linear dual e técnicas de relaxação. Dois níveis de relaxação são identificados no algoritmo:

Relaxação de nível inferior:

Neste nível, o problema a ser resolvido é:

$$\begin{aligned} Min \qquad f &= c^t x \\ Ax &\leq b \\ x_{min} \leq x &\leq x_{max} \end{aligned} \tag{2.17}$$

que corresponde à formulação de um problema de FPO , *sem* levar em conta restrições de segurança (algoritmo original de Stott, indicado com linha segmentada na figura (2.3)).

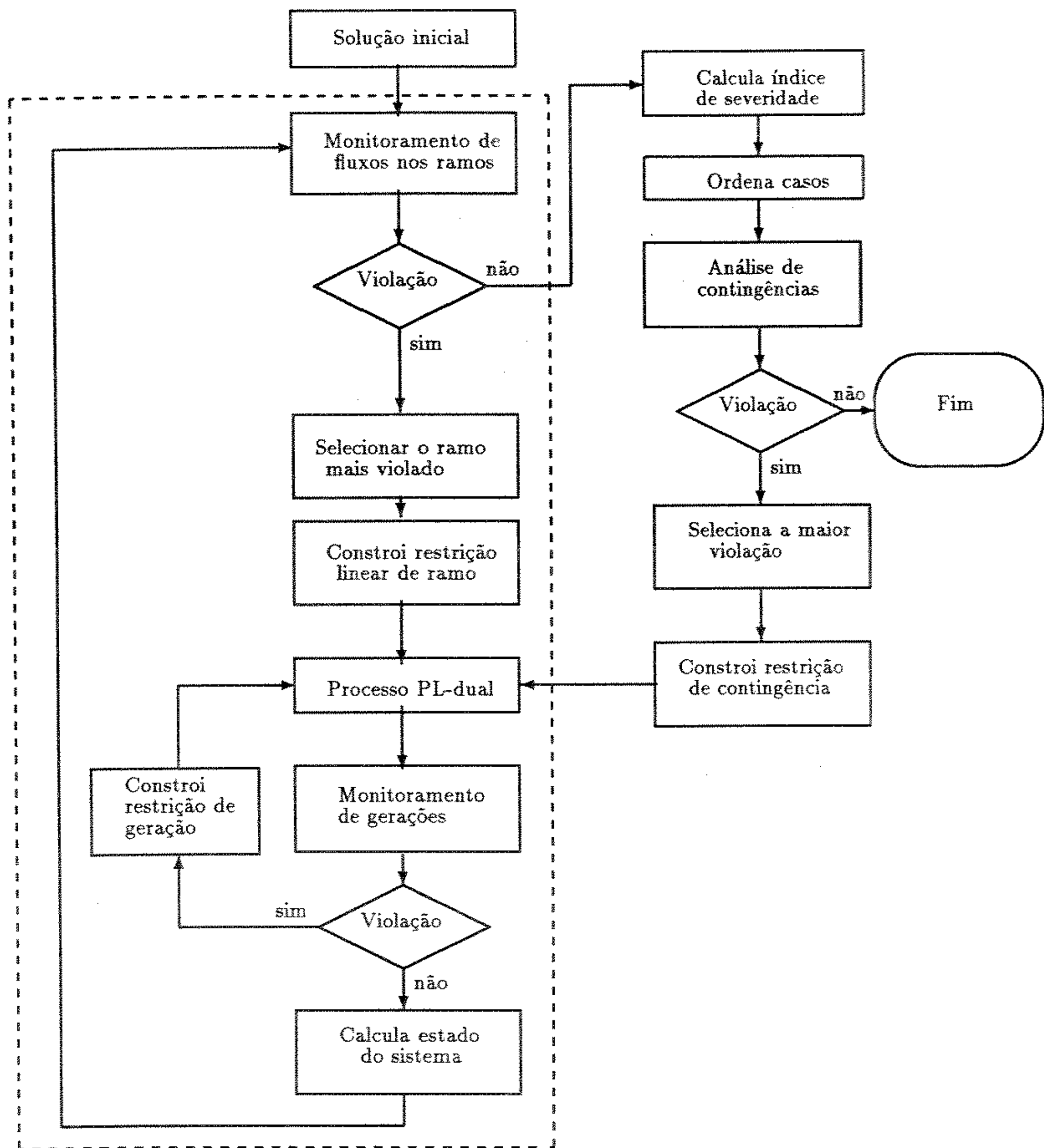

Figura 2.3: Algoritmo seqüencial do FPORS

Este problema será chamado daqui em diante de *caso base*. O processo é inicializado com uma solução "otimista" (dual factível) obtida relaxando as restrições (2.14). Se o objetivo a minimizar é custo de operação (objetivo econômico), a solução inicial pode ser obtida através de um simples despacho de "ordem de mérito". Este despacho consiste em carregar os geradores em ordem ascendente de custo incremental até satisfazer o balanço geração/carga, respeitando as restrições (2.16). No caso do objetivo mínimo desvio de um ponto especificado, o estado inicial já é o ótimo (solução otimista). Um monitor de fluxos baseado em FCCC (fluxo de carga de corrente contínua) verifica se eles estão dentro dos limites. Se não há violação, então a solução é ótima. Caso contrário, é selecionada a maior violação, para a qual é construída uma restrição em termos das variáveis de controle (ver apêndice B). Esta restrição é introduzida no caso base obtendo-se um novo ponto de operação. A obtenção de uma nova solução sempre é restrita a não ter violações nas variáveis de controle (restrições (2.16)). Como resultado deste processo, a solução é gradualmente melhorada pela adição seqüencial de restrições (2.14), até atingir a factibilidade de todas elas.

Relaxação de nível superior:

Neste nível todas as restrições de contingências são relaxadas (2.15). A solução inicial para este estágio é a resultante do nível inferior de relaxação. Essa solução -caso base - é otimista em termos de segurança, pois na sua obtenção foram ignoradas todas as restrições de contingência. Em torno do caso base é construída uma lista de contingências prováveis ordenadas por um índice de severidade (ver capítulo 3). Mediante um bloco de análise de contingências baseado em fluxo de carga CC e compensação [13], as contingências são monitoradas. Se não há casos que levem a violação pós-contingência, então a solução é ótima-segura. Caso contrário é selecionada a contingência mais severa. Constrói-se uma restrição em termos das variáveis de controle que é introduzida no caso base. Logo, analogamente ao nível anterior, aqui a solução é melhorada gradualmente em termos de segurança pela incorporação de restrições dessa natureza.

Uma breve revisão do algoritmo de programação linear utilizado neste trabalho é apresentado no apêndice A. No capítulo a seguir, são apresentados alguns conceitos de análise de segurança, com ênfase em potência ativa.

Capítulo 3

Análise de Segurança

A análise de segurança tem duas funções. Primeiro, verificar se há violações no estado operativo atual. Segundo, identificar violações em estados potenciais (análise de contingências) que decorram em perda de componentes do sistema (linhas, transformadores, geradores, etc.).

A identificação das violações no estado *operativo atual*, na sua forma mais simples, envolve o monitoramento de fluxos, tensões, etc. e sua comparação com limites pré-especificados. Também pode incluir análise de tendências com objetivo de identificar violações no futuro próximo.

A segunda função -análise de contingências- é a mais demandada em análise de segurança. Ela envolve a simulação da perda de componentes do sistema, tais como linhas, transformadores e geradores. A análise de contingências pode ser feita utilizando-se modelos dinâmicos (estabilidade transitória) ou estáticos (fluxo de carga), sendo que neste trabalho estamos interessados nestes últimos.

Praticamente é inviável se considerar todas as contingências possíveis. A análise de contingências é processada segundo uma lista prefixada, fornecida pelo planejador ou operador, ou então obtida através de critérios automáticos de seleção e ordenação, pelos quais são identificados os casos mais críticos a serem simulados. O processo de análise de contingências é tratado com maiores detalhes a seguir.

3.1 Análise de Contingências

O processo de análise de contingências é realizado com uma lista de casos "prováveis" de contingências (contingências simples ou múltiplas). Essas contingências que, se ocorrerem, podem levar a emergências, devem ser identificadas e ordenadas de acordo com a sua severidade. Dependendo de critérios de operação e da severidade do problema, haverá uma resposta a essa lista. Essa resposta pode ser implementada pelo operador do sistema

e/ou por uma função automática (função com restrição de segurança) e pode ser dos seguintes tipos:

- Modificar o estado pré-contingência com a finalidade de aliviar ou eliminar a emergência resultante de uma contingência específica.
- Realizar uma estratégia de controle que aliviará a emergência, se ela ocorrer.
- Nada fazer, na hipótese da emergência pós-contingência ser pequena e/ou improvável de acontecer.

O processo de análise de contingências "on-line" é dividido em três estágios: definição, seleção e avaliação de contingências.

3.1.1 Definição de Contingências

É a função que consome menos tempo. Neste estágio é construída a lista a ser processada, composta de casos com probabilidade de ocorrência considerada suficientemente alta. Esta lista de casos "prováveis" pode variar com a topologia do sistema, carga e fatores ambientais. Também pode incluir chaveamentos secundários (isto é, quando uma contingência resulta em várias outras contingências).

Candidatas para análise são:

- Contingências simples.
- Contingências múltiplas independentes.
- Contingências múltiplas dependentes.
- Contingências relacionadas com subestações.

A probabilidade de ocorrências múltiplas independentes em operação normal e em condições ambientais adequadas é pequena e elas podem, na maioria dos casos, serem excluídas da análise. Sob condições ambientais especiais, isto pode não ser verdade e a lista deverá ser estendida.

Para contingências simples, é fácil gerar a lista automaticamente e ela considera casos de perda de componentes do sistema tais como ramos, geradores, etc. Para contingências múltiplas dependentes e para ocorrências relacionadas com subestações, a geração da lista torna-se mais complicada e, tipicamente, tem sido uma tarefa do operador do centro de controle. Esta tarefa é apropriada para aplicações de sistemas baseados no conhecimento.

3.1.2 Seleção de Contingências

O objetivo da seleção de contingências é identificar o subconjunto de casos que causam violação no sistema. O número de candidatos é grande, e além da exatidão do processo de seleção, a velocidade é essencial. Utilizam-se modelos aproximados do sistema com técnicas computacionais adequadas para obter um resultado rápido, porém de exatidão limitada. É necessário um esquema especial de ordenação, com o fim de obter os casos de contingência classificados nos primeiros lugares. Os esquemas de ordenação baseiam-se em um índice escalar que quantifica a severidade da contingência. Estes índices escalares são normalmente identificados na literatura como *índice de severidade* ou *índice de desempenho*. Existem vários esquemas de ordenação, porém eles podem ser classificados em dois grupos: métodos diretos e métodos indiretos.

Métodos Diretos

Estes métodos envolvem a simulação de cada caso de contingência para a sua classificação, utilizando um modelo de fluxo de potência aproximado. Monitorando o cenário pós-contingência (fluxos e/ou tensões), a severidade do caso pode ser quantificada diretamente de acordo com um critério heurístico para fins de ordenação (índice de severidade).

Métodos Indiretos

Estes métodos geram *diretamente* os índices de severidade para classificação, sem calcular as grandezas contingentes monitoradas. Estes métodos são mais rápidos do que os diretos, porém menos flexíveis.

A maioria dos trabalhos de seleção de contingências têm sido desenvolvidos para o problema de potência ativa, utilizando modelos de fluxo de carga CC (FCCC). Mais adiante, na seção (3.2), são tratados especificamente estes índices de severidade para potência ativa.

3.1.3 Avaliação de Contingências

O processo de seleção ordena os casos de contingência em ordem decrescente de severidade. O processo de avaliação de contingências consiste em simular cada caso da lista em ordem decrescente utilizando fluxo de carga não linear, e ele chega até o ponto onde não é observada violação pós-contingência, ou até um número máximo de casos a serem considerados, ou até passado um período determinado de tempo.

A avaliação de uma contingência envolve tipicamente uma solução de fluxo de potência desacoplado rápido convencional, utilizando técnicas de compensação [13], não sendo requerida alta precisão na convergência.

Em alguns casos, os processos de seleção e avaliação de contingências estão fundidos num processo só.

3.2 Índices de Severidade para Potência Ativa

Os índices quantificam a severidade da contingência para determinar a ordem na lista de casos. Três desses índices baseados na severidade para potência ativa são apresentados a seguir. Os primeiros dois índices baseiam-se no princípio de mínimo esforço [15](a distribuição de fluxos de potência ativa na rede se dá segundo uma lei de mínimo esforço), e o terceiro baseia-se no nível de sobrecarga causado pela contingência.

3.2.1 Índice Baseado no Nível de Carregamento de Potência Ativa

O primeiro e mais simples índice de desempenho (PI, "perfomance index") utilizado para ordenação por severidade é baseado no nível pré-contingência de carga dos ramos. Em outras palavras, é baseado em informação local, relativa às condições individuais de carga do ramo:

$$PI_j = (T_j / T_j^{max}), \tag{3.1}$$

que é definido para cada ramo j, onde T_j é o fluxo de potência ativa do caso-base , e T_j^{max} é a capacidade máxima de transmissão.

3.2.2 Gradiente do Índice de Desempenho

O segundo índice de desempenho é um índice global, no sentido que mede as condições de carga na rede toda, e é baseado na seguinte expressão:

$$PI = \frac{1}{2} \sum_{j=1}^{m} \left(\frac{T_j^2}{\gamma_j} \right) \tag{3.2}$$

onde T_j é o fluxo de potência ativa no ramo j e γ_j é a susceptância do ramo. A aproximação de primeira ordem para a variação ΔPI_i, causada por $\Delta\gamma_i$ na susceptância do ramo, é dada por:

$$\Delta PI_i = -\frac{1}{2} \psi_i^2 \Delta\gamma_i, \tag{3.3}$$

onde ψ_i é a diferença angular no ramo i.
A expressão (3.3) é o gradiente do índice de desempenho (3.2) para a perda do componente i do sistema.

3.2.3 Índice Baseado na Sobrecarga

O terceiro índice de desempenho é dado por:

$$PI = \sum_{j=1}^{m} \left(\frac{w_j}{2n}\right) \left(\frac{T_j}{T_j^{max}}\right)^{2n} \tag{3.4}$$

onde T_j é o fluxo de potência ativa no ramo j, T_j^{max} é a capacidade de transmissão do ramo, w_j é um fator de ponderação, m é o número de ramos, e n é um inteiro. O índice PI é calculado exatamente para cada contingência utilizando os resultados de um fluxo de potência linearizado (modelo CC).

Seleção do Expoente

O expoente $2n$ é importante no comportamento do índice. Um expoente grande classificará em primeiro lugar as maiores violações. Um expoente pequeno pode classificar violações iguais ou inferiores dos casos sem violação. Este fenômeno é conhecido como *mascaramento*, e surge quando a violação em um ramo é compensado pelo decrescimento dos fluxos em outras linhas de transmissão. Na prática, os mais eficientes e confiáveis métodos para cálculo do índice de desempenho restringem o valor de n à unidade.

Capítulo 4

Processamento Paralelo

4.1 Introdução

A utilização de computadores paralelos em aplicações científicas tem sido motivada pela necessidade de se obter soluções no menor tempo possível. Uma definição de processamento paralelo amplamente aceita foi formulada em [20]:

Processamento paralelo é uma forma de processamento da informação em que dois ou mais processadores, através de um sistema de comunicação, cooperam na solução de um problema.

O processo de paralelização de um algoritmo envolve dois aspectos: *decomposição* e *coordenação*. A decomposição do algoritmo implica o desacoplamento em tarefas que possam ser executadas concorrentemente de forma eficiente. Por outro lado, algum tipo de coordenação deve existir no processo paralelo, de tal forma que a contribuição dos processadores seja adequadamente coordenada e seja garantida uma evolução segura e eficiente do processo para a solução do problema.

Neste capítulo são revisadas algumas classificações de arquiteturas multiprocessadoras e são tratadas em forma sumária suas principais características. Mais para o fim do capítulo, são revisados alguns conceitos relacionados com o desempenho de algoritmos paralelos, visando dar um suporte básico para a compreensão da proposta deste trabalho, que será apresentada no próximo capítulo.

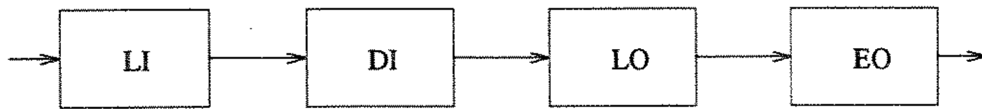

Figura 4.1: Estágios de execução de uma instrução.

4.2 Computadores Paralelos

Os computadores paralelos podem ser divididos em três tipos [20]:

i) Computadores "pipeline".

ii) Computadores "array".

iii) Computadores multiprocessadores.

Um computador "pipeline" realiza processo de cálculos sobrepostos ("overlapping"), explorando um *paralelismo temporal.* Os computadores "array" utilizam múltiplas unidades lógicas aritméticas sincronizadas para conseguir um *paralelismo espacial.* Um sistema de multiprocessamento realiza um *paralelismo assíncrono*, através de um conjunto de processadores interativos que compartilham recursos, tais como memória, dados, etc. Estes três modelos de computadores paralelos não são mutuamente excludentes, existindo na prática muitas configurações híbridas.

i) Computadores "pipeline"

Normalmente, o processo de execução de uma instrução em um computador digital envolve quatro passos (figura (4.1)):

LI: Ler a instrução na memória principal.

DI: Decodificação da instrução e identificação da operação.

LO: Ler o operando, se necessário na execução.

EO: Execução da operação decodificada.

Em um computador que não seja "pipeline", estes 4 passos devem ser completados antes que uma próxima instrução comece a ser processada. Em um computador "pipeline", instruções sucessivas são executadas em uma maneira sobreposta ("overlapping"), como é mostrado na figura (4.3).

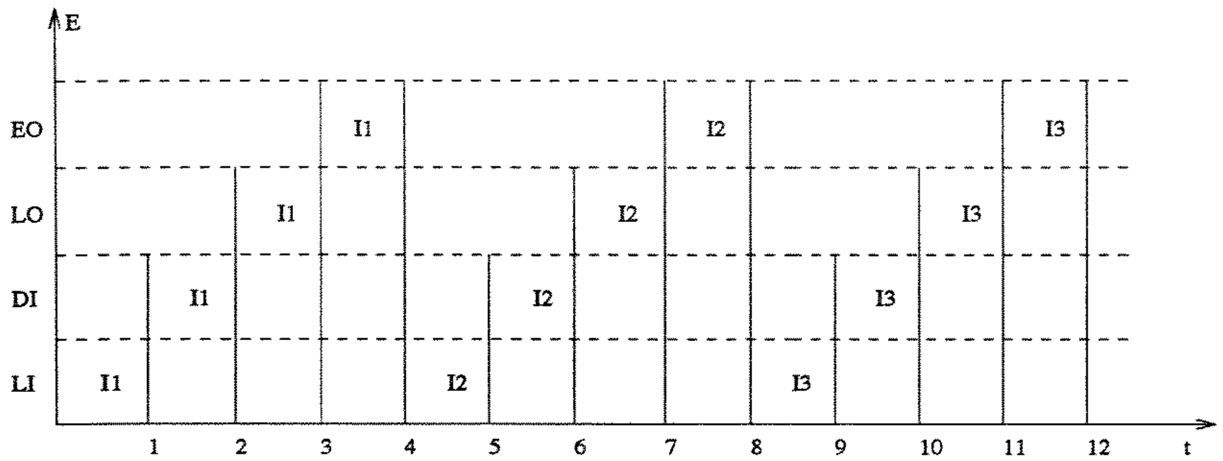

Figura 4.2: Diagrama espaço-tempo para um processador sem "pipeline".

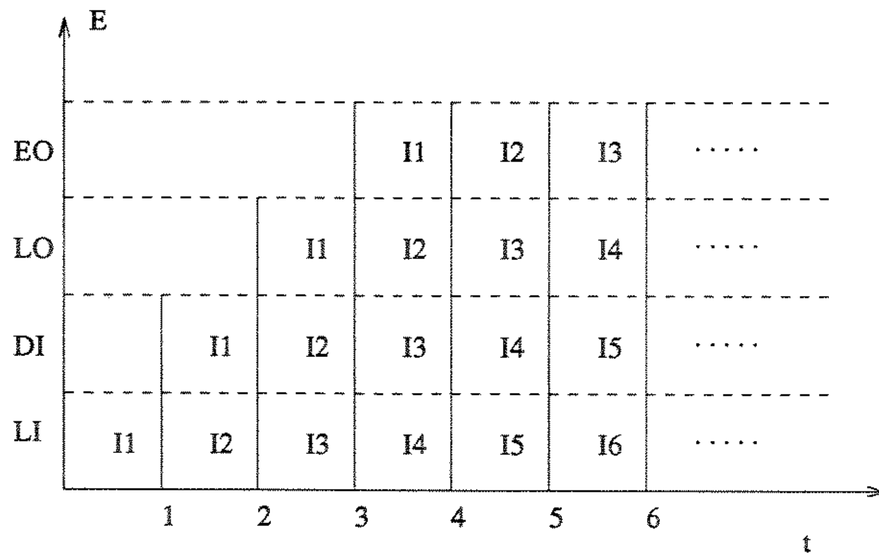

Figura 4.3: Diagrama espaço-tempo para um processador "pipeline".

O diagrama espaço-tempo da figura (4.2) mostra o caso da uma execução seqüencial dos estágios da figura (4.1). O diagrama espaço-tempo da figura (4.3) mostra o caso quando há sobreposição de execução de estágios, que corresponde ao modo de funcionamento dos computadores "pipeline".

ii) Computadores "array"

Um processador matricial ou "array" é um computador paralelo "síncrono" com múltiplas unidades de lógica e aritmética (ULA), chamadas de elemento de processamento (EP), sob a supervisão de uma única unidade de controle. Os computadores "array" executam uma única instrução cada vez, operando sobre um conjunto de dados simultâneos e são especialmente projetados para realizar operações vetoriais ou sobre matrizes. A estrutura básica de um processador "array" é mostrada na figura (4.4).

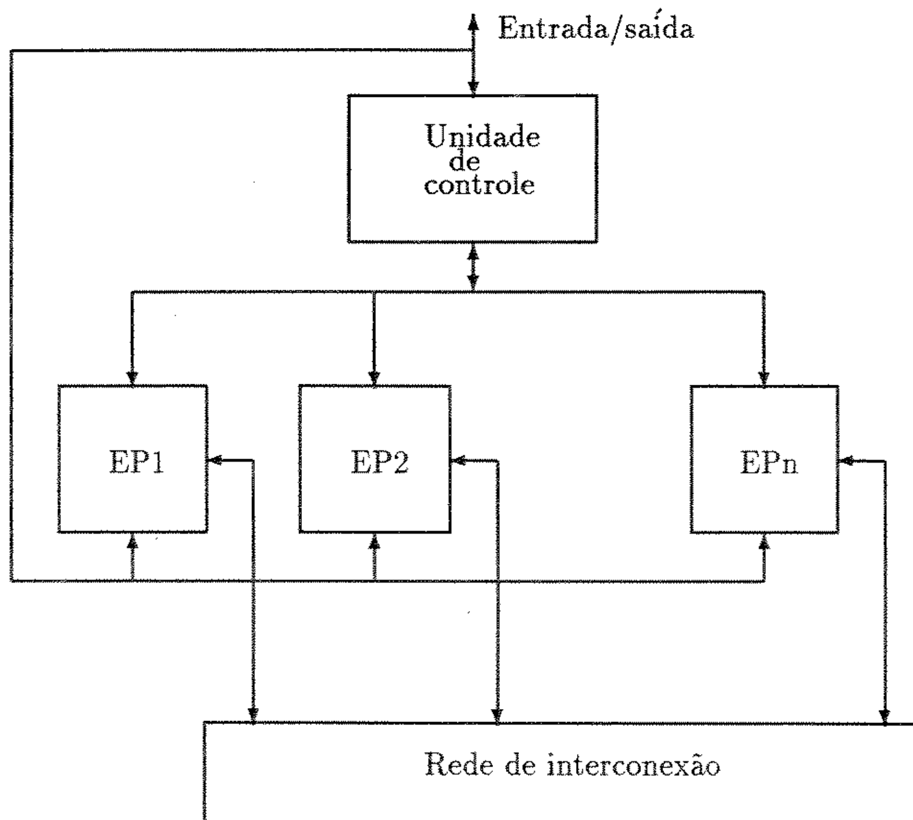

Figura 4.4: Estrutura básica de um computador "array".

ii) Computadores Multiprocessadores

Estes sistemas possuem dois ou mais processadores de capacidades comparáveis, que operam de maneira assíncrona. Todos os processadores têm memória local e outros recursos privados e podem se comunicar com outros processadores através de memórias compartilhadas ou através de uma rede de interconexão. O tratamento deste tipo de máquinas será retomado mais adiante.

4.3 Classificação de Flynn para Arquiteturas de Computadores

M. J. Flynn [20] propôs uma classificação muito utilizada do paralelismo presente em um processador. Ela é baseada no fluxo de instruções e de dados simultâneos visto pelo processador durante a execução de um programa. Flynn define quatro grupos:

- *SISD* (Única instrução - único dado / "single instruction stream - single data stream")
- *SIMD* (Única instrução - múltiplos dados / "single instruction stream - multiple data stream")
- *MISD* (Múltiplas instruções - único dado / "multiple instruction stream - single data stream")
- *MIMD* (Múltiplas instruções - múltiplos dados / "multiple instruction stream - multiple data stream")

Estes quatro tipos são ilustrados na figura (4.5). Observa-se que esta classificação depende da multiplicidade de eventos simultâneos nos componentes do sistema. Para entender esta classificação basta apenas considerar três componentes:

a) Módulos de memória (*MM*): Neles residem os dados e instruções.

b) Unidade de controle (*UC*): Tem duas funções: decodificar as instruções e enviá-las já decodificadas para a unidade de processamento.

c) Unidade de processamento (*UP*): Executa as instruções e envia os dados resultantes ao módulo ou módulos de memória.

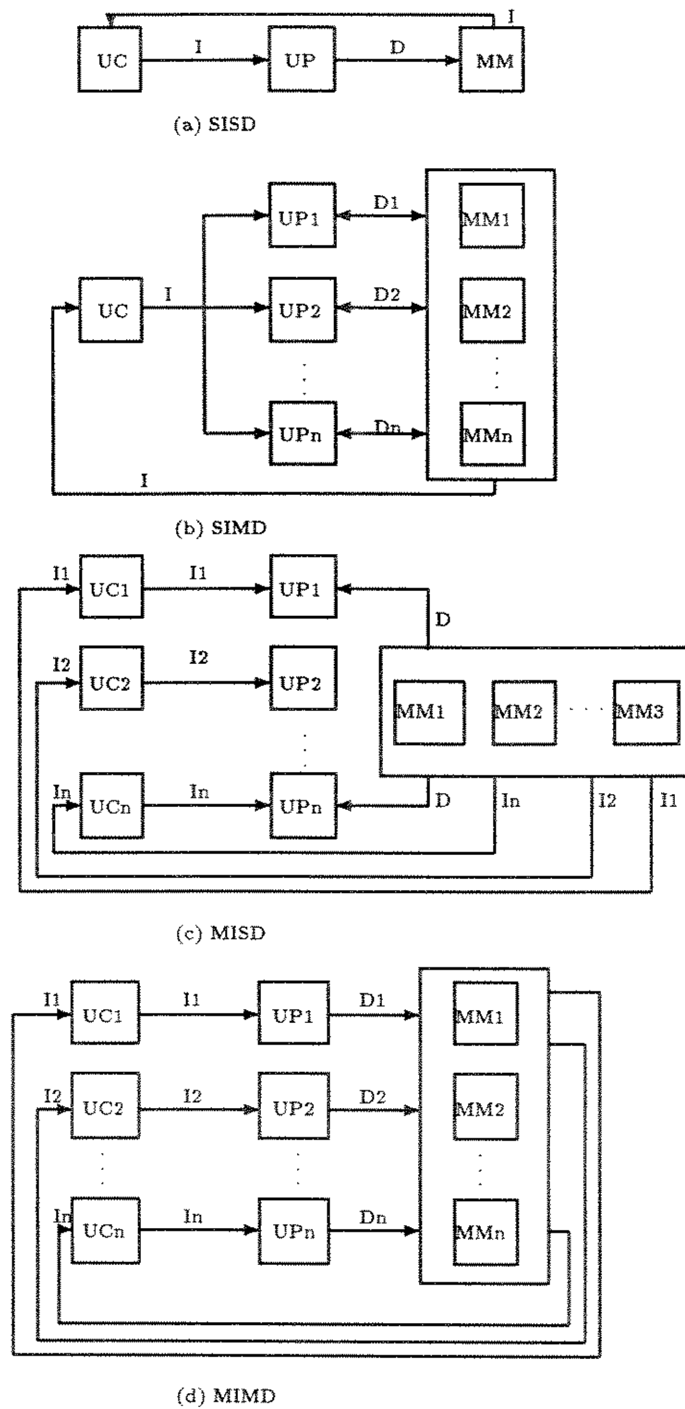

Figura 4.5: Classificação de Flynn: I= instruções, D= dados, MM= módulo de memória, UP= unidade de processamento, UC= unidade de controle

O fluxo de dados entre o módulo de memória e o processador circula em forma bidirecional; em categorias superiores, utilizam-se módulos de memória múltipla, que pela sua vez produzem fluxo de dados múltiplos [20]. Na figura (4.5) nota-se que cada fluxo de instruções é gerado por uma unidade de controle independente; o sistema de entrada/saída foi omitido pois não é fator determinante neste tipo de classificação.

SISD (Única instrução - único dado)

Este tipo de arquitetura corresponde à maioria dos computadores seqüenciais disponíveis atualmente. Apesar da execução das instruções poder ser superposta (esquema "pipeline"), os computadores desta categoria podem decodificar apenas uma instrução por vez.

SIMD (Única instrução - múltiplos dados)

Os computadores"array" entram nesta categoria. Um processador "array" executa um fluxo de instruções, mas possui várias unidades de processamento aritmético, cada uma capaz de colher e manipular seus próprios dados. Logo, em um instante determinado, uma única operação está no mesmo estado de execução em várias unidades de processamento, cada qual manipulando dados diferentes.

MISD (Múltiplas instruções - único dado)

Esta arquitetura apresenta n elementos processadores e cada um deles recebe distintas instruções que operam sobre os mesmos dados. Os resultados de um processador são a entrada (operandos) do próximo processador e assim por diante. Esta arquitetura não tem recebido muita atenção até hoje e não existem implementações comerciais dela.

MIMD (Múltiplas instruções - múltiplos dados)

Esta categoria corresponde à maioria dos sistemas de multiprocessamento. Por ser esta categoria de fundamental interesse neste trabalho, ela será tratada separadamente na próxima seção.

Devido ao surgimento dos modernos supercomputadores, a classificação de Flynn tem sido discutida [19] [16], pela dificuldade de incluir essas novas arquiteturas na classificação. Recentemente, em [16] foi proposta uma classificação de alto nível para arquiteturas de computadores paralelos, que basicamente preserva os elementos da taxonomia de Flynn e incorpora arquiteturas recentes que apresentam méritos de serem incluídas dentro das arquiteturas paralelas.

4.4 Computadores Multiprocessadores

As estruturas baseadas em multiprocessadores podem ser caracterizadas por dois atributos: Uma arquitetura com multiprocessadores deve ser considerada como um computador

integrado, com múltiplos processadores. Estes processadores podem comunicar-se e cooperar em diferentes níveis para resolver um determinado problema.

Existem algumas semelhanças entre arquiteturas baseadas em *multiprocessadores* e sistemas de computadores múltiplos, *os multicomputadores*, pois ambos são motivados com o mesmo objetivo básico, que é permitir operações concorrentes no sistema. Porém, existe uma importante diferença entre eles, baseada no grau de compartilhamento de recursos e no nível de cooperação para a solução de um problema. Um sistema de computadores múltiplos consiste de vários computadores autônomos que podem ou não se comunicar. Um sistema multiprocessador é controlado por um único sistema operacional, que gerencia a interação entre os processadores e seus programas nos níveis de processo e estrutura de dados.

Nos computadores multiprocessadores, a comunicação entre processadores pode ser realizada de duas formas:

- Via memória comum; neste casos os processadores compartilham uma memória e através dela transferem mensagens. As arquiteturas com este modelo de comunicação são conhecidas como de *memória compartilhada.*
- Mediante troca de mensagens; cada processador tem sua memória local própria. Esta arquitetura é conhecida como de *memória distribuída.*

4.4.1 Arquiteturas de Memória Compartilhada

Esta arquitetura também é conhecida como de memória comum, sendo que estas duas denominações serão utilizadas indistintamente no decorrer deste trabalho. Nesta arquitetura os processadores intercambiam mensagens através de uma memória comum. Cada processador possui uma pequena memória local ou um "buffer" de alta velocidade (memória "cache"). Este tipo de arquitetura se caracteriza pela velocidade de comunicação interprocessador, que é definida pela largura de banda de cada processador [19]. Porém, pode acontecer congestionamento no acesso à memória devido ao tipo de barramento utilizado e sincronização de acesso aos dados compartilhados. Em outras palavras, são necessários mecanismos de sincronização para evitar que endereços compartilhados sejam acessados por mais de um processo simultaneamente.

Nas figuras (4.6) e (4.7) são apresentados dois esquemas de conexão entre processadores e memórias. O mais simples (figura 4.6), consiste em um barramento ao qual estão conectados os processadores e os blocos de memória compartilhada. Como o barramento também é um recurso compartilhado, deve existir um mecanismo que gerencie o acesso; estes mecanismos incluem a utilização de: a) prioridades fixas (ou estáticas), e b) filas tipo "fifo", ou "daisy chain". Este esquema não pode suportar muitos processadores, e

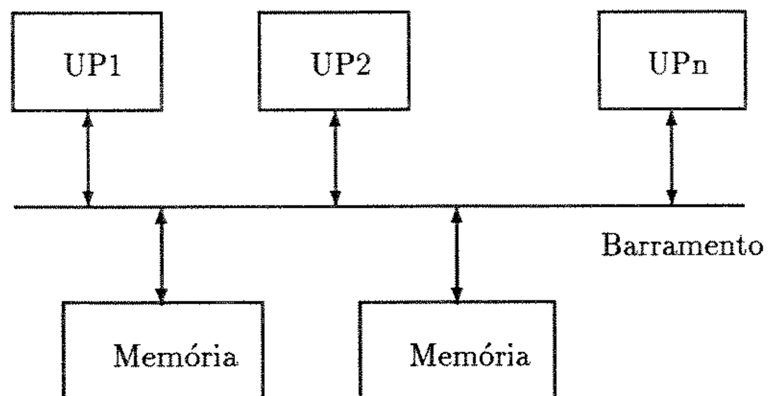

Figura 4.6: Modelo de barramento comum e memória compartilhada.

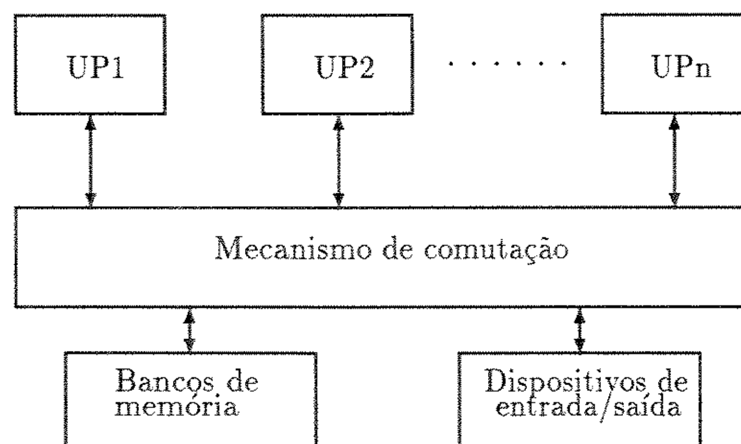

Figura 4.7: Modelo com mecanismo de comutação central

apenas um de cada vez pode acessar o barramento.

Outro esquema [19] assume que os processadores conectam-se a uma memória global compartilhada através de um mecanismo de comutação central ("switching", figura 4.7). Existe uma grande variedade de formas de implementar este mecanismo de comutação; dois exemplos são a utilização de um comutador tipo "crossbar", ou uma rede de interconexão multi-estágio. No caso de sistemas que utilizam "crossbar", o custo do comutador pode ser um fator dominante, limitando o número de processadores que podem ser conectados. Uma alternativa são os multiprocessadores com redes de interconexão multi-estágio, que podem conter um grande número de processadores. Uma rede multi-estágio $n * n$, conectando n processadores a n memórias, apresenta múltiplos *estágios* ou bancos de comutadores nas rotas através da rede de interconexão. Uma característica deste tipo de interconexão é que consegue uma significante *expandibilidade*.

Alguns dos problemas que podem afetar a eficiência das arquiteturas de memória compartilhada são os relacionados com o fenômeno de contenção ("contention"), que decorre na limitação do número de processadores [28] [29]. O fenômeno de contenção pode apresentar-se de três formas:

a) *Contenção de memória*: Acontece quando vários processadores tentam acessar um determinado módulo de memória ao mesmo tempo.

b) *Contenção de comunicação*: Ocorre quando vários processadores tentam utilizar a rede para acessar módulos de memória.

c) *Tempo de latência*: O tempo gasto pelos processadores no acesso aos módulos de memória através do sistema de interconexão.

Dependendo do tipo de aplicação (por exemplo, processos com elevada taxa de comunicação), o fenômeno de contenção pode ter um considerável impacto na eficiência. Os sistemas com barramento comum podem suportar eficientemente poucos processadores - na ordem de dez - pudendo chegar até perto de 30 [21]. Fora deste intervalo, a contenção do barramento provoca forte degradação no desempenho da arquitetura. No caso das arquiteturas com conexão através de redes, a alta complexidade e custo delas com relação à taxa de expandibilidade obtida as torna pouco atrativas.

Algumas estratégias para reduzir os problemas contenção nas arquiteturas de memória compartilhada são tratadas em [28] [29].

4.4.2 Arquiteturas de Memória Distribuída

Um sistema de memória distribuída consiste de múltiplos processadores autônomos que não compartilham memória, mas que cooperam através de troca de mensagens usando uma rede de comunicação [23]. Nestes sistemas, cada processador executa suas próprias

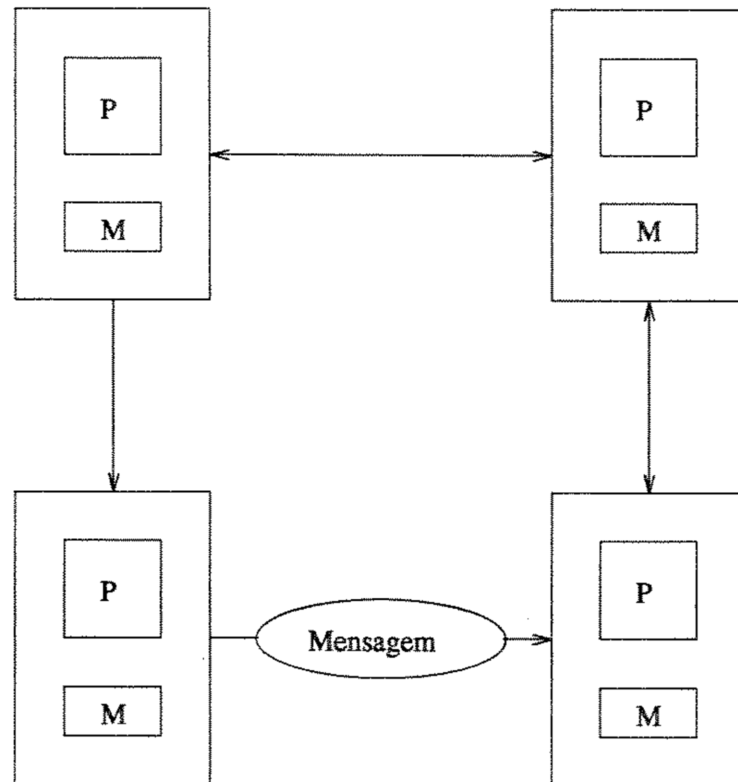

Figura 4.8: Modelo básico de memória distribuída.

instruções e dados, ambos armazenados em sua memória local. A comunicação é realizada através de um mecanismo de *troca de mensagens*. As arquiteturas de memória distribuída basicamente obedecem à estrutura da figura (4.8).

Estas arquiteturas têm sido motivadas visando estruturas de multiprocessamento *escaláveis* (que possam ser expandidas) e que satisfaçam requerimentos de aplicações científicas de grande porte. Várias topologias de interconexão têm sido propostas com o objetivo de permitir a *expandibilidade* da arquitetura e fornecer condições para um desempenho eficiente para programas paralelos com diferentes padrões de comunicação. Na figura (4.9) são apresentadas algumas topologias para arquiteturas de memória distribuída.

Arquiteturas em Topologia Tipo Anel

Nas topologias tipo Anel, o diâmetro de comunicação (maior "distância" entre dois processadores) é $N/2$, onde N é o número de processadores que formam parte do anel. Este

diâmetro pode ser reduzido pelo adição de conexões extras entre alguns processadores (conexões tipo "secante"). Usando esta estratégia ou anéis múltiplos, esta arquitetura aumenta seu nível de tolerância a falhas. A arquitetura tipo Anel é mais apropriada para poucos processadores, executando algoritmos com baixa taxa de comunicação.

Arquiteturas em Topologia Tipo Malha

Uma topologia tipo malha bi-dimensional tem $N = n^2$ nós conectados segundo a figura (4.9-b). O diâmetro de comunicação é $2(n-1)$. Conexões adicionais entre extremos da malha permitem algumas vezes reduzir o diâmetro de comunicação para $2 * int(n/2)$. A pesquisa deste tipo de arquitetura tem sido estimulada pela correspondência entre malhas e algoritmos orientados a matrizes.

Arquiteturas em Topologia Tipo Árvore

As arquiteturas tipo Arvore têm sido desenvolvidas para suportar algoritmos de busca e ordenação, processamento de imagens e outros. O diâmetro de comunicação é $2(l-1)$, onde l é o nível de profundidade da árvore.

Arquiteturas em Topologia Tipo Cubo

A topologia *hipercubo* ou cubo booleano está formada por $N = 2^n$ processadores arranjados em um cubo n-dimensional. Cada nó tem n "links" que o conectam aos nós adjacentes. Cada nó é identificado de maneira unívoca com valores numéricos de n bits, que vão de 0 até $N-1$ e associados de forma que o valor de cada nó adjacente é diferenciado em apenas um bit. O diâmetro de comunicação do hipercubo é n.
A pesquisa da arquitetura hipercúbica tem sido fortemente influenciada pelo interesse de desenvolver arquiteturas escaláveis que suportem os requerimentos de desempenho das aplicações científicas em três dimensões [16].

4.5 Desempenho de Algoritmos Paralelos

Quando um multiprocessador está operando no máximo do seu desempenho, todos os processadores realizam trabalho útil, não existindo processadores ociosos e não são executadas tarefas duplicadas. Nesse nível de desempenho, os n processadores que compõem o computador paralelo contribuem efetivamente no desempenho global e o tempo de execução equivale ao tempo da solução seqüencial dividido por n. Na maioria das aplicações, o desempenho máximo raramente é atingido [21], exceto em algumas aplicações de algoritmos assíncronos [19]. Existem vários fatores que contribuem para a queda da eficiência. Alguns deles são:

- Atraso introduzido pela comunicação entre processadores.

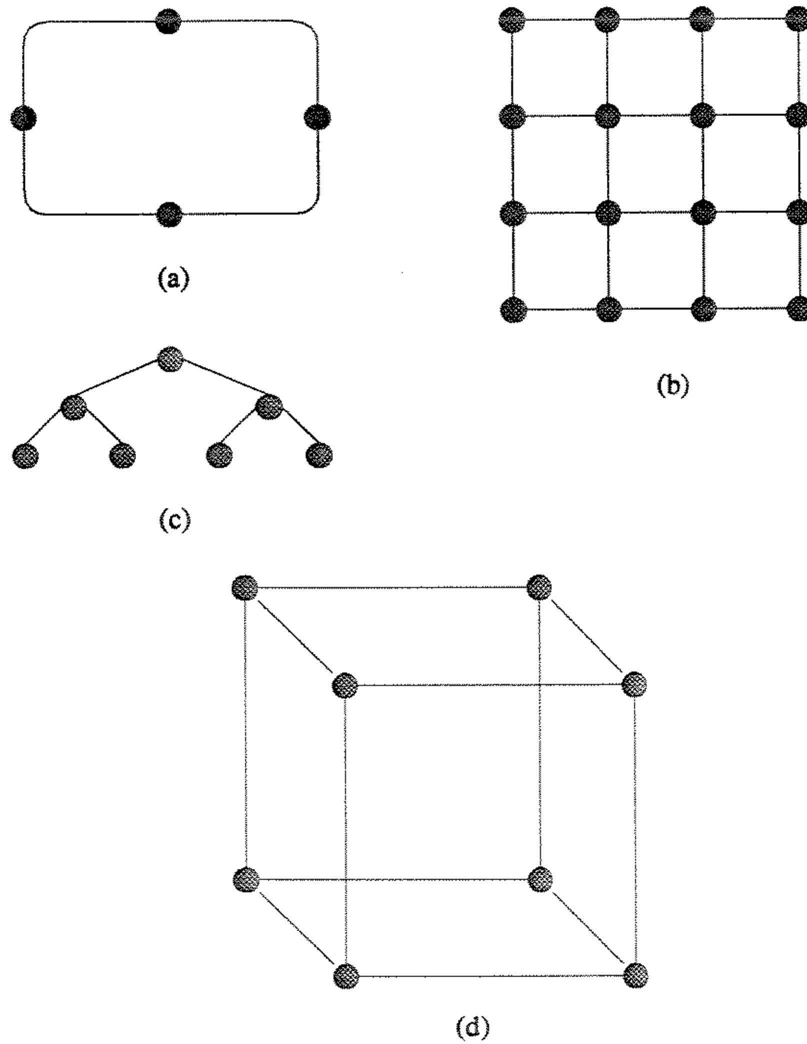

Figura 4.9: Topologias de redes de interconexão em máquinas de memória distribuída: (a) anel; (b) malha; (c) árvore; (d) hipercubo.

- O "overhead" devido ao nível de sincronismo entre tarefas. Este fator está relacionado com a dependência entre tarefas alocadas em diferentes processadores na resolução do processo global.
- O "overhead" devido ao esforço despendido por alguns processadores, quando mais de um executa a mesma tarefa.

O primeiro fator está relacionado com a taxa de granularidade (definida a seguir) do processo paralelo. O segundo fator leva ao desbalanceamento da carga, pois alguns processadores ficam esperando resultados de outros processadores para começar a trabalhar. O terceiro fator está relacionado com a coordenação do processo paralelo.

Na implementação paralela da solução de um problema, estes fatores deverão ser levados em conta, pois eles contribuem para a ineficiência. Os processos de ***decomposição*** em tarefas independentes ou fracamente acopladas e a ***coordenação*** do processo global devem ser realizados visando minimizar o impacto desses fatores.

Os problemas de ineficiência tendem a crescer com o número de processadores. Na prática, observa-se que a eficiência decresce rapidamente, levando à saturação da curva de ganho (ou seja, o aumento do número de processadores ativos não consegue diminuir o tempo global de processo), e eventualmente a ganho negativo (a participação de mais processadores provoca o aumento do "overhead" de comunicação). Uma estimativa do ganho ideal a ser obtido com um algoritmo paralelo pode ser obtido a partir da *Lei de Amdahl* [19]:

$$G = T/(T_s + T_p/n) \tag{4.1}$$

onde:
T: Tempo de execução seqüencial do algoritmo completo;
T_s: Tempo de execução da porção não-paralelizável do programa;
T_p: Tempo de execução da porção paralelizável do programa;
n: Número de processadores.

A expressão (4.1) pode ser rescrita assim:

$$G = 1/(f_s + f_p/n) \tag{4.2}$$

onde f_s e f_p são as frações não paralelizáveis e paralelizáveis, respectivamente, do algoritmo seqüencial. Quando n cresce, o limitante superior de G é dado por $1/f_s$; ou seja, o *máximo* ganho do algoritmo paralelo estará limitado pela fração não paralelizável do algoritmo. Esta estimativa é ideal, pois não está levando em conta os aspectos de comunicação, sincronização, contenção, etc.

A lei de Amdahl é, em princípio, um argumento forte em contra do futuro da computação paralela. Porém, existem algoritmos paralelos que têm poucas operações seqüenciais;

conseqüentemente, o argumento de Amdahl é útil como uma maneira de determinar se um algoritmo é um bom candidato para ser paralelizado [19].

Granularidade

O desempenho de uma implementação paralela é fortemente afetado pelo processo de decomposição do problema original. A taxa de granularidade é dada pela razão *tempo de processamento (P)* v-s *tempo de comunicação (C)*. Um paralelismo de alta granularidade implica em uma taxa P/C relativamente alta, onde cada unidade de cálculo produz um nível de comunicação relativamente pequeno. Em um paralelismo de *grão fino*, a taxa P/C é muito pequena , tendo um "overhead" por comunicação relativamente grande comparado com o tempo de processamento da tarefa. Uma boa decomposição deve identificar tarefas com altas taxas P/C, pois assim o impacto do "overhead" de comunicação é minimizado.

4.5.1 Medidas de Qualidade de Algoritmos Paralelos

Duas importantes medidas da qualidade dos algoritmos paralelos quando são implementados em computadores multiprocessadores, são o "speed-up" ou ganho e a eficiência.

"Speed-up"

O "speed-up" ou ganho, é definido pela razão:

$$S = t_s/f_p \tag{4.3}$$

onde t_s é o tempo de execução do melhor algoritmo seqüencial e t_p é o tempo do algoritmo paralelo utilizando p processadores.

Outras versões definem o "speed-up" considerando o tempo levado pelo algoritmo em um processador, dividido pelo tempo de execução em p processadores. Esta definição é enganosa, pois o algoritmo paralelo freqüentemente contém operações extras para acomodar o paralelismo. Esta definição leva a valores de "speed-up" otimistas, pois mascara o "overhead" do algoritmo paralelo.

Eficiência

A eficiência de um algoritmo paralelo executado em p processadores é definido como:

$$\epsilon = S/p \tag{4.4}$$

onde S é o "speed-up" atingido com p processadores. Assim, por exemplo, se o tempo da melhor versão seqüencial é 8 segundos, e um algoritmo paralelo resolve o mesmo problema em 2 segundos utilizando 5 processadores, então o algoritmo apresenta um "speed-up" de 4 com cinco processadores e uma eficiência de 80 %.

4.5.2 Portabilidade

No desenvolvimento de "software" aplicado, a maior porcentagem dos esforços está sendo direcionado para obter um produto que possa operar em vários ambientes computacionais diferentes. A portabilidade das aplicações é cada vez mais citada como uma meta requerida no desenvolvimento de "software". Porém, não há ainda concordância com relação ao exato significado do conceito de "portabilidade". Uma definição útil foi proposta por Mooney [27]:

> *Uma aplicação é portável através de uma classe de ambientes, na medida que o esforço requerido para transportá-la e adaptá-la ao novo ambiente é menor que o esforço de refazê-la.*

O objetivo primário da portabilidade é facilitar a atividade de "migração" da aplicação de um ambiente, onde ela atualmente se encontra, para outro. Neste processo observam-se dois aspectos:

i) *Transporte* : Movimentação física do código e dados da aplicação para o novo ambiente.

ii) *Adaptação* : Modificação da informação na medida que for necessário para uma operação satisfatória no novo ambiente.

Os problemas de transporte físico envolvem a utilização de meios compatíveis ou canais de comunicação, interpretação e transformação de formatos de arquivo, códigos, representação de dados, etc. A adaptação envolve modificações de alto nível que são necessárias para ajustar o programa para operar no novo ambiente. Estes ajustes podem ser de tipo obrigatórios, devido à diferença de ambientes, por razões de eficiência, ou outro critérios.

Uma atitude que facilita o processo migratório é observar a meta de portabilidade já durante o desenvolvimento da aplicação. Em outras palavras, a aplicação deve ser desenvolvida pensando que ela poderá (ou deverá) migrar para outras arquiteturas computacionais. A utilização de modelos ou paradigmas computacionais auxiliam esta tarefa [17]. A portabilidade direta raramente é atingida, pois alguns ajustes deverão ser feitos, seja para incluir as facilidades disponíveis no novo ambiente, ou por razões de eficiência.

Capítulo 5

Solução Concorrente do Fluxo de Potência Ótimo com Restrições de Segurança

Como foi analisado no capítulo 2, no algoritmo a ser utilizado são identificados dois níveis de relaxação. O nível de relaxação inferior corresponde ao problema originalmente formulado por Stott e Marinho para despacho de potência ativa [4], sendo amplamente reconhecida sua eficiência e rapidez nesse tipo de aplicação. O nível de relaxação superior é uma extensão do algoritmo original e envolve a inclusão de restrições de segurança. Como foi comentado na seção (2.3), o número de contingências a ser considerado pode atingir vários milhões, o que torna atrativa a resolução mediante processamento paralelo desta fase.

Este capítulo apresenta a proposta de implementação concorrente do problema de fluxo de potência ótimo com restrições de segurança e está assim organizado:

Inicialmente se retoma a formulação linear seqüencial do problema; faz-se uma análise para identificação das tarefas que serão executadas em paralelo. Utiliza-se um modelo de programação de programação abstrato que independe do ambiente físico a ser utilizado para execução das tarefas. O algoritmo concorrente proposto é apresentado. Mais para o fim são tratadas as implementações em duas arquiteturas particulares disponíveis (memória compartilhada e memória distribuída).

5.1 Decomposição do Problema

Esta seção discute a paralelização da solução do problema do fluxo de potência ótimo com restrições de segurança. Inicialmente o modelo de programação concorrente é discutido

de maneira abstrata; mais para o fim da seção é então apresentado o mapeamento em arquitetura paralela.

5.1.1 Granularidade e "Overheads"

Para o mapeamento do problema num ambiente concorrente é necessário decompor o algoritmo de resolução em "tarefas". Este processo de decomposição passa necessariamente pela identificação das tarefas que levem a "grãos" relativamente grandes; em outras palavras, tarefas que apresentem uma alta relação tempo de processo versus tempo de comunicação. A escolha destas tarefas influenciará diretamente o desempenho da implementação concorrente da solução. Por outro lado, o processo de decomposição não leva necessariamente a uma carga computacional que seja uniformemente distribuída entre os processadores da máquina paralela.

Assim, uma decomposição eficiente requer a minimização de dois tipos diferentes de *overheads*:

(1) Um overhead devido à necessidade de comunicação entre as tarefas nas quais o problema original é decomposto;

(2) Um overhead causado pelo desbalanceamento de carga entre os vários processadores da máquina paralela.

O primeiro *overhead* está relacionado ao tamanho do "grão" computacional que resultará da decomposição. A segunda fonte de ineficiência está relacionada ao nível de dependência das tarefas a serem executadas em paralelo. A existência de pontos de sincronismo entre elas motiva ociosidade de processadores, à espera que outros terminem suas tarefas.

5.1.2 O que será paralelizado

Relembrando a formulação linearizada do problema de fluxo de potência ótimo com restrições de segurança, a figura (5.1) mostra de maneira abreviada o algoritmo seqüencial de solução do problema.

Como foi comentado no capítulo (2), o nível de relaxação inferior do algoritmo (identificado na figura (5.1) como *pl caso base*) tem desempenho extremamente eficiente, não sendo convidativa sua decomposição para implementação concorrente (grãos computacionais finos). Por outro lado, o nível superior de relaxação se adequa naturalmente à implementação concorrente, pelas seguintes razões:

(1) Tarefas de granularidade adequada (análise de contingências).

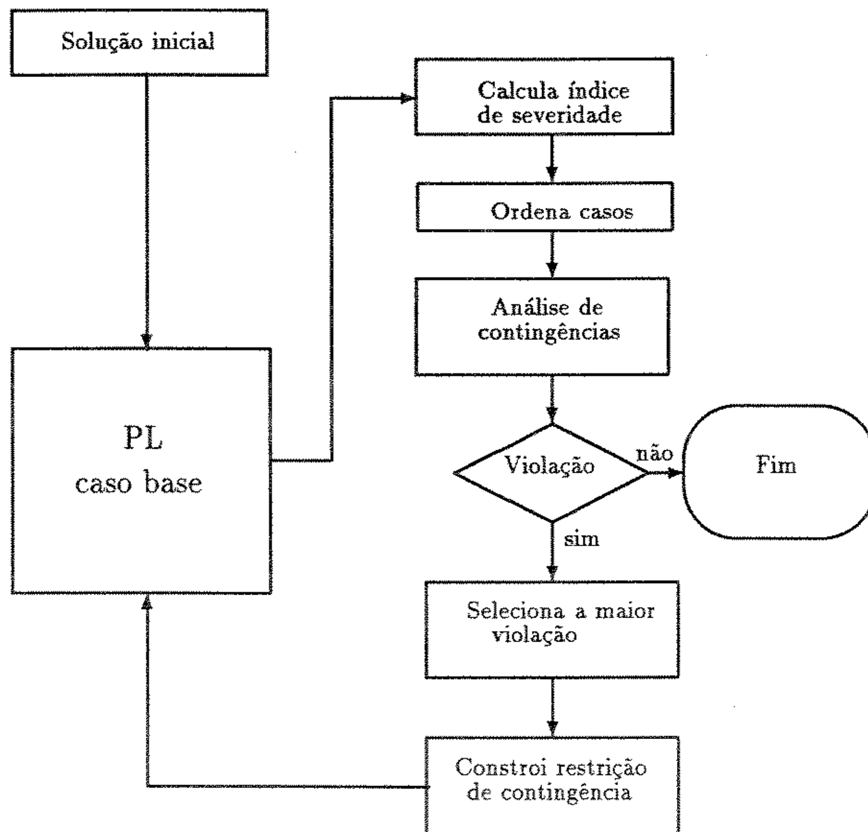

Figura 5.1: Algoritmo seqüencial abreviado.

(2) Cada cenário pós-contingência pode ser tratado de maneira quase independente.

O processo de análise de contingência tem boa granularidade, pois para sua execução precisa-se do estado atual do sistema (vetor de estado) e de um caso para analisar (o índice que identifica o caso). Como foi comentado na seção (2.2.4), o número de restrições que ficam ativas no ponto ótimo é pequeno, logo o número de vezes que o estado do sistema deve ser remetido ao processo de análise de contingências também é pequeno.

A independência do tratamento dos cenários pós-contingência não é completa, devido à solução do caso base ser alterada como resultado da adição de novas restrições; logo, pode acontecer que as contingências tenham que ser verificadas novamente para esse ponto de operação. Pelas mesmas razões citadas no parágrafo anterior, esta dependência é muito fraca, pois o ponto de operação é alterado um número de vezes relativamente pequeno, comparado com o número de contingências a serem analisadas. Por outro lado, na prática, o tipo mais freqüente de contingência relacionado com potência ativa é do tipo *dependente* (ver seção (3.1.1)). Logo, é freqüente que a eliminação de uma violação (em geral a maior), decorra na eliminação de outras violações. Isto motiva que a expectativa de evolução ao ótimo seja drasticamente superada, pois não se verificarão subseqüentes alterações na solução ótima atual, nem violações nos cenários pós-contingência.

Identificação das Tarefas

Para minimizar ou reduzir o *overhead* de comunicação, a decomposição deve ser feita de tal modo que a relação entre tempos de comunicação e os tempos de computação seja minimizada. No caso do fluxo de potência ótimo com restrições de segurança a decomposição leva a quatro tipos básicos de tarefas:

(a) otimização ;

(b) ordenação de contingências;

(c) cálculo de índice de severidade de contingências;

(d) análise de contingência e geração de restrições.

Este tipo de decomposição que é naturalmente sugerido pela estrutura do problema original, leva a tamanhos de grãos para os quais pode-se conseguir relações comunicação/computação aceitáveis.

Para minimizar ou reduzir o *overhead* por desbalanceamento de carga, adota-se um estilo de programação assíncrona. Isto é, as tarefas (a-d) podem ser executadas *assincronamente* em um ou mais processadores.

5.2 Modelos de Programação

Na formulação da solução concorrente do problema são invocados modelos ou paradigmas de programação que constituem abstrações da máquina paralela e do tipo do problema a ser resolvido. Assim, em vez de simplesmente paralelizar um algoritmo seqüencial existente, procuram-se soluções que, além se ser eficientes, preservem um bom grau de portabilidade entre diferentes arquiteturas e possam abranger formulações alternativas do problema [17].

5.2.1 Modelo de Programação Concorrente

O modelo de programação concorrente é dado na Figura (5.2). Este modelo dá uma visão abstrata do programa real que vai ser executado em paralelo. Esta abstração independe do ambiente físico no qual o programa vai ser executado; logo, os esforços são concentrados nos métodos de solução em vez de desviar as atenções para as particularidades de cada máquina, facilitando o desenvolvimento de soluções que independam, até um certo grau é claro, da arquitetura utilizada. O nível de portabilidade total dificilmente é atingido, pois, a partir de certo ponto a eficiência dos programas pode ser comprometida.

O modelo de programação da figura (5.2) (também chamado de paradigma de programação) é composto de quatro estruturas básicas:

- Tarefa mestre.
- Tarefas escravas.
- Comunicações.
- Base de dados.

Na implementação atual do método proposto, as tarefas que foram obtidas na decomposição do problema são mapeadas neste modelo da seguinte forma:

Tarefa Mestre: A Tarefa Mestre é composta das seguintes subtarefas:

- *Inicialização* de tarefas escravas
- Algoritmo de *otimização*
- *Ordenação* das contingências
- Critério de parada

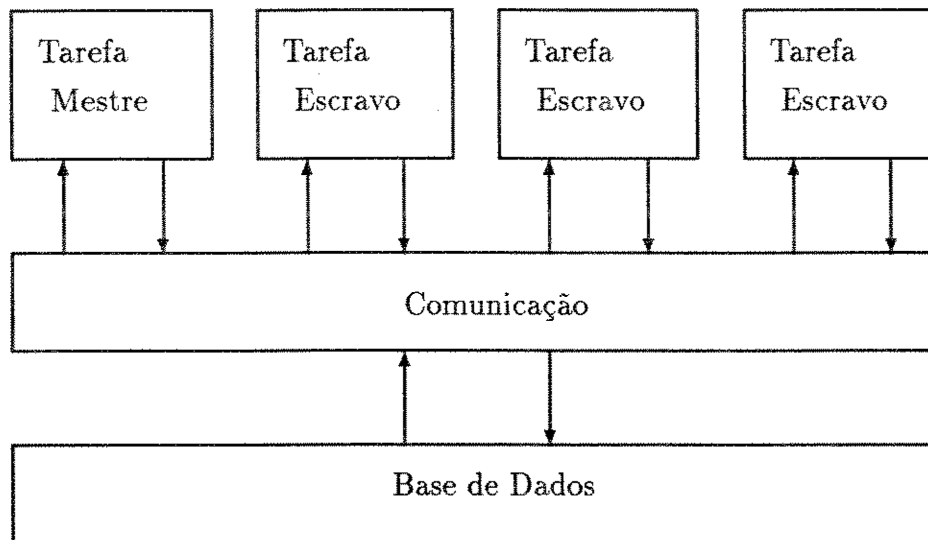

Figura 5.2: Modelo de Programação Concorrente.

Tarefas Escravas:

As tarefas escravas são compostas de duas subtarefas básicas:

- *Cálculo de índices* de severidade de contingências;
- Análise de contingências e *geração de restrições.*

Modelo Produtor-Consumidor

As subtarefas de *otimização*, *ordenação*, *cálculo de índices* e *geração de restrições* podem ser executadas assincronamente, em um ou mais processadores. Cada uma dessas subtarefas produz dados que são consumidos por uma ou mais das demais subtarefas, e, ao mesmo tempo, consome dados produzidos pelas outras subtarefas, conforme sumarizado na Figura (5.3). Este é um caso típico de abstração *produtor-consumidor* [18], normalmente utilizada em uma ampla gama de aplicações de programação concorrente.

Assim a subtarefa de *otimização* é consumidora de restrições e gera novos pontos de operação (novos estados); lembrar que inicialmente todas as restrições de segurança estão relaxadas e a solução básica inicial é obtida por esta subtarefa considerando o subproblema (2.17). Na medida em que novas restrições ativas são encontradas por outras subtarefas, a subtarefa *otimização* altera seu ponto base de operação.

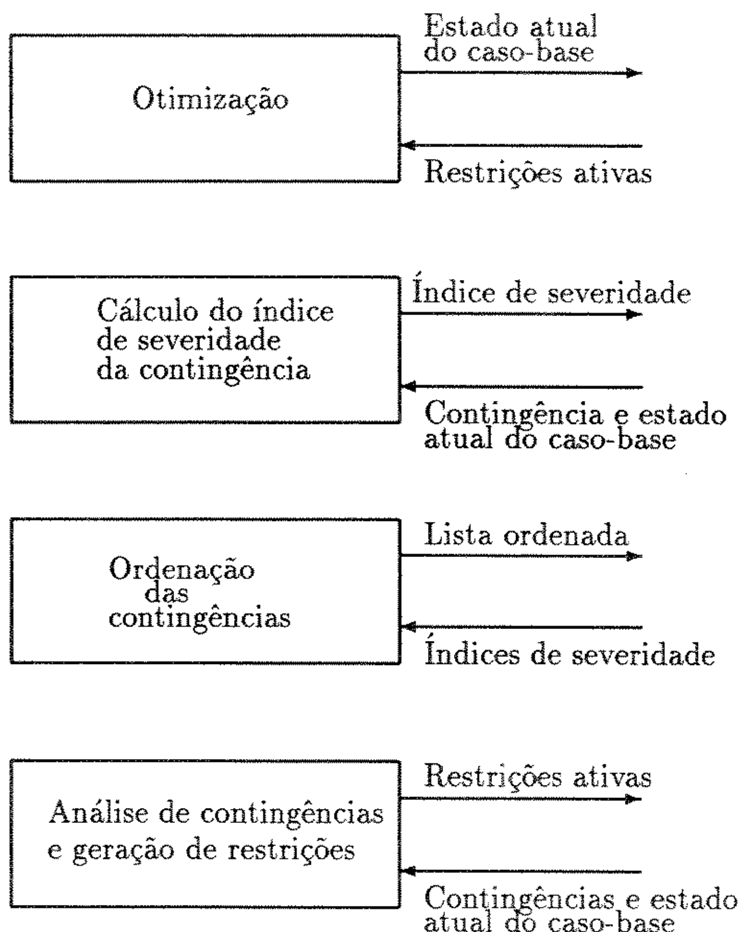

Figura 5.3: Subtarefas do modelo produtor-consumidor.

A subtarefa *ordenação* é consumidora de índices de severidade e produtora de uma lista ordenada. A ordenação da lista de contingências prováveis (que pode ser pré-especificada) é feita de acordo com índices de severidade determinados pela subtarefa *cálculo de índices*, e seleciona um subconjunto de contingências mais críticas (o processo de análise de contingências se concentra neste subconjunto de casos, relaxando temporariamente o resto da lista, pois há boa probabilidade desses casos serem responsáveis de toda ou de uma considerável porcentagem da infactibilidade do ponto de operação atual). Normalmente a subtarefa *ordenação* só é chamada uma vez, no início do processamento paralelo (opcionalmente, se a lista disponível na base de dados já estiver ordenada, a tarefa não precisa ser executada nem uma vez).

A subtarefa *cálculo de índices* consome o caso de contingência a ser simulado e o estado atual do sistema, produzindo os correspondentes índices de severidade para cada contingência. Estas subtarefas são ativadas concorrentemente pelas tarefas escravas e calculam os índices de severidade de acordo com as equações da seção (3.2).

A subtarefa *geração de restrições* consome casos de contingência para serem simulados e estados (valor mais recente do vetor de estado do sistema), gerando como produto restrições de contingências para os casos que apresentem violação. Esta tarefa é ativada concorrentemente pelas tarefas *escravas*.

O estilo de programação assíncrono adotado neste trabalho implica que o consumo e geração de informação por parte de cada bloco da figura (5.3) seja assíncrono. Assim, tarefas mais rápidas podem fornecer resultados mais rapidamente e, eventualmente consumir dados com maior velocidade. Por outro lado, uma tarefa que produz um resultado, deverá remetê-lo à outra tarefa para assim ela ficar liberada. Isto é resolvido utilizando fila tipo *fifo* (first-in-first-out), que formam parte da biblioteca de comunicação utilizada [18], cujos detalhes serão tratados mais adiante. Estas filas são utilizadas para colocar os resultados produzidos por uma tarefa, para sua posterior utilização por outra. Assim, identificam-se basicamente duas filas:

- Fila de restrições
- Fila de índices de severidade

Estado do sistema

O estado do sistema (vetor de ângulos de barras do sistema, também obtido indiretamente através do vetor das variáveis de controle), após ter sido processada uma restrição é transmitido pela tarefa *mestre* a todas as tarefas escravas no modelo da figura (5.2). Ainda, no início do processo concorrente, muitas violações são encontradas pelas tarefas escravas e as restrições correspondentes colocadas na fila, pudendo causar um congestionamento temporário. Logo, algumas restrições pegas pela tarefa *mestre* podem ter sido geradas com um estado anterior. Para evitar trabalho extra ao processo de otimização e, eventualmente aproveitar essa restrição, tem sido implementado na tarefa *mestre* um rápido teste de validade da restrição, que é feito utilizando a equação (C.4) dos apêndices.

5.2.2 Critério de Parada

Dada a natureza altamente assíncrona do algoritmo proposto, é preciso tomar cuidado especial com o critério de parada, ou seja, é preciso ter uma maneira precisa de se saber quando a solução do problema é atingida. Então, esta estratégia deve observar dois aspectos:

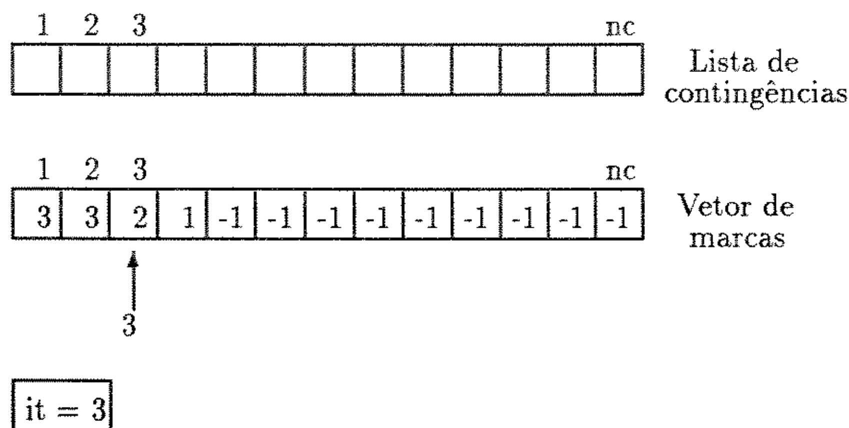

Figura 5.4: Estratégia de marcas.

(1) Que a otimalidade da solução não seja afetada.

(2) Que os processadores envolvidos na solução do problema não executem mais trabalho do que o necessário para atingir a solução final.

A perda da otimalidade pode acontecer se o aviso de parada do processo é ativado quando ainda não foram testados todos os cenários para a solução final. A segunda situação envolve a realização pelos processadores, de atividades repetidas sobre os mesmos dados, ou utilizando informações de estágios passados, pudendo afetar a eficiência do algoritmo de solução.

Para atender estes dois requisitos, foi implementada a seguinte estratégia:

Cada contingência no processo de otimização é identificada com um par (i, j), onde i é o índice de identificação da contingência dentro da lista de casos, e j é o índice do estado para o qual essa contingência já foi testada. Um escalar (it), acessável por todas as tarefas, indica qual é o *estado atual*, ou seja, o último estado gerado pela tarefa *mestre*. Os elementos j (aqui chamados de "marca") são armazenados num vetor adjunto à lista de contingências (inicialmente, todos os j são fixados em -1). Assim, por exemplo, se na figura (5.4) $it = 3$ (ou seja, o estado foi alterado três vezes como produto de adição de restrições), as contingências 1 e 2 não serão consideradas pois elas estão "marcadas" com o valor de j igual a 3. A contingência 3 está marcada com $j = 2$; se agora ela é analisada por um processador e não é detectada violação, então o seu j assume também o valor atual de it. Logo, um caso verificado não é resubmetido para análise, exceto quando exista discrepância entre os valores de j e it.

Com esta estratégia, o critério de parada decorre de maneira natural. A solução é atingida caso não existam mais casos sem marcas para serem analisados pelas subtarefas *análise*

de contingência e geração de restrições.

É claro que o acesso a uma restrição e atualização da sua marca deve ser feito com exclusividade (apenas uma subtarefa pode acessar e atualizar esse par). Estes aspectos serão desenvolvidos na próxima seção.

5.2.3 Mapeando o Problema na Máquina Real

Para que vários processos trabalharem juntos e cooperativamente em um ambiente de multiprocessamento, eles precisam de comunicação e sincronização [20] [19]. A comunicação interprocesso pode ser feita através de variáveis compartilhadas ou através de troca de mensagens. A comunicação também leva a requerimentos de sincronização; um processo é executado com alguma velocidade e gera ações ou eventos que devem ser reconhecidos por outros processos. Os mecanismos de sincronização colocam restrições na ordem desses eventos. Quando variáveis compartilhadas são utilizadas para comunicação entre tarefas, dois tipos de sincronização são utilizados: exclusão mútua e sincronização condicional. A exclusão mútua garante que um recurso seja utilizado de maneira indivisível. A sincronização condicional é requerida em situações onde um objeto compartilhado não encontra-se em um estado apropriado para execução de uma operação; qualquer tarefa que tente realizar a operação deverá ser retardada.

Nas próximas seções são descritos aspectos mais específicos da implementação da solução concorrente do problema de FPORS em uma máquina de memória comum (ou compartilhada) e em uma máquina de memória distribuída. Aspectos envolvendo as bibliotecas de comunicação e outros conceitos relacionados são tratados sumariamente.

5.3 Máquina de Memória Compartilhada

5.3.1 Computador Utilizado.

O modelo de programação dado na figura (5.2) pode ser mapeado em vários ambientes físicos diferentes. A primeira implementação do modelo foi feita em um computador *PP* (Processador Preferencial), desenvolvido no Centro de Pesquisas e Desenvolvimento da Telebrás (CPqD), que é descrito com mais detalhes em [24]. O computador *PP* é do tipo memória compartilhada, com 9 processadores conectados por um barramento comum, tendo uma unidade de memória comum ligada ao barramento. Sua arquitetura básica é mostrada na figura (5.5). Cada uma das 9 placas possui um processador iAPX-286 com memória RAM interna de 512 Kbytes e a memória comum tem uma capacidade de 128 Kbytes, acessada através de um barramento global com capacidade de escoamento de 10

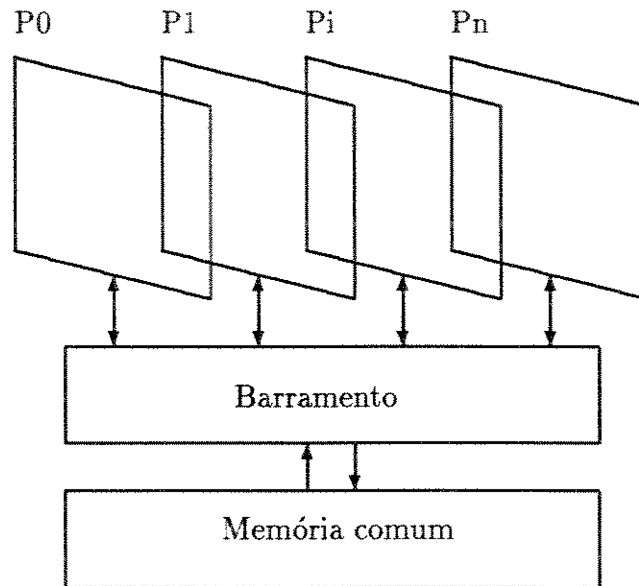

Figura 5.5: Arquitetura do computador *PP*.

Mbytes/s e com um esquema *daisy-chain* de controle de acessos concorrentes. Todas as placas processadoras podem executar programas usando tanto dados locais, armazenados na memória local, quanto dados globais, armazenados na memória compartilhada e acessíveis a todos os processadores. Note-se, portanto, que o computador *PP* apresenta características de arquitetura híbrida, ou seja memória comum mais memória distribuída.

A linguagem computacional utilizada para a aplicação foi FORTRAN, rodando sob sistema operacional MS-DOS. Para gerenciar o acesso à memória compartilhada foi necessária a implementação de uma biblioteca paralela. Essa biblioteca foi escrita em linguagem C e seu desenvolvimento é parte do trabalho descrito em [18].

5.3.2 Ferramentas de Programação Concorrente.

O mapeamento do modelo de programação descrito anteriormente em uma arquitetura paralela real, requer o desenvolvimento ou a existência prévia de certas ferramentas de programação concorrente [18]. Nesta seção duas rotinas (estruturas) básicas (*Getsub* e *Fifo*) usadas na comunicação de tarefas e subtarefas concorrentes (processos) são descritas de maneira sumária. Um elemento básico chamado de *semáforo*, utilizado na construção das duas rotinas concorrentes mencionadas anteriormente também é brevemente apresentado. As rotinas *Getsub* e *Fifo*, que são implementadas em termos de semáforos, são

usadas pela versão concorrente do fluxo de potência ótimo com restrições de segurança, para gerenciar o acesso a dados comuns (memória compartilhada).

Semáforos

Semáforos [10] são normalmente utilizados em programação concorrente para regular o acesso a recursos compartilhados (por exemplo, memória) por tarefas distintas, que são executadas em um ou mais processadores. Semáforos são basicamente utilizados em operações de *bloqueio* e *liberação* de recursos. Assim, a exclusão mútua será garantida sempre que uma tarefa ou subtarefa (ou um processo, de modo geral) bloqueia o acesso a um determinado recurso; outras tarefas só serão capazes de acessar o mesmo recurso quando este for liberado pela tarefa que o bloqueou. Semáforos são estruturas relativamente simples (baixo nível) que podem ser utilizadas diretamente pela aplicação (o fluxo de potência ótimo com restrições de segurança) ou podem fazer parte de estruturas mais complexas (alto nível), no caso presente as rotinas *Getsub* e *Fifo*. O uso de rotinas de mais alto nível tornam o acesso a dados compartilhados transparentes para a aplicação, o que resulta em programação mais simples e maior facilidade na transferência de programas entre arquiteturas diferentes.

Rotina Getsub

O estilo de programação assíncrono adotado neste trabalho facilita o balanceamento dinâmico da carga computacional dos vários processadores que constituem a máquina paralela, e assim tem um efeito benéfico sobre a eficiência da resolução concorrente do fluxo de potência ótimo com restrições de segurança. Neste modelo de programação a alocação de tarefas deve ser feita de maneira flexível e independente. Através da rotina *Getsub* uma tarefa (ou subtarefa) pode pegar um índice da lista (ou vetor) de contingências, assumindo assim a tarefa de analisar a respectiva contingência, para efeito de cálculo de índice de severidade ou geração de restrição ativa (Não se deve confundir aqui *índice* de um vetor, que é um número inteiro, com *índice de severidade*, que é um número real calculado, por exemplo, de acordo com as expressões da seção 3.2). Assim, uma subtarefa que está sendo executada em um dos processadores pode pegar um número (índice de vetor) de contingência para ser analisada simplesmente chamando a rotina Getsub (do inglês *Get subscript*). A rotina Getsub é um código concorrente que pode ser executado por qualquer tarefa escrava ou pela tarefa mestre (ou subtarefas, é claro). Ela garante exclusão mútua no acesso a dados partilhados; no caso particular da lista de contingências, isto significa que duas subtarefas distintas jamais pegarão o mesmo índice (*subscript*).

Esta rotina foi implementada com a ajuda de semáforos que bloqueiam o acesso a variáveis inteiras que representam a numeração das contingências. Os dados manipulados por

Getsub residem na memória comum, e são inicializados pela tarefa master.

Rotina Fifo

Na medida em que as tarefas escravas encontram casos de contingências que levam a sobrecargas, elas *produzirão* restrições ativas (violadas), que serão mais tarde incluídas no subproblema *mestre*. De acordo com o estilo assíncrono adotado no trabalho, estas restrições são colocadas em uma fila gerenciada pela rotina *Fifo*, que pode ser chamada tanto por *escravos* (*produtores*) como pelo *mestre* (*consumidor*). Note que em um estilo síncrono, mais convencional, para cada nova restrição ativa, ou subconjunto de restrições ativas, encontradas pelos *escravos*, o *mestre* seria interrompido para a devida inclusão dessas restrições. No procedimento assíncrono proposto neste trabalho, as restrições entram em uma fila tipo fifo, first-in-first-out, para consumo subseqüente pelo *mestre*. Os dados manipulados por Fifo residem em memória comum e são inicializados pelo *mestre*.

A rotina Fifo tem três funções associadas, que se encarregam de retirar (*Fifoget*), colocar (*Fifoput*) dados na fila, e uma terceira que é para verificar o estado da fila (*Fifostat*); esta última função verifica se há dados ou não na fila, evitando-se uma situação de espera, que é incompatível com o modelo assíncrono adotado. Aspectos mais detalhados envolvendo a implementação destas funções são encontrados em [18].

Vetor de Estado.

O vetor de estado (vetor de ângulos) do sistema deve ser transmitido tão rápido como seja possível cada vez que é gerado um novo valor dele pela tarefa *otimização*. Ele é escrito em uma região da memória comum, de onde é lido pelas demais tarefas. O acesso a essa região é restrito só quando essa variável é submetida a atualização pela tarefa *otimização*.

Marca dos Casos.

Como foi tratado anteriormente, utilizando a rotina Getsub, uma tarefa escrava pode pegar o índice da contingência que ela processará. O acesso a esse índice é exclusivo, sendo esta característica estendida também para a marca correspondente. No instante que a marca é atualizada, essa posição é bloqueada para impedir acesso de solicitações provenientes de outras tarefas.

Além da lista de contingência, vetor de estado e vetor de marcas, também fica residente na memória comum o *contador de casos já marcados*. O contador pode ser incrementado por todos os processadores; cada vez que um processador encontra um caso que não leva a violação, o contador é incrementado em 1 e verificado se atingiu o número total de contingências. Se ele atingiu esse limite, o escravo envia um sinal para o mestre, através da *fifo*. Desta forma, o processo é encerrado assim que um processador verificar que todos os casos estão já marcados.

5.3.3 Implementação no Ambiente Físico

A maneira mais simples de mapear o modelo neste tipo de "hardware", consistiria em alocar a tarefa *mestre* em um dos processadores, digamos o processador 0, e tarefas escravas nos demais processadores (até um máximo de 8); a base de dados residiria na memória comum; e as comunicações seriam feitas através do barramento. Uma outra alternativa, a que realmente foi implementada, tem uma tarefa escrava rodando no processador 0, que é ativada sempre que a tarefa *mestre* estiver ociosa. Devido às características do problema (muitos cenários pós-contingência a serem simulados), o processador 0 fica sem executar a tarefa *mestre* durante um considerável intervalo de tempo; a execução da tarefa *escravo* nesse período torna a estratégia implementada muito eficiente, tanto em "speed-up" assim como em balanço de carga. A seguir são apresentadas as estruturas dos programas *mestre* e *escravo* escritos em uma linguagem pseudo-FORTRAN.

```
program MESTRE
*Leitura de dados
*Otimiza caso base
do while(caso < nc )
  if(Fila não vazia)then
      *recebe índices de desempenho dos escravos
  else
      *calcula índice de desempenho
  endif
enddo
*Ordena lista de contingências
Do while(existam casos não marcados)
  if(Fila não vazia)then
      *Otimização com contingências
      *Remete novo ponto de operação
  elseif(Fila não vazia)then
      *Processo de análise de contingências
      if(caso causa violação)then
          *Otimização com contingências
          *Remete novo ponto de operação
      endif
  endif
enddo
*Stop
```

Figura 5.6: Estrutura do programa *mestre*.

```
program ESCRAVO
do while(caso < nc )
  *Calcula índices de desempenho
  *Coloca índice na Fila
enddo
do while(Existam casos não marcados)
  *Processo de análise de contingências
  *Coloca restrição na Fila
enddo
```

Figura 5.7: Estrutura do programa *escravo.*

Os resultados obtidos com a implementação na máquina de memória comum são apresentados e discutidos no próximo capítulo.

A seguir, para mostrar o desacoplamento entre o modelo de programação da figura (5.2) e os ambientes físicos onde eles são implementados, é apresentada a implementação em uma arquitetura baseada em troca de mensagem (computador hipercubo).

5.4 Máquina de Memória Distribuída

Os sistemas com comunicação baseados em troca de mensagens (fracamente acoplados) não possuem conflitos nem colisões na manipulação de memória, como ocorre nos sistema de memória compartilhada (fortemente acoplados). Cada processador possue uma memória local suficientemente grande como para armazenar a maioria das instruções e dados.

Devido à mudança radical na filosofia de comunicação entre processadores, a implementação na máquina de memória distribuída deverá utilizar a biblioteca de comunicação e sua funções, disponíveis na nova arquitetura.

5.4.1 O Computador de Memória Distribuída.

O computador de memória distribuída utilizado é um computador *NCUBE-2* [26]. Sua arquitetura, considerando 8 nós, é mostrada na figura (5.8). Ele é formado por três partes

básicas: nós, canais de comunicação e processadores de entrada/saída.

Nós (Elementos de processamento)

A base do computador *NCUBE* é uma rede de unidades de processamento independentes (CPU) de 64 bits, cada uma tendo integrada uma unidade de ponto flutuante, memória local de 4 a 64 Mbyte e "hardware" de comunicação. Em cada nó roda uma cópia completa do sistema operacional *nCX*. Quando é executada uma aplicação, em cada nó roda um "elemento de programa" ou "programa local". O programa paralelo compreende todos os elementos de programa rodando em todos os nós alocados pelo programa. O computador *NCUBE-2* pode executar vários programas paralelos simultaneamente, com cada programa alocado a uma diferente coleção de nós. Cada nó é um computador seqüencial operando a 2.5 Mflops. O número de nós é "escalável", ou seja, pode aumentar-se a memória e o número de nós, que no caso da máquina disponível pode chegar até 1024 nós. A atual configuração tem 64 nós, ou seja, dimensão máxima de 6, e 4 Mbytes de memória local em cada nó.

Canais de Comunicação

Cada nó está conectado aos outros através de um "hardware" de roteamento de mensagens. Cada processador tem 14 canais bidirecionais DMA (acesso direto a memória). Estes canais suportam transferências de dados e mensagens de controle entre dois nós quaisquer na razão de até 2.2 Mbytes/s. Apesar que só existem canais entre vizinhos adjacentes na rede, a velocidade do "hardware" de roteamento permite uma comunicação entre nós distantes tão rápida como entre vizinhos adjacentes.

Processadores de entrada/saída

Além dos canais de comunicação entre nós descritos anteriormente, cada nó tem também um canal separado dedicado à transferência de/para os processadores de entrada/saída. Estes canais operam à mesma razão dos canais de comunicação inter-nós. Os processadores de entrada/saída têm a mesma CPU e configuração de memória dos nós, porém com interfaces adicionais de software e "hardware" para discos, monitores e outros periféricos. Assim como os nós, os processadores de entrada/saída também são escaláveis.
Na configuração atual, o computador *NCUBE* está ligado a um "host" constituído por uma estação de trabalho SPARC 1, modelo 330, e através dela a uma rede local.

Linguagem

O ambiente do *NCUBE* suporta linguagens de programação standard, tais como o ANSI C, FORTRAN 77 com extensões VAX e C++. O ambiente de software paralelo inclui interfaces para sistema UNIX V versão 4, bibliotecas UNIX standard e ferramentas de

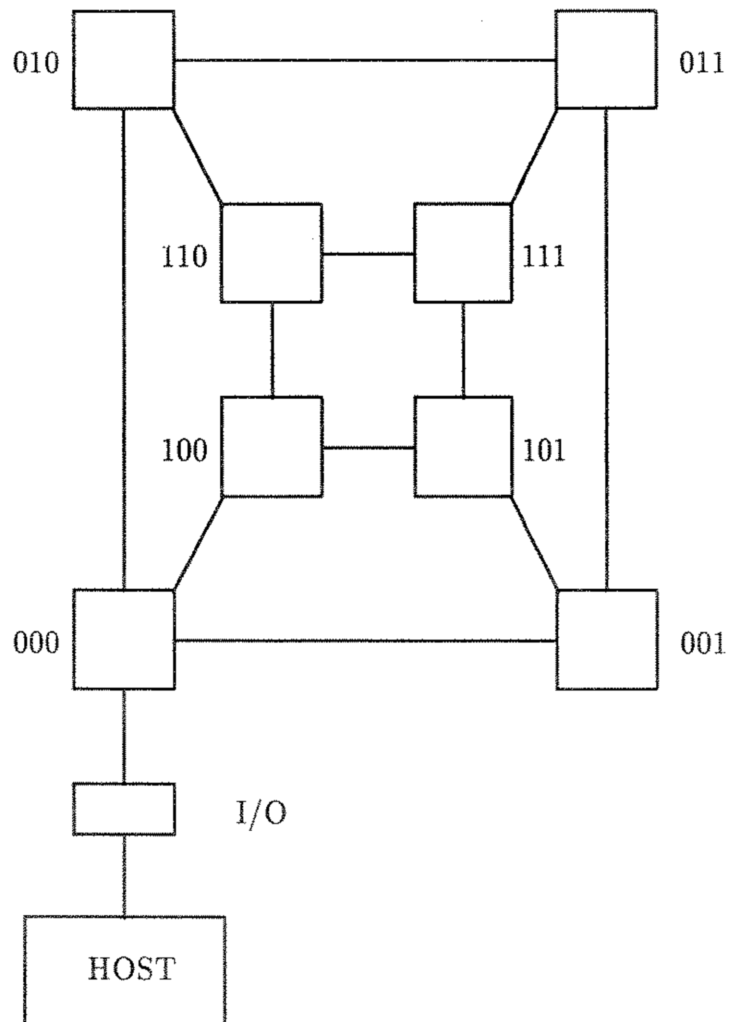

Figura 5.8: Arquitetura do NCUBE com dimensão 3.

desenvolvimento baseadas em ferramentas UNIX standard tais como *cc* e *ld.*

Portando a Aplicação

Nas próximas seções são descritas e discutidas as alterações que são necessárias para o algoritmo migrar para esta nova arquitetura.

5.4.2 Obtenção da Contingência a ser Analisada.

Na arquitetura disponível não existe ainda a opção de criação de uma memória comum virtual [25]. Porém, ainda que existisse esta possibilidade, a implementação por essa via da obtenção do caso (índice de um caso na lista de contingências) se torna muito ineficiente, pois cada vez teria-se que acessar o sistema de comunicação. Uma solução que substitui a rotina Getsub é obtida utilizando a função *WHOAMI*, da biblioteca paralela do computador. A função tem o formato:

WHOAMI(mynode, mypid, myhost,dim)

onde:
mynode : número do nó no cubo alocado.
mypid : os 16 bits menos significativos contém o número relativo do nó; os 16 bits mais significativos fornecem o número de identifição do processo corrente.
myhost : retorna o número de identificação do "host".
dim : retorna a dimensão do cubo alocado.

Entre outras informações, *WHOAMI* retorna o número do nó e a dimensão do cubo que foi alocado. Todos os processadores têm disponíveis a lista de casos e suas marcas. A seleção do caso que vai ser analisado por uma tarefa que está rodando em um nó determinado é parametrizada com o número de identificação deste último. Assim, por exemplo, o nó *mynode* pegará os casos (índices da lista):

mynode
mynode + nodes
*mynode +2*nodes*
*mynode +3*nodes*
....
....

onde *nodes* é o número de nós do cubo alocado. Supõe-se que o número de contingências é suficientemente maior do que *nodes.*
Esta estratégia muito simples preserva a idéia de "acesso exclusivo" a um índice de contingência, utilizada na rotina Getsub da implementação em memória comum.

5.4.3 Envio de Restrições.

Na arquitetura de memória comum, o envio de restrições de escravo para mestre é feito mediante um buffer tipo "fifo". Uma rotina do mesmo nome gerencia o acesso a essa fila utilizando as funções Fifoget, Fifoput e Fifostat. A estrutura básica desta etapa é preservada quase totalmente, apenas tendo-se que substituir as funções de comunicação pelas disponíveis na biblioteca paralela do *NCUBE*. Logo, para *enviar* informação, utiliza-se a função *NWRITE*. A função *NWRITE* coloca uma mensagem a ser enviada em uma fila de transmissão e tem o seguinte formato:

NWRITE(buffer, nbytes, dest, type, flag)

Onde:

buffer : endereço de onde a mensagem deve ser lida para ser transmitida.

nbytes : número de bytes a ser transmitidos.

dest : numero do processo (16 bits menos significativos) e do processador (16 bits mais significativos) de destino. Se *dest* é fixado em -1, todos os processadores receberão a mensagem. Se *dest* é fixado em 0, apenas o processador 0 receberá a mensagem.

type : número de identificação da mensagem a ser enviada.

flag : sem uso.

Especificamente, se a tarefa *mestre* está alocada no processador 0, e a identificação para o tipo de mensagem (restrição de contingência) é 10, então uma tarefa *escrava* deverá invocar esta função assim:

NWRITE(restrição, nbytes, 0, 10, flag)

Para *receber* uma mensagem utiliza-se a função *NREAD*. A função *NREAD* pega uma determinada mensagem da fila de transmissão e a coloca em um endereço específico da memória local do processador. O formato da função *NREAD* é o seguinte:

NREAD(buffer, nbytes, source, type, flag)

Onde:

buffer : endereço onde a mensagem será colocada.

nbytes : número de bytes a ser lidos.

source : numero do processo (16 bits menos significativos) e do processador (16 bits mais significativos) que enviou a mensagem. Se *source* é fixado em -1, então qualquer

mensagem de um tipo especificado destinado para o processador corrente será lido.

type : número de identificação da mensagem a ser lida.

flag : sem uso.

No caso específico do *mestre* receber restrições, os argumentos de *NREAD* devem ser fixados para "receber um tipo de mensagem (restrição de contingência) de qualquer processador". Neste caso:

NREAD(restrição, nbytes, -1, 10, flag)

Função *NTEST*

A função *NREAD* tem um caráter bloqueante; isto é, se uma mensagem esperada ainda não está na fila, ele fica esperando, o que é pernicioso para a filosofia assíncrona assumida neste trabalho. A função *NTEST* da biblioteca paralela do *NCUBE* realiza a mesma função da Fifostat (rotina Fifo). Ela verifica se há na fila uma mensagem do tipo especificado no seu argumento . Se há, ele retorna um número não negativo; caso contrário retorna um valor negativo. Esta função é muito rápida e é utilizada aqui junto com *NREAD* na leitura de restrições. O formato é:

NTEST(source, type)

Onde:

source : identificação do processador que enviou a mensagem. Se *source* =-1, então serão aceitos mensagens de qualquer emissor.

type : número de identificação da mensagem que está sendo esperada. Se *type* =-1, então qualquer tipo de mensagem dentro de um certo intervalo será aceita.

5.4.4 Estado do Sistema

O estado atual do sistema, que na arquitetura de memória compartilhada é remetido para as tarefas escravas através da memória comum, aqui é transmitido utilizando a função *NBROADCAST*. A função *NBROADCAST* envia uma mensagem a todos os nós do cubo alocado ou de um subcubo contido nele. A mensagem tem um tamanho de *length* bytes,

sendo *buf* seu endereço de origem no nó fonte *bnode*. O formato é:

NBROADCAST(buf, length, bnode, type, mask)

Onde :

buf : Este argumento especifica o endereço onde está a mensagem a ser enviada.

length : Especifica o tamanho em bytes da mensagem.

bnode : Especifica o número do nó que transmitirá a mensagem.

type : Este argumento indica o tipo da mensagem a ser usada em todas as comunicações via *NBROADCAST*.

mask : O valor deste argumento determina qual porção do cubo receberá a mensagem. Se o valor é -1, todos os nós do cubo receberão a mensagem.

Logo no caso específico em que o nó 0 envia o estado para todos os outros nós do cubo alocado, a chamada a esta função fica assim:

NBROADCAST(estado, nbytes, 0, 20, -1)

onde 20 é, por exemplo, o tipo da mensagem e *nbytes* é a dimensão em bytes do vetor *estado*.

A leitura de estado pelas tarefas escravas é feita utilizando a função *NREAD*. Como esta função é bloqueante, sua utilização é acompanhada da função *NTEST*. Assim, *NREAD* é ativada só se houver um novo estado disponível; caso contrário, continua sendo utilizado o estado antigo.

5.4.5 Critério de Parada

No algoritmo correspondente ao computador de memória comum, cada processador ajuda na contabilização dos casos já marcados, pois o contador fica na memória comum. Logo, quando todos os casos já estão marcados, o escravo que primeiro detectar isto envia um sinal ao mestre através da *fifo*. O processador mestre pode diretamente verificar o contador, quando está executando a tarefa escrava.

No algoritmo do computador de memória distribuída, cada processador é proprietário de um segmento da lista de contingências. Logo, quando todos os casos estiverem marcados em um determinado processador, é remetido um sinal ao mestre, informando a convergência do problema para essa sub-lista. O mestre encerra o processo quando houver recebido essa informação de todos os processadores.

5.4.6 Comentários

A utilização de modelos de programação neste trabalho tem sido motivada pela idéia de "facilitar" a migração da aplicação de uma arquitetura para outra. Aqui os programas têm sido portados desde uma máquina de memória compartilhada de 9 nós para um computador *NCUBE*. Alterações decorrem de: a) Diferente filosofia de comunicação entre as arquiteturas e portanto diferente biblioteca e funções disponíveis. Este aspecto tem seu impacto na substituição da biblioteca de comunicação incluindo as chamadas às funções correspondentes nos programas da aplicação. b) A mudança de ambiente torna mais adequada a substituição da função Getsub por uma estratégia de escolha do índice da contingência parametrizada no número de identificação do processador. A diferença é principalmente filosófica. No caso de memória compartilhada, a função Getsub garante o acesso exclusivo a um índice de contingência, que pode ser qualquer dentro da lista toda. Aqui, a parametrização utilizando a função *WHOAMI* também garante a exclusividade do índice, porém cada processador é "proprietário" de um segmento da lista de casos. Esta implementação leva a uma distribuição balanceada de casos entre os processadores (proporção de casos que levam a violação e casos que não levam).

Da mesma forma que na implementação concorrente da arquitetura de memória comum, aqui o processador mestre pode executar a tarefa escrava quando está ocioso, ou seja, quando não encontra restrições na fila para a tarefa mestre processar.

As estruturas dos programas instalados no *NCUBE* continuam sendo as mesmas das figuras (5.6) e (5.7). Logo, a solução concorrente para o problema de fluxo de potência ótimo com restrições de segurança proposta aqui, apresenta um considerável grau de portabilidade entre as arquiteturas utilizadas neste trabalho.

No apêndice D são incluídas versões simplificadas dos programas Mestre e Escravo que foram implementados nos computadores *PP* e *NCUBE*. No próximo capítulo são apresentados, analisados e discutidos os resultados práticos obtidos utilizando as duas arquiteturas computacionais disponíveis.

Capítulo 6

Resultados de Testes

6.1 Introdução

Neste capítulo são apresentados e discutidos os resultados obtidos com a proposta de solução para o problema de fluxo de potência ótimo com restrições de segurança.

Para avaliação do desempenho do algoritmo são utilizados três sistemas de energia elétrica: o sistema IEEE118 e os sistemas brasileiros de 725 e 1663 barras.

Para avaliar a portabilidade da proposta são utilizadas duas arquiteturas computacionais: um computador de memória compartilhada de 9 nós e um computador de memória distribuída de 64 nós com arquitetura hipercubo.

Função objetivo

Nos testes implementados, a função objetivo a ser otimizada é *mínimo desvio de um ponto de operação especificado de potência ativa.* As variáveis de controle são as gerações de potência ativa. As listas de contingências estão formadas por contingências simples de ramo. Previamente foram excluídos casos que levam a ilhamento no sistema.

As características básicas dos sistemas de energia elétrica utilizados aqui são mostradas na tabela 6.1. Os sistemas A e B são redes que representam parte do sistema sudeste brasileiro operando em carga leve. A tabela (6.2) apresenta informações relacionadas com a solução ótima para cada sistema. Os megawatts realocados correspondem à soma dos desvios de potência ativa com relação ao ponto inicial, necessários para eliminar as infactibilidades. Na função objetivo -mínimo desvio quadrático de um ponto de operação- cada gerador é ponderado com o inverso de sua capacidade.
Estes três sistemas, além das dimensões, apresentam características diferentes (nível de carga, distribuição dos geradores, topologia), o que os torna interessantes para serem

Sistema	Número de barras	Número de ramos	geradores controláveis	lista de contingências
IEEE118	118	179	51	170
Sist. A	725	1212	76	900
Sist. B	1663	2349	99	1555

Tabela 6.1: Sistemas de testes

Sistema	Restrições de contingência ativas	Mw realocados	Função objetivo (mínimo desvio)
IEEE118	4	566	0.264
Sistema A	6	3290	0.421
Sistema B	7	1230	0.795

Tabela 6.2: Resultados do processo de otimização para os sistemas-testes. Na solução inicial, duas últimas colunas são zero.

utilizados na avaliação do algoritmo.

Foi elaborado um conjunto de testes objetivando avaliação do desempenho do algoritmo proposto neste trabalho, focalizando os seguintes aspectos:

(i) Impacto da ordenação inicial da lista de contingências no tempo de processamento total.

(ii) O efeito do número de cenários na eficiência global, assim como no "speed-up" do algoritmo concorrente.

Impacto da ordenação inicial

A ordenação de contingências é crítica para o modelo assíncrono proposto neste trabalho; quanto antes o programa *mestre* receba informações relativas às contingências mais críticas (contingências que realmente causarão mudanças no ponto de operação do caso base), mais rápido progredirá para a solução. Por outro lado, o processo de análise de contingências é realizado com o ponto de operação mais atual, o qual vai mudando na medida que novas restrições são incorporadas. Conseqüentemente, a análise de algumas contingências deverá ser processada mais do que uma vez. Portanto, conhecendo-se antecipadamente as contingências mais críticas, e incluíndo essas contingências o mais rápido possível no

processo de resolução, obter-se-á uma redução do tempo global de cálculo.

Logo, duas situações são simuladas:

(a) Supõe-se que a lista foi ordenada previamente com um determinado critério e se encontra pronta na base de dados para ser utilizada.

(b) O processo de ordenação da lista faz parte do processo paralelo.

A situação (a) decorre do fato que nos centros de controle normalmente a lista de contingências é armazenada na base de dados ordenada de acordo com um critério de severidade (experiência prévia do operador, índices de desempenho, probabilidade de ocorrência, etc).

Para testar o impacto da qualidade de diferentes tipos de ordenação no desempenho do algoritmo, são considerados três tipos diferentes de critérios de ordenação da lista de contingências. Deve-se enfatizar que não existe a intenção de defender ou recomendar a adoção de algum desses critérios, os quais têm sido selecionados com o único fim de gerar ordenações diferentes.

Critério 1: Este critério consiste em assumir uma lista sem ordenação, ou seja com ordem aleatória.

Critério 2: A ordenação é feita utilizando o gradiente do índice de desempenho (equação (3.3)).

Critério 3: Ordenação baseada no índice de sobrecarga dado pela equação (3.4)).

Teoricamente, estes índices fornecem listas de qualidades crescentes (isto é, o critério 3 fornece uma lista de melhor qualidade do que o critério 2, etc.) e um esforço computacional para sua determinação também crescente. Como foi dito no capítulo anterior, a ordenação através do critério 2 é muito rápida, sendo então realizada pela tarefa mestre. A ordenação através do índice de sobrecarga (critério 3) envolve um considerável esforço computacional, pois, para cada simulação de contingência, precisa realizar cálculos similares aos do processo de análise de contingência. Este processo, então, é paralelizado.

Os tempos de processo apresentados excluem o processo de leitura de dados, fluxo de carga inicial e relatórios. Todos os tempos têm sido normalizados pelo tempo da versão seqüencial com o critério 1 (ordem aleatória). Os ganhos ("speed-up") são dados com relação ao melhor tempo seqüencial (um processador) do teste correspondente.

Conjunto Crítico

O algoritmo trabalha com um conjunto de contingências consideradas "suspeitas" de ficarem ativas na base do processo de otimização. Na atual implementação, este conjunto é identificado baseando-se no índice de ordenação. Os casos com um valor de índice maior

do que uma tolerância prática (por exemplo 10^{-3} p.u.) são incluídos dentro deste conjunto crítico. Conseqüentemente, a qualidade e o tamanho do conjunto crítico irão depender da eficiência do índice escolhido.

6.2 Resultados Obtidos no Computador *PP*

Os resultados obtidos no computador de memória comum (*PP*) de nove processadores (processadores AT-286 sem coprocessador) são apresentados a seguir. Em um processador é carregada a tarefa *mestre* e a tarefa *escrava*; esta última é ativada quando a tarefa *mestre* não tiver restrições para processar. Nos oito processadores restantes são carregadas as tarefas *escravas*.

Por problemas de capacidade de memória no computador *PP*, apenas são apresentados resultados com os sistemas IEEE118 e o sistema A.

Nas tabelas (6.3) e (6.4) são apresentados os resultados obtidos com o sistema IEEE118 para 170 contingências. Na tabela (6.3), o processo de ordenação (obtido com três critérios) faz parte do processo paralelo. Na tabela (6.4), assume-se que a lista, obtida com os mesmos três critérios de ordenação, já está disponível em um banco de dados. O tempo de normalização é de $t_b = 402$ segundos e corresponde ao tempo seqüencial sem ordenação (critério 1).

Analogamente, nas tabelas (6.5) e (6.6) são apresentados os resultados com o sistema brasileiro de 725 barras, 1212 ramos, para uma lista de 900 contingências. A tabela (6.5) contém tempos e ganhos incluíndo o tempo de ordenação dos critérios adotados. A tabela (6.6) contém os resultados excluindo o processo de ordenação. Para as duas tabelas, o tempo de normalização é $t_b = 13388$ segundos, que corresponde ao tempo seqüencial sem ordenação. A ordem numérica dos tempos é elevada, devido às dimensões do problema e à baixa capacidade de processamento dos nós.

Em cada tabela (6.3) - (6.6) os ganhos ou "speed-up" são dados com relação ao melhor tempo seqüencial.

Comentários

Na tabela (6.3) o melhor desempenho foi obtido com o critério 2. O critério 3 leva a um lista de melhor qualidade, porém não suficiente para justificar o esforço computacional adicional. Observe-se que os dois critérios levam a dimensões similares para o conjunto crítico (para o critério 2, 118 casos e para critério 3, 116 casos), enquanto que as restrições de contingência que realmente ficam na base são 4.

Quando a lista já está disponível na base de dados, o melhor desempenho é com a lista

ordenada com o critério 3, ou seja, com a lista de melhor qualidade. Aqui, é denotado um comportamento superlinear com dois processadores (eficiência de 106.5 %), enquanto que com nove processadores a eficiência atingida é de 67.2 % (com uma média de 18 contingências por processador).

Na tabela (6.5) (sistema de 725 barras) o melhor tempo seqüencial corresponde à versão com critério 3 (em geral, a melhor versão seqüencial nem sempre leva à melhor versão paralela). Aqui, o critério 2 gera um conjunto de 500 casos enquanto que o critério 3, 20 casos (em um universo de 900 contingências). O critério 3 consegue capturar eficientemente as contingências críticas e seu impacto é observado na versão seqüencial. Porém, ainda o peso do processo de ordenação é muito elevado, comparado com uma ordenação de menor qualidade (mas muito mais barata).

Se a lista é considerada ordenada previamente, o impacto da qualidade da ordenação fica evidente. A versão com a ordenação pelo critério 3 é claramente mais eficiente, apresentando ainda um desempenho superlinear (este aspecto será discutido mais adiante). Devido ao elevado número de cenários a tratar, os processadores ficam com bom balanço de carga (em média 100 contingências por processador).

Sistema de 118 barras - (*PP*)						
Número de nós	Critério 1		Critério 2		Critério 3	
	Tempo	Ganho	Tempo	Ganho	Tempo	Ganho
1	100.0	0.73	72.63*	1.00	92.54	0.79
2	53.16	1.37	37.54*	1.93	43.96	1.65
3	37.31	1.95	26.02*	2.79	31.34	2.32
4	29.10	2.50	20.45*	3.55	25.52	2.88
5	24.04	3.02	17.69*	4.11	21.60	3.36
6	20.66	3.52	14.70*	4.94	18.29	3.97
7	18.45	3.94	13.36*	5.43	17.14	4.24
8	16.56	4.39	12.94*	5.60	14.89	4.88
9	15.06	4.82	12.55*	5.79	14.32	5.07

Tabela 6.3: Tempos de execução normalizados (%) e ganhos como uma função do número de processadores para o sistema IEEE118 no computador *PP*. São considerados 3 tipos diferentes de ordenação da lista de contingências (170 casos); $t_b = 402$ segundos; os ganhos são relativos ao melhor tempo seqüencial ($t = 72.63$ %).

Sistema de 118 barras - (*PP*)						
Número	Critério 1		Critério 2		Critério 3	
de nós	Tempo	Ganho	Tempo	Ganho	Tempo	Ganho
1	100.0	0.68	72.13	0.94	68.16*	1.00
2	53.16	1.28	37.01	1.84	31.93*	2.13
3	37.31	1.82	25.50	2.67	22.91*	2.97
4	29.10	2.34	19.92	3.42	18.98*	3.59
5	24.04	2.83	17.16	3.97	16.50*	4.13
6	20.66	3.29	14.18	4.80	13.95*	4.88
7	18.45	3.70	12.84	5.31	13.42*	5.08
8	16.56	4.11	12.44	5.48	11.74*	5.81
9	15.06	4.52	12.03	5.67	11.26*	6.05

Tabela 6.4: Tempos de execução normalizados (%) e ganhos como uma função do número de processadores para o sistema de 118 barras no computador *PP*. A ordenação não faz parte do processo paralelo; a lista é considerada previamente ordenada por três critérios e já disponível na base de dados (170 casos); t_b = 402 segundos; os ganhos são relativos ao melhor tempo seqüencial (t = 68.16 %).

Sistema de 725 barras - (*PP*)						
Número	Critério 1		Critério 2		Critério 3	
de nós	Tempo	Ganho	Tempo	Ganho	Tempo	Ganho
1	100.0	0.70	81.60	.86	70.40*	1.00
2	59.30	1.19	32.90*	2.14	35.40	1.99
3	39.70	1.77	22.10*	3.18	23.70	2.97
4	29.90	2.35	16.70*	4.20	17.90	3.93
5	23.90	2.95	13.40*	5.25	14.50	4.85
6	20.00	3.52	11.30*	6.23	12.14	5.80
7	17.20	4.09	9.70*	7.25	10.60	6.64
8	15.10	4.66	8.60*	8.18	9.30	7.57
9	13.40	5.25	7.60*	9.26	8.40	8.38

Tabela 6.5: Tempos de execução normalizados (%) e ganhos como uma função do número de processadores para o sistema de 725 barras no computador *PP*. São considerados 3 tipos diferentes de ordenação da lista de contingências (900 casos); t_b = 13388 segundos; os ganhos são relativos ao melhor tempo seqüencial (t = 70.40 %).

Sistema de 725 barras - (*PP*)						
Número de nós	Critério 1		Critério 2		Critério 3	
	Tempo	Ganho	Tempo	Ganho	Tempo	Ganho
1	100.0	0.57	81.60	0.70	56.80*	1.00
2	59.30	0.96	32.80	1.73	25.00*	2.27
3	39.70	1.43	22.00	2.58	16.70*	3.40
4	29.90	1.90	16.60	3.42	12.60*	4.50
5	23.90	2.37	13.35	4.25	10.30*	5.51
6	20.00	2.84	11.18	5.08	8.70*	6.53
7	17.20	3.30	9.63	5.90	7.60*	7.47
8	15.10	3.76	8.48	6.70	6.70*	8.47
9	13.40	4.24	7.56	7.51	6.10 *	9.31

Tabela 6.6: Tempos de execução normalizados (%) e ganhos como uma função do número de processadores para o sistema de 725 barras no computador *PP*. A ordenação não faz parte do processo paralelo; a lista é considerada previamente ordenada por três critérios e já disponível na base de dados (900 casos); $t_b = 13388$ segundos; os ganhos são relativos ao melhor tempo seqüencial ($t = 56.80$ %)

Eficiências (%) - (*Computador PP*)		
Processadores	IEEE118	Sistema A
1	100.0	100.0
2	106.5	113.5
3	99.0	113.3
4	89.7	112.5
5	82.6	110.2
6	81.3	108.8
7	72.6	106.7
8	72.6	105.8
9	67.2	103.4

Tabela 6.7: Eficiências para os casos correspondentes ao critério 3 para os sistemas IEEE118 e de 725 barras; ordenação excluída do processo paralelo.

A tabela (6.7) mostra as eficiências para os sistemas IEEE118 e 725 barras, considerando o critério 3 nas tabelas (6.4) e (6.6). É notório que, quando aumenta o número de cenários a tratar, a eficiência aumenta. A figura (6.1) mostra, para o sistema de 725 barras, os ganhos para três diferentes dimensões da lista de contingências (50,100 e 900). Para este, as três listas contêm o conjunto das restrições que ficam ativas na solução ótima, isto é, a solução é a mesma para os três casos. Desta forma, fica mais fácil observar o efeito do número de cenários no ganho. Quando o conjunto é pequeno (50-100 contingências), para 2 e 3 processadores ainda é observado um desempenho superlinear, porém o ganho decresce rápidamente na medida que mais processadores participam do processo de resolução. Com 9 processadores e uma lista de 50 contingências, o ganho fica em torno de 4.5, com uma distribuição de casos por processador entre 5 e 6. Com a lista de 100 contigências, o ganho com 9 processadores aumenta para 5.6 e a distribuição de casos por processador fica em torno de 11. Já para a lista de 900 casos, o desempenho do algoritmo é muito bom, com ganhos superlineares até com 9 processadores.

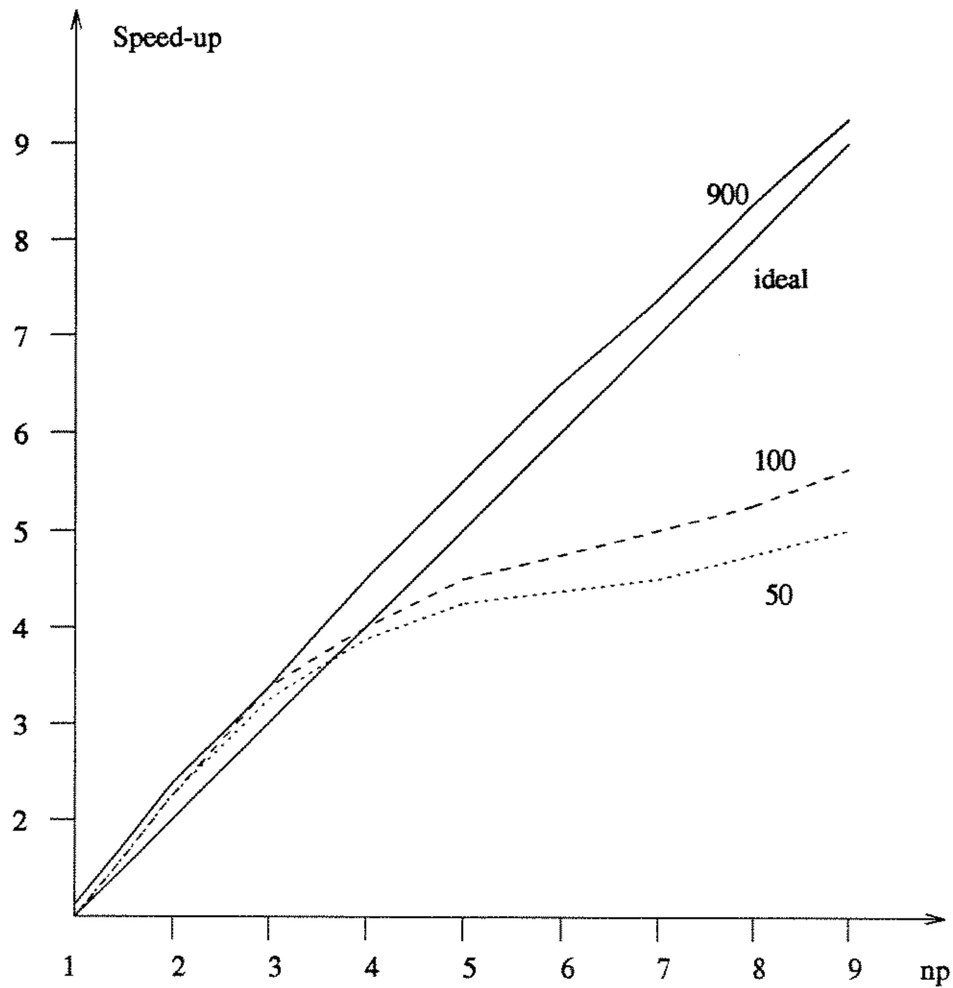

Figura 6.1: Sistema de 725 barras: ganhos para três diferentes dimensões da lista de contingências (50,100 e 900), considerando o critério 3 (ordenação excluída); *np*: número de processadores.

Comentários sobre superlinearidade

O algoritmo proposto pode apresentar desempenho superlinear, pois o modelo assíncrono permite dispor das informações mais recentes [9]; além disso, ao se dispor em paralelo da análise de vários cenários, o processo de otimização pode receber informação que leve mais rápido ao ótimo do que no caso seqüencial [19].O desempenho superlinear é uma caractéristica do algoritmo assíncrono. A seguir apresenta-se um exemplo que ilustra um caso que pode levar a desempenho superlinear.

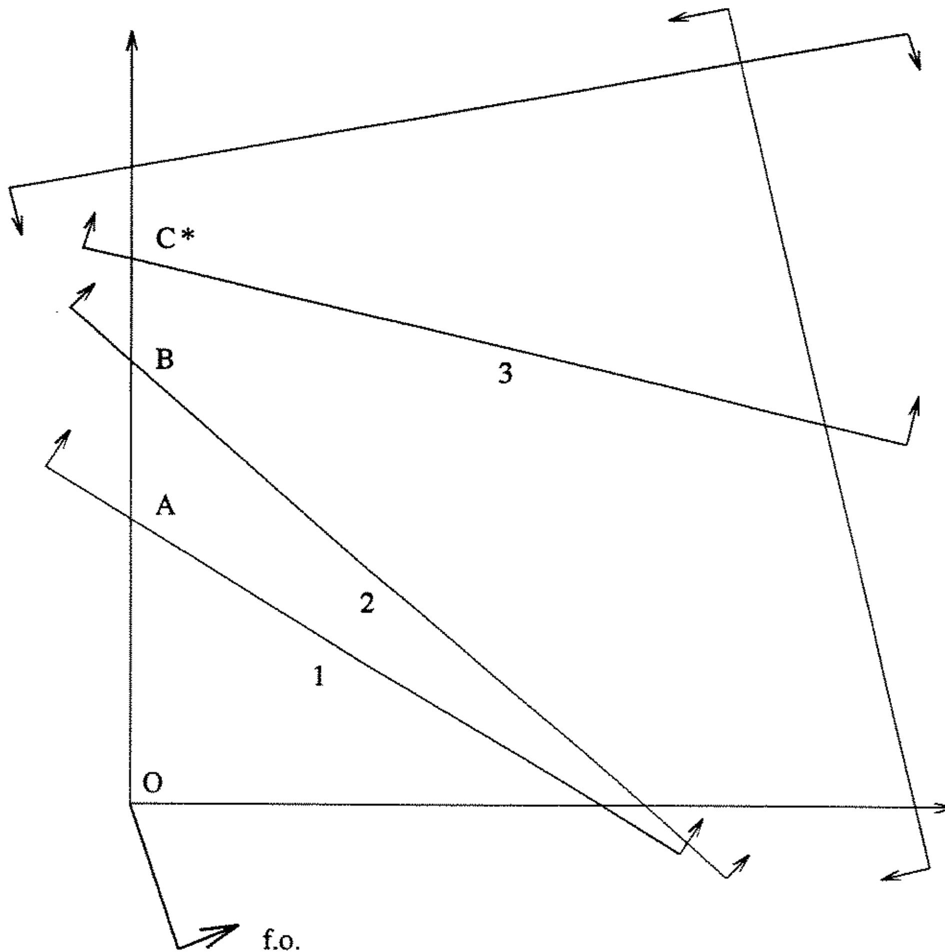

Figura 6.2: Exemplo que pode levar a superlinearidade: O ponto inicial (ótimo relaxado) está em O. A solução ótima é o ponto C.

Suponha-se uma lista de 5 restrições, onde as três primeiras causam violação no caso

base inicial, porém, na solução final apenas fica ativa a restrição 3 (ver figura (6.2)). Uma versão seqüencial irá passo a passo de O para C, passando por A e B. Em uma configuração de três processadores, a tarefa de otimização pode pegar de imediato a restrição 3 e em um passo atingir o ótimo.

6.3 Resultados Obtidos no Computador *NCUBE*

No computador *NCUBE* não existem as restrições de memória presentes no *PP*, o que permite realizar testes com sistemas de grande porte. São aqui apresentados resultados com os sistemas brasileiros de 725 e 1663 barras. Aqui, a diferença do *PP*, onde a comunicação é quase instantânea, existe uma penalidade na troca de mensagens, refletindo-se de alguma maneira na eficiência. Por outro lado, a capacidade de processamento é muito maior, o que tornam interessantes os resultados apresentados a seguir.

Nas tabelas (6.8) e (6.9) são resumidos os resultados para o sistema de 725 barras, 1212 ramos, com uma lista de 900 contingências com os três critérios de ordenação adotados. A ordem dos tempos de processamento caíram drasticamente; o pior tempo, que corresponde à base de normalização (versão seqüencial com lista em ordem aleatória) é de 152.48 segundos. A tabela (6.8) apresenta os resultados considerando os processos de ordenação, enquanto que a tabelas (6.9) considera a lista de casos já previamente ordenada segundo os três critérios e disponível em uma base de dados. Os ganhos ou "speed-up" são dados com relação ao melhor tempo seqüencial, que corresponde ao obtido com o critério 3 ($t = 67.46$ % para a tabela (6.8) e $t = 44.80$ % para tabela (6.9)).

Analogamente, nas tabelas (6.10) e (6.11) são apresentados os resultados obtidos no computador *NCUBE* para o sistema brasileiro de 1663 barras, 2349 ramos e uma lista de 1555 contingências. Aqui também os tempos são normalizados pelo tempo seqüencial obtido com ordem aleatória (t_b = 467.27 segundos). Na tabela (6.10), onde são resumidos os resultados considerando a ordenação dentro do processo paralelo, os ganhos são com referência ao melhor tempo seqüencial ($t = 70.35$ %), obtido com ordenação pelo critério 2. Na tabela (6.11), onde a ordenação não faz parte do processo paralelo, o melhor tempo seqüencial é $t = 63.6$ %, obtido com o critério 3.

Comentários

No sistema A (725 barras), o critério 3 captura adequadamente as contingências críticas, selecionando um número de 20. O critério 2 seleciona um número de candidatas muito grande (500). Aqui estão presentes três elementos de discussão: capacidade de processamento, qualidade da lista, e relação nc/np (número de contingências v-s número de processadores). Na tabela (6.8), com poucos processadores (inclusive, o caso seqüencial) o melhor desempenho é obtido com o critério 3. Ou seja, devido ao aumento da capaci-

dade de processamento em relação ao *PP*, o esforço computacional adicional para obter uma lista de boa qualidade (notar que *nem* sempre este esforço computacional leva a uma lista de qualidade proporcional) é compensado com um processamento mais eficiente das contingências que realmente são importantes.

Sistema de 725 barras - (*NCUBE*)							
Dimensão do cubo	Número de nós	Critério 1		Critério 2		Critério 3	
		Tempo	Ganho	Tempo	Ganho	Tempo	Ganho
0	1	100.0	0.68	71.03	0.95	67.46*	1.00
1	2	50.67	1.33	38.15	1.77	35.05*	1.92
2	4	25.34	2.66	19.58	3.44	17.52*	3.85
3	8	13.04	5.17	10.13	6.65	9.52*	7.09
4	16	6.90	9.78	5.45	12.37	5.28*	12.76
5	32	3.78	17.86	3.03*	22.24	3.16	21.35
6	64	2.28	29.60	1.88*	35.85	2.14	31.54

Tabela 6.8: Tempos de execução normalizados (%) e ganhos como uma função do número de processadores para o sistema de 725 barras no computador *NCUBE*. São considerados 3 tipos diferentes de ordenação da lista de contingências (900 casos); $t_b = 152.48$ s; os ganhos são relativos ao melhor tempo seqüencial ($t = 67.46$ %).

Sistema de 725 barras - (*NCUBE*)							
Dimensão do cubo	Número de nós	Critério 1		Critério 2		Critério 3	
		Tempo	Ganho	Tempo	Ganho	Tempo	Ganho
0	1	100.0	0.448	70.86	0.63	44.80*	1.00
1	2	50.67	0.885	38.00	1.18	22.46*	1.99
2	4	25.34	1.77	19.43	2.31	11.54*	3.88
3	8	13.04	3.45	9.98	4.49	6.38*	7.03
4	16	6.90	6.56	5.31	8.44	3.66*	12.25
5	32	3.78	12.00	2.88	15.55	2.25*	19.93
6	64	2.28	20.00	1.73	25.87	1.65*	27.22

Tabela 6.9: Tempos de execução normalizados (%) e ganhos como uma função do número de processadores para o sistema de 725 barras no computador *NCUBE*. A ordenação não faz parte do processo paralelo; a lista é considerada previamente ordenada por três critérios e já disponível na base de dados (900 casos); $t_b = 152.48$ s; os ganhos são relativos ao melhor tempo seqüencial ($t = 44.80$ %).

Sistema de 1663 barras - (*NCUBE*)							
Dimensão do cubo	Número de nós	Critério 1		Critério 2		Critério 3	
		Tempo	Ganho	Tempo	Ganho	Tempo	Ganho
0	1	100.0	0.70	70.35*	1.00	78.51	0.90
1	2	50.38	1.39	35.84*	1.96	39.95	1.76
2	4	25.71	2.73	17.95*	3.92	20.58	3.42
3	8	13.27	5.30	9.30*	7.56	10.60	6.64
4	16	6.56	10.66	4.85*	14.50	5.46	12.88
5	32	3.51	20.06	2.77*	25.71	3.03	23.22
6	64	2.05	34.30	1.54*	45.71	1.81	38.84

Tabela 6.10: Tempos de execução normalizados (%) e ganhos como uma função do número de processadores para o sistema de 1663 barras no computador *NCUBE*. São considerados 3 tipos diferentes de ordenação da lista de contingências (1555 casos); t_b = 467.27 segundos; os ganhos são relativos ao melhor tempo seqüencial (t = 70.35 %).

Sistema de 1663 barras - (*NCUBE*)							
Dimensão do cubo	Número de nós	Critério 1		Critério 2		Critério 3	
		Tempo	Ganho	Tempo	Ganho	Tempo	Ganho
0	1	100.0	0.63	70.28	0.91	63.68*	1.00
1	2	50.38	1.26	35.78	1.78	32.36*	1.97
2	4	25.71	2.47	17.88	3.56	16.45*	3.87
3	8	13.27	4.79	9.24	6.89	8.48*	7.50
4	16	6.56	9.66	4.78	13.31	4.29*	14.82
5	32	3.51	18.18	2.70	23.59	2.37*	26.83
6	64	2.05	31.10	1.47	43.31	1.43*	44.36

Tabela 6.11: Tempos de execução normalizados (%) e ganhos como uma função do número de processadores para o sistema de 1663 barras no computador *NCUBE*. A ordenação não faz parte do processo paralelo; a lista é considerada previamente ordenada por três critérios e já disponível na base de dados (1555 casos); t_b = 467.27 segundos; os ganhos são relativos ao melhor tempo seqüencial (t = 63.68 %).

Note-se também que o impacto da *boa ordenação* é amortecido quando a relação nc/np diminui. Em outras palavras, quando há disponibilidade de um considerável número de processadores, tal que a média de contingências por cada um deles esteja perto da unidade, uma ordenação de baixa qualidade (ou ordem aleatória) pode tornar-se competitiva. Por exemplo, na tabela (6.8) a diferença de percentual de tempo para dois processadores entre os critérios 1 e 3 (melhor tempo) é de 32.1 %. Com 64 processadores, a diferença de tempo entre critérios 1 e 2 (melhor tempo) cai para 21.3 % (média de 14 contingências por processador). Porém, razões nc/np baixas afetam a eficiência do processo global pela diminuição do balanço de carga entre processadores.

A tabela (6.9), que considera a lista já ordenada e disponível, mostra claramente o impacto da ordenação que neste caso foi de muita boa qualidade (critério 3). Com 64 processadores, o tempo de processamento foi de 2.5 segundos. Na tabela (6.8), o melhor tempo foi 2.86 segundos (critério 2).

Para o sistema B (1663 barras), os critérios 2 e 3 identificam conjuntos críticos com muitos elementos, comparados com o número de restrições que realmente ficam na base (7 restrições de contingência). Em ambos os casos, o número de contingências consideradas críticas foi de 565 casos. Na tabela (6.10), a relação esforço/qualidade é favorável ao critério 2. Note-se que o critério 3 ainda fornece uma lista de melhor qualidade, porém não suficiente para justificar o alto esforço computacional. Isto fica claro na tabela (6.11), onde é suposto que a lista já está ordenada na base de dados. O critério 3 tem claramente o melhor desempenho, conseguindo-se um tempo de processamento de 6.7 segundos (1.43 %). Observe que, com um processador, o tempo com esta ordenação é menor do que a metade do caso sem ordenação.

A tabela (6.12) mostra as eficiências para os sistemas 725 e 1663 barras, considerando dada a lista ordenada de casos. Novamente, o crescimento do número de cenários decorre no aumento da eficiência global. A figura (6.3) mostra graficamente o ganho desses dois casos.

Comentários Relativos à Portabilidade

A principal característica das máquinas de memória compartilhada é sua rapidez de comunicação. Na máquina utilizada, o tempo de acesso à memória comum está na faixa dos nanosegundos [24], que comparado aos tempos de processamento local (que estão na faixa de segundos a minutos) é um tempo desprezível; logo a via de comunicação através da memória comum pode ser considerada instantânea.

A migração da aplicação do computador *PP* para o *NCUBE* (ambiente de memória distribuída) decorre em um certo nível de penalização da comunicação. O mecanismo de troca de mensagens leva os tempos de comunicação para a faixa dos milisegundos [26], que não

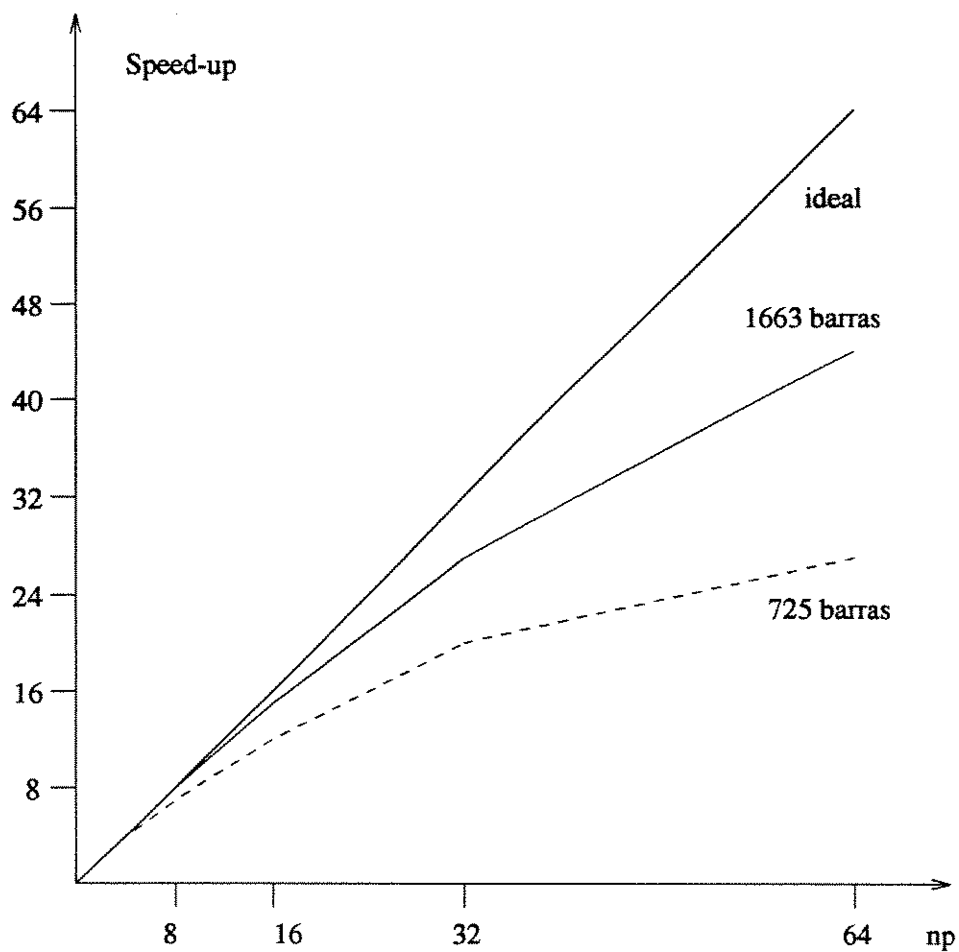

Figura 6.3: "Speed-up" para os sistemas de 725 e 1663 barras com o computador *NCUBE*, assumindo a lista de contingências ordenada previamente e disponível na base de dados; np: número de processadores.

Eficiências (%) - (*Computador NCUBE*)		
Processadores	Sistema A	Sistema B
1	100.0	100.0
2	99.5	98.5
4	97.0	96.7
8	87.9	93.7
16	76.6	92.6
32	62.3	83.8
64	42.5	69.3

Tabela 6.12: Eficiências obtidas no computador *NCUBE*, para os casos correspondentes ao critério 3 para os sistemas A (725 barras) e B (1663 barras); ordenação excluída do processo paralelo.

permite a reprodução de resultados tão bons como os obtidos na primeira arquitetura. Por exemplo, para o sistema de 725 barras, com oito nós no *PP* obtem-se uma eficiência de 105.8 % e no computador *NCUBE*, 87.8 %. Nesta análise deve-se levar em conta que a capacidade de processamento de um nó do *NCUBE* é muito superior ao do *PP*, que é baseado no processador *iAPX286*. Por outro lado, como foi dito na seção (4.4.1), o principal obstáculo da arquitetura de memória compartilhada é a escala. O bom desempenho obtido está diretamente ligado ao baixo número de processadores conectados ao barramento; o aumento desse número é acompanhado pela degradação do desempenho da arquitetura. No global, a implementação no computador *NCUBE* se apresenta eficiente e dentro do esperado. À medida que mais contingências são levadas em conta (lembrar que as listas utilizadas nos exemplos-testes apenas consideram contingências simples; se são incluídas contingências múltiplas, a lista torna-se imensa) a eficiência cresce. Os tempos obtidos nesta implementação são altamente competitivos ($t = 2.5$ segundos para sistema de 725 barras, tabela (6.9), $t = 6.7$ segundos para sistema de 1663 barras; ambos tempos com 64 processadores). Mostra-se então que, através de poucas alterações - praticamente transparentes para a aplicação - os programas podem migrar do ambiente de memória comum para memória distribuída; as alterações, comentadas no capítulo anterior, visam essencialmente eficiência do algoritmo concorrente.

6.4 Recapitulação

Neste capítulo foram apresentados os resultados da implementação concorrente do problema de fluxo de potência ótimo com restrições de segurança. Os testes implementados

visam avaliar a proposta sob duas perspectivas: desempenho e portabilidade.

Na avaliação de desempenho utilizam-se duas medidas de qualidade de algoritmos paralelos: o ganho ou "speed-up" e a eficiência. Os testes implementados focalizam dois aspectos: a) Analisar o impacto da ordenação da lista de contingências no tempo de processamento global, e b) O efeito da dimensão da lista de contingências na eficiência e no ganho no algoritmo concorrente.

As arquiteturas disponíveis são um computador de memória compartilhada de nove nós (*PP*) e um computador *NCUBE* de 64 nós (memória distribuída). Para testes no computador *PP* foram utilizados o sistema IEEE118 e um sistema brasileiro de 725 barras (por problemas de capacidade de memória, não foi possível testar sistemas maiores); no computador *NCUBE* foi utilizado o mesmo sistema de 725 barras e outro de 1663 barras.

Desempenho do Algoritmo Proposto

Sem dúvida, a ordenação da lista de contingência influencia diretamente o desempenho do algoritmo proposto. Isto pode ser verificado quando se assume que a lista já está ordenada em uma base de dados e disponível para seu uso.

Por outro lado, quando processo de ordenação é incluído no processo paralelo (no caso em que se começa com uma lista ordenada existente na base de dados, esta situação seria equivalente a um processo de reordenação), chega-se a uma decisão de compromisso, que depende da eficiência da metodologia de ordenação. Às vezes, o esforço de ordenação não é compensado com a geração de uma lista de boa qualidade. Isto leva a que, em termos de tempos globais, seja mais barata uma ordenação de "baixa qualidade" (que decorrerá, é claro, em trabalho extra dos processadores, verificando casos que não são críticos). Esta situação foi observada em alguns casos, onde o índice considerado mais sofisticado (critério 3) nem sempre consegue identificar adequadamente um conjunto de contingências críticas.

Note-se, também, que o impacto da ordenação é maior quando a razão nc/np (número de contingências por processador) é grande; quando esta relação vá para a unidade, a ordenação tem efeito mínimo, pois há suficientes processadores para tratar rapidamente todos os casos.

Por outra parte, observa-se que o aumento do tamanho do sistema (e também da lista de contingências) decorre em aumento da eficiência e do "speed up" do algoritmo. O aumento de cenários permite um melhor balanço de carga entre os processadores, amortecendo o desbalançeamento que acontece no final do processo concorrente.

O desempenho do algoritmo mostra-se eficiente. No computador *PP*, onde o custo de comunicação é praticamente zero, para altas razões nc/np observa-se comportamento superlinear do algoritmo. No computador *NCUBE* os resultados não atingem esse nível

de espetacularidade, mas são satisfatórios; na prática, o número de contingências a serem analisadas é muito maior do que os exemplos-testes aqui apresentados, pois incluem contingências múltiplas; para esses casos, a eficiência e "speed-up" devem acompanhar o aumento do tamanho do sistema e do número de cenários a tratar.

Portabilidade

No capítulo anterior apresentaram-se as alterações necessárias para a migração dos programas do computador *PP* para o *NCUBE*. A alteração básica envolve a obtenção do índice do caso que será analisado. Esta tarefa é feita utilizando recursos diferentes, porém, do ponto de vista da aplicação, é praticamente transparente. Conseqüentemente, a migração da aplicação foi consumada de maneira simples e rápida, preservando-se a filosofia assíncrona e obtendo-se uma boa eficiência.

Capítulo 7

Conclusões

Este trabalho apresentou um algoritmo concorrente assíncrono para a solução do fluxo de potência ótimo com restrições de segurança em computadores paralelos. Duas características se destacam no algoritmo proposto e as contribuições deste trabalho estão associadas a elas: eficiência e bom grau de portabilidade.
No global, este trabalho também pode ser visto como uma abordagem sistemática para aplicar o multiprocessamento de maneira eficiente a problemas de sistemas de potência relacionados com o controle ótimo-seguro, visando as metas de eficiência de desempenho e portabilidade entre arquiteturas diferentes.

Formulação

É utilizada uma versão linearizada do problema de fluxo de potência ótimo com restrições de segurança e resolvida com uma extensão do método proposto por Stott e Marinho. A combinação de programação linear dual e técnicas de relaxação tornam a formulação naturalmente adequada para processamento paralelo. Também, o acoplamento fraco entre as subtarefas em que o problema original é decomposto sugere um tratamento assíncrono delas.

Portabilidade

As primeiras iniciativas da utilização de multiprocessamento para a solução de problemas de fluxo de potência ótimo têm sido paralelizar, em uma máquina determinada, algoritmos já existentes, visando basicamente reduzir custos de desenvolvimento de "software". Por outro lado, junto com o surgimento de máquinas paralelas com variadas arquiteturas, a migração de programas entre esses ambientes diferentes torna-se uma meta cada dia mais requisitada. É desejável que um produto possa ser transferido de uma arquitetura para outra com pouco esforço de adaptação. Considerando estes aspectos, neste trabalho o desenvolvimento do algoritmo tem sido realizado com o auxílio de paradigmas ou modelos de programação, que objetivam principalmente ter uma visão abstrata da máquina multiprocessadora. Procedendo-se assim, o esforço de desenvolvimento do algoritmo é

concentrado na metodologia ds solução, desvinculando o processo das particularidades da máquina utilizada e decorrendo na obtenção de um produto com um razoável grau de portabilidade. A portabilidade, na sua máxima expressão, é difícil de ser atingida, pois alguns ajustes deverão ser feitos, seja para incluir as facilidades disponíveis no novo ambiente computacional ou por razões de eficiência.

Eficiência

Existe uma tendência natural que leva à simples paralelização de algoritmos seqüenciais. Isto, em geral, não é uma boa solução, pois nem sempre a melhor implementação seqüencial leva à melhor versão paralela. Neste trabalho tem-se optado por introduzir conceitos e construções de programação concorrente no desenvolvimento do algoritmo. Logo, para modelar o intercâmbio de informação entre tarefas em que o processo é decomposto, tem sido adotado um modelo ou paradigma *produtor-consumidor*. Este paradigma permite a execução assíncrona das tarefas concorrentes, que decorre em bom balanço de carga entre processadores e alta eficiência do algoritmo.

Resultados

O algoritmo concorrente proposto foi testado em uma máquina paralela de memória comum de 9 nós e logo portado com sucesso para um computador de memória distribuída de 64 nós *NCUBE*. Os sistemas de energia elétrica utilizados foram o *IEEE118* e dois sistemas brasileiros de 725 e 1663 barras, respectivamente. Um conjunto de testes foi implementado, visando avaliar a proposta sob as perspectivas de desempenho e portabilidade. Três critérios de ordenação das contingências foram considerados; a escolha desses índices tem como único objetivo fornecer diferentes tipos de ordenação, sem entrar no mérito da idoneidade deles. Para os objetivos deste estudo, os critérios adotados foram suficientes e cobriram praticamente todas as situações de interesse.

Observou-se que a ordenação tem grande impacto no desempenho do algoritmo quando o número de processadores que participam do processo é muito menor do que o número de contingências. Esse efeito tende a diminuir, na medida que o número de processadores cresce. Similarmente, o aumento do tamanho dos sistemas de energia elétrica vai acompanhado do aumento da lista de contingências, que decorre em melhor balanço de carga entre processadores. Nos testes apresentados neste trabalho, foram consideradas apenas contingências simples. Na prática, o número de contingências a serem analisadas é muito maior, pois são acrescentadas contingências múltiplas.

Extensões e trabalhos futuros

O algoritmo proposto possui o potencial de abranger uma família de problemas relacionados com fluxo de potência ótimo-seguro. Uma formulação, que envolve a representação das capacidades corretivas do sistema, se adapta adequadamente com o algoritmo aqui apresentado. Formulações simplificadas também podem ser cobertas pelo algoritmo. A

implementação da inclusão das capacidades corretivas deverá ser o próximo passo. A meta final desta linha de pesquisa é um programa que suporte todos os problemas relacionados com fluxo de potência ótimo-seguro e que migre confortavelmente de uma arquitetura para outra. Para cumprir essa meta, a formulação do problema de otimização também deverá ser complementada pela representação das variáveis e restrições reativas.

Bibliografia

[1] H.P. Asal, H. Glavitsch, G.Schaffer, "Requirements for Future Energy Management Systems", PICA Conf. Seattle, 1989.

[2] B. Stott, O. Alsac, A. Monticelli, "Security Analysis and Optimization", Proceedings of the IEEE, Special Issue on Computers in Power System Operations, Vol.75, pp 1623-1644, Dez. 1987.

[3] O. Alsac and B. Stott, "Optimal Load Flow with Steady-State Security", IEEE Transactions on Power Apparatus and Systems, Vol.93, pp.745-751, Maio/Junho 1974.

[4] B. Stott, J.L. Marinho, "Linear Programming for Power System Network Security Applications" , IEEE Transactions on Power Apparatus and Systems, Vol.98, pp 837-848, Maio/Junho 1979.

[5] B. Stott, J.L. Marinho, O. Alsac, "Review of Linear Programming Applied to Power System Rescheduling", IEEE PICA Conf., pp 142-154 , Cleveland, May 1979.

[6] EPRI, "Concurrent Processors for Computation of power System Reliability: Phase 1", Report EL-4598, Agosto 1986.

[7] A. Monticelli, M. V. F. Pereira, S. Granville, "Security-Constrained Optimal Power Flow with Post-Contingency Corrective Rescheduling", IEEE Trans. on Power Systems, Vol. PWRS-2, No 1, Fev. 1987.

[8] M.J. Teixeira, H.J.C.P. Pinto, M.V.F. Pereira and M.F. McCoy, "Developing Concurrent Processing Applications to Power System Planning and Operations", IEEE Transactions on Power Systems, Vol.5, pp.659-664, Maio 1990.

[9] H.J.C.P. Pinto, M.V.F. Pereira, M.J. Teixeira, "New Parallel Algorithms for the Security-Constrained Dispatch with Post-Contingency Corrective Actions", PSCC Conference Proceedings, Graz, 1990.

[10] E.W. Dijsktra , "Cooperating Sequencial Processes", F. Genuys, Programming Languages, Academic Press, New York, 1968.

[11] J. Boyle et alli, "Portable Programs for Parallel Processors", Holt. Rinehart and Winston, Inc., New York, 1987.

[12] A. Monticelli, A. Garcia, O.R. Saavedra, "Fast Decoupled Load Flow: Hypothesis, Derivations and Testing", IEEE Trans. Power Syst., Vol 5, pp 1245-1431, nov. 1990.

[13] A. Monticelli, "Fluxo de Carga em redes de Energia Elétrica", Ed. E. Blucher Ltda., São Paulo, 1983

[14] J. Carpentier,"Toward a Secure and Optimal Automatic Operation of Power Systems", Proceedings PICA, Canadá, maio 1987.

[15] A. Monticelli,"Análise Estática de Contingências em Sistemas de Energia Elétrica", Tese de Livre Docência, UNICAMP, 1980.

[16] R. Duncan,"A survey of Parallel Computer Architectures", Computer, Vol 23 No 2, Fevereiro 1990.

[17] M. Rodrigues, O.R. Saavedra, A. Monticelli, "Asynchronous Programming Model for the Concurrent Solution of Security Constrained Optimal Power Flow Problem", paper WP93-237, IEEE summer Meeting, 1993.

[18] A. Monticelli, M. Rodrigues, O. R. Saavedra, "Um Esquema Produtor-Consumidor para a Solução Concorrente do Problema de Fluxo de Potência Ótimo com Restrições de Segurança", IV Simpósio Brasileiro de Arquitetura de Computadores- Processamento de Alto Desempenho, São Paulo, Brasil, Outubro 1992.

[19] M. Quinn,"Designing Efficient Algorithms for Parallel Computers", Ed. McGraw-Hill, 1987.

[20] K. Hwang e F. Briggs, "Computer Architecture and Parallel Processing", Ed. McGraw-Hill, 1984.

[21] H. Stone, "High Perfomance Computer Architectures", Addison-Wesley Publishing Co., 1987.

[22] IEEE Committee Report, "Parallel Processing in Power Systems Computation", IEEE Transactions on Power Systems, Vol.7, pp 629-638, Maio 1992

[23] H. Bal, J. Steiner and A. Tanenbaum, "Programming Languages for Distribuited Computing Systems", ACM Computing Surveys, Vol 21 No 3, Set. 1989.

[24] E. Cavalli e M.C. Zabeu, "Sistema Hardware para Processamento Paralelo", I Simpósio Brasileiro de Arquitetura de Computadores - Processamento Paralelo, pp 91-100, Gramado, RS, maio 1987.

[25] NCUBE-Departamento de Apoio, comunicação verbal, novembro 1992.

[26] NCUBE-2 Processor Manual, dezembro 1990.

[27] J. Mooney, "Strategies for Supporting Application Portability", Computer, Vol 23 No 11, novembro 1990.

[28] P. Stenström, "Reducing Contention in Shared-memory Multiprocessors", Computer, Vol 21 No 11, novembro 1988.

[29] R. Rettberg e R. Thomas, "Contention is no Obstacle to Shared-memory Multiprocessing", Communications of de ACM, Vol 29 No 12, dezembro 1986.

Apêndice A

Breve Revisão do Algoritmo de Programação Linear Utilizado.

Modelo Linear Incremental

Usando um modelo linear, as relações entre os ângulos das tensões e as injeções de potência ativa nas barras são descritas pelas equações:

$$\Delta P = B' \Delta \theta \tag{A.1}$$

onde ΔP e $\Delta \theta$ são os vetores incrementais de injeções de potência e de ângulos, respectivamente. B' é a matriz do método desacoplado rápido segundo algoritmo Primal [12].
O fluxo incremental no ramo $k - m$ é dado por :

$$\Delta f_{km} = -b_{km}(\Delta \theta_k - \Delta \theta_m) \tag{A.2}$$

O vetor esparso ΔP contém os incrementos de injeção de potência ativa de barras (exceto na barra slack), cujos elementos não zeros correspondem às barras de geração controlada.

Monitoramento de fluxos nos ramos

Para o propósito de testar sobrecargas em ramos utiliza-se a expressão esparsa dos fluxos (A.2), isto é:

$$\Delta f_{km}^{min} \leq \Delta f_{km} \leq \Delta f_{km}^{max} \tag{A.3}$$

Metodologia de Otimização

O estado operativo do sistema no inicio de cada iteração PL é definido pela equação-base seguinte:

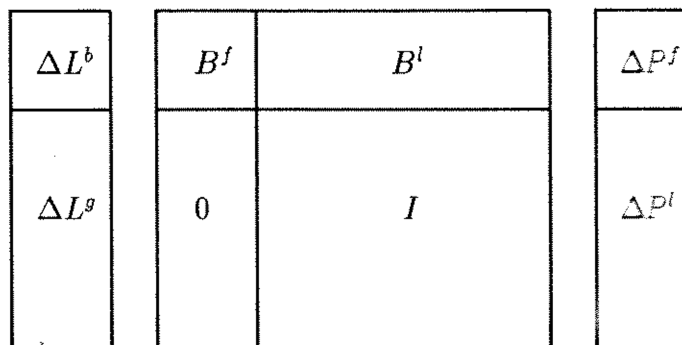

Figura A.1: Estrutura da base reduzida.

$$\Delta L = [B]\Delta P \tag{A.4}$$

onde $[B]$ é matriz base de dimensão $n * n$, que é formada pelas restrições ativas associadas ao ponto de operação atual. O vetor ΔL é composto pelos limites incrementais ativos (de geradores e ramos) e ΔP é o vetor incremental das variáveis de controle. A primeira linha de (A.4) corresponde à equação de balanço que sempre deve estar presente na base:

$$\beta^t \Delta P = 0 \tag{A.5}$$

O vetor β contém os fatores de perda de transmissão incremental e sua obtenção é dada mais para o fim. Se a variação das perdas é desprezível, então o vetor β é unitário. A forma de (A.4) é ilustrada na figura (A.1), onde o vetor ΔP^f tem dimensão m e ΔP^l dimensão $(n - m)$.

A matriz $[B]$ aparece particionada em 4 submatrizes, sendo que B^f é uma matriz $m * m$, onde a dimensão m é em geral pequena, B^l é uma matriz não quadrada $m * (n - m)$, e I é matriz identidade. O índice f está associado com gerações livres (variáveis básicas), enquanto que l está associado com gerações fixadas num limite (variáveis não básicas). As linhas de 2 até n contém uma mistura de restrições ativas de ramos e geração. Se $(m - 1)$ limites de ramos estão ativos, então existem exatamente m gerações livres . A figura (A.1) apresenta uma forma canônica da base reduzida, onde as $(m - 1)$ restrições não esparsas de ramo aparecem nas linhas de 2 até m.

Logo, apenas são não esparsas as primeiras m linhas da figura (A.1) as que são armazenadas explicitamente.

$$\Delta L^b = [B^f]\Delta P^f + [B^l]\Delta P^l = [B^r]\Delta P \tag{A.6}$$

As gerações livres podem ser colocadas em funções dos limites incrementais de fluxo e geração:

$$\Delta P^f = [B^f]^{-1}[\Delta L^b - [B^l]\Delta P^l] \tag{A.7}$$

O esforço computacional para obter a inversa ou os fatores da matriz $[B^f]$ (dimensão $(m * m)$) é pequeno, desde que m é tipicamente muito pequena (na ordem de 10).

O estado operativo do sistema no início de cada iteração PL é definido por (A.4). As gerações ou fluxos violados são fixados no limite correspondente, pela introdução de restrições de igualdade na base de maneira seqüencial. Para a fixação de uma geração ou fluxo, outra geração/fluxo que esteja fixado deverá ser *relaxada*, isto é, retirada da base.

Escolha da variável entrante

A escolha da variável entre as restrições violadas para ser introduzida na base durante uma iteração é, em princípio, arbitrária. Porém, por razões de eficiência aqui é assumida a seleção da restrição mais violada.

Escolha da variável que sai da base

A restrição a ser retirada da base deve ser selecionada com otimalidade. Esta seleção ótima é realizada em dois passos:

- As restrições ativas existentes que estão na base são testadas para *eligibilidade*. Uma restrição é elegível quando, se liberada com o fim de aliviar uma violação, ela abandona o limite no qual estava fixada. Se não há restrição elegível, então o problema é *infactível.*

- A escolha ótima entre as restrições elegíveis é feita utilizando um *teste de razão mínima* (minimização de $|\lambda_k/S_k|$, onde λ_k é o custo relativo e S_k é a sensibilidade da restrição entrante com relação à restrição candidata k).

A seguir resume-se uma iteração PL, lembrando que maiores detalhes são encontrados na referência [4]

Resumo dos passos de cada iteração PL

Três possíveis tipos de restrições podem ser incorporadas ao processo PL :

- Restrição de geração.

- Restrição de ramo.

- Restrição de contingência.

Seja A o vetor restrição, particionado assim:

$$A^t = [A^f|A^l] \tag{A.8}$$

A primeira partição contém os índices correspondentes aos geradores livres (variáveis básicas) e a partição restante aos geradores em limite.

Se A é uma restrição de geração, então todos seus elementos serão nulos, exceto aquele com índice correspondente ao gerador a ser fixado, que contém o valor 1. Por outro lado, se A é uma restrição de ramo, então ela é em geral não esparsa e sua dedução é dada no apêndice B. Por fim, se A é uma restrição de contingência, ela é também não esparsa e o procedimento de identificação de seus elementos é dado no apêndice C.

Uma iteração PL envolve os seguintes passos:

i) Calcular as sensibilidades da restrição entrante com relação às variáveis num limite (ramos, geradores), isto é:

$$S^b = A^f[B^f]^{-1} \tag{A.9}$$

$$S^g = A^l - S^b[B^l] \tag{A.10}$$

onde S^b é o vetor de sensibilidades da restrição entrante com relação aos ramos que estão num limite na base. S^g é o vetor de sensibilidades da restrição entrante com relação aos geradores fixados num limite.

ii) Com as sensibilidades calculadas no passo anterior, identificar o conjunto de restrições ativas elegíveis (aquelas que, se liberadas, aliviarão ou eliminarão a violação na restrição entrante).

iii) Calcular o custo incremental λ, particionado e calculado da seguinte forma:

$$\lambda^b = C^f[B^f]^{-1} \tag{A.11}$$

$$\lambda^g = C^l - \lambda^b[B^l] \tag{A.12}$$

onde :

$$C^t = [C^f|C^l] \tag{A.13}$$

é o vetor de custo da função objetivo.

iv) Utilizando (A.9), (A.10), (A.11) e (A.12) aplicar o teste de razão mínima para identificar a restrição a ser relaxada:

$$r_k = |\lambda_k / S_k| \tag{A.14}$$

v) Atualizar a equação base.

vi) Atualizar os geradores livres utilizando (A.7).

Determinação dos fatores de perdas de transmissão

Dada a equação de balanço:

$$\beta_1 \Delta P_1 + \beta_2 \Delta P_2 + ... + \beta_n \Delta P_n = 0 \tag{A.15}$$

Para a barra i, o fator de perdas é:

$$\beta_i = 1 - \frac{\partial P_p}{\partial P_i} \tag{A.16}$$

onde P_p representa as perdas de transmissão. Em forma vetorial, a derivada das perdas com relação às injeções de potência ativa nas barras é dada por:

$$\frac{\partial P_p}{\partial P} = \left[\frac{\partial \theta}{\partial P}\right]^t \left[\frac{\partial P_p}{\partial \theta}\right] = [H]^{-1} \left[\frac{\partial P_p}{\partial \theta}\right] \tag{A.17}$$

onde H é a submatriz jacobiana de sensibilidades entre as injeções de potência ativa e os ângulos nas barras. Considerando o modelo linear, H é substituída pela matriz B' do método desacoplado rápido:

$$\frac{\partial P_p}{\partial P} = [B']^{-1} \left[\frac{\partial P_p}{\partial \theta}\right] \tag{A.18}$$

A derivada das perdas com relação ao ângulo da barra k é determinada por:

$$\frac{\partial P_p}{\partial \theta_k} = 2 \sum_{m \, \in \, \omega_k} V_k V_m G_{km} sen\theta_{km} \tag{A.19}$$

onde ω_k é o conjunto das barras vizinhas à barra k, G_{km} é o elemento $k-m$ da matriz de condutância do sistema e $\theta_{km} = \theta_k - \theta_m$.

Apêndice B

Restrições de Ramos

Seja o modelo linear incremental da rede :

$$\Delta P = B'\Delta\theta \tag{B.1}$$

onde ΔP e $\Delta\theta$ são os vetores incrementais de injeções de potência e de ângulos, respectivamente. B' é a matriz do método desacoplado rápido segundo algoritmo Primal [12].
O fluxo incremental no ramo $k-m$ é dado por :

$$\Delta f_{km} = -b_{km}(\Delta\theta_k - \Delta\theta_m) \tag{B.2}$$

o que é equivalente a :

$$\Delta f_{km} = e^t_{km}[B']^{-1}\Delta P \tag{B.3}$$

onde e^t_{km} é um vetor com todos seus elementos nulos, exceto nas posições k e m , onde há 1 e -1 , respectivamente. O elemento b_{km} é a susceptância do ramo $k-m$.

De (B.1) e (B.3) tem-se :

$$e^t_{km}\Delta\theta = e^t_{km}[B']^{-1}\Delta P \tag{B.4}$$

Finalmente, de (B.3) e (B.4) tem-se o fluxo incremental em termos das variáveis de controle incrementais :

$$\Delta f_{km} = A^t_{km}\Delta P \tag{B.5}$$

onde : $A^t_{km} = -b_{km}[B']^{-1}e_{km}$, lembrando que B' é simétrica.

Apêndice C

Restrições de Contingências

Para a saída do ramo $i-j$, o fluxo incremental contingente no ramo $k-m$ em termos dos ângulos do estado de pré-contingência vem dado por:

$$\Delta f_{km}^{c} = -b_{km}(\Delta\theta_k - \Delta\theta_m - \alpha_{km}(\Delta\theta_i - \Delta\theta_j)) \tag{C.1}$$

onde α_{km} é o fator de distribuição de fluxos, e b_{km} é a susceptância do ramo $k-m$.

De maneira similar do que em (A) a expressão (B.1) pode ser colocada na forma :

$$\Delta f_{km}^{c} = -b_{km}(e_{km}^{t} - \alpha_{km}e_{ij}^{t})\Delta\theta \tag{C.2}$$

porém, de (B.1) tem-se que :

$$\Delta f_{km}^{c} = -b_{km}(e_{km}^{t} - \alpha_{km}e_{ij}^{t})[B']^{-1}\Delta P \tag{C.3}$$

Logo, o fluxo incremental contingente, em termos das variáveis incrementais de controle do estado de *pré-contingência*, vem dado por:

$$\Delta f_{km}^{c} = A_{km}^{t}\Delta P \tag{C.4}$$

onde agora A_{km} é calculado como :

$$A_{km} = -b_{km}[B']^{-1}(e_{km} - \alpha_{km}e_{ij}) \tag{C.5}$$

O fator de distribuição pode ser calculado facilmente utilizando os fatores triangulares de $[B']^{-1}$. Utilizando compensação [13]:

$$\alpha_{km} = \gamma\mu \tag{C.6}$$

Onde :

$$\gamma = e_{km}^{t}[B']^{-1}e_{ij} \tag{C.7}$$

e :

$$\mu = 1/((1/\Delta b_{ij}) + (1/b_{ij}^{eq})) \tag{C.8}$$

onde Δb_{ij} é a susceptância do ramo que está em contingência e b_{ij}^{eq} é a susceptância equivalente entre os nós i e j , que pode ser rapidamente calculado assim :

$$b_{ij}^{eq} = 1/z_{ij}^{eq} \tag{C.9}$$

onde :

$$z_{ij}^{eq} = e_{ij}^{t}[B']^{-1}e_{ij} \tag{C.10}$$

Apêndice D

Programas Fontes

Neste apêndice, apresentam-se versões simplificadas dos programas Mestre e Escravo escritos para os computadores de memória comum e memória distribuída. O objetivo de incluir estes programas, é dar uma idéia da forma que assumem os códigos nas duas máquinas e também ilustrar a utilização de algumas funções da biblioteca paralela do computador *NCUBE*.

D.1 Programas Mestre e Escravo para o Computador de Memória Compartilhada (*PP*)

```
* ***********************************************************
* *************** Processador Preferencial *******************
* ***********************************************************

      Program Mestre
      include'cabez.inc'

      iviola=1

***********************************************************
*        c++++++ [ Inic. ponteiro para as variáveis] +++++++++
*       c++++++ [ que ficam na memória comum ] +++++++++
***********************************************************
c     Variáveis que ficam na memória comum:
c           marca(.)
c           klassif(.)
c           DP(.) ( vetor de var. de controle)
c           iteta

      call GMEMSET
***********************************************************
*        ++++[ identifica nó ]++++
***********************************************************

      mynode=MYNODE()
***********************************************************
*        c++++++ [ Inic. fifo e barreira] +++++++++
*       c++++++ [ inicialização vetor de marcas ] +++++++++
***********************************************************
      call INIC_MEM
c++++++ [ Libera início primer estágio] +++++++++
      call SETSEMAF(inicio, -2)

      do i=1,nso
          marca(i) = -1
      end do

c       write(*,*)' INICIO PROCESSO PARALELO '
***********************************************************
*        ++++[ Cálculo do índice e ordenação ]++++
***********************************************************

6     nnt = FIFOSTAT(fifo, 6)

      if(nnt.ge.0.)then
c      [recebe índice ]
          irfifo = FIFOGET(fifo, fila, 6)
          call REC_IND(kx)
          if(kx.ge.nso)go to 5
      else
c      [calcula índice]
          ll = GETSUB(monoptm, nso)
          ioutlocal = klasif(ll)
          call CALC_IND(kx)
          if(kx.ge.nso)then
            goto 5
          else
            goto 6
          endif
      endif
```

```
*************************************************************
* ++++[ ordena e escreve lista ordenada na mem. comum ]++++
*************************************************************

5    call REORDE
*************************************************************
* ++++[ PROCESSO OTIMIZAC,ÃO COM CONTING. ]++++
*************************************************************

c++++++ [ Inic. fifo e barreira] ++++++++++++++++
     call INIC_MEM
c++++++ [ Libera início segundo estagio] +++++++++
   call SETSEMAF(inicio, -1)
     iteta = 0
1    call PL(CE)
     iteta = iteta + 1
*************************************************************
*     ++++[ Escreve novo solução na memória comum ]++++
*************************************************************

     call NOVO_EST
     write(*,*)' [nova soluc,ão ] iteta= ', iteta
*************************************************************
*     ++++[ Processo concorrente mestre/escravo]++++
*************************************************************

3    nnt = FIFOSTAT(fifo, 6)
     if(nnt.gt.0)then
c ———[ Mestre pega restrição do buffer ]————
        irfifo = FIFOGET(fifo, fila, 6)
        if(irfifo.lt.0)then
           write(*,*)' Todos os casos estão marcados '
           go to 4
        endif
        call VERIFI
        if(iviola.gt.0)then
c      [inclui a restricao verificada no mestre ]
         call RECEBE
         go to 1
        endif
     elseif(nnt.lt.0)then
c—— [ Mestre pega caso para analisar contingencia ]——
        ll = GETSUB(monoptm, nso)
        ioutlocal=klasif(ll)
        if(marca(ll).ne.iteta)then
          call ACONT
          if(iviola.eq.1) go to 1
        else
c         [ caso marcado, pega outro ]
          if(Kasos_marcados.ge.nso) go to 4
        endif
     endif
     go to 3
```

```
c     [ relatórios finais ]
4      call RELAT2
      call FECHA
      stop
      end
* ************************************************************
* ************************************************************
* ************************************************************
```

```
* ***********************************************************
* *************** Processador Preferencial ****************
* ***********************************************************

      program ESCRAVO

      include'cabez.inc'

***********************************************************
*     ++++++[ Inic. ponteiro para as variáveis] +++++++++
*     ++++++[ que ficam na memória comum ] +++++++++
***********************************************************

      call GMEMSET
***********************************************************
*       ++++[ identifica nó ]++++
***********************************************************

      mynode=MYNODE()

***********************************************************
*++++[ Esperando ordem do Mestre para começar ]++++
***********************************************************

      call LIBERA_PROC
*************************************************************
*      ++++[ calcula ind. de sobrecarga em paralelo]++++
*************************************************************

      do while(ll.gt.0)
         ll = GETSUB(monoptm, nso)
         ioutlocal = klasif(ll)
         call SCAN
         irfifo = fifoput(fifo_pi, fila, 6)
      enddo

*************************************************************
*++++[ Esperando ordem para começar segundo estágio ]++++
*************************************************************

      call LIBERA_PROC
```

```
************************************************************
*      ++++[ Pega caso para análise de contingências ]++++
************************************************************

2    do while(kasos_marcados.lt.nso)

       do while(ll.gt.0)

       ll = getsub(monoptm, nso)
       ioutlocal=klasif(ll)
       mlocal=marca(ll)
       itlocal=iteta

* ++++[ Lé estado na memória comum e o copia em memória local ]++++

       call LE_ESTADO
       nnt = NTEST(-1,msgtyp)
        call CALC_T

       if(mlocal.ne.iteta)then

          call ACONT

          if(iviol.gt.0)then
c               [ envia restrição ]
                irfifo = ffifoput(fifo_a, buffer, sizebuffer)
          else
                marca(ll)=itlocal
          endif
       else
          call CONTADOR
       endif

       enddo

     enddo
******************************************************************
* ++++[ Escravo terminou e detectou que não há casos sem marca. ]++++
* ++++[ Envia aviso de convergência ao Mestre (nó 0)             ]++++
******************************************************************

     irfifo = ffifoput(fifo, buffer, 0)

     stop' escravo parou'
     end

* ************************************************************
* ************************************************************
* ************************************************************
```

D.2 Programas Mestre e Escravo para o Computador de Memória Distribuída (*NCUBE*)

```
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE

      subroutine Mestre
      include'cabez.inc'
      dimension iaux(nlmax),pi(2),ntip(64)

      iviola=1

***********************************************************
*      ++++[ define tipos para dados a transmitir]++++
***********************************************************
c     tipo      dado
c      25     indice de sobrecarga
c      34     klassif
c      45     DP ( vet. de controle)
c      56     restrição de conting.
c      67     flag de parada(it_escravo)
***********************************************************
*      ++++[ identifica nó ]++++
***********************************************************

      mynode=WHOAMI()

c       [ inicialização vetor de marcas ]
      do i=1,nso
          marca(i) = -1
      end do

      nbase=nodes
      nofinal=nodes-1
c       write(*,*)' INICIO PROCESSO PARALELO '
***********************************************************
*      ++++[ Cálculo do índice e ordenação ]++++
***********************************************************
      msgtyp=21
      nbytes =2*8
      nodall=-1
      ll=mynode + 1
      kx=1

6     nnt = ntest(-1,msgtyp)

      if(nnt.ge.0.)then
c       [recebe índice ]
          call NREAD(pi,nbytes,-1,msgtyp,iflag)
          call REC_IND(kx)
          if(kx.ge.nso)go to 5
      else
c       [calcula indice]
          ioutlocal = klasif(ll)
          call CALC_IND(kx)
          if(kx.ge.nso)then
            goto 5
          else
            goto 6
          endif
      endif

c     [ordena a lista]

5     call REORDE
```

```
*************************************************************
*      ++++[ Remete Lista ordenada para escravos ]++++
*************************************************************
      ncc=ncrit
      mask=-1
      msgtyp=34
      nbytes =(nso+1)*4
      nodall=0
      klasif(0)=ncrit

      call NBROADCAST(klasif, nbytes,nodall, msgtyp, mask)
*************************************************************
*      ++++[ PROCESSO OTIMIZAC,ÃO COM CONTING. ]++++
*************************************************************

      iteta = 0
      ll= mynode+1

1     call PL(CE)

      iteta = iteta + 1
*************************************************************
*      ++++[ Remete estado novo para escravos ]++++
*************************************************************
      mask=-1
      dpt(0)=iteta
      msgtyp=45
      nbytes =(ng+1)*8
      nodall=0

      if(nodes.gt.1)then
         dpt(0)=iteta
         call NBROADCAST(dpt,nbytes,nodall, msgtyp, mask)
      endif
      write(*,*)' [nova soluc,ão ] iteta= ', iteta
*************************************************************
*     ++++[ Processo concorrente mestre/escravo]++++
*************************************************************

      msgtyp=56
      nbytes =(ng+5)*8
      nodall=-1

3     nnt = ntest(-1,56)

      if(nnt.gt.0)then

c ——[ Mestre pega restrição do buffer ] ——— c
         call NREAD(alocal,nbytes,nodall,msgtyp,iflag )
         call VERIFI
         if(iviola.gt.0)then
c       [inclui a restricao verificada no mestre ]
          call RECEBE
          go to 1
         endif

      elseif(nnt.lt.0)then
```

```
c—— [ Mestre pega caso para analisar contingencia ]—— c
          ioutlocal=klasif(ll)
          if(marca(ll).ne.iteta)then
            call ACONT
            if(iviola.eq.1) go to 1
          else
c           [ caso marcado, pega outro ]
            if(ll.ge.nso) go to 4
            ll=ll+ nbase
          endif
      endif
      go to 3

c ——[ Lê as flags enviadas pelos escravos ]——
c ——[ e testa critério de parada          ]——

4     msgtyp=67
      nbytes =1*4
      nodall=-1
      write(*,*)' Lendo flags dos escravos'

      do i=1,nofinal
          nn=ntip(i)
          if(nn.ne.iteta)then
            call NREAD(it_escravo,nbytes,i,msgtyp,iflag )
            if(it_escravo.ne.iteta)go to 3
            ntip(i)=iteta
          endif
      end do

c     [ manda escravos parar ]
      istop=1
      nodall=0
      nbytes=1*4
      msgtyp=70

      call NBROADCAST(istop,nbytes,nodall,msgtyp, mask)

c     [ relatórios finais ]

      call RELAT2
      call FECHA
      stop
      end

* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
```

```
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE

      subroutine ESCRAVO(mynode,nodesçe)

      include'cabez.inc'
      dimension ce(max),pi(2)
************************************************************
*      ++++[ define tipos para dados a transmitir]++++
************************************************************
*      tipo      dado
*      25        ind. de sobrecarga
*      34        klassif
*      45        DP ( vet. de controle)
*      56        restrição de conting.
*      67        flag de parada (it_escravo)
* ************************************************************
*      ++++[ Identifica nó]++++
************************************************************

      mynode=WHOAMI

************************************************************
*      ++++[ calcula ind. de sobrecarga em paralelo]++++
************************************************************

      nbase=nodes
      mmynod=mynode+1
      msgtyp0=21
      nbytes0 =2*8
      nodall=0

      do ll=mmynod, nso, nbase
         ioutlocal = klasif(ll)
         call SCAN
         call NWRITE(pi,nbytes0,nodall,msgtyp0,iflag)
      enddo

      do i=1, nso
         marca(i)=-1
      end do
************************************************************
*       ++++[ Lé Klasif() do Mestre ]++++
************************************************************

      msgtyp=34
      nbytes =(nso+1)*4
      nodall=0
      call NREAD(klasif,nbytes,nodall,msgtyp,iflag )
      ncrit =klasif(0)
```

```
*****************************************************************
*       ++++[ Pega caso para análise de contingências ]++++
*****************************************************************

      it_escravo=0
      msgtyp=45
      nbytes1 =(ng+1)*8
      nbytes2 =(ng+5)*8
      nbytes3 =1*4
      msgtyp1=56
      msgtyp2=67
      nodall=0
      mmynod=mynode+1
      nnc=ncrit

2      do while(it_escravo.eq.0)
      it_escravo=1

        do ll=mmynod, nso,nbase

        ioutlocal=klasif(ll)
c       [ Pega estado mais rescente ]

        nnt = NTEST(-1,msgtyp)
        if(nnt.ge.0)then
                call NREAD(dpt,nbytes1,nodall,msgtyp, iflag )
                call CALC_T
        endif

        if(marca(ll).ne.iteta)then

           it_escravo=0
           call ACONT

           if(iviol.gt.0)then
c                [ envia restrição ]
                 call NWRITE(a,nbytes2,nodall,msgtyp1,iflag)
           else
                 marca(ll)=iteta
           endif
        endif

        enddo

      enddo
*******************************************************************
*       ++++[ envia aviso de convergência ao Mestre(nó 0) ]++++
*******************************************************************

      it_escravo=iteta
      call NWRITE( it_escravo,nbytes3,nodall,msgtyp2,iflag)
*******************************************************************
*       ++++[ Escravo recebe sinal para parar ]++++
*******************************************************************
      msgtyp3=70
      nodall=0

      call nread(istop,nbytes3,nodall,msgtyp3,iflag)
      stop' escravo parou'
      return
      end

* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
* nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE nCUBE
```