Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Pós-Graduação em Ciência da Computação

Alternativas de Alto Desempenho para a Multiplicação Vetor-descritor

Pedro Antônio Madeira de Campos Velho

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes

Porto Alegre, outubro de 2006.

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
BIBLIOTECA CENTRAL IRMÃO JOSÉ OTÃO

Dados Internacionais de Catalogação na Publicação (CIP)

C198a Campos Velho, Pedro Antônio Madeira de
Alternativas de alto desempenho para a multiplicação vetor-descritor / Pedro Antônio Madeira de Campos Velho. – Porto Alegre, 2006.
82 f.

Diss. (Mestrado em Ciência da Computação) – Fac. de Informática, PUCRS.
Orientação: Prof. Dr. Luiz Gustavo Leão Fernandes.

1. Informática. 2. Avaliação de Desempenho (Informática). 3. Álgebra Tensorial. I. Fernandes, Luiz Gustavo Leão.

CDD 004.2

Ficha Catalográfica elaborada pelo
Setor de Processamento Técnico da BC-PUCRS

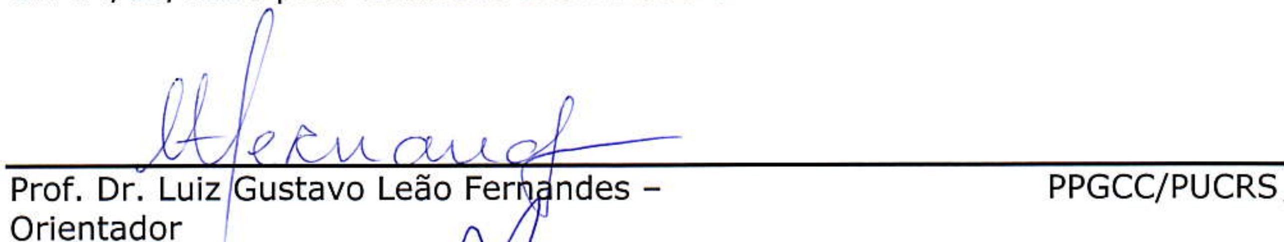
Av. Ipiranga, 6681 - Prédio 16
CEP 90619-900 - Porto Alegre - RS - Brasil
http://www.pucrs.br/biblioteca

Fone: 0 (xx) 51 3320-3544
FAX: 0 (xx) 51 3320-3548
e-mail: bceadm@pucrs.br

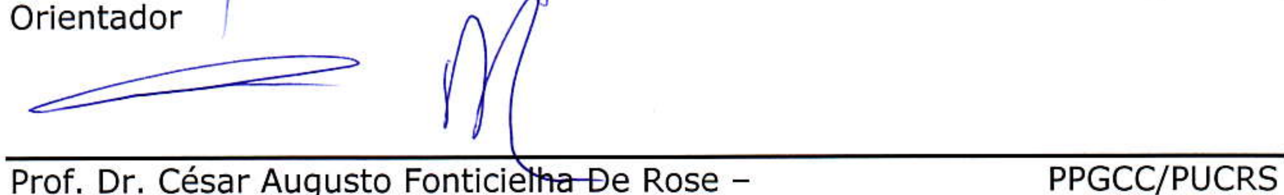
Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada ***"Alternativas de Alto Desempenho para a Multiplicação Vetor-descritor"***, apresentada por Pedro Antônio Madeira de Campos Velho, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 14/09/2006 pela Comissão Examinadora:



Prof. Dr. Luiz Gustavo Leão Fernandes – Orientador — PPGCC/PUCRS



Prof. Dr. César Augusto Fonticielha De Rose – PPGCC/PUCRS

Prof. Dr. Paulo Henrique Lemelle Fernandes – PPGCC/PUCRS

Prof. Dr. Philippe Olivier Alexandre Navaux – UFRGS

Homologada em 09/10/2006, conforme Ata No. 025 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 16 – sala 106 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

Agradecimentos

Gostaria de agradecer primeiramente as pessoas que trabalharam comigo no Centro de Desenvolvimento de Aplicações Paralelas (**CAP**) durante o desenvolvimento do presente trabalho. **Lucas Baldo**, meu fiel amigo, colega de mestrado e sócio com qual dividi e pretendo continuar dividindo momentos de alegria e realização profissional. Tivemos a oportunidade de trabalhar juntos desde cedo e são incontáveis as coisas que aprendi e ainda aprendo com essa grande pessoa que tu és. **Márcio Castro**, talvez o melhor aluno que o curso de Ciência da Computação da PUCRS venha a ter em toda a história. **Mateus Raeder**, um grande companheiro e um excelente profissional, principalmente na redação de artigos quando os prazos não estavam a nosso favor. **Thiago Tasca Nunes**, pela sua paciência e amizade que não tem preço, parabéns por ser a pessoa que tu és e esse profissional altamente capaz e sensato. **Gustavo da Silva Serra**, talvez o mais completo programador C/C++ que eu venha conhecer em toda minha vida, muito obrigado pela tua paciência em responder minhas dúvidas. Todos vocês são grandes profissionais, altamente capazes com os quais eu tive e tenho orgulho de continuar trabalhando. Nesse mesmo manifesto de gratidão, gostaria de mencionar o nome do meu amigo, orientador, padrinho e sócio **Luiz Gustavo Leão Fernandes**. A tua ajuda, atenção e amizade foi essencial para a realização profissional que obtive até então. Gostaria de deixar aqui registrado uma dívida eterna de gratidão que tenho para com todos vocês. Sempre que vocês precisarem de alguma coisa que esteja ao meu alcance eu estarei muito satisfeito em poder retribuir.

Gostaria de agradecer também ao pessoal do Performance Evaluation Group **PEG**. **Thais Webber**, **Ricardo Melo Czekster** e **Paulo Fernandes**, vocês foram muito importantes na escolha do tema e me ajudaram muito ao permitir que eu utilizasse o protótipo desenvolvido por vocês. Tenho muita satisfação em ter trabalhado com vocês e me orgulho muito dos artigos que publicamos em conjunto.

Finalmente, agradeço a minha família que participou ativamente da etapa de minha vida que resultou nessa dissertação. Obrigado mãe, **Rosália Maria Pereira Madeira**. Tu sempre me deste muito apoio, atenção, amor e carinho. Espero poder te dar mais motivos de orgulho. Igualmente agradeço a minha Vó Rita (**Rita Pereira Madeira**) pelo fato de me amar incondicionalmente e nunca ter questionado meus motivos e decisões. Vó Maria (**Maria Hugo Velho**), por ter investido tanto tempo e dinheiro na minha educação, os quais espero estar retribuindo com muito esforço e dedicação. Aos demais familiares e amigos gostaria de pedir desculpas por não colocar os seus nomes nessa página e dizer que se não o fiz foi realmente por falta de espaço. Obrigado a todos vocês.

Resumo

A modelagem analítica pode ser utilizada para prever desempenho, detectar deficiências e avaliar estratégias para melhorar sistemas. No contexto da modelagem computacional, diversos formalismos para a modelagem analítica estão se popularizando devido ao fato de proverem alto-nível de abstração e modularidade. No entanto, para inferir estimativas de desempenho destes modelos, é necessário resolver um sistema de equações. Em modelos analíticos estruturados, tais sistemas não se apresentam na forma tradicional, $Ax = b$, pois a matriz de coeficientes (A) é trocada por uma expressão algébrica (Q), denominada Descritor Markoviano (ou só descritor). Logo, a multiplicação convencional, Ax é substituída pela multipicação vetor-descritor (MVD), Qx. Dois algoritmos foram propostos recentemente para implementar a MVD: *shuffle* e *slice*. Ambos apresentam um alto custo computacional, que eleva drasticamente o tempo necessário para resolver modelos complexos. O objetivo do presente trabalho está relacionado com a utilização de técnicas de alto desempenho para propor versões mais rápidas, tanto para o algoritmo *shuffle* quanto para o *slice*.

Abstract

Analytical modeling can be used to predict performance, detect unexpected behavior and evaluate strategies in order to enhance systems. In the subject of modeling computational environments, a multitude of analytical modeling formalisms are becoming popular due to the fact that they enable the use of high level abstractions and modularity. However, to achieve performance statistics of a given analytical model, it is necessary to solve a linear equations system. In structured formalisms, this system is not presented in the usual notation, $Ax = b$, since the coefficients of matrix (A) are replaced by an algebraic expression Q, called Markovian Descriptor (or descriptor, for short). Indeed, the original multiplication, Ax is often changed for a vector-descriptor multiplication (MVD), Qx. Recently, two algorithms that implement the MVD have been proposed: shuffle and slice. Both demand high computational cost, which drastically increases the time necessary to solve complex models. The goal of this work is to exploit the use of high performance techniques in order to provide faster versions of shuffle and slice algorithms.

Lista de Figuras

1 Cadeia de Markov modelando a disputa de dois processos por um recurso. 16
2 Modelo SAN onde dois processos disputam um recurso. 16

3 Multiplicação do último fator normal ($nleft_N \otimes Q_N \otimes nright_N$). 27
4 Exemplo da multiplicação de um AUNF pela última matriz de um termo. 29

5 Diagrama de comunicação para a MVD em paralelo. 33
6 Abordagem de distribuição de carga para o *shuffle* sem considerar custos. . 35

7 Exemplo de descritor com dois termos, cada um com três matrizes. 42
8 Exemplo de arquivo de entrada para descritor da figura 7. 43
9 Aceleração do *Shuffle*, sem considerar custos (A) e considerando custos (B) para o teste SC. 45
10 Aceleração do *Shuffle*, sem considerar custos (A) e considerando custos (B) para o teste Misto. 46
11 Aceleração do *Shuffle*, sem considerar custos (A) e considerando custos (B) para o teste Denso A. 46
12 Aceleração do *Shuffle*, sem considerar custos (A) e considerando custos (B) para o teste Denso B. 47
13 Detalhe do algoritmo de balanceamento de carga, *shuffle*, teste Misto. . . 47
14 Aceleração do pré-processamento para o algoritmo *slice*. 49
15 Aceleração do *Slice* considerando cada termo como uma tarefa (A) e considerando cada AUNF como uma tarefa (B) para o teste SC 50
16 Aceleração do *Slice* considerando cada termo como uma tarefa (A) e considerando cada AUNF como uma tarefa (B) para o teste Misto 50
17 Aceleração do *Slice* considerando cada termo como uma tarefa (A) e considerando cada AUNF como uma tarefa (B) para o teste Denso A 51
18 Aceleração do *Slice* considerando cada termo como uma tarefa (A) e considerando cada AUNF como uma tarefa (B) para o teste Denso B 51

19 Comparação do tempo de execução entre *shuffle* (A) e o *slice* (B) para caso de teste SC. 54

20 Comparação do tempo de execução entre *shuffle* (A) e o *slice* (B) para caso Misto. 54
21 Comparação do tempo de execução entre *shuffle* (A) e o *slice* (B) para caso Denso A. 55
22 Comparação do tempo de execução entre *shuffle* (A) e o *slice* (B) para caso Denso B. 55

23 Modelo SAN para o problema da SC 62
24 Modelo SAN para o problema da SC com dois processos 63

25 Modelo SAN do caso de teste SC. 65
26 Modelo SAN do caso de teste Misto. 66

Lista de Tabelas

1 Descritor em SAN normalizado. 25

2 Detalhamento dos casos de teste 44
3 Comparativo entre custo de pré-processamento e o de uma iteração com o *slice*. 48

4 Shuffle - escalonamento sem considerar custos - teste MISTO. 67
5 Shuffle - escalonamento sem considerar custos - teste SC. 68
6 Shuffle - escalonamento sem considerar custos - teste DENSO A. 69
7 Shuffle - escalonamento sem considerar custos - teste DENSO B. 70
8 Shuffle - escalonamento considerando custos - teste SC. 71
9 Shuffle - escalonamento considerando custos - teste MISTO. 72
10 Shuffle - escalonamento considerando custos - teste DENSO A. 73
11 Shuffle - escalonamento considerando custos - teste DENSO B. 74
12 Slice - escalonamento por termo - teste SC. 75
13 Slice - escalonamento por termo - teste MISTO. 76
14 Slice - escalonamento por termo - teste DENSO A. 77
15 Slice - escalonamento por termo - teste DENSO B. 78
16 Slice - escalonamento por AUNF - teste SC. 79
17 Slice - escalonamento por AUNF - teste MISTO. 80
18 Slice - escalonamento por AUNF - teste DENSO A. 81
19 Slice - escalonamento por AUNF - teste DENSO B. 82

Lista de Símbolos e Abreviaturas

CM	Cadeia de Markov	14
DM	Descritor Markoviano	14
MVD	Multiplicação Vetor-descritor	14
SPN	*Stochastic Petri Nets*	15
PEPA	*Performance Evaluation Process Algebra*	15
SAN	*Stochastic Automata Networks*	15
AUNF	*Additive Unitary Normal Factor*	28
NOW	*Network of Workstations*	31
MPI	*Message Passing Interface*	31
SC	Seção Crítica	43
MMP	Milhares de Multiplicações em Ponto-flutuante	47

Lista de Algoritmos

1 Estratégia de paralelização em alto nível. 33
2 *Shuffle* utilizando abordagem de escalonamento sem considerar custos. . . . 34
3 *Shuffle* utilizando segunda abordagem de escalonamento, considerando custos. 36
4 *Slice* utilizando abordagem simples de escalonamento 38
5 *Slice*, detalhe do pré-processamento, considerando cada AUNF como uma tarefa. 39
6 *Slice* utilizando abordagem que considera cada AUNF como uma tarefa. . 40

Sumário

RESUMO **3**

ABSTRACT **4**

LISTA DE TABELAS **7**

LISTA DE SÍMBOLOS E ABREVIATURAS **8**

Capítulo 1: Introdução **14**

1.1 Motivação 15
1.2 Objetivos 16
1.3 Organização do Trabalho 17

Capítulo 2: Multiplicação Vetor-descritor **18**

2.1 Álgebra Tensorial 19
2.1.1 Produto Tensorial 19
2.1.2 Soma Tensorial 21
2.1.3 Termo Produto-tensorial 22
2.2 Descritor Markoviano 23
2.3 Exemplos de Descritores 23
2.3.1 Descritor em SAN 24
2.3.2 Descritor em Redes de Petri Estocásticas 24
2.4 Algoritmos para a MVD 25
2.4.1 Shuffle 26
2.4.2 Slice 28

Capítulo 3: MVD **de Alto Desempenho** **30**

3.1 Aglomerados de Computadores 31
3.2 MVD em Paralelo 32
3.3 Shuffle Paralelo 34

3.3.1 Abordagem de Escalonamento Simples 34
3.3.2 Escalonamento Considerando Custos 36
3.4 Slice Paralelo 37
3.4.1 Primeira Abordagem de Escalonamento 38
3.4.2 Escalonamento por AUNF 39

Capítulo 4: Resultados **41**
4.1 Plataforma de Teste 41
4.2 Casos de Teste 42
4.3 Resultados Shuffle 44
4.4 Resultados Slice 48

Capítulo 5: Conclusão **53**
5.1 Comparação entre Shuffle e Slice 53
5.2 Trabalhos Futuros 56
5.3 Consideraçõres Finais 56

REFERÊNCIAS BIBLIOGRÁFICAS **58**

Apêndice A: SAN **- Exemplo de Modelo Analítico Estrutural** **62**

Apêndice B: Semântica dos Modelos de Teste Utilizados **65**

Apêndice C: Detalhamento dos Resultados **67**

Capítulo 1

Introdução

A análise de desempenho vem se mostrando útil em diversas situações: prever o desempenho de aplicações paralelas, avaliar qualidade de serviços em sistemas distribuídos, etc. [6, 9, 10]. Entretanto, os formalismos para modelagem analítica apresentam limitações como o tempo necessário para inferir estimativas de desempenho e a alta necessidade de armazenamento à medida que os modelos são complexos [34].

Os formalismos estruturados constituem em um tipo de modelagem analítica. Esses se caracterizam por apresentarem uma maneira estruturada e modular de projetar sistemas gerando resultados comprovadamente equivalentes a **Cadeias de Markov** [5] (CM). Apesar desses formalismos serem resolvidos da mesma maneira, o mapeamento de modelos analíticos estruturados em uma CM equivalente não é desejado porque exigiria um grande custo de armazenamento [12]. Por esse motivo, tais formalismos utilizam algoritmos para realizar as operações necessárias na resolução dos modelos. Atualmente, os dois algoritmos que realizam esta operação, *shuffle* [18] e *slice* [34], apresentam um elevando custo computacional na resolução deste tipo de problema.

Formalismos para modelagem analítica estruturados caracterizam-se por possuírem um **Descritor Markoviano** (DM ou descritor para abreviar) [5]. Esta estrutura é equivalente a um gerador infinitesimal, uma matriz que armazena as taxas de transição entre os diversos estados de uma CM. Da mesma maneira que as CMs, a resolução desses modelos é efetuada resolvendo-se um sistema de equações lineares. Desta forma, a operação central na resolução de modelos analíticos estruturados é a multiplicação de um vetor por uma matriz. Os algoritmos *shuffle* e *slice* manipulam a estrutura do descritor efetuando esta multiplicação, a chamada **Multiplicação Vetor-descritor** (MVD) [2, 5, 22].

Utilizar máquinas mais potentes para aumentar o desempenho de aplicações é uma técnica que vem sendo constantemente empregada nos dias de hoje. Atualmente, aglomerados de computadores, do inglês *clusters of computers*, são atraentes uma vez que são construídos com tecnologias para o consumidor final e conseqüentemente apresentam uma boa relação custo/benefício. Essas máquinas estão ganhando popularidade no meio acadêmico onde são utilizadas para pesquisa de ponta nas mais diversas áreas [7, 17, 25, 33].

1.1 Motivação

O objetivo de realizar a MVD é efetuar a resolução de um sistema de equações lineares, independente do método de resolução empregado (Método de Newton, Método da Potência, etc.) [10, 29]. Atualmente, os métodos numéricos mais utilizados para resolver sistemas lineares são conhecidos como métodos iterativos. O que caracteriza um método iterativo (para resolução de sistemas de equações lineares) é que, a cada iteração, ele se aproxima da resolução do sistema até que atinja um erro mínimo ou um número máximo de iterações. Todavia, cada uma dessas iterações efetua a multiplicação de um vetor de resultados aproximado por uma matriz de coeficientes. Logo, a operação da MVD é executada diversas vezes (tipicamente centenas ou milhares de vezes) até que o problema seja resolvido.

Existem diversas situações nas quais é desejado analisar o desempenho de sistemas computacionais, tais como: avaliar o desempenho de um novo processador; prever o desempenho de configurações de ambientes e algoritmos; avaliar qualidade de serviço de aplicações distribuídas; assim por diante. Existem atualmente três técnicas empregadas para avaliar o desempenho nestas situações: *bechmarks*, modelos de simulação e modelos analíticos. *Benchmarks* consistem em uma série de aplicações (de custo elevado em termos de processamento) que deverão ser executadas de forma sistemática para qualificar e quantificar o desempenho de dispositivos específicos [4, 16].

A modelagem por simulação visa a construção de aplicações que simulem o comportamento de um sistema. Essa técnica pode ser utilizada para avaliar a qualidade de sistemas como redes de computadores, por exemplo [30]. Em nosso trabalho, focamos nos métodos analíticos.

A modelagem analítica visa a construção de um modelo que represente a realidade sendo estudada. Mais precisamente, tais modelos caracterizam-se por agregar diversas informações de transição (representando a interação entre entidades do sistema) e a freqüência com as mesmas ocorrem. Existem diversos formalismos para modelagem analítica, dentre as quais destacam-se: **Redes de Autômatos Estocásticos** [18], **Redes Ativas Estocásticas** [31], **Redes de Filas de Espera** [21], **Redes de Petri Estocásticas** (SPN) [2], **Álgebra de Processos para Avaliação de Desempenho** (PEPA) [22][1]. O escopo deste trabalho é restrito aos métodos de modelagem analítica estruturados. Estes métodos caracterizam-se por utilizarem álgebra tensorial para representar, através de uma expressão algébrica, um gerador infinitesimal comprovadamente equivalente a Cadeias de Markov [12, 15].

A vantagem de utilizar modelos analíticos estruturados envés das Cadeias de Markov está na obtenção de um maior poder de abstração. Para ilustrar, na figura 1, é apresentado um exemplo de Cadeia de Markov que modela dois processos disputando o acesso a um

[1]As siglas dos formalismos derivam de suas respectivas siglas internacionais em inglês: *Stochastic Petri Nets* (SPN), *Performance Evaluation Process Algebra* (PEPA) e *Stochastic Automata Networks* (SAN).

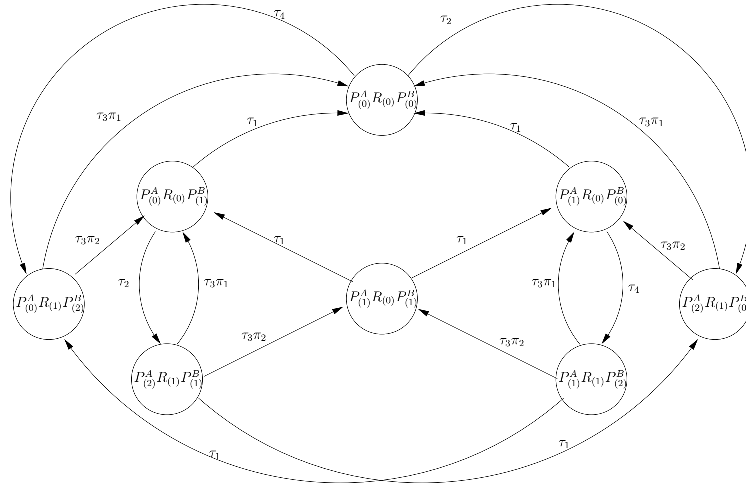


Figura 1: Cadeia de Markov modelando a disputa de dois processos por um recurso.

recurso compartilhado. Utilizando esse formalismo, somente um módulo é construído onde os estados representam a combinação de um estado de cada entidade do sistema. Já utilizando um formalismo de modelagem analítica estruturada, como por exemplo o modelo SAN visto na Figura 2, é possível separar em módulos cada entidade do sistema. Assim, acrescenta-se poder de abstração uma vez que ambos modelos são equivalentes [19].

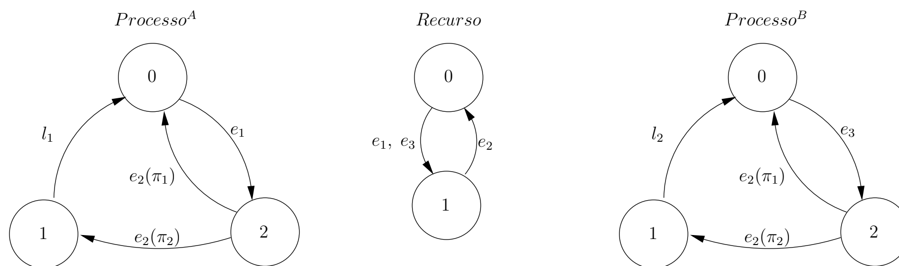


Figura 2: Modelo SAN onde dois processos disputam um recurso.

Os algoritmos para os quais apresentamos alternativas de alto desempenho podem ser utilizados para a MVD independente do formalismo utilizado [2, 22]. Entretanto, cada formalismo possui técnicas específicas para o armazenamento das estruturas algébricas, e conseqüentemente desempenhos diferentes. Este cenário dificulta a elaboração de uma biblioteca de primitivas para realizar a MVD.

1.2 Objetivos

O objetivo deste trabalho é empregar técnicas de alto desempenho nos algoritmos *shuffle* e *slice* visando acelerar a Multiplicação Vetor-descritor, a operação empregada na resolução de modelos analíticos estruturados.

A plataforma de alto desempenho utilizada para a execução dos testes é um aglomerado de computadores. Essas plataformas, conhecidas em inglês como *clusters*, são multi-computadores que compartilham dados utilizando uma rede. O padrão mais utilizado para a troca de mensagens neste tipo de plataforma é MPI, por tratar-se de um padrão consolidado para a troca de mensagens em multi-computadores, portável, de comprovada eficácia e com ampla documentação disponível [24].

1.3 Organização do Trabalho

No capítulo 2, o problema da multiplicação vetor-descritor é apresentado seguido de uma explicação dos algoritmos *shuffle* e *slice* utilizados para realizar esta operação atualmente. No capítulo 3 são apresentadas as estratégias para realizar a MVD em uma plataforma de computação de alto desempenho onde detalhes de escalonamento são fornecidos. A seguir, no capítulo 4, são mostrados como os testes foram realizados e em seguida os resultados para cada um dos algoritmos. Finalmente, no capítulo 5 algumas conclusões e perspectivas, para trabalhos futuros, são discutidas. O apêndice A mostra um exemplo de formalismo para modelagem analítica e como o Descritor Markoviano é gerado. No apêndice B, é apresentada uma descrição gráfica dos modelos utilizados para testar o desempenho das implementações propostas ao longo deste trabalho. Por fim, no apêndice C, são apresentados os detalhes numéricos dos resultados.

Capítulo 2

Multiplicação Vetor-descritor

Os formalismos analíticos estruturados caracterizam-se por apresentarem uma forma modular de representar sistemas reais. Esses formalismos são mais atraentes que Cadeias de Markov por permitirem projetar um sistema grande em pequenas partes independentes, especificando as interações entre cada módulo quando necessário. Dessa forma, os formalismos conservam as propriedades das CMs de forma que podem inferir as mesmas estimativas de desempenho de um determinado sistema.

O mapeamento de um modelo analítico estruturado em uma CM é realizado utilizando as operações de álgebra tensorial. Estas operações combinam os estados independentes de cada um dos módulos do modelo de forma a gerar uma estrutura equivalente ao gerador infinitesimal. O gerador infinitesimal é uma matriz que contém as taxas de transições entre cada um dos estados em um modelo descrito utilizando Cadeias de Markov. Todavia, como o gerador infinitesimal gerado neste processo é muito esparso, é preferível armazenar as matrizes de forma a minimizar a utilização de memória. Para tratar este problema, o gerador infinitesimal é então substituído por um Descritor Markoviano (ou somente descritor para abreviar), uma estrutura algébrica que quando resolvida é equivalente ao gerador infinitesimal.

Para resolver modelos descritos utilizando CM, é necessário resolver um sistema de equações lineares. Ao final deste processo, o vetor solução do sistema conterá informações referentes à permanência em cada estado do modelo. Em outras palavras, apresentará estimativas de desempenho para o sistema real sendo modelado. Como os formalismos estruturados apresentam uma equivalência a CM, o método de resolução é análogo. No entanto, o gerador infinitesimal não existe nestes últimos, pois é substituído pelo descritor. Neste ponto a operação de multiplicação do vetor solução pela matriz de coeficientes, que é o gerador infinitesimal em CM, é substituído pela multiplicação do **vetor** pelo **Descritor Markoviano**. E portanto esta operação de multiplicação é chamada de **multiplicação vetor-descritor**.

A resolução de sistemas de equações lineares é realizada geralmente utilizando-se métodos iterativos. Métodos iterativos caracterizam-se por aproximarem o vetor resultado

do sistema a cada iteração. Neste caso, o número de iterações necessárias para resolver um sistema varia muito, pois é determinado por um erro mínimo do resultado aproximado ou um número máximo de iterações. A MVD, é portanto, uma operação que é realizada diversas vezes até que o modelo seja resolvido. O custo computacional desta operação determina o custo total de resolução de um modelo, quantificado em termos de tempo de processamento, por isso existe um grande apelo para que o custo desta operação seja o menor possível.

Uma vez que a Multiplicação Vetor-descritor(MVD) é uma operação algébrica que substitui a multiplicação de um vetor por uma matriz, torna-se necessário compreender as operações algébricas envolvidas nesta expressão. Na seção 2.1, os aspectos relevantes da álgebra tensorial são apresentados, bem como algumas propriedade relevantes para a construção dos algoritmos e conseqüentemente importantes para compreender a paralelização destes. Em seguida, na seção 2.3, os descritores de dois formalismos são apresentados com o intuito de justificar a escolha do formato de entrada escolhido. Finalmente, na seção 2.4 os dois algoritmos que realizam a MVD, *shuffle* e *slice*, são apresentados de forma a ressaltar os aspectos que permitem distribuir o cálculo.

2.1 Álgebra Tensorial

Para entender como a Multiplicação Vetor-descritor (MVD) é efetuada, é necessário conhecer as operações algébricas que são utilizadas para mapear um modelo analítico em uma cadeia de Markov. Estas operações são conhecidas como operações de álgebra tensorial clássica ou também por operações de Kronecker [15]. Apesar do Descritor Markoviano apenas utilizar o produto tensorial, a soma tensorial é freqüentemente utilizada e, portanto, se torna relevante para compreensão dos algoritmos que realizam a MVD.

As operações de soma e produto tesorial são definidas entre duas matrizes reais. Como no contexto dos formalismos analíticos estruturados, estas matrizes são sempre quadradas. Neste trabalho são apresentadas definições e exemplos para casos que contemplam somente matrizes quadradas. O leitor interessado em encontrar mais informações sobre a álgebra tensorial pode consultar as referências.

2.1.1 Produto Tensorial

O produto tensorial é uma operação algébrica definida entre duas matrizes reais. Dado duas matrizes A e B, onde A é uma matriz n_1[1], e B é uma matriz de ordem n_2, o produto tensorial entre estas duas matrizes (denotado por $A \otimes B$), gera uma matriz C de ordem $n_1 \times n_2$, onde cada elemento $C_{i,j}$ é dado segundo a equação 1. Nesta equação, x e y são

[1]Como foi explicado anteriormente, as matrizes no contexto deste trabalho são sempre quadradas. Dessa forma a ordem de uma matriz representa o número de linhas e colunas.

números reais positivos. O operador unário $\lfloor x \rfloor$ representa o número inteiro imediatamente inferior a x, e a operação $x \diamond y$ calcula o resto da divisão inteira de x dividido por y.

$$C_{i,j} = A_{\lfloor (i-1)/n_2 \rfloor +1, \lfloor (j-1)/n_2 \rfloor +1} \times B_{((i-1) \diamond n_2)+1, ((j-1) \diamond n_2)+1} \quad (1)$$

A idéia do produto tensorial é combinar cada elemento da matriz A com cada elemento da matriz B. Para ilustrar está idéia observe as matrizes abaixo:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

A matriz resultante do produto tensorial entre A e B no exemplo acima fica:

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \otimes \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} = \left[\begin{array}{cc|cc} 1 \times 3 & 1 \times 4 & 2 \times 3 & 2 \times 4 \\ 1 \times 5 & 1 \times 6 & 2 \times 5 & 2 \times 6 \\ \hline 2 \times 3 & 2 \times 4 & 1 \times 3 & 1 \times 4 \\ 2 \times 5 & 2 \times 6 & 1 \times 5 & 1 \times 6 \end{array}\right]$$

Repare que esta operação mapeia em cada elemento da matriz resultante o produto de um elemento de A por um elemento de B. Em outras palavras, o resultado desta operação é uma matriz que contém a combinação de todas as multiplicações possíveis entre elementos das matrizes A e B. Para ilustrar melhor esta idéia, abaixo é apresentado como a multiplicação de cada elemento a_{ij} da matriz A multiplica cada elemento b_{ij} da matriz B, gerado ao final do produto tensorial.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \otimes \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & a_{12} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \\ & \\ a_{21} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & a_{22} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \end{bmatrix}$$

2.1.2 Soma Tensorial

A soma tensorial é definida com base no produto tensorial. Esta operação, dentro do escopo deste trabalho, também recebe como operandos duas matrizes reais quadradas. Dado duas matrizes A e B, a soma tensorial destas matrizes (denotada por $A \oplus B$), é definida pela equação 2. Onde, I_M denota uma matriz identidade[2] da mesma dimensão da matriz M. Note que a soma tensorial gera igualmente uma matriz de ordem $n_1 \times n_2$.

$$A \oplus B = (A \otimes I_B) + (I_A \otimes B) \tag{2}$$

Para ilustrar o cálculo de uma soma tensorial utilizaremos as mesmas matrizes que utilizamos para exemplificar o produto tensorial:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Para este exemplo, a matriz resultante da soma tensorial entre A e B fica:

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \oplus \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} = \left(\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) + \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} \right) =$$

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 2 \\ 2 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 4 & 0 & 0 \\ 5 & 6 & 0 & 0 \\ 0 & 0 & 3 & 4 \\ 0 & 0 & 5 & 6 \end{bmatrix} = \left[\begin{array}{cc|cc} 1+3 & 0+4 & 2+0 & 0 \\ 0+5 & 1+6 & 0 & 2+0 \\ \hline 2+0 & 0 & 1+3 & 0+4 \\ 0 & 2+0 & 0+5 & 1+6 \end{array} \right]$$

A soma tensorial é amplamente utilizada para o mapeamento dos modelos descritos em formalismos analíticos estruturados em cadeias de Markov. No entanto, é preferível decompor esta operação em produtos tensoriais para homogeneizar o tratamento numérico realizado. Este aspecto se torna relevante para compreender os algoritmos que realizam a MVD e consequentemente faz-se necessário apresentar como esta decomposição pode ser realizada.

Dada uma série de somas tensoriais com N matrizes, esta propriedade diz que as N

[2]Matriz identidade é uma matriz cujo todos os elementos localizados na diagonal principal são iguais a 1 e todos os outros elementos são iguais a *zero*.

somas tensoriais podem ser compostas na soma de N produtos tensoriais. Para ilustrar está idéia veja o exemplo abaixo:

$$\begin{aligned}
\begin{bmatrix} 8 & 7 \\ 0 & 9 \end{bmatrix} \oplus \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} \oplus \begin{bmatrix} 3 & 2 \\ 0 & 1 \end{bmatrix} \quad = \quad & \begin{bmatrix} 8 & 7 \\ 0 & 9 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad + \\
& \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad + \\
& \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 3 & 2 \\ 0 & 1 \end{bmatrix}
\end{aligned}$$

Generalizando o exemplo acima para uma série de somas tensoriais com N matrizes temos:

$$\begin{array}{rccccccccccc}
M_1 \oplus M_2 \oplus M_3 \oplus \ldots \oplus M_N = & M_1 & \otimes & I_2 & \otimes & I_3 & \otimes & \cdots & \otimes & I_N & + \\
& I_1 & \otimes & M_2 & \otimes & I_3 & \otimes & \cdots & \otimes & I_N & + \\
& I_1 & \otimes & I_2 & \otimes & M_3 & \otimes & \cdots & \otimes & I_N & + \\
& \vdots & \otimes & \vdots & \otimes & \vdots & \otimes & \ddots & \otimes & \vdots & + \\
& I_1 & \otimes & I_2 & \otimes & I_3 & \otimes & \cdots & \otimes & M_N &
\end{array}$$

2.1.3 Termo Produto-tensorial

A idéia de um **termo produto tensorial** (ou somente termo, para abreviar) é representar diversas operações de produto tensorial consecutivas. Formalmente, um termo composto pelo produto tensorial de N matrizes M_1 a M_N ficará como na equação 3. Note que esta operação é bastante parecida com um somatório onde ao contrário de somas as operações realizadas são produtos tensoriais.

$$\bigotimes_{i=1}^{N} M_i = M_1 \otimes M_2 \otimes \cdots \otimes M_N \tag{3}$$

2.2 Descritor Markoviano

Um conceito fundamental para a compreensão da MVD é como o Descritor Markoviano é armazenado. Dependendo do formalismo utilizado para modelagem, a estrutura algébrica do descritor varia. Para abstrair os detalhes específicos de cada formalismo, adotamos o DM como sendo a soma consecutiva de diversos termos produto-tensorial. Dada uma série de somas de T termos, onde cada termo é composto pelo produto tensorial entre N matrizes, o descritor é apresentado como visto na equação 4. Neste contexto, é necessário restringir que cada matriz Q_i possuirá a mesma ordem, independente do termo k no qual se encontra. Dessa forma, é possível garantir que cada termo que compõe o descritor resulta em uma matriz, se resolvido, de mesma ordem.

$$\sum_{k=1}^{T} \bigotimes_{i=1}^{N} Q_i^{(k)} \tag{4}$$

Logo, um descritor composto por T termos com N matrizes cada fica como abaixo:

$$\sum_{k=1}^{T} \bigotimes_{i=1}^{N} Q_i^{(k)} = \begin{array}{ccccccccccc} Q_1^{(1)} & \otimes & Q_2^{(1)} & \otimes & Q_3^{(1)} & \otimes & \cdots & \otimes & Q_N^{(1)} & + \\ Q_1^{(2)} & \otimes & Q_2^{(2)} & \otimes & Q_3^{(2)} & \otimes & \cdots & \otimes & Q_N^{(2)} & + \\ Q_1^{(3)} & \otimes & Q_2^{(3)} & \otimes & Q_3^{(3)} & \otimes & \cdots & \otimes & Q_N^{(3)} & + \\ \vdots & \otimes & \vdots & \otimes & \vdots & \otimes & \ddots & \otimes & \vdots & + \\ Q_1^{(T)} & \otimes & Q_2^{(T)} & \otimes & Q_3^{(T)} & \otimes & \cdots & \otimes & Q_N^{(T)} & + \end{array}$$

Note que o símbolo $Q^{(k)}$ não denota potenciação e sim funciona como um índice para o somatório (para fazer esta distinção são utilizados os parênteses). Durante o texto a expressão $Q^{(k)}$ denota o k-ésimo termo do somatório do descritor. Para representar o descritor inteiramente, o símbolo Q é utilizado sem índices.

2.3 Exemplos de Descritores

Nesta seção apresentaremos dois exemplos de descritores de dois formalismos analíticos estruturados. O objetivo aqui é exemplificar como descritores diferentes podem ser manipulado algebricamente gerando uma soma de termos tensoriais como apresentado na equação 4. Ou, em outras palavras, como os diferentes descritores de cada formalismo podem ser algebricamente manipualdos para a mesma estrutura.

Os dois formalismos abordados nessa seção, Redes de Autômatos Estocásticos (SAN) e Redes de Petri Estocásticas (PEPA), foram escolhidos pela sua popularidade em trabalhos

científicos recentes [8, 6, 13].

2.3.1 Descritor em SAN

A idéia de uma Rede de Autômatos Estocásticos é descrever um modelo global de um sistema em diversos módulos (subsistemas) independentes entre si, onde as interações entre esses subsistemas podem ocorrer em alguns casos. Cada módulo independente é definido como um autômato estocástico. A modelagem de cada autômato é realizada parametrizando-se estados, transições e eventos. Uma vez que eventos podem representar sincronismo entre dois ou mais autômatos, os dados sobre um modelo são armazenados em matrizes que definem o comportamento local e matrizes que definem o comportamento sincronizante [18, 19, 20, 34]. O Descritor Markoviano, em SAN , apresenta-se como visto na equação 5. Nessa equação, N representa o número de autômatos e E o total de eventos que modelam interações entre os autômatos (eventos sincronizantes).

$$Q = \bigoplus_{i=1}^{N} Q_l^{(i)} + \sum_{e=1}^{E} \left(\bigotimes_{i=1}^{N} Q_{e^+}^{(i)} + \bigotimes_{i=1}^{N} Q_{e^-}^{(i)} \right) \tag{5}$$

Como pode ser observado, o descritor de SAN representa uma parte descrita como a soma tensorial entre matrizes. Como apresentando anteriormente, é possível decompor uma série de somas tensoriais aplicando-se a propriedade da decomposição em fatores normais. Logo, é possível **normalizar** [18] o descritor de SAN em um descritor que somente possui termos produto-tensorial. A tabela 1 demonstra como fica o descritor de SAN após a normalização da parte local.

Foge do escopo desse trabalho a apresentação do formalismo SAN em detalhes. Entretanto, devido à utilização de estudos de caso utilizando este formalismo, no final do trabalho uma explicação mais aprofundada sobre como os modelos SAN são construídos é fornecida no apêndice A.

2.3.2 Descritor em Redes de Petri Estocásticas

Redes de Petri Estocásticas (SPN) fornecem uma maneira de modelar sistemas baseados no conceito clássico de Redes de Petri. De maneira geral, os conceitos existentes em tal formalismo se aproximam dos utilizados em SAN, uma vez que ambos apresentam módulos independentes com pontos de interações específicos [11, 13, 2, 32]. De maneira análoga a SAN, o descritor utilizando SPN, apresenta-se algebricamente dividido em uma parte local e outra destinada a representar as interações entre módulos (parte sincronizante), visto na equação 6 [14].

Tabela 1: Descritor em SAN normalizado.

<table>
<tr><td rowspan="9">$\sum$</td><td colspan="2" rowspan="5">N</td><td>$Q_l^{(1)} \otimes_g I_{n_2} \otimes_g \cdots \otimes_g I_{n_{N-1}} \otimes_g I_{n_N}$</td></tr>
<tr><td>$I_{n_1} \otimes_g Q_l^{(2)} \otimes_g \cdots \otimes_g I_{n_{N-1}} \otimes_g I_{n_N}$</td></tr>
<tr><td>$\vdots$</td></tr>
<tr><td>$I_{n_1} \otimes_g I_{n_2} \otimes_g \cdots \otimes_g Q_l^{(N-1)} \otimes_g I_{n_N}$</td></tr>
<tr><td>$I_{n_1} \otimes_g I_{n_2} \otimes_g \cdots \otimes_g I_{n_{N-1}} \otimes_g Q_l^{(N)}$</td></tr>
<tr><td rowspan="4">$2E$</td><td rowspan="2">e^+</td><td>$Q_{1^+}^{(1)} \otimes_g Q_{1^+}^{(2)} \otimes_g \cdots \otimes_g Q_{1^+}^{(N-1)} \otimes_g Q_{1^+}^{(N)}$
$\vdots$</td></tr>
<tr><td>$Q_{E^+}^{(1)} \otimes_g Q_{E^+}^{(2)} \otimes_g \cdots \otimes_g Q_{E^+}^{(N-1)} \otimes_g Q_{E^+}^{(N)}$</td></tr>
<tr><td rowspan="2">e^-</td><td>$Q_{1^-}^{(1)} \otimes_g Q_{1^-}^{(2)} \otimes_g \cdots \otimes_g Q_{1^-}^{(N-1)} \otimes_g Q_{1^-}^{(N)}$
$\vdots$</td></tr>
<tr><td>$Q_{E^-}^{(1)} \otimes_g Q_{E^-}^{(2)} \otimes_g \cdots \otimes_g Q_{E^-}^{(N-1)} \otimes_g Q_{E^-}^{(N)}$</td></tr>
</table>

$$R' = \bigoplus_{i=1}^{N} R^{(i)} + \sum_{\tau_j \in T} \bigotimes_{i=1}^{N} R^{(i,j)} \tag{6}$$

Da mesma forma que em SAN , as somas tensoriais do descritor para SPN apresentado na equação 6 podem ser substituídas pela soma consecutiva de diversos termos produto-tensorial resultando em um descritor como o apresentado na equação 4. Esses exemplos indicam que, em geral, os formalismos de modelagem analítica estruturados podem beneficiar-se das versões dos algoritmos propostos nesse trabalho.

2.4 Algoritmos para a MVD

Algebricamente, o problema da Multiplicação Vetor-descritor consiste na multiplicação de um vetor x por um termo algébrico, o descritor, Q. Formalizando essa idéia, temos a equação 7 substituindo a multiplicação original Ax da matriz de coeficientes pelo vetor.

$$Qx = \left(\sum_{k=1}^{m} \bigotimes_{i=1}^{N} Q^{i}_{(k)} \right) x \tag{7}$$

Atualmente, existem dois algoritmos que manipulam o DM: *shuffle* e *slice*. A geração de uma única matriz, resolvendo a expressão algébrica do DM, não é desejada pois exigiria um grande custo computacional de armazenamento. Os algoritmos que realizam a

MVD tornam-se necessários para manipular esta estrutura de forma a manter o tamanho ocupado pelo modelo gerenciável em termos de necessidade de memória. A seguir serão apresentados os princípios básicos dos dois algoritmos que atualmente realizam a MVD.

2.4.1 Shuffle

A demanda por algoritmos que realizassem a MVD de maneira que não fosse necessário resolver o descritor em um gerador infinitesimal fez surgir o algoritmo *shuffle*. Inicialmente proposto por diversos autores, os principais trabalhos que fazem menção a este algoritmo são [18] e [19].

De maneira geral, algumas propriedades da álgebra tensorial são imperativas para a realização da MVD sem incorrer na geração do gerador infinitesimal. O *shuffle* se baseia na decomposição de um termo em um **produto ordinário de fatores normais**, vide equação 8. Esta propriedade permite que um termo produto-tensorial entre N matrizes seja decomposto na multiplicação de N termos, cada um com N matrizes. No entanto, após realizada esta decomposição, cada termo apresenta apenas uma matriz com valores originais enquanto todas as outras $N-1$ matrizes são substituídas por uma matriz identidade que conserva a ordem da matriz de mesma posição no termo. O objetivo de utilizar propriedade é realizar o mapeamento dos elementos de cada matriz no gerador infinitesimal equivalente.

$$
\begin{array}{rccccccccccc}
Q_1 \otimes Q_2 \otimes Q_3 \otimes \ldots \otimes Q_N = & (Q_1 & \otimes & I_2 & \otimes & I_3 & \otimes & \cdots & \otimes & I_N) & \times \\
& (I_1 & \otimes & Q_2 & \otimes & I_3 & \otimes & \cdots & \otimes & I_N) & \times \\
& (I_1 & \otimes & I_2 & \otimes & Q_3 & \otimes & \cdots & \otimes & I_N) & \times \\
& \vdots & \otimes & \vdots & \otimes & \vdots & \otimes & \ddots & \otimes & \vdots & \times \\
& (I_1 & \otimes & I_2 & \otimes & I_3 & \otimes & \cdots & \otimes & Q_N) &
\end{array} \quad (8)
$$

O algoritmo *shuffle* define os operadores $nleft_i$ e $nright_i$ para mapear os elementos das matrizes de um termo no gerador infinitesimal. O operador $nleft_i$ é um operador que dado uma matriz qualquer Q_i, de um termo qualquer, realiza o produtório da ordem de todas as matrizes que estão à esquerda de Q_i. No caso da matriz Q_1, ou seja, a primeira matriz de um termo, o operador $nleft_1$ por definição resulta em 1, uma vez que não há matrizes à esquerda deste termo. O operador $nright_i$ se diferencia de $nleft_i$ por retornar o produtório da ordem das matrizes que estão à direita de uma matriz Q_i. De forma análoga, este operador é por definição 1 quando a matriz está localizada na extremidade direita de um termo, ou seja, $nright_N = 1$, para um termo com N matrizes. Assim, o algoritmo *shuffle* se baseia na propriedade da decomposição em fatores normais para

reescrever cada termo do descritor como visto na equação 8:

$$
\begin{array}{rccccccc}
Q_1 \otimes Q_2 \otimes Q_3 \otimes \ldots \otimes Q_N = & nleft_1 & \otimes & Q_1 & \otimes & nright_1 & \times \\
& nleft_2 & \otimes & Q_2 & \otimes & nright_2 & \times \\
& nleft_3 & \otimes & Q_3 & \otimes & nright_3 & \times \\
& \vdots & \otimes & \vdots & \otimes & \vdots & \times \\
& nleft_N & \otimes & Q_N & \otimes & nright_N &
\end{array}
$$

O *shuffle* implementa a multiplicação de cada termo $Q^{(k)}$ utilizando o resultado da decomposição de um termo em fatores normais ordinários. Para cada matriz de um termo Q_i, o vetor deve ser multiplicado por esta matriz utilizando os resultados de $nleft_i$ e $nright_i$ com o objetivo de mapear a localização dos elementos da i-ésima matriz deste no gerador infinitesimal. Por exemplo, a multiplicação do vetor pela última matriz de cada termo ficará como apresentado na figura 3.

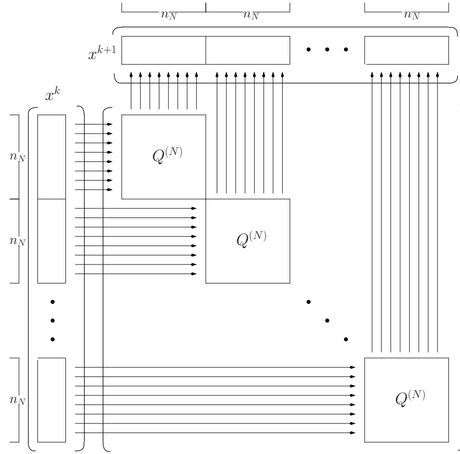


Figura 3: Multiplicação do último fator normal ($nleft_N \otimes Q_N \otimes nright_N$).

Neste caso particular, o operador $nright_N$ é igual a 1 e não representa mudança no resultado do produto tensorial. Logo, a matriz Q_N deve ser replicada em $nleft_N$ blocos diagonais no gerador infinitesimal como visto na figura 3. Entretanto, para as outras matrizes de um termo, esta operação utiliza blocos não contíguos do vetor e, conseqüentemente, exige a alteração de diversos elementos.

O custo computacional em número de multiplicações de ponto flutuante necessárias para a multiplicação do vetor x por um termo qualquer t, é dado pela equação 9 [18]. Onde $n_i^{(t)}$ é a ordem da i-ésima matriz do termo t e $nz_i^{(t)}$ corresponde ao número de elementos diferentes de zero na i-ésima matriz do termo t.

$$C_t = \prod_{i=1}^{N} n_i^{(t)} \times \sum_{i=1}^{N} \frac{nz_i^{(t)}}{n_i^{(t)}} \tag{9}$$

2.4.2 Slice

O algoritmo *slice* é relativamente novo e já tem apresentado resultados interessantes na resolução de determinados modelos [20]. A idéia central neste algoritmo consiste na decomposição dos termos produto-tensorial em pequenas somas algébricas. Acredita-se que o *slice* possa aproveitar melhor o poder de processamento de aglomerados de computadores, uma vez que permite a decomposição do problema em menores fatias.

O algoritmo *slice* é baseado na propriedade da decomposição aditiva. Esta propriedade demonstra que o produto tensorial consecutivo, entre duas ou mais matrizes, pode ser descrito como a soma da multiplicação de diversas matrizes unitárias. Esta propriedade é formalizada na equação 10. O princípio básico do algoritmo *slice* é aplicar essa propriedade sem considerar a última matriz de cada termo. As outras matrizes do termo são representadas por diversas matrizes unitárias, cada matriz unitária é chamada de **Fator Normal Unitário Aditivo** (AUNF ou fator para abreviar), essa sigla deriva da nomenclatura internacional, em inglês, *Aditive Unitary Normal Factor*.

$$Q^{(1)} \otimes \cdots \otimes Q^{(N)} = \sum_{i_1=1}^{n_1} \cdots \sum_{i_N=1}^{n_N} \sum_{j_1=1}^{n_1} \cdots \sum_{j_N=1}^{n_N} \left(\hat{q}^{1}_{(i_1 j_1)} \otimes \cdots \otimes \hat{q}^{N}_{(i_N j_N)} \right) \tag{10}$$

Para gerar os AUNFs, é efetuado o produto tensorial das $N-1$ matrizes de cada termo produto-tensorial. Por exemplo, supondo um termo produto-tensorial que envolva três matrizes Q_1, Q_2 e Q_3 ($Q_1 \otimes Q_2 \otimes Q_3$), a propriedade da decomposição aditiva é efetuada apenas considerando o produto $Q_1 \otimes Q_2$. Para exemplificar, supondo que as Q_1, Q_2 e Q_3 sejam:

$$Q_1 = \begin{bmatrix} 0 & 2 \\ 0 & 0 \end{bmatrix} \quad Q_2 = \begin{bmatrix} 3 & 0 & 4 \\ 0 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix} \quad Q_3 = \begin{bmatrix} 0 & 6 & 0 \\ 0 & 7 & 0 \\ 8 & 0 & 9 \end{bmatrix}$$

Utilizando a propriedade da decomposição aditiva todos os elementos das matrizes Q_1 e Q_2 são multiplicados separadamente para depois serem multiplicados pela matriz Q_3, como abaixo:

$$[(2 \times 3) \otimes Q_3] + [(2 \times 4) \otimes Q_3] + [(2 \times 5) \otimes Q_3]$$

Este procedimento se diferencia do realizado pelo *shuffle* principalmente por este mapear apenas a localização da última matriz de cada termo no DM. Para realizar este mapeamento, cada AUNF guarda a informação de linha e coluna onde este estaria localizado na matriz resultante do produto tensorial entre as $N-1$ matrizes mais à esquerda de um termo. Guardando estes índices, a geração de todos os fatores (AUNFS) só precisa ser realizada uma vez.

Esta técnica, portanto permite que os diversos termos tensoriais sejam desmembrados em pequenas multiplicações. Em máquinas de alto desempenho, como aglomerado de computadores, essa abordagem apresenta-se bastante atraente uma vez que torna possível considerar cada fator como uma tarefa independente. Este aspecto diferencia o *slice* do *shuffle* principalmente pela quantidade de dados manipulados na multiplicação de cada fator.

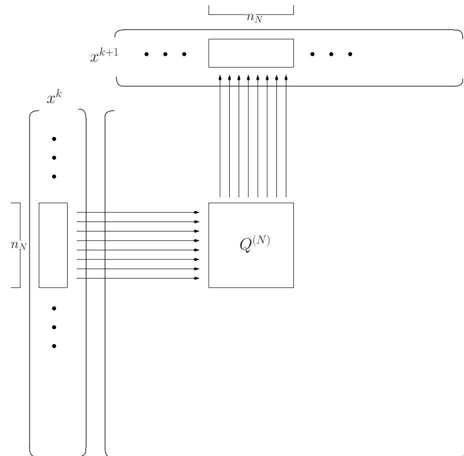


Figura 4: Exemplo da multiplicação de um AUNF pela última matriz de um termo.

Para ilustrar essa idéia, a figura 4 apresenta um exemplo de multiplicação de um AUNF pela última matriz de um termo. Neste caso, o número de elementos não nulos na última matriz do termo (nz_N), determina a quantidade de multiplicações que serão realizadas. No pior caso, onde a última matriz do termo, Q_N, não possui zeros, serão necessárias $n_N \times n_N$ multiplicações [20, 34].

Capítulo 3

MVD de Alto Desempenho

Atualmente, diversas aplicações utilizam computação de alto desempenho. O mapeamento do DNA humano [3], a predição de terremotos [1], a renderização de imagens para auxílio ao diagnóstico de doenças [27], são alguns exemplos. Modelos analíticos apresentam-se de grande aplicação prática [6, 9]. Estes podem ser utilizados, por exemplo, para modelar e apresentar estimativas de desempenho dos mais diversos sistemas computacionais. Uma aplicação prática possível, por exemplo, é a utilização para analisar sistemas de rede, auxiliando no diagnóstico de problemas como gargalos e roteadores sub-utilizados.

Os formalismos de modelagem analítica, de maneira geral, apresentam restrições na complexidade dos modelos que podem ser resolvidos por dois motivos: memória e tempo de resolução [12]. Tipicamente, a quantidade de memória utilizada pelos modelos aumenta à medida que a complexidade destes aumenta. Atualmente, para mitigar este problema, são utilizados algoritmos que realizam a MVD. A principal vantagem da utilização destes algoritmos é evitar armazenar o gerador infinitesimal reduzindo drasticamente a demanda de armazenamento. Em casos não raros, obtém-se uma redução de até 95% [20]. No entanto, o tempo necessário para a resolução de alguns modelos pode ser um fator limitador. Em trabalhos recentes, diversas abordagens vêm sendo estudadas, o próprio algoritmo *slice* neste contexto apresenta avanços significativos.

O presente capítulo mostra como as técnicas de alto desempenho são utilizadas para reduzir o custo de armazenamento e ainda como a MVD pode ser distribuída. As técnicas aqui empregadas são relacionadas a plataforma de execução pretendida, no caso, um aglomerado de computadores. Por esta razão a seção 3.1, recapitula as principais características deste tipo de plataforma, assim como as técnicas de programação consagradas em tais ambientes. Para tornar a leitura mais clara, os algoritmos *shuffle* e *slice*, são apresentados respectivamente nas seções 3.3 e 3.4 separadamente. Para cada versão de alto desempenho de um algoritmo são propostas duas abordagens de distribuição de carga (escalonamento).

3.1 Aglomerados de Computadores

Um aglomerado de computadores (conhecida pela sigla internacional NOW, *Network of Workstations*) é uma rede dedicada a unir diversos computadores com o objetivo de cooperarem na resolução de problemas complexos. A conexão das máquinas é realizada utilizando-se uma infraestrutura de rede e *software* especial.

O que caracteriza a arquitetura de um aglomerado é cada computador possuir seu espaço de endereçamento exclusivo. Uma vez que a memória não é compartilhada, cada conjunto de dados relevante para o cálculo deve ser atribuído a uma das partições do espaço total de memória existente. Desta forma, os dados precisam ser explicitamente particionados. Ao mesmo tempo que essa característica adiciona complexidade na programação ela encoraja o desenvolvedor a explorar a localidade. Localidade é a características de aproximar o dado relevante ao processamento de cada tarefa do processador que está executando a tarefa [23]. Trantando-se portanto de uma característica importante para atingir bons resultados em plataformas de alto desempenho. Uma vez que permite aos processadores explorarem ao máximo o uso de suas memórias *cache* [23, 35].

Das características citadas anteriormente surge um fator que deve ser levado em consideração no desenvolvimento de aplicações em aglomerados: sempre que dois processos trocam dados é necessário realizar um processo de sincronização. O processo de sincronização é realizado utilizando-se primitivas para envio e recebimento de mensagens. Tais primitivas são a base para o desenvolvimento de aplicações paralelas em aglomerados. A terminologia troca de mensagens serve para denominar o paradigma utilizado na construção dessas aplicações.

As soluções de alto desempenho apresentadas neste trabalho utilizam uma plataforma do tipo aglomerado de computadores. O padrão de comunicação adotado é MPI [23, 35] (do inglês, *Message Passing Interface*) um padrão consolidado para troca de mensagens em aglomerados de computadores [24]. Na programação por troca de mensagens são criados diversos processos. Cada processo sendo designado a um computador. MPI provê primitivas para cada processo receber um identificador único e conhecer o número total de processos em uma dada execução. Um resumo das primitivas MPI utilizadas nesse trabalho e o seu significado é apresentado a seguir:

- `MPI_Comm_rank` - retorna o identificador único do processo;
- `MPI_Comm_size` - retorna o número total de processos executando a aplicação;
- `MPI_Send` - envia dados para um determinado processo;
- `MPI_Recv` - recebe dados de um determinado processo;
- `MPI_Bcast` - transmite dados para todos os processos do sistema;

- MPI_Reduce - recebe dados de todos os processo do sistema aplicando uma operação algébrica;

A primitiva MPI_Send sincroniza com a primitiva MPI_Recv e vice-versa. Tratando-se portanto de uma das principais primitivas envolvidas na implementação dos algoritmos para aglomerados de computadores. MPI_Bcast sincroniza diversos processos, funcionando como uma barreira, onde ao final todos os processos recebem o mesmo conjunto de dados. MPI_Reduce agrupa dados de diversos processo em um único processo ao mesmo tempo que realiza uma operação algébrica sobre os dados transmitidos.

Os aspectos de programação utilizando MPI se tornam relevantes para a compreensão do presente trabalho. Importante ressaltar que cada processo conhece o número total de processos (MPI_Comm_size) e o seu identificador único em uma execução (MPI_Comm_rank). Assim, a comunicação pode ser facilmente implementada sem a configuração de parâmetros de baixo nível dependentes do protocolo de comunicação, tais como porta TCP ou endereço IP das máquinas. Uma vez que MPI encapsula as primitivas básicas de comunicação, o uso dessa biblioteca permite uma fácil adaptação a diferentes tecnologias de rede.

3.2 MVD em Paralelo

A idéia da paralelização da Multiplicação Vetor-descritor (MVD) é distribuir algebricamente a multiplicação do vetor dentro do somatório. Desta forma, temos por exemplo a possibilidade de executar a multiplicação do vetor por cada um dos termos de forma independente, como pode ser visto na equação 11. Todavia, a maneira de distribuição da multiplicação depende da propriedade algébrica utilizada pelo algoritmo. Isso diferencia as versões paralelas do *shuffle* e do *slice*.

$$\left(\sum_{k=1}^{m} \bigotimes_{i=1}^{N} Q_k^i \right) x = \sum_{k=1}^{m} \left(\bigotimes_{i=1}^{N} Q_k^i \, x \right) \tag{11}$$

A MVD ocorre diversas vezes (tipicamente centenas ou milhares de vezes) para resolver um modelo. Todavia, a paralelização dos algoritmos *shuffle* e *slice*, leva em consideração que a cada passo os processos calculam diversos resultados parciais. Precisamente, cada processo p calcula um vetor parcial x_p. O algoritmo 1 ilustra como funciona a paralelização de uma iteração onde a MVD é realizada. Na notação do algoritmo 1, assim como em outros algoritmos deste trabalho, adota-se a variável *procs* para determinar o número total de processos existentes no ambiente de execução (equivalente a uma chamada da função MPI_Comm_size) e a variável p para determinar cada processo (equivalente a uma chamada da função MPI_Comm_rank) [24].

Algoritmo 1 Estratégia de paralelização em alto nível.

1: *//Enquanto não atingi um critério de parada realiza a* MVD
2: **enquanto** *erro* $<$ *mínimo* **OU** *iteração* $>$ *máximo* **faça**
3: *//Cada processo p calcula vetor parcial* x_p^k
4: *//Cada processo p envia vetor parcial* x_p^k *para processo* 0
5: *//Processo* 0 *calcula novo vetor* $x^{k+1} = \sum_{p=0}^{procs} x_p^k$
6: *//Processo* 0 *envia* x^{k+1} *os outros processos*
7: **fim enquanto**

Repare que na linha 2 do algoritmo 1, a MVD é realizada por um número indeterminado de iterações. Todavia, para solucionar modelos com um grau de confiança razoável, o erro aceito é em torno de 10^{-9} o que exige um número elevado de iterações [18]. Alguns métodos iterativos para a solução de sistemas de equações lineares podem não convergir para um resultado com o erro mínimo estipulado. Quando isso ocorre, os valores do vetor oscilam sem uma tendência clara a cada passo. Para evitar que o método execute por um número indeterminado de iterações, na linha 2, existe um número máximo de iterações toleráveis para que o sistema chegue a um resultado. Caso este número seja atingido, mesmo que a resposta ainda não tenha a precisão desejada, o algoritmo termina.

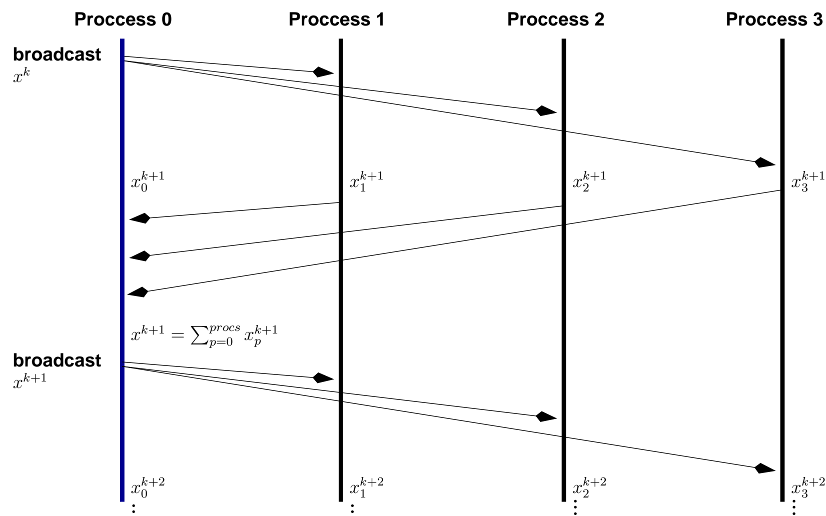


Figura 5: Diagrama de comunicação para a MVD em paralelo.

Na linha 3 do algoritmo 1, cada processo p multiplica uma parte do descritor pelo vetor da iteração atual (x^k) gerando parte do vetor da próxima iteração (x_p^{k+1}). A parte do descritor que cada processo multiplica varia dependendo do algoritmo. Porém, a multiplicação de cada termo de forma independente pode ser realizada diretamente, como visto na equação 11. Para obter-se o vetor da próxima iteração, é necessário somar todos os vetores parciais (x_p^{k+1}). Assim, nas linhas 4 e 5, cada processo envia o seu vetor parcial para o processo de identificação 0 e este fica encarregado de somar os vetores paraciais (x_p^{k+1}) construindo o vetor da próxima iteração x^{k+1}. Ao final, na linha 6, o processo

0 envia o vetor da próxima iteração (x^{k+1}) para todos os outros processos (ou seja, em *broadcast*). Ao final deste algoritmo, cada processo possui o vetor da próxima iteração e uma nova etapa de MVD pode iniciar.

O resultado do procedimento da MVD realizada em paralelo é que a cada iteração os processos recebem o vetor da iteração atual x^k, cada processo multiplica este por uma parte do descritor gerando um vetor parcial da próxima iteração x_p^{k+1}. O processo de menor identificação (processo 0) acumula os vetores parciais gerando o vetor da próxima iteração (x^{k+1}) e este é transmitido para todos os outros processos. Este ciclo se repete até que o resultado atinja um erro mínimo ou um número máximo de iterações previamente estipulados. A figura 5, ilustra como funciona a comunicação entre os processos em alto nível. Neste diagrama, a cada iteração k o vetor é transmitido para todos os processos que calculam resultados parciais. Os resultados parciais são posteriormente somados no processo 0, gerando o vetor da próxima iteração que novamente é enviado a todos processos iniciando um novo ciclo.

3.3 Shuffle Paralelo

O cálculo específico que será realizado por cada processo é independente da estratégia apresentada no algoritmo 1. Dependendo do algoritmo utilizado, *shuffle* ou *slice*, as possibilidades de distribuir o processamento variam. Dessa maneira, o processamento realizado por cada processo, linha 3, é diferente em cada caso específico. Apresentaremos a seguir a estratégia de paralelização para o algoritmo *shuffle* onde são discutidas duas abordagens de escalonamento (distribuição de carga).

3.3.1 Abordagem de Escalonamento Simples

Na primeira abordagem, cada termo constitui uma tarefa independente e o número total de termos é conhecido. Uma vez que o descritor é a soma de diversos termos, podemos efetuar a multiplicação de cada termo pelo vetor de forma independente. Assim, é possível distribuir o laço que distribuí a multiplicação do vetor por cada termo.

Algoritmo 2 *Shuffle* utilizando abordagem de escalonamento sem considerar custos.

1: *//Cada processo p começa a computar o termo de mesmo índice*
2: $t = p$
3: *//Enquanto existem termos para computar*
4: **enquanto** $t < terms$ **faça**
5: *//Aplica o shuffle no termo t e no vetor atual* x^k*, acumula o resultado em* x_p^{k+1}
6: $x_p^{k+1} = x_p^{k+1} +$ Shuffle(x^k, t)
7: *//Calcula o próximo termo de p baseando-se no número total de processos*
8: $t = t + procs$
9: **fim enquanto**

No algoritmo 2, é apresentado como a divisão dos termos que cada processo deverá computar é realizada. Nesse algoritmo, *terms* representa o número total de termos do descritor executando em um ambiente com *procs* processos, cada um identificado pela variável p. Para garantir que não haja cálculo redundante, em outras palavras, que dois ou mais processos não realizarão o mesmo cálculo, cada processador p começa calculando o termo (ou tarefa) de mesmo índice que a identificação do processo, linha 2. Esse índice recebe, a cada execução do laço o número total de processos (*procs*) mais ele mesmo $t = t + procs$, linha 8. Assim, garanti-se que as tarefas designadas a cada processo não coincidam. Mesmo quando a divisão do número de tarefas pelo número de processadores não é exata, esse algoritmo é capaz de distribuir o resto da divisão.

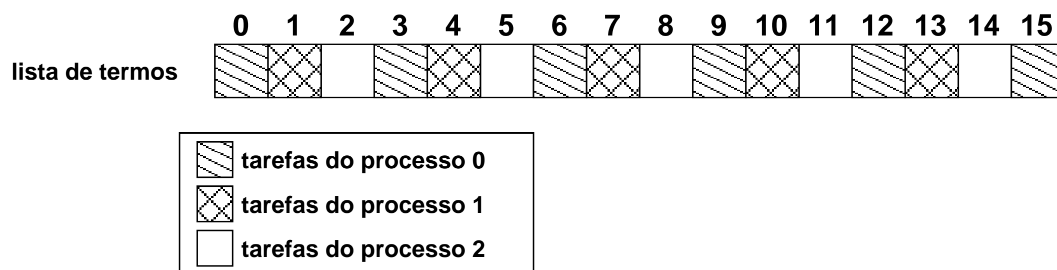


Figura 6: Abordagem de distribuição de carga para o *shuffle* sem considerar custos.

Para exemplificar esta abordagem de escalonamento, apresentamos uma situação de execução hipotética com 16 termos (numerados de 0 a 15) sendo executado por 3 processos (identificados de 0 a 2), vista na figura 6. Na figura cada índice da lista de termos é designado para um processo, onde o padrão de preenchimento indica o processo para o qual um termo foi designado. Nesse caso, o primeiro processo (identificador 0) receberia os termos de índices 0, 3, 6, 9, 12 e 15 (prenchidos com ▧). O segundo processo, de identificador 1, receberia os termos de índice 1, 4, 7, 10 e 13 (prenchidos com ⊠). O terceiro processo receberia os termos de índices 2, 5, 8, 11 e 14 (prenchidos com □). Garantindo que os processos não realizem cálculo redundante e recebam aproximadamente o mesmo número de tarefas. Repare que esse algoritmo não impõe restrições ao número de processos que deverão executar uma aplicação. No entanto, não faz sentido executar a aplicação com um número de processos maior que o número de tarefas (ou termos). Pois nessa situação, algum processo não realizaria cálculo.

Apesar dessa abordagem apresentar um método direto e eficaz de realizar a distribuição de carga, o fato de não considerar o custo das tarefas pode ocasionar, em determinadas situações onde os termos possuem custos muito discrepante, um resultado pouco eficiente. Apresentaremos a seguir uma versão de escalonamento para o *shuffle* que leva em conta o custo computacional das tarefas.

3.3.2 Escalonamento Considerando Custos

A segunda abordagem de escalonamento considera os custos computacionais envolvidos no cálculo de cada uma das tarefas. Uma vez que o número de elementos nulos varia em cada uma das matrizes, a multiplicação de cada termo pelo vetor apresenta tipicamente diferentes custos de um termo para outro. Nesse sentido, foi proposto um algoritmo que tenta equilibrar a carga total de trabalho em cada processo. A idéia no algoritmo é criar uma lista de tarefas (ou seja, termos) contendo o custo e índice de cada um. Posteriormente, ordena-se esta lista por custo em ordem decrescente. Uma vez que a lista está ordenada, se percorre a lista designando uma tarefa sempre ao processo com menor carga.

Algoritmo 3 *Shuffle* utilizando segunda abordagem de escalonamento, considerando custos.

1: *//Gera lista contendo o índice e o custo de cada tarefa*
2: **para** $t = 0$ **até** $terms - 1$ **faça**
3:     *//Calcula custo de processamento de cada termo, usando equação 9*
4:     ListaDeTarefas[t].custo = $\prod_{i=1}^{N} n_i^{(t)} \times \sum_{i=1}^{N} \frac{nz_i^{(t)}}{n_i^{(t)}}$
5:     *//Associa o termo correspondente a tarefa*
6:     ListaDeTarefas[t].indice = t
7: **fim para**
8: *//Ordena a lista de tarefas por custo em ordem decrescente*
9: OrdenaDecrescente(ListaDeTarefas)
10: **para** $t = 0$ **até** $terms - 1$ **faça**
11:     $menosCarregado = 0$
12:     *//Encontra o processo com menos carga*
13:     **para** $p = 0$ **até** $procs - 1$ **faça**
14:         **se** (Carga[p] < Carga[menosCarregado]) **então**
15:             menosCarregado = p
16:         **fim se**
17:     **fim para**
18:     *//Adiciona na lista de tarefas do processo com menos carga a tarefa atual*
19:     Tarefas[menosCarregado].adiciona(ListaDeTarefas[t].indice)
20:     *//Atualiza a informação de carga total desse processo com a carga da tarefa atual*
21:     Carga[menosCarregado] = Carga[menosCarregado] + (ListaDeTarefas[t].custo)
22: **fim para**

O algoritmo 3, apresenta essa idéia de escalonamento de forma detalhada. Primeiramente, o custo computacional do cálculo de cada termo é realizado na linha 4 utilizando-se a equação 9 apresentada no capítulo 2. A medida que o custo de cada termo é calculado os termos são inseridos numa lista com seus respectivos índices, linha 6. Após esse procedimento, ordena-se a lista de tarefas por custo em ordem decrescente, colocando a tarefa de maior custo no início da lista[1]. As etapas descritas anteriormente, tornam possível

[1]A implementação utiliza o método de ordenação *quicksort* que apresenta complexidade $O(nlogn)$ [28].

percorrer a lista de termos de forma a atribuir cada tarefa ao processo com menor custo, procedimento realizado nas linhas 10 a 22. Nesse laço, a cada iteração, seleciona-se o processo com menor carga, linhas 13 a 17, atribuindo-lhe a tarefa. Uma vez que a tarefa é atribuída, linha 19, atualiza-se a carga total do processo, linha 21. Esse procedimento é repetido até que todas as tarefas tenham sido designadas aos processos.

Ao final desse procedimento, as tarefas designadas a cada processo são conhecidas por todos. Isso implica que todos os processos realizam o algoritmo de distribuição de carga simultaneamente. Com isso, acrescenta-se uma parcela de pré-processamento realizada antes da MVD. Entretanto, este processamento não possui um custo significativo pois, na maioria dos casos, o número de termos é tipicamente pequeno.

3.4 Slice Paralelo

O algoritmo *slice* utiliza a propriedade da decomposição de um produto tensorial em fatores normais unitários aditivos (ou fatores). Um fator é basicamente um elemento composto pela multiplicação sucessiva de apenas um elemento não nulo de cada matriz de um termo. Esta propriedade foi apresentada no capítulo 2, equação 10. A idéia que deu origem ao algoritmo *slice* foi a de realizar a geração dos fatores em um pré-processamento guardando apenas estes e a última matriz de cada termo.

Para realizar o mapeamento dos elementos da última matriz de um termo no DM, guarda-se o índice desta e associa-se um índice a cada fator (AUNF) juntamente com o resultado das multiplicações sucessivas. Em outras palavras, cada fator é composto por três informações (E, i, j), onde E é o resultado das multiplicações sucessivas de cada elemento não nulo das $N-1$ primeiras matrizes de um termo; i, j são respectivamente os índices de linha e coluna deste elemento na matriz gerada resultante do produto tensorial entre as $N-1$ matrizes de um termo.

A geração dos fatores (AUNFs) é realizada em uma etapa de pré-processamento uma única vez. Portanto, essa etapa não representa uma porção significativa do cálculo. Uma vez gerada a lista de fatores, a mesma é armazenado e utilizada a cada etapa da MVD. Independente do número total de iterações necessárias para a resolução dos modelos esses valores permanecem constantes. A presente abordagem apresenta um custo de memória diferente do *shuffle* que armazena apenas as matrizes utilizando compactação HBF[2], pois uma vez que a lista de fatores é gerada torna-se necessário armazenar somente essa e a última matriz de cada termo.

[2]HBF é um formato compacto para armazenamento de matrizes, a sigla faz menção as pessoas envolvidas na construção do formato: Harwell Boeing Form.

3.4.1 Primeira Abordagem de Escalonamento

Essa abordagem é bastante semelhante a primeira técnica utilizada para o *shuffle*. Apesar de não explorar os aspectos positivos da divisão de tarefas em um grão menor proporcionadas pelo *slice*, tal abordagem se torna atraente por permitir que a geração dos AUNFs de cada termo possa ser realizada de forma distribuída.

Como a geração dos fatores é realizada para cada termo e cada termo é uma tarefa independente, não é necessário que todos os processos realizem a etapa de pré-processamento. Cada processo pode computar apenas a lista de fatores dos termos que irá efetivamente multiplicar a cada iteração. Desta forma, antes de iniciar o procedimento iterativo que realiza diversas vezes a MVD (algoritmo 1), pode-se realizar o pré-processamento de forma distribuída. Devido a considerar cada termo como uma tarefa, essa abordagem assemelha-se bastante com a primeira abordagem de escalonamento proposta para o *shuffle*.

Algoritmo 4 *Slice* utilizando abordagem simples de escalonamento

```
1: //Cada processo p efetua o pré-processamento dos termos relevantes
2: t = p
3: enquanto t < terms faça
4:    //Realiza a geração dos fatores para o termo
5:    ListaFatoresNormais[t] = calculaFatoresNormais(t)
6:    t = t + procs
7: fim enquanto
8: //Enquanto não atingi um critério de parada realiza a MVD
9: enquanto erro < mínimo OU iteração > máximo faça
10:    t = p
11:    enquanto t < terms faça
12:       //Aplica o slice no termo t e no vetor atual x^k, acumula o resultado em x_p^k
13:       x_p^k = x_p^k + slice(ListaFatoresNormais[t], t, x^k)
14:       t = t + procs
15:    fim enquanto
16: fim enquanto
```

De forma geral, o algoritmo 4 mostra como a estratégia de paralelização pode incluir o pré-processamento. As linhas 1 a 7, mostram a geração de uma lista de fatores normais para cada termo t. Posteriormente, nas linhas 8 a 16, a resolução do sistema é realizada por diversas iterações onde a cada passo o processamento é realizado de forma idêntica ao apresentado na primeira abordagem de escalonamento do *shuffle* (linhas 10 a 15).

A principal vantagem desta abordagem é não existir interdependência entre as tarefas, possibilitando o particionamento do conjunto de dados. Pois cada termo somente precisa ser armazenado pelo processo ao qual foi designado. Entretanto, esta abordagem não se beneficia da característica do *slice* de poder quebrar as tarefas em grãos menores. A próxima abordagem visa justamente implementar um algoritmo de balanceamento de carga com essa característica.

3.4.2 Escalonamento por AUNF

O objetivo principal dessa abordagem é considerar cada fator (AUNF) como sendo uma tarefa independente das outras. A motivação para tal abordagem é a possibilidade de quebrar o problema em unidades de dados com menores custos de processamento. Em outras palavras, aumenta-se o número de tarefas reduzindo o custo computacional de cada uma.

A idéia central é criar uma lista de fatores que serão multiplicados por cada processo. Assim, durante a etapa de pré-processamento, é possível atribuir cada fator (tarefa) a um processo, uma vez que o número total de tarefas é conhecido a priori. Apesar de tornar o grão da aplicação menor, essa abordagem impossibilita a distribuição da etapa de pré-processamento.

Algoritmo 5 *Slice*, detalhe do pré-processamento, considerando cada AUNF como uma tarefa.

1: *//Computa o número total de fatores*
2: $TotalDeAunfs = 0$
3: **para** $t = 0$ **até** $terms - 1$ **faça**
4: $produto = 1$
5: $N =$ t.NumeroDeMatrizes
6: *//Da primeira a penúltima matriz*
7: **para** $i = 0$ **até** $N - 2$ **faça**
8: *//Efetua o produtório dos elementos não nulos de cada matriz i do termo*
9: $produto =$(t.pegaMatriz(i)).NumNaoNulos()$\times produto$
10: **fim para**
11: $TotalDeAunfs = TotalDeAunfs + produto$
12: **fim para**
13: *//Calcula a divisão inteira do número de fatores*
14: Fatores =TotalDeAunfs $\div$ *procs*
15: *//Calcula resto da divisão*
16: Resto =TotalDeAunfs % *procs*
17: *//Calcula quantos fatores são designados a cada processo p*
18: **para** $p = 0$ **até** $procs - 1$ **faça**
19: *//Recebe a divisão inteira do número de tarefas*
20: TotalDeTarefas[p] = Fatores
21: *//Se a divisão não é exata, recebe parte do resto*
22: **se** ($p <$ Resto) **então**
23: TotalDeTarefas[p] = TotalDeTarefas[p] + 1
24: **fim se**
25: **fim para**

No algoritmo 5, é mostrado como o pré-processamento é realizado de forma a gerar uma lista de tarefas para cada processo. Após a execução de tal etapa, ainda no pré-processamento, é efetuado a geração dos fatores. Neste processo, a última matriz de cada termo é associada a cada tarefa. Garantindo assim que a multiplicação dos fatores pela

última matriz de cada termo possa ser realizada de forma independente do termo ao qual estão associadas.

Algoritmo 6 *Slice* utilizando abordagem que considera cada AUNF como uma tarefa.

1: *//Cada processo p percorre a sua lista de tarefas*
2: **para** $tarefa = ListaDeTarefas[p][0]$ **até** $UltimaTarefa$ **faça**
3: *//Aplica o slice na tarefa atual gerando o vetor parcial (x_p^{k+1}) da próxima iteração*
4: $x_p^{k+1} = x_p^{k+1} +$ Slice(x^k, $tarefa$)
5: **fim para**

Utilizando essa idéia, o cálculo que cada processo efetua em uma iteração fica como no algoritmo 6. Nesse algoritmo os processos consideram como cada tarefa independente um AUNF o que propicia uma distribuição de carga mais justa. Dividindo-se os termos em pedaços menores pode-se realizar uma distribuição de carga bastante precisa, na qual a diferença de processamento de um processo para outro não afeta de maneira significativa o desempenho geral da aplicação.

Capítulo 4

Resultados

Os resultados obtidos pelas implementações dos algoritmos propostos no presente trabalho serão apresentados através de gráficos de aceleração (*speedup*). Essa métrica visa comparar o tempo de execução da aplicação seqüencial com o obtido após a paralelização [35]. Formalisando, se uma aplicação seqüencial leva um tempo T_{seq} para uma determinada entrada de dados, e executando-a em paralelo com p processadores a mesma aplicação leva um tempo T_p, diz-se que o *speedup* obtido com esta aplicação é dado pela razão entre o tempo seqüencial e o tempo paralelo, como apresentado na equação 12.

$$speedup = \frac{T_{seq}}{T_p} \tag{12}$$

Os resultados apresentados foram obtidos utilizando a média de 10 execuções cada uma realizando 100 iterações. O tempo de uma iteração foi medido utilizando o tempo real da execução com um cronômetro interno à aplicação. Neste tempo foi incluído o tempo necessário para a transmissão do vetor e dos vetores parciais. A qualidade dos resultados apresentados foi garantida através da observação do desvio padrão. Por razões de clareza e espaço os detalhes de cada execução não são apresentados nesta seção mas no apêndice C. Nesse apêndice são apresentados a média, o desvio padrão, o *speedup* e a eficiência para cada execução realizada.

4.1 Plataforma de Teste

A plataforma de testes utilizada é um aglomerado de computadores, *cluster*, denominado I-CLUSTER2. Esta máquina foi o primeiro supercomputador francês baseado na família de processadores Itanium-2 de 64 bits. O desempenho do I-CLUSTER2 foi medido utilizando o *benchmark* Linpack. Atingindo a marca de 561 *GigaFlops* (bilhões de operações de ponto flutuante por segundo). Em novembro de 2003, esta máquina ficou em 283°

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 9 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix} +$$

$$\begin{bmatrix} 2 & 6 & 7 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 9 & 0 \\ 1 & 0 & 0 & 2 \end{bmatrix} \otimes \begin{bmatrix} 2 & 0 & 0 \\ 3 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} +$$

Figura 7: Exemplo de descritor com dois termos, cada um com três matrizes.

lugar na lista das 500 máquinas mais velozes do mundo (TOP 500) [26].

Cada um dos 104 nós desta plataforma possui dois processadores Itanium-2[1] de 64 bits operando a 900 Mhz. Equipados com 3 GB de memória e 72 GB de disco rígido. Os computadores são interconectados por uma rede de baixa latência MYRINET. No total, o I-CLUSTER2 é composto por 208 processadores, 312 GB de memórias, somando uma capacidade de armazenamento em disco de 7,5 TB (TeraBytes). Em relação ao software, cada computador que compõe o ICLUSTER-2 utiliza sistema operacional Linux (kernel 2.4.21-32.0.1.EL global) com a distribuição Red Hat (Enterprise Linux AS 3). A biblioteca utilizada para a programação MPI é LAMMPI.

Para garantir que outras aplicações não interferissem na execução dos testes, pelo uso da rede, o ICLUSTER-2 foi utilizado exclusivamente durante os testes. Desta forma, afirmamos que não havia transmissões de outras aplicações ocorrendo durante cada execução. Para mitigar o efeito de *cache* a cada experimento, o ambiente de execução era restaurado. Isto foi realizado utilizando o comando `lamclean` que remove arquivos temporários, desalocando recursos e cancelando registros de processos. A utilização deste comando garante que a cada execução realizada o estado de cada um dos nós é semelhante ao obtido após uma reinicialização dos `daemons` do MPI.

4.2 Casos de Teste

Como descrito no capítulo 2, a entrada da aplicação é um descritor markoviano (DM) que consiste na soma consecutiva de uma série de termos produto-tensoriais. Estes termos são carregados utilizando como entrada um arquivo composto por diversas matrizes. Supondo um descritor com 2 termos, cada um contendo 3 matrizes como visto na figura 7, o arquivo de entrada seria como visto na figura 8. Neste formato de entrada, uma linha precedida por "#" indica um comentário.

Foram utilizados quatro casos de teste para verificar o desempenho obtido com as implementações paralelas propostas nesse trabalho. Dois casos de teste foram estudos de caso reais de modelagem analítica estruturada utilizando o formalismo SAN. O primeiro

[1] Estes processadores possuem cada um uma memórica *cache* de 3MB.

```
# termo tensorial 0
3 # quantidade de matrizes que compõe este termo

4 # ordem da primeira matriz
1 0 1 0
0 1 1 0
0 0 9 0
0 0 1 2

3 # ordem da segunda matriz
1 0 0
0 4 0
0 0 1

3 # ordem da terceira matriz
1 0 1
0 1 0
3 0 1

# termo tensorial 1
3 # quantidade de matrizes que compõe este termo

4
2 6 7 0
0 0 8 0
0 0 9 0
1 0 0 2

3
2 0 0
3 1 0
0 0 1

3
0 0 0
4 0 0
0 0 0
```

Figura 8: Exemplo de arquivo de entrada para descritor da figura 7.

modelo real é referenciado durante o texto por seção crítica (SC) pois representa a modelagem do problema onde 16 recursos são disputadas por 4 processos. O segundo, batizado como MISTO, modela 9 módulos que exploram diversos parâmetros. Uma descrição gráfica mais detalhada de tais modelos pode ser visualizada no apêndice B.

Um parâmetro relevante para determinar o aumento de desempenho de uma aplicação paralela é a quantidade de cálculo distribuída. Esse parâmetro se torna ainda mais relevante em aglomerados de computadores, uma vez que nessas arquiteturas a comunicação representa um fator limitante da aceleração. A quantidade de cálculo realizada por cada algoritmo varia de forma significativa. Porém, existe um fator que colabora para o aumento de complexidade simultaneamente em ambos algoritmos. Se um dado modelo possui o mesmo número de matrizes em cada termo e se a ordem destas matrizes são as mesmas, o fator relevante dentro do escopo do trabalho que diferenciará estes modelos será o número de elementos não nulos nas matrizes. Desta forma, defini-se que a esparsidade de um descritor é dada pela média da razão entre o número de elementos não nulos (nz_i) e o número total de elementos (${n_i}^2$) de cada matriz i. O resultado desta operação é a porcentagem de elementos não nulos geral do descritor. A equação 13, mostra como a esparsidade é calculada em um descritor composto por T termos, cada um com N matrizes, onde $nz_i^{(k)}$ e $n_i^{(k)}$ representam respectivamente o número de elementos não nulos e a ordem da i-ésima matriz do k-ésimo termo.

$$\left(\frac{\sum_{k=1}^{T} \sum_{i=1}^{N} \frac{nz_i^{(k)}}{n_i^{(k)} \times n_i^{(k)}}}{N \times T} \right) \times 100 \tag{13}$$

Tabela 2: Detalhamento dos casos de teste

Teste	Nº de Termos	Esparsidade	Nº de AUNFs	Tempos Seqüenciais (s)	
				Shuffle	*Slice*
SC	32	46,25 %	9469952	477,28	13,68
MISTO	16	19,88 %	3280080	52,43	5,95
DENSO A	16	22,13 %	10649600	57,11	13,14
DENSO B	16	27,52 %	48771072	80,21	64,45

Assim, mais dois testes foram elaborados com base no modelo MISTO com o intuito de verificar o comportamento dos algoritmos em situações que envolvessem um grande volume de multiplicações. Para gerar esses dois modelos, chamados de DENSO A e DENSO B, foram acrescentados elementos não nulos aleatoriamente nas matrizes do modelo MISTO.

A tabela 2 mostra alguns parâmetros relevantes de cada caso de teste. O primeiro parâmetro é o número de termos de cada descritor. Esse parâmetro se torna relevante por constituir uma tarefa nos algorimtos *shuffle* (utilizando ambas abordagens de escalonamento) e *slice* (primeira abordagem), limitando o número de processadores que podem ser utilizados nesses casos. Na segunda coluna, o parâmetro esparsidade mostra a porcentagem média de elementos não nulos das matrizes de cada descritor. A terceira coluna apresenta o número de fatores (AUNF) relevante por constituir o grão (uma tarefa indivisível) na segunda abordagem de escalonamento do algoritmo *slice*. A quarta e a quinta coluna mostram os tempos de execução seqüencial (ou seja, com apenas um processador) necessários para realizar um passo da MVD utilizando os algoritmos *shuffle* e *slice* respectivamente. O tempo de execução do *slice* neste caso **não considera** o tempo de pré-processamento.

Nas próximas seções, os resultados obtidos com as abordagens de escalonamento de ambos os algoritmos são apresentados. Na seção que aborda os resultados obtidos com o algoritmo *slice* os tempos de pré-processamento são apresentados. Por uma questão estrutural a comparação entre os dois algoritmos que realizam a MVD é realizada no capítulo seguinte.

4.3 Resultados Shuffle

Na presente seção são mostrados os resultados obtidos com as implementações do *shuffle* abordadas no capítulo 3. Os resultados são apresentados de forma a traçar as principais diferenças entre os algoritmos de escalonamento propostos comparando-se o tempo de execução obtido com um determinado número de processadores. Cada gráfico traça uma relação entre a aceleração obtida (eixo y) e o número de processadores utilizados (eixo x). O número total de processadores utilizados é limitado pelo número de termos

nas implementação do *shuffle*.

A figura 9 apresenta dois gráficos com os resultados para as duas técnicas de escalonamento apresentadas para o *shuffle* utilizando o caso de teste SC. A primeira técnica, mostrada na figura 9 (A), mostra valores de aceleração próximos do ideal em alguns pontos. Esses resultados já apresentam-se satisfatórios se considerado a simplicidade dessa solução. Por outro lado, observando a curva da técnica que considera custos (figura 9 (B)), conclui-se que essa abordagem, para este caso de teste, não apresenta vantagem. Outro aspecto que chama atenção é que em determinados pontos ambas as curvas apresentam uma estagnação nos valores de aceleração. Por exemplo, utilizando 16, 17, 18, 19, 20 e 21 processadores o *speedup* permanece praticamente constante. Isso se repete em outros intervalos, como: 13 a 15 e 22 a 31. Acredita-se que esses pontos de estagnação sejam devido a distribuição de carga. Essa afirmação pode ser comprovada observando-se que, nos pontos cujo o número de processadores é um múltiplo do número de tarefas (4, 8, 16 e 32), a aceleração está bastante próxima do ideal.

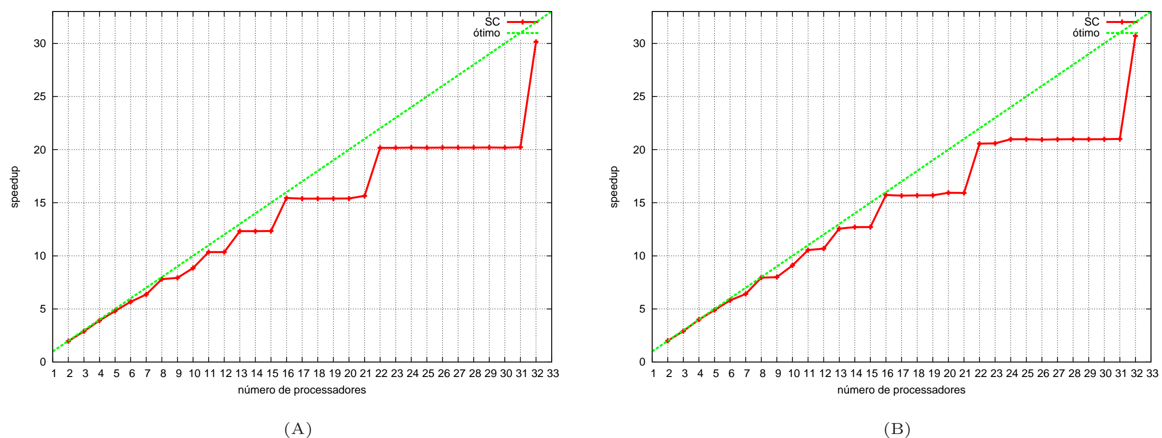


Figura 9: Aceleração do *Shuffle*, sem considerar custos (A) e considerando custos (B) para o teste SC.

Na figura 10 observam-se os gráficos que apresentam os resultados para as duas abordagens de escalonamento utilizando o caso de teste Misto. Semelhantemente aos resultados analisados anteriormente, ambas as técnicas apresentaram acelerações significativas. Novamente, percebe-se um comportamento bastante parecido. O que chama atenção nesses dois gráficos é o *speedup* obtido com 8 processadores com os algoritmos de escalonamento. Sem considerar custos, figura 10 (A), a aceleração com 8 processadores é maior do que usando 7 processadores com essa mesma versão. Já considerando custos, figura 10 (B), o resultado com 8 processadores se monstra inferior ao obtido com 9. Esses aspectos reforçam a impressão de que, na abordagem que considera custos, a relação entre a quantidade de termos não precisa ser um múltiplo do número de processadores.

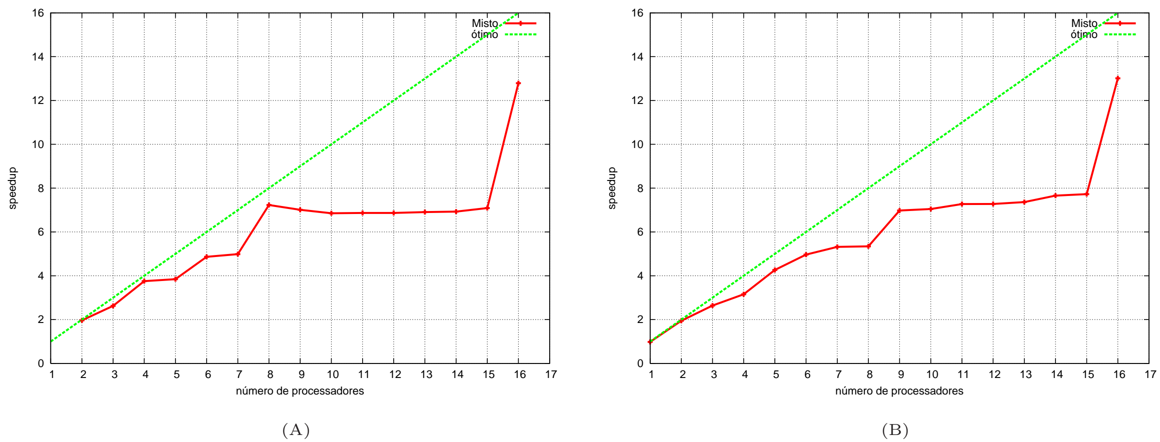


Figura 10: Aceleração do *Shuffle*, sem considerar custos (A) e considerando custos (B) para o teste MISTO.

A figura 11 e a figura 12 apresentam os resultados obtidos respectivamente com os casos de teste DENSO A e DENSO B utilizando o *shuffle*. Observe que para esses dois testes as acelerações são praticamente as mesmas, independente da técnica de escalonamento utilizada. Isto acontece devido ao fato que todas as tarefas possuem o mesmo custo nesses casos de teste. A idéia desses dois testes é demonstrar que mesmo com a abordagem que considera custos, o algoritmo *shuffle* pode apresentar um comportamento pouco escalável quando os termos possuem custos muito semelhantes. Isso acontece também devido ao tamanho do grão ser sempre limitado pelo número total de termos.

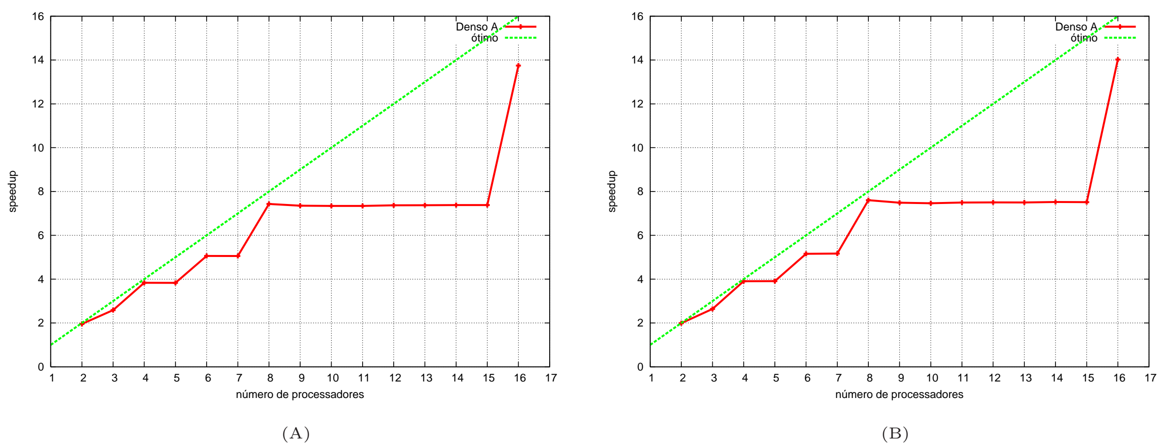


Figura 11: Aceleração do *Shuffle*, sem considerar custos (A) e considerando custos (B) para o teste DENSO A.

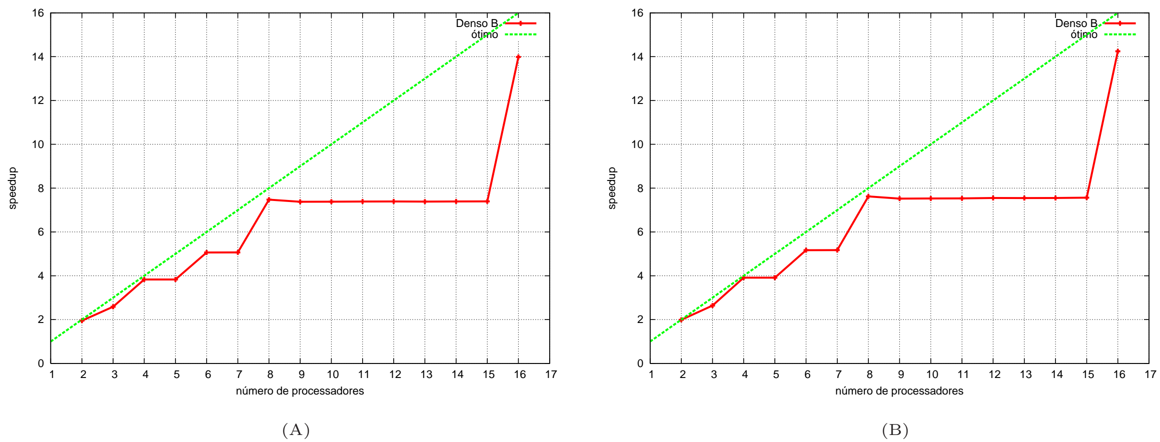


Figura 12: Aceleração do *Shuffle*, sem considerar custos (A) e considerando custos (B) para o teste Denso B.

A figura 13 apresenta o detalhe da carga atribuída a cada processo para algumas configurações do caso de teste Misto. A carga é quantificada em número de multiplicações em ponto flutuantes. Nesta figura, os três gráficos na parte superior apresentam o detalhe do cálculo realizado com 7, 8 e 9 processos respectivamente da esquerda para a direita. A carga atribuída a cada processo é apresentada em milhares de operações de ponto flutuante (MMP). Com 7 processos, o desempenho é limitado por um processo que realiza aproximadamente 26 MMPs, já com 8 processos o balanceamento de carga se apresenta de forma mais eficiente uma vez que o processo com mais carga recebe em torno de 15 MMPs. Porém, ao aumentar o número de processos para 9 a abordagem de escalonamento não evolui. Como pode ser observado o processo mais carregado computa aproximadamente 15 MMPs também com 9 processadores limitando o aumento de desempenho geral da aplicação.

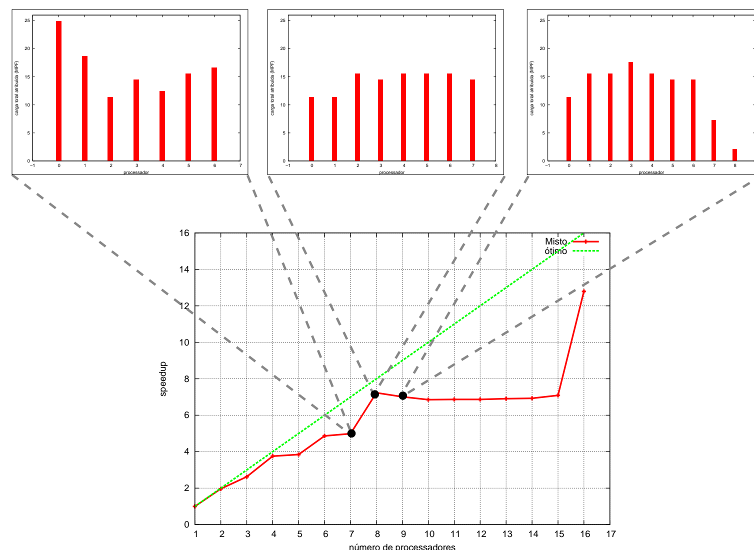


Figura 13: Detalhe do algoritmo de balanceamento de carga, *shuffle*, teste Misto.

Tabela 3: Comparativo entre custo de pré-processamento e o de uma iteração com o *slice*.

Teste	Pré-processamento (s)		Tempo Seqüencial (s)	
	Média	Desvio Padrão	Média	Desvio Padrão
SC	39,959692	0,012540	13,688620	0,052211
Misto	10,688436	0,065441	5,953364	0,001414
Denso A	36,805741	0,032112	13,014060	0,019974
Denso B	150,569193	0,081030	80,211800	0,091440

Logo, pode-se concluir que como o grão de cada tarefa para esses métodos sempre será grande, os resultados obtidos não apresentam uma boa escalabilidade quando o número de tarefas não é divisível pelo número de processadores. O desempenho geral da versão paralela do *shuffle* foi significativo. Entretanto, o comportamento da aceleração nos testes não evolui em alguns intervalos. Isso se deve as características intrínsecas deste algoritmo que não permite dividir um termo em pequenas tarefas. Um das principais razões que motivaram a paralelização do *slice* foi a possibilidade de quebrar os termos em mais tarefas.

4.4 Resultados Slice

Esta seção tem como objetivo apresentar os resultados obtidos com o algoritmo *slice* utilizando os mesmos casos de teste anteriores. Como já foi discutido no capítulo 3, o *slice* de alto desempenho inclui uma etapa de pré-processamento que deve ser realizada uma única vez. Nesse sentido, a utilização da primeira abordagem de escalonamento, que considera cada termo como uma tarefa independente, foi utilizada por permitir a distribuição desse cálculo.

A tabela 3 mostra os tempos de pré-processamento para quatro casos de teste utilizando a primeira abordagem de escalonamento. Nessa tabela, são apresentados também o custo computacional para realizar uma iteração da MVD. Apesar do custo de uma iteração ser menor que o custo de pré-processamento, o custo de processamento não é impactante no tempo total de execução. Isso porque são necessárias centenas de iterações para resolver um problema. O resultado é que o custo de pré-processamento é diluído ao longo de algumas iterações.

Por exemplo, supondo que para o caso de teste SC necessita-se de 200 iterações para o sistema de equações ser resolvido. Ao todo seriam necessários cerca de 2600 segundos (200 vezes o tempo seqüencial de aproximadamente 13 segundos) para a resolução do modelo. Conseqüentemente, nessa situação o tempo de pré-processamento, que para esse teste é aproximadamente 39 segundos, representaria pouco mais de 1% do tempo total de execução. No entanto, os métodos iterativos de resolução de sistemas de equações lineares não são determinísticos. Com isso, o número de iterações necessárias para realizar o pré-

processamento pode ser ou não ser significativo dependendo do modelo. Dessa forma, faz-se interessante paralelizar a etapa de pré-processamento para abranger tais situações.

A figura 14 mostra a aceleração obtida na etapa de pré-processamento quando utilizando a abordagem que considera cada termo como uma tarefa. Essa figura apresenta as acelerações obtidas com os casos de teste SC (figura 14 (A)), MISTO(figura 14 (B)), DENSO A (figura 14 (C)) e DENSO B (figura 14 (D)). Com o teste SC é possível observar que o desempenho é melhor quando a quantidade de processadores é múltiplo do número de tarefas (4, 8, 16 e 32 processos). O caso de teste MISTO apresenta um comportamento semelhante. É importante observar que nos casos de teste onde as tarefas possuem o mesmo custo, figura 14 C e D, que a aceleração apresenta um comportamento unicamente crescente. Isso ocorre porque nesses casos a distribuição de carga é uniforme, uma vez que, não existe praticamente diferença nos custos de cada termo.

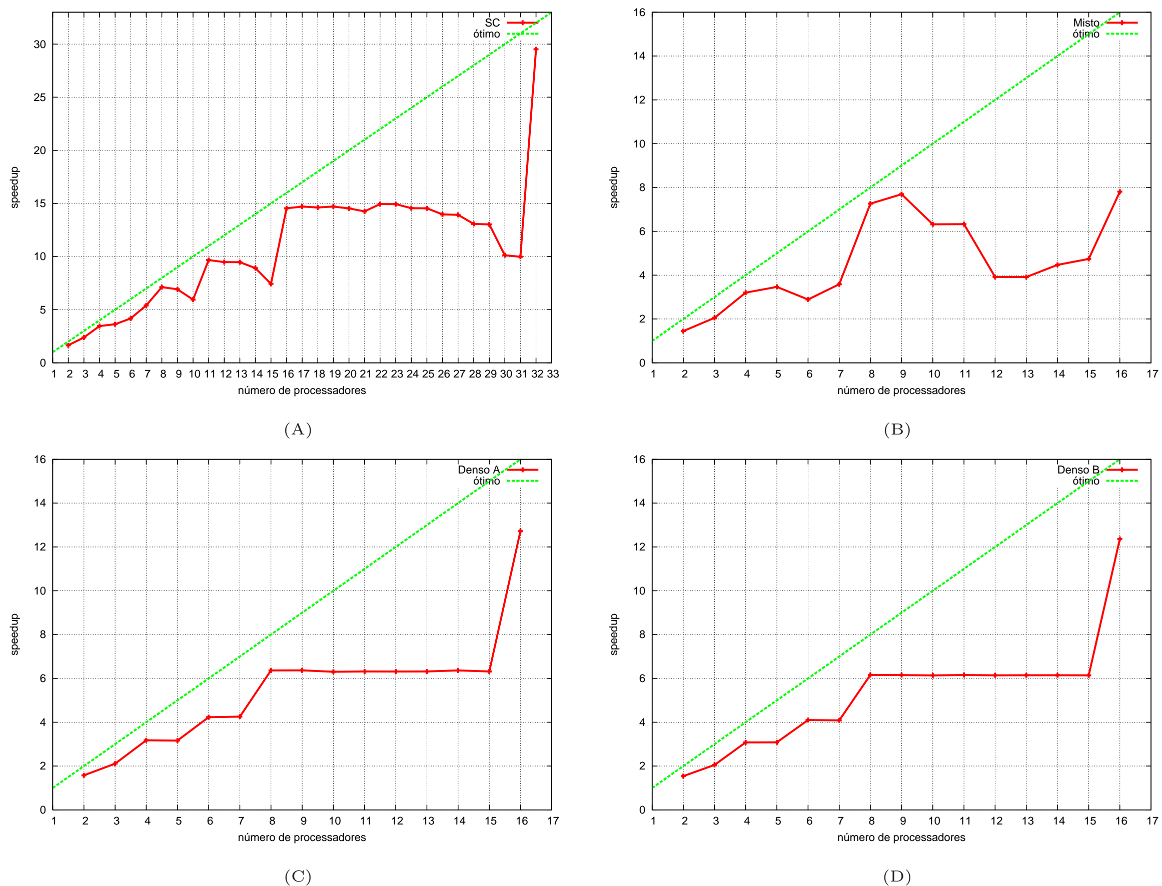


Figura 14: Aceleração do pré-processamento para o algoritmo *slice*.

Apesar da distribuição do cálculo de pré-processamento ter sido satisfatória em alguns casos, a necessidade de paralelizar esta etapa é menos significativa do que a MVD. Isso porque o pré-processamento é computado somente uma vez. Já a MVD é uma operação efetuada diversas vezes até que um modelo seja solucionado. Nesse ponto, o *slice* apresenta uma vantagem em sua segunda abordagem de escalonamento por que esta considera cada

AUNF como sendo uma tarefa. Tornando possível quebrar os termos em mais tarefas de custo menor (grão menor).

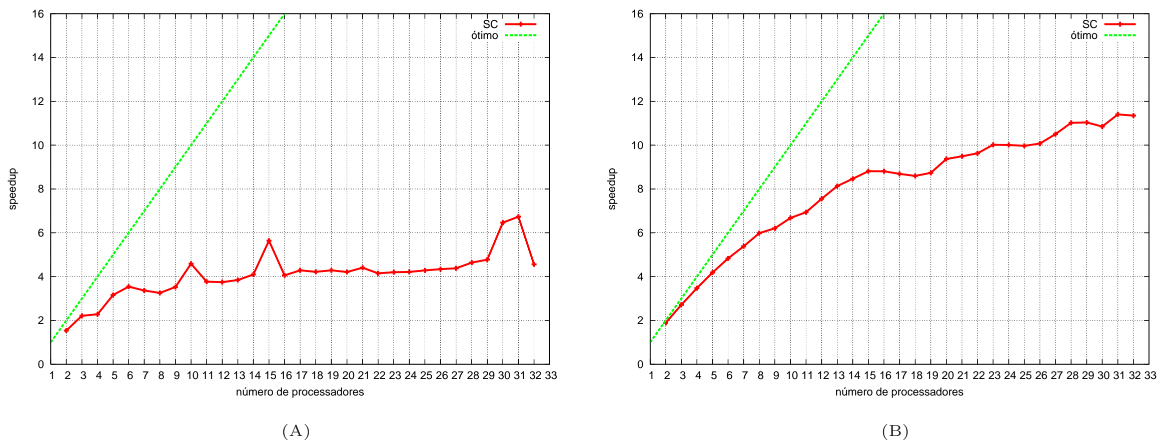


Figura 15: Aceleração do *Slice* considerando cada termo como uma tarefa (A) e considerando cada AUNF como uma tarefa (B) para o teste SC

Na figura 15 a aceleração obtida com a paralelização da MVD para ambas técnicas de escalonamento é apresentada utilizando o caso de teste SC. Faz-se importante ressaltar que, para fins comparativos, o cálculo de pré-processamento não foi considerado nos resultados apresentados. É perceptível, comparando-se os gráficos (figura 15 (A) e (B)), que a técnica de escalonamento considerando cada termo como uma tarefa apresenta maior escalabilidade. Isso comprova que a utilização de um grão menor é melhor. Mesmo considerando que a segunda abordagem de escalonamento impede a paralelização da etapa de pré-processamento.

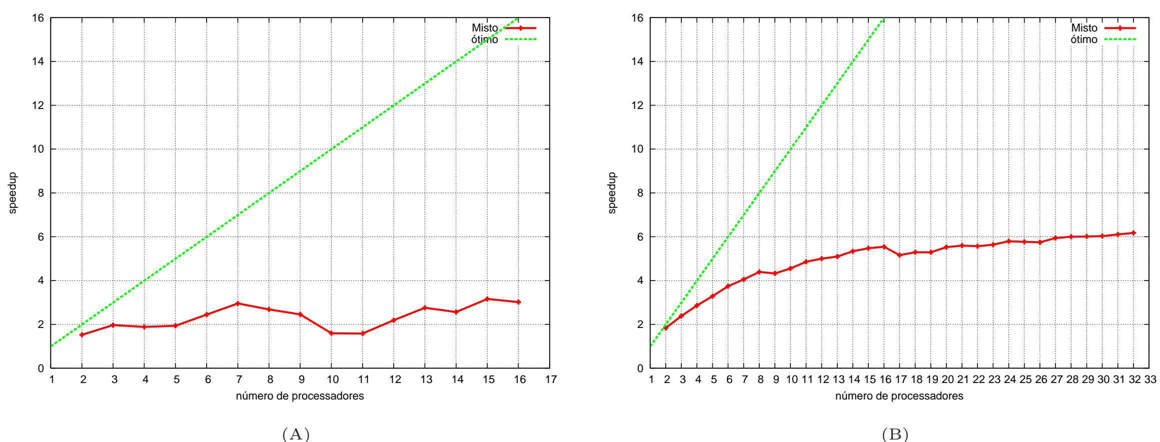


Figura 16: Aceleração do *Slice* considerando cada termo como uma tarefa (A) e considerando cada AUNF como uma tarefa (B) para o teste MISTO

Na figura 16 são apresentados os resultados para o caso de teste MISTO com as duas abordagens de escalonamento. De forma similar ao comportamento obtido com o caso

de teste SC, a utilização da segunda abordagem gera uma curva mais escalável. Isso confirma a hipótese de que a aplicação é mais adaptável a um grão menor. Um aspecto que chama atenção nos gráficos da figura 16 é a distância das curvas de *speedup* da curva ideal. Nesse teste tal comportamento é justificado porque o tempo de execução é bastante pequeno utilizando a versão seqüencial do algoritmo *slice*, veja tabela 2. A conseqüencia disso, é que como o tempo de execução não é muito significativo, o tempo necessário para transmissão dos dados não compensa tanto a utilização da versão paralela.

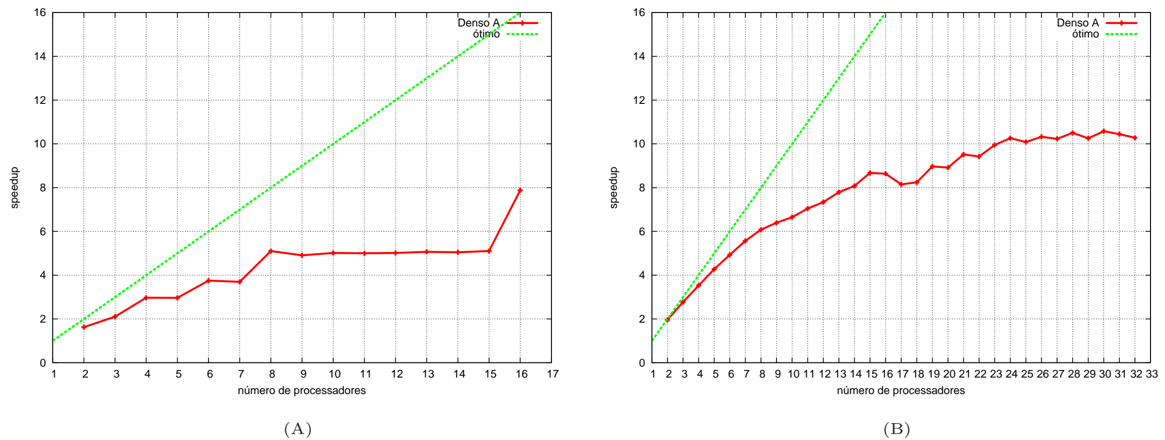


Figura 17: Aceleração do *Slice* considerando cada termo como uma tarefa (A) e considerando cada AUNF como uma tarefa (B) para o teste Denso A

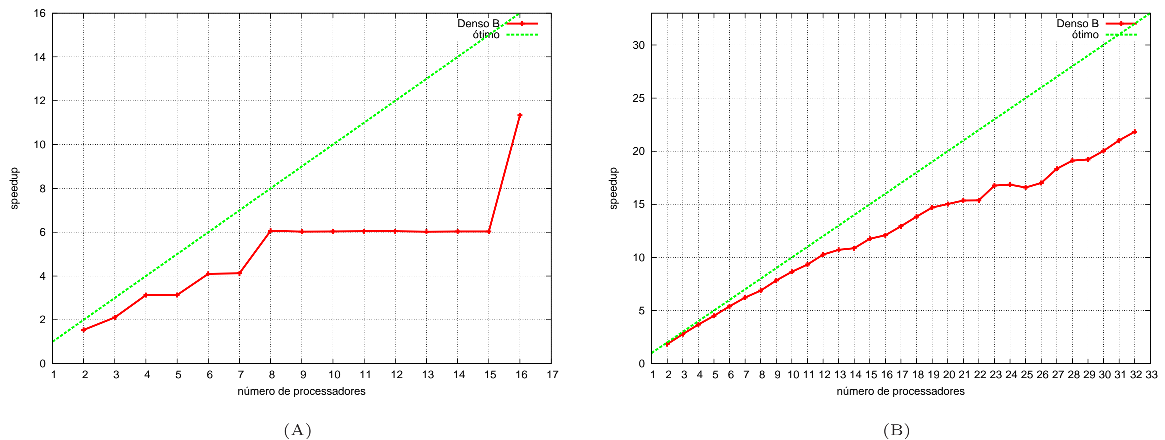


Figura 18: Aceleração do *Slice* considerando cada termo como uma tarefa (A) e considerando cada AUNF como uma tarefa (B) para o teste Denso B

Nas figura 17 e 18 são apresentados, respectivamente, os resultados para os casos de teste Denso A e Denso B. Com discutido anteriormente, esses dois modelos hipotéticos apresentam situações onde considerar ou não o custo de cada termo não influencia no resultado de escalonamento. Isso ocorre porque o custo computacional de cada tarefa nesses dois casos é o mesmo. Porém, ao utilizar a segunda abordagem de escalonamento com

o *slice* vislumbra-se uma curva que apresenta um comportamento escalável. Observando essas duas figura é possível notar que, a medida que o tempo seqüencial do caso de teste é maior, as curvas de aceleração apresentam um comportamento mais próximo do ideal.

Em geral, os resultados com a segunda abordagem de escalonamento do *slice* apresentam um comportamento escalável e próximo do ideal. Faz-se ressalva aos casos de teste que possuem um custo computacional baixo para a versão seqüencial. Porém, ainda obtém-se resultados expressivos em tais situações.

Capítulo 5

Conclusão

O presente trabalho apresentou como técnicas de alto desempenho podem ser empregadas para acelerar a Multiplicação Vetor-descritor. Essa operação é realizada diversas vezes para obter estimativas de desempenho em modelos que utilizam um formalismo analítico estruturado. Esses modelos são muito utilizados na predição de desempenho de sistemas em geral, pois apresentam uma forma mais eficiente de armazenamento do que Cadeias de Markov. As soluções propostas nesse trabalho utilizam uma estrutura algébrica comum aos diversos formalismos de modelagem analítica estruturados existentes e portanto podem ser empregadas em diferentes situações.

O foco principal da paralelização foi o processo de uma etapa da MVD, operação que é realizada diversas vezes para inferir estimativas de desempenho de um modelo. Porém, os detalhes de distribuição de tarefas foram delineados com base nas operações algébricas envolvidas em cada um dos dois algoritmos para realizar tal operação: *shuffle* e *slice*. Para cada um dos algoritmos foram apresentadas duas abordagens de escalonamento explorando as características específicas de cada um.

Em todos os testes realizados, obteve-se uma aceleração (*speedup*) considerável independentemente do algoritmo utilizado e da abordagem de escalonamento. Nesse mesmo contexto, a utilização de diferentes abordagens de escalonamento tornou possível estudar aspectos relevantes na adaptação de aplicações em ambientes de alto desempenho, mais especificamente, aglomerados de computadores (*clusters*). Apesar de uma completa avaliação das diferenças entre o *shuffle* e o *slice* fugir do escopo desse trabalho, discutiremos a seguir as principais diferenças entre as versões paralelas desses algoritmos utilizando os casos de teste propostos no capítulo 4.

5.1 Comparação entre Shuffle e Slice

Para comparar os resultados obtidos com *shuffle* e *slice* são mostrados dois gráficos de tempo de execução. Esses gráficos relacionam o número de processadores utilizados (eixo x) e o tempo de execução obtido (eixo y) para realizar uma iteração da MVD. No

caso do algoritmo *slice* os tempos de pré-processamento, apresentados na tabela 3, não são considerados, pois esses são referentes a uma parte do cálculo que é realizada somente uma vez e. Portanto, ao longo centenas ou milhares de iterações o tempo de pré-processamento não é impactante no tempo total de execução do mesmo.

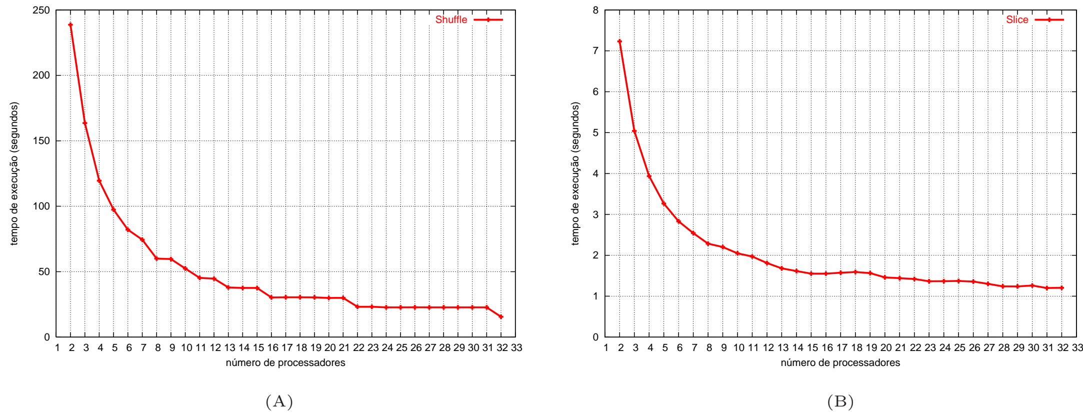


Figura 19: Comparação do tempo de execução entre *shuffle* (A) e o *slice* (B) para caso de teste SC.

Os melhores resultados obtidos com *shuffle* foram utilizando a abordagem de escalonamento que considera o custo das tarefas. Já com o *slice*, os melhores resultados apresentados foram utilizando a abordagem de escalonamento que considera cada fator como uma tarefa. Assim, compara-se os resultados obtidos com cada algoritmo utilizando essas duas abordagens.

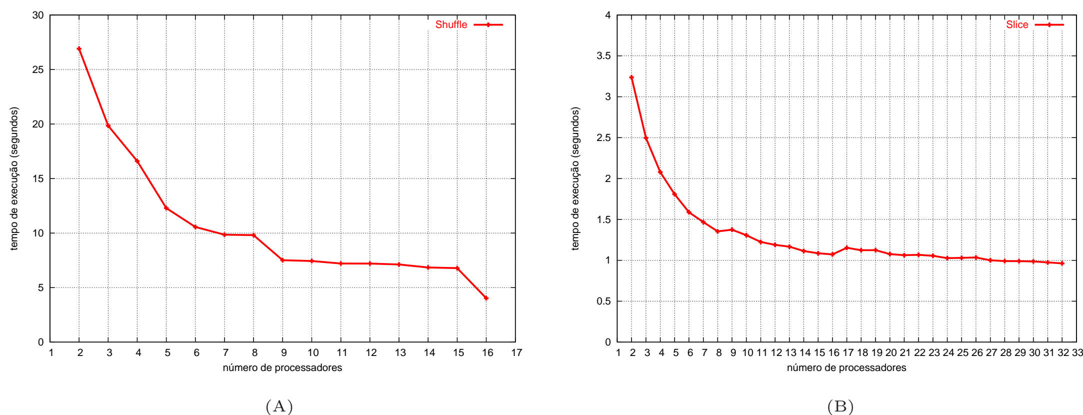


Figura 20: Comparação do tempo de execução entre *shuffle* (A) e o *slice* (B) para caso Misto.

Na figura 19, os resultados em tempo de execução para os dois algoritmos utilizando o caso de teste SC são apresentados. Note que a escala utilizada nos gráficos é diferente. Isso porque o tempo de execução do algoritmo *slice* (figura 19 (B)) é significativamente menor que o do *shuffle* (figura 19 (A)). Entretanto, ambos algoritmos apresentam um comportamento bastante escalável.

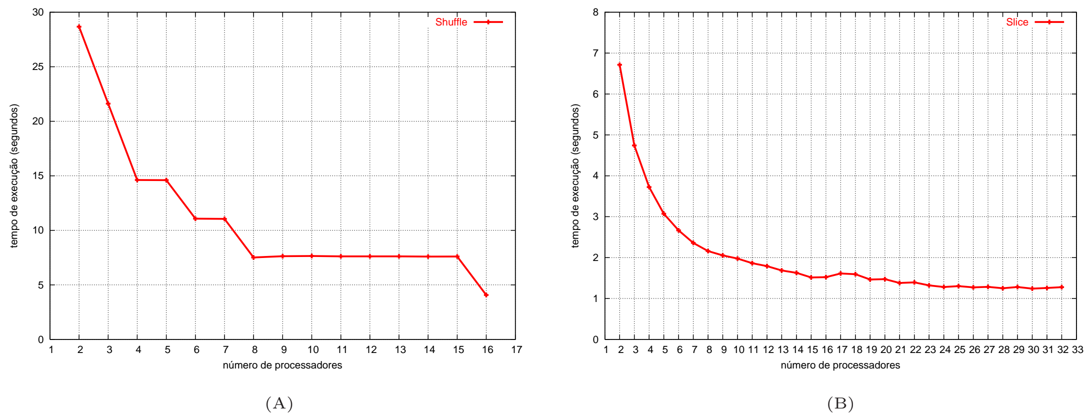


Figura 21: Comparação do tempo de execução entre *shuffle* (A) e o *slice* (B) para caso Denso A.

A figura 20 mostra os resultados em tempo de execução com o caso de teste Misto. Nesse caso, as mesmas afirmações feitas anteriormente se aplicam pois, o comportamento das curvas é bastante semelhante. Ainda, os tempos de execução obtidos com o *slice* (figura 20 (B)) são bem inferiores aos obtidos utilizando o *shuffle* (figura 20 (A)) e por isso as escalas dos gráficos são diferentes. Faz-se importante ressaltar que o número de processadores utilizados com o *slice*, nesse caso de teste, é superior à quantidade de termos. Como o número de termos é um fator limitante do número de processadores que podem ser utilizados, a escalabilidade do *shuffle* é sempre limitada por esse fator. Por outro lado, o *slice* pode escalar em um número bem maior de processadores.

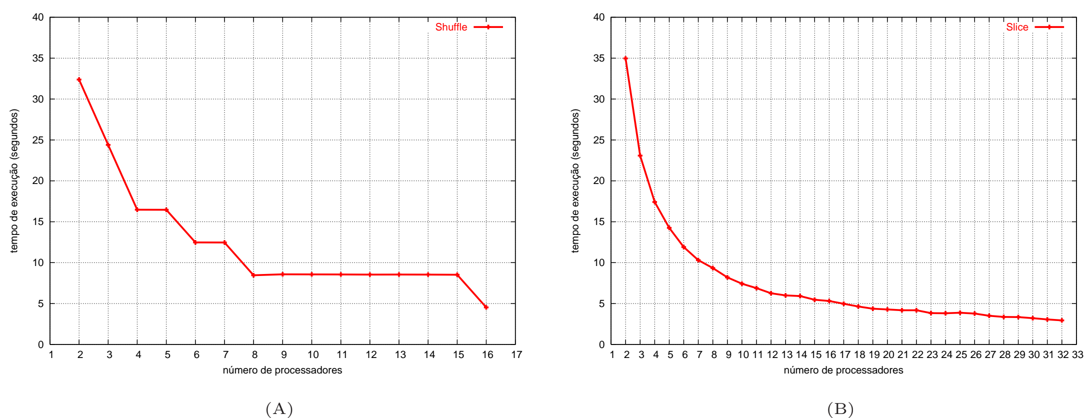


Figura 22: Comparação do tempo de execução entre *shuffle* (A) e o *slice* (B) para caso Denso B.

Os gráficos da figura 21, que mostram os resultados para o teste Denso A, apresentam as mesmas características já mencionadas. Entretanto, nos gráficos com o caso de teste Denso B (figura 22), em alguns pontos é possível observar que os resultados obtidos com o *shuffle* são melhores. Isso ocorre nas execuções com 4, 8 e 16 processadores. Não por coincidência, esses são os casos cujo número de processadores é múltiplo da quantidade de

termos. Em princípio, outro fator que contribui para esse comportamento é a semelhança do tempo seqüencial de execução do *shuffle* e do *slice* nesse caso de teste.

5.2 Trabalhos Futuros

Apesar do *slice* ter apresentado melhores resultados que o *shuffle* na maioria dos casos de teste, estudos recentes apontam que a solução ideal para realizar a MVD é uma abordagem híbrida [20], que misture aspectos de ambos algoritmos. Nesse sentido, a paralelização deste algoritmo híbrido poderá explorar aspectos das abordagens de escalonamento aqui apresentadas.

O estudo do impacto de paralelização com relação à quantidade de memória utilizada é também outra possível continuidade ao trabalho aqui apresentado. Principalmente em modelos que apresentam um elevado número de estados, tornando a memória um fator limitante.

Agrupar AUNFs para otimizar o volume de dados transmitidos a cada iteração é outro possível assunto a ser abordado em futuras investigações. Entretanto, como o número de fatores tipicamente é bastante elevado, considerar as características de cada fator um-a-um pode levar em altos custos. O estudo de como esse agrupamento pode ser realizado mantendo um compromisso entre tempo de escalonamento e a redução na comunicação torna-se necessário.

5.3 Considerações Finais

O objetivo desse trabalho foi apresentar como técnicas de alto desempenho podem ser empregadas para acelerar a análise de modelos analíticos estruturados. Nesse contexto, foi contemplado o uso de MPI, uma ferramenta popular para o desenvolvimento em aglomerados de computadores. A escolha desse tipo de plataforma, foi motivada principalmente pelo baixo custo envolvido em sua construção. Estabelecendo um compromisso entre custo e benefício que se adapta de maneira a acompanhar a evolução rápida que ocorre com os micro-processadores.

A principal impressão deixada após a análise dos resultados é que a versão paralela do algoritmo *slice* apresenta melhores resultados que o *shuffle*. No entanto, uma análise aprofundada mostra que isso não é sempre verdade. Os resultados obtidos com o último caso de teste (Denso B) para o *shuffle*, por exemplo, em alguns casos superam os resultados do *slice*. Também é importante levar em consideração que o *slice* necessita de uma etapa de pré-processamento. Logo, uma solução híbrida, que explore características de ambos os algoritmos aparentemente seja mais eficiente.

Em geral, o trabalho realizado até o momento apresentou resultados positivos. Essa afirmação é atestada pelos bons desempenhos obtidos nos testes realizados. Mesmo

quando os tempos seqüenciais dos algoritmos não apresentavam um custo considerável para a realização da MVD estes mostraram-se adaptáveis ao tipo de arquitetura utilizada. Reforçando as escolhas efetuadas ao longo do trabalho.

A principal contribuição do presente trabalho é apontar diretivas para o desenvolvimento em aglomerados de computadores. Mesmo em diferentes formalismos, os algoritmos propostos podem ser utilizados. Entretanto, o trabalho iniciado aqui não se apresenta completamente esgotado, restando assuntos a serem abordados em trabalhos futuros.

Referências Bibliográficas

[1] S. Abe, D. Place, and P. Mora. A Parallel Implementation of the Lattice Solid Model for the Simulation of Rock Mechanics and Earthquake Dynamics. *Pure and Applied Geophysics*, 161(11–12):2265–2277, 2004.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets, Wiley Series in Parallel Computing*. John Wiley and Sons, England, 1995.

[3] Y. Akiyama, K. Onizuka, T. Noguchi, and M. Ando. Parallel Protein Information Analysis (PAPIA) System Running on a 64-Node PC Cluster. In *Workshop on Genome Informatics*, volume 9, pages 131–140, 1998.

[4] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. *Lecture Notes in Computer Science*, 2104:1–10, 2001.

[5] K. Atif and B. Plateau. Stochastic Automata Networks for Modelling Parallel Systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

[6] L. Baldo, L. Brenner, L. G. Fernandes, and A. Sales. Performance Models for Master/Slave Parallel Programs. *Electronic Notes in Theoretical Computer Science*, 104(1):65–84, 2005.

[7] L. Baldo, D. M. Cláudio, L. G. Fernandes, P. Fernandes, M. Kolberg, P. Velho, and T. Webber. Parallel Selfverified Method for Solving Linear Systems. In *7th International Meeting of High Performance Computing for Computational Science - VECPAR*, pages 1–12, Rio de Janeiro, 2006.

[8] L. Baldo, L. G. Fernandes, P. Roisenberg, P. Velho, and T. Webber. Parallel PEPS Tool Performance Analysis Using Stochastic Automata Networks. In *Euro-Par*, pages 214–219, Pisa, Italy, 2004.

[9] C. Bertolini, L. Brenner, A. Sales, P. Fernandes, and A. Zorzo. Structured Stochastic Modeling of Fault-Tolerant Systems. In New York: IEEE Press, editor, *12th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis, and*

Simulation of Computer and Telecommunication Systems, MASCOTS'04, pages 139–146, Volendam, Netherlands, 2004.

[10] L. Brenner, L. G. Fernandes, P. Fernandes, and A. Sales. Performance Analysis Issues for Parallel Implementations of Propagation Algorithm. In Publisher ACM Press, editor, *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, pages 183–190, São Paulo, Brazil, 2003.

[11] P. Buchholz. Hierarchical Markovian Models - Symmetries and Reduction. In R. Pooley and J. Hillston, editors, *6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 305–319, Edinburgh, September 1992.

[12] P. Buchholz and T. Dayar. Block SOR for Kronecker structured representations. In *Proceedings of the 2003 International Conference on the Numerical Solution of Markov Chains*, pages 121–143, Urbana and Monticello, Illinois, USA, 2003. Springer-Verlag.

[13] P. Buchholz and P. Kemper. Hierarchical Reachability Graph Generation for Petri nets. *Formal Methods in Systems Design*, 21(3):281–315, 2002.

[14] G. Ciardo and M. Tilgner. On the Use of Kronecker Operators for the Solution of Generalized Stochastic Petri Nets. Technical Report 96(35), ICASE, 1996.

[15] M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Transactions on Computers*, C-30(2):116–125, 1981.

[16] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803 – 820, 2003.

[17] L. G. Fernandes, E. Bezerra, F. Oliveira, M. Raeder, P. Velho, and L. Amaral. Probe Effect Mitigation in the Software Testing of Parallel Systems. In *Latin-American Test Workshop LATW*, pages 153–158, Buenos Aires, 2006.

[18] P. Fernandes. *Méthodes Numériques pour la Solution de Systèmes Markoviens à Grand Espace d'États*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, 1998.

[19] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-Vector Multiplication in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381 – 414, 1998.

[20] P. Fernandes, R. Presotto, A. Sales, and T. Webber. An Alternative Algorithm to Multiply a Vector by a Kronecker Represented Descriptor. In *21th Annual UK Performance Engineering Workshop, UKPEW*, pages 57–67, Newcastle, England, 2005.

[21] E. Gelenbe. G-Networks: Multiple Classes of Positive Customers, Signals, and Product Form Results. *Lecture Notes in Computer Science*, 2459:1–16, 2002.

[22] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA Nets: A Structured Performance Modelling Formalism. *Performance Evaluation*, 17(10):79–104, 2003.

[23] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing - Second Edition*. Pearson - Addison Wesley, England, 1996.

[24] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI Portable Parallel Programming with the Message-Passing Interface - Second Edition*. The MIT Press, London, England, 1999.

[25] M. Kolberg, L. Baldo, P. Velho, L. G. Fernandes, and D. M. Cláudio. Optimizing a Parallel Self-verified Method for Solving Linear Systems. In *Workshop on State-of-the-Art in Scientific and Parallel Computing - PARA*, Umea, 2006.

[26] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 Supercomputer Sites. http://www.top500.org, July 2006.

[27] J. Montagnat, F. Bellet, H. Benoit Cattin, V. Breton, L. Brunie, H. Duque, Y. Legré, I. E. Magnin, L. Maigne, S. Miguet, J.-M. Pierson, L. Seitz, and T. Tweed. Medical Images Simulation, Storage, and Processing on the European DataGrid Testbed. *Journal of Grid Computing*, 2(4):387–400, 2004.

[28] B.M.E. Moret and H.D. Shapiro. *Algorithms from P to NP*. The Benjamin/Cummings Publishing Company, INC., Redwood City, CA, USA, 1991.

[29] B. Philippe, Y. Saad, and W.J. Stewart. Numerical Methods in Markov Chain Modelling. *Operations Research*, 40(6):1156–1179, 1992.

[30] G. F. Riley. The Georgia Tech Network Simulator. In Publisher ACM Press, editor, *ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, pages 5–12, Karlsruhe, Germany, 2003.

[31] W. H. Sanders and J. F. Meyer. Stochastic Activity Networks: Formal Definitions and Concepts. *Lectures on Formal Methods and Performance Analysis: First EEF/Euro Summer School on Trends in Computer Science*, 2090:315–343, 2001.

[32] M. Scarpa and A. Bobbio. Kronecker Representation of Stochastic Petri Nets with Discrete PH Distributions. In *3rd Int. Performance & Dependability Symposium (IPDS '98)*, pages 52–61, Durham (NC), 1998.

[33] P. Velho, L. G. Fernandes, M. Raeder, M. Castro, and L. Baldo. A Parallel Version for the Propagation Algorithm. *Lecture Notes in Computer Science*, 3606:403–412, 2005.

[34] T. Webber. Alternativas para o Tratamento Numérico Otimizado da Multiplicação Vetor-Descritor. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação - PPGCC, PUCRS, Faculdade de Informática - FACIN, Porto Alegre, Brasil, 2003.

[35] A.Y. Zomaya. *Parallel and Distributed Computing Handbook*. McGraw-Hill, New York, NY, USA, 1996.

Apêndice A

SAN - Exemplo de Modelo Analítico Estrutural

Modelos SAN são descritos geralmente de forma gráfica onde um grafo dirigido é projetado para cada módulo do sistema. A representação gráfica fornece uma maneira de esquematizar o sistema em módulos independentes. Muitas vezes, esta separação ajuda a projetar o sistema, uma vez que módulos com funções diferentes podem ser modelados separadamente e conseqüentemente, facilmente replicados. Um exemplo de modelo SAN que ilustra isto pode ser visto na figura 23. O problema modelado neste exemplo é conhecido como *problema da seção crítica* (SC). O problema da SC consiste no acesso exclusivo de um processo a um determinado recurso. Neste exemplo, temos apenas um processo acessando apenas um recurso. Os estados do modelo são representados pelos vértices do grafo, neste caso temos dois autômatos: um para representar o **processo** (com três estados) e outro chamado **recurso** (com dois estados). Repare como o processo pode ser modelado independentemente do recurso facilitando a expansão do modelo. Para modelar o problema com um processo a mais, por exemplo, basta acrescentar um autômato recurso (como mostra a figura 24).

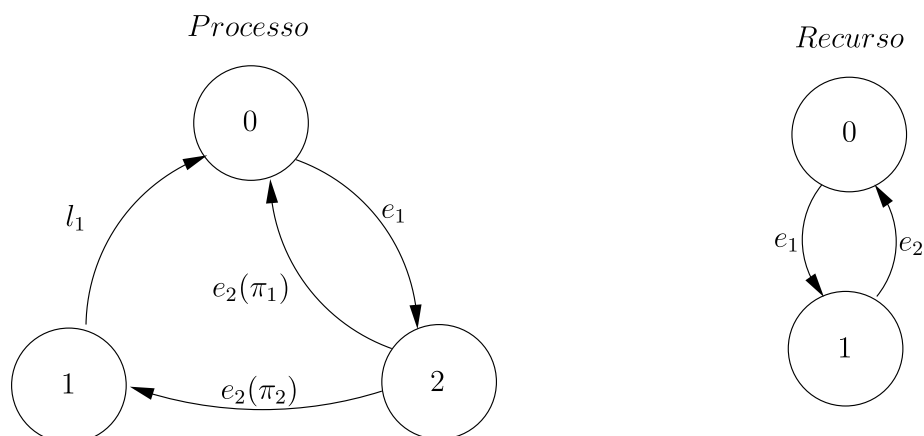


Figura 23: Modelo SAN para o problema da SC

No entanto, as afirmações anteriores não são suficientes para modelar como essas interagem duas entidades (**processo** e **recurso**). Para modelar este comportamento, é utilizado o conceito de eventos. Eventos são graficamente representados por arestas. Em SAN, dispomos de dois tipos básicos de eventos: *locais* (pois alteram apenas o estado de um autômato) e *sincronizantes* (pois representam uma interação entre dois ou mais autômatos). Em nosso exemplo, podemos detectar um evento sincronizante quando o identificador aparece nos dois autômatos simultaneamente, por exemplo, o evento e_2 é um evento sincronizante. Foge do escopo deste trabalho fornecer uma explicação mais detalhada sobre modelagem utilizando SAN. Os leitores interessados em buscar mais informações podem consultar as referências [10, 18, 34].

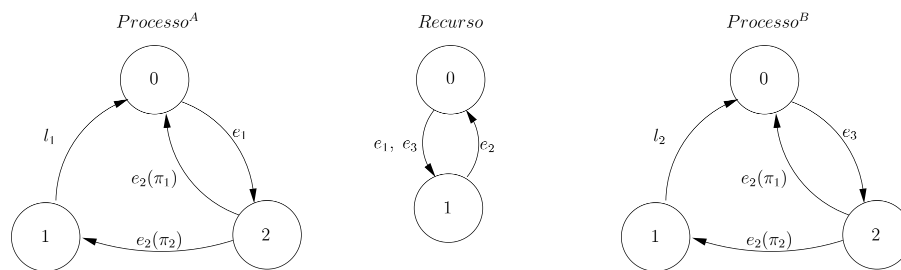


Figura 24: Modelo SAN para o problema da SC com dois processos

O Descritor Markoviano (DM) é uma estrutura algébrica que armazena informações sobre: as transições locais de cada autômato e os eventos sincronizantes. Para representar as transições locais, são armazenadas uma matriz para cada autômato. Nestas matrizes, cada transição não sincronizante de um estado i para outro estado j é representada por um número real positivo. Este número indica a freqüência com que esta transição ocorre no mundo real. Por exemplo, considerando o modelo SAN para o problema da SC visto na figura 23. Supondo-se que a transição local do autômato processo (l_1), do estado 1 para o estado 0, aconteça com uma freqüência μ, a matriz que representa algebricamente o comportamento local deste autômato será:

$$Q_l^{(P)} = \begin{bmatrix} 0 & 0 & 0 \\ \mu & -\mu & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Para representar algebricamente as interações entre os autômatos são necessárias duas matrizes para cada autômato do modelo. De maneira análoga ao processo anterior, os elementos destas matrizes representam a freqüência com que estas transições ocorrem na realidade. Por exemplo, supondo que o evento e_1 ocorra na realidade com uma freqüência κ, para cada um dos dois autômatos do exemplo da figura 23[1] serão criadas duas matrizes como abaixo:

[1]Os autômatos *recurso* e *processo* foram abreviados para R e P respectivamente.

$$Q_{e_1^+}^{(R)} = \begin{bmatrix} 0 & \kappa \\ 0 & 0 \end{bmatrix} \qquad Q_{e_1^-}^{(R)} = \begin{bmatrix} -\kappa & 0 \\ 0 & 0 \end{bmatrix}$$

$$Q_{e_1^+}^{(P)} = \begin{bmatrix} 0 & \kappa & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Q_{e_1^-}^{(P)} = \begin{bmatrix} -\kappa & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Note que cada autômato tem uma matriz positiva e outra negativa para o evento e_1. A matriz positiva tem a mesma função da matriz local, ela indica qual transição está associada a freqüência com que o evento ocorre na realidade. A parte negativa faz um ajuste diagonal, neste ajuste cada elemento da diagonal principal contém um valor que, se somados todos os elementos de uma linha da matriz positiva com a negativa, resultará zero.

O DM é então uma expressão algébrica que mapeia as diversas matrizes que compõe os modelos SAN em uma única matriz. A equação 14, generaliza como fica o DM para um modelo SAN com N autômatos e E eventos sincronizantes.

$$Q = \bigoplus_{i=1}^{N} Q_l^{(i)} + \sum_{e=1}^{E} \left(\bigotimes_{i=1}^{N} Q_{e^+}^{(i)} + \bigotimes_{i=1}^{N} Q_{e^-}^{(i)} \right) \tag{14}$$

Para o exemplo apresentado na figura 23, o DM ficará:

$$\left(Q_l^{(P)} \oplus Q_l^{(R)}\right) + \left(\left(Q_{e_1^+}^{(P)} \otimes Q_{e_1^+}^{(R)} + Q_{e_2^-}^{(P)} \otimes Q_{e_2^-}^{(R)}\right) + \left(Q_{e_1^-}^{(P)} \otimes Q_{e_1^-}^{(R)} + Q_{e_2^+}^{(P)} \otimes Q_{e_2^+}^{(R)}\right)\right)$$

Apesar de ser possível resolver a expressão algébrica do DM, gerando uma única matriz que represente o modelo, isto não é desejado. Uma das vantagens de se utilizar modelos SAN em relação a Cadeias de Markov é a utilização reduzida de memória, gerando-se uma única matriz essa vantagem desaparece. Neste ponto, entra em ação a Multiplicação Vetor-descritor, ao invés de uma multiplicação de um vetor por uma matriz (Ax) o sistema de equações a ser resolvido se apresenta na forma da multiplicação do Descritor Markoviano pelo vetor (Qx).

Apêndice B

Semântica dos Modelos de Teste Utilizados

O modelo SAN da figura 25 apresenta a descrição gráfica do modelo de teste denominado SC. Baseado em um caso real, este modelo representa uma situação onde 16 recursos são disputados por 4 processos. Cada processo pode estar utilizando um recurso (estado U) ou livre (estado L). De forma semelhante um recurso pode estar sendo utilizado por um processo (estado U) ou estar livre (estado L).

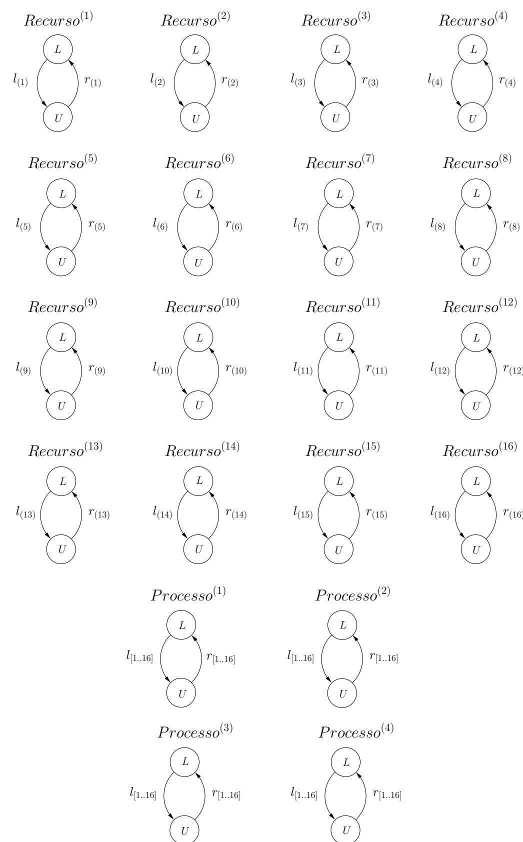


Figura 25: Modelo SAN do caso de teste SC.

Na figura 26, é apresentado o modelo SAN referente ao caso de teste Misto. Este caso de teste explora diversos aspectos diferentes de transições entre os estados, bem como diversifica o número de estados de cada autômato.

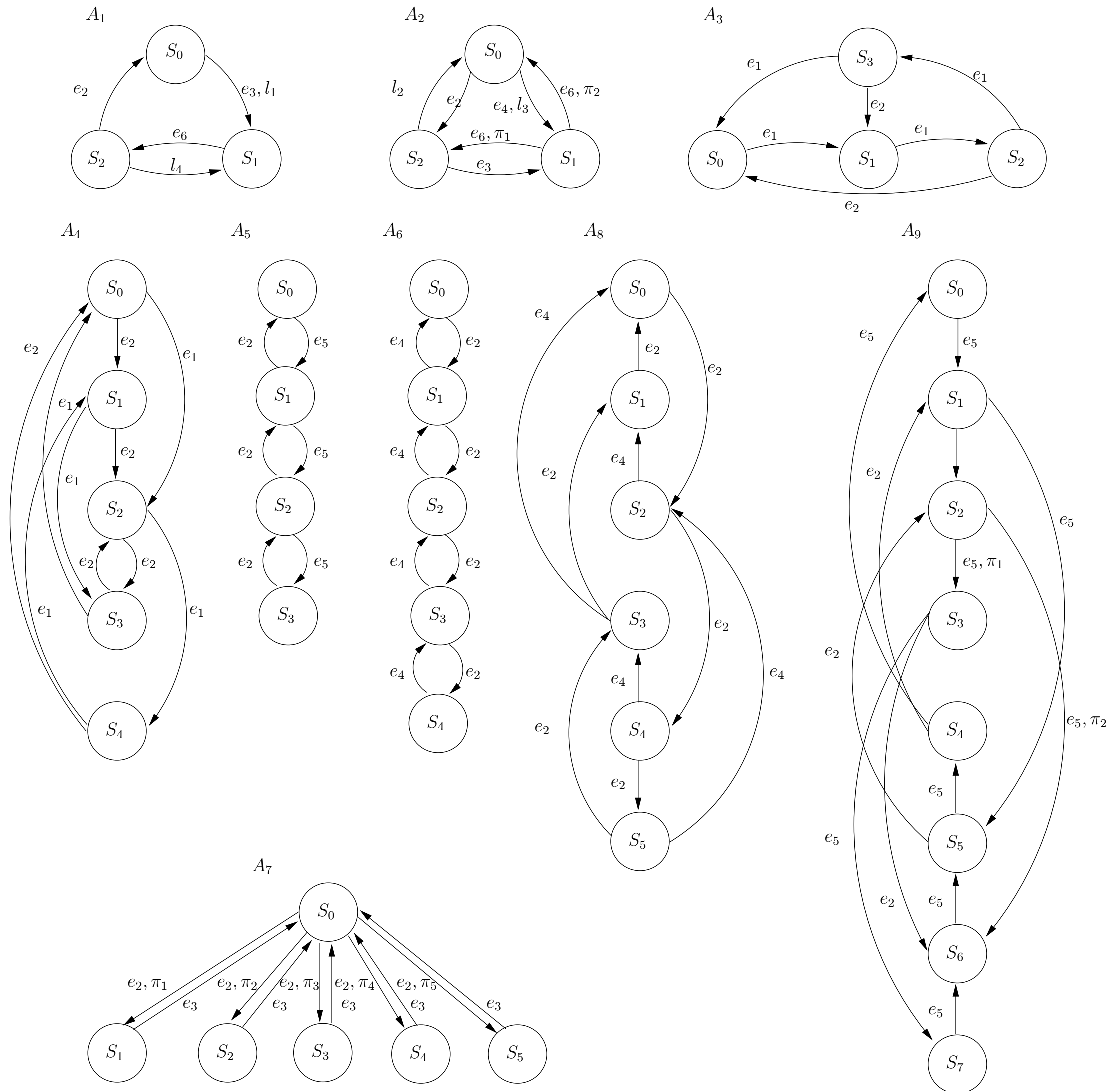


Figura 26: Modelo SAN do caso de teste Misto.

Apêndice C

Detalhamento dos Resultados

Tabela 4: Shuffle - escalonamento sem considerar custos - teste MISTO.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	52,432280	0,006782	-	- %
2	26,647920	0,016673	1,967593	98,00 %
3	19,970660	0,015362	2,625465	87,00 %
4	13,965560	0,028442	3,754398	93,00 %
5	13,638080	0,028548	3,844549	76,00 %
6	10,776360	0,056347	4,865490	81,00 %
7	10,507700	0,026210	4,989891	71,00 %
8	7,249580	0,058000	7,232457	90,00 %
9	7,476920	0,041557	7,012550	77,00 %
10	7,651952	0,060324	6,852144	68,00 %
11	7,632818	0,030740	6,869321	62,00 %
12	7,632166	0,046411	6,869908	57,00 %
13	7,592220	0,099854	6,906053	53,00 %
14	7,566224	0,062457	6,929781	49,00 %
15	7,395264	0,027910	7,089980	47,00 %
16	4,099242	0,049699	12,790725	79,00 %

Tabela 5: Shuffle - escalonamento sem considerar custos - teste SC.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	477,777000	0,288315	-	- %
2	242,942400	0,226192	1,966626	98,00 %
3	164,506200	0,201737	2,904309	96,00 %
4	121,996200	0,065505	3,916326	97,00 %
5	99,167940	0,035199	4,817857	96,00 %
6	84,036420	0,049507	5,685356	94,00 %
7	74,916560	0,096041	6,377455	91,00 %
8	61,135760	0,076391	7,815016	97,00 %
9	60,181460	0,048000	7,938939	88,00 %
10	53,943100	0,096659	8,857054	88,00 %
11	46,088540	0,025416	10,366503	94,00 %
12	46,077380	0,052839	10,369014	86,00 %
13	38,709540	0,072580	12,342616	94,00 %
14	38,720240	0,042083	12,339205	88,00 %
15	38,672260	0,029478	12,354514	82,00 %
16	30,894900	0,053131	15,464591	96,00 %
17	31,011020	0,054506	15,406684	90,00 %
18	31,019880	0,049608	15,402283	85,00 %
19	31,002920	0,096524	15,410709	81,00 %
20	30,984240	0,059598	15,420000	77,00 %
21	30,497700	0,094254	15,666001	74,00 %
22	23,662160	0,057480	20,191605	91,00 %
23	23,664500	0,016386	20,189608	87,00 %
24	23,634980	0,029563	20,214825	84,00 %
25	23,658360	0,021236	20,194848	80,00 %
26	23,641040	0,029376	20,209643	77,00 %
27	23,634900	0,074060	20,214894	74,00 %
28	23,637060	0,067542	20,213046	72,00 %
29	23,621680	0,042166	20,226207	69,00 %
30	23,644940	0,014647	20,206310	67,00 %
31	23,597060	0,037322	20,247310	65,00 %
32	15,831000	0,054332	30,179837	94,00 %

Tabela 6: Shuffle - escalonamento sem considerar custos - teste Denso A.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	58,276200	0,094138	-	- %
2	29,225900	0,049152	1,993991	99,00 %
3	22,057380	0,037456	2,642027	88,00 %
4	14,892500	0,026962	3,913124	97,00 %
5	14,907260	0,047812	3,909249	78,00 %
6	11,285020	0,038910	5,164031	86,00 %
7	11,294160	0,044944	5,159852	73,00 %
8	7,684970	0,020780	7,583139	94,00 %
9	7,768198	0,012569	7,501894	83,00 %
10	7,776966	0,038013	7,493436	74,00 %
11	7,779560	0,078179	7,490937	68,00 %
12	7,748988	0,069368	7,520491	62,00 %
13	7,744112	0,029732	7,525226	57,00 %
14	7,734744	0,041303	7,534341	53,00 %
15	7,737362	0,048000	7,531791	50,00 %
16	4,154754	0,053282	14,026390	87,00 %

Tabela 7: Shuffle - escalonamento sem considerar custos - teste DENSO B.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	64,450880	0,067394	-	- %
2	32,379200	0,054147	1,990502	99,00 %
3	24,397160	0,021071	2,641736	88,00 %
4	16,470320	0,031352	3,913152	97,00 %
5	16,464200	0,042449	3,914607	78,00 %
6	12,469040	0,088277	5,168872	86,00 %
7	12,458760	0,013490	5,173137	73,00 %
8	8,454172	0,029949	7,623559	95,00 %
9	8,569142	0,063261	7,521275	83,00 %
10	8,559234	0,061497	7,529982	75,00 %
11	8,556078	0,025748	7,532759	68,00 %
12	8,536032	0,049919	7,550449	62,00 %
13	8,540486	0,068614	7,546511	58,00 %
14	8,534100	0,037456	7,552158	53,00 %
15	8,518720	0,042883	7,565793	50,00 %
16	4,524182	0,031890	14,245863	89,00 %

Tabela 8: Shuffle - escalonamento considerando custos - teste SC.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	477,777000	0,028315	-	- %
2	238,738800	0,025204	2,001254	100,00 %
3	163,605600	0,070434	2,920297	97,00 %
4	119,542400	0,087487	3,996715	99,00 %
5	97,413100	0,018366	4,904648	98,00 %
6	82,103680	0,086792	5,819191	96,00 %
7	74,464140	0,010955	6,416202	91,00 %
8	60,042180	0,045243	7,957355	99,00 %
9	59,662300	0,015805	8,008021	88,00 %
10	52,434420	0,038072	9,111896	91,00 %
11	45,260960	0,095163	10,556050	95,00 %
12	44,690220	0,065169	10,690862	89,00 %
13	37,987740	0,093861	12,577136	96,00 %
14	37,564220	0,045464	12,718938	90,00 %
15	37,543680	0,087738	12,725896	84,00 %
16	30,338340	0,043600	15,748290	98,00 %
17	30,463280	0,063937	15,683701	92,00 %
18	30,411100	0,038039	15,710612	87,00 %
19	30,406080	0,102117	15,713206	82,00 %
20	29,952320	0,035341	15,951251	79,00 %
21	29,970260	0,018401	15,941703	75,00 %
22	23,208200	0,025670	20,586559	93,00 %
23	23,175560	0,068876	20,615553	89,00 %
24	22,751800	0,045044	20,999525	87,00 %
25	22,750480	0,050685	21,000743	84,00 %
26	22,792820	0,088107	20,961732	80,00 %
27	22,759080	0,042743	20,992808	77,00 %
28	22,748360	0,049030	21,002700	75,00 %
29	22,755220	0,095587	20,996369	72,00 %
30	22,743360	0,032186	21,007318	70,00 %
31	22,716620	0,060695	21,032046	67,00 %
32	15,543940	0,058369	30,737187	96,00 %

Tabela 9: Shuffle - escalonamento considerando custos - teste Misto.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	52,432280	0,060782	-	- %
2	26,904400	0,017029	1,948836	97,00 %
3	19,841840	0,024413	2,642510	88,00 %
4	16,606380	0,029308	3,157357	78,00 %
5	12,288420	0,004242	4,266804	85,00 %
6	10,556380	0,033436	4,966880	82,00 %
7	9,854974	0,003605	5,320387	76,00 %
8	9,812808	0,013114	5,343249	66,00 %
9	7,512562	0,046335	6,979280	77,00 %
10	7,438968	0,053944	7,048327	70,00 %
11	7,209346	0,041279	7,272820	66,00 %
12	7,205354	0,030594	7,276849	60,00 %
13	7,120532	0,030265	7,363534	56,00 %
14	6,846824	0,035057	7,657898	54,00 %
15	6,784160	0,014317	7,728632	51,00 %
16	4,028760	0,024083	13,014495	81,00 %

Tabela 10: Shuffle - escalonamento considerando custos - teste Denso A.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	58,276200	0,094138	-	- %
2	28,672600	0,099317	2,032470	100,00 %
3	21,616240	0,013038	2,695945	89,00 %
4	14,614560	0,050338	3,987543	99,00 %
5	14,604640	0,042602	3,990252	79,00 %
6	11,071720	0,060415	5,263518	87,00 %
7	11,049360	0,021540	5,274169	75,00 %
8	7,511606	0,005099	7,758154	96,00 %
9	7,625970	0,033689	7,641808	84,00 %
10	7,649860	0,087766	7,617943	76,00 %
11	7,616278	0,024166	7,651532	69,00 %
12	7,611996	0,046508	7,655836	63,00 %
13	7,615510	0,055416	7,652304	58,00 %
14	7,592870	0,041315	7,675121	54,00 %
15	7,600610	0,038716	7,667305	51,00 %
16	4,072288	0,040595	14,310431	89,00 %

Tabela 11: Shuffle - escalonamento considerando custos - teste DENSO B.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	64,450880	0,067394	-	- %
2	32,379200	0,054147	1,990502	99,00 %
3	24,397160	0,021071	2,641736	88,00 %
4	16,470320	0,031352	3,913152	97,00 %
5	16,464200	0,042449	3,914607	78,00 %
6	12,469040	0,088277	5,168872	86,00 %
7	12,458760	0,013490	5,173137	73,00 %
8	8,454172	0,029949	7,623559	95,00 %
9	8,569142	0,063261	7,521275	83,00 %
10	8,559234	0,061497	7,529982	75,00 %
11	8,556078	0,025748	7,532759	68,00 %
12	8,536032	0,049919	7,550449	62,00 %
13	8,540486	0,068614	7,546511	58,00 %
14	8,534100	0,037456	7,552158	53,00 %
15	8,518720	0,042883	7,565793	50,00 %
16	4,524182	0,031890	14,245863	89,00 %

Tabela 12: Slice - escalonamento por termo - teste SC.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	13,688620	0,052211	-	- %
2	8,894016	0,067216	1,539082	76,00 %
3	6,176964	0,076980	2,216075	73,00 %
4	5,992858	0,088187	2,284155	57,00 %
5	4,327158	0,036124	3,163420	63,00 %
6	3,862116	0,085644	3,544331	59,00 %
7	4,064464	0,079799	3,367878	48,00 %
8	4,199792	0,086879	3,259356	40,00 %
9	3,878644	0,069627	3,529228	39,00 %
10	2,980424	0,070171	4,592843	45,00 %
11	3,629782	0,042544	3,771196	34,00 %
12	3,649434	0,011532	3,750888	31,00 %
13	3,558264	0,035298	3,846993	29,00 %
14	3,340428	0,047528	4,097864	29,00 %
15	2,424022	0,052915	5,647069	37,00 %
16	3,371016	0,042684	4,060680	25,00 %
17	3,190004	0,031984	4,291098	25,00 %
18	3,243402	0,063458	4,220451	23,00 %
19	3,190162	0,011180	4,290885	22,00 %
20	3,248610	0,018248	4,213685	21,00 %
21	3,105942	0,030413	4,407236	20,00 %
22	3,296434	0,012328	4,152553	18,00 %
23	3,255902	0,073545	4,204248	18,00 %
24	3,243776	0,045738	4,219964	17,00 %
25	3,192346	0,062825	4,287949	17,00 %
26	3,150564	0,022045	4,344815	16,00 %
27	3,122566	0,068738	4,383772	16,00 %
28	2,949378	0,024103	4,641188	16,00 %
29	2,864984	0,037643	4,777904	16,00 %
30	2,115834	0,016852	6,469609	21,00 %
31	2,030804	0,038249	6,740492	21,00 %
32	2,998770	0,054534	4,564744	14,00 %

Tabela 13: Slice - escalonamento por termo - teste Misto.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	5,953364	0,001414	-	- %
2	3,897752	0,015132	1,527383	76,00 %
3	3,023084	0,026608	1,969301	65,00 %
4	3,154838	0,025357	1,887058	47,00 %
5	3,070490	0,045967	1,938897	38,00 %
6	2,429842	0,046249	2,450103	40,00 %
7	2,010514	0,028106	2,961115	42,00 %
8	2,215460	0,079195	2,687190	33,00 %
9	2,420620	0,049325	2,459437	27,00 %
10	3,731030	0,024617	1,595635	15,00 %
11	3,757254	0,038678	1,584498	14,00 %
12	2,716148	0,040124	2,191840	18,00 %
13	2,154746	0,086873	2,762907	21,00 %
14	2,319506	0,071819	2,566651	18,00 %
15	1,881302	0,011789	3,164491	21,00 %
16	1,971398	0,080709	3,019869	18,00 %

Tabela 14: Slice - escalonamento por termo - teste DENSO A.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	13,014060	0,019974	-	- %
2	8,086612	0,064676	1,609334	80,00 %
3	6,241818	0,048754	2,084979	69,00 %
4	4,418798	0,122584	2,945158	73,00 %
5	4,430600	0,052943	2,937313	58,00 %
6	3,503386	0,038613	3,714709	61,00 %
7	3,552140	0,085796	3,663723	52,00 %
8	2,573908	0,094836	5,056148	63,00 %
9	2,676764	0,036674	4,861863	54,00 %
10	2,619364	0,023769	4,968404	49,00 %
11	2,625688	0,036537	4,956438	45,00 %
12	2,619202	0,046141	4,968711	41,00 %
13	2,592868	0,066279	5,019175	38,00 %
14	2,603626	0,049040	4,998436	35,00 %
15	2,573324	0,057697	5,057295	33,00 %
16	1,668218	0,052076	7,801174	48,00 %

Tabela 15: Slice - escalonamento por termo - teste DENSO B.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	80,211800	0,0914400	-	- %
2	41,638520	0,0924639	1,926384	96,00 %
3	30,411480	0,0752480	2,637550	87,00 %
4	20,503440	0,0371160	3,912114	97,00 %
5	20,491720	0,0393700	3,914351	78,00 %
6	15,658280	0,0865740	5,122644	85,00 %
7	15,567400	0,0112200	5,152549	73,00 %
8	10,588460	0,0384160	7,575398	94,00 %
9	10,652080	0,0958270	7,530153	83,00 %
10	10,643200	0,0499090	7,536436	75,00 %
11	10,621360	0,0375890	7,551933	68,00 %
12	10,622940	0,0477280	7,550809	62,00 %
13	10,662780	0,0383600	7,522597	57,00 %
14	10,639660	0,0603980	7,538943	53,00 %
15	10,641680	0,0329620	7,537512	50,00 %
16	5,664326	0,0964880	14,160872	88,00 %

Tabela 16: Slice - escalonamento por AUNF - teste SC.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	13,688620	0,052211	-	- %
2	7,230356	0,006164	1,893215	94,00 %
3	5,040522	0,029732	2,715714	90,00 %
4	3,937008	0,040743	3,476909	86,00 %
5	3,267300	0,055848	4,189581	83,00 %
6	2,831166	0,046335	4,834976	80,00 %
7	2,541796	0,048383	5,385412	76,00 %
8	2,258990	0,066324	6,059619	75,00 %
9	2,203872	0,052810	6,211168	69,00 %
10	2,048420	0,053366	6,682526	66,00 %
11	1,971498	0,089788	6,943258	63,00 %
12	1,811098	0,013416	7,558188	62,00 %
13	1,682134	0,042284	8,137651	62,00 %
14	1,615564	0,027802	8,472966	60,00 %
15	1,552416	0,064342	8,817623	58,00 %
16	1,553196	0,070781	8,813195	55,00 %
17	1,574482	0,053469	8,694046	51,00 %
18	1,591332	0,016639	8,601988	47,00 %
19	1,565582	0,012020	8,743470	46,00 %
20	1,460170	0,027712	9,374675	46,00 %
21	1,441544	0,054662	9,495804	45,00 %
22	1,420794	0,010000	9,634486	43,00 %
23	1,365770	0,017088	10,022639	43,00 %
24	1,366814	0,016309	10,014983	41,00 %
25	1,372324	0,014847	9,974772	39,00 %
26	1,358112	0,018867	10,079154	38,00 %
27	1,302340	0,010992	10,510788	38,00 %
28	1,241488	0,011489	11,025978	39,00 %
29	1,239436	0,078185	11,044233	38,00 %
30	1,260680	0,066385	10,858124	36,00 %
31	1,199684	0,078032	11,410188	36,00 %
32	1,205244	0,086151	11,357550	35,00 %

Tabela 17: Slice - escalonamento por AUNF - teste Misto.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	5,953364	0,001414	-	- %
2	3,237224	0,026362	1,839033	91,00 %
3	2,497186	0,046946	2,384029	79,00 %
4	2,078450	0,039268	2,864328	71,00 %
5	1,809018	0,054516	3,290936	65,00 %
6	1,588262	0,059455	3,748351	62,00 %
7	1,467162	0,033211	4,057741	57,00 %
8	1,354258	0,026172	4,396033	54,00 %
9	1,374858	0,051971	4,330166	48,00 %
10	1,307164	0,051932	4,554412	45,00 %
11	1,224246	0,073341	4,862882	44,00 %
12	1,189710	0,036986	5,004046	41,00 %
13	1,167276	0,035397	5,100219	39,00 %
14	1,113842	0,033660	5,344890	38,00 %
15	1,086278	0,020784	5,480516	36,00 %
16	1,072648	0,014491	5,550156	34,00 %
17	1,153292	0,025729	5,162061	30,00 %
18	1,123958	0,042520	5,296785	29,00 %
19	1,124946	0,042755	5,292133	27,00 %
20	1,076650	0,026870	5,529525	27,00 %
21	1,063270	0,054827	5,599108	26,00 %
22	1,067802	0,042284	5,575344	25,00 %
23	1,055414	0,032046	5,640785	24,00 %
24	1,026576	0,010677	5,799243	24,00 %
25	1,030986	0,024433	5,774437	23,00 %
26	1,035368	0,054671	5,749998	22,00 %
27	1,000854	0,011747	5,948284	22,00 %
28	,991356	0,012727	6,005273	21,00 %
29	,989672	0,030870	6,015492	20,00 %
30	,986363	0,042367	6,035672	20,00 %
31	,974733	0,048528	6,107686	19,00 %
32	,963382	0,032202	6,179650	19,00 %

Tabela 18: Slice - escalonamento por AUNF - teste Denso A.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	13,014060	0,019974	-	- %
2	6,714582	0,005656	1,938178	96,00 %
3	4,742060	0,014120	2,744389	91,00 %
4	3,724884	0,073952	3,493816	87,00 %
5	3,073160	0,058438	4,234748	84,00 %
6	2,667406	0,038118	4,878919	81,00 %
7	2,360068	0,031921	5,514273	78,00 %
8	2,160596	0,013101	6,023365	75,00 %
9	2,053854	0,059084	6,336409	70,00 %
10	1,976526	0,039818	6,584310	65,00 %
11	1,864110	0,093861	6,981379	63,00 %
12	1,790886	0,021367	7,266827	60,00 %
13	1,685814	0,070220	7,719748	59,00 %
14	1,627366	0,067786	7,997008	57,00 %
15	1,514380	0,097805	8,593655	57,00 %
16	1,521684	0,057332	8,552406	53,00 %
17	1,612196	0,044436	8,072256	47,00 %
18	1,593610	0,082669	8,166402	45,00 %
19	1,465108	0,012512	8,882662	46,00 %
20	1,473656	0,072855	8,831138	44,00 %
21	1,380690	0,024418	9,425765	44,00 %
22	1,394504	0,067069	9,332393	42,00 %
23	1,321062	0,036762	9,851210	42,00 %
24	1,280812	0,067734	10,160788	42,00 %
25	1,303096	0,059749	9,987030	39,00 %
26	1,272532	0,040385	10,226901	39,00 %
27	1,284482	0,010970	10,131757	37,00 %
28	1,250974	0,076830	10,403141	37,00 %
29	1,281332	0,049959	10,156665	35,00 %
30	1,241858	0,089367	10,479507	34,00 %
31	1,257948	0,084190	10,345467	33,00 %
32	1,277820	0,040516	10,184579	31,00 %

Tabela 19: Slice - escalonamento por AUNF - teste DENSO B.

Número de Processadores	Tempo de Execução		*Speedup*	Eficiência
	Média	Desvio Padrão		
1	80,211800	0,0914400	-	- %
2	34,964300	0,030185	2,294105	114,00 %
3	23,084420	0,099854	3,474715	115,00 %
4	17,412180	0,028220	4,606648	115,00 %
5	14,253360	0,031137	5,627571	112,00 %
6	11,903380	0,049103	6,738573	112,00 %
7	10,307940	0,031018	7,781554	111,00 %
8	9,336952	0,073866	8,590790	107,00 %
9	8,195608	0,037783	9,787168	108,00 %
10	7,424568	0,030153	10,803564	108,00 %
11	6,881746	0,030313	11,655733	105,00 %
12	6,252424	0,079466	12,828912	106,00 %
13	5,990370	0,053782	13,390124	103,00 %
14	5,909862	0,062863	13,572533	96,00 %
15	5,461962	0,051222	14,685528	97,00 %
16	5,309690	0,073413	15,106682	94,00 %
17	4,964894	0,020465	16,155793	95,00 %
18	4,639856	0,031063	17,287562	96,00 %
19	4,369046	0,032417	18,359110	96,00 %
20	4,272940	0,032095	18,772039	93,00 %
21	4,180504	0,061879	19,187112	91,00 %
22	4,177090	0,085882	19,202794	87,00 %
23	3,829066	0,030170	20,948137	91,00 %
24	3,809730	0,050741	21,054457	87,00 %
25	3,873178	0,062584	20,709556	82,00 %
26	3,774622	0,062709	21,250286	81,00 %
27	3,499926	0,043922	22,918141	84,00 %
28	3,358450	0,066261	23,883577	85,00 %
29	3,340434	0,062299	24,012388	82,00 %
30	3,205754	0,024458	25,021196	83,00 %
31	3,054564	0,045728	26,259656	84,00 %
32	2,941724	0,0121872	27,266935	85,00 %