PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# UMA ABORDAGEM PARALELA PARA O ALGORITMO SPLIT

FELIPE FRANCIOSI

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Paulo Henrique Lemelle Fernandes

**Porto Alegre**
**2008**

**Dados Internacionais de Catalogação na Publicação (CIP)**

F817a Franciosi, Felipe
Uma abordagem paralela para o algoritmo Split / Felipe Franciosi. – Porto Alegre, 2008.
70 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Paulo Hentique Lemelle Fernandes.

1. Informática. 2. Algoritmos. 3. Álgebra Tensorial. 4. Avaliação de Desempenho (Informática). I. Fernandes, Paulo Henrique Lemelle. II. Título.

CDD 004.2

**Ficha Catalográfica elaborada pelo**
**Setor de Tratamento da Informação da BC-PUCRS**

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 **"Uma Abordagem Paralela para o Algoritmo Split"**, apresentada por Felipe Mainieri Franciosi, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 28/02/08 pela Comissão Examinadora:

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

Prof. Dr. Avelino Francisco Zorzo – PPGCC/PUCRS

Prof. Dr. Philippe Olivier Alexandre Navaux – UFRGS

Homologada em 12/04/2011, conforme Ata No. 005/11. pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

**Campus Central**
Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

*Aos atuais e futuros pesquisadores de avaliação de desempenho.*

# AGRADECIMENTOS

Agradeço ao professor Paulo Fernandes, amigo e orientador, pelos incontáveis auxílios durante meu tempo de PUCRS e pela oportunidade e voto de confiança depositado em mim nesta pós-graduação.
Ao professor Avelino Zorzo, por todo o tempo dedicado em me ajudar neste trabalho, assim como pelas dicas e técnicas compartilhadas na arte de espetar e assar carne.
Aos meus colegas do GSP pelo apoio, compreensão e paciência durante nossos trabalhos no ano de 2007.
Aos colegas e professores do CPAD, hoje CPPH, pela convivência que resultou em grandes amizades que manteremos no decorrer de nossas carreiras. Em especial, ao professor César De Rose pela disponibilização dos equipamentos necessários para obtenção dos resultados desta pesquisa.
Aos funcionários e professores da Faculdade de Informática e do Programa de Pós-Graduação pelos seus excelentes desempenho e seriedade profissional.
Ao meu colega Felipe Meneguzzi, pela hospitalidade em Londres durante variadas épocas deste ano e pelas ajudas com LaTeX.
Aos meus colegas Ricardo Melo Czekster e Thais Webber, pela hospitalidade em Grenoble e auxílios na elaboração deste trabalho.
Por último, mas não menos importante, à minha família por todo o suporte, paciência, carinho e amor dedicados durante todo o meu tempo nesta universidade.

# UMA ABORDAGEM PARALELA PARA O ALGORITMO SPLIT

## RESUMO

A análise comportamental de um processo permite a detecção de deficiências, assim como avaliar o impacto no desempenho do mesmo causado por mudanças no ambiente. O uso de modelos analíticos para descrever o processo em observação fornece estes dados através da resolução de sistemas de equações. No caso de modelagens feitas com a utilização de formalismos como Rede de Autômatos Estocásticos, a resolução destes sistemas depende da multiplicação de vetores por estruturas definidas através da álgebra tensorial. Por ter um alto custo computacional, diversos algoritmos foram propostos para resolver esta multiplicação. Recentemente a solução iterativa Split foi proposta, e o objetivo deste trabalho é apresentar alternativas paralelas e otimizações para a mesma, buscando um melhor desempenho da solução.

**Palavras-chave**: Avaliação de Desempenho, Álgebra Tensorial, Alto Desempenho, Split.

# A PARALLEL APPROACH TO THE SPLIT ALGORITHM

## ABSTRACT

The behavioral analysis of a process allows the detection of deficiencies, as well as assessing performance impact caused by environmental changes. The use of analytical models to describe the observed process provides these data through the resolution of equation systems. In the case where modeling is made using formalisms such as Stochastic Automata Network, the resolution of these systems depends on the multiplication of vectors by structures defined using tensor algebra. In view of these multiplications having a high computational cost, various algorithms have been proposed to solve it. Recently, the Split iterative solution was proposed, and the goal of this work is to provide a parallel optimized alternative for it, looking for an improved performance of the solution.

**Keywords**: Performance Evaluation, Tensorial Algebra, High Performance, Split.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ALGORITMOS

# LISTA DE SIGLAS

| | |
|---|---|
| ASP | *Alternate Service Patterns* |
| ATC | *Álgebra Tensorial Clássica* |
| ATG | *Álgebra Tensorial Generalizada* |
| AUNF | *Additive Unitary Normal Factor* |
| CM | Cadeia de Markov |
| Q | Gerador Infinitesimal |
| SAN | *Stochastic Automata Network* |
| DM | Descritor Markoviano |
| MT | Matriz de Transição |
| SPN | *Stochastic Petri Nets* |
| ATC | Álgebra Tensorial Clássica |
| MVD | Multiplicação Vetor Descritor |
| ATG | Álgebra Tensorial Generalizada |
| COW | *Cluster of Workstations* |
| MIMD | *Multiple Instructions Multiple Data* |
| MPI | *Message Passing Interface* |

# SUMÁRIO

# 1. INTRODUÇÃO

Para estudar o comportamento de um sistema, uma possível abordagem é elaborar um modelo que permita simular os cenários e estados em que ele possa se encontrar. Estas simulações geralmente são baseadas na geração de números aleatórios que atuam nas decisões de comportamento e interação dos componentes do sistema analisado. Após executar a simulação até que um pré-determinado critério de parada seja atingido, os resultados são analisados e, através de métodos estatísticos, conclusões são obtidas.

Uma outra abordagem capaz de obter estas conclusões é realizada através de uma modelagem distinta. Neste caso, identifica-se todos os possíveis estados do sistema, as transições que levam o ambiente de um estado para outro e as taxas em que as mesmas ocorrem. Estes dados são apresentados na forma de um autômato, denominado Cadeia de Markov (CM), que também pode ser representado por uma matriz contendo as taxas destas transições e conhecida como Gerador Infinitesimal (Q).

Apesar deste mecanismo servir o seu propósito, o mapeamento de sistemas com muitos estados pode se tornar uma tarefa complicada principalmente pela quantidade de memória consumida no processo. Estas complicações deram surgimento a outros formalismos, como Rede de Autômatos Estocásticos (SAN, do inglês *Stochastic Automata Network*), onde procura-se mapear subsistemas do ambiente em autômatos menores. Nestes autômatos, as transições são denominadas eventos, que podem ser locais ou sincronizantes, sendo os últimos responsáveis pela integração dos autômatos por promover a ocorrência do evento em mais de um autômato simultaneamente.

Visto que toda SAN possui uma CM correspondente, é justo afirmar que uma SAN também pode ser armazenada através de um Gerador Infinitesimal. No entanto, uma forma mais econômica de armazenar estes dados é oferecida pelo Descritor Markoviano (DM) que, de uma forma geral, trata-se do somatório dos produtos tensoriais das matrizes de transição de cada autômato da rede. Assim sendo, pode-se afirmar que o DM é uma descrição compacta de Q.

Dando seqüência ao estudo comportamental de um modelo, busca-se então a permanência nos estados do sistema. Para isto, procura-se um vetor de probabilidades $\pi$, denominado auto vetor, cuja multiplicação por Q resulte no próprio $\pi$. A medida que os sistemas analisados crescem, torna-se necessário utilizar o DM ao invés de Q, uma vez que a memória necessária para armazenar todos os dados pertinentes ao processo cresce além dos limites computacionais atuais. Por causa disto, o uso de soluções iterativas também aparece como alternativa aos métodos diretos para obtenção de $\pi$, uma vez que seu consumo de memória é menor e, desta forma, é possível atingir um equilíbrio entre a quantidade de memória e o tempo de processamento necessário para que uma solução seja atingida.

Mesmo assim, existem sistemas grandes o suficiente para que a busca de suas soluções demore mais do que o desejado. Nestes casos, a paralelização destes algoritmos aparece como uma alternativa para acelerar suas execuções. Este trabalho propõem a paralelização de um destes algoritmos

iterativos, denominado *Split*, através de um mecanismo capaz de analisar o problema em questão e realizar este processamento distribuído. Ele está organizado da seguinte forma: o Capítulo 2 introduz o estado da arte no que diz respeito ao formalismo markoviano, discutindo CM, SAN e abordando o formato tensorial para a explicação do DM. No Capítulo 3, o processo de multiplicar um vetor de probabilidades pelo DM é explicado, discutindo-se três algoritmos iterativos para realizá-lo. No Capítulo 4, a paralelização de um destes algoritmos é discutida e um protótipo é apresentado juntamente a otimizações e resultados. Por fim, o Capítulo 5 apresenta as conclusões da pesquisa e trabalhos futuros.

# 2. ESTADO DA ARTE

Este capítulo apresenta o estado da arte relacionado ao formalismo markoviano. Para isto, inicialmente os conceitos de Cadeia de Markov (CM) são explicados, seguidos de Redes de Autômatos Estocásticos (SAN) e do Descritor Markoviano (DM). Para a compreensão do DM, torna-se necessário introduzir a álgebra de tensores, formalismo matemático que define as operações realizadas entre as matrizes do descritor. Os conceitos apresentados neste capítulo foram retirados de [STE94].

## 2.1 Cadeia de Markov

De acordo com o que foi introduzido no Capítulo 1, a Cadeia de Markov é um autômato geralmente utilizado na análise estocástica de um dado sistema. Este autômato é descrito através de um grafo dirigido, onde cada nodo representa um estado do sistema e onde cada aresta representa uma possível transição de um estado para outro, associada de uma taxa de ocorrência desta transição.

Para ilustrar o conceito de CM, imagina-se uma rede hipotética de transmissão de dados composta por mais de um nodo. Cada nodo pode estar desligado, ocioso ou em comunicação. Não há restrição no que diz respeito aos nodos estarem ligados ou desligados, no entanto, sempre que um nodo entrar em modo de comunicação, os demais integrantes da rede deverão acompanhá-lo para este estado. Ao encerrar a troca de dados, todos os nodos entram em ociosidade, porém se algum nodo falhar durante a comunicação e se desligar, os demais componentes devem tornar-se ociosos.

A Figura 2.1 ilustra uma possível modelagem em Cadeia de Markov desta rede, considerando um ambiente composto por dois nodos. Para nomear os estados da cadeia, utilizou-se a seguinte convenção: o nome de cada estado será definido por duas letras, a primeira representando a situação do primeiro nodo e a segunda representando a situação do segundo. Estas letras serão:

- O para desligado (do inglês *off*);
- I para ocioso (do inglês *idle*);
- C para comunicando (do inglês *communicating*).

Neste modelo, as taxas das transições foram especificadas através de valores simbólicos, sendo:

- $T_l$ a taxa com que um nodo liga, ou seja, passa de desligado (O) para ocioso (I);
- $T_d$ a taxa com que um nodo desliga, ou seja, passa de ocioso (I) para desligado (O);
- $T_c$ a taxa com que um nodo comunica, ou seja, passa de ocioso (I) para comunicando (C);
- $T_e$ a taxa com que um nodo encerra sua comunicação, ou seja, passa de comunicando (C) para ocioso (I);

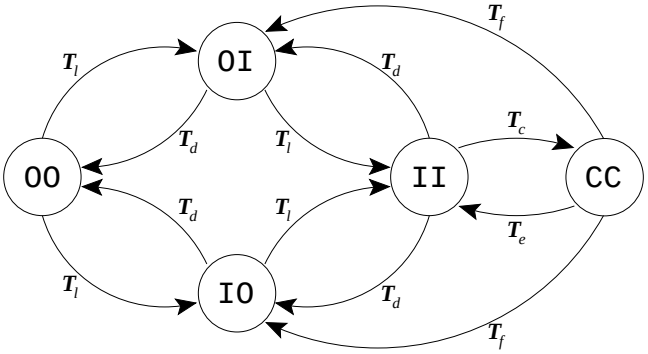


Figura 2.1: Modelagem em Cadeia de Markov de uma rede de transmissão de dados.

- $T_f$ a taxa com que um nodo falha, ou seja, passa de comunicando (C) para desligado (O), enquanto que os demais nodos passam de comunicando (C) para ocioso (I).

Esta cadeia é representada através de uma Matriz de Transição (MT), também denominada de Gerador Infinitesimal (Q). Para o modelo da Figura 2.1, esta matriz se apresenta na Equação 2.1. Nela, os estados são ordenados com uma regra pré-estabelecida (neste caso, a ordem lexicográfica do nome dos mesmos) e distribuídos de forma que cada célula $C_{ij}$ indica a taxa de transição do estado $i$ para o estado $j$.

$$Q = \begin{pmatrix} 0 & T_e & T_f & T_f & 0 \\ T_c & 0 & T_d & T_d & 0 \\ 0 & T_l & 0 & 0 & T_d \\ 0 & T_l & 0 & 0 & T_d \\ 0 & 0 & T_l & T_l & 0 \end{pmatrix} \tag{2.1}$$

No entanto, para que esta matriz seja realmente considerada um Gerador Infinitesimal, por definição é necessário realizar um ajuste para que a soma de cada linha seja igual a *zero* [STE94]. Este ajuste é realizado adicionando-se o complemento da soma de todos os elementos da linha ao elemento da diagonal principal. Para formalizar este conceito tem-se que, se $C_{ij}$ é a célula da linha $i$ e coluna $j$ da matriz, a Equação 2.2 realiza o ajuste em questão.

$$C_{xx} = -\sum_{y=1}^{n} C_{xy}, \text{para todo } x \text{ de 1 a } n \tag{2.2}$$

Executando este procedimento para a matriz em questão, tem-se um Gerador Infinitesimal como o da Equação 2.3.

$$Q = \begin{pmatrix} -(2*T_f+T_e) & T_e & T_f & T_f & 0 \\ T_c & -(2*T_d+T_c) & T_d & T_d & 0 \\ 0 & T_l & -(T_l+T_d) & 0 & T_d \\ 0 & T_l & 0 & -(T_l+T_d) & T_d \\ 0 & 0 & T_l & T_l & -(2*T_l) \end{pmatrix} \tag{2.3}$$

De posse de Q, torna-se possível calcular a taxa de permanência em cada elemento do espaço de estados descrito, ou seja, a distribuição estacionária do modelo. Para o exemplo, este dado permite que se calcule, por exemplo, qual a probabilidade do modelo encontrar-se com todos os nodos ligados (através da interpretação das probabilidades de permanência nos estados II e CC).

Esta distribuição estacionária é dada pelo auto vetor da Matriz de Transição, ou seja, por um vetor $\pi$ que multiplicado por $Q$ resulte no próprio $\pi$. Uma das formas de obtenção deste vetor é dada por um algoritmo iterativo denominado Método da Potência [STE94], que consiste na sucessiva multiplicação de um vetor pela MT. Para que seja possível realizar tal operação, a dimensão do vetor deve ser igual a ordem da matriz, e a soma de todos os elementos do vetor devem resultar em *um*.

É importante salientar que, para que o Método da Potência atinja convergência, Q seja antes transformado em uma matriz de probabilidades. Para que isto aconteça, duas condições precisam ser cumpridas. A primeira trata-se da necessidade de que não haja valores negativos na tabela, uma vez que não existem probabilidades negativas. A segunda exige que a soma de todos os elementos de uma linha seja igual a *um*, significando que todas as probabilidades de transição daquele estado estão cobertas.

Este procedimento de transformação é chamado de normalização e pode ser realizado através de dois passos. O primeiro consiste em encontrar o maior elemento em módulo da matriz e dividir toda a matriz por ele. Com isto, não haverá nenhum elemento maior do que *um* na matriz. O segundo passo trata-se de somar uma matriz identidade a MT, adicionando-se assim o valor escalar *um* à diagonal principal da matriz em transformação.

Como a soma dos elementos de cada linha do Gerador Infinitesimal era *zero* antes da normalização, agora ela passa a ser *um*, conforme o exigido. Também é interessante observar que após estes procedimentos não é possível haver valores negativos na matriz, uma vez que, se houvessem tais valores na matriz original, eles estariam na diagonal principal. Com a execução do primeiro passo, todos os valores passam a ser menores do que *um* (mesmo que negativos) e, com a execução do segundo passo, todos os valores negativos da diagonal principal passam a ser positivos.

Apesar do uso de Cadeias de Markov se mostrar uma abordagem simples, ele começa a se tornar computacionalmente inviável a medida em que o espaço de estados do sistema aumenta. Isto ocorre uma vez que a matriz de transição cresce de tamanho ao ponto de exceder os limites computacionais atuais capazes de mantê-la de maneira integral em memória primária. A necessidade de utilizar sistemas secundários de armazenamento de dados para a realização de operações como o *swap* para operar todos os dados torna o processo de resolução mais lento que o desejável.

No caso do modelo da rede de comunicação de dados apresentado nesta seção, a quantidade de estados da CM é dada pelas possíveis combinações de estados O e I para cada nodo, mais um estado representando todos os nodos em comunicação (CC). Sendo $m$ o número de autômatos, este valor é obtido por $2^m + 1$. Desta forma, se um modelo semelhante fosse criado para uma rede com oito nodos, a CM teria 257 estados, mostrando-se assim o crescimento exponencial do modelo. Para contornar este problema, novos formalismos foram introduzidos. Este trabalho irá utilizar o formalismo denominado Rede de Autômatos Estocásticos.

## 2.2 Rede de Autômatos Estocásticos

Uma Rede de Autômatos Estocásticos (SAN, do inglês *Stochastic Automata Network*) [PLA91] trata-se de um conjunto de autômatos representando partes de um sistema a ser analisado. Seu uso, em contraste ao uso de Cadeias de Markov, além de proporcionar uma maior facilidade na criação de modelos, uma vez que o processo é simplificado pela utilização de diversos autômatos menores, permite que a limitação apresentada na seção anterior seja rompida. Isto ocorre uma vez que, ao invés de uma grande matriz de transição, torna-se possível resolver uma SAN através de operações tensoriais em um conjunto de matrizes menores, necessitando-se assim de menos memória para manter todos os dados acessíveis.

Para o entendimento do funcionamento de uma SAN, apresenta-se a rede hipotética de transmissão de dados descrita na seção anterior. Em SAN, a rede, também composta por dois nodos, pode ser modelada através de dois autômatos, $N^{(1)}$ e $N^{(2)}$ conforme a Figura 2.2. Cada autômato possui três estados, seguindo a mesma nomenclatura utilizada na representação em CM.

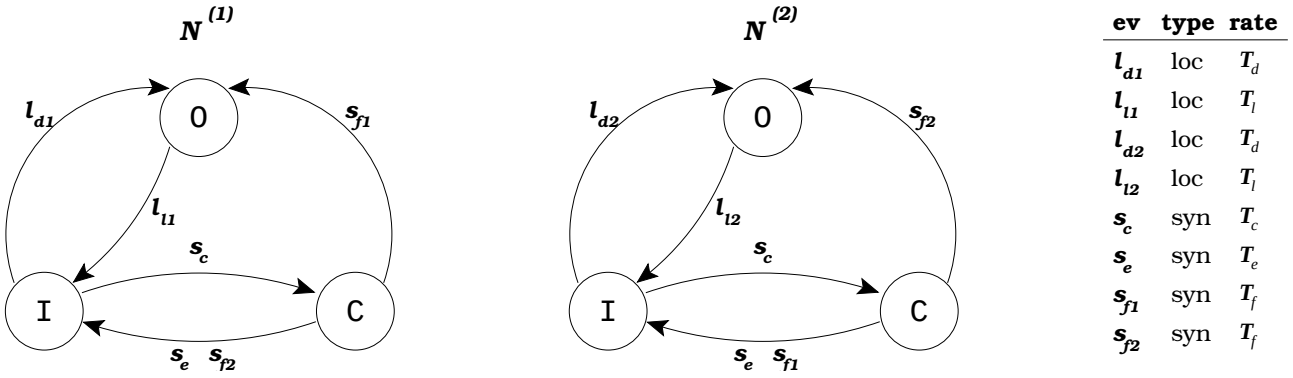


Figura 2.2: Modelagem em SAN de uma rede de transmissão de dados.

Os autômatos trocam de estado através de eventos, que podem ser locais ou sincronizantes. Os eventos locais, representados na figura como $l_i$, alteram apenas o estado do autômato em questão. Já os eventos sincronizantes, representados na figura como $s_i$, alteram os estados de dois ou mais autômatos simultaneamente. As taxas de ocorrência destes eventos são apresentadas na tabela que acompanha os autômatos.

Ainda sobre as taxas de ocorrência dos eventos, observa-se que é possível a especificação de taxas baseadas em funções. Estas funções, em geral, incluem dependência sobre outros estados do modelo e podem indicar, por exemplo, se um evento pode ou não ocorrer dependendo do número de autômatos da SAN encontrarem-se em um certo estado. Estas funções podem facilitar a representação de um modelo, mas podem ser substituídas por eventos sincronizantes [STE94], não sendo estritamente necessárias. Pelo objetivo deste trabalho não depender das mesmas, este assunto não será aprofundado.

Outra importante observação é verificar que todo modelo descrito em SAN também pode ser representado através de uma CM, uma vez que ambos formalismos são baseados nas mesmas propriedades e conceitos. Assim sendo, *estado local* é como se referencia o estado em que um autômato

qualquer se encontra, enquanto que *estado global* se trata de um conjunto de estados, representando aquilo que seria o estado na CM equivalente ao modelo em estudo.

Compreendendo estes conceitos, cita-se a existência de uma outra função cujo objetivo não é a determinação de taxas de eventos. Trata-se da *função de atingibilidade*, que representa quais estados globais da SAN são atingíveis. O produto cartesiano da dimensão de todos os autômatos de uma SAN representa o espaço de estados do modelo, no entanto, conforme o exemplo da Figura 2.2, nem todos são atingíveis. Esta função torna-se importante para a definição inicial do vetor de probabilidades utilizado no Método da Potência e será discutida novamente no Capítulo 4.

Assim como a matriz de transição apresentada na Seção 2.1 para as Cadeias de Markov, as SANs também possuem uma representação numérica. No entanto, conforme já foi discutido, esta representação deve consumir menos memória do que a matriz equivalente para a CM, uma vez que um dos objetivos do uso de SAN é a redução de memória necessária para a resolução de um problema. A próxima seção apresenta o formato tensorial, introduzindo as operações da álgebra de tensores e, em seguida, o Descritor Markoviano, que se trata da denominação do conjunto de matrizes e operações que definem esta representação mais compacta do modelo.

## 2.3 Formato Tensorial

Uma das vantagens de se trabalhar com SAN é a possibilidade de descrever todo o sistema em um formato mais compacto, denominado Descritor Markoviano. Apesar de outros sistemas poderem ser representados através de DMs, como é o caso das Redes de Petri Estocásticas (SPN, do inglês *Stochastic Petri Nets*) [AJM84], esta seção irá tratar o caso particular onde o sistema descrito é uma SAN.

Para que seja possível o entendimento do DM, antes é necessária a apresentação dos dois operadores da Álgebra Tensorial Clássica (ATC): o Produto Tensorial (também chamado de *Produto de Kronecker*) e a Soma Tensorial. Estes conceitos foram retirados do estudo realizado em [FER98].

### 2.3.1 Produto Tensorial

O produto tensorial de duas matrizes $A$ e $B$ de dimensões $(\alpha_1 \times \alpha_2)$ e $(\beta_1 \times \beta_2)$, respectivamente, definido por $A \otimes B$, trata-se de uma matriz, ou um tensor, de dimensões $(\alpha_1\beta_1 \times \alpha_2\beta_2)$. Como exemplo, apresenta-se as matrizes $A$ e $B$:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

Definidas estas matrizes de exemplo, a Equação 2.4 apresenta a matriz $C$, que se trata do resultado de $A \otimes B$.

$$C = A \otimes B = \left(\begin{array}{ccc|ccc} a_{11}b_{11} & a_{11}b_{12} & a_{11}b_{13} & a_{12}b_{11} & a_{12}b_{12} & a_{12}b_{13} \\ a_{11}b_{21} & a_{11}b_{22} & a_{11}b_{23} & a_{12}b_{21} & a_{12}b_{22} & a_{12}b_{23} \\ a_{11}b_{31} & a_{11}b_{32} & a_{11}b_{33} & a_{12}b_{31} & a_{12}b_{32} & a_{12}b_{33} \\ \hline a_{21}b_{11} & a_{21}b_{12} & a_{21}b_{13} & a_{22}b_{11} & a_{22}b_{12} & a_{22}b_{13} \\ a_{21}b_{21} & a_{21}b_{22} & a_{21}b_{23} & a_{22}b_{21} & a_{22}b_{22} & a_{22}b_{23} \\ a_{21}b_{31} & a_{21}b_{32} & a_{21}b_{33} & a_{22}b_{31} & a_{22}b_{32} & a_{22}b_{33} \end{array}\right) \tag{2.4}$$

Conforme pode ser observado neste exemplo, o tensor resultante pode ser visualizado como $(\alpha_1 \times \alpha_2)$ quadrantes de dimensões $(\beta_1 \times \beta_2)$, uma vez que o produto é obtido através da multiplicação de cada elemento escalar $a_{ij}$ da matriz $A$ pela matriz $B$, colocando-se a matriz resultante no quadrante $ij$.

Assim sendo, o espaço de estados do produtório em questão (dimensão do tensor resultante) pode ser calculado previamente através da multiplicação das dimensões das matrizes envolvidas na operação. Juntamente com este conceito, apresenta-se a definição de $nleft_i$ e $nright_i$, que se tratam do espaço de estados a esquerda ou a direita, respectivamente, da $i$-ésima matriz do produtório.

O cálculo destes valores é realizado através da multiplicação das ordens de todas as matrizes a esquerda (para o $nleft$) ou a direita (para o $nright$) da matriz $i$, considerando-se dois casos especiais: o $nleft$ da primeira matriz e o $nright$ da última matriz do produtório que serão, por definição, iguais a *um*. Estes conceitos serão utilizados nos algoritmos apresentados no Capítulo 3.

Uma vez compreendido o Produto Tensorial, torna-se possível o entendimento do conceito de Fator Normal, que é utilizado para a apresentação da Seção 2.3.2. O Fator Normal é um caso particular do Produto Tensorial onde os termos envolvidos são uma matriz quadrada e uma matriz identidade de qualquer ordem. Tomando a matriz $A$ do exemplo anterior e uma matriz identidade $I_3$ (de ordem 3), torna-se possível combinar dois produtos: $A \otimes I_3$ e $I_3 \otimes A$. O resultado destas operações encontra-se nas Equações 2.5 e 2.6, respectivamente.

$$A \otimes I_3 = \left(\begin{array}{ccc|ccc} a_{11} & 0 & 0 & a_{12} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{12} & 0 \\ 0 & 0 & a_{11} & 0 & 0 & a_{12} \\ \hline a_{21} & 0 & 0 & a_{22} & 0 & 0 \\ 0 & a_{21} & 0 & 0 & a_{22} & 0 \\ 0 & 0 & a_{21} & 0 & 0 & a_{22} \end{array}\right) \tag{2.5}$$

$$I_3 \otimes A = \left(\begin{array}{cc|cc|cc} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & a_{11} & a_{12} & 0 & 0 \\ 0 & 0 & a_{21} & a_{22} & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & a_{11} & a_{12} \\ 0 & 0 & 0 & 0 & a_{21} & a_{22} \end{array}\right) \tag{2.6}$$

O conceito de Fator Normal, além de facilitar a visualização da operação de um Produto Tensorial, é especialmente importante para a compreensão dos algoritmos utilizados neste trabalho para

realizar a Multiplicação Vetor Descritor (MVD), apresentada no Capítulo 3. Uma vez compreendido este conceito, pode-se apresentar a Soma Tensorial.

### 2.3.2 Soma Tensorial

A Soma Tensorial, apesar de fazer uso do Produto Tensorial, que é definido para matrizes de quaisquer dimensões, apresenta-se apenas para matrizes quadradas. Esta restrição ocorre pela necessidade do uso de matrizes identidades da mesma ordem dos elementos da soma na operação. A Equação 2.7 define como é esta relação, apresentando a soma de duas matrizes $X$ e $Y$, de ordens $m$ e $n$, respectivamente.

$$X_m \oplus Y_n = (X_m \otimes I_n) + (I_m \otimes Y_n) \tag{2.7}$$

Fazendo uso das matrizes $A$ e $B$ definidas na Seção 2.3.1 como exemplo, a operação de soma tensorial pode ser visualizada na Equação 2.10. Como a matriz $A$ é uma matriz quadrada de ordem dois e a matriz $B$ é uma matriz quadrada de ordem três, o processo ocorre através da soma dos tensores resultantes entre o produto tensorial de $A$ por uma matriz identidade de ordem três (Equação 2.8) e de $B$ por uma matriz identidade de ordem dois (Equação 2.9).

$$(A \otimes I_3) = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \left(\begin{array}{ccc|ccc} a_{11} & 0 & 0 & a_{12} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{12} & 0 \\ 0 & 0 & a_{11} & 0 & 0 & a_{12} \\ \hline a_{21} & 0 & 0 & a_{22} & 0 & 0 \\ 0 & a_{21} & 0 & 0 & a_{22} & 0 \\ 0 & 0 & a_{21} & 0 & 0 & a_{22} \end{array}\right) \tag{2.8}$$

$$(I_2 \otimes B) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \left(\begin{array}{ccc|ccc} b_{11} & b_{12} & b_{13} & 0 & 0 & 0 \\ b_{21} & b_{22} & b_{23} & 0 & 0 & 0 \\ b_{31} & b_{32} & b_{33} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & b_{11} & b_{12} & b_{13} \\ 0 & 0 & 0 & b_{21} & b_{22} & b_{23} \\ 0 & 0 & 0 & b_{31} & b_{32} & b_{33} \end{array}\right) \tag{2.9}$$

$$A \oplus B = (A \otimes I_3) + (I_2 \otimes B) =$$

$$= \left(\begin{array}{ccc|ccc} a_{11}+b_{11} & b_{12} & b_{13} & a_{12} & 0 & 0 \\ b_{21} & a_{11}+b_{22} & b_{23} & 0 & a_{12} & 0 \\ b_{31} & b_{32} & a_{11}+b_{33} & 0 & 0 & a_{12} \\ \hline a_{21} & 0 & 0 & a_{22}+b_{11} & b_{12} & b_{13} \\ 0 & a_{21} & 0 & b_{21} & a_{22}+b_{22} & b_{23} \\ 0 & 0 & a_{21} & b_{31} & b_{32} & a_{22}+b_{33} \end{array}\right) \tag{2.10}$$

Os conceitos apresentados nesta seção definem o Formalismo Tensorial necessário para a compreensão do Descritor Markoviano, que será introduzido a seguir. No entanto, este formalismo,

também conhecido como Álgebra Tensorial Clássica, não é suficiente para tratar SANs que apresentem taxas funcionais. Para tal, torna-se necessário o estudo da Álgebra Tensorial Generalizada (ATG). Conforme discutido na Seção 2.2, este trabalho não aborda tais casos de SAN, por isso não discute tal formalismo.

### 2.3.3 Descritor Markoviano

O Descritor Markoviano é apresentado através de operações tensoriais entre matrizes de transições. O uso da Álgebra Tensorial para a descrição de SANs foi inicialmente proposto por Plateau [PLA84]. Para sua melhor definição e entendimento, o DM pode ser dividido em duas partes: uma local e uma sincronizante. A Equação 2.11 introduz este conceito.

$$DM = DM_l + DM_s \tag{2.11}$$

A parte local ($DM_l$) é composta pela soma tensorial de MTs $Q_l^{(i)}$, sendo $i$ um índice para cada autômato, e sendo cada matriz as transições dos eventos locais deste autômato. A Equação 2.12 define este somatório tensorial, apresentando o símbolo da soma tensorial ($\oplus$) como um somatório e $N$ como o número total de autômatos.

$$DM_l = \bigoplus_{i=1}^{N} Q_l^{(i)} \tag{2.12}$$

A parte sincronizante ($DM_s$) do DM é definida através de uma abordagem diferente. Para cada autômato, existem dois conjuntos (um positivo e outro negativo) de matrizes para cada evento. Sendo assim, se $E$ é o número de eventos sincronizantes e $N$ o número de autômatos, $DM_s$ terá $2 \times E \times N$ matrizes de transições. A Equação 2.13 define como é a interação destas matrizes.

$$DM_s = \sum_{e \in \varepsilon} \left( \bigotimes_{i=1}^{N} Q_{e^+}^{(i)} + \bigotimes_{i=1}^{N} Q_{e^-}^{(i)} \right), \text{ sendo } \varepsilon \text{ o conjunto de eventos sincronizantes} \tag{2.13}$$

No que diz respeito aos conjuntos positivos e negativos de matrizes, tem-se que a matriz positiva $Q_{e^+}^{(i)}$ representa as transições do evento sincronizante $e$ no autômato $i$, enquanto que a matriz negativa $Q_{e^-}^{(i)}$ representa o ajuste relacionado a matriz positiva. Um detalhe importante destes conjuntos é que as taxas do evento apenas serão colocadas na matriz relacionada ao autômato que iniciou o evento, utilizando-se o valor *1* nas demais matrizes. Este conceito existe para facilitar a compreensão dos eventos sincronizantes, indicando que o evento é do tipo mestre no autômato que o inicia e escravo nos demais.

Uma vez compreendido o fundamento das partes locais e sincronizantes do Descritor Markoviano, a Equação 2.14 é apresentada, unindo os conceitos introduzidos até aqui e representando o Gerador Infinitesimal de uma SAN.

$$Q = \bigoplus_{i=1}^{N} Q_l^{(i)} + \sum_{e \in \mathcal{E}} \left( \bigotimes_{i=1}^{N} Q_{e^+}^{(i)} + \bigotimes_{i=1}^{N} Q_{e^-}^{(i)} \right) \tag{2.14}$$

Conforme discutido na Seção 2.3.1, a álgebra tensorial permite decompor uma série de somas tensoriais em fatores normais. Isto permite a eliminação de tais operações do DM, fazendo com que a sua parte local, assim como a parte sincronizante, seja composta apenas de produtos tensoriais. A Tabela 2.1 apresenta as operações da Equação 2.14, porém permitindo uma outra visualização do DM.

Tabela 2.1: Descritor Markoviano

| | | | |
|---|---|---|---|
| $\sum$ | $N$ | | $Q_l^{(1)} \otimes I_{n_2} \otimes \cdots \otimes I_{n_{N-1}} \otimes I_{n_N}$<br>$I_{n_1} \otimes Q_l^{(2)} \otimes \cdots \otimes I_{n_{N-1}} \otimes I_{n_N}$<br>$\vdots$<br>$I_{n_1} \otimes I_{n_2} \otimes \cdots \otimes Q_l^{(N-1)} \otimes I_{n_N}$<br>$I_{n_1} \otimes I_{n_2} \otimes \cdots \otimes I_{n_{N-1}} \otimes Q_l^{(N)}$ |
| | $2E$ | $e^+$ | $Q_{1^+}^{(1)} \otimes Q_{1^+}^{(2)} \otimes \cdots \otimes Q_{1^+}^{(N-1)} \otimes Q_{1^+}^{(N)}$<br>$\vdots$<br>$Q_{E^+}^{(1)} \otimes Q_{E^+}^{(2)} \otimes \cdots \otimes Q_{E^+}^{(N-1)} \otimes Q_{E^+}^{(N)}$ |
| | | $e^-$ | $Q_{1^-}^{(1)} \otimes Q_{1^-}^{(2)} \otimes \cdots \otimes Q_{1^-}^{(N-1)} \otimes Q_{1^-}^{(N)}$<br>$\vdots$<br>$Q_{E^-}^{(1)} \otimes Q_{E^-}^{(2)} \otimes \cdots \otimes Q_{E^-}^{(N-1)} \otimes Q_{E^-}^{(N)}$ |

De acordo com o observado na Tabela 2.1, a modificação na parte local do DM elimina as somas tensoriais, deixando apenas operações do tipo produto tensorial. Esta modificação torna-se fundamental para a multiplicação do DM por um vetor de probabilidades, como será explicado no Capítulo 3.

Para concluir a compreensão de como o DM descreve uma SAN, o modelo hipotético de rede de comunicação utilizado neste capítulo será representado no formato recém discutido. Inicialmente, a parte local do Descritor Markoviano é apresentada através das matrizes $Q_l^{(i)}$. Em seguida, a parte sincronizante é descrita pelas matrizes $Q_{e^+}^{(i)}$ e $Q_{e^-}^{(i)}$.

No que diz respeito a parte local, é possível observar o modelo na Figura 2.2 e verificar que cada autômato possui dois eventos locais. No primeiro autômato, os eventos são $l_{d1}$ e $l_{l1}$, com taxas $T_d$ e $T_l$, respectivamente. No segundo autômato, os eventos são $l_{d2}$ e $l_{l2}$, com as mesmas taxas que o primeiro. Desta forma, a parte local do DM deste modelo é descrita através da definição de $Q_l^{(1)}$

(com as taxas dos eventos locais do primeiro autômato), $Q_l^{(2)}$ (com as taxas dos eventos locais do segundo autômato) e $I_3$ como uma matriz identidade de ordem três, conforme segue:

$$Q_l^{(1)} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -T_d & T_d \\ 0 & T_l & -T_l \end{pmatrix} \quad Q_l^{(2)} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -T_d & T_d \\ 0 & T_l & -T_l \end{pmatrix} \quad I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$DM_l = (Q_l^{(1)} \otimes I_3) + (I_3 \otimes Q_l^{(2)})$$

$$DM_l = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -T_d & T_d \\ 0 & T_l & -T_l \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & -T_d & T_d \\ 0 & T_l & -T_l \end{pmatrix}$$

Já para a parte sincronizante, torna-se necessário formar as matrizes a partir dos eventos, em contraste com a parte local, onde as matrizes foram formadas a partir dos autômatos. Assim sendo, observa-se novamente a Figura 2.2 e analisa-se os quatro eventos sincronizantes $s_c$, $s_e$, $s_{f1}$ e $s_{f2}$. Para cada um deles, será necessário elaborar dois conjuntos (um positivo e outro negativo) contendo duas matrizes cada (uma para cada autômato).

No primeiro conjunto constam as matrizes sincronizantes positivas. Assim sendo, para cada evento, constam duas matrizes indicando as taxas com as quais eles ocorrem em cada autômato.

Para o evento $s_c$ e $s_e$, são elas:

$$Q_{s_c^+}^{(1)} = \begin{pmatrix} 0 & 0 & 0 \\ T_c & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_c^+}^{(2)} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_e^+}^{(1)} = \begin{pmatrix} 0 & T_e & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_e^+}^{(2)} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$Q_{s_c^-}^{(1)} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -T_c & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_c^-}^{(2)} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_e^-}^{(1)} = \begin{pmatrix} -T_e & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_e^-}^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Para o evento $s_{f1}$ e $s_{f2}$, são elas:

$$Q_{s_{f1}^+}^{(1)} = \begin{pmatrix} 0 & 0 & T_f \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_{f1}^+}^{(2)} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_{f2}^+}^{(1)} = \begin{pmatrix} 0 & T_f & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_{f2}^+}^{(2)} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$Q_{s_{f1}^-}^{(1)} = \begin{pmatrix} -T_f & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_{f1}^-}^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_{f2}^-}^{(1)} = \begin{pmatrix} -T_f & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q_{s_{f2}^-}^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Outra importante observação sobre a parte sincronizante do DM trata-se da presença do sinal

negativo no ajuste das matrizes. Como esta parte do DM é formada por produtos tensoriais, apenas é necessário negativar os elementos de ajuste em uma matriz de cada evento, visto que o processo de multiplicação irá propagar o sinal negativo para as demais matrizes.

## 2.4 Considerações Finais

Este capítulo apresentou os conceitos básicos do estado da arte relacionado a esta pesquisa. Inicialmente, discutiu Cadeia de Markov e Rede de Autômatos Estocásticos. Em seguida, o formalismo tensorial foi apresentado, dando base para a discussão da representação numérica de SAN: o Descritor Markoviano.

Estes conceitos completam o fundamento necessário para a discussão da multiplicação de um vetor de probabilidades pelo DM. Este processo, que leva o nome de Multiplicação Vetor-Descritor, é a base para a solução Split, alvo de paralelização neste trabalho.

# 3. MULTIPLICAÇÃO VETOR-DESCRITOR

Conforme foi discutido na Seção 2.1, a solução estacionária de uma Cadeia de Markov pode ser obtida através da multiplicação de um vetor de probabilidades pelo gerador infinitesimal do modelo. Para uma SAN, no entanto, o processo torna-se mais complicado devido ao formato em que o seu gerador infinitesimal (no caso, o Descritor Markoviano) se apresenta. Devido a estas diferenças, o processo recebe o nome de Multiplicação Vetor-Descritor (MVD), uma vez que ele consiste na multiplicação de um vetor de probabilidades pelo Descritor Markoviano (DM).

Uma forma de realizar esta operação seria resolver todas as operações tensoriais contidas no DM, formando assim uma única matriz (ou tensor) e multiplicar o vetor de probabilidades por ele, como numa operação normal de multiplicação de matrizes. Apesar de estudos terem sido feitos com a utilização de memória virtual para viabilizar esta técnica [DEA98], este procedimento anula a vantagem principal da utilização do DM, que é exatamente a possibilidade de se trabalhar com matrizes menores.

Assim sendo, para que seja possível realizar a MVD, algoritmos foram desenvolvidos permitindo que a operação seja executada sem a prévia resolução do DM em uma única matriz. Estas soluções consideram que o vetor $\pi$ está sendo multiplicado por uma *linha* do DM, ou seja, por um produtório tensorial ao invés de um somatório de produtos tensoriais. A Equação 3.1 mostra como $\pi$ pode ser deslocado para dentro do somatório, através do uso da propriedade distributiva.

$$\pi \times \sum_{j=1}^{N+2E} \left( \bigotimes_{i=1}^{N} Q_j^{(i)} \right) = \sum_{j=1}^{N+2E} \left( \pi \times \bigotimes_{i=1}^{N} Q_j^{(i)} \right) \tag{3.1}$$

Para a realização deste trabalho, três algoritmos iterativos foram estudados e serão apresentados na próxima seção. Os conceitos discutidos aqui foram retirados de [CZE07].

## 3.1 Soluções Iterativas

No que diz respeito a realização da MVD, o uso de algoritmos iterativos aparece como uma alternativa interessante. Por se tratarem de processos que aproximam a solução a cada iteração, é possível controlar o critério de parada tanto pela precisão obtida quanto pelo tempo de execução do algoritmo. Não importando o critério escolhido, consegue-se calcular o erro existente na solução obtida até o ponto de parada, controlando-se também a qualidade da solução calculada.

Os algoritmos descritos nesta seção são baseados no Método da Potência, já discutido na Seção 2.1, e que consiste na sucessiva multiplicação de um vetor por uma matriz de probabilidades. A cada iteração, o vetor se aproxima da solução estacionária do modelo descrito pela matriz, sendo que o algoritmo considera-se encerrado quando um critério de parada pré-estabelecido é atingido. Sendo $\pi$ e $P$ o vetor e a matriz de probabilidades, respectivamente, o método é ilustrado pela Equação 3.2.

$$\pi \times P^{\infty} = \pi \tag{3.2}$$

Em seguida, apresenta-se três soluções para realizar esta multiplicação. Cada uma destas soluções possui vantagens no que diz respeito a otimização e ganho de desempenho dependendo de certas características do DM, conforme será discutido.

### 3.1.1 Sparse

A solução Sparse, como o nome sugere, é ideal para a multiplicação de um vetor de probabilidades por um produto tensorial de matrizes esparsas, ou seja, com uma quantidade dominante de elementos nulos. Isto ocorre visto que o algoritmo, inicialmente, monta uma tabela representando todos os elementos diferentes de zero que estariam presentes no tensor resultante do produto tensorial envolvido no processo.

Cada linha desta tabela irá conter três colunas, sendo a primeira o elemento correspondente do tensor resultante e, as outras duas, índices utilizados no processo de multiplicação. O primeiro deles, denominado $base_{in}$, representa a posição do vetor de probabilidades pela qual o elemento da tabela deverá ser multiplicado. O segundo, denominado $base_{out}$, representa a posição do vetor resultante em que o resultado da multiplicação deverá ser armazenado.

Para que o processo seja realizado corretamente, é necessário atribuir o valor *zero* a todos os elementos do vetor resultante no início de cada iteração. Assim sendo, ao se realizar uma multiplicação, o valor resultante deve ser incrementado no vetor de resultados na posição indicada, uma vez que esta posição pode se repetir em diferentes linhas da tabela. Esta solução pode ser descrita como no Algoritmo 1.

Algoritmo 3.1: Representação algorítmica da solução Sparse - $\Upsilon = \upsilon \times \otimes_{i=1}^{N} Q^{(i)}$

$\Upsilon = 0$;
**para cada** $i_1, ..., i_N, j_1, ..., j_N \in \theta(1...N)$ **faça**
  $e = 1$;
  $base_{in} = base_{out} = 0$;
  **para cada** $k = 1, 2, ..., N$ **faça**
    $e = e \times q^{(k)}_{(i_k, j_k)}$;
    $base_{in} = base_{in} + ((i_k - 1) \times nright_k)$;
    $base_{out} = base_{out} + ((j_k - 1) \times nright_k)$;
  **fim**
  $\Upsilon[base_{out}] = \Upsilon[base_{out}] + \upsilon[base_{in}] \times e$;
**fim**

É importante observar que, como as linhas da tabela são utilizadas apenas uma vez por iteração, este algoritmo guarda somente os valores utilizados na iteração em questão (para as variáveis $e$, $base_{in}$ e $base_{out}$). Conforme será visto no Capítulo 4, onde o objetivo não é realizar apenas um

processo de multiplicação, mas sim uma solução iterativa otimizada, todos os valores são gerados inicialmente e guardados para utilizações futuras.

Um custo teórico desta solução pode ser dado pela Equação 3.3, que computa o número de multiplicações em ponto flutuante realizadas pelo algoritmo. No entanto, este valor considera o cálculo de $e$ recém discutido (o cálculo de $base_{in}$ e $base_{out}$ são operações sobre números inteiros e, por isso, desconsiderados). Na equação, $N$ define o número de matrizes do produto tensorial, enquanto que $nz_i$ define o número de elementos diferentes de zero na $i$-ésima matriz.

$$N \times \prod_{i=1}^{N} nz_i \tag{3.3}$$

Conforme sugerido, se uma tabela fosse realmente montada com todos os valores de $e$, $base_{in}$ e $base_{out}$, o custo de cada iteração poderia ser calculado sem a inclusão das operações gastas desta etapa inicial. O número de multiplicações em ponto flutuante neste novo caso é dado pela Equação 3.4.

$$\prod_{i=1}^{N} nz_i \tag{3.4}$$

Por este algoritmo tratar o produto tensorial como um único tensor, consumindo memória para realizar o mínimo de multiplicações necessárias a cada iteração, ele pode ser considerado como eficiente em processamento. Em contrapartida, se o computador utilizado na execução de uma implementação desta solução não possuir memória suficiente para armazenar esta tabela, o sistema operacional necessitará realizar operações de *swap*, gravando partes desta tabela em disco rígido e degradando o tempo de execução total. Outra forma de contornar esta questão é trabalhar com o processo de MVD sem a resolução completa do produtório tensorial, conforme será discutido na apresentação da próxima solução.

### 3.1.2 Shuffle

Em contraste com a solução Sparse recém apresentada, a solução Shuffle [FER98a] busca otimizar o processo MVD onde os elementos não-nulos são predominantes nas matrizes do DM. Para isto, ela faz uso da propriedade algébrica tensorial apresentada na Seção 2.3.1 que permite a decomposição de um produto tensorial em uma multiplicação de fatores normais, conforme o exemplo a seguir.

$$\begin{aligned} Q_a^{(1)} \otimes Q_b^{(2)} \otimes \ldots \otimes Q_c^{(N-1)} \otimes Q_d^{(N)} \quad = \quad & ( \; Q_a^{(1)} \otimes I_b \otimes \ldots \otimes I_c \otimes I_d \; ) \times \\ & ( \; I_a \otimes Q_b^{(2)} \otimes \ldots \otimes I_c \otimes I_d \; ) \times \\ & ( \; I_a \otimes I_b \otimes \ldots \otimes Q_c^{(N-1)} \otimes I_d \; ) \times \\ & ( \; I_a \otimes I_b \otimes \ldots \otimes I_c \otimes Q_d^{(N)} \; ) \end{aligned}$$

No exemplo, é possível observar o produto tensorial entre $N$ matrizes quadradas $Q_n^{(i)}$, sendo $i$ o

índice da matriz e $n$ a sua ordem. Após a decomposição em fatores normais, novos produtos são apresentados, cada um envolvendo apenas uma das matrizes previamente citadas. De acordo com estes novos fatores normais, é possível distinguir três categorias em que eles se encaixam.

A primeira categoria é o caso particular onde a matriz $Q^{(i)}$ encontra-se a esquerda do produtório, tendo apenas matrizes identidades a sua direita ($Q^{(1)} \otimes I_{nright_1}$). A segunda é o caso onde a matriz encontra-se entre matrizes identidades ($I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$). Por fim, a terceira é o outro caso particular onde a matriz encontra-se a direita do produtório ($I_{nleft_N} \otimes Q^{(N)}$).

Esta divisão em categorias permite identificar onde os elementos da matriz $Q^{(i)}$ irão se encontrar no produto tensorial. No caso do fator normal onde a matriz identidade encontra-se a direita no produto tensorial, cada elemento $q_{(k,l)}$ da matriz $Q^{(1)}$ aparecerá $nright_1$ vezes no tensor resultante. Suas localizações serão dadas por $((k \times nright_1) + \alpha, (l \times nright_1) + \alpha)$, com $\alpha$ variando de zero a $nright_1$. O exemplo abaixo ilustra este conceito utilizando uma matriz $A$ de ordem dois e uma matriz identidade de ordem três.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \left(\begin{array}{ccc|ccc} a_{11} & 0 & 0 & a_{12} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{12} & 0 \\ 0 & 0 & a_{11} & 0 & 0 & a_{12} \\ \hline a_{21} & 0 & 0 & a_{22} & 0 & 0 \\ 0 & a_{21} & 0 & 0 & a_{22} & 0 \\ 0 & 0 & a_{21} & 0 & 0 & a_{22} \end{array}\right)$$

Para o caso do fator normal onde a matriz identidade encontra-se a esquerda no produto tensorial, cada elemento $q_{(k,l)}$ da matriz $Q^{(N)}$ aparecerá $nleft_N$ vezes no tensor resultante, sendo suas localizações dadas por $(k + (nleft_N \times \alpha), l + (nleft_N \times \alpha))$, com $\alpha$ variando de zero a $nleft_N$. O exemplo a seguir ilustra este conceito utilizando os mesmos parâmetros do exemplo anterior.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \left(\begin{array}{cc|cc|cc} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & a_{11} & a_{12} & 0 & 0 \\ 0 & 0 & a_{21} & a_{22} & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & a_{11} & a_{12} \\ 0 & 0 & 0 & 0 & a_{21} & a_{22} \end{array}\right)$$

No caso genérico onde existe uma matriz identidade a esquerda e outra a direita da matriz $Q^{(i)}$, é possível identificar a posição dos elementos $q_{(i,j)}$ através de uma combinação das outras duas categorias apresentadas até agora. O Algoritmo 3.2, apresentado a seguir, utiliza estes conceitos para localizar quais elementos de um vetor $v$ são necessários para realizar a multiplicação por cada fator normal.

Observando-se o algoritmo, nota-se a utilização de um vetor $z_{in}$, preenchido na linha 7, que contém somente os elementos pertinentes de $v$ para a multiplicação por cada matriz $Q^{(i)}$. Esta multiplicação ocorre na linha 10, sendo o vetor resultante chamado de $z_{out}$. Em seguida, na linha 13, $v$ é atualizado com os valores de $z_{out}$.

Algoritmo 3.2: Representação algorítmica da solução Shuffle - $\Upsilon = \upsilon \times \otimes_{i=1}^{N} Q^{(i)}$

**para cada** $i = 1,2,\ldots,N$ **faça**
  $base = 0$;
  **para cada** $m = 0,1,2,\ldots,nleft_i - 1$ **faça**
    **para cada** $j = 0,1,2,\ldots,nright_i - 1$ **faça**
      $index = base + j$;
      **para cada** $l = 0,1,2,\ldots,n_i - 1$ **faça**
        $z_{in}[l] = \upsilon[index]$;
        $index = index + nright_i$;
      **fim**
      multiply $z_{out} = z_{in} \times Q^{(i)}$;
      $index = base + j$;
      **para cada** $l = 0,1,2,\ldots,n_i - 1$ **faça**
        $\upsilon[index] = z_{out}[l]$;
        $index = index + nright_i$;
      **fim**
    **fim**
    $base = base + (nright_i \times n_i)$;
  **fim**
**fim**
$\Upsilon = \upsilon$

Em suma, este procedimento toma um vetor $z_{in}$ a partir de $\upsilon$ e o multiplica por cada $Q^{(i)}$, sempre atualizando $\upsilon$ a cada iteração com o resultado do processo. Isto acaba sendo o mesmo que quebrar o produto tensorial em diversos fatores normais e multiplicar $\upsilon$ pelo primeiro fator normal, utilizando o vetor resultante na multiplicação pelo segundo fator normal e assim por diante. No entanto, o processo proposto pelo algoritmo descarta a necessidade desta quebra em fatores normais, que de outra forma consumiria memória em demasia.

No que diz respeito ao custo desta solução, tem-se que o processo monta, para o processo de multiplicação, $nleft_i \times nright_i$ vetores de tamanho $n_i$. Considerando que as matrizes são armazenadas em um formato esparso (somente os elementos diferentes de *zero* são armazenados), o número de multiplicações realizados por esta solução pode ser dado por $nleft_i \times nright_i \times nz_i$, sendo $nz_i$ o número de elementos diferentes de *zero* da matriz $Q^{(i)}$. A Equação 3.5 formaliza este conceito.

$$\sum_{i=1}^{N} nleft_i \times nright_i \times nz_i = \prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} \frac{nz_i}{n_i} \tag{3.5}$$

Esta solução, em contraste com a solução Sparse apresentada na Seção 3.1.1, realiza um número maior de multiplicações. No entanto, por tratar de forma eficiente a separação do produto tensorial em fatores normais (não necessitando armazenar em memória os fatores normais), ela consome menos memória, tornando-se mais eficiente para produtos que contém matrizes mais densas.

Tabela 3.1: Divisão do produto tensorial para aplicação da solução Split

| Split $\sigma$ | Termo do Produto Tensorial |
|---|---|
| 0 | $\overset{\sigma}{\downarrow}$ $\underbrace{Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes Q^{(N-1)} \otimes Q^{(N)}}_{shuffle}$ |
| 1 | $\overset{\sigma}{\downarrow}$ $\underbrace{Q^{(1)}}_{sparse} \otimes \underbrace{Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes Q^{(N-1)} \otimes Q^{(N)}}_{shuffle}$ |
| 2 | $\underbrace{Q^{(1)} \otimes \overset{\sigma}{\downarrow} Q^{(2)}}_{sparse} \otimes \underbrace{\ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes Q^{(N-1)} \otimes Q^{(N)}}_{shuffle}$ |
| $\vdots$ | $\vdots$ |
| N-2 | $\underbrace{Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes \overset{\sigma}{\downarrow} Q^{(N-2)}}_{sparse} \otimes \underbrace{Q^{(N-1)} \otimes Q^{(N)}}_{shuffle}$ |
| N-1 | $\underbrace{Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes \overset{\sigma}{\downarrow} Q^{(N-1)}}_{sparse} \otimes \underbrace{Q^{(N)}}_{shuffle}$ |
| N | $\overset{\sigma}{\downarrow}$ $\underbrace{Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes Q^{(N-1)} \otimes Q^{(N)}}_{sparse}$ |

### 3.1.3 Split

Tendo em vista as soluções apresentadas, uma nova abordagem surgiu para tratar eficientemente casos onde o produto tensorial possa se encontrar em uma classificação intermediária entre o esparso e o denso. Nestes casos, ambas as soluções Sparse e Shuffle deixam de aproveitar suas particularidades, enquanto que uma proposta híbrida poderia utilizar o que cada uma delas possui de melhor, inclusive nos casos em que o produto seria melhor tratado por uma destas duas soluções.

A solução Split surge com esta proposta e, como o nome sugere, consiste em dividir os produtos tensoriais em dois conjuntos de matrizes. Mantendo as devidas dependências entre estes conjuntos, um deles é tratado pela solução Sparse enquanto que o outro é tratado pela solução Shuffle. O ponto em que esta divisão é realizada é representado pela letra grega $\sigma$, e exemplos destes cortes podem ser visualizados na Tabela 3.1.

Conforme pode ser observado na Tabela 3.1, quando $\sigma$ for igual a *zero*, a solução Split se comporta exatamente igual a solução Shuffle. No outro caso extremo, quando $\sigma$ for igual a $N$, ou seja, ao número de matrizes no produto tensorial, a solução Split se comporta como a solução Sparse.

Para todos os casos intermediários, a solução quebra o produto tensorial em duas partes. Para a parte da esquerda é gerada uma tabela igual a discutida na Seção 3.1.1, com cada linha contendo um elemento escalar $e$ e seus respectivos valores $base_{in}$ e $base_{out}$. Um vetor $v_{in}$ do tamanho de $nright_{\sigma}$ é então montado, contendo as partes pertinentes de $v$ (de acordo com $base_{in}$) e já multiplicado por

cada elemento $e$ da tabela esparsa. Este vetor é então multiplicado pela parte direita do produto tensorial, utilizando o algoritmo Shuffle, e o vetor solução $\upsilon_{out}$ é acumulado de volta em $\upsilon$ de acordo com $base_{out}$. A solução é apresentada no Algoritmo 3.3.

Algoritmo 3.3: Representação algorítmica da solução Split - $\Upsilon = \upsilon \times \otimes_{i=1}^{N} Q^{(i)}$

$\Upsilon = 0$;
**para cada** $i_1, \ldots, i_\sigma, j_1, \ldots, j_\sigma \in \theta(1 \ldots \sigma)$ **faça**
  $e = 1$;
  $base_{in} = base_{out} = 0$;
  **para cada** $k = 1, 2, \ldots, \sigma$ **faça**
    $e = e \times q^{(k)}_{(i_k, j_k)}$;
    $base_{in} = base_{in} + ((i_k - 1) \times nright_{(k)})$;
    $base_{out} = base_{out} + ((j_k - 1) \times nright_{(k)})$;
  **fim**
  **para cada** $l = 0, 1, 2, \ldots, nright_\sigma - 1$ **faça**
    $\upsilon_{in}[l] = \upsilon[base_{in} + l] \times e$;
  **fim**
  **para cada** $i = \sigma + 1, \ldots, N$ **faça**
    $base = 0$;
    **para cada** $m = 0, 1, 2, \ldots, \frac{nleft_i}{nleft_\sigma} - 1$ **faça**
      **para cada** $j = 0, 1, 2, \ldots, nright_i$ **faça**
        $index = base + j$;
        **para cada** $l = 0, 1, 2, \ldots, n_i - 1$ **faça**
          $z_{in}[l] = \upsilon_{in}[index]$;
          $index = index + nright_i$;
        **fim**
        multiply $z_{out} = z_{in} \times Q^{(i)}$;
        $index = base + j$;
        **para cada** $l = 0, 1, 2, \ldots, n_i - 1$ **faça**
          $\upsilon_{in}[index] = z_{out}[l]$;
          $index = index + nright_i$;
        **fim**
      **fim**
      $base = base + (nright_i \times n_i)$;
    **fim**
  **fim**
  **para cada** $l = 0, 1, 2, \ldots, nright_\sigma - 1$ **faça**
    $\Upsilon[base_{out} + l] = \Upsilon[base_{out} + l] + \upsilon_{in}[l]$;
  **fim**
**fim**

Conforme pode ser observado, ambos Algoritmos 1 e 3.3 são iguais até a linha 9, com a diferença que o último irá calcular os valores de $e$, $base_{in}$ e $base_{out}$ apenas para as matrizes até $\sigma$, em comparação ao primeiro, que os calcula até $N$. Isto se deve ao fato da tabela esparsa estar sendo gerada apenas para a parte esquerda do produto tensorial utilizado na MVD.

Para a parte direita do produto tensorial, pode-se observar as semelhanças entre as linhas de

número 10 a 31 do Algoritmo 3.3 com as linhas de 1 a 19 do Algoritmo 3.2. No caso da solução Shuffle, nota-se o uso de $\upsilon$ em contraste ao uso de $\upsilon_{in}$ na solução Split, uma vez que $\upsilon_{in}$ já se trata da parte pertinente de $\upsilon$ multiplicada por $e$. Outra diferença é o uso de $\sigma$ ao invés de *zero* para os índices referentes ao produto tensorial.

Nas linhas finais do Algoritmo 3.3 (a partir da linha 32), ocorre então o acúmulo de $\Upsilon$ a partir do vetor $\upsilon_{in}$. Ainda é possível notar que o uso de um vetor $\upsilon_{out}$ é desnecessário, uma vez que $\upsilon_{in}$ é utilizado para representar $\Upsilon$ na parte Shuffle da solução Split.

Assim como nas soluções anteriores, o custo desta solução também pode ser teorizado em termos do número de multiplicações em ponto flutuante. Neste caso, o custo é dado pelo número de multiplicações realizadas para gerar os elementos não nulos de cada termo da parte Sparse, somado ao custo de multiplicar o vetor de probabilidades pelo produto tensorial do conjunto de matrizes da parte Shuffle. Este valor, multiplicado pelo número de matrizes unitárias, é apresentado na Equação 3.6 e determina o custo computacional do Split.

$$\prod_{i=1}^{\sigma} nz_i \times \left[ \left( (\sigma - 1) + \prod_{i=\sigma+1}^{N} n_i \right) + \left( \prod_{i=\sigma+1}^{N} n_i \sum_{i=\sigma+1}^{N} \frac{nz_i}{n_i} \right) \right] \tag{3.6}$$

## 3.2 Considerações Finais

Este capítulo apresentou o que é a Multiplicação Vetor-Descritor e quais são as complicações do processo. Também foram apresentados três algoritmos iterativos para a realização da MVD, sendo o Split um deles e proposta de paralelização neste trabalho. O próximo capítulo irá discutir formas de distribuir a MVD e focar esta distribuição no algoritmo Split.

# 4. SOLUÇÃO PROPOSTA

Para a realização de um protótipo capaz de executar uma versão distribuída da solução Split, foram disponibilizados quatro computadores Hewlett-Packard modelo dx2600, cujas características pertinentes são dois processadores Itanium$^2$ (da família IA64) com *clock* de 1.5 GHz, 2 GB de memória RAM e uma interface de rede Ethernet Gigabit que interliga os mesmos em uma rede dedicada. Conforme [HEN02], configurações com estes processadores são indicadas para aplicações matemáticas. Este agregado, classificado pela literatura como Máquinas Agregadas (COW, do inglês *Cluster of Workstations*) de acordo com [DER03] ou ainda como *Multiple Instructions Multiple Data* (MIMD) conforme a classificação por fluxo de instruções de Flynn [FLY66], influencia diretamente sobre como esta solução será elaborada.

Por se tratar de um agregado homogêneo, ou seja, de máquinas iguais interligadas por uma rede chaveada onde o custo de comunicação entre qualquer nodo é o mesmo, uma abordagem paralela como Mestre-Escravo [TAN06] [HWA98] [HWA92] pode ser utilizada. Esta abordagem consiste em designar a tarefa de Mestre para um processo e as tarefas de escravos para os demais. O Mestre fica então encarregado de centralizar as informações do processamento e distribuir trabalhos para os escravos executarem, determinando ainda os critérios de parada.

Partindo destas premissas, um estudo sobre como paralelizar a MVD foi conduzido, buscando-se formas de dividir o processamento realizado pelo algoritmo Split. Em seguida, ainda foi necessário realizar a implementação de uma versão seqüencial da solução para que valores de referência fossem obtidos. Finalmente, um protótipo paralelo foi elaborado a partir da implementação seqüencial, fornecendo novos números para comparação.

## 4.1 Paralelização da MVD

Conforme foi discutido no Capítulo 3, as soluções Sparse, Shuffle e Split são baseadas no Método da Potência, que se trata de um método iterativo onde cada iteração $i$ depende do vetor resultante da iteração $i-1$ para que possa ser executada. Por causa desta dependência, não é possível distribuir diferentes iterações em diferentes processos concorrentes. Assim sendo, para que seja possível realizar uma versão paralela deste método, primeiro deve-se analisar o processo em si para que os pontos passíveis de paralelização sejam identificados.

Analisando a solução Split, cuja paralelização é a proposta deste trabalho, pode-se verificar os pontos da mesma que são passíveis de distribuição onde não haja dependência de dados. Um destes pontos torna-se evidente se o Descritor Markoviano for revisado. Lembrando que uma iteração consiste em tomar um vetor de probabilidades $\upsilon$ e multiplicá-lo por cada produto tensorial do DM, acumulando o resultado de cada um destes processos em $\Upsilon$, nota-se que não há dependência de dados entre estes produtos tensoriais.

A Figura 4.1 ilustra este processo, mostrando que o mesmo $\upsilon$ é multiplicado pelos diferentes

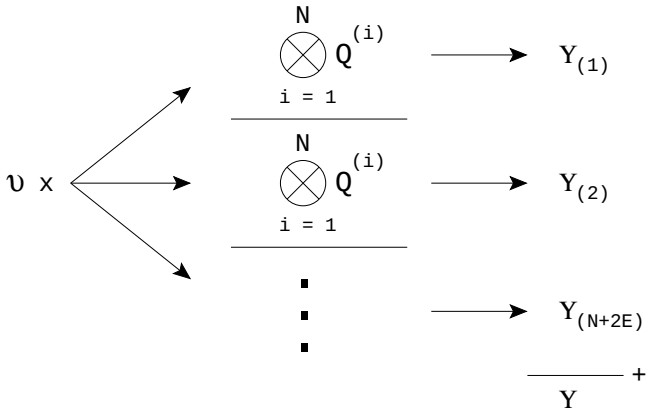


Figura 4.1: Ilustração do processo Multiplicação Vetor-Descritor.

produtos tensoriais existentes no Descritor Markoviano. Esta ilustração também representa a Equação 3.1, apresentada no início do Capítulo 3. No que diz respeito a paralelização, isto também mostra que é possível colocar cada processo de multiplicação de $\upsilon$ por *um* produto tensorial em fluxos de execução concorrentes, visto que não há dependência de dados.

Levando esta idéia para a solução particular Split, entra-se no mérito de como paralelizar o processo de multiplicação de $\upsilon$ por *um* produto tensorial no algoritmo em questão. Conforme foi discutido, o algoritmo consiste em montar uma tabela contendo os elementos escalares $e$ e multiplicá-los pelo produto tensorial a direita de $\sigma$, juntamente com as partes pertinentes de $\upsilon$. A Figura 4.2 ilustra esta idéia.

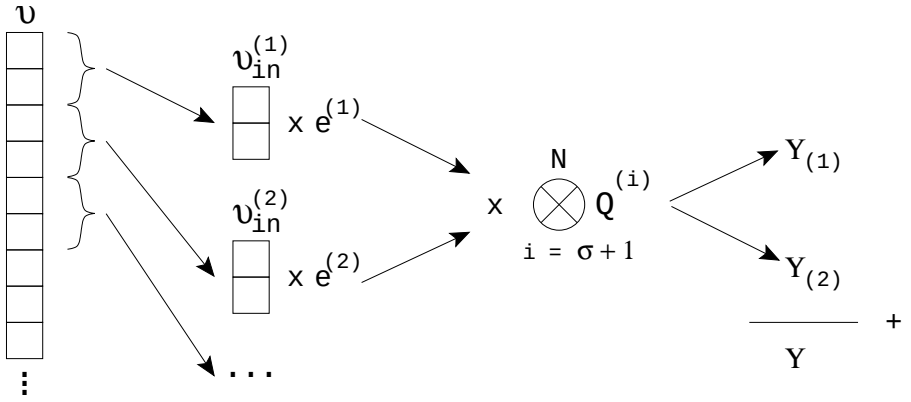


Figura 4.2: Ilustração da multiplicação de $\upsilon$ por um produto tensorial utilizando a solução Split.

A figura mostra vetores $\upsilon_{in}$ formados a partir de segmentos de $\upsilon$ e multiplicados por $e$, conforme a linha 11 do Algoritmo 3.3. Um por vez, estes vetores são então multiplicados pelo produtório tensorial remanescente (que inicia em $\sigma+1$ e vai até $N$), utilizando o algoritmo Shuffle, conforme as linhas de 13 a 31 do mesmo algoritmo. Cada uma destas etapas de multiplicação acumula seus resultados em $\Upsilon$, tendo-se, ao final da operação, o vetor resultante da MVD.

Observando esta figura, pode-se também notar que cada vetor $\upsilon_{in}$ independe dos demais, visto que cada um é formado de diferentes partes de $\upsilon$ e multiplicado individualmente pelo produtó-

rio tensorial. Novamente, tendo-se identificado um ponto onde não haja dependência de dados, encontra-se assim uma forma mais particular da solução Split para distribuir o trabalho realizado pelo algoritmo em diferentes fluxos de execução.

De acordo com os pontos passíveis de paralelização identificados no algoritmo e do agregado de computadores disponíveis para este trabalho, conforme discutido no início deste capítulo, uma possível abordagem de paralelização pode ser feita através da técnica mestre-escravo. Para a solução Split, a aplicação da técnica pode ser dada através da designação a um processo mestre da tarefa de organizar quais pedaços de $\upsilon$ irão para cada escravo. Os escravos recebem esta informação, executam a parte Shuffle da solução e enviam seus vetores resultantes de volta ao mestre, antes do acúmulo dos resultados em $\Upsilon$.

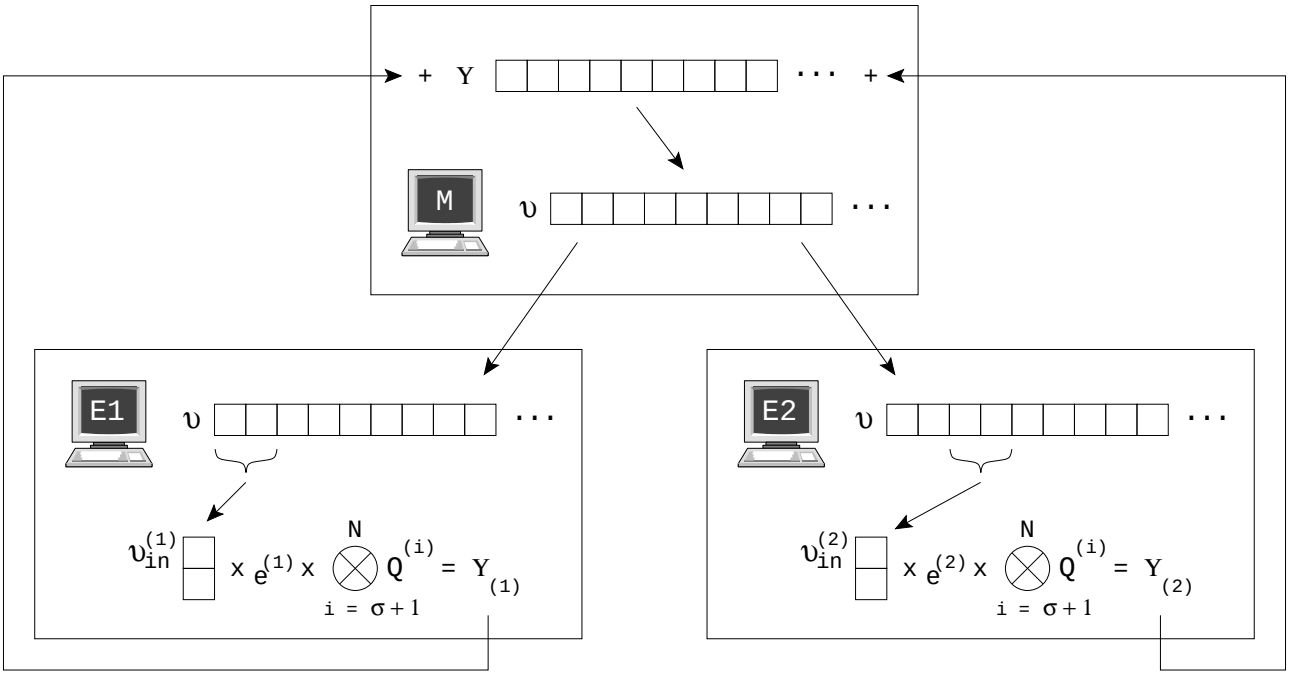


Figura 4.3: Ilustração da distribuição da solução Split utilizando a técnica Mestre-Escravo.

A Figura 4.3 ilustra como esta separação é realizada através da utilização da técnica Mestre-Escravo. Na figura, o processo mestre M envia $\upsilon$ para os escravos E1 e E2. Os escravos, por sua vez, montam os vetores temporários $\upsilon_{in}$ e aplicam o algoritmo Split, multiplicando $\upsilon_{in}$ pelo elemento escalar $e$ pertinente e, em seguida, pelo produtório tensorial através do algoritmo Shuffle. O vetor $\Upsilon$ resultante é então enviado de volta ao mestre, que acumula os resultados e prepara um novo $\upsilon$ para a próxima iteração.

Um ponto interessante desta abordagem é a obtenção dos dados iniciais, visto que o mestre necessita montar em memória toda a estrutura do DM, o ajustando, normalizando e separando em duas partes conforme um $\sigma$ pré-estabelecido para cada produto tensorial. Para a parte da esquerda ainda é necessário calcular a tabela Sparse, contendo todos os elementos escalares $e$ e seus valores de $base_{in}$ e $base_{out}$ associados. Feito isto, o mestre necessita definir o vetor $\upsilon$ inicial, de acordo com a função de atingibilidade do modelo.

Juntando esta observação ao fato de que parte da seção Sparse e toda a parte Shuffle do DM são necessárias por cada escravo, nota-se que é possível reduzir o tempo necessário na inicialização do ambiente. Considerando que a solução paralela é arquitetada para um agregado homogêneo, isto é obtido fazendo-se com que todos os nodos do ambiente realizem os cálculos mencionados simultaneamente no momento em que a execução é iniciada. Desta forma, poupa-se a etapa em que o mestre distribui esta estrutura para todos os escravos, sendo somente necessário que os vetores $\upsilon$ e $\Upsilon$ trafeguem pela rede.

Antes da implementação de um protótipo paralelo capaz de validar o modelo descrito nesta seção, torna-se fundamental a elaboração de um protótipo seqüencial. A solução seqüencial previamente elaborada para as medições apresentadas em [CZE07] não pôde ser utilizada uma vez que, no momento em que este trabalho para uma versão paralela foi iniciado, ela ainda se encontrava em constantes alterações para validações de novas teorias.

## 4.2 Split Seqüencial

Para a elaboração de uma solução seqüencial, inicialmente se estudou como gerar os dados de entrada do modelo SAN a ser resolvido. Tomando a ferramenta PEPS [BEN03] como modelo, que define uma linguagem para descrição de uma SAN e elabora arquivos de dados contendo as matrizes já ajustadas para a formação do DM, decidiu-se utilizar a linguagem de programação C e o DM gerado pelo próprio PEPS. É válido ressaltar que a ferramenta PEPS também apresenta implementações paralelas de seus algoritmos de soluções [BAL04].

Assim sendo, o protótipo elaborado toma como entrada os arquivos de controle do PEPS que descrevem as matrizes ainda não normalizadas do DM. Em seguida, organiza estas matrizes em memória no formato de estruturas produtos tensoriais, já anotando valores como $nright$ e $nleft$, ordem das matrizes, etc. Como próximo passo, normaliza o Descritor Markoviano e, por fim, toma um valor de $\sigma$ e quebra cada produto tensorial em duas partes, gerando a tabela Sparse para a parte da esquerda de cada produto. Na Figura 4.4, este passo está ilustrado como $\alpha$.

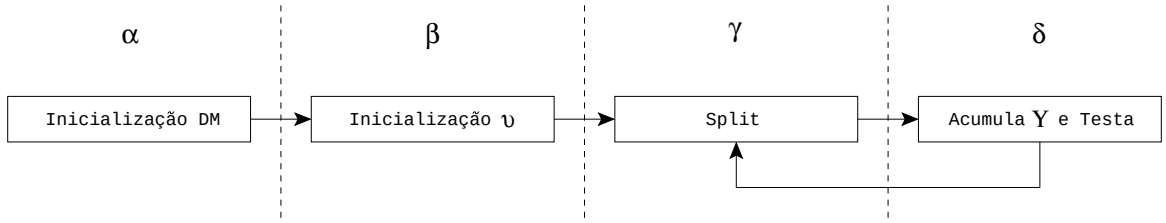


Figura 4.4: Fluxo de execução seqüencial da solução Split.

O segundo quadro da figura ($\beta$) indica a inicialização de $\upsilon$ a partir da função de atingibilidade da SAN. Esta função também é obtida pelos arquivos de dados do PEPS, no entanto foi necessária a implementação de um *parser* para interpretá-la. Este passo é fundamental para a inicialização de qualquer vetor utilizado no Método da Potência uma vez que, para o método atingir convergência, a soma de todos os elementos do vetor de probabilidades inicial necessita ser *um*. Além disto,

outro critério para que exista convergência consiste em iniciar com *zero* todas as posições do vetor referentes aos estados globais não atingíveis da Cadeia de Markov equivalente [STE94].

O terceiro quadro da figura ($\gamma$) trata-se da solução Split em si. Na prática, ele consiste em multiplicar $\upsilon$ por cada linha do DM (um produtório tensorial por vez) utilizando a solução Split. Como estes produtórios já foram quebrado em duas partes na etapa $\alpha$, resta apenas separar os vetores $\upsilon_{in}$ e aplicar a solução Shuffle na parte direita de cada produtório remanescente, acumulando os resultados em $\Upsilon$ ao final das operações.

Já a quarta etapa da figura, ilustrada por $\delta$, representa o acúmulo final dos vetores $\Upsilon$ de cada produto tensorial. Isto forma o vetor resultante do processo $\upsilon \times DM$, caracterizando o final de uma iteração. Por causa disto, une-se a esta etapa o teste referente ao critério de parada da solução, seja ela um número pré-definido de iterações ou a precisão desejada da solução atingida.

Após concluir esta implementação, os resultados foram validados através de comparações com a ferramenta PEPS. Utilizando os modelos propostos em [BAL05] e [DOT05] e executando-os no PEPS por um número fixo de iterações, acompanhou-se os vetores $\Upsilon$ em cada iteração e pode-se comprovar que os valores eram idênticos na precisão proposta aos do protótipo descrito nesta seção.

Para verificar o tempo gasto em cada etapa da Figura 4.4, montou-se um modelo de doze nodos da rede imaginária utilizada como exemplo no Capítulo 2. Com esta quantidade de nodos, o modelo gerado apresentou um espaço de 531.441 ($3^n$) estados globais, sendo 4.097 ($2^n + 1$) deles atingíveis.

A Tabela 4.1 apresenta o tempo gasto em cada etapa do processamento, medidos em milissegundos. Estes dados foram obtidos após a realização de 100 execuções e utilizando-se a média de um intervalo de confiança de 95% para *uma* iteração. A paralelização proposta tem como objetivo reduzir o tempo de $\gamma$, não alterando os demais estágios. Por fim, é interessante notar que o estágio $\delta$, por se tratar de um simples somatório seguido de um conjunto de testes, possui um tempo quase insignificante na execução da solução, dada as diferenças de magnitudes com os demais estágios.

Tabela 4.1: Tempos da solução seqüencial agrupados por estágio.

| Etapa | Tempo (ms) |
|---|---|
| $\alpha$ | 5.463 |
| $\beta$ | 2.097 |
| $\gamma$ | 17.643 |
| $\delta$ | 663 |

## 4.3 Split Paralelo

Para distribuir o processamento conforme estudado na Seção 4.1, foi decidido utilizar a tecnologia Interface de Passagem de Mensagens (MPI, do inglês *Message Passing Interface*), versão 2 [MPI97]. A implementação escolhida foi a realizada pelo Argonne National Laboratory, nos Estados Unidos, e se chama MPICH-2 [MPI07]. Esta decisão foi tomada visto que os pontos passíveis de paralelização identificados no algoritmo necessitavam de um mecanismo confiável para troca de

mensagens entre processos executando em diferentes processadores de um mesmo computador ou de diferentes computadores.

De acordo com os pontos de distribuição identificados no algoritmo, a paralelização se dará na etapa $\gamma$ do fluxo de execução apresentado para o Split seqüencial. Esta etapa deve ser dividida de forma que o mestre envie o vetor $\upsilon$ pronto para os escravos o processarem. Os escravos, por sua vez, recebem $\upsilon$ e montam os vetores $\upsilon_{in}$ pertinentes, multiplicando-os por $e$ e executando a parte Shuffle da solução. Por fim, os escravos enviam o vetor $\Upsilon$ resultantes para que o mestre possa acumulá-los e testar o vetor resultante final. Este esquema está ilustrado na Figura 4.5.

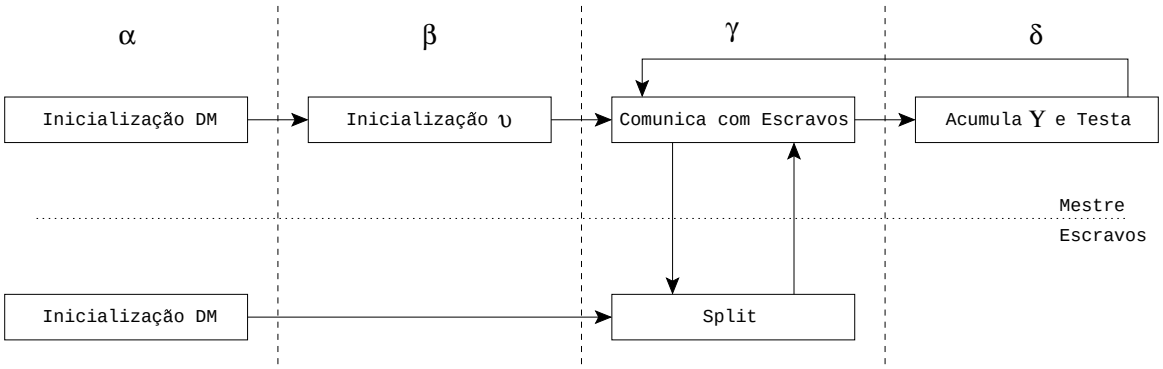


Figura 4.5: Fluxo de execução paralela da solução Split.

Como o número de escravos é pré-fixado, o padrão MPI2 permite que seja calculado, em tempo de execução, o número total de nodos que fazem parte do ambiente. Com isto, cada escravo pode calcular quais linhas da tabela Sparse ele deve utilizar para montar seus vetores $\upsilon_{in}$. Isto é feito pela divisão inteira do número de linhas desta tabela pelo número de escravos, sendo o resto desta divisão assinalado aos primeiros escravos, ou seja, se sobrarem dois elementos da tabela, o primeiro e o segundo escravos do ambiente utilizarão um elemento a mais cada.

É importante salientar que todos os escravos realizam a etapa $\alpha$ do processamento, ou seja, cada processo monta toda a estrutura do Descritor Markoviano em sua memória. Isto não é um problema do ponto de vista do processamento distribuído, uma vez que o agregado é homogêneo e todas as máquinas levam aproximadamente o mesmo tempo para realizar a tarefa. Da mesma forma, como isto precisa ser feito por pelo menos um processo do ambiente, se todos realizarem o cálculo, poupa-se o tempo de comunicar o DM entre os nodos do ambiente.

Por outro lado, o nodo responsável pela disseminação de $\upsilon$ no ambiente é o mestre. Assim sendo, a etapa $\beta$ fica apenas assinalada a ele. Do ponto de vista da abordagem paralela, isto ajuda a reduzir o período de sincronização que existe antes de $\gamma$, pois garante que todos os escravos terão tempo de terminar $\alpha$ antes de começarem a receber o vetor $\upsilon$ para executar o Split.

Olhando mais de perto a etapa $\gamma$, conforme detalhado pela Figura 4.6, observa-se que o envio do vetor $\upsilon$ do mestre para os escravos é feito através da função `MPI_Broadcast()` do MPICH2. Esta função é executada simultaneamente em todos os nodos e recebe como um de seus parâmetros o identificador do nodo mestre. Desta forma, cada processo sabe que o mestre estará enviando

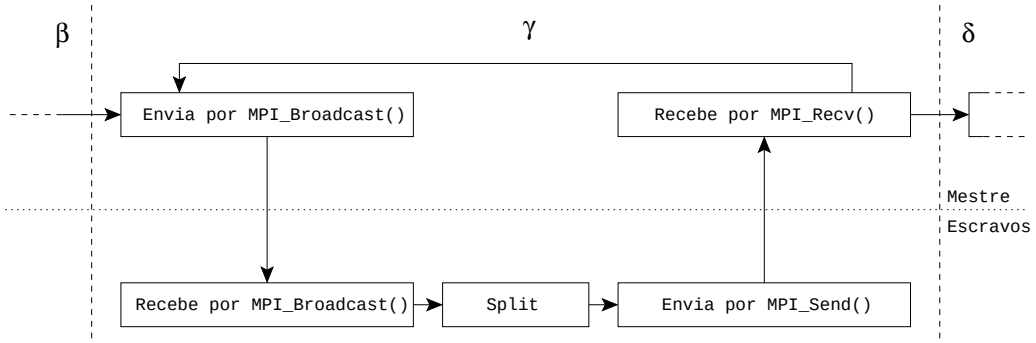


Figura 4.6: Detalhamento da etapa $\gamma$.

informações e os demais nodos estarão recebendo. A devolução do vetor $\Upsilon$, no entanto, necessita ser feita através das instruções `MPI_Send()` e `MPI_Recv()`, uma vez que os escravos estarão enviando dados diferentes diretamente para o mestre.

Outro ponto relevante que pode ser observado nesta figura é o laço que existe no nodo mestre entre o envio de $\upsilon$ e o recebimento de $\Upsilon$. Este laço reflete a execução da solução Split em cada linha do DM. Para termos práticos de medição, é interessante analisar o tempo total de multiplicar $\upsilon$ por todas as linhas do DM, uma vez que elas são diferentes umas das outras, conforme já mostrado na Seção 2.3.3.

Para verificar a melhoria de tempo proposta por esta nova solução, gerou-se uma tabela comparativa com o estágio $\gamma$ da solução seqüencial. A Tabela 4.2 apresenta os tempos da solução seqüencial e da versão paralela descrita nesta seção, variando-se o número de escravos de 1 até 7, visto que o ambiente dispõem de 8 processadores (2 por máquina em um agregado de 4 máquinas).

Tabela 4.2: Tempos comparativos da solução paralela, variando-se o número de escravos.

| **N$^o$ de Nodos** | **Tempo (ms)** | **SpeedUp** |
|---|---|---|
| Seqüencial | 17.643 | 1 |
| 1+1 | 20.766 | 0.8496 |
| 1+2 | 15.740 | 1.1209 |
| 1+3 | 14.156 | 1.2463 |
| 1+4 | 13.360 | 1.3206 |
| 1+5 | 16.093 | 1.0963 |
| 1+6 | 18.560 | 0.9506 |
| 1+7 | 20.594 | 0.8567 |

A tabela ainda apresenta uma terceira coluna, denominada SpeedUp, que se trata de uma medida referente ao ganho existente em relação a um parâmetro de comparação. Neste caso, o parâmetro de comparação é o tempo da execução seqüencial, fazendo com que o SpeedUp desta linha da tabela receba o valor *um*. Para as demais linhas, calcula-se este valor através da divisão do tempo da solução seqüencial pelo tempo em questão. A Figura 4.7 coloca estes valores em forma de gráfico, podendo-se visualizar o ganho ou perda de cada configuração.

Conforme é possível observar, a implementação paralela do estágio $\gamma$ mostrou-se mais lenta que a versão seqüencial, quando utilizadas apenas duas máquinas no processo. Neste caso, uma máquina

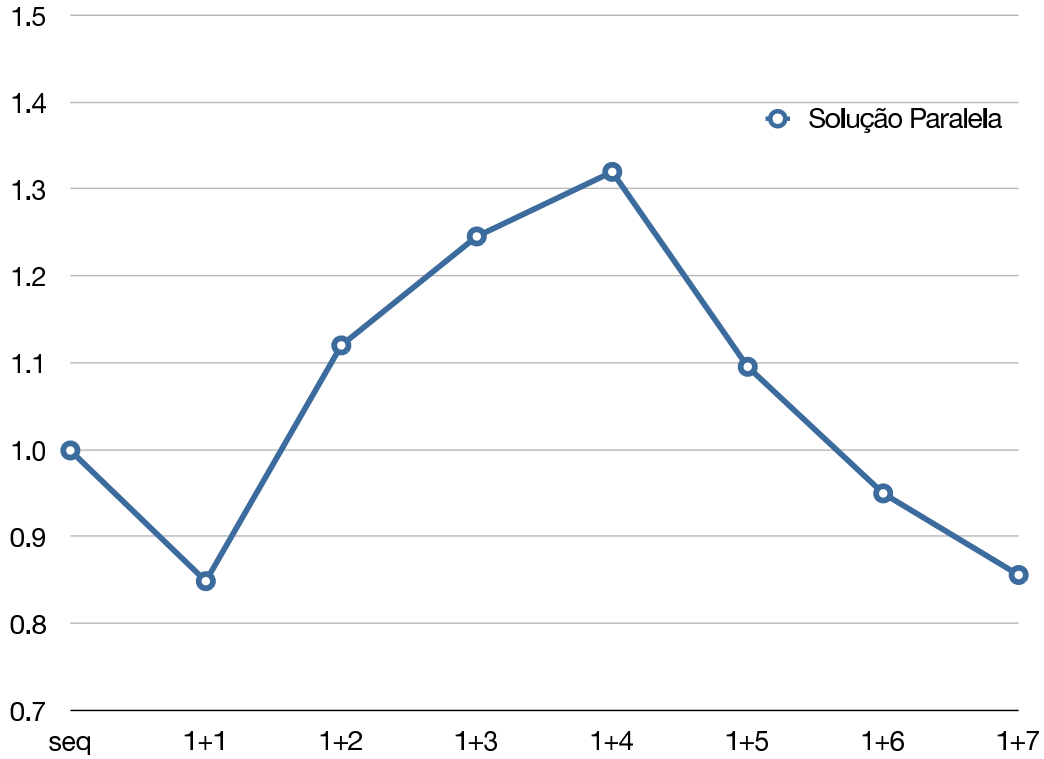


Figura 4.7: Gráfico de SpeedUp para a solução paralela.

hospedou o processo mestre, enquanto que a outra hospedou o processo escravo. No entanto, a próxima configuração, que faz uso de três máquinas, já apresenta um ganho superior a versão seqüencial. Este ganho ocorreu novamente nas configurações seguintes, chegando ao máximo com um mestre e quatro escravos, estando $32.06\%$ mais rápido do que a versão seqüencial.

O ponto em que continuar dividindo o trabalho em mais processos começa a degradar o tempo de execução é conhecido como *trashing*. Neste modelo, ele pode ser explicado pelo fato de coincidir com o momento em que mais de um processo *escravo* é colocado em uma mesma máquina. Isto pode ser justificado pelo fato da comunicação entre os processos ser onerosa e, visto que cada máquina possui dois processadores porém apenas uma placa de rede, colocar mais de um processo em uma máquina pode iniciar a situação de *trashing*.

É interessante notar, no entanto, que não ocorre *trashing* na configuração onde há um mestre e quatro escravos, mesmo havendo dois processos em uma mesma máquina. Isto é explicado pelo fato de toda comunicação ocorrer entre os escravos e o mestre, não havendo caso onde os escravos troquem mensagens entre si. Assim sendo, a configuração $1+4$ coloca o mestre na máquina onde residem dois processos. Desta forma, o MPI é capaz de evitar que a comunicação entre o mestre e o escravo que estão na mesma máquina chegue ao controlador de rede, enviando as mensagens diretamente de um processo ao outro.

Para melhor esclarecer como esta alocação é feita, a Tabela 4.3 apresenta como o MPI foi configurado para designar os processos escravos e o mestre nos processadores disponíveis do ambiente, dependendo do número de nodos alocados. Na tabela, M representa o mestre e $E_i$ representa um escravo de índice $i$. Para cada computador disponível, $IA64_j$ representa uma máquina Itanium[2] de

índice $j$ e, por fim, $P_k$ representa o processador $k$ da máquina em questão.

Tabela 4.3: Distribuição do mestre e escravos, pelo MPI, em relação ao número de nodos alocados em quatro máquinas.

| **nodos alocados** | **IA64$_1$** | | **IA64$_2$** | | **IA64$_3$** | | **IA64$_4$** | |
|---|---|---|---|---|---|---|---|---|
| | **P$_1$** | **P$_2$** | **P$_1$** | **P$_2$** | **P$_1$** | **P$_2$** | **P$_1$** | **P$_2$** |
| **2** (1+1) | M | | E$_1$ | | | | | |
| **3** (1+2) | M | | E$_1$ | | E$_2$ | | | |
| **4** (1+3) | M | | E$_1$ | | E$_2$ | | E$_3$ | |
| **5** (1+4) | M | E$_4$ | E$_1$ | | E$_2$ | | E$_3$ | |
| **6** (1+5) | M | E$_4$ | E$_1$ | E$_5$ | E$_2$ | | E$_3$ | |
| **7** (1+6) | M | E$_4$ | E$_1$ | E$_5$ | E$_2$ | E$_6$ | E$_3$ | |
| **8** (1+7) | M | E$_4$ | E$_1$ | E$_5$ | E$_2$ | E$_6$ | E$_3$ | E$_7$ |

Para confirmar estas observações, as medições de tempo foram refeitas. Nestas novas medições, no entanto, descontou-se o tempo gasto com a comunicação entre o mestre e os escravos. Analisando novamente a Figura 4.6, esta nova medição conta somente o tempo gasto dentro do quadro denominado Split. Os resultados seguem na Tabela 4.4.

Tabela 4.4: Tempos comparativos da solução paralela, descontados os gastos de comunicação.

| **N$^o$ de Nodos** | **Tempo (ms)** | **SpeedUp** |
|---|---|---|
| Seqüencial | 17.643 | 1 |
| 1+1 | 17.826 | 0.9897 |
| 1+2 | 8.918 | 1.9784 |
| 1+3 | 5.947 | 2.9667 |
| 1+4 | 4.456 | 3.9594 |
| 1+5 | 3.686 | 4.7865 |
| 1+6 | 3.097 | 5.6968 |
| 1+7 | 2.632 | 6.7033 |

Os novos dados de SpeedUp obtidos também podem ser colocados em um gráfico. A Figura 4.8 apresenta estes resultados, onde é possível observar a quase linearidade existente. Isto confirma que a comunicação é um obstáculo apresentado pela paralelização deste algoritmo, devido ao tamanho dos vetores que são transmitidos entre os nodos do ambiente. Para tentar reduzir este impacto, um novo modelo foi proposto e é apresentado na próxima seção.

## 4.4 Modelo Paralelo Otimizado

As conclusões da Seção 4.3 mostram que os custos envolvidos na comunicação do ambiente impedem um melhor desempenho da versão paralela. Partindo desta premissa, buscou-se uma forma de reduzir este custo através de uma nova análise da versão distribuída proposta.

Estudando novamente a Figura 4.3, é possível observar que não é estritamente necessário enviar todo o vetor $v$ para cada escravo. Visto que os escravos utilizam apenas partes deste vetor para a geração dos seus $v_{in}$, o mestre é capaz de identificar quais as partes de $v$ são necessárias por cada escravo e fazer este envio separadamente. A Figura 4.9 ilustra esta nova idéia, estando hachuradas as partes de $v$ que não foram recebidas nos escravos e que por eles não serão utilizadas.

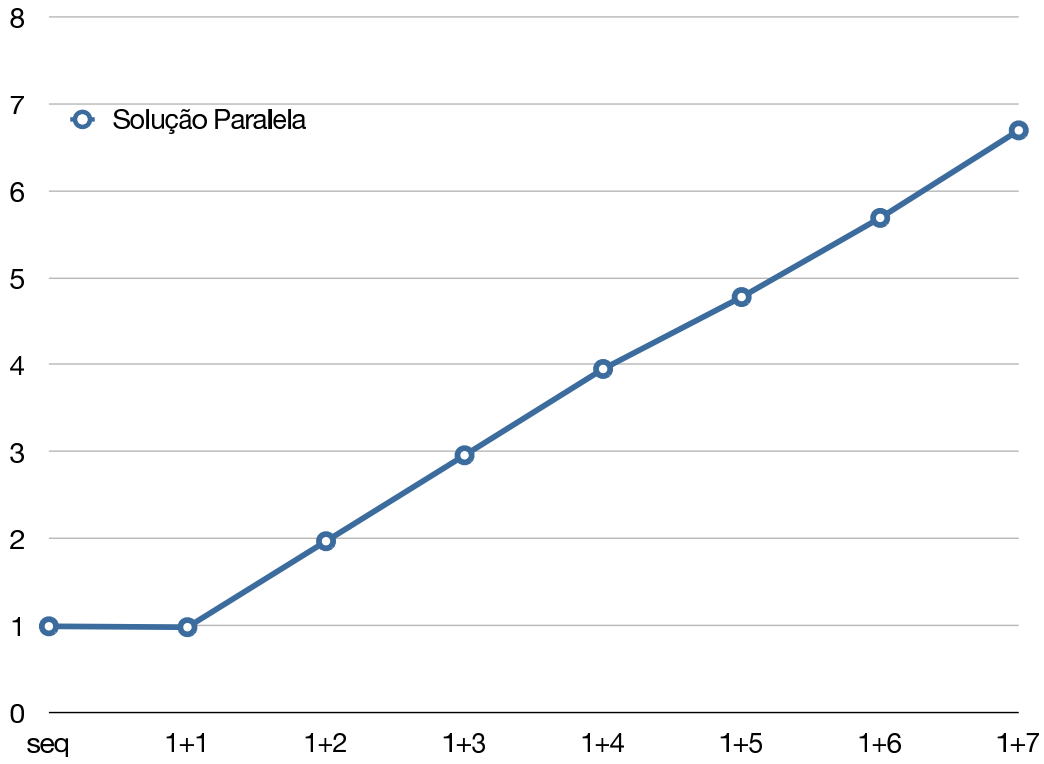


Figura 4.8: Gráfico de SpeedUp para a solução paralela, descontados os gastos de comunicação.

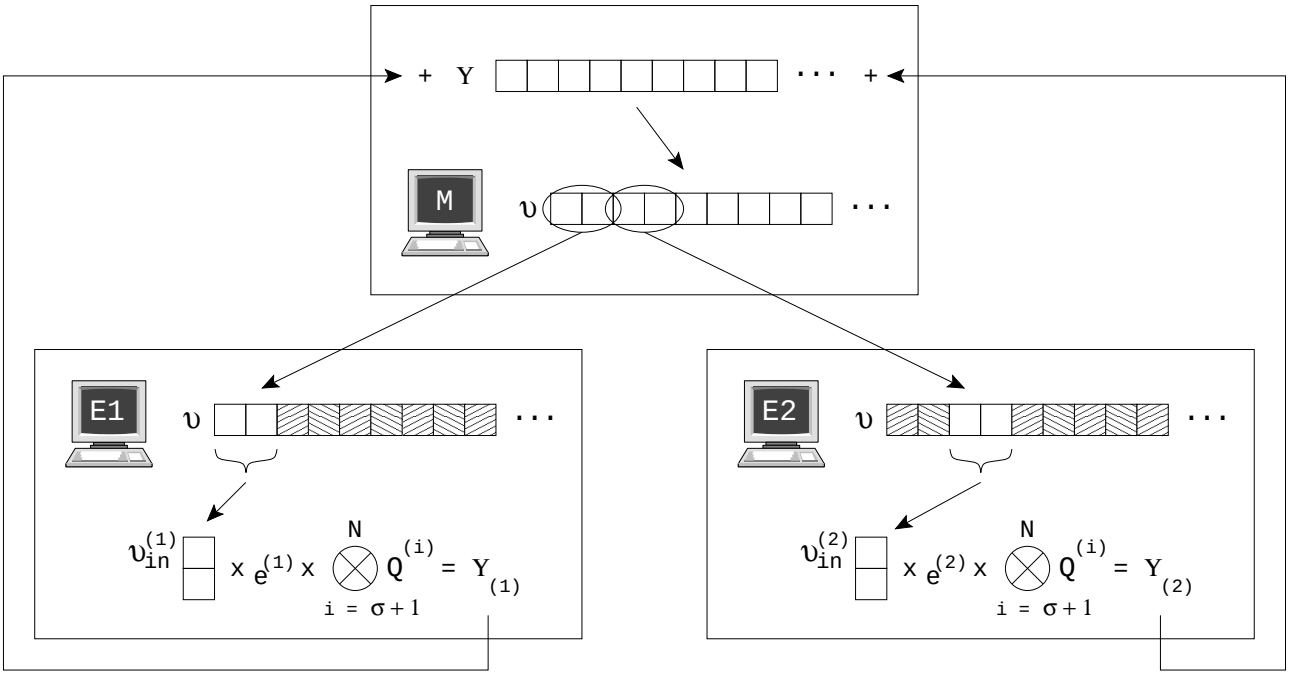


Figura 4.9: Ilustração da distribuição da solução Split utilizando a técnica Mestre-Escravo, sem o envio total de $\upsilon$.

Para implementar este novo conceito, no entanto, é necessário alterar a forma como o vetor $\upsilon$ é enviado pelo MPI. Visto que a função `MPI_Broadcast()` envia um conjunto de dados para todos os escravos, torna-se necessário trocá-la pela função `MPI_Send()`. A Figura 4.10 ilustra o detalhamento da etapa $\gamma$ nesta nova versão otimizada.

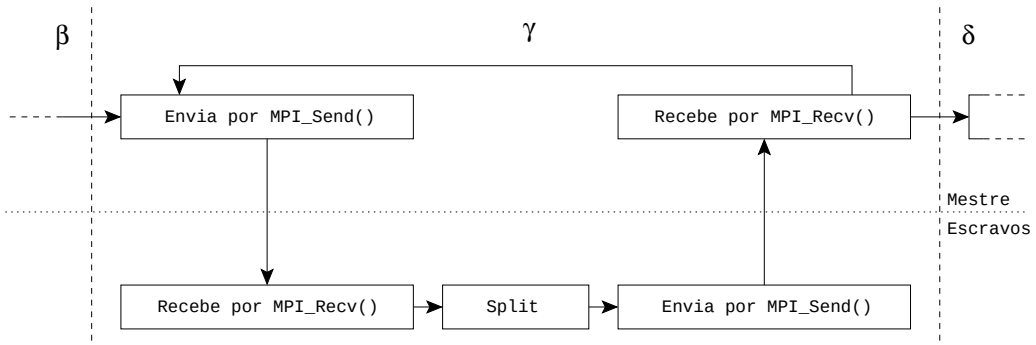


Figura 4.10: Detalhamento da etapa $\gamma$ na versão otimizada.

Este novo conceito também pode ser observado na linha 11 do Algoritmo 3.3, que mostra como $\upsilon_{in}$ é montado a partir de $\upsilon$ e $base_{in}$. Esta linha ainda mostra que os pedaços de $\upsilon$ necessários são contíguos, visto que variam do $base_{in}$ de cada elemento da tabela Sparse até $base_{in} + nright_{\sigma} - 1$.

Concentrando-se nesta informação, notou-se que era possível reordenar a tabela Sparse, antes de dividí-la entre os escravos, de acordo com os valores de $base_{in}$. Desta forma, é possível reduzir a parte de $\upsilon$ enviada para cada escravo. Para ilustrar como isto afeta a comunicação, a Figura 4.11 apresenta dois cenários. O primeiro mostra uma situação onde a tabela Sparse não está ordenada, fazendo com que partes de $\upsilon$ sejam enviadas em demasia pela rede. O segundo cenário mostra como a ordenação da tabela pode reduzir estas comunicações.

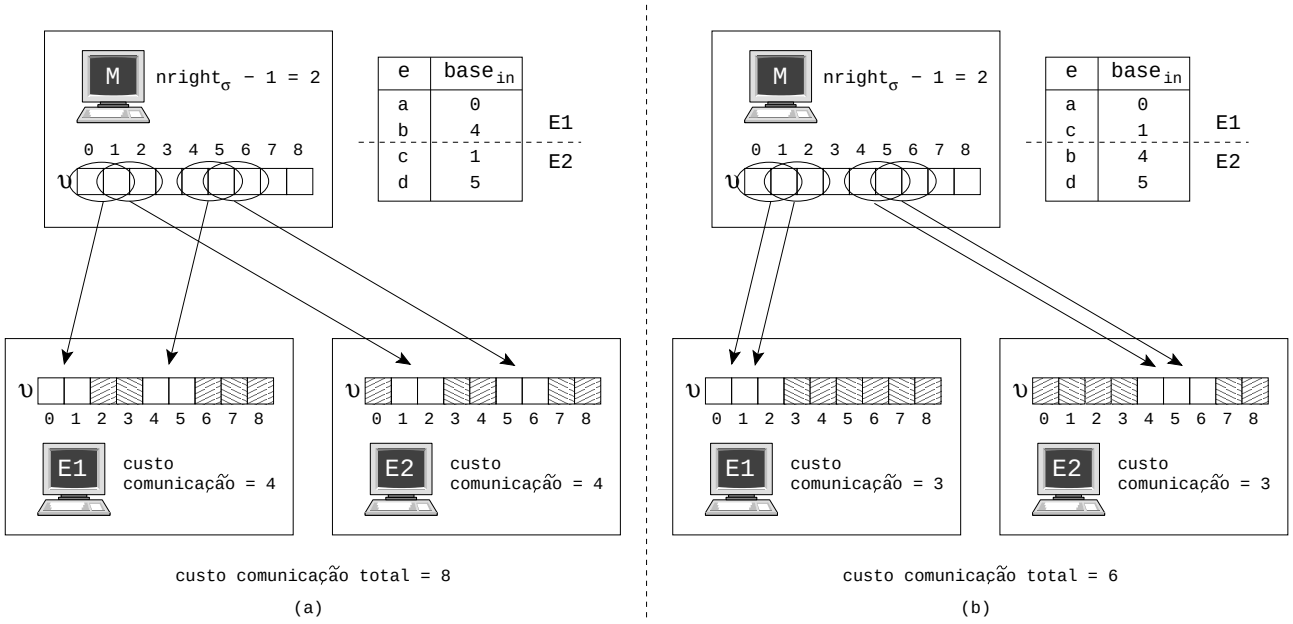


Figura 4.11: (a) Cenário onde a tabela Sparse não é ordenada. (b) Cenário onde a tabela Sparse é ordenada.

Conforme visto no exemplo proposto na Figura 4.11, a ordenação por $base_{in}$ da tabela Sparse pode reduzir ainda mais o custo de comunicação envolvido no processo. Assim sendo, realizou-se novamente as medições de tempo da etapa $\delta$, buscando-se avaliar se houve ou não uma melhoria em relação ao primeiro modelo paralelo proposto. Os resultados destas medições comparados aos tempos obtidos previamente estão colocados na Tabela 4.5.

Tabela 4.5: Tempos comparativos da solução paralela otimizada.

| Nº de Nodos | Solução Original | | Solução Otimizada | |
|---|---|---|---|---|
| | Tempo (ms) | SpeedUp | Tempo (ms) | SpeedUp |
| Seqüencial | 17.643 | 1 | 17.643 | 1 |
| 1+1 | 20.766 | 0.8496 | 19.770 | 0.8924 |
| 1+2 | 15.740 | 1.1209 | 13.151 | 1.3416 |
| 1+3 | 14.156 | 1.2463 | 11.732 | 1.5038 |
| 1+4 | 13.360 | 1.3206 | 11.302 | 1.5611 |
| 1+5 | 16.093 | 1.0963 | 12.841 | 1.3740 |
| 1+6 | 18.560 | 0.9506 | 14.215 | 1.2412 |
| 1+7 | 20.594 | 0.8567 | 16.349 | 1.0791 |

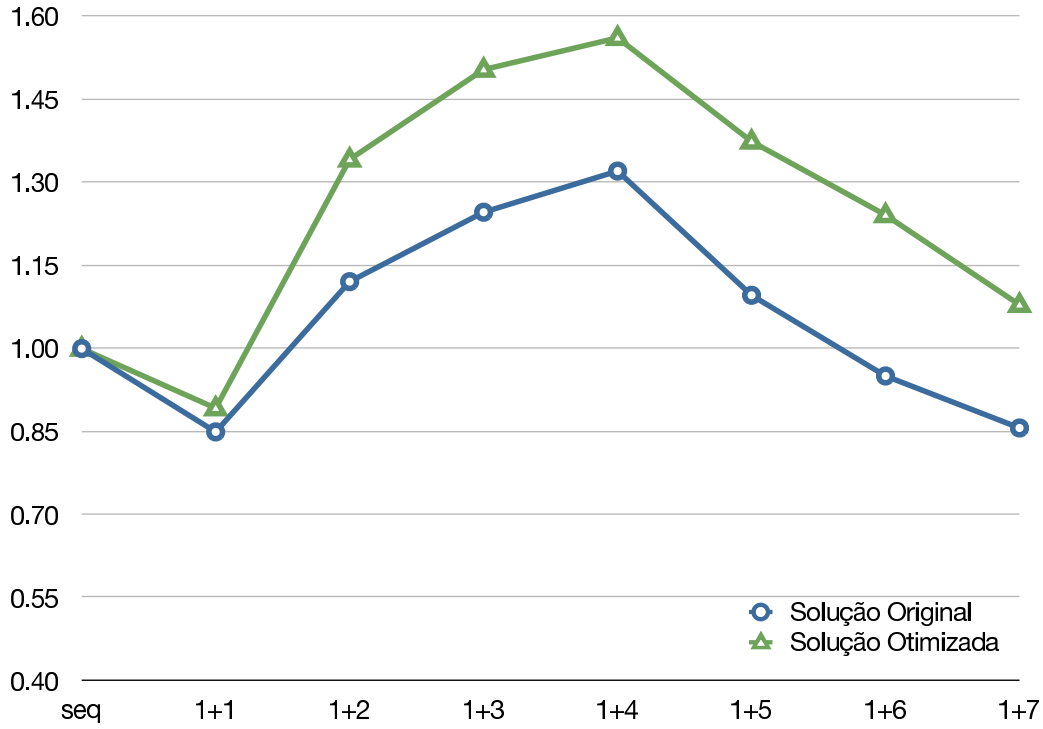


Figura 4.12: Gráfico de SpeedUp para a solução paralela otimizada.

A Figura 4.12 mostra, através de um gráfico, a comparação entre o modelo paralelo original utilizando `MPI_Broadcast()` e a solução otimizada que envia apenas as partes pertinentes de $\upsilon$. Conforme pode ser observado, o novo modelo atingiu, para a configuração $1+4$, um ganho de $56,11\%$ em relação a solução seqüencial, ou seja, $24,05\%$ a mais do que a solução paralela original com o mesmo número de máquinas.

As medições apresentadas neste capítulo foram realizadas com o intuito de orientar quais pontos da primeira solução deveriam ser melhorados. De posse das duas soluções elaboradas, foi possível

realizar outras comparações, observando-se como é o comportamento do sistema pela variação de, por exemplo, o número de máquinas ou a distribuição de processos nas mesmas pelo MPI.

## 4.5 Análise Comparativa

Com o intuito de aprofundar os resultados obtidos sobre o protótipo desenvolvido, as medições realizadas na Seção 4.4 foram refeitas, porém variando-se o número de máquinas envolvidas no ambiente e o balanceamento de processos feito pelo MPI dentro do agregado. Vale ressaltar que, uma vez observado que o protótipo otimizado utilizando as instruções `MPI_Send()` e `MPI_Recv()` apresentou um ganho superior a $24\%$ no melhor caso em relação ao protótipo original, que faz uso da instrução `MPI_Broadcast()`, as medições realizadas nesta seção serão feitas com a implementação mais nova.

Em um primeiro momento, as medições do modelo escolhido (rede hipotética com 12 nodos) foram refeitas utilizando-se apenas uma máquina. Desta forma, calculou-se os tempos e os respectivos SpeedUps de cada configuração, sempre fazendo uso de um processo mestre e variando-se o número de escravos até que o desempenho da configuração ficasse inferior ao da implementação seqüencial. Os resultados obtidos estão colocados na Tabela 4.6.

Tabela 4.6: Tempos de execução variando-se o número de processos em uma máquina.

| **N$^o$ de Processos** | **Tempo (ms)** | **SpeedUp** |
|---|---|---|
| **Seqüencial** | 17.643 | 1 |
| **1+1** | 18.074 | 0.9761 |
| **1+2** | 11.662 | 1.5129 |
| **1+3** | 11.657 | 1.5135 |
| **1+4** | 11.764 | 1.4997 |
| **1+5** | 12.425 | 1.4200 |
| **1+6** | 13.308 | 1.3257 |
| **1+7** | 14.264 | 1.2369 |
| **1+8** | 15.488 | 1.1391 |
| **1+9** | 16.566 | 1.0650 |
| **1+10** | 17.633 | 1.0006 |
| **1+11** | 18.785 | 0.9392 |

De acordo com a tabela, observa-se que o uso de um mestre e três processos escravos proporcionou o melhor SpeedUp possível, com um ganho de $51.35\%$ sobre a versão seqüencial. O fato sugere que, apesar da máquina utilizada ter apenas dois processadores, o uso de $1+3$ processos apresentou o melhor desempenho. Este resultado reforça que a comunicação entre o mestre e os escravos é significativa, uma vez que os processos não conseguiram tomar o processador completamente (fez-se uso de quatro processos em dois processadores).

A Figura 4.13 ilustra o gráfico dessas medições, dando forma aos SpeedUps obtidos. Nela, fica claro que o uso de dois ou três escravos proporciona o melhor ganho possível para o cenário de uma máquina. O uso de mais do que três escravos começa a deteriorar o tempo de execução, causando uma queda na linha do SpeedUp.

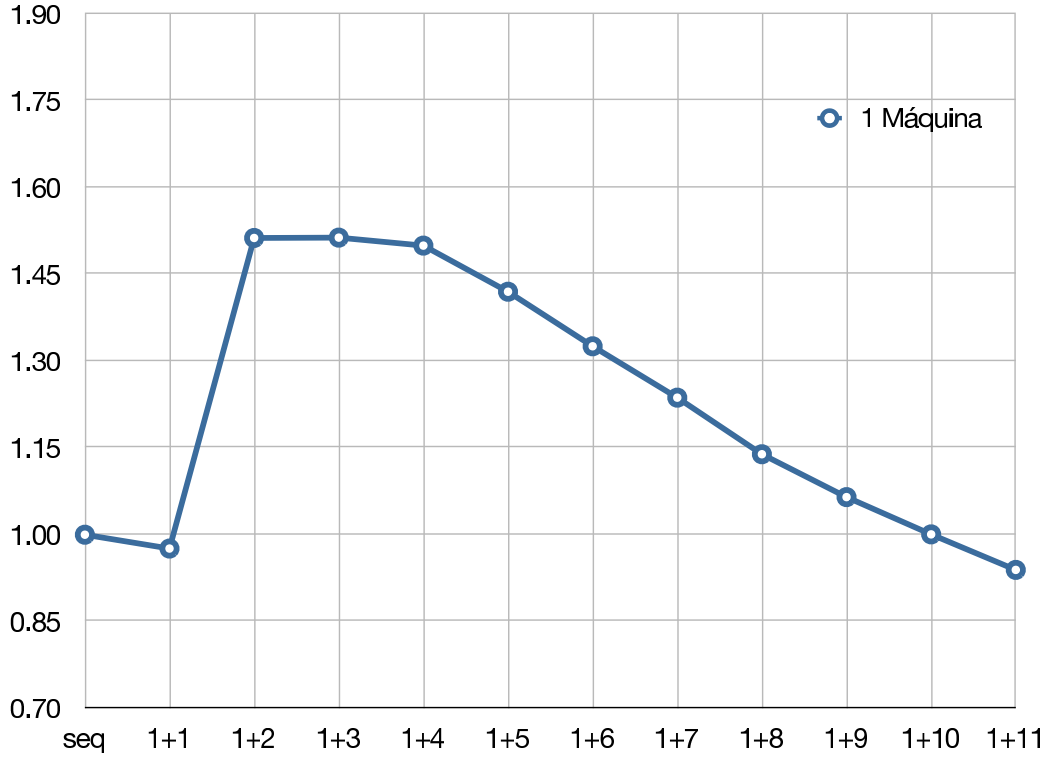


Figura 4.13: Gráfico de SpeedUp para execuções utilizando-se uma máquina.

Para o uso de duas ou mais máquinas, diferentes combinações são possíveis para a distribuição dos processos. Para estudar como estas variações afetam os tempos de execução, inicialmente foi realizada uma distribuição como a da seção anterior. Esta distribuição, que é o padrão para a implementação do MPI utilizado (MPICH2), procura ocupar todos os processadores disponíveis de forma equilibrada. Para o caso analisado a seguir, com duas máquinas, a Tabela 4.7 mostra como a distribuição é feita.

Tabela 4.7: Distribuição do mestre e escravos, em relação ao número de nodos alocados em duas máquinas.

| **nodos alocados** | **IA64$_1$** | | **IA64$_2$** | |
|---|---|---|---|---|
| | **P$_1$** | **P$_2$** | **P$_1$** | **P$_2$** |
| **2** (1+1) | M | | E$_1$ | |
| **3** (1+2) | M | E$_2$ | E$_1$ | |
| **4** (1+3) | M | E$_2$ | E$_1$ | E$_3$ |
| **5** (1+4) | M | E$_2$ E$_4$ | E$_1$ | E$_3$ |
| **6** (1+5) | M | E$_2$ E$_4$ | E$_1$ | E$_3$ E$_5$ |
| ... | ... | ... | ... | ... |

Utilizando a distribuição de processos proposta, a Tabela 4.8 apresenta os resultados das medições realizadas. É interessante observar que tanto o cenário $1+1$ quanto o cenário $1+2$ apresentam SpeedUps inferiores aos da situação onde apenas uma máquina foi utilizada. Novamente, isto é explicado pelo custo de comunicação envolvido no processo. Para as demais distribuições, a nova

Tabela 4.8: Tempos de execução variando-se o número de processos em duas máquinas.

| Nº de Processos | Tempo (ms) | SpeedUp |
|---|---|---|
| **Seqüencial** | 17.643 | 1 |
| **1+1** | 19.771 | 0.8924 |
| **1+2** | 13.117 | 1.3450 |
| **1+3** | 11.166 | 1.5801 |
| **1+4** | 10.389 | 1.6982 |
| **1+5** | 12.363 | 1.4271 |
| **1+6** | 13.060 | 1.3509 |
| **1+7** | 15.111 | 1.1676 |
| **1+8** | 16.344 | 1.0795 |
| **1+9** | 18.526 | 0.9524 |
| **1+10** | 19.793 | 0.8914 |
| **1+11** | 22.490 | 0.7845 |

configuração proporcionou uma melhoria de desempenho, começando novamente a ficar mais lenta no cenário $1+6$.

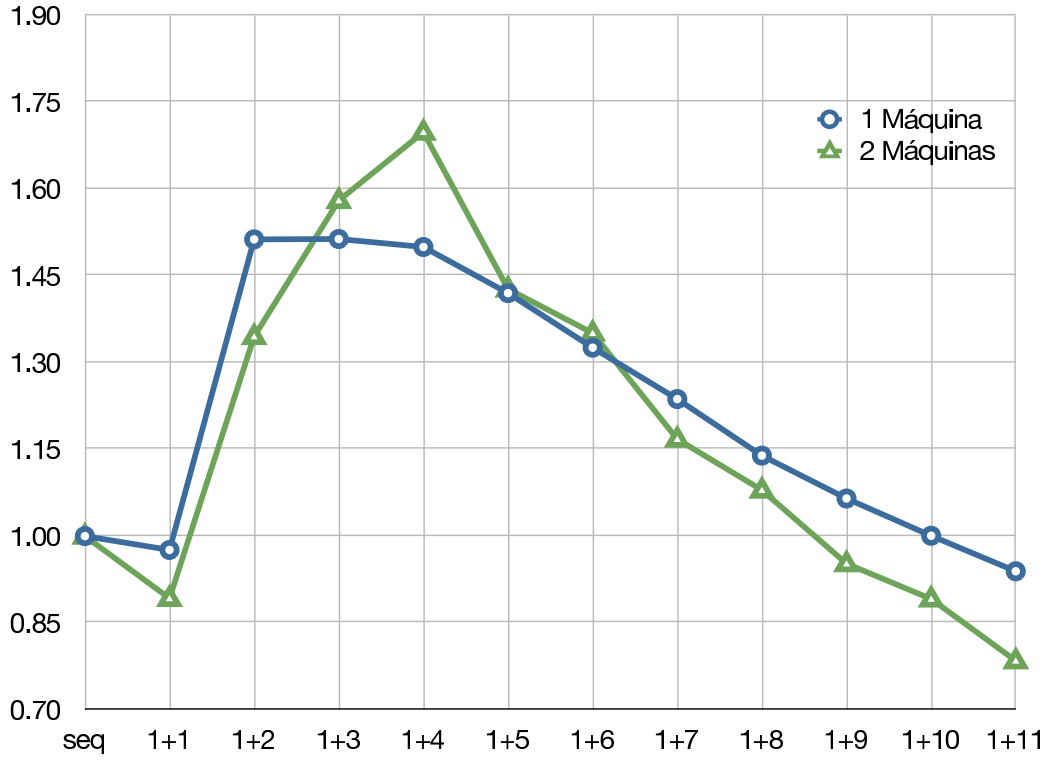


Figura 4.14: Gráfico comparativo de SpeedUp para execuções utilizando-se uma e duas máquinas.

Na Figura 4.14, os valores medidos são ilustrados através de um gráfico. Para fins de comparação, este gráfico foi traçado sobre o o cenário onde apenas uma máquina estava sendo utilizada. Como resultado, pode-se observar os pontos em que a distribuição de processos feita pelo MPI não favorece o uso de dois computadores.

Da mesma forma que para uma e duas, as medições foram realizadas para três máquinas e os resultados estão expressos na Tabela 4.9. Juntando estes resultados com os SpeedUps obtidos para

quatro máquinas na Seção 4.4, a Figura 4.15 apresenta o gráfico sobrepondo as medições.

Tabela 4.9: Tempos de execução variando-se o número de processos em três máquinas.

| Nº de Processos | Tempo (ms) | SpeedUp |
|---|---|---|
| **Seqüencial** | 17.643 | 1 |
| **1+1** | 19.771 | 0.8924 |
| **1+2** | 13.168 | 1.3398 |
| **1+3** | 10.889 | 1.6203 |
| **1+4** | 11.196 | 1.5758 |
| **1+5** | 12.745 | 1.3843 |
| **1+6** | 13.165 | 1.3401 |
| **1+7** | 15.444 | 1.1424 |
| **1+8** | 17.609 | 1.0019 |
| **1+9** | 18.767 | 0.9401 |
| **1+10** | 21.257 | 0.8300 |
| **1+11** | 23.998 | 0.7352 |

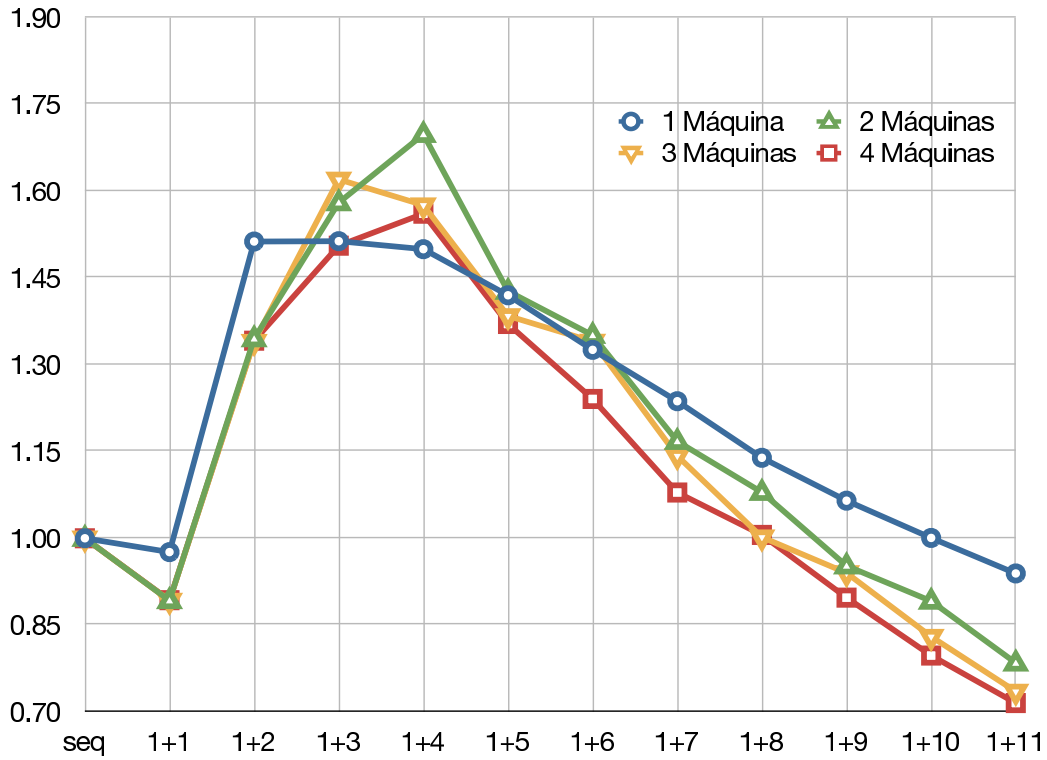


Figura 4.15: Gráfico comparativo de SpeedUp para execuções utilizando-se uma, duas, três e quatro máquinas.

Sobre a Figura 4.15, que engloba os resultados obtidos para todas as configurações possíveis de distribuição de processos com até quatro máquinas, nota-se que o cenário que apresenta o maior SpeedUp foi o $1+4$ utilizando-se duas máquinas. Conforme já foi discutido, as configurações com mais máquinas não conseguiram prover um SpeedUp tão bom por causa do custo de comunicação entre processos em máquinas diferentes.

Por causa disto, o uso de mais máquinas no agregado não garante que o SpeedUp da solução seja melhor. Assim sendo, buscou-se uma forma não convencional de distribuição dos processos para

que um cenário com mais máquinas seja, no pior dos casos, igual ao cenário com uma máquina a menos.

## 4.6 Distribuição Alternativa de Processos

Para atingir o objetivo de distribuir os processos no ambiente de forma a proporcionar o melhor SpeedUp possível, a solução encontrada foi seguir o caminho oposto ao proposto pela configuração padrão do MPI. Conforme discutido, o MPI distribui os processos de forma a nenhuma máquina ter mais processos do que outra quando não for necessário, ou seja, quando o número de processos não for divisível pelo número de máquinas. Nesta nova proposta, busca-se aumentar o número de processos escravos junto ao mestre para reduzir os custos de comunicação.

A forma encontrada de realizar esta busca consiste em criar um vetor $v_{maq}$, do tamanho do número de máquinas disponíveis no ambiente, e preencher com *zero* todas as suas posições. Cada posição deste vetor indica o número de processos escravos designados para cada máquina, considerando-se que o mestre roda na primeira máquina do ambiente, ou seja, os escravos contados em $v_{maq}[0]$ estarão junto ao mestre.

Em seguida, incrementa-se a primeira posição deste vetor e mede-se o SpeedUp da configuração por ele proposta, ou seja, o simples cenário $1+1$ onde os processos mestre e escravo encontram-se na mesma máquina. Em seguida, decrementa-se a primeira posição do vetor e incrementa-se a segunda, realizando a medição novamente. Este processo se repete até o fim do vetor, comparando-se assim os SpeedUps e obtendo-se a melhor configuração possível para o caso $1+1$.

A partir da melhor configuração encontrada, o procedimento é repetido, incrementando-se o número de processos utilizados no ambiente. No momento em que não se encontre uma configuração com SpeedUp superior ao último encontrado, a busca é encerrada. Estas idéias são expressadas em forma algorítmica no Algoritmo 4.1.

Sobre o algoritmo, vale ressaltar que as linhas 3, 11 e 20 representam a cópia de todas as posições de um vetor para outro, e não simplesmente o uso de ponteiros ou referências. Da mesma forma, observa-se que o laço iniciado na linha 4 usa a condição *sempre*, indicando que o laço continuará acontecendo até que a instrução da linha 18 seja executada.

Por causa do formato da curva do SpeedUp, que segue um padrão crescente, atinge um pico e depois decresce, o algoritmo encontra sempre uma distribuição melhor, ou pelo menos igual, do que a distribuição padrão feita pelo MPI. Isto pode ser afirmado pela forma na qual ele testa as configurações em questão, partindo sempre do SpeedUp mais alto encontrado até o momento. Observando novamente a Figura 4.15, o algoritmo iria, por exemplo, manter-se sempre, no mínimo, na curva formada ao topo do gráfico.

Para o problema da rede hipotética utilizada neste trabalho, cada iteração do algoritmo encontraria como melhor combinação para $v_{maq}$ as seguintes configurações:

Algoritmo 4.1: Representação algorítmica da busca pela distribuição ideal de processos

```
v_maq = 0;
melhor = 0;
melhorV = v_maq;
enquanto sempre faça
    melhorou = 0;
    para cada i = 0,1,2,...,M−1 faça
        v_maq[i] = v_maq[i] + 1;
        SpeedUp = Executa iteração com distribuição v_maq;
        se SpeedUp > melhor então
            melhor = SpeedUp;
            melhorV = v_maq;
            melhorou = 1;
        fim
        v_maq[i] = v_maq[i] − 1;
    fim
    se melhorou == 0 então
        Imprime melhorV;
        Encerra;
    fim
    v_maq = melhorV;
fim
```

| | | | |
|---|---|---|---|
| Iteração 1: | $v_{maq} = \{ 1, 0, 0, 0 \}$ | Tempo = 18.074 ms | SpeedUp = 0.9761 |
| Iteração 2: | $v_{maq} = \{ 1, 1, 0, 0 \}$ | Tempo = 11.672 ms | SpeedUp = 1.5116 |
| Iteração 3: | $v_{maq} = \{ 2, 1, 0, 0 \}$ | Tempo = 9.724 ms | SpeedUp = 1.8144 |
| Iteração 4: | $v_{maq} = \{ 2, 1, 1, 0 \}$ | Tempo = 10.375 ms | SpeedUp = 1.7005 |

Ao atingir a quarta iteração, o algoritmo detecta que o SpeedUp obtido foi menor do que o conseguido na terceira iteração, selecionando assim o vetor $\{ 2, 1, 0, 0 \}$ e encerrando sua execução. Traçando os SpeedUps obtidos pelo algoritmo a cada iteração, inclusive a quarta, sobre o gráfico comparativo da Figura 4.15, elabora-se a Figura 4.16.

No gráfico ilustrado na figura, pode-se comprovar visualmente o ganho oferecido pela distribuição otimizada dos processos utilizando-se um mestre e três escravos. De acordo com o vetor $v_{maq}$ obtido, esta distribuição foi configurada com dois escravos na mesma máquina que o mestre e um terceiro escravo em uma segunda máquina, não se fazendo uso de todo o agregado para este problema.

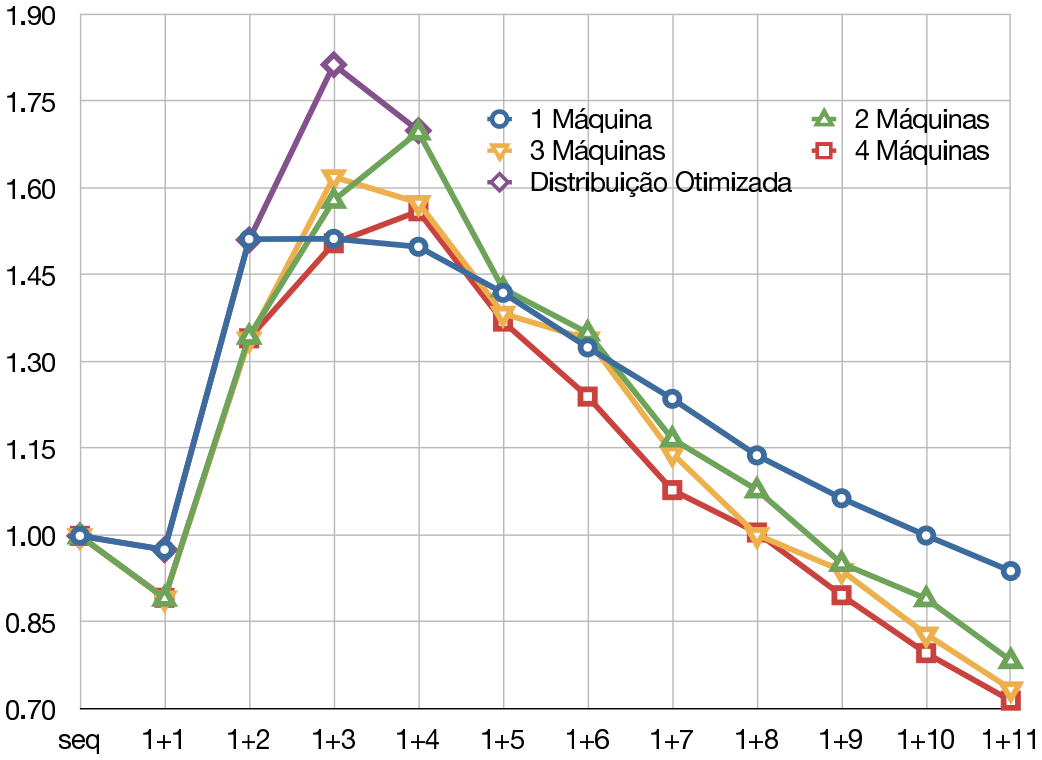


Figura 4.16: Gráfico comparativo de SpeedUp incluindo a distribuição otimizada.

# 5. CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou os conceitos básicos sobre Cadeia de Markov e Rede de Autômatos Estocásticos. Para poder discutir a representação destes formalismos, o Formato Tensorial também foi apresentado, mostrando-se exemplos de Produtos Tensoriais, Fatores Normais e Somas Tensoriais. Com estas bases teóricas, a forma como o Descritor Markoviano é representado numericamente foi então discutida.

O Capítulo 3 mostrou como a multiplicação de um vetor de probabilidades pelo Descritor Markoviano é feita, apresentando ainda três soluções iterativas para realizá-la. Em seguida, o Capítulo 4 inicia a discussão sobre as formas de paralelizar a MVD, focando em particular na questão da solução Split. Ainda neste capítulo, mostrou-se a criação de um protótipo e resultados quanto ao tempo de execução desta implementação.

Dando seqüência a proposta do trabalho, a implementação foi então modifica para incorporar instruções MPI capaz de torná-la paralela. Novas medições foram então realizadas, mostrando que o custo da comunicação sobre o modelo implementado era considerável se comparado ao processamento requerido pelo algoritmo. Por causa disto, uma implementação alternativa foi proposta e prototipada, otimizando a comunicação entre os processos. As medições sobre estas otimizações comprovaram o ganho sobre a primeira versão paralela.

Por fim, uma análise comparativa foi realizada e, destas medições, descobriu-se que a distribuição padrão de processos do MPI entre as máquinas do agregado impede a versão paralela de atingir seu total potencial. Por causa disto, um algoritmo capaz de encontrar uma melhor distribuição foi elaborado. Utilizando-o mostrou-se, para um problema escolhido como exemplo, que o uso de quatro processos distribuídos em duas máquinas consegue atingir um ganho de $81,44\%$ sobre a versão seqüencial, não se encontrando uma outra distribuição capaz de melhorar este resultado.

Concluído este trabalho sobre a distribuição da solução Split, pode-se pensar em trabalhos futuros que ofereceriam continuidade a pesquisa. Entre eles, estariam:

**testes sobre diferentes modelos:** conforme visto nas especificações do DM, cada modelo SAN possui tamanhos de matrizes distintos e, dependendo da quantidade de eventos envolvidos em cada autômato, um número variável de elementos diferentes de zero nestas matrizes. Estes fatores influenciam diretamente sobre ao menos dois pontos: o espaço de estados do DM (pelo tamanho das matrizes) e o trabalho necessário para realizar a MVD (pelo número de elementos diferentes de zero). Por causa disto, um estudo mais aprofundado sobre como a variação destes fatores influenciaria implementações paralelas do Split torna-se uma alternativa interessante de continuidade.

**alterações no grão da solução mestre-escravo:** uma das vantagens de se utilizar uma abordagem paralela baseada em mestres e escravos trata-se da possibilidade de variação da quantidade de trabalho, ou grão, designado pelo mestre para cada escravo. Visto que a divisão da tabela

esparsa proposta neste trabalho não precisa ser em tamanhos fixos, o tamanho do grão pode ser diferente para cada escravo do ambiente. Assim sendo, mesmo durante a execução da implementação paralela, o processo mestre é capaz de observar se algum escravo está demorando mais do que outros e balancear a carga entre eles, seja por causa da quantidade de trabalho envolvida, pela capacidade de processamento do agregado ou mesmo pelo tempo de comunicação gasto no processo.

**uso de outras técnicas de paralelização:** diferentes técnicas de paralelização oferecem diferentes benefícios em relação aos ganhos da versão paralela sobre uma versão seqüencial. Apesar deste trabalho ter sido elaborado com o conceito mestre-escravo, permitindo variações no tamanho dos grãos e na mobilidade de processos no ambiente, técnicas como *fases paralelas* [DER03] permitem que cada nodo do ambiente, por exemplo, reserve memória para apenas parte do problema, resolvendo outras complicações impostas por modelos maiores.

# Bibliografia


[AJM84] M. Ajmone-Marsan, G. Conte e G. Balbo. "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems", ACM Transactions on Computer Systems, vol. 2-2, Maio 1984, pp. 93–122.

[BAL04] L. Baldo, L. G. Fernandes, P. Roisenberg, P. Velho e T. Webber. "Parallel PEPS Tool Performance Analysis using Stochastic Automata Networks", Lecture Notes in Computer Science, vol. 3149, Agosto/Setembro 2004, pp. 214–219.

[BAL05] L. Baldo, L. Brenner, L. G. Fernandes, P. Fernandes e A. Sales. "Performance Models for Master/Slave Parallel Programs", Electronic Notes In Theoretical Computer Science, vol. 128-4, Abril 2005, pp. 101–121.

[BEN03] A. Benoit, L. Brenner, P. Fernandes, B. Plateau e W. J. Stewart. "The PEPS Software Tool", Lecture Notes in Computer Science, vol. 2794, Setembro 2003, pp. 98–115.

[CZE07] R. M. Czekster, P. Fernandes, J. M. Vincent e T. Webber. "Split: a flexible and efficient algorithm to vector-descriptor product", In: Proceedings of the $2^{nd}$ International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS), 2007, 8p.

[DER03] C. A. F. De Rose e P. O. A. Navaux. "Arquiteturas Paralelas", Porto Alegre, Brasil: Editora Sagra Luzzatto, 2003. Série Livros Didáticos.

[DEA98] D. D. Deavours e W. H. Sanders. "An efficient disk-based tool for solving large Markov models", Performance Evaluation, vol. 33-1, Junho 1998, pp. 67–84.

[DOT05] F. L. Dotti, P. Fernandes, A. Sales e O. M. Santos. "Modular Analytical Performance Models for Ad Hoc Wireless Networks", In: Proceedings of the $3^{rd}$ International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt), 2005, pp. 164–173.

[FER98] P. Fernandes. "Méthodes numériques pour la solution de systèmes Markoviens à grand espace d'états", Tese de Doutorado, Institut National Polytechnique de Grenoble, France, 1998, 262p.

[FER98a] P. Fernandes, B. Plateau e W. J. Stewart. "Efficient descriptor - Vector multiplication in Stochastic Automata Networks", Journal of the ACM, vol. 45-3, Maio 1998, pp. 381–414.

[FLY66] M. J. Flynn. "Very high-speed computing systems", In: Proceedings of the IEEE, vol. 54-12, Dezembro 1966, pp. 1901–1909.

[HEN02] J. L. Hennessy e D. A. Patterson. "Computer architecture: a quantitative approach", San Mateo, USA: Morgan Kaufmann, 2002, $3^a$ edição.

[HWA92] K. Hwang. "Advanced computer architecture: parallelism, scalability, programmability", Boston, USA: McGraw-Hill, 1992, $1^a$ edição.

[HWA98] K. Hwang e Z. Xu. "Scalable parallel computing: technology, architecture, programming", Boston, USA: McGraw-Hill, 1998, $1^a$ edição.

[MPI97] "MPI-2 standard", http://www.mpi-forum.org/docs/mpi2-report.pdf, acessado em 12 de fevereiro de 2008.

[MPI07] "MPICH2 User's Guide", ftp://ftp.mcs.anl.gov/pub/mpi/mpich2-doc-user.pdf, acessado em 30 de janeiro de 2008.

[PLA84] B. Plateau. "De l'evaluation du paralllisme et de la synchronisation", Tese de Doutorado, Paris-Sud, Orsay, France, 1984.

[PLA91] B. Plateau e K. Atif. "Stochastic Automata Networks for modeling parallel systems", IEEE Transactions on Software Engineering, vol. 17-10, Outubro 1991, pp. 1093–1108.

[STE94] W. J. Stewart. "Introduction to the numerical solution of Markov chains", Princeton University Press, 1994, 539p.

[TAN06] A. S. Tanenbaum e M. van Steen. "Distributed systems: principles and paradigms", USA: Prentice Hall, 2006, $2^a$ edição.