**Modelos Analíticos de
Desempenho para
Sistemas de Memória
Compartilhada Virtual**

Marco Dimas Gubitoso

TESE APRESENTADA
NO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU
DE
DOUTOR EM MATEMÁTICA APLICADA

Área de Concentração: **Computação**
Orientador: **Prof. Dr. Siang Wun Song**

-São Paulo, dezembro de 1996-

Modelos Analíticos de Desempenho para Sistemas de Memória Compartilhada Virtual

Este exemplar corresponde à redação
final da tese devidamente corrigida
e defendida por Marco Dimas Gubitoso
e aprovada pela Comissão julgadora
São Paulo, dezembro de 1996

Banca Examinadora:

• Prof. Dr. Siang Wun Song (Presidente)	IME-USP
• Prof. Dr. Carlos Humes Júnior	IME-USP
• Prof. Dr. Jairo Panetta	IEAv-CTA
• Prof. Dr. Edson dos Santos Moreira	ICMSC-USP
• Prof. Dr. Philippe Olivier Alexandre Navaux	II-UFRGS

Resumo

Análise e predição de desempenho vêm se tornando cada vez mais importantes na medida em que as opções de uso de sistemas paralelos aumentam e pela alta sensibilidade do desempenho com relação a pequenas variações em um programa. Neste sentido é fundamental a compreensão não apenas do desempenho obtido mas das causas que levam a um certo resultado. A abordagem analítica, por ser baseada em uma descrição detalhada do funcionamento do programa, permite um melhor entendimento do desempenho. Este trabalho procura explorar este tipo de abordagem na análise teórica do comportamento de sistemas paralelos e na predição de desempenho de um sistema de memória compartilhada virtual. Os casos teóricos estudados são o da alocação de processadores para laços do tipo *DOALL* independentes e o do tempo total de execução em aplicações com decomposição de domínio com comunicação assíncrona. Para o sistema de memória compartilhada virtual é desenvolvido um método para a construção de modelos analíticos de desempenho que se mostrou bastante preciso para as aplicações estudadas.

A meus pais.

Agradecimentos

Em primeiro lugar, eu quero agradecer ao Prof. Siang W. Song pela orientação e valorosos conselhos, bem como pelas diversas portas que me abriu.

Gostaria também de agradecer ao grupo do Prof. Wolfgang Schröder-Preikshat, do GMD-first, pela boa acolhida e em especial a Jörg Cordsen cuja colaboração foi absolutamente fundamental no desenvolvimento deste trabalho.

Ao Prof. Thomas Fahringer, “old chap”, quem primeiro me chamou a atenção para a importância desta área de pesquisa.

Ao Markus Endler, pelo apoio constante, pelas sugestões e pelo projeto PLENA, sem o qual este trabalho não seria o que é.

Ao grande amigo Jordão, e ao Luis Renato, pelas frutíferas discussões sobre as bolhas.

Aos colegas do MAC, e em particular a Kunio Okuda, pela competição saudável e pela vibração a cada etapa vencida.

Às secretárias do departamento pela paciência e constante simpatia.

À dona Vilma, pelos inúmeros cafezinhos.

À Edna Gubitoso, pelo carinho, paciência e tudo o que fez por este trabalho.

E ao meu filho Thomas, que soube tão bem compreender a pouca atenção lhe dei na fase final da tese.

Abstract

Performance analysis and prediction are becoming more and more important with the growth in the number of options of usage of parallel systems and because of the high sensibility of performance with small changes in a program. In this sense, it is fundamental to understand not only the resulting performance, but the causes of that particular result. The analytical approach, being based on a detailed description of the program, allows a better performance understanding. This work seeks to explore this approach in the theoretical behavior analysis of parallel systems and in the performance prediction for a virtual shared memory system. The theoretical situations studied are the processor allocation for *DOALL* loops and the total execution time in domain decomposition applications with asynchronous communication. For the virtual shared memory system, a method to build analytical performance models is developed which produced very precise models for the applications studied.

Sumário

1	Introdução	1
1.1	Modelagem de desempenho	2
1.2	Objetivos do trabalho	3
1.2.1	Análise teórica de comportamento	3
1.2.2	Memória compartilhada virtual	3
1.3	Resumo do trabalho	5
2	Análise de Desempenho	6
2.1	Definições	7
2.2	Sobrecargas	9
2.2.1	Comunicação	10
2.2.2	Desbalanceamento de carga	11
2.2.3	Contenção	12
2.2.4	Barreiras	12
2.3	Modelagem	13
2.3.1	Modelagem analítica	15
2.3.2	Modelagem estatística	15
2.3.3	Simulação	16
2.3.4	Empírica	16
2.3.5	Modelos híbridos	16
2.3.6	A modelagem usada neste trabalho	17
3	Trabalhos Relacionados	18
3.1	Predição automática de desempenho	18
3.1.1	P^3T : Parallel Performance Prediction Tool	18
3.1.2	Lost Cycles Toolkit	19
3.1.3	Carnival e Waiting Time Analysis	20
3.2	Outros trabalhos	21

4	Influência de Sobrecargas na Alocação de Processadores para Laços Independentes	22
4.1	Introdução	22
4.2	Definição do problema	22
4.3	Tempos e estratégias de execução	24
4.3.1	Casos sem sobrecarga	25
4.3.2	Casos com sobrecarga	27
4.4	Casos particulares	28
4.4.1	Modelo constante	28
4.4.2	Modelo linear	28
4.4.3	Um único laço	29
5	Decomposição de Domínio com Comunicação Assíncrona	30
5.1	Comunicação síncrona	30
5.2	Comunicação assíncrona	32
5.3	Simulações	36
6	Sistemas de Memória Compartilhada Virtual	40
6.1	Memória compartilhada virtual	41
6.1.1	Coerência	42
6.1.2	Manutenção da consistência	43
6.2	Sistema básico	43
6.2.1	MANNA	44
6.2.2	PEACE e VOTE	44
7	Modelagem	50
7.1	Impacto de cada sobrecarga no <i>VOTE</i>	52
7.2	Notação	53
7.3	Base do método	53
7.3.1	Suposições	54
7.3.2	Dados de entrada	55
7.4	Modelos básicos	56
7.4.1	Modelos das falhas de acesso	56
7.4.2	Modelos de primitivas	57
7.4.3	Modelo de barreiras	59
7.5	Método	60
7.5.1	Objetos compartilhados	61
7.5.2	Eventos	63

7.5.3	Instante de cada evento	67
7.5.4	Construção dos modelos para grupos de eventos	68
7.5.5	Construção do modelo final	69
7.6	Situações especiais	70
7.6.1	Decomposição de domínio	70
8	Resultados	72
8.1	Multiplicação de matrizes	72
8.1.1	Modelo da versão “padrão”	72
8.1.2	Resultados experimentais	73
8.1.3	Versão otimizada	76
8.2	Sobre relaxação	79
8.2.1	Construção do modelo	80
8.2.2	Resultados experimentais	83
8.2.3	Otimizações	83
9	Conclusões	93
9.1	Contribuições	93
9.2	Trabalho futuro	95

Lista de Tabelas

6.1	Tempos de tratamento de falhas de acesso no VOTE	48
7.1	Casos de falhas de acesso	57
7.2	Parâmetros de escrita e leitura	58
8.1	Tempos previsto e medido para multiplicação de matrizes . . .	73
8.2	Tempos para sobre-relaxação (consistência seqüencial)	84
8.3	Tempos para a versão com propagação	85
8.4	Tempos para a versão com envio direto de páginas	87

Lista de Figuras

2.1	Exemplo de desbalanceamento	11
2.2	Exemplo de sobrecarga em barreiras	12
5.1	Propagação de atrasos nos processadores	33
5.2	Atrasos sucessivos em um linha de processadores	35
5.3	Atraso efetivo para uma linha de 10 processadores	37
5.4	Atraso efetivo para uma linha de 900 processadores	37
5.5	Atraso efetivo de uma grade 5x5	38
5.6	Atraso efetivo de uma grade 30x30	38
5.7	Comparação entre uma linha 1x900 e uma grade 30x30	39
6.1	Memória compartilhada virtual	42
6.2	Modos de compartilhamento no <i>VOTE</i>	45
6.3	Detalhamento de uma falha de leitura	49
7.1	<i>Pipeline</i> no centro de consistência	56
7.2	Multiplicação de matrizes	62
8.1	Comparação entre os tempos previstos e medidos	74
8.2	Comparação entre as acelerações prevista e medida	75
8.3	Alterações no Código de Multiplicação de Matrizes	77
8.4	Aceleração com diferentes números de <i>advisors</i>	78
8.5	SOR : versão com consistência seqüencial	79
8.6	Consistência seqüencial	88
8.7	Modificações para propagação	89
8.8	Tempos da versão com propagação	90
8.9	Modificações para troca de mensagens	91
8.10	Tempos para a versão com envio direto de páginas	92

Capítulo 1

Introdução

Never underestimate the bandwidth of a station wagon full of tapes.

– Dr. Warren Jackson, Director, UTCS

Ambientes de computação paralela vêm se difundindo como uma alternativa viável onde o desempenho é vital. A um custo relativamente baixo, uma escolha adequada de equipamento e programas pode produzir resultados incomparáveis.

Um dos principais problemas encontrados atualmente no desenvolvimento e uso de programas e sistemas paralelos está relacionado à grande diversidade de estratégias e arquiteturas disponíveis. Isto se reflete em diversos planos, desde a escolha do ambiente de desenvolvimento ideal até a alocação de recursos financeiros e humanos para cada projeto.

Durante todo o ciclo de desenvolvimento de um projeto surgem questões relacionadas ao desempenho:

- Qual a melhor combinação entre equipamentos e estratégias de programação?
- Qual é o maior tamanho do problema que pode ser tratado pela combinação escolhida?

- A implementação de um programa em uma certa máquina é viável?
- Qual o desempenho esperado?
- Qual o desempenho obtido?
- Como melhorar este desempenho?

Estas perguntas só podem ser respondidas com uma descrição razoavelmente detalhada do comportamento do sistema.

1.1 Modelagem de desempenho

Normalmente o estudo do desempenho de uma aplicação se dá através do processo de medição-correção [11]. Neste processo iterativo, uma aplicação é instrumentada e tem sua execução monitorada de forma a medir os principais parâmetros de desempenho. No entanto, este método é insuficiente se se procura um entendimento mais profundo do que acontece durante a execução do programa, principalmente em se tratando de sobrecargas, como comunicação, tempo gasto com administração, etc.

Este entendimento é importante para que se possa definir estratégias de compilação [40], para se fazer uma escolha mais segura de uma configuração ou topologia e até mesmo para um refinamento da análise teórica de um algoritmo [16]. De pouco adianta se obter um conjunto detalhado de parâmetros de desempenho relacionados a sobrecargas se não se conhece a influência real, na aplicação, de cada uma delas.

A modelagem de desempenho procura descrever o funcionamento do programa ao invés de fornecer apenas dados numéricos sobre alguns aspectos pré-determinados. Isto pode ser obtido através de um simulador, de um estudo estatístico ou por meio de funções matemáticas que descrevem o funcionamento das componentes do programa (modelagem analítica)[5].

A abordagem que permite a descrição mais detalhada e que se adequa melhor ao estudo de situações hipotéticas é justamente a modelagem analítica. Infelizmente, um modelo analítico suficientemente preciso é com frequência complicado demais para ser usado. O grande desafio é a escolha do nível de detalhe suficientemente fino para que o modelo tenha precisão, mas que não seja tão fino a ponto de tornar o modelo resultante intratável.

Uma forma de evitar modelos analíticos demasiadamente detalhados é pela combinação com outras formas de modelagem [12]. Neste caso, o desafio passa

a ser a necessidade de manter a característica mais desejável dos modelos analíticos, ou seja, a possibilidade de avaliar situações diversas.

1.2 Objetivos do trabalho

O principal objetivo deste trabalho é explorar os recursos da modelagem analítica de desempenho em diversas situações e mostrar o potencial deste tipo de abordagem, ainda que algumas vezes aliado a outras técnicas, tanto na análise teórica de comportamento como na predição de desempenho de um sistema específico.

A capacidade de avaliação prévia, ou mesmo puramente teórica de aplicações, ganha importância maior quando se percebe que não são raros os casos onde o impacto da escolha de uma ou outra opção é surpreendente, em termos de desempenho.

1.2.1 Análise teórica de comportamento

Considere o caso de dois laços do tipo *DOALL* independentes e consecutivos. Em uma execução paralela pode-se optar pela alocação de todos os processadores em cada um dos laços, executando-se um após o outro. Outra possibilidade é executar os laços em paralelo, alocando um subconjunto dos processadores para cada um.

Embora aparentemente não exista diferença, em termos de desempenho, entre estas duas estratégias, a presença de sobrecargas pode alterar o desempenho final. Modelos teóricos e genéricos podem ser construídos de modo a auxiliar na decisão [26].

Outra situação analisada diz respeito a programas baseados em decomposição de domínio em máquinas com comunicação assíncrona. Estimar o tempo total de execução neste caso não é uma tarefa trivial, se o tempo de cada iteração em um processador não for constante. Aqui, novamente um modelamento teórico, auxiliado por simulações, é utilizado para escrever limites superior e inferior de tempo.

1.2.2 Memória compartilhada virtual

Uma área onde a compreensão do desempenho tem um papel central é na discussão das vantagens e desvantagens dos paradigmas de memória compar-

tilhada e distribuída.

Embora o paradigma de memória compartilhada seja atraente pela facilidade de programação que oferece [44, 4], existem basicamente dois problemas: a necessidade da administração do acesso concorrente à memória e a dificuldade na construção de máquinas com um número maior de processadores. Estes problemas são evitados em máquinas com memória distribuída mas com a necessidade de se fazer a comunicação explicitamente através de troca de mensagens.

Uma solução intermediária é o uso de memória compartilhada virtual, onde um sistema com memória distribuída usa troca de mensagens em um nível mais baixo, mas que se apresenta como memória compartilhada no nível de programação [38]. Isto implica em um controle rigoroso dos acessos a páginas de memória compartilhada, pois é preciso que todos os processadores concordem entre si com relação ao conteúdo da memória [13, 22].

A sobrecarga relacionada com a manutenção da consistência da memória, juntamente com o custo da camada de *software* que torna transparente a comunicação, tende a degradar o desempenho desta solução.

Para amenizar o custo da manutenção da consistência da memória, surgiram implementações mistas onde o controle sobre os acessos é relaxado, ficando o problema da coerência a cargo do programador [9, 10]. Nestas implementações, são definidos diversos modos ou modelos de compartilhamento, cada um com um custo próprio.

A escolha do melhor modo de compartilhamento e a estimativa do custo adicional imposto por cada estratégia é auxiliada em muito por modelos de desempenho. Portanto é natural o interesse em se produzir estes modelos de um modo mais geral.

Em alguns casos, como no sistema de memória compartilhada virtual *VOTE*[9], a produção de modelos é facilitada pela forma em que os modos de compartilhamento são implementados. No caso particular da implementação na máquina *MANNA* [6], os modelos de desempenho se tornam mais simples.

O que se buscou foi um método capaz de gerar modelos de desempenho para aplicações baseadas neste sistema, partindo-se apenas do código fonte e do conhecimento específico que o programador tenha *da aplicação em si*.

O modelo resultante deve ser capaz de descrever o comportamento da aplicação sobre cada modo de compartilhamento, com precisão apenas suficiente para permitir uma comparação confiável. Entretanto a precisão observada em experimentos mostrou-se melhor do que a esperada [24, 23].

1.3 Resumo do trabalho

O capítulo 2 define os conceitos básicos utilizados no trabalho e faz um breve apanhado dos métodos mais comuns de modelagem de desempenho.

No capítulo 3 são descritos trabalhos relacionados com esta tese.

O capítulo 4 estuda a estratégia de alocação de processadores para dois laços paralelos (tipo *DOALL*) consecutivos e independentes e descobre a melhor divisão na presença de sobrecargas.

O capítulo 5 mostra outro problema, onde a modelagem analítica é combinada com simulação para a previsão de desempenho de programas baseados em decomposição de domínio em máquinas com comunicação assíncrona.

Os três capítulos seguintes se concentram em sistemas de memória compartilhada virtual. No primeiro deles é feita uma discussão geral sobre os métodos de compartilhamento de memória e o sistema *VOTE-MANNA*, utilizado neste trabalho é introduzido, destacando-se as propriedades importantes para o desenvolvimento de modelos de desempenho.

O capítulo 6 constrói os modelos de desempenho para as funções fundamentais do sistema *VOTE-MANNA* e desenvolve um método para a geração de modelos analíticos de desempenho para aplicações desenvolvidas no mesmo.

No capítulo 7 são construídos modelos para algumas aplicações. Estes modelos são validados experimentalmente e os tempos medidos são comparados com aqueles previstos (capítulo 8).

As conclusões se encontram no capítulo 9.

Capítulo 2

Análise de Desempenho

É um fato conhecido que a velocidade e a capacidade de programas paralelos ainda são muito dependentes de sua implementação, seja por causa de uma escolha inadequada do algoritmo ou do método de paralelização, seja pela interação com o *hardware*, como por exemplo a influência de *cache* ou da representação de dados na memória [39, 28, 37, 47]. Com isto, pequenas variações no programa podem provocar grandes variações no tempo de execução, mesmo em sistemas que possuem um comportamento mais conhecido.

A compreensão mais detalhada do desempenho de um certo sistema é importante para a melhor identificação destes problemas, para melhorar a paralelização e para a construção de modelos capazes de prever o desempenho em outras situações (maior número de processadores, tamanho do problema, outra arquitetura).

A lista a seguir é um resumo do que se busca com o estudo do desempenho de um sistema paralelo, especialmente através de modelos:

- Verificação do uso eficiente do sistema, possivelmente com indicações de gargalos e otimizações. Na maioria dos casos não é claro se uma certa implementação está fazendo um bom uso do equipamento, pois existe um grande número de parâmetros que podem influenciar a execução do programa.
- Estudo da viabilidade de implementação de programas complexos. A construção de um programa paralelo eficiente muitas vezes implica em um grande investimento de tempo e equipamento. A possibilidade de uma análise prévia capaz de estimar a qualidade do resultado final pode ajudar na decisão da alocação (ou não) de recursos.

- Capacidade de prever o desempenho em situações diversas. Mesmo quando se possui um programa executando de modo eficiente em um certo equipamento, pode ser necessário prever seu comportamento quando alguns parâmetros são modificados. Exemplos incluem o número de processadores da máquina paralela, o tamanho do problema e a arquitetura.

Em algumas situações, é possível também identificar problemas de modelagem (*bugs*, por exemplo), quando o comportamento difere muito do esperado.

2.1 Definições

Nesta seção são apresentados e discutidos os principais conceitos relacionados à análise e predição de desempenho utilizados neste trabalho.

Definição 1 *Programa*: *É a implementação de um algoritmo e de uma estrutura de dados em uma linguagem de programação de alto nível e compilada para a máquina em estudo.*

Definição 2 *Sistema paralelo*: *É o conjunto formado pela máquina paralela em estudo, incluindo a estrutura da memória, o programa e o conjunto dos dados alimentados no programa.*

Como já foi dito, o desempenho é muito sensível à implementação. O sistema que está sendo estudado caracteriza-se pelo programa paralelo, a máquina onde ele é executado e pelo conjunto dos dados tratados pelo programa.

A idéia de desempenho está intimamente ligada ao tempo de execução de um programa. Desta forma, é natural que se procure estudar as relações entre os tempos de execução nas situações de interesse.

Um dos parâmetros básicos de desempenho é o tempo seqüencial, isto é o tempo que o programa gasta quando não está sendo executado em paralelo. Entende-se que esta execução se dá na máquina paralela em estudo. Em alguns casos, especialmente quando se estuda a eficiência de algoritmos, pode-se querer comparar o tempo de execução com n processadores e o tempo da melhor implementação seqüencial do algoritmo.

Definição 3 *Tempo seqüencial:* É o tempo gasto pelo programa quando executado em um único processador da máquina em estudo.

Notação: $T(1)$

A métrica mais tradicional para desempenho de sistemas paralelos, embora apresente alguns problemas [45, 27], é a aceleração, também conhecida como *speedup*:

Definição 4 *Aceleração* de n processadores é a relação entre o tempo seqüencial e o tempo gasto pelo programa quando executado com n processadores.

$$S(n) = \frac{T(1)}{T(n)}$$

Uma outra métrica, associada a esta e bastante utilizada é a *eficiência*:

Definição 5 *Eficiência:*

$$E(n) = \frac{T(1)}{n \cdot T(n)} = \frac{S(n)}{n}$$

Esta definição indica uma melhor utilização da máquina quando $E(n)$ for próxima a 1.

Poder-se-ia esperar que a aceleração de um programa paralelo, em condições ideais, fosse linear, isto é $S(n) = n$. Porém, foi observado em 1967 por Amdahl [3], que este valor não pode ser atingido por um programa normal, já que sempre existem partes intrinsecamente seqüenciais dentro do programa.

É possível então dividir o tempo de execução serial $T(1)$ em duas partes: t_s que é o tempo gasto pelos trechos do programa que devem ser executados em um único processador, e t_p , o tempo total gasto por trechos que *podem*, em princípio, ser executados por mais de um processador [14].

Neste caso, $T(n)$ é dado por

$$T(n) \geq t_s + \frac{t_p}{n} \quad (2.1)$$

e a aceleração se escreve da seguinte forma:

$$S(n) \leq \frac{T(1)}{t_s + \frac{t_p}{n}} \quad (2.2)$$

onde se percebe a inexistência da linearidade devida a t_s . Esta perda de desempenho é conhecida como *paralelismo insuficiente* [11], e motiva a definição do grau de paralelismo de um trecho de programa [45]:

Definição 6 *Grau de paralelismo* é o número máximo de processadores que podem estar computacionalmente ativos em um dado instante de tempo, durante a execução de um programa.

No entanto, a expressão 2.2 não leva em conta que o grau de paralelismo pode assumir valores entre 1 e n . Se t_k indicar o tempo gasto em trechos com grau k , a expressão 2.1 deve ser reescrita como segue:

$$T(n) = t_s + \sum_{k=2}^n \frac{t_k}{k} \quad (2.3)$$

Ainda assim, existem outros fatores que podem provocar a degradação do desempenho, conhecidos coletivamente como sobrecarga (*overhead*).

Definição 7 *Sobrecarga* é o nome genérico para as fontes de degradação do desempenho de uma particular implementação.

Como será discutido na próxima seção, a sobrecarga aumenta com o número de processadores, na maioria dos casos [14]. O tempo gasto com sobrecargas será designado por $T_o(n)$, e a equação mais completa para $T(n)$ é

$$T(n) = t_s + \sum_{k=2}^n \frac{t_k}{k} + T_o(n) \quad (2.4)$$

Um programa paralelo pode ser decomposto em vários trechos, caracterizados pela existência de uma sincronização global:

Definição 8 *Uma era* é uma parte da execução do programa entre dois pontos de sincronização de todos os processadores.

Definição 9 *Caminho crítico* é a linha de execução formada pela união das linhas de execução, em cada era, do processador mais lento.

O início e o término de um programa também limitam eras.

2.2 Sobrecargas

Sendo a principal fonte de degradação de desempenho, as sobrecargas são de importância fundamental tanto na análise como na predição de desempenho. Na verdade pode-se dizer que o principal objetivo da análise de desempenho

é a identificação e quantificação das sobrecargas presentes no programa em questão.

As sobrecargas podem ser classificadas de acordo com sua origem. As principais fontes são as seguintes:

- Comunicação
- Desbalanceamento na carga
- Contenção de recursos
- Barreiras de sincronização

O impacto de cada uma delas depende muito da aplicação, da implementação e do equipamento utilizado. As características de cada uma delas são apresentadas nas seções seguintes.

2.2.1 Comunicação

Este é provavelmente o tipo mais importante de sobrecarga, pois a menos que o programa seja uma coleção de tarefas *completamente* independentes, sempre existirá comunicação entre os processadores.

A comunicação pode ser síncrona ou assíncrona. No primeiro caso, o processador que está enviando a mensagem espera até que o recipiente a tenha recebido. Isto não ocorre no último caso, desde que as filas de mensagem tenham tamanho suficiente. Em ambas as situações podem ocorrer *deadlocks* [30].

O tempo gasto com comunicação depende dos seguintes fatores:

- Rede de interconexão, que inclui elementos como taxa de transferência, tempo de preparação do canal de comunicação (latência) e distância entre os processadores.
- Número de transferências.
- Volume de dados transmitidos.

2.2.2 Desbalanceamento de carga

Mesmo quando existe paralelismo suficiente nem sempre é possível manter todos os processadores ocupados permanentemente pois pode ocorrer que alguns fiquem ociosos esperando que outros terminem uma tarefa. Nesta situação, em alguns casos pelo menos, deveria ser possível melhorar o desempenho com uma distribuição melhor do trabalho.

Pela sua frequência e dificuldade de identificação, esta é uma das principais fontes de degradação. A figura 2.1 mostra um exemplo onde um código idêntico em dois processadores podem causar um desbalanceamento de carga, se os dados não forem favoráveis. Neste exemplo um vetor A é manipulado dentro de um laço com iterações independentes que é distribuído entre dois processadores idênticos. Porém se existirem mais elementos de A maiores do que K para índices acima de 100 do que para índices entre 1 e 100 e se f1 tomar mais tempo do que f2, apesar do laço ser um *DOALL*¹, o tempo não cai pela metade pois o processador P1 fica ocioso enquanto P2 trabalha na computação mais lenta. Para um outro vetor A ou uma outra distribuição do laço (por exemplo, índices pares para P1 e ímpares para P2) a aceleração poderia ser maior.

Um exemplo mais sutil pode ser encontrado no capítulo 4.

Processador P1	Processador P2
sincronização	sincronização
DOALL i = 1 to 100	DOALL i = 101 to 200
if A[i] > K then	if A[i] > K then
f1(A[i])	f1(A[i])
else	else
f2(A[i])	f2(A[i])
endif	endif
ENDDO	ENDDO
sincronização	sincronização

Figura 2.1: Exemplo de desbalanceamento

¹Em um laço do tipo *DOALL* as iterações são independentes, podendo ser executadas em paralelo.

2.2.3 Contenção

A contenção ocorre quando dois ou mais processos buscam um recurso não compartilhável simultaneamente. Exemplos de recursos deste tipo são canais de comunicação, os próprios processadores e memória. Conseqüentemente ela afeta outros tipos de sobrecarga, tornando os modelos mais complicados.

Dependendo do tipo, a sobrecarga pode ser grande, pois a contenção implica na serialização de operações, eventualmente com um custo administrativo adicional.

Outro fator importante é que, a menos de situações especiais, a contenção não tem um caráter regular, já que depende da coincidência nas requisições dos recursos. O problema tratado no capítulo 5 ilustra a complexidade da modelagem deste tipo de sobrecarga.

2.2.4 Barreiras

Dependendo do tipo de aplicação, mas principalmente no caso de memória compartilhada, é necessário garantir que todos os processadores tenham atingido o mesmo ponto de execução dentro do programa.

Em algumas situações esta sincronização é decorrente do próprio padrão de comunicação usado pelo programa, mas em outras ela deve ser realizada explicitamente. Isto é feito através de barreiras de sincronização e sua implementação envolve comunicação e algoritmos especializados.

Figura 2.2: Exemplo de sobrecarga em barreiras

Na prática o que acontece é que ao atingir o ponto de sincronização o

processador entra em um estado de espera até receber um sinal avisando que ele pode prosseguir. Este sinal é disparado para todos os processadores assim que o último atingir a barreira, como ilustra a figura 2.2.

Como se pode ver, estão envolvidos custos de comunicação e desbalanceamento (uma vez que alguns processadores ficam ociosos). Esta fonte de sobrecarga foi destacada por ser freqüente e também por permitir uma modelagem em separado.

Se não se considerar como parte da barreira o custo relativo ao desbalanceamento de carga, o custo se reduz basicamente ao tempo de reativação dos processadores que em geral é bem pequeno e tem modelagem simples. Nos casos de memória compartilhada, este custo é importante devido à freqüência com que sincronizações são necessárias.

2.3 Modelagem

A modelagem de programas surge como uma alternativa ao método iterativo de medição-modificação [11, 32], que pressupõe um sistema já implementado e que deve ser aperfeiçoado.

Embora o método de medição-modificação não deva ser descartado, pois serve como validação dos modelos, ele não é suficiente para o entendimento completo do que ocorre com a aplicação, pois a instrumentação correta do programa já implica na compreensão mais profunda da execução do programa.

Os modelos de desempenho, por outro lado, tentam capturar o funcionamento do programa baseando-se em sua estrutura e formas de interação entre as diversas partes. A criação destes modelos, no entanto, oferece diversas dificuldades, principalmente no caso paralelo. A seguir são apresentadas propriedades de sistemas seqüenciais e paralelos que influenciam a construção de modelos:

Caso seqüencial:

- O tempo está diretamente ligado ao número de instruções executadas. Se todas as instruções têm a mesma duração, o tempo total é proporcional ao próprio número. Caso contrário, pode-se dividir as instruções em classes definidas pela duração e somar os números de instruções de cada classe ponderados pela duração correspondente.
- Existem poucas sobrecargas. São basicamente aquelas relacionadas ao sistema, onde não se possui controle, e aquelas relacionadas a memória

cache.

- A instrumentação do código é relativamente simples assim como as medições de tempo.
- Como consequência da primeira propriedade, o tempo total é a soma dos tempos das partes do programa.

Caso paralelo:

- O tempo de execução é dado pelo tempo do caminho crítico.
- A quantidade de sobrecargas significativas é muito maior.
- Dependência forte da arquitetura.
- Modelos analíticos tendem a ser muito complexos. A execução simultânea de instruções com tempos diferentes e a existência de sobrecargas torna a modelagem analítica distante da análise teórica.
- A instrumentação do programa e medições são mais delicadas. Deve-se garantir que o custo adicional relacionado a estas operações seja o menor possível e igual para todos os processadores. Caso contrário pode-se afetar o comportamento de modo a esconder o efeito de alguma sobrecarga, como por exemplo, desbalanceamento de carga.

Um modelo de desempenho, além de tratar dos elementos acima, deve satisfazer algumas condições para ser útil:

Precisão O modelo deve ser suficientemente preciso, de modo que os resultados façam sentido. O grau de precisão pode depender da finalidade do modelo. Por exemplo, se se busca apenas uma estimativa da aceleração, os valores exatos dos tempos de execução são menos importantes.

Custo Deve ser possível gerar o modelo em menos tempo do que produzir e executar o programa em estudo, mesmo no caso em que se estuda uma aplicação já existente.

Modularidade Embora não seja vital, é interessante poder aumentar o modelo de forma incremental, incluindo o tratamento de novos parâmetros sem a necessidade de reconstrução total.

Fidelidade O modelo deve descrever fielmente o comportamento da aplicação, pela descrição da aplicação em si ou da evolução dos seus parâmetros.

Existe mais de uma forma de classificar os modelos de desempenho. A classificação mais simples se baseia no tipo de abordagem utilizada. As abordagens mais comuns são discutidas nas seções seguintes.

2.3.1 Modelagem analítica

Modelos analíticos fazem uso de um conjunto de equações e fórmulas para descrever o funcionamento da máquina e da aplicação de um modo abstrato.

Um modelo deste tipo é em geral barato do ponto de vista computacional e pode possuir uma grande modularidade, permitindo seu transporte para outras arquiteturas e situações.

No entanto a modelagem de sistemas muito complexos pode requerer a utilização e tratamento de um número muito grande de parâmetros e funções, tornando o modelo intratável. Neste caso grandes simplificações devem ser feitas prejudicando a precisão. Este é o caso de comportamentos dinâmicos, como alocação de processadores durante a execução, algoritmos adaptativos, etc.

Outro problema potencial é que a geração destes modelos exige um conhecimento mais profundo da aplicação e da arquitetura.

Em resumo, a construção de um modelo analítico deve levar em conta dois pontos: o nível de detalhe a ser modelado (e conseqüentemente sua precisão) e a tratabilidade do modelo.

2.3.2 Modelagem estatística

Neste tipo de abordagem as funções dos modelos analíticos são substituídas por distribuições que fornecem o comportamento esperado dos diversos parâmetros. Desta forma o comportamento exato não precisa ser conhecido, mas perde-se precisão.

A principal vantagem é que se ganha flexibilidade com o modelo, mas por outro lado estes modelos fornecem um comportamento assintótico e algumas ferramentas são custosas demais. Redes de Petri, por exemplo, embora sejam úteis para a verificação da estabilidade e algumas outras propriedades, são inviáveis para a análise do tempo de execução de um sistema complexo.

2.3.3 Simulação

Dentro de um simulador se possui um controle maior dos diversos parâmetros de execução e portanto é mais fácil obter informações para construir e validar modelos de desempenho.

É possível a construção de um simulador específico, a emulação de uma arquitetura em outra, a utilização de traços de execução [35], ou ainda a utilização de informações fornecidas pelo sistema de compilação [12] (neste último caso combinadas com modelos analíticos).

Os maiores problemas com esta abordagem são o alto custo e a necessidade de um conhecimento detalhado do equipamento a ser simulado. Com isto a precisão não é sempre satisfatória.

2.3.4 Empírica

Modelos empíricos se baseiam em medições de trechos de programas que representam as operações mais comuns como por exemplo *broadcasting*, sincronização de barreiras, chamada remota de procedimento, criação de canais de comunicação virtual, etc. Desta forma, obtêm-se diretamente os parâmetros necessários, evitando a construção de modelos que descrevam os níveis mais básicos da aplicação[5].

O maior problema deste método é a dependência no modo de construção da aplicação, isto é, o tipo da máquina, as bibliotecas e o compilador utilizado.

Uma variação desta abordagem são os modelos dinâmicos, que usam informações retiradas de uma execução do programa para completar o modelo. Neste caso, o modelo se presta mais à análise de desempenho do que à predição.

2.3.5 Modelos híbridos

Nada impede a combinação das diversas abordagens de modo a construir um modelo híbrido, que tenta minimizar as desvantagens de cada método.

Em [12], é apresentada uma ferramenta de predição e análise de desempenho que faz uso de três abordagens diferentes: parâmetros fundamentais são obtidos a partir de uma execução simplificada do programa e modelos analíticos são acoplados a uma simulação de modo a obter o modelo final.

Outra estratégia, mais geral, é usada por [11] onde o programa é instrumentado e uma série de execuções simples são feitas. O resultado das

diversas medições é comparado com modelos analíticos que representam os comportamentos mais freqüentes, através de regressões.

2.3.6 A modelagem usada neste trabalho

O objetivo principal é a construção de modelos com uma boa capacidade de descrição, o que torna a modelagem analítica uma escolha natural. No entanto, outras abordagens são utilizadas em conjunto com a analítica de modo a obter um modelo final mais simples.

O capítulo 5 faz uso de simulações para estimar o tempo de execução em grades de processadores com comunicação assíncrona.

No capítulo 7 são gerados modelos analíticos dependentes de parâmetros que são obtidos a partir de experimentos específicos (abordagem empírica).

Capítulo 3

Trabalhos Relacionados

O interesse em análise e previsão de desempenho vem crescendo e existe um número cada vez maior de trabalhos nesta direção, tanto do ponto de vista teórico como no estudo de implementações específicas e de métodos semi-automáticos de geração de modelos.

As próximas seções descrevem os trabalhos mais recentes e relacionados com esta dissertação.

3.1 Predição automática de desempenho

A seção 3.1.1 mostra uma ferramenta automática para a criação de modelos de desempenho. O trabalho apresentado nesta dissertação foi em parte inspirado naquele.

As seções 3.1.2 e 3.1.3 apresentam ferramentas e métodos que evitam o uso direto de modelos analíticos. Ao contrário do que é apresentado aqui (capítulo 7), em ambos os casos o processo depende fortemente de uma execução monitorada.

3.1.1 P^3T : Parallel Performance Prediction Tool

Em [12], Fahringer descreve uma ferramenta de predição de desempenho para o *Vienna Fortran Compilation System* (*VFCS*), um sistema de compilação paralela que evoluiu do *SUPERB*[21, 17, 18, 29, 19, 20]. Partindo de uma execução sequencial de uma parte do programa, são obtidos parâmetros fundamentais que são posteriormente utilizados na produção final de modelos.

Exemplos destes parâmetros são a taxa de desvio em condicionais (*true ratio*), a frequência de execução de blocos e comandos e o número de iterações em laços. Estas medições são feitas em cima de uma versão especialmente instrumentada e otimizada do programa original.

O programa paralelo é então simulado a partir do grafo de execução gerado pelo compilador e um novo conjunto de valores, específicos da execução em paralelo, são estimados. Estes valores alimentam uma série de modelos analíticos pré-definidos, para cada uma das principais sobrecargas.

O valores modelados são os seguintes:

- Distribuição de carga nos processadores.
- Número total de mensagens trocadas.
- Volume de dados transferidos.
- Tempo total gasto em comunicação.
- Contenção na rede.

Os resultados são bastante satisfatórios e servem de guia para que o compilador verifique ou procure novas otimizações.

Esta ferramenta é no entanto específica para o *VFCS* e se baseia exclusivamente em decomposição de domínios [17].

3.1.2 Lost Cycles Toolkit

Em [11] é proposto um formalismo de caracterização do desempenho de uma aplicação baseado nos estados de operação em cada processador. Esta caracterização é descrita por “predicados de desempenho”.

Estes predicados são escolhidos de tal modo que sejam incluídas todas as formas de degradação de desempenho e que sejam mutuamente exclusivos. Com eles é possível a identificação de estados onde não existe nenhum trabalho útil sendo realizado. O tempo gasto nestes estados é chamado de “ciclos perdidos” (*lost cycles*).

Como os predicados são construídos a partir de elementos monitoráveis, o tempo gasto nos diversos estados pode ser obtido diretamente a partir da instrumentação e da execução controlada do programa.

Com os dados obtidos experimentalmente, são construídas tabelas e os modelos de desempenho que melhor se adaptam são identificados através de

regressão. Quando não é possível decidir qual o melhor modelo, o usuário é avisado para que sejam feitas novas experiências ou sugeridas alternativas.

Este método, embora possua um nível de automação razoável e consiga identificar e prever gargalos, não consegue fornecer a razão ou origem de uma degradação de desempenho.

3.1.3 Carnival e Waiting Time Analysis

Meira procura resolver o problema do *Lost Cycles Toolkit*, utilizando uma monitoração mais detalhada e com uma interação maior com o usuário. O trabalho — descrito em [32], [34] e principalmente em [33] — está em fase de desenvolvimento.

O ponto de partida é uma ferramenta de monitoração mais sofisticada, chamada *Carnival*, que parte de diversos níveis de informação: o programa fonte, execução monitorada e a organização dos dados de desempenho de acordo com a estrutura do programa.

Durante a monitoração, os eventos selecionados são gravados em um arquivo de acompanhamento para análise posterior. Cada evento é caracterizado pelo processador que o originou, escopo, categoria de processamento e instante em que ocorreu.

As categorias de processamento permitem armazenar o tipo de trabalho sendo executado e incluem:

- computação local
- acesso a dado externo
- chamadas ao sistema
- comunicação
- barreiras
- etc.

Toda informação produzida por *Carnival* é utilizada para a identificação de estados de espera em cada processador. Para trabalhar com estes resultados é sugerido um método de manipulação das linhas de execução de modo a se obter uma descrição minimal das origens de atraso.

O *Lost Cycles Toolkit* pode ser incluído no sistema para a criação de modelos para cada trecho e posterior integração.

3.2 Outros trabalhos

Sun e Zhu tratam da modelagem de sistemas com memória compartilhada virtual em [46] e estudam o desempenho em detalhe para a *KSR-1*, mas sua ênfase é a utilização de uma nova métrica de aceleração, definida em [45].

Chandra, Larus e Rogers [8] fazem uma comparação entre troca de mensagens e compartilhamento de memória para quatro aplicações distintas, usando simulação. Embora os autores consigam ilustrar os resultados, a precisão dos dados é insuficiente.

O trabalho descrito no capítulo 4, de nossa autoria [26], é uma extensão do tratamento que Polychronopoulos [40] faz do problema de paralelização de laços do tipo *DOALL* e sua otimização.

Capítulo 4

Influência de Sobrecargas na Alocação de Processadores para Laços Independentes

4.1 Introdução

A alocação de processadores para a execução em paralelo de um laço onde as iterações são independentes (*DOALL*) é tratado em [40]. Quando todas as iterações têm o mesmo tamanho o problema é trivial, pois basta dividi-las igualmente entre os processadores.

Entretanto, este pode não ser o caso quando se têm dois laços independentes deste tipo, um após o outro. Não é claro se é melhor usar todos os processadores para cada um deles ou se se deve alocar subconjuntos distintos e executar os dois laços em paralelo.

Este capítulo contribui com um aprofundamento desta discussão [26]. Será visto que a presença de sobrecarga pode influenciar a escolha da melhor estratégia de modo não trivial. Isto pode ser esperado porque as sobrecargas não são em geral paralelizáveis, provocando uma aceleração sub-linear na maioria dos casos [15, 14, 1].

4.2 Definição do problema

Considera-se o problema da alocação de n processadores para a execução de dois laços *DOALL* consecutivos e independentes.

A definição de um *laço DOALL* é dada a seguir.

Definição 10 *Laço DOALL* é um laço onde cada iteração é completamente independente de todas as outras.

Assim, para um laço *DOALL*, as iterações podem ser executadas em qualquer ordem ou em paralelo.

A situação em estudo é esta:

```
DOALL I = 1, m1
    S1;
    S2;
    ⋮
    Ss
ENDDO
DOALL I = 1, m2
    R1;
    R2;
    ⋮
    Rr
ENDDO
```

Como é suposto que os dois laços são independentes, qualquer um pode ser executado primeiro, ou os dois podem ser executados em paralelo.

Existem basicamente duas estratégias de alocação;

1. Alocam-se n processadores para executar o primeiro laço e em seguida os mesmos n processadores para executar o segundo. Esta estratégia será denominada de execução consecutiva.
2. Particionam-se os n processadores em dois grupos de n_1 e n_2 processadores cada, de forma que

$$n = n_1 + n_2$$

e executa-se o primeiro laço com n_1 e o segundo com n_2 processadores, simultaneamente. Esta estratégia será denominada de execução simultânea.

Pergunta-se em quais situações uma é preferível à outra.

Um caso particular deste problema é o seguinte, com um único laço *DOALL*.

```
DOALL I = 1 to m1 + m2
    S1;
    S2;
    ⋮
    Sk
ENDDO
```

No que concerne à semântica, qualquer laço *DOALL* pode ser visto como dois laços consecutivos e independentes, e uma solução para o caso geral se aplica.

4.3 Tempos e estratégias de execução

Cada uma das estratégias propostas na seção anterior será analisada considerando-se a presença ou não de sobrecarga. Existem portanto quatro situações distintas e procura-se a condição de tempo mínimo para cada uma delas.

No que segue, valem as seguintes convenções:

1. O número total de processadores é n .
2. Deseja-se usar todos os processadores.
3. Nos casos onde os laços são executados concorrentemente:
 - αn é o número de processadores utilizados no primeiro laço.
 - βn é o número de processadores utilizados no segundo laço.

Pela hipótese anterior, tem-se $\alpha + \beta = 1$

4. A equação de tempo do laço i ($i \in \{1, 2\}$) é dada por

$$t_i(n) = t_{s_i} + t_{o_i}(n) + \frac{t_{p_i}}{n}$$

onde, seguindo a notação introduzida no capítulo 2,

- t_{s_i} é o tempo de inicialização. (Em um laço *DOALL* “puro” esta componente é muitas vezes inexistente ou desprezível, dependendo da implementação, mas ela será mantida de modo a deixar a análise mais geral.)
- $t_{o_i}(n)$ é uma função que descreve o tempo gasto com sobrecargas.
- $\frac{t_{p_i}}{n}$ corresponde à parte paralelizável do laço.

4.3.1 Casos sem sobrecarga

Neste caso o tempo de execução é

$$t(n) = t_s + \frac{t_p}{n}$$

(a) Execução consecutiva dos dois laços, cada um com todos os n processadores

Esta é a opção usual[40]. O tempo total é dado por

$$\begin{aligned} T_I &= t_{s_1} + \frac{t_{p_1}}{n} + t_{s_2} + \frac{t_{p_2}}{n} \\ &= t_{s_1} + t_{s_2} + \frac{t_{p_1} + t_{p_2}}{n} \end{aligned} \quad (4.1)$$

(b) Execução simultânea dos dois laços, cada qual com uma fração dos processadores

O tempo é o seguinte¹

$$\begin{aligned} T_{II} &= \max \left(t_{s_1} + \frac{t_{p_1}}{\alpha n}, t_{s_2} + \frac{t_{p_2}}{\beta n} \right) \\ &= \max (t_1(\alpha n), t_2(\beta n)) \end{aligned} \quad (4.2)$$

¹É possível obter um tempo ainda melhor, se for usada esta partição de processadores durante $|t_{s_1} - t_{s_2}|$ e todos os n trabalhando no laço mais lento após $\min(t_1(n), t_2(n))$. Esta opção, entretanto, não será considerada.

Comparação

A condição para que a execução simultânea (caso (b)) seja mais vantajosa do que o caso (a) pode ser obtida impondo-se $T_I \geq T_{II}$:

$$t_{s_1} + t_{s_2} + \frac{t_{p_1} + t_{p_2}}{n} \geq t_1(\alpha n) \quad (4.3)$$

$$t_{s_1} + t_{s_2} + \frac{t_{p_1} + t_{p_2}}{n} \geq t_2(\beta n) \quad (4.4)$$

Desenvolvendo-se (4.3),

$$t_{s_2} + \frac{t_{p_1} + t_{p_2}}{n} \geq \frac{t_{p_1}}{\alpha n} \quad (4.5)$$

$$t_{s_2} \geq \frac{\beta t_{p_1} - \alpha t_{p_2}}{\alpha n} \quad (4.6)$$

para (4.4), obtém-se

$$t_{s_1} + \frac{t_{p_1} + t_{p_2}}{n} \geq \frac{t_{p_2}}{\beta n} \quad (4.7)$$

$$t_{s_1} \geq \frac{\alpha t_{p_2} - \beta t_{p_1}}{\beta n} \quad (4.8)$$

Supondo $t_{s_2} \ll \frac{t_{p_1} + t_{p_2}}{n}$, pode-se ignorar o lado esquerdo de (4.6), e obtém-se

$$\frac{\alpha}{\beta} \geq \frac{t_{p_1}}{t_{p_2}} \quad (4.9)$$

Analogamente, se $t_{s_1} \ll \frac{t_{p_1} + t_{p_2}}{n}$,

$$\frac{\alpha}{\beta} \geq \frac{t_{p_2}}{t_{p_1}} \quad (4.10)$$

De (4.9) e (4.10), conclui-se que

$$\frac{t_{p_1}}{t_{p_2}} = \frac{\alpha}{\beta} \quad (4.11)$$

Se $t_{p_1} = t_{p_2}$, vale $\alpha = \beta$.

No caso particular onde $t_{s_1} = t_{s_2}$, α assume um valor simples:

$$\alpha = \frac{t_{p_1}}{t_{p_1} + t_{p_2}}$$

e pode-se verificar facilmente que o tempo é igual ao tempo da execução consecutiva.

Sob estas condições é indiferente o uso de uma ou outra estratégia. Entretanto não é verdade se houver uma diferença nas componentes sequenciais dos dois laços.

4.3.2 Casos com sobrecarga

O mesmo tipo de análise pode ser repetido para o caso com sobrecargas. O tempo de execução consecutivo é

$$\begin{aligned} T_I &= t_{s_1} + t_{o_1}(n) + \frac{t_{p_1}}{n} + t_{s_2} + t_{o_2}(n) + \frac{t_{p_2}}{n} \\ &= t_{s_1} + t_{s_2} + t_{o_1}(n) + t_{o_2}(n) + \frac{t_{p_1} + t_{p_2}}{n} \end{aligned} \quad (4.12)$$

O tempo para a execução simultânea é

$$\begin{aligned} T_{II} &= \max \left(t_{s_1} + t_{o_1}(\alpha n) + \frac{t_{p_1}}{\alpha n}, t_{s_2} + t_{o_2}(\beta n) + \frac{t_{p_2}}{\beta n} \right) \\ &= \max(t_1(\alpha n), t_2(\beta n)) \end{aligned} \quad (4.13)$$

Supondo, para simplificar a análise e sem perda de generalidade, que vale

$$t_1(\alpha n) \geq t_2(\beta n)$$

a execução simultânea é preferível quando $T_I \geq T_{II}$:

$$\begin{aligned} t_{s_2} + t_{o_1}(n) + t_{o_2}(n) + \frac{t_{p_1} + t_{p_2}}{n} &\geq t_{o_1}(\alpha n) + \frac{t_{p_1}}{\alpha n} \\ t_{s_2} + t_{o_1}(n) + t_{o_2}(n) - t_{o_1}(\alpha n) &\geq \frac{t_{p_1} - \alpha(t_{p_1} + t_{p_2})}{\alpha n} \end{aligned} \quad (4.14)$$

No caso típico onde a sobrecarga cresce com n , observa-se imediatamente que o lado esquerdo da desigualdade (4.14) cresce com n ao passo que o lado direito decresce, qualquer que seja o valor de α .

A conclusão é que, com sobrecarga crescente com n , a desigualdade (4.14) é sempre verificada para um valor suficientemente grande do número de processadores. Portanto, sob estas condições e quando o número de processadores for grande, a execução simultânea dos dois laços, cada um com uma fração dos processadores é melhor do que a execução usual, consecutiva.

4.4 Casos particulares

É interessante analisar a desigualdade (4.14) com modelos particulares de sobrecarga e a seguir verificar o caso de um único laço *DOALL* grande.

4.4.1 Modelo constante

Este é o modelo mais simples. A sobrecarga não varia com o número de processadores. Observe que neste caso a observação ao final da última seção não se aplica.

A condição que favorece a execução simultânea pode ser obtida reescrevendo-se (4.14) e usando $t_{o_1}(n) = t_{o_1}(\alpha n) = t_{o_1}$ e $t_{s_2}(n) = t_{s_2}$:

$$t_{s_2} \geq \frac{t_{p_1} - \alpha(t_{p_1} + t_{p_2})}{\alpha n} - t_{s_2} \quad (4.15)$$

Deve-se lembrar que está sendo suposto $T_I \geq T_{II}$. Foi obtido um limite inferior para a sobrecarga do laço mais rápido com esta escolha da distribuição dos processadores. Obviamente pode-se considerar o problema inverso e determinar α de modo a satisfazer (4.15), de um modo análogo ao da seção 4.3.1.

4.4.2 Modelo linear

No modelo linear a sobrecarga é $t_o(n) = an + b$. Este é um modelo muito comum, especialmente quando a sobrecarga é dominada por comunicação.

O lado esquerdo de (4.14) fica:

$$t_{s_2} + t_{o_1}((1 - \alpha)n) + t_{s_2}(n) - b$$

Seja $t_{o_i}(n)$ escrito como $a_i n + b_i$. Como os parâmetros a_i e b_i em geral dependem apenas da máquina, pode-se supor $a_1 = a_2 = a$ e $b_1 = b_2 = b$.

Assim

$$t_{o_1}((1 - \alpha)n) + t_{o_2}(n) = (2 - \alpha)an + 2b$$

e (4.14) fica

$$t_{s_2} + (2 - \alpha)an + 2b \geq \frac{t_{p_1} - \alpha(t_{p_1} + t_{p_2})}{\alpha n}$$

4.4.3 Um único laço

Aqui é considerado o caso de um único laço e são apresentadas as condições sob as quais é melhor quebrá-lo em dois laços menores, do mesmo tamanho, para serem executados simultaneamente.

Esta questão é pertinente se existir alguma espécie de sobrecarga, como desbalanceamento, acesso de dados externos aos processadores, etc.

Para um laço deste tipo, tem-se

$$\begin{aligned} t_{s_1} &= t_{s_2} = 0 \\ t_{p_1} &= t_{p_2} = t_p \\ t_{o_1} &= t_{o_2} = t_o \end{aligned}$$

A desigualdade (4.14) fica muito simples:

$$2t_o(n) - t_o(\alpha n) \geq \frac{t_p(1 - 2\alpha)}{\alpha n}$$

Pode-se observar que mesmo sob o modelo constante, o lado esquerdo pode ser maior do que o lado direito com um número não muito grande de processadores.

Capítulo 5

Decomposição de Domínio com Comunicação Assíncrona

Mesmo uma situação aparentemente simples como decomposição de domínio pode oferecer sérios desafios para a construção de modelos puramente analíticos [2]. A contribuição deste capítulo é apresentar uma visão original do problema.

O caso discutido a seguir não oferece dificuldades à primeira vista, mas sua modelagem é muito difícil, sendo necessário que se recorra a técnicas como simulação e estudos qualitativos.

A próxima seção apresenta o caso síncrono, mais simples. O caso assíncrono é discutido em seguida.

5.1 Comunicação síncrona

Tipicamente um programa baseado em decomposição de domínios tem como parte central um laço com o seguinte formato:

```
DO I = 1, N
    <cálculo>
    <troca de informações>
ENDDO
```

Este laço é executado em cada um dos P processadores, cada qual com sua região do domínio.

A parte de troca de informações corresponde ao envio e recebimento dos dados de fronteira de cada região.

O caso mais simples é quando a duração do cálculo independe do processador e da iteração. Neste caso, se T_{calc} for o tempo relativo a computação e T_{troca} o tempo gasto com a comunicação em uma iteração, o tempo total paralelo é dado simplesmente por:

$$T_{par} = N \cdot (T_{troca} + T_{calc})$$

O número de total de cálculos é dado pelo produto do número de iterações pelo número de processadores: $N \cdot P$.

O tempo de execução seqüencial é, portanto:

$$T_{seq} = N \cdot P \cdot T_{calc}$$

Neste caso simples, não faz muita diferença se a comunicação é síncrona ou não, pois todos os processadores estão gastando o mesmo tempo. O mesmo não acontece quando a duração do cálculo varia a cada iteração, como por exemplo em uma situação do tipo:

```
DO I = 1, N
  IF(cond) THEN
    <cálculo 1>
  ELSE
    <cálculo 2>
  ENDIF
  <troca de informações>
ENDDO
```

onde o valor da expressão *cond* é variável e as durações de <cálculo 1> e <cálculo 2> diferem.

Chamando a duração de <cálculo 1> de C_1 e a de <cálculo 2> de C_2 e supondo, sem perda de generalidade, que $C_1 > C_2$, pode-se dizer que há um 'atraso' de tamanho $\delta = C_1 - C_2$ cada vez que *cond* se mostrar verdadeira.

De um modo um pouco mais geral, será dito que um processador poderá sofrer um atraso com probabilidade α a cada cálculo efetuado. No exemplo acima, α corresponde ao *true ratio* do condicional.

No restante deste capítulo, considera-se que tanto α como o tempo de comunicação é independente do processador e da iteração.

Descobrir o tempo total nesta situação é equivalente a descobrir o atraso total do sistema.

No caso seqüencial, se não houver atrasos, o tempo de execução é simplesmente $N \cdot P \cdot C_2$. Com atrasos, o tempo esperado de execução é dado por:

$$\langle T_{seq} \rangle = N \cdot P \cdot (C_2 + \alpha \delta)$$

A distribuição de probabilidade de atrasos é uma binomial:

$$\mathcal{P}(m \text{ atrasos}) = B(m, N, \alpha) = \binom{N}{m} \alpha^m (1 - \alpha)^{N-m}$$

Se a comunicação for síncrona, em cada iteração existe uma sincronização forçada e seu tempo é o do processador mais lento. Em outras palavras a probabilidade de atraso em uma iteração é igual à probabilidade de atraso de pelo menos um processador, $(1 - \text{probabilidade de nenhum atraso})$:

$$\mathcal{P}(\text{atraso em uma iteração}) = \rho = 1 - (1 - \alpha)^P \quad (5.1)$$

e o tempo esperado da execução paralela é

$$\langle T_{par} \rangle = N \cdot (T_{troca} + (C_2 + \rho \cdot \delta))$$

A distribuição de probabilidade para o número de iterações atrasadas, $\mathcal{A}$, é novamente uma binomial:

$$\mathcal{A}_{sinc}(m) = B(m, N, \rho)$$

5.2 Comunicação assíncrona

No caso anterior, um atraso em um processador provoca o atraso de toda uma iteração. Se a comunicação for assíncrona, entretanto, processadores diferentes podem estar executando diferentes instâncias da iteração ao mesmo tempo e o número de atrasos é diferente para cada um, como está ilustrado na figura 5.1.

Na figura 5.1 os quadros apresentam o número de atrasos de cada processador de uma grade 5×5 em passos sucessivos. Em cada instante, os processadores que sofreram um atraso espontâneo estão sublinhados. Até o estágio (f), quatro processadores atrasaram espontaneamente mas o atraso total foi apenas de três.

Figura 5.1: Propagação de atrasos nos processadores

Como durante a fase de troca de informações cada processador deve esperar pelos seus vizinhos, os atrasos acabam se propagando, formando “bolhas” que vão se expandindo, até que eventualmente toda a região esteja tomada.

Este comportamento segue o seguinte modelo:

1. Cada processador p tem um inteiro associado, $n_a[p]$ indicando o seu número total de atrasos.
2. Inicialmente $n_a[p] = 0, \forall p$.
3. A cada passo, para cada p :
 - $n_a[p] \leftarrow \max\{n_a[i] | i \in \{\text{vizinhos de } p\}\}$.
 - $n_a[p]$ é incrementado com probabilidade α .

O tempo total de execução é dado por

$$N \cdot (C_2 + A \cdot \delta + T_{troca})$$

onde $A = \max\{n_a[p]\}$, ao final das N iterações.

Limite superior de tempo Se a rede de interconexão for um grafo completo, um atraso é propagado para todos os processadores imediatamente e o tempo paralelo é equivalente ao do caso síncrono.

Limite inferior de tempo O outro extremo é representado pelo caso onde não há propagação de atrasos, como se os processadores estivessem desconectados.

Como neste caso todos os processadores são independentes, cada um deles é equivalente ao caso seqüencial. A distribuição do número de atrasos do conjunto é portanto a distribuição do máximo de P binomiais.

$$\begin{aligned} \mathcal{A}_{assinc}(m) &= \mathcal{P}(m \text{ máximo}) \\ &= \mathcal{P}(m = n_a[p] \text{ para algum } p \in P \text{ e } \nexists p' \in P | n_a[p'] > m) \end{aligned}$$

Este limite no entanto é muito baixo e uma expressão mais adequada pode ser obtida a partir das seguintes observações:

- O atraso espontâneo independe do processador e da iteração.
- Um processador pode atrasar de duas formas: espontaneamente e de forma “induzida” por um de seus vizinhos. Se σ for a probabilidade conjunta, é claro que $\sigma \geq \alpha$, já que α corresponde apenas ao atraso espontâneo.
- Um atraso global só ocorre se um dos processadores mais atrasados sofrer um novo atraso.

Para o limite inferior, desconsidera-se o atraso induzido nos vizinhos de um processador. Assim, se a rede de interconexão for um grafo regular de grau d e para uma região onde todos os processadores têm o mesmo número de atrasos, vale:

$$\sigma \geq 1 - (1 - \alpha)^{d+1} \quad (5.2)$$

onde o fator $d + 1$ indica o processador atrasou espontaneamente na iteração corrente ou sofreu um atraso induzido por um de seus d vizinhos na iteração anterior.

A existência de pelo menos uma região de $d + 1$ processadores ou mais com atraso máximo é garantida porque (5.2) leva em conta pelo menos uma propagação. Se um processador tem atraso máximo imediatamente antes da fase de troca, então é certo que ele e todos os seus vizinhos estarão com o mesmo atraso no início da iteração seguinte.

Conseqüentemente, um limite inferior para o número esperado de atrasos num grafo regular é dado por (5.3):

$$A = \left(1 - (1 - \alpha)^{d+1}\right) \cdot N \quad (5.3)$$

Caso geral O caso mais geral, com um grafo arbitrário, é muito complexo, principalmente por causa da interação entre duas bolhas.

Um novo atraso global só é gerado se um dos processadores mais atrasados em um dado instante atrasar espontaneamente.

É interessante notar que processadores vizinhos podem ter uma diferença entre seus atrasos abitrariamente grande, já que um processador pode atrasar mais de uma unidade em cada passo. A figura 5.2 ilustra uma diferença crescente numa linha de processadores. Novamente os processadores que sofrem atrasos “espontâneos” estão sublinhados.

Figura 5.2: Atrasos sucessivos em um linha de processadores

Em uma máquina real, isto significa que o número de mensagens na fila de comunicação pode crescer indefinidamente, provocando instabilidade no sistema. A probabilidade de que isso aconteça é muito baixa (o atraso deve ocorrer sempre na fronteira de uma bolha.)¹

Embora a probabilidade de um atraso espontâneo ser conhecida (α), o número de processadores com o atraso máximo muda constantemente, sem uma regra definida. Mesmo assim, é possível fazer algumas considerações de ordem mais geral:

- Para valores de α maiores do que 0.5, espera-se que em média ocorra um atraso a cada dois processadores, causando um atraso total a cada iteração, pois a propagação é imediata.

$$\alpha \gg 0.5 \Rightarrow A \cong N$$

- Similarmente, para valores muito pequenos de α , o esperado é que aconteça no máximo um atraso em cada iteração e o número total de atrasos seja o mesmo do caso seqüencial.

$$\alpha \ll 0.5 \Rightarrow A \cong \alpha \cdot N$$

Para este caso, α deve ser menor do que $1/(P \cdot \text{diâmetro do grafo})$

¹Por outro lado, o atraso pode ser provocado por alguma informação sendo transmitida de processador a processador na fase de troca de dados. Neste caso a borda da bolha sempre atrasa.

- Como a probabilidade de ocorrer pelo menos um atraso em uma iteração cresce com P (equação 5.1), tem-se

$$\lim_{P \rightarrow \infty} A = N$$

A solução para se obter uma estimativa mais detalhada dos tempos é através de simulações.

5.3 Simulações

Esta seção apresenta os resultados de simulações para grades e linhas de processadores de diversos tamanhos.

Cada simulação constou de 500 iterações e foi considerado o número de atrasos médio em 20 execuções, para valores de α variando entre 0.0 e 1.0, em intervalos de 0.05.

As figuras 5.3 e 5.4 mostram o atraso efetivo ($\frac{A}{N}$) para linhas de 10 e 900 processadores, respectivamente. As barras indicam o atraso mínimo e máximo entre as 20 execuções.

As simulações de grades resultaram nos gráficos das figuras 5.5 e 5.6, para os tamanhos 5×5 e 30×30 .

Finalmente, a figura 5.7 mostra duas topologias com o mesmo número de processadores: uma grade 30×30 e uma linha de tamanho 900 e ilustra um resultado surpreendente: se for possível escolher entre um anel e uma grade, para certos valores de α e δ o anel pode ser *muito* melhor, se o aumento na comunicação não for muito grande (o anel tem a desvantagem de provocar fronteiras maiores entre as regiões).

Por exemplo, para α entre 0.10 e 0.30, a diferença no atraso efetivo passa de 20%. O impacto real desta diferença depende do valor de δ .

Figura 5.3: Atraso efetivo para uma linha de 10 processadores

Figura 5.4: Atraso efetivo para uma linha de 900 processadores

Figura 5.5: Atraso efetivo de uma grade 5x5

Figura 5.6: Atraso efetivo de uma grade 30x30

Figura 5.7: Comparação entre uma linha 1x900 e uma grade 30x30

Capítulo 6

Sistemas de Memória Compartilhada Virtual

O uso de comunicação através de variáveis compartilhadas é um dos paradigmas mais antigos em computação paralela [4] pois, a menos do problema do acesso concorrente a uma determinada posição de memória, sua semântica é bastante simples.

No entanto, existem dois problemas sérios relacionados ao compartilhamento: a necessidade do tratamento do acesso concorrente e a dificuldade na construção de máquinas com um número maior de processadores.

Estes problemas são evitados em máquinas com memória distribuída, onde a comunicação é feita explicitamente através de troca de mensagens. Para tanto é necessária a inclusão de primitivas especiais de envio e recebimento das informações.

A principal vantagem do compartilhamento de memória é que ele permite o desenvolvimento de programas de uma forma mais intuitiva e mais próxima da forma de trabalho em máquinas sequenciais [44]. Em princípio é possível compilar um programa para uma máquina com memória compartilhada com pouca ou nenhuma modificação.

Além disso, algumas aplicações onde os dados não têm uma localização bem definida em termos de processador encontram a memória compartilhada como um meio natural para processamento paralelo. Em [31], este autor, juntamente com os co-autores, acabaram por construir um sistema especial de chamada remota de procedimentos para a paralelização de um sistema de modelagem geométrica na rede *NECTAR*, da Carnegie Mellon University. Este sistema faz uso de um grande número de ponteiros que necessitaram

de uma conversão em cada nó, pois não haviam garantias de que as cópias locais dos objetos referenciados ocupavam o mesmo endereço em todos os processadores.

Outras vantagens da memória compartilhada [4] incluem o fato de a troca de informação poder ser feita de modo completamente assíncrono (os processos nem precisam estar ativos ao mesmo tempo) e que não existe a necessidade de identificação do receptor. Por outro lado, o compartilhamento exige uma sincronização explícita (na troca de mensagens, em um sistema de memória distribuída, a própria comunicação fornece mecanismos de sincronização).

6.1 Memória compartilhada virtual

Uma solução intermediária é o uso de compartilhamento virtual, que implementa o compartilhamento em cima de um sistema com memória distribuída. Isto é feito de tal forma que se tem a ilusão de um único espaço de endereçamento para todos os processadores, com os blocos de memória sendo transmitidos pelos níveis mais baixos, seja por *software* ou *hardware*.

O método mais imediato (e mais comum) é considerar a memória compartilhada como um conjunto de páginas que são distribuídas entre os processadores. Eventualmente pode haver réplicas de algumas páginas em mais de um processador. Desta forma, a memória local de cada processador se comporta de modo similar a um *cache* da memória virtual.

Na figura 6.1 seis processadores, cada um com uma memória local de 4 páginas, compartilham 15 páginas (A–O) de memória virtual. Na situação ilustrada todos os processadores possuem uma cópia da página D.

O tamanho de cada página depende da implementação, variando entre dezenas de bytes (se o controle é feito por *hardware*), até poucos kbytes [38]. O uso de páginas maiores pode parecer mais interessante, já que pelo princípio da localidade dos dados o número de falhas de acesso deve ser menor (há uma tendência maior de que acessos consecutivos à memória referenciem endereços próximos), mas também ocorre o compartilhamento falso (*false sharing*), onde duas variáveis independentes habitam a mesma página. Neste caso pode acontecer que dois ou mais processadores comecem a competir pela mesma página, causando sobrecarga no sistema de manutenção de consistência.

Embora esta solução permita a construção de máquinas com um número maior de processadores, permanece o problema da consistência dos dados e da contenção, como será visto mais adiante.

Figura 6.1: Memória compartilhada virtual

6.1.1 Coerência

O problema principal é o da consistência dos dados, isto é, de que forma e por qual meio garantir que os processadores concordem no uso da memória [13, 22].

A semântica mais intuitiva é o da consistência seqüencial [38], que implica que qualquer leitura traz o resultado da última escrita na mesma posição de memória virtual. Uma forma de se garantir este comportamento é proibindo cópias de páginas. Assim apenas um processador pode escrever ou ler cada página em um dado instante (veja o modelo *mobile* adiante). De outra forma o modelo fica vago, pois a noção de ordem nem sempre é clara em um sistema distribuído.

Esta forma de consistência pode gerar um número excessivamente grande de sobrecargas. Uma forma mais amena é definida permitindo-se a existência de várias cópias de cada página com permissão de leitura, mas sem que haja permissão de escrita em mais de uma delas.

O modelo pode ser relaxado ainda mais, de diversas formas (veja a seção 6.2.2), mas neste caso o sistema deve fornecer meios adicionais para garantir a coerência, como barreiras de sincronização, por exemplo.

6.1.2 Manutenção da consistência

Existem diversos métodos para implementar a manutenção da consistência. Os problemas que devem ser resolvidos, quando um processador não possui uma cópia válida da página, são a localização da informação e a garantia da validade do modelo escolhido.

Um meio bastante simples que resolve os dois problemas mas que implica em uma grande quantidade de comunicação na vasta maioria dos casos é o endereçamento estático: as páginas têm posições fixas, distribuídas ao longo dos processadores e nunca se movem nem são copiadas. Neste caso, é gerada comunicação cada vez que um processador faz um acesso a uma posição de memória residente em outro nó.

Se for permitida a movimentação das páginas, é possível a utilização dos modelos de coerência menos rígidos, além de uma utilização mais eficiente da consistência sequencial. Mas cada processador que precisa obter uma cópia válida deve descobrir a localização da página necessária.

O método mais imediato é usar uma função de propagação de mensagens (*broadcast*) eficiente para pedir uma cópia válida da página necessitada, caso não se tenha uma local. Este método tem o problema de sobrecarregar todos os processadores cada vez que uma requisição é feita, pois a mensagem tem que ser tratada, além de introduzir contenção na rede.

Outro método, um pouco melhor, consiste no uso de centros de consistência responsáveis pela administração de um subconjunto das páginas. Toda movimentação de páginas inclui o envio de informação para um centro destes e o principal inconveniente é a contenção causada.

Um quarto método coloca a consistência junto ao processador ‘responsável’ pela página (aquele que tem a permissão de escrita, por exemplo). Inicialmente todos os processadores sabem qual é o dono de cada página e as requisições de cópias podem ser feitas normalmente. Se um processador receber uma requisição de uma página que ele não possui mais, ele providencia que o pedido seja enviado ao novo dono. Este método se torna menos eficiente se houver uma grande movimentação de páginas.

6.2 Sistema básico

O método desenvolvido neste trabalho (capítulo 7) usa como base a implementação do sistema *VOTE* na máquina *MANNA* que, além de ser um siste-

ma real e que vem sendo usado em diversas aplicações. apresenta propriedades que simplificam a criação de modelos de um modo substancial. A seguir são discutidas estas características e como se pode tirar proveito delas.

6.2.1 MANNA

A máquina *MANNA* [6] é composta de vários nós interconectados por uma chave *cross-bar* com uma taxa de transferência bi-direcional de 50 Mbytes/s. Cada nó contém dois processadores RISC i860 XP compartilhando 32 Mb de RAM.

A rede de interconexão, sendo uma chave *cross-bar*, é equivalente a um grafo completo, fornecendo um tempo de comunicação independente da distância entre processadores e evitando completamente a contenção nos canais de comunicação da rede¹. Com isto, o custo de comunicação pode ser modelado em função apenas do tipo de operação.

O relógio centralizado, que fornece o mesmo tempo em todos os processadores, permite a medição precisa dos tempos de todas as tarefas que envolvem algum tipo de comunicação.

6.2.2 PEACE e VOTE

VOTE[9, 10] é um sistema de memória compartilhada virtual implementada no ambiente *PEACE* [42].

Na verdade *PEACE* é uma família de sistemas operacionais onde podem coexistir diversas configurações do *kernel*. Sua implementação é orientada a objetos e tem suporte especial para a execução de programas paralelos.

O *VOTE* atualmente funciona sob um membro da família *PEACE*, cujo *kernel* tem a forma de uma biblioteca de comunicação, conseqüentemente reduzindo a sobrecarga de sistema a um mínimo.

Dentro do sistema *VOTE*, não é toda a memória que é compartilhada, mas apenas os objetos derivados de classes especiais, que definem o modelo de compartilhamento desejado. Na versão implementada na máquina *MANNA*, toda a memória é real (não existe disco).

Os modos de compartilhamento disponíveis até o momento estão ilustrados na figura 6.2 e são os seguintes:

¹É claro que pode haver contenção de comunicação nos processadores, como em uma operação do tipo *broadcast*, por exemplo.

Figura 6.2: Modos de compartilhamento no *VOTE*

mobile é o modelo básico e minimal. Objetos desta classe possuem uma única cópia (*single reader/single writer*).

sequential este modelo permite a existência de várias cópias com permissão de leitura, mas apenas uma com permissão de escrita. As atualizações são feitas de modo a garantir a consistência seqüencial.

original fornece acesso exclusivo temporário a um processador. Ou seja, um processador pode requisitar uma página da memória virtual apenas para si, de tal forma que ela seja a única cópia válida no sistema. Enquanto durar esta posse, qualquer acesso feito por outros processadores é inválido.

carbon é uma variação do original: permite várias cópias de leitura, mas apenas um processador pode escrever.

delay garante a “posse” exclusiva de um objeto por um tempo pré-determinado. Nos dois modos anteriores existe a possibilidade de bloqueios (*deadlocks*).

yellow é um modelo baseado no conceito de um produtor e diversos consumidores, isto é, a permissão de escrita não muda de processador. É uma espécie de versão permanente do modo *carbon*.

pending permite que falhas de leitura sejam tratadas como falhas de escrita, para casos onde ocorre uma leitura seguida de atribuição. Inicialmente o comportamento é o mesmo do modo *sequential*, o tratamento especial é ativado através de uma chamada a um procedimento específico.

continuous permite o tratamento de um conjunto de páginas contíguas ser tratado como uma única página.

writeonce permite apenas uma alteração no objeto, por exemplo para inicialização. A escrita depende de uma chamada explícita a um método.

writeagain como o anterior, mas permite que o objeto seja reinicializado diversas vezes.

O modo de controle das falhas de acesso a páginas de memória se baseia em três elementos fundamentais, dois deles implementados como processos leves (*threads*):

- Agente de captura ou *Catcher* — é a rotina responsável por executar a manutenção adequada, chamada toda vez que ocorre uma falha de acesso.
- Centro de Consistência ou *Advisor* — é um processo leve que funciona como um centro de gerenciamento das páginas compartilhadas. Ele mantém um mapa da localização das páginas sob seu controle, bem como as permissões de acesso de cada processador.

Um *Advisor* pode rodar em qualquer processador e podem existir vários *Advisors* na mesma aplicação, cada um tratando de uma parte da memória compartilhada. Em geral coloca-se um *Advisor* em um processador dedicado, se isto for possível.

- *Actors* — são processos leves responsáveis pelo tratamento *local*, isto é, em cada processador, de um objeto compartilhado.

O objetivo de um *Actor* é fornecer ao *Advisor* uma interface para controlar o gerenciamento de memória e a movimentação dos dados.

Existe um *Actor* para o próprio *Advisor* que otimiza algumas operações.

Em cada processador, a unidade de administração de memória (*MMU*) é programada para mapear as páginas compartilhadas e que possuem cópia

local. A tentativa de acesso a uma página remota caracteriza uma falha (*page fault*), provocando uma interrupção e a ativação do *Actor* correspondente que inicia o tratamento, comunicando-se com o *Advisor* adequado.

Tipos de falhas de acesso

As situações onde ocorre uma falha de acesso e seu tratamento são descritas a seguir:

O processador tenta ler uma página remota O *Actor* pede uma cópia ao *Advisor*. Se este tiver a cópia, ela é enviada ao processador, caso contrário o *Advisor* solicita a página a algum processador que a possua. É garantido que existe pelo menos uma cópia física real de cada página compartilhada.

O processador tenta escrever em uma página local Só ocorre falha se o processador não tiver permissão de escrita. Neste caso a permissão é solicitada ao *Advisor*, que invalida as outras cópias existentes antes de liberar o processador para escrever.

O processador tenta escrever em uma página remota A página é requisitada ao *Advisor*, que deve invalidar as outras cópias antes de providenciar a transferência.

A tabela 6.1 mostra os tempos de tratamento para cada um dos casos possíveis de falha de acesso [25]. Note que não é possível existir mais de uma cópia válida de uma página sem que esta tenha passado pelo *Advisor*.

No caso particular onde o *Advisor* não possui a cópia da página, uma falha de escrita é mais rápida do que a falha de leitura porque a página pode ser enviada diretamente ao processador destino. No caso de falhas de leitura, o *Advisor* sempre mantém uma cópia da página.

A figura 6.3 mostra as várias etapas do tratamento de uma falha de leitura, onde o *Advisor* (em B) tem a cópia. No instante T_0 a aplicação em A provoca uma falha de leitura, gerando uma interrupção e conseqüentemente ativando o *Catcher*. O *Catcher* aloca uma página de memória física e chama o *Advisor* responsável pelo endereço procurado.

No processador B, o *Advisor* encontra uma cópia local da página e dispara o *Actor*, para que este faça uma transferência assíncrona para a página física no processador A. Em seguida o *Advisor* atualiza suas tabelas.

Tipo de Falha	Número de Cópias			Tempo (μs)
	Local	Advisor	Outros	
Leitura	0	1	n	667
Leitura	0	0	1	1670
Escrita	0	0	1	1396
Escrita	0	1	0	794
Escrita	0	1	n	794 + n * 340
Escrita	1	1	0	628
Escrita	1	1	n	628 + n * 340

Tabela 6.1: Tempos de tratamento de falhas de acesso no VOTE

Em A, o *Catcher* dispara o *Actor* local, que faz o mapeamento do endereço físico no endereço lógico e o tratamento da interrupção é completado. Conforme a figura 6.3 o tempo total é de $667\mu s$, como indicado na primeira linha da tabela 6.1.

Figura 6.3: Detalhamento de uma falha de leitura

Capítulo 7

Modelagem

Este capítulo introduz um método novo para a construção de modelos de desempenho para o sistema *VOTE-MANNA*. Em seguida o método será aplicado a alguns exemplos.

O principal objetivo deste método é modelar o custo das sobrecargas, se possível de um modo que permita a adaptação do modelo final para outras arquiteturas.

O custo estritamente computacional é dado em parte pelo próprio algoritmo utilizado e também pode ser estimado de diversas maneiras, diretas ou indiretas.

A informação que o método precisa está disponível em tempo de compilação (à exceção dos valores de alguns parâmetros), mas é necessário algum conhecimento da semântica do programa. A idéia é que o programador consiga desenvolver um modelo para o seu programa sem a necessidade de contar com um especialista no funcionamento interno do sistema *VOTE-MANNA*. Toda a informação específica do sistema pode em princípio ser encapsulada em sub-modelos “pré-fabricados”.

Em princípio torna-se possível, para algumas classes de aplicações, a construção automática de modelos analíticos de desempenho.

O objetivo de se buscar um modelo fundamentalmente analítico como resultado é que este tipo de modelagem permite um entendimento melhor do desempenho pois, em geral, o resultado final é uma descrição detalhada do que ocorre durante a execução do programa.

Neste sentido, vale a pena sacrificar a precisão do modelo final em favor de sua simplicidade, desde que seu poder descritivo do comportamento da aplicação seja satisfatório. Como subproduto, o modelo resultante funciona

como uma ferramenta auxiliar de depuração de desempenho.

A construção de modelos analíticos, como foi visto no capítulo 2, apresenta alguns problemas, nem sempre de solução simples. Os fatores mais críticos são resumidos a seguir:

- Grau de detalhamento $\times$ precisão — Quanto maior a precisão desejada, mais detalhado deve ser o modelo. Por outro lado, um detalhamento muito refinado pode conduzir a um modelo intratável do ponto de vista prático. Neste trabalho, procura-se construir modelos de desempenho que auxiliem na compreensão do que ocorre durante a execução eventualmente em detrimento da precisão.
- Fatores externos [11] — Alguns fatores dependem fortemente das condições em tempo de execução. Por exemplo, o tempo de duração de uma certa parte do programa pode variar com os valores dos dados que ela está manipulando.
- Interferência de sobrecargas — Frequentemente é possível construir modelos específicos para cada tipo de sobrecarga, mas é complicado adaptá-los para situações onde mais de uma sobrecarga ocorre simultaneamente.

O princípio do método é identificar todos os pontos do programa que podem gerar algum tipo de sobrecarga. Estes pontos serão chamados de *eventos*.

Como a maior fonte de comunicação, nos casos mais comuns, são as falhas de acesso na memória local de cada processador, o centro do método consiste no rastreamento do movimento das cópias das páginas da memória virtual.

Pode-se deduzir, desde já, que o método se adapta melhor a programas onde o movimento de páginas é regular. Mesmo quando isso não ocorre, o método pode ser usado mas neste caso uma estratégia melhor seria uma simulação detalhada.

No entanto, estes casos em geral são aqueles onde o paradigma de memória compartilhada tende a não se adaptar muito bem. Por outro lado, como tipicamente um programa é paralelizado nos laços, o padrão de acesso à memória tende a ser regular.

No texto que segue, o termo *falhas* sempre será usado para indicar falhas de acesso à memória compartilhada, seja ela de escrita ou leitura. O termo *modelo* sempre indicará modelo de desempenho.

As funções do *VOTE* disponíveis ao usuário, tais como os métodos dos objetos compartilhados, serão designadas genericamente por *primitivas*.

O termo *interferência* indicará a situação onde o custo de uma sobrecarga é afetado pela presença de uma outra fonte de sobrecarga. O exemplo mais claro é a contenção no centro de consistência, quando dois processadores fazem uma requisição de página simultaneamente.

7.1 Impacto de cada sobrecarga no *VOTE*

No que segue será discutida a influência de cada uma das diversas espécies de sobrecarga potencialmente presentes no sistema *VOTE-MANNA*.

1. Comunicação

A maior fonte de comunicação são as falhas de acesso aos objetos compartilhados. Além do custo relativamente alto, o número e o tipo de cada falha dependem fortemente da aplicação.

As barreiras de sincronização também envolvem o envio de mensagens a um nó central (em geral diferente do centro de consistência), mas com um custo muito menor e serão consideradas em separado. O tempo gasto em cada barreira basicamente depende apenas do número de processadores envolvidos.

Outras fontes de comunicação são algumas das primitivas do *VOTE*, que aparecem explicitamente no código fonte e que podem ser modeladas à parte, de forma parecida com o que acontece com as barreiras.

2. Desbalanceamento de carga — O desbalanceamento de carga pode ter várias causas, entre elas as características da própria aplicação. A existência de barreiras de sincronização facilita bastante a modelagem, uma vez que elas impedem a propagação do atraso.
3. Contenção de recursos — É uma componente importante por causa do controle centralizado (centros de consistência). Juntamente com a comunicação, é a principal causa da degradação de desempenho no *VOTE*.
4. Barreiras de sincronização — O custo relacionado ao uso de uma barreira de comunicação é relativamente pequeno, se comparado ao da

comunicação e, em geral, ao da computação. No entanto, em um ambiente de memória compartilhada é comum o uso freqüente de barreiras e o tempo gasto com este recurso passa a ser significativo.

7.2 Notação

No restante deste capítulo, bem como no próximo, serão seguidas as seguintes convenções, além daquelas definidas no capítulo 2.

$\mathcal{P}$ é o conjunto de todos os processadores. A letra P indica o número de processadores.

T^p indica o tempo gasto pelo processador $p \in \mathcal{P}$.

$T_{Era[k]}$ é o tempo total gasto durante a k -ésima era (pág. 9).

T_{nome} é o tempo relacionado com a sobrecarga *nome*

$\mathcal{E}$ é o conjunto de todas as eras.

$\mathcal{S}$ é o conjunto de todas as sobrecargas.

As notações anteriores foram escolhidas de modo a ser possível combiná-las. Por exemplo, o tempo gasto pelo processador 4, na era 2, durante uma execução do programa com 7 processadores, é representado por

$$T_{Era[2]}^4(7)$$

7.3 Base do método

O método procura se aproveitar da simplicidade da implementação do *VOTE* e do comportamento regular de suas primitivas. Além disso, o fato de que existem poucos tipos de sobrecargas com grande influência no desempenho facilita a construção de modelos analíticos.

7.3.1 Suposições

O cálculo do tempo total de processamento e da contribuição de cada sobrecarga é feito com base nas seguintes suposições:

- O tempo total de processamento é dado pela soma dos tempos de cada era.

$$T(P) = \sum_{i \in \mathcal{E}} T_{Era[i]} \quad (7.1)$$

- O tempo de uma era é o tempo gasto pelo processador mais lento, somado ao tempo de reativação dos processadores, quando a era terminar com uma barreira de sincronização (que é o caso típico).

$$T_{Era[i]} = \max\{T_{Era[i]}^p, p \in \mathcal{P}\} + T_{reativação}(P) \quad (7.2)$$

- O tempo gasto por um processador dentro de uma era é dado pela soma do tempo de computação com a soma dos tempos gastos pelas diversas sobrecargas.

$$T_{Era[i]}^p = T_{comp} + \sum_{o_i \in \mathcal{S}} T_{o_i}^p \quad (7.3)$$

- O gasto envolvido em computação em cada processador independe dos outros processadores. Em outras palavras não existe interferência de computação.
- O tempo gasto com sobrecargas pode depender do que ocorre nos outros processadores. Existe interferência entre as sobrecargas. Esta interferência surge principalmente devido à contenção nas barreiras ou nos centros de consistência.
- O modelo para o tempo de computação é conhecido a priori.
- O tempo de reativação de barreira é considerado o mesmo para todos os processadores.
Na verdade não é isso o que ocorre, pois as mensagens partem de um nó central e são distribuídas (*broadcasted*) através de uma árvore, mas a diferença é relativamente pequena e é compensada pela própria sincronização.

Embora as próximas condições não sejam estritamente necessárias, o método conta com elas para a produção de modelos viáveis:

- O paralelismo é fundamentalmente explorado nos laços do programa. Isto é o que ocorre na grande maioria das vezes. Uma grande simplificação é possível se existirem barreiras de sincronização dentro de laços, impedindo a propagação de atrasos por desbalanceamento.
- Os laços apresentam um padrão regular de acesso às variáveis compartilhadas. Isto permite uma capacidade maior de compactação na descrição e conseqüentemente um modelo mais simples.

7.3.2 Dados de entrada

As informações necessárias para a construção do modelo de desempenho podem ser divididas em duas classes: as relativas à aplicação específica que se quer modelar e os dados relativos ao sistema *VOTE-MANNA*.

As informações relativas à aplicação podem ser obtidas diretamente do programa fonte eventualmente enriquecidas com o conhecimento que o programador tenha *da aplicação*.

Os dados relativos ao *VOTE-MANNA* são os seguintes:

1. Modelos básicos de tempo para as funções primitivas do *VOTE*. Estes modelos podem ser construídos com base em experimentos e conhecimento do comportamento interno do *VOTE* e do *PEACE*. No entanto eles são independentes da aplicação e alguns se encontram desenvolvidos na seção 7.4.2.
2. Modelos para o custo de barreiras de sincronização (seção 7.4.3).
3. Modelos básicos para o custo de falhas de acesso (seção 7.4.1).
4. Modelos para grupos de sobrecargas que podem estar ativas ao mesmo tempo e interferindo. Ou seja, são necessários modelos especiais de tempo para cada combinação de sobrecargas. Isto implica em um número grande de pequenos modelos, mas este número pode ser reduzido supondo-se que o tempo de sobrecargas interferentes é a soma dos tempos de cada uma. Embora esta suposição seja verdade na maioria dos casos, ou pelo menos seja uma simplificação razoável, ela não é válida para falhas de acesso, como será visto na seção 7.4.1, onde serão apresentados os modelos específicos para este caso.

7.4 Modelos básicos

Nesta seção são construídas funções que modelam eventos específicos do sistema *VOTE*. Elas podem ser usadas como blocos fundamentais na modelagem de aplicações mais complexas e além disso facilitam a comparação do desempenho com o de outros sistemas.

7.4.1 Modelos das falhas de acesso

O tempo básico para cada tipo de falha de acesso é dado pela tabela 6.1, mas deve-se levar em conta as interferências, no caso de requisições simultâneas ao centro de consistência.

A chegada de requisições simultâneas, oriundas de processadores diferentes, provoca um atraso adicional por causa da contenção no *Advisor*.

Independentemente da existência desta interferência, a tabela 6.1 sugere uma classificação das falhas de acordo com o tempo. Elas são *rápidas* se o *Advisor* tiver uma cópia da página requisitada, ou *lentas* caso contrário.

Existe ainda um caso especial de falha de escrita quando o processador onde ela ocorreu tem uma cópia válida da página mas não tem a permissão de escrita. Neste caso não há a necessidade de cópia de página e basta propagar uma mensagem de invalidação para os outros processadores que possuam uma cópia. Neste caso a falha é chamada *super-rápida*.

No caso de requisições simultâneas ocorre um *pipeline* no *Advisor*. A figura 7.1 ilustra este efeito com 4 requisições simultâneas, onde T_{inic} , T_{trat} e T_{fim} indicam os tempos gastos com inicialização, tratamento propriamente dito e finalização do atendimento, respectivamente (veja também a figura 6.3).

Figura 7.1: *Pipeline* no centro de consistência

O custo genérico para r requisições tem o seguinte formato:

$$T_{inic} + r \cdot T_{trat} + T_{fim}$$

e T_{trat} depende do tipo de falha tratada pelo *Advisor*.

A tabela 7.1 mostra os tipos de falha e o nome e argumentos das funções usadas para modelagem. Em todos os casos r indica o número de requisições simultâneas e, nas falhas de escrita, n indica o número de cópias que devem ser invalidadas.

Função	Tipo de falha
$F_{Lr}(r)$	Falha de leitura rápida
$F_{Ll}(r)$	Falha de leitura lenta
$F_{Er}(r, n)$	Falha de escrita rápida
$F_{El}(r, n)$	Falha de escrita lenta
$F_{Es}(r, n)$	Falha de escrita super-rápida

Tabela 7.1: Casos de falhas de acesso

As expressões completas são dadas a seguir:

$$F_{Lr}(r) = T_{inic} + r \cdot T_l^L + T_{fim} \quad (7.4)$$

$$F_{Ll}(r) = T_{inic} + r \cdot T_r^L + T_{fim} \quad (7.5)$$

$$F_{Er}(r, n) = T_{inic} + r \cdot (T_r^E + n \cdot T_{inv}) + T_{fim} \quad (7.6)$$

$$F_{El}(r, n) = T_{inic} + r \cdot (T_l^E + n \cdot T_{inv}) + T_{fim} \quad (7.7)$$

$$F_{Es}(r, n) = T_{inic} + r \cdot (T_s^E + n \cdot T_{inv}) + T_{fim} \quad (7.8)$$

onde T_l^L , T_r^L são os tempos de tratamento de leituras, lenta e rápida, respectivamente, T_{inv} o tempo gasto em uma invalidação e T_l^E , T_r^E os tempos para o tratamento de falhas de escrita, lenta e rápida, respectivamente.

Os valores dos diversos parâmetros foram medidos e se encontram na tabela 7.2. Estes tempos são reais (de parede).

7.4.2 Modelos de primitivas

A maioria das funções especiais do *VOTE* não chegam a gerar eventos, sendo apenas chamadas locais, isto é, ao *Actor* localizado no mesmo processador da chamada.

Parâmetro	Tempo(ms)
T_{inic}	0.187
T_{fim}	0.120
T_l^L	1.363
T_r^L	0.360
T_l^E	0.909
T_r^E	0.487
T_s^E	0.321
T_{inv}	0.340

Tabela 7.2: Parâmetros de escrita e leitura

No entanto, algumas primitivas envolvem uma comunicação rápida com o centro de consistência. Outro aspecto é que o próprio *VOTE* está em evolução e novas primitivas podem surgir. Para cada uma delas pode ser gerado um sub-modelo independente que é colecionado em uma “biblioteca”.

Aqui são apresentados os modelos para duas destas primitivas: *propagate* e *bypass*.

- *propagate* — Esta primitiva permite o envio de um conjunto de páginas a todos os processadores (*broadcasting*). O protótipo desta função é:

```
int propagate(void *from, void *to, int mode);
```

As páginas propagadas são aquelas entre os endereços *from* e *to*. O parâmetro *mode* indica a permissão final.

A propagação é feita através de uma árvore, e o custo de envio de uma página para P processadores obedece à seguinte expressão:

$$T_{propagate}(P) = P_0 + \log(P) \cdot P_1 \quad (7.9)$$

Medições experimentais de P_0 e P_1 fornecem, em milisegundos:

$$\begin{aligned} P_0 &= 1.2 \\ P_1 &= 0.8 \end{aligned}$$

- *bypass* — Esta primitiva implementa o envio de páginas assincronamente a um processador destino, permitindo o uso de troca de mensagens. Do lado do receptor deve ser chamada a função *acquire*, que bloqueia o processador até o recebimento da mensagem. *acquire* é uma chamada local, com retorno praticamente instantâneo se a página já estiver presente.

bypass no entanto tem o custo da comunicação e um custo extra na sua primeira chamada.

Este custo extra se deve a uma consulta ao centro de consistência para a obtenção do endereço real do processador destino. Uma vez obtida esta informação, ela é armazenada localmente e não há necessidade de novas comunicações com o *advisor* quando o processador destino for o mesmo.

O custo de comunicação é constante, pois a rede de interconexão é um *crossbar*, que se comporta como um grafo completo:

$$T_{bypass} = Bp_2 \quad (7.10)$$

A consulta ao *advisor* envolve contenção quando mais de um processador faz uma requisição simultaneamente. O custo é linear:

$$T_{bypass.setup}(p) = Bp_0 + (p - 1) \cdot Bp_1 \quad (7.11)$$

Os parâmetros, dados em milisegundos, valem:

$$Bp_0 = 0.150$$

$$Bp_1 = 0.350$$

$$Bp_2 = 0.800$$

Relacionada a estas duas primitivas, existe ainda mais uma *forceW*, local, que força a permissão de escrita em uma página, quebrando a consistência. A semântica das operações fica ao cargo do programador.

7.4.3 Modelo de barreiras

As barreiras são implementadas através de um servidor central. Cada processo leve deve enviar uma mensagem com o número total de processos que devem ser sincronizados e entrar em estado de espera. Ao chegar a última

mensagem, um aviso de reativação é distribuído a todos liberando novamente a execução.

A reativação é enviada através de uma árvore especial, construída com base nos recursos da rede de interconexão do *MANNA*. O tempo de reativação de um processo é $115\mu s$ e o de 16 processos é $309\mu s$. Este tempo é pequeno em relação a outras sobrecargas e não será levado em conta explicitamente.

O custo maior se dá por contenção no servidor, quando ocorrem chamadas simultâneas. Neste caso acontece um efeito tipo *pipeline* e pode-se usar um modelo linear:

$$T_{barreira}(P) = B_0 + P \cdot B_1 \quad (7.12)$$

onde P é o número de processadores sendo sincronizados.

B_0 e B_1 foram medidos experimentalmente e seus valores em milisegundos são:

$$B_0 = 0.027$$

$$B_1 = 0.088$$

7.5 Método

A construção do modelo é feita em cinco fases:

1. Relação dos objetos compartilhados:

Todos os objetos que são compartilhados devem ser identificados, bem como seus modos de compartilhamento, distribuição inicial de suas páginas e quais os centros de consistência responsáveis.

2. Identificação de eventos:

Todas as possíveis fontes de sobrecarga são registradas. O caso mais crítico é a identificação das falhas de acesso.

3. Cálculo dos instantes onde cada evento ocorre:

Este dado é particularmente importante para a verificação de contenções e desbalanceamento de carga (no caso das barreiras.)

Nem sempre é possível obter esta informação antecipadamente, pois a duração dos trechos do programa pode variar com os dados de entrada. Neste caso deve-se trabalhar com uma distribuição de valores (modelagem estatística) ou obter uma estimativa.

4. Construção dos modelos para cada grupo de eventos:
Quando dois ou mais destes eventos ocorrem dentro de um curto intervalo de tempo, pode haver interferência entre as sobrecargas que eles representam.
Os eventos devem ser então colecionados em função do instante onde ocorrem e da existência de interferência.
5. Construção do modelo final:
As fases 2, 3 e 4 geram informações necessárias para a aplicação das equações (7.3), (7.2) e (7.1). Em particular, (7.1) fornece o modelo para o tempo total do programa.

Cada uma destas fases é discutida em detalhe nas próximas seções. A título de exemplo, será construído o modelo para um programa de multiplicação de matrizes, apresentado na figura 7.2.

O próximo capítulo apresenta modelos para diversas situações e comparações entre os tempos previstos e medidos.

7.5.1 Objetos compartilhados

Os objetos compartilhados podem ser identificados diretamente no programa fonte, pois têm uma declaração especial. No caso particular de C++ eles pertencem a classes derivadas das classes especiais de compartilhamento do *VOTE*.

Exemplo:

O programa de multiplicação de matrizes define uma classe `vsm`, na linha 6, derivada da classe `pending`.
Nas linhas 9, 10 e 11, são declaradas três matrizes compartilhadas (`a`, `b` e `c`)

Para cada objeto compartilhado, deve-se manter o seguinte conjunto de informações:

- Conjunto de páginas da memória virtual que compõe o objeto. Neste nível não é importante a estrutura interna do mesmo.
O objetivo é se ter a capacidade de identificar, em cada acesso ao objeto, qual a página da memória compartilhada envolvida e quais os processadores afetados.

```

01: #include "primer.h"
02: #include "pending.h" // Avoid faulting R/W accesses
03: #define SIZE 512
04:
05: struct matrix { double m[SIZE][SIZE]; };
06: class vsm : public matrix, public pending { };
07:
08: void matmul (int argc, char** argv) {
09:     vsm& a = *new ("a", argv[2]) vsm;
10:     vsm& b = *new ("b", argv[2]) vsm;
11:     vsm& c = *new ("c", argv[2]) vsm;
12:     int id = a.voteId () - 1;
13:
14:     int pa = SIZE/atoi(argv[1]); int re = SIZE%atoi(argv[1]);
15:     int begin = id * pa + ((id < re) ? id : re);
16:     int end = begin + pa + ((id < re) ? 1 : 0);
17:
18:     for (i = begin; i < end; i++)
19:         for (j = 0; j < SIZE; j++)
20:             for (k = 0; k < SIZE; k++)
21:                 c.m[i][j] += a.m[i][k] * b.m[k][j];
22: }

```

Figura 7.2: Multiplicação de matrizes

Exemplo:

Os objetos da classe `vsm` são matrizes de `double`, de tamanho 512×512 .

Como cada `double` ocupa 8 bytes, e como cada página possui 4 kbytes no *VOTE-MANNA*, conclui-se que cada página contém uma linha da matriz.

- Distribuição inicial das páginas. Esta informação também está completamente contida no código fonte, pois o *VOTE* permite o controle desta distribuição.

Em particular, todas as variáveis alocadas dinamicamente se encontram inicialmente no *advisor* responsável.

Exemplo:

O operador sobrecarregado `new` recebe 2 parâmetros: um nome interno para o objeto alocado e o nome do *advisor* responsável pela manutenção deste objeto. No exemplo acima, o nome *advisor* é fornecido pela linha de comando (`argv[2]`).

Inicialmente, todas as páginas alocadas dinamicamente se encontram no centro de consistência correspondente.

- Centros de consistência responsáveis pelo objeto. No *VOTE*, a declaração dos centros responsáveis por cada objeto é explícita. Na declaração de um centro de consistência, são fornecidos os objetos e as páginas que este centro deve administrar.

Exemplo:

No exemplo existe apenas um *advisor*, responsável pelas três matrizes compartilhadas.

- Modo de compartilhamento. O modo inicial de compartilhamento é fornecido pela classe da qual o objeto faz parte e eventualmente pelos parâmetros da declaração. No entanto o modo de compartilhamento pode mudar com o decorrer da execução.

Exemplo:

A classe *vsm* é derivada da classe *pending*, que implementa o modo *pending* do *VOTE*. Neste modo é possível ativar e desativar a qualquer momento um tratamento especial de falhas de leitura, no qual estas são tratadas como falhas de escrita (seção 6.2.2).

Inicialmente esta característica está desabilitada e o comportamento é o mesmo do modo *sequential*.

7.5.2 Eventos

Alguns eventos correspondem a chamadas de funções especiais e barreiras de comunicação. Estes podem ser identificados diretamente no código, pois aparecem explicitamente.

Os casos mais delicados são as falhas de escrita e leitura, que não aparecem no programa fonte e dependem do histórico dos acessos.

Identificação de falhas de acesso A identificação deve ser feita para cada processador em separado e em alguns casos pode ser complicada, como quando existem acessos indiretos (através de ponteiros).

A princípio qualquer acesso a um objeto compartilhado pode gerar uma interação com o centro de consistência mas é preciso filtrar aqueles que realmente produzem uma falha.

Para se determinar as falhas, deve-se primeiro identificar os acessos. Isto é imediato pois basta verificar no programa fonte as referências às variáveis compartilhadas.

Para cada acesso a um objeto compartilhado, deve-se determinar:

1. Qual a página ou páginas envolvidas. A correspondência é dada pela estrutura do objeto.

Exemplo:

Os acessos às matrizes a, b e c se encontram na linha 21.

Como para cada uma das matrizes uma linha corresponde a uma página, os acessos são:

matriz	página (linha)	
a	i	
b	k	
c	i	(2 acessos)

2. Se o acesso é de escrita ou leitura. O acesso é de escrita se o objeto estiver sendo alterado (atribuições, operadores de auto incremento e auto decremento, etc).

Exemplo:

Os acessos a a e b são de leitura.

Para a matriz c existe um acesso de leitura seguido de um acesso de escrita (operador +=).

3. O modo corrente de compartilhamento do objeto. O modo inicial já é conhecido e toda alteração de modo é feita através de chamadas de primitivas específicas para este fim.

É portanto possível acompanhar todas as mudanças de modo pela semântica do programa, ou na pior das hipóteses por uma simulação.

Exemplo:

O modo de compartilhamento para as três matrizes é o *sequential*, pois não houve a chamada de habilitação do modo *pending*. Este modo vale para toda a execução do programa pois não existe nenhuma requisição de alteração de modo no programa todo. Esta requisição seria feita por alguma primitiva do *VOTE*.

4. Se existe uma cópia válida no processador. Esta informação só pode ser obtida através de um rastreamento do movimento de páginas. Isto em geral implica em uma simulação do programa, mas para aplicações *SPMD* com acessos regulares esta simulação muitas vezes pode ser feita em alto nível, pois basta acompanhar apenas os objetos compartilhados.

Exemplo:

Inicialmente apenas o *advisor* possui cópias das páginas. Conseqüentemente existe uma falha de acesso na primeira vez que cada processador acessa cada linha das matrizes.

Neste ponto se faz necessária uma explicação um pouco mais detalhada do código: `begin` e `end` são variáveis que definem em cada processador os limites do laço mais externo (em `i`), e são inicializadas de modo que a distribuição das iterações em cada processador seja a mais uniforme possível (linhas 14–16). A variável `id` contém a identificação do processador.

Assim, em cada processador:

- Existem (`end – begin`) falhas de leitura nas matrizes `a` e `c`.
- As falhas em `a` ocorrem todas as vezes em que `k` for igual a 0 (primeiro acesso).
- As falhas em `c` ocorrem quando `j` for igual a 0.
- Existem `SIZE` falhas de leitura na matriz `b`, toda vez em que `j` for igual a 0.
- Para cada falha de leitura em `c` existe, logo em seguida, uma falha de escrita na mesma página.^a
- Todas as falhas são rápidas. As falhas de escrita são super-rápidas.

^aIsto indica que o modo *pending* deve ser ativado para esta matriz, mas isto será analisado no capítulo 8

5. Permissão de escrita. Quando o processador escreve em um objeto compartilhado, deve-se verificar se, além de possuir a cópia da página, ele tem a permissão de escrita.

Exemplo:

Quando ocorre uma falha de escrita na matriz `c`, o processador já possui a cópia da página, mas não tem a permissão de escrita ainda.

Chamadas a funções especiais Estas aparecem explicitamente no código, portanto são identificadas imediatamente.

Exemplo:

A única primitiva chamada no exemplo é o método `voteId`, do objeto `a`, na linha 12.

Esta função apenas retorna a identificação do processador (um número entre 1 e P). Seu custo é praticamente nulo e ela não gera nenhum tipo de comunicação.

Barreiras Por serem implementadas com funções especiais, a situação é praticamente a mesma como no caso anterior, porém as barreiras exigem uma modelagem um pouco mais detalhada pois envolvem mais de um processador e um nó administrador.

7.5.3 Instante de cada evento

O objetivo do cálculo dos instantes é duplo: a determinação da duração de cada era e a presença de interferência entre sobrecargas.

De um modo geral, esta identificação só pode ser feita através de uma simulação detalhada do programa ou medindo-se os tempos diretamente. Entretanto, os instantes são calculados para cada era independentemente, ou seja, após cada sincronização pode-se “zerar o cronômetro”.

Isto significa que basta calcular os instantes relativamente ao início de cada era, o que simplifica bastante o estudo de programas onde existe uma sincronização global dentro de um laço.

Como a duração da era é dada pelo processador mais lento, pode-se fazer uma estimativa pelo pior caso ou se construir uma distribuição de probabilidade para o tempo, dado o grafo de execução.

Para que ocorra interferência, é necessário que dois eventos que compartilham o mesmo recurso ocorram em instantes suficientemente próximos. A definição de “suficientemente próximo” depende da duração de cada evento envolvido e da superposição necessária para que aconteça uma interferência de fato.

Exemplo:

Apesar de não haver barreiras de sincronização, cada iteração dos laços toma o mesmo tempo. Portanto, quando ocorre um acesso em um processador, existe um acesso similar em todos os outros, simultaneamente.

Examinando-se o código, observa-se o seguinte padrão de acesso:

i	j	k	falhas
begin	0	0	<i>P</i> leituras a <i>c</i> [<i>i</i>] e a <i>a</i> [<i>i</i>] <i>P</i> leituras a <i>b</i> [0] <i>P</i> escritas a <i>c</i> [<i>i</i>]
begin	0	1	<i>P</i> leituras a <i>b</i> [1]
begin	0	2	<i>P</i> leituras a <i>b</i> [2]
⋮	⋮	⋮	⋮
begin+1	0	0	<i>P</i> leituras a <i>c</i> [<i>i</i>] e a <i>a</i> [<i>i</i>] <i>P</i> escritas a <i>c</i> [<i>i</i>]
⋮	⋮	⋮	⋮
begin+2	0	0	<i>P</i> leituras a <i>c</i> [<i>i</i>] e a <i>a</i> [<i>i</i>] <i>P</i> escritas a <i>c</i> [<i>i</i>]
⋮	⋮	⋮	⋮
end-1	0	0	<i>P</i> leituras a <i>c</i> [<i>i</i>] e a <i>a</i> [<i>i</i>] <i>P</i> escritas a <i>c</i> [<i>i</i>]
⋮	⋮	⋮	⋮

7.5.4 Construção dos modelos para grupos de eventos

Para se saber se existe realmente interferência quando um conjunto de eventos acontece (quase) simultaneamente, é necessário um conhecimento do funcionamento interno do *VOTE*, do *PEACE* e do *MANNA*. Mas felizmente, pode-se manter um conjunto de grupos pré-definidos e apenas verificar a presença de cada um.

Neste trabalho a única forma de interferência que é tratada é a contenção no centro de consistência decorrente de falhas simultâneas de acesso.¹

Assim, basta verificar o agrupamento de falhas de mesmo tipo, no mesmo *advisor*.

¹A rigor, barreiras são formas de interferência, mas como já foi dito, são tratadas à parte.

Exemplo:

Para a matriz c, a falha de leitura é rápida e a falha de escrita é super-rápida. Elas ocorrem $\frac{SIZE}{P}$ vezes (ou seja end – begin) e o custo relativo a elas é dado por

$$(F_{Lr}(P) + F_{Es}(P, 0)) \cdot \frac{SIZE}{P}$$

A matriz a tem um custo parecido, a única diferença é que não ocorrem falhas de escrita:

$$F_{Lr}(P) \cdot \frac{SIZE}{P}$$

Já a matriz b tem todas as linhas referenciadas por todos os processadores e o custo associado a ela é

$$F_{Lr}(P) \cdot SIZE$$

7.5.5 Construção do modelo final

Na última fase, obtém-se uma estimativa dos custos finais de cada sobrecarga, em cada era, para cada processador ($T_{o_i}^p$).

Com esta estimativa e com o modelo para o custo puramente computacional pode-se escrever (7.3) e conseqüentemente, ao serem adicionados os custos relativos a barreiras de sincronização, obter (7.2) e (7.1), que corresponde ao modelo de desempenho procurado.

Para o caso geral, nem sempre o modelo resultante é simples o suficiente para ser tratável, mas é comum o caso onde existem várias eras consecutivas com as mesmas características de acesso, tornando a equação (7.2) relativamente simples.

Exemplo:

Como a aplicação é completamente paralelizável, o tempo de cálculo é inversamente proporcional ao número de processadores:

$$T_{comp}(P) = \frac{T_{comp}(1)}{P}$$

Existe apenas uma única era no programa, e pode-se escrever:

$$\begin{aligned} \sum_{o_i \in S} T_{o_i} &= (F_{Lr}(P) + F_{Es}(P, 0)) \cdot \frac{SIZE}{P} + F_{Lr}(P) \cdot \frac{SIZE}{P} \\ &\quad + F_{Lr}(P) \cdot SIZE \\ &= (2 \cdot F_{Lr}(P) + F_{Es}(P, 0)) \cdot \frac{SIZE}{P} + F_{Lr}(P) \cdot SIZE \end{aligned}$$

O modelo final é dado por:

$$\begin{aligned} T(P) &= \frac{T_{comp}(1)}{P} + (2 \cdot F_{Lr}(P) + F_{Es}(P, 0)) \cdot \frac{SIZE}{P} \\ &\quad + F_{Lr}(P) \cdot SIZE \end{aligned}$$

No próximo capítulo são apresentados resultados experimentais que são comparados a alguns modelos gerados por este método.

7.6 Situações especiais

Em algumas classes de aplicações, o padrão de eventos pode ser obtido de um modo genérico e é possível fazer uma análise preliminar sobre os modelos finais. Um exemplo típico é de decomposição de domínios.

7.6.1 Decomposição de domínio

Uma forma bastante popular de explorar paralelismo, principalmente em sistemas que envolvem simulação, é a decomposição de domínios.

Nesta técnica as variáveis, do tipo vetor ou matriz, que são manipuladas pela aplicação são divididas entre os diversos processadores e cada um opera apenas sobre o pedaço que lhe coube, isto é, sua “região” (capítulo 5).

De tempos em tempos, tipicamente uma vez a cada iteração, é preciso que se faça alguma troca de informação entre vizinhos para que os resultados

se propaguem sobre toda a variável. A informação trocada são os valores na “fronteira” de cada região.

A popularidade deste método se dá pela facilidade de implementação em máquinas com memória distribuída e pelo fato de que em geral é possível obter uma boa aceleração.

O formato típico de um programa que usa decomposição de domínio é:

```
DO
    computação
    troca de informações
ENDDO
```

sendo que em cada cálculo, toda a região é atualizada.

Em um sistema de memória compartilhada é necessária a sincronização ao final de cada iteração para garantir a consistência dos dados, mas por outro lado a troca de mensagens é implícita:

```
DO
    computação
    sincronização
ENDDO
```

O que se pode esperar do desempenho de um programa deste tipo no *VOTE*?

A existência de sincronização faz com que o tempo gasto em cada iteração seja o mesmo (veja o capítulo 5). Deve ser notado que em sistemas de memória compartilhada com comunicação síncrona o mesmo fenômeno acontece, pois a troca entre vizinhos força uma sincronização global.

Outro ponto é a distribuição inicial dos objetos. À medida que os processadores fazem acesso às páginas, estas podem fazer uma migração inicial. Depois deste “assentamento”, apenas os acessos correspondentes aos dados de fronteira provocam falhas de acesso.

Se o tempo de cálculo em cada iteração for aproximadamente o mesmo para todos os processadores, acontecerá que todos farão requisições ao(s) centro(s) de consistência simultaneamente, provocando contenção. Este problema pode ser amenizado com um número maior de centros de consistência, diminuindo a contenção em cada um deles, ou tratado através das extensões especiais do *VOTE*. Para um exemplo, veja a seção 8.2 no próximo capítulo.

Capítulo 8

Resultados

Vários experimentos foram realizados para validar a técnica apresentada no capítulo anterior.

Para cada aplicação escolhida, foi feita uma implementação “padrão” na máquina *MANNA*, isto é, usando o modelo mais direto do *VOTE* e estudadas várias alternativas, com outros recursos do sistema.

Para todos os casos foram desenvolvidos os modelos de desempenho [24, 23]. Estes modelos foram por sua vez usados para prever os tempos de execução e para análise de situações hipotéticas.

O tempo gasto com cálculo em cada aplicação foi medido diretamente.

8.1 Multiplicação de matrizes

Multiplicação de matrizes é um exemplo padrão de processamento paralelo em diversos estudos. A implementação “padrão” se encontra na figura 7.2.

O modelo de desempenho correspondente está desenvolvido no capítulo 7. Uma versão otimizada deste código é analisada e modelada na seção 8.1.3.

8.1.1 Modelo da versão “padrão”

O desenvolvimento do modelo para este caso foi apresentado como exemplo no capítulo anterior. O modelo resultante é reapresentado aqui:

$$T(P) = \frac{T_1}{P} + (2 \cdot F_{Lr}(P) + F_{Es}(P, 0)) \cdot \frac{SIZE}{P} + F_{Lr}(P) \cdot SIZE$$

O tempo para um processador foi medido diretamente.

8.1.2 Resultados experimentais

Os tempos previstos pelo modelo são apresentados na tabela 8.1, juntamente com os tempos obtidos na execução do programa na máquina *MANNA*. Pode-se notar nesta tabela que o erro percentual máximo não atinge 4%. O erro médio observado é de 1.5%.

O gráfico 8.1 compara as duas seqüências de números e ilustra a fidelidade do modelo e na figura 8.2 as acelerações prevista e medida são comparadas.

Número de processadores	Tempo (ms)		Erro %
	Previsto	Medido	
1	131200	131200	0.00
2	66025	65849	0.27
3	44423	44204	0.49
4	33714	33338	0.97
5	27363	27179	0.67
6	23190	23025	0.71
7	20262	20123	0.69
8	18112	17742	2.04
9	16480	16122	2.17
10	15212	15046	1.09
11	14208	13932	1.94
12	13402	13073	2.45
13	12749	12510	1.87
14	12215	11886	2.69
15	11777	11578	1.69
16	11416	10962	3.98

Tabela 8.1: Tempos previsto e medido para multiplicação de matrizes

Figura 8.1: Comparação entre os tempos previstos e medidos

Figura 8.2: Comparação entre as acelerações prevista e medida

8.1.3 Versão otimizada

Existem duas possibilidades de melhoria no código: a ativação do modo *pending* e o uso de mais de um centro de consistência. Com o modo *pending* ativado, o custo $F_{Es}(p, 0)$ deixa de existir, já que a falha de escrita é tratada em conjunto com a de leitura. O uso de vários centros evita a contenção: pode-se usar um centro para cada variável ou centros diferentes para subconjuntos disjuntos das páginas de cada objeto.

As modificações estão indicadas na figura 8.3 e são as seguintes:

- As declarações e comandos entre as linhas 8 e 11, fazem com que o *Advisor* identificado por `argv[2]` seja responsável pela administração das primeiras metades das matrizes *a* e *b* e pela segunda metade da matriz *c*. O *Advisor* identificado por `argv[3]` fica responsável pelo conjunto complementar.
- Entre as linhas 11 e 12 o modo *pending* é ativado para a matriz *c*.
- O laço mais interno (em *k*) é quebrado em dois, de modo que os acessos remotos às páginas de *a* e *b* não coincidam.

Com estas modificações, os tratamentos das falhas de acesso das matrizes *a* e *c* se sobrepõem completamente eliminando o fator 2 na equação (8.1). Se existirem *na* *advisors* trabalhando em paralelo, o modelo de tempo para a nova versão é

$$T(P) = \frac{T_1}{P} + F_{Lr} \left(\frac{P}{na} \right) \cdot \frac{SIZE}{P} + F_{Lr} \left(\frac{P}{na} \right) \cdot SIZE$$

A figura 8.4 mostra as curvas de aceleração com 1, 2, 4 e 16 *advisors* e as acelerações medidas com 1, 2 e 4 *advisors*. Uma reta indicando a aceleração ideal (linear) serve de referência.

```

08: void matmul (int argc, char** argv) {
    int ok;
    advisorpool po1 (2, &ok); advisorpool po2 (2, &ok);
    po1.fill (0, argv[2]); po1.fill (1, argv[3]);
    po2.fill (0, argv[3]); po2.fill (1, argv[2]);
09:   vsm& a = *new ("a", po1) vsm;
10:   vsm& b = *new ("b", po1) vsm;
11:   vsm& c = *new ("c", po2) vsm;
    c.pendingOn ();
12:   int id = a.voteId () - 1;

18:   for (i = begin; i < end; i++)
19:       for (j = 0; j < SIZE; j++) {
           for (k = begin; k < SIZE; k++)
21:               c.m[i][j] += a.m[i][k] * b.m[k][j];
           for (k = 0; k < begin; k++)
               c.m[i][j] += a.m[i][k] * b.m[k][j];
       }
22: }

```

Figura 8.3: Alterações no Código de Multiplicação de Matrizes

Figura 8.4: Aceleração com diferentes números de *advisors*

8.2 Sobre relaxação

O algoritmo de relaxações sucessivas (SOR — Successive Over Relaxation) é facilmente paralelizável e ilustra o caso de decomposição de domínios, juntamente com o problema da contenção no *advisor*. A versão mais direta, seguindo a implementação em [7], e usando consistência seqüencial, se encontra na figura 8.5.

Para evitar o uso de uma matriz “rascunho”, esta implementação usa duas matrizes, *r* e *b*, e a iteração é quebrada em duas fases: uma para linhas ímpares e outra para linhas pares.

Como foi dito, para paralelização foi utilizada a técnica de decomposição de domínio e cada processador se ocupa de uma faixa de cada matriz.

```
1: void sor(float r[M+2][N+1],float b[M+2][N+1],int begin,int end){
2:   int i, j, k;
3:
4:   for (i = 0; i < iterations; i++) {
5:     for (j = begin; j <= end; j++) {
6:       for (k = 0; k < N; k++)
7:         b[j][k] = (r[j-1][k]+r[j+1][k]+r[j][k]+r[j][k+1])/4.0;
8:       if ((j += 1) > end) break;
9:       for (k = 1; k <= N; k++)
10:        b[j][k] = (r[j-1][k]+r[j+1][k]+r[j][k-1]+r[j][k])/4.0;
11:     }
12:     barrier();
13:     for (j = begin; j <= end; j++) {
14:       for (k = 1; k <= N; k++)
15:         r[j][k] = (b[j-1][k]+b[j+1][k]+b[j][k-1]+b[j][k])/4.0;
16:       if ((j += 1) > end) break;
17:       for (k = 0; k < N; k++)
18:         r[j][k] = (b[j-1][k]+b[j+1][k]+b[j][k]+b[j][k+1])/4.0;
19:     }
20:     barrier();
21:   }
22: }
```

Figura 8.5: SOR : versão com consistência seqüencial

8.2.1 Construção do modelo

Novamente a principal fonte de sobrecargas são as falhas de acesso à memória. Porém, diferentemente do que ocorre com multiplicação de matrizes, as falhas acontecem de um modo regular e não apenas até que os processadores consigam uma cópia dos dados.

Outra diferença é a existência de barreiras de sincronização, que definem várias eras. Existem duas eras em cada iteração (barreiras nas linhas 12 e 20).

Relação dos objetos compartilhados Os objetos compartilhados neste caso são as matrizes r e b e inicialmente elas já se encontram distribuídas de tal modo que cada processador contenha a faixa entre as linhas `begin` e `end` de cada matriz. `begin` e `end` são calculados de modo similar ao exemplo de multiplicação de matrizes e portanto não há sobreposição.

Da mesma forma que o caso anterior, cada linha corresponde a uma página de memória.

Existe um único centro de consistência e ele não contém nenhuma cópia de páginas.

Identificação dos eventos Além das barreiras das linhas 12 e 20, existem falhas de acesso nas regiões de fronteira.

As matrizes r e b são referenciadas nas linhas 7, 10, 15 e 18.

Para o laço entre as linhas 5 e 11 (em j), observa-se que existe acesso remoto em $r[\text{begin}-1][0]$ e $r[\text{end}+1][0]$, provocando falhas lentas de leitura. Os dois acessos ocorrem na linha 7 ou na linha 10, dependendo da paridade de `begin` e `end`. O mesmo ocorre com relação a matriz b nas linhas 15 e 18.

Como a página correspondente à linha $b[\text{begin}]$ em um certo processador é a mesma da linha $b[\text{end}+1]$ do processador imediatamente anterior, as linhas 7 e 10 também provocam falhas de escrita em b , no início e no fim do laço em j . De um modo similar, existem falhas de escrita para r nas linhas 15 e 18. Estas falhas são lentas pois o processador perde a permissão de escrita ao ser fornecida uma cópia.

Em resumo, o que ocorre em termos de falhas de acesso é o seguinte:

1. Na primeira iteração do laço da linha 5:

- $r[\text{begin}-1]$ é lida e o processador anterior perde o direito de escrita sobre esta página.
 - $b[\text{begin}]$ é escrita.
2. No última iteração deste laço :
- $r[\text{end}+1]$ é lida e o processador posterior perde o direito de escrita sobre esta página.
 - $b[\text{end}]$ é escrita.
3. Na primeira iteração do laço da linha 13:
- $b[\text{begin}-1]$ é lida e o processador anterior perde o direito de escrita sobre esta página.
 - $r[\text{begin}]$ é escrita.
4. No última iteração deste laço :
- $b[\text{end}+1]$ é lida e o processador posterior perde o direito de escrita sobre esta página.
 - $r[\text{end}]$ é escrita.

Cálculo dos instantes A presença das duas barreiras forçam o sincronismo dos processadores e as falhas de cada laço ocorrem simultaneamente.

era	i	j	falhas
1	0	begin	$P - 1$ leituras lentas a $r[\text{begin}-1]$
1	0	begin	$P - 1$ escritas lentas a $b[\text{begin}]$
1	0	$\vdots$	
1	0	end	$P - 1$ leituras lentas a $r[\text{end}+1]$
1	0	end	$P - 1$ escritas lentas a $b[\text{end}]$
2	0	begin	$P - 1$ leituras lentas a $b[\text{begin}-1]$
2	0	begin	$P - 1$ escritas lentas a $r[\text{begin}]$
2	0	$\vdots$	
2	0	end	$P - 1$ leituras lentas a $b[\text{end}+1]$
2	0	end	$P - 1$ escritas lentas a $r[\text{end}]$
3	1	begin	$P - 1$ leituras lentas a $r[\text{begin}-1]$
3	1	begin	$P - 1$ escritas lentas a $b[\text{begin}]$
3	1	$\vdots$	
3	1	end	$P - 1$ leituras lentas a $r[\text{end}+1]$
3	1	end	$P - 1$ escritas lentas a $b[\text{end}]$
4	1	begin	$P - 1$ leituras lentas a $b[\text{begin}-1]$
4	1	begin	$P - 1$ escritas lentas a $r[\text{begin}]$
4	1	$\vdots$	
4	1	end	$P - 1$ leituras lentas a $b[\text{end}+1]$
4	1	end	$P - 1$ escritas lentas a $r[\text{end}]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Construção dos modelos para cada grupo de eventos Em cada era, existem $P - 1$ requisições ao *advisor* simultâneas.

Em todas as falhas de escrita, existe apenas uma página adicional a ser invalidada (ou no processador anterior ou no posterior).

O custo das falhas de leitura por era é

$$F_{Ll}(P - 1)$$

De modo similar, o custo de falhas de escrita é

$$F_{El}(P - 1, 1)$$

A exemplo da multiplicação de matrizes, a computação nesta aplicação é igualmente distribuída em todos os processadores e portanto o tempo de

cálculo cai na proporção $1/p$. Mais ainda, o trabalho computacional é o mesmo em cada uma delas, igual a

$$\frac{T(1)}{2 \cdot \text{iterations}}$$

O custo de cada era é dado por:

$$T_{Era[i]}(P) = \frac{T(1)}{2 \cdot \text{iterations}} + F_{Ll}(P-1) + F_{El}(P-1, 1) + T_{barreira}(P)$$

O custo por iteração é

$$\frac{T(1)}{\text{iterations}} + 2 \cdot (F_{Ll}(P-1) + F_{El}(P-1, 1) + T_{barreira}(P)) \quad (8.1)$$

Modelo final O modelo final é dado simplesmente pela multiplicação de 8.1 por iterations.

$$T(P) = \frac{T_1}{P} + 2 \cdot \text{iterations} \cdot \quad (8.2)$$

$$(F_{Lr}(P-1) + F_{El}(P-1, 1) + T_{barreira}(P)) \quad (8.3)$$

8.2.2 Resultados experimentais

A comparação entre os valores previstos pela equação (8.2) pode ser feita pela tabela 8.2. O gráfico correspondente se encontra na figura 8.6. O erro médio é de 2% e novamente o erro máximo não atinge 4%.

Os valores mostram um nivelamento no tempo a partir de 10 processadores, o que pode ser verificado diretamente do modelo (veja a curva teórica). Esta degradação deve-se basicamente à contenção no centro de consistência, que impõe um termo linear no tempo de tratamento das falhas (equações 7.8).

As barreiras também têm um custo linear, porém muito mais baixo.

8.2.3 Otimizações

O modelo e os resultados apresentados na seção anterior sugerem modificações no programa de modo a reduzir o tempo de execução.

Basicamente o que deve ser feito é a diminuição de requisições ao centro de consistência por meio de alguma das extensões do *VOTE*.

A seguir são discutidas duas alternativas, com graus de sucesso diferentes.

Número de processadores	Tempo (ms)		Erro %
	Previsto	Medido	
1	117706	117706	0.00
2	60138	62331	3.65
3	41285	41903	1.50
4	32241	32362	0.38
5	27120	27279	0.59
6	23961	24236	1.15
7	21924	22246	1.47
8	20587	20956	1.79
9	19717	20071	1.80
10	19174	19615	2.30
11	18868	19283	2.20
12	18741	19262	2.78
13	18752	19272	2.77
14	18870	19410	2.86
15	19074	19660	3.07
16	19348	20042	3.59

Tabela 8.2: Tempos para sobre-relaxação (consistência seqüencial)

Propagação

A primeira idéia é eliminar completamente as falhas de acesso, antecipando-se ao código e fazendo-se uma propagação das páginas procuradas.

Isto requer uma modificações relativamente pequenas após os laços e antes das barreiras. O método `propagate()` envia uma cópia das páginas que contém o segmento de memória descrito em seus dois primeiros argumentos para *todos* os processadores.

A figura 8.7 mostra as linhas incluídas após a linha 11. Um conjunto similar é incluído após a linha 19. O argumento `SyncRemain` inibe mensagens de invalidação, evitando falhas de leitura.

Modelo de tempo O novo modelo pode ser construído diretamente a partir de (8.2). Nesta nova situação, os custos relativos às falhas de acesso são substituídos pelo custo da função `propagate`.

Estes custos são dados pela expressão (7.9).

Existem 4 chamadas de propagação por iteração, cada uma envolvendo $p - 1$ processadores. Assim o custo total é escrito como:

$$T_{prop}(p) = \frac{T_1}{p} + \text{iterations} \cdot (4 \cdot (p - 1) \cdot Prop(p) + 2 \cdot T_{barreira}(p)) \quad (8.4)$$

Número de processadores	Tempo (ms)		Erro %
	Previsto	Medido	
1	118121	118121	0.00
2	59762	60017	0.43
3	41037	41328	0.71
4	32301	32830	1.64
5	27604	28278	2.44
6	24954	25720	3.07
7	23491	24375	3.76
8	22783	23548	3.36
9	22589	23590	4.43
10	22764	23779	4.46
11	23212	24350	4.90
12	23870	25013	4.79
13	24696	27461	11.20
14	25656	27544	7.36
15	26727	28814	7.81
16	27891	29805	6.86

Tabela 8.3: Tempos para a versão com propagação

A tabela 8.3 e o gráfico da figura 8.8 comparam os tempos previsto e medido para a versão com propagação de páginas. Desta vez o erro máximo foi maior, chegando a 11%, com 4% de erro médio.

Mesmo nestas condições, o gráfico mostra que o modelo é na verdade uma boa aproximação e que pode ser usado para prever o comportamento com mais processadores.

O uso de propagação piorou a aceleração.

Envio direto de páginas

A propagação das páginas a todos os processadores, embora evite a falhas de acesso, ainda deixa muito a desejar em termos de desempenho por dois

motivos básicos:

- Excesso de comunicação: as páginas são enviadas a mais processadores do que o necessário. Embora isto seja feito de modo eficiente, cada processador precisa apenas da informação de seus vizinhos.
- Contenção. Esta estratégia não resolveu o problema das requisições simultâneas ao *advisor*.

A solução para estes dois problemas é através do envio direto de páginas, emulando um sistema de troca de mensagens.

O *VOTE* permite o uso de troca de mensagens, com duas diferenças: cada mensagem deve ser um conjunto de páginas contíguas na memória e ainda persiste a necessidade de barreiras de sincronização.

As modificações que devem ser feitas no código se encontram indicadas na figura 8.9. Novamente é apresentada apenas uma parte das alterações: um código similar para a matriz *r* deve ser incluído após a linha 19.

Novamente o novo modelo pode ser obtido diretamente de (8.2). Os custos das falhas de acesso são substituídos pelos custos da primitiva *bypass*, dados por (7.10) e (7.11).

Como existem quatro fases de comunicação por iteração, o modelo final é dado por

$$T_{direct} = \frac{T_1}{p} + 4 \cdot T_{bypass.setup}(p) + 4 \cdot (Bp_2 + T_{barrier}(p)). \quad (8.5)$$

Como na maior parte do tempo a computação é maior do que as sobrecargas, o modelo é mais fiel que nos casos anteriores e a aceleração mais próxima da linear. Isto pode ser observado na tabela 8.4 e na figura 8.10. Para esta versão, o erro percentual máximo é de 1.34%. O erro médio é de 0.77%.

Número de processadores	Tempo (ms)		Erro %
	Previsto	Medido	
1	118505	118505	0.00
2	59471	59513	0.07
3	39816	40028	0.53
4	30007	30199	0.64
5	24136	24302	0.69
6	20233	20409	0.87
7	17456	17688	1.33
8	15382	15551	1.10
9	13776	13961	1.34
10	12499	12651	1.22
11	11461	11599	1.20
12	10601	10692	0.86
13	9880	9984	1.05
14	9267	9321	0.58
15	8740	8821	0.93
16	8283	8302	0.23

Tabela 8.4: Tempos para a versão com envio direto de páginas

Figura 8.6: Consistência seqüencial

```
11:     }  
    if ((id = black.voteId ()) > 1)  
        black.propagate (&b[begin][0], &b[begin][N], SyncRemain);  
    if (id < PROCS)  
        black.propagate (&b[end][0], &b[end][N], SyncRemain);  
12:    barrier();
```

Figura 8.7: Modificações para propagação

Figura 8.8: Tempos da versão com propagação


```

11:     }
    if ((id = black.voteId ()) > 1)
        black.bypass (&b[begin][0], &b[begin][N], id-1);
    if (id < PROCS)
        black.acquire (&b[end+1][0], &b[end+1][N]);
    barrier();
    if (id < PROCS)
        black.bypass (&b[end][0], &b[end][N], id+1);
    if (id > 1)
        black.acquire (&b[begin-1][0], &b[begin-1][N]);
    black.forceW (&b[begin][0], &b[end][N]);
12:    barrier();

```

Figura 8.9: Modificações para troca de mensagens

Figura 8.10: Tempos para a versão com envio direto de páginas

Capítulo 9

Conclusões

A modelagem analítica em geral implica em um compromisso entre simplicidade e precisão, mas por outro lado é uma ferramenta útil em diversas situações, principalmente se o interesse for maior no estudo do comportamento de uma aplicação do que nos tempos de execução.

O objetivo deste trabalho é aplicar a modelagem analítica onde seu uso é natural — essencialmente nas análises teóricas de comportamento — e mostrar que mesmo onde outras técnicas em princípio são mais adequadas, como no estudo de desempenho de aplicações e ambientes mais específicos, este tipo de abordagem pode e deve ser usado, ainda que de modo complementar.

Em particular, em um sistema de memória compartilhada com as características do *VOTE/PEACE*, a geração de modelos analíticos para o conjunto de sobrecargas relacionados ao sistema é simples e direta. É claro que o modelo final, dependendo da aplicação, pode se tornar complicado demais, mas ainda assim alguns gargalos podem ser identificados, e informações que auxiliam na compreensão mais exata do comportamento do sistema são obtidas.

9.1 Contribuições

As contribuições deste trabalho se dividem basicamente em três classes, apresentadas a seguir:

Estratégia de compilação

O capítulo 4 mostra, através de modelos genéricos, que a simples presença de sobrecargas muda a estratégia de paralelização no caso de laços indepen-

dentes.

Deve-se notar que no estágio de compilação as outras técnicas de análise de desempenho não estão disponíveis, com exceção da simulação, que pode ser muito custosa.

Para compiladores, não é tão importante o tempo exato de execução, mas a relação entre os tempos tomados por alternativas diferentes, que é a situação típica onde a modelagem analítica se aplica.

Atrasos em redes de inter-conexão

Atrasos aleatórios em sistemas com comunicação assíncrona tornam difícil a previsão do tempo de execução de um programa paralelo.

Este problema é tratado no capítulo 5, onde curvas para o limite inferior e superior de tempo são estabelecidas para sistemas onde o atraso é uniforme e independente do processador. Ali também são discutidos os comportamentos assintóticos para um número grande de processadores e para valores extremos da probabilidade de atraso.

Os resultados são complementados por simulações que mostram que, onde valem as condições do estudo, é preferível a utilização de uma topologia do tipo anel ao invés de grade para problemas com a estrutura típica de decomposição de domínios.

Modelagem de sistemas com memória compartilhada

No capítulo 7 são estudadas as características do sistema *VOTE-PEACE* e da máquina *MANNA* que favorecem a modelagem analítica e é desenvolvido um roteiro para a construção de modelos mistos.

As operações fundamentais do *VOTE* são modeladas por funções genéricas, que podem ser usadas como componentes de modelos mais complexos.

Estas funções modelam um nível relativamente baixo do *VOTE* (controle da movimentação de páginas e funções primitivas), o que permite que elas “sobrevivam” a extensões e modificações do sistema.

Por outro lado, elas se encontram em um nível suficientemente alto com relação ao *PEACE* e à máquina, evitando detalhes demasiados e conseqüentemente assumindo uma forma bastante simples. Isto também facilita em parte a migração de modelos para outras arquiteturas.

No capítulo 8 são construídos modelos segundo o roteiro do capítulo 7 e os resultados são validados experimentalmente, com uma precisão surpre-

endente, muito melhor do que a esperada. Estes experimentos mostram como os modelos podem ser utilizados para a comparação do desempenho de estratégias diferentes de uso dos recursos do *VOTE* e na identificação de gargalos em cada uma das aplicações.

9.2 Trabalho futuro

As três linhas apresentadas na seção anterior permitem uma continuidade de pesquisa, mas pretende-se concentrar esforços nas duas últimas.

No caso da estimativa do número de atrasos em redes de inter-conexão, tanto o limite superior como inferior teóricos podem ser melhorados em princípio. Outro ponto de interesse é o estudo de casos mais gerais e a análise de estabilidade do sistema.

O ponto de maior interesse, contudo, é o desenvolvimento de modelos para o sistema *VOTE-PEACE* e similares. Uma linha a ser seguida é o aprimoramento do roteiro de modo a se atingir a geração semi-automática de modelos.

Casos especiais também merecem um estudo em separado, que facilite a construção automática de modelos. Entre eles se destacam programas baseados em equações de recorrência uniforme, bastante estudados em áreas como vetorização e sistemas sistólicos, e que permitem uma descrição formal bastante direta [36, 41, 43].

Outra área de grande interesse é a de “exportação de modelos”, isto é, o transporte de um modelo feito para uma determinada máquina para outras [47]. Isto é particularmente interessante à medida que tanto o *PEACE* como o *VOTE* estão sendo traduzidos para outras plataformas e adicionalmente outros sistemas de memória compartilhada virtual vêm surgindo, com características parecidas com as estudadas neste trabalho .

Referências Bibliográficas

- [1] ADAMS, L. M.; CROCKET, T. W. Modelling algorithm execution time on processor arrays. *Computer*, v.17, n.7, p.38-43, 1984.
- [2] ADVE, V. S.; VERNON, M. K. The influence of random delays in parallel execution times. In: ACM SIGMETRICS, California, 1993. *Proceedings*. ACM.
- [3] AMDHAL, G. Validity of the single-processor approach to achieving large scale computing capabilities. In: CONFERENCE OF THE AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES, Reston, Va., 1967. *Proceedings*. AFIPS. v. 20, p. 483-5.
- [4] BAL, H. E.; TANENBAUM, A. S. Distributed programming with shared data. *Computer Languages*, v.16, n.2, p.129-46, 1991.
- [5] BALASUNDARAM, V.; FOX, G.; KENEDDY, K.; KREMER, U. A static performance estimator to guide data partitioning decisions. In: THIRD ACM SIG PLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, Williamsburg, 1991. *Proceedings*. ACM.
- [6] BRÜNING, U.; GILOI, W.; SCHRÖDER-PREIKSCHAT, W. Latency hiding in message passing architectures. In: INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM, Cancun, Mexico, 1994. *Proceedings*. IEEE.
- [7] CARTER, J. B. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*. Houston, Texas, Sep. 1993, Tese (doutorado), Rice University.

- [8] CHANDRA, S.; LARUS, J. R.; ROGERS, A. Where is time spent in message-passing and shared-memory programs? In: CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS VI, s.l., 1994. *Proceedings*. p. 61.
- [9] CORDSEN, J. Basing virtually shared memory on a family of consistency models. In: INTERNATIONAL WORKSHOP ON SUPPORT FOR LARGE SCALE SHARED MEMORY ARCHITECTURES, Cancun, Mexico, 1994. *Proceedings*. p. 58-72.
- [10] CORDSEN, J.; CORDSEN, M.; GERISHER, A.; SCHRÖER, B. O. F. W.; SODAN, A. Evaluation of the SODA DVSM consistency framework. GMD-first, Berlin. /Workpackage 2 final deliverable/.
- [11] CROVELLA, M. E. *Performance Prediction and Tuning of Parallel Programs*. Rochester, 1994, 115p., Tese (doutorado), University of Rochester.
- [12] FAHRINGER, T. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. Viena, Sep. 1993, 193p., Tese (doutorado), Technischen Universität Wien.
- [13] FALSAFI, B.; LEBECK, A. R.; REINHARDT, A. K.; SCHOINAS, I.; HILL, M. D.; LARUS, J. R.; ROGERS, A.; WOOD, D. A. Application-specific protocols for user-level shared memory. In: SUPERCOMPUTING '94, s.l., 1994. *Proceedings*. IEEE.
- [14] FLATT, H. P. Further results using the overhead model for parallel systems. *IBM journal of research and development*, v.35, n.5/6, p.721, Sep. 1991.
- [15] FLATT, H. P.; KENNEDY, K. Performance of parallel processors. *Parallel Computing*, v.12, n.1, p.1-20, 1989.
- [16] FRAIGNIAUD, P.; LAZARD, E. *Methods and problems of communication in usual networks*. Lyon, École Normale Supérieure de Lyon, 1991.
- [17] GERNDT, M. Array distribution in SUPERB. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 3, Kreta, 1989. *Proceedings*. ACM. p. 164-74.

- [18] GERNDT, M. *Parallelization of multigrid programs in SUPERB*. Viena, Austrian Center for Parallel Computing, 1990.
- [19] GERNDT, M. Updating distributed variables in local computations. *CONCURRENCY: Practice and experience*, 2, n.3, Sep. 1990.
- [20] GERNDT, M. Work distribution in parallel programs for distributed memory multiprocessors. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, Köln, 1991. *Proceedings*. ACM.
- [21] GERNDT, M.; ZIMA, H. *SUPERB: experiences and future research*. Viena, Austrian Center for Parallel Computing, 1990.
- [22] GHARACHORLOO, K.; GUPTA, A.; HENESSY, J. Performance evaluation of memory consistency models for shared memory multiprocessors. In: CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS-IV, s.l., 1992. *Proceedings*. ACM.
- [23] GUBITOSO, M. D.; CORDSEN, J. *Performance considerations in a distributed shared memory system*. Berlin, GMD-first Berlin, 1995.
- [24] GUBITOSO, M. D.; CORDSEN, J. *Performance considerations in VOTE for PEACE*. São Paulo, IME-USP, 1995. (Relatório Técnico RT-MAC-9514).
- [25] GUBITOSO, M. D.; CORDSEN, J. *Overheads in virtual shared memory and message passing systems*. Berlin, GMD-first Berlin, 1996. /em preparação/.
- [26] GUBITOSO, M. D.; SONG, S. W. Influence of overhead on processor allocation for multiple loops. In: VII SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES E PROCESSAMENTO DE ALTO DESEMPENHO, Canela, RS, 1995. *Anais*. SBC. p. 399-407.
- [27] HOCKNEY, R. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, v.17, p.1111-30, 1991.
- [28] JAKOBS, A.; GERLING, R. W. Scaling aspects for the performance of parallel algorithms. *Parallel Computing*, v.19, p.1063, 1993.

- [29] KREMER, U.; BAST, H.-J.; GERNDT, M.; ZIMA, H. Advanced tools and techniques for automatic parallelization. *Parallel Computing*, p. 387–93, 1988.
- [30] KUMAR, P. R.; I., S. T. Dynamic instabilities and stabilization methods in distributed real time scheduling of manufacturing systems. *IEEE Transactions on Automatic Control*, v.AC-35, n.3, p.289–98, Mar. 1990.
- [31] KUNG, H.; STEENKISTE, P.; GUBITOSO, M. D.; KHAIRA, M. Parallelizing a new class of large applications over high speed networks. In: THIRD ACM SIG PLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, Williamsburg, 1991. *Proceedings*. ACM.
- [32] MEIRA JR., W. *Modeling performance of parallel programs*. Rochester, The University of Rochester Computer Science Department, 1995. (Technical Report 589).
- [33] MEIRA JR., W. *Parallel Performance Understanding via Integration Modeling and Diagnosis*. Rochester, New York, 1996, 65p., Thesis proposal, Department of Computer Science of The College of Arts and Sciences of the University of Rochester.
- [34] MEIRA JR., W.; LEBLANC, T. J.; POULOS, A. Performance visualization and tuning with Carnival. Rochester, University of Rochester. /in preparation/.
- [35] MENDES, C. L. Performance prediction by trace transformation. In: V SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES E PROCESSAMENTO DE ALTO DESEMPENHO, s.l., 1993. *Anais*. SBC.
- [36] MOLDOVAN, D. I.; FORTES, J. A. B. Partitioning and mapping algorithms into fixed systolic arrays. *IEEE Transactions on Computers*, c-35, n.1, Jan. 1986.
- [37] MÜLLER-WICHARDS, D.; RÖNSCH, W. Scalability of algorithms: An analytic approach. *Parallel Computing*, 21, 1995.
- [38] NITZBERG, B.; LO, V. Distributed shared memory: A survey of issues and algorithms. *Computer*, v.24, n.8, p.52–60, Aug. 1991.

- [39] PANETTA, J. Problemas de compilação no IBM-3090 da petrobrás. Apresentado no CCCAD, Rio de Janeiro, Jul. 92.
- [40] POLYCHRONOPOULOS, C. D. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.
- [41] RIBAS, H. B. *Automatic Generation of Systolic Programs from Nested Loops*. Pittsburgh, June 1990, 103p., Tese (doutorado), Carnegie Mellon University.
- [42] SCHRÖDER-PREIKSCHAT, W. *The Logical Design of Parallel Operating Systems*. Prentice Hall, 1994.
- [43] SHANG, W.; FORTES, J. A. B. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE transactions on computers*, v.40, n.6, p.723, June 1991.
- [44] STUMM, M.; ZHOU, S. Algorithms implementing distributed shared memory. *Computer*, v.23, n.5, p.54-64, May 1990.
- [45] SUN, X.-H.; GUSTAFSON, J. L. Toward a better parallel performance metric. *Parallel Computing*, v.17, p.1093, 1991.
- [46] SUN, X.-H.; ZHU, J. Performance considerations of shared memory machines. *IEEE Transactions on Parallel and Distributed Systems*, v.6, n.11, p.1185-94, Nov. 1995.
- [47] VOLKEN, W.; KROPF, P.; NEUENSCHWANDER, H.; SCHWAB, P. Portability and performance study on Parsytec GC Power Plus and PowerXplorer, Feb. 1995.