ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

MARLIM PEREIRA MENEZES

**METODOLOGIA PARA PARALELIZAÇÃO E OTIMIZAÇÃO DE
MODELOS MATEMÁTICOS E COMPUTACIONAIS, UTILIZANDO UMA
NOVA LINGUAGEM DE PROGRAMAÇÃO**

São Paulo

2013

MARLIM PEREIRA MENEZES

**METODOLOGIA PARA PARALELIZAÇÃO E OTIMIZAÇÃO DE
MODELOS MATEMÁTICOS E COMPUTACIONAIS, UTILIZANDO UMA
NOVA LINGUAGEM DE PROGRAMAÇÃO**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção
de título de Doutor em Ciências.

Área de Concentração:
Engenharia Elétrica – Sistemas de
Potência

Orientador:
Prof. Dr. Carlos Eduardo de Moraes
Pereira

São Paulo

2013

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 15 de outubro de 2013.

Assinatura do autor _____

Assinatura do orientador _____

FICHA CATALOGRÁFICA

Menezes, Marlim Pereira

**Metodologia para paralelização e otimização de modelos matemáticos e computacionais, utilizando uma nova linguagem de programação / M.P.Menezes. -- versão corr. -- São Paulo, 2013.
230 p.**

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Energia e Automação Elétricas.

1. Metodologia e técnicas de programação 2. Modelagem computacional 3. Programação paralela 4. Linguagem de programação 5. Sistemas elétricos I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Energia e Automação Elétricas II. t.

DEDICATÓRIA

Dedico este trabalho à minha esposa Vera e ao meu querido filho Yan, que me apoiaram plenamente durante todo o período de seu desenvolvimento.

AGRADECIMENTOS

Ao Prof. **Dr. Carlos Eduardo de Moraes Pereira** do PEA/POLI/USP pela orientação e pelo constante estímulo transmitido durante todo este trabalho.

À Profa. **Dra. Liria Matsumoto Sato** do PCS/POLI/USP pelas sugestões e comentários apresentados em meu exame de qualificação e pelo incansável apoio e constante estímulo transmitido durante todo o desenvolvimento deste trabalho.

Ao Prof. **Dr. José Aquiles Baesso Grimoni** do PEA/POLI/USP pelas sugestões e comentários apresentados em meu exame de qualificação e na defesa desta Tese.

Ao Prof. **Dr. Fransisco Isidro Massetto** da UFABC pelas sugestões e comentários apresentados na defesa desta Tese.

À Profa. **Dra. Isabel Cristina Italiano**, da EACH/USP/LESTE pelas sugestões e comentários apresentados na defesa desta Tese.

Ao Prof. **Dr. Jean Marcos Laine** da FATEC pelas sugestões e comentários apresentados na defesa desta Tese.

A CAPES e ao CNPq pela bolsa de estudo concedida.

A todos aqueles que colaboraram direta ou indiretamente, na execução deste trabalho.

“O futuro é construído pelas nossas
decisões diárias, inconstantes e mutáveis, e
cada evento influencia todos os outros.”

Alvin Toffler

RESUMO

Ao final desta pesquisa deseja-se que haja uma metodologia eficiente, cuja finalidade será auxiliar o usuário na transformação de modelos matemáticos e computacionais codificados para computadores sequenciais, em modelos paralelos otimizados para executarem em microcomputadores pessoais modernos, constituídos de CPU com múltiplos núcleos ou de híbridos (CPU + GPGPU) integrados no mesmo chip, com ou sem processadores gráficos (GPGPU) densamente paralelos instalados, mantendo a qualidade de seus resultados originais, com respeito à sua precisão numérica, mas com uma diminuição considerável no tempo de processamento.

A emergência, em meados da década 2000, dessas novas arquiteturas de hardware elevou a capacidade de processamento dos microcomputadores pessoais aos patamares dos computadores de grande porte de apenas alguns anos atrás.

Este trabalho de pesquisa apresenta duas metodologias, onde a primeira metodologia é composta de três partes e a segunda de duas partes. Somente a terceira parte da primeira metodologia é dependente de tecnologias de hardware.

Palavras chave: Metodologia e técnicas de programação. Modelagem computacional. Programação paralela. Linguagem de programação. Sistemas elétricos.

ABSTRACT

At the end of this research project, an efficient methodology is expected with the purpose of assisting users in the processing of mathematical and computer models coded for sequential computers in parallel models that are optimized to run on modern personal computers, consisting of a CPU with multiple or hybrid (CPU + GPGPU) cores integrated into the same chip, with or without massively parallel graphics processors (GPGPU) installed. This will ensure the original quality of the results with respect to numerical accuracy, but with a considerable reduction in processing time.

The emergence of these new hardware architectures in the mid-2000s increased the processing power of personal computers to the levels of mainframe computers from just a few years previously.

This research work presents two methodologies, where the first methodology is composed of three parts and the second methodology is composed of two parts. Only the third part of the first methodology is dependent on hardware technologies.

Keywords: Methodology and programming techniques. Computational modeling. Parallel programming. Programming language. Electrical systems.

LISTA DE ABREVIATURAS E SIGLAS

ADT	- <i>Abstract Data Type</i>
API	- <i>Application Programming Interface</i>
BSP	- <i>Bulk Synchronous Parallel model</i>
BSP-L	- <i>Bulk-Synchronous Parallel Language</i>
CPU	- <i>Central Processing Unit (Unidade Central de Processamento)</i>
CUDA	- <i>Compute Unified Device Architecture</i>
DSM	- <i>Distributed Shared Memory</i>
ESPG	- <i>Expressão Semântica de Processamento Geral</i>
GPU	- <i>Graphic Processing Unit (Unidade de Processamento Gráfico)</i>
GPGPU	- <i>General-Purpose GPU (GPU de Propósito Geral)</i>
HPC	- <i>High-Performance Computing (Computação de Alto Desempenho)</i>
iGPU	- <i>Integrated GPU (GPU integrada)</i>
LAN	- <i>Local Area Network (Rede de área local)</i>
MDI	- <i>Método das Divisões Internas</i>
MDIM	- <i>Método das Divisões Internas Melhorado</i>
MGP	- <i>Método Gráfico Prático</i>
MIMD	- <i>Multiple Instruction, Multiple Data stream</i>
MISD	- <i>Multiple Instruction, Single Data stream</i>
MPI	- <i>Message Passing Interface (Interface de passagem de mensagens)</i>
MPMD	- <i>Multiple Program – Multiple Data stream</i>
MPP	- <i>Massively Parallel Processor (Processador densamente paralelo)</i>
NUMA	- <i>NonUniform Memory Access (Acesso à Memória Não Uniforme)</i>
cc-NUMA	- <i>cache-coherent NUMA (NUMA com cache coerente)</i>
OOP	- <i>Oriented-Object Programming (Programação Orientada a Objetos)</i>
OpenCL	- <i>Open Computing Language</i>
OpenMP	- <i>Open Multi-Processing (Multiprocessamento aberto)</i>
PE	- <i>Elemento de processamento (Processing Element)</i>
PRAM	- <i>Parallel Random Access Machine (Máquina de acesso randômico paralela)</i>
PVM	- <i>Parallel Virtual Machine (Máquina virtual paralela)</i>
RAM	- <i>Random Access Machine (Máquina de acesso randômico)</i>
RAM	- <i>Random Access Memory (Memória de acesso randômico)</i>
SIMD	- <i>Single Instruction, Multiple Data stream</i>
SISD	- <i>Single Instruction, Single Data stream</i>
SMP	- <i>Symmetric Multiprocessors (Multiprocessadores simétricos)</i>
SPMD	- <i>Single Program, Multiple Data (Programa Único, Múltiplos Dados)</i>
UE	- <i>Unidade de Execução (Unit of Execution)</i>
UMA	- <i>Uniform Memory Access (Acesso à Memória Uniforme)</i>
WAN	- <i>Wide Area Network (Rede de longa distância)</i>

SUMÁRIO

1 INTRODUÇÃO	1
1.1 Motivação	2
1.2 Organização do Trabalho	3
2 TRABALHOS RELACIONADOS	7
2.1 Metodologias para Programação Paralela	8
2.2 Considerações.....	12
3 RESUMO DA TEORIA DE COMPUTAÇÃO PARALELA.....	13
3.1 Modelos Computacionais	15
3.1.1 RAM – Random-Access Machine	16
3.1.2 PRAM – Parallel Random-Access Machine.....	17
3.1.3 BSP – Bulk Synchronous Parallel model	18
3.2 Arquiteturas Paralelas	20
3.2.1 Hardware	20
3.2.1.1 Taxonomias de Flynn.....	21
3.2.1.2 Pipeline	26
3.2.1.3 Computação em Grade (<i>Grid Computing</i>)	29
3.2.1.4 Hipercubo	30
3.3.2 Software	31
3.3.2.1 Fator ganho de velocidade (<i>Speedup Factor</i>).....	32
3.3.2.2 Custo Computacional.....	33
3.3.2.3 Lei de Amdahl.....	34
3.3.2.4 Lei de Gustafson.....	36
3.3 Técnicas de Computação Paralela	37

3.3.1 Sincronismo	38
3.3.2 Condições de Corrida (<i>Race Conditions</i>)	38
3.3.3 Impasse (<i>Deadlock</i>)	40
3.3.4 Semáforo	42
3.3.5 Monitor	42
3.3.6 Exclusão Mútua (<i>Mutex</i>)	43
3.3.7 Balanceamento de Carga	43
3.3.8 Análise de dependência de dados	44
3.3.9 Granularidade	47
3.3.10 Memória Compartilhada	48
3.3.11 Memória Distribuída ou Computação Distribuída	49
3.3.12 Sistemas Híbridos	50
3.4 Ambientes de Programação Paralela	51
4 METODOLOGIAS PROPOSTAS	57
4.1 Escopo Completo da Metodologia de Paralelização Proposta	60
4.1.1 Modelo a Paralelizar e Otimizar – ENTRADA	61
4.1.2 Metodologia de Paralelização e Otimização – TRANSFORMAÇÃO	61
4.1.3 Modelo Paralelizado e Otimizado – SAÍDA	62
5 EXTRAINDO PARALELISMO	64
5.1 Especificações da Extração de Paralelismo	64
5.2 Paralelizando Expressões Aritméticas	72
6 ESPG – EXPRESSÃO SEMÂNTICA DE PROCESSAMENTO GERAL	76
6.1 Especificações da ESPG	76
6.2 Exemplos Práticos de ESPGs	80
7 Ψ – UMA NOVA LINGUAGEM DE PROGRAMAÇÃO	84
7.1 Detalhamentos das Regiões de um Programa Ψ	87
7.2 Formato do Arquivo de Cabeçalho da Linguagem Ψ	88
7.3 Exemplos de Programas Escritos em Linguagem Ψ	90
8 CÁLCULO DE DETERMINANTE	94
8.1 Método Proposto	96
8.1.1 MDI – Método das Divisões Internas	97
8.1.2 MGP – Método Gráfico Prático	99
8.1.3 Modelos Computacionais	101

8.1.3.1 Modelos Sequenciais.....	101
8.1.3.2 Modelo Paralelo Tradicional	103
8.2 MDIM – Método das Divisões Internas Melhorado	107
9 ESTUDOS DE CASO E RESULTADOS.....	116
9.1 Estudo de Caso 1: Multiplicação de Matrizes	117
9.1.1 Solução Analítica	117
9.1.2 Extração de Paralelismo	119
9.1.3 Obtenção das ESPGs.....	120
9.1.4 Codificação em Ψ	121
9.1.5 Codificação em C++	123
9.1.6 Resultados.....	125
9.2 Estudo de Caso 2: Curto Circuito em Sistemas de Potência	129
9.2.1 Solução Analítica	131
9.2.2 Extração de Paralelismo	136
9.2.3 Obtenção das ESPGs.....	142
9.2.4 Codificação em Ψ	144
9.2.5 Codificação em C++	149
9.2.6 Resultados.....	156
10 CONCLUSÕES E TRABALHOS FUTUROS	160
REFERÊNCIAS.....	164
REFERÊNCIAS COMPLEMENTARES	169
APÊNDICES	174
APÊNDICE A	174
LINGUAGEM Ψ	174
A.1 Especificação Simplificada.....	175
APÊNDICE B	183
RESUMO DA TEORIA DE DETERMINANTES	183
B.1 Trabalhos Relacionados	184
B.2 Regra de Chió	185
B.3 Referências	187
APÊNDICE C	189
CÁLCULO DE CURTO CIRCUITO EM SISTEMAS DE POTÊNCIA.....	189
ANEXOS	198

Anexo A: Código fonte seqMDIM.cpp.....	198
Anexo B: Código fonte parMDIM.cpp	202
Anexo C: Código fonte seqMatMul.cpp.....	206
Anexo D: Código fonte parMatMul.cpp	209
Anexo E: Código fonte seqShortCircuit.cpp.....	213
Anexo F: Código fonte parShortCircuit.cpp	217

1 INTRODUÇÃO

Esta pesquisa tem como objetivo principal atingir um elevado grau de qualidade no que concerne ao desenvolvimento de uma nova Metodologia que visa a Transformação de Modelos Matemáticos e Computacionais Sequenciais ou Híbridos em seus equivalentes Paralelos de forma otimizada, porém sem a perda de sua generalidade, ou seja, mantendo a qualidade de seus resultados originais, com respeito à sua precisão numérica, mas com uma diminuição considerável no tempo de processamento. Há, também, uma segunda Metodologia constituída de um método analítico e um método gráfico, para o cálculo de determinantes de ordem n , forma direta, porém com vantagem sobre a Regra de Chió em n iterações.

Cada parte dessas metodologias é chamada de método ou processo, assim na primeira metodologia, intitulada, resumidamente, de *Metodologia de Paralelização e Otimização* temos um método chamado *Extraíndo Paralelismo* como primeira parte, um processo chamado *Expressão Semântica de Processamento Geral (ESPG)* como segunda parte e, por fim, uma terceira parte que constitui a especificação, parcial, de uma nova linguagem de programação, que é implicitamente paralela e chamada de Ψ (*Psi*). A segunda metodologia possui dois métodos, sendo o primeiro chamado de *Método das Divisões Internas Melhorado (MDIM)* e o segundo chamado de *Método Gráfico Prático (MGP)* que dá suporte ao anterior.

Nos diversos segmentos da matemática há situações em que o paralelismo existe naturalmente, entretanto, há muitas outras situações nas quais o paralelismo não se

apresenta explicitamente, sendo necessário apresentar esse paralelismo oculto de forma explícita através de técnicas especiais.

Alguns segmentos da matemática possuem seus algoritmos essencialmente paralelos, entretanto tendo sido projetados para ser computacionalmente sequenciais. Isso ocorreu devido o fato dos computadores de arquiteturas sequenciais possuírem a tecnologia dominante durante várias décadas, desde seu surgimento, já que são mais baratos de se projetar e construir e, portanto, altamente disponíveis em relação aos computadores de arquiteturas paralelas, que são muito mais complexos, caros e difíceis de ser construídos (MIKLOSKO; KOTOV, 1984).

O paralelismo natural está presente, principalmente, nos algoritmos de computação numérica tais como nas equações diferenciais parciais e na álgebra linear, pois nessas áreas encontra-se elevado grau de *independência* entre muitos de seus elementos matemáticos, o que é fundamental para o paralelismo.

1.1 Motivação

A necessidade de se aprimorar os modelos matemáticos e computacionais, de modo a se fazer uso das modernas arquiteturas paralelas de forma mais completa e eficiente, foi o principal fator motivador que nos levou ao desenvolvimento deste trabalho.

As modernas CPUs (Central Processing Units), constituídas de *muitos núcleos* (multicores), possuem de dois até algumas dezenas de núcleos físicos em um único encapsulamento. Entretanto, alguns desses modelos apresentam a característica de dobrar virtualmente a quantidade de CPUs, aumentando, dessa forma, sua capacidade computacional.

Já os modernos processadores gráficos de propósitos gerais, mais conhecidos como GPGPU (General-Purpose Graphic Processing Unit), possuem de alguns até milhares de núcleos físicos e são também classificados como processadores massivamente (densamente) paralelos. A sua classificação em termos do número de núcleos físicos de processamento é *muitos núcleos* (many-core).

Há, ainda, uma novíssima arquitetura de processadores que traz um híbrido das CPUs multicore com as GPGPUs no mesmo invólucro. Neste caso a GPGPU integrada ao mesmo encapsulamento da CPU é chamada de iGPU (do inglês: Integrated GPU), que ainda trás a capacidade de ser programada pelo usuário, dando a possibilidade de aumentar ainda mais a capacidade de processamento das aplicações computacionais nos computadores mais recentes.

1.2 Organização do Trabalho

Este trabalho está organizado em alguns capítulos, conforme apresentado nas linhas que se seguem:

- **1. INTRODUÇÃO:** Esclarece o objetivo deste trabalho e a motivação que nos levou a desenvolvê-lo e, por fim a organização do texto como um todo.
- **2. REVISÃO DA LITERATURA:** Apresenta uma revisão bibliográfica de algumas metodologias utilizadas como comparativo à metodologia em desenvolvimento neste trabalho bem como um resumo histórico de pesquisas que foram realizadas sobre computação paralela.
- **3. RESUMO DA TEORIA DE COMPUTAÇÃO PARALELA:** Este capítulo é dedicado à Teoria de Computação Paralela, onde são apresentados um resumo histórico de sua evolução e conceitos sobre as diferentes taxonomias e técnicas de paralelização empregadas no corrente trabalho.
- **4. METODOLOGIAS PROPOSTAS:** Apresenta uma visão geral das metodologias propostas que serão desenvolvidas ao longo do restante do texto. São duas as metodologias propostas, sendo que a primeira é *Metodologia de Paralelização e Otimização* constituída de um método, um processo e a especificação de uma nova linguagem de programação paralela. A segunda metodologia é constituída de dois métodos, sendo um analítico e outro gráfico, que são empregados no cálculo direto de determinantes de matrizes densas de ordem n .
- **5. EXTRAINDO PARALELISMO:** Neste capítulo é apresentado um método de obtenção de paralelismo a partir de um dado problema, em detalhes. Este

método gera resultados para os dois métodos seguintes, que constituem a *Metodologia de Paralelização e Otimização*. Este método é conceitual e, portanto, independente de tecnologias de hardware.

- 6. *ESPG – EXPRESSÃO SEMÂNTICA DE PROCESSAMENTO GERAL*: A ESPG é um processo e é a segunda parte da *Metodologia de Paralelização e Otimização*. Ela é independente de tecnologia de hardware tendo sido desenvolvida para comportar-se como um repositório para o resultado obtido na primeira etapa e preparar uma estrutura para a terceira parte. Este método também é conceitual e independente de tecnologias de hardware.
- 7. Ψ (*Psi*) – *UMA NOVA LINGUAGEM DE PROGRAMAÇÃO*: A linguagem de programação Ψ constitui a terceira parte da *Metodologia de Paralelização e Otimização*. Este capítulo faz uma apresentação resumida da especificação da linguagem proposta, mas com clareza suficiente para o adequado entendimento da sua importância, facilidade de uso e capacidades de processamento. Parte da estrutura da linguagem é apresentada. Também é apresentada a sua estreita relação com as duas primeiras partes da *Metodologia de Paralelização e Otimização* através de exemplos práticos. Por ser uma linguagem de programação, esta tem elevada dependência de tecnologias de hardware.
- 8. *CÁLCULO DE DETERMINANTE*: Este capítulo apresenta a segunda metodologia desenvolvida ao longo deste trabalho. Trata-se de um método empregado no cálculo direto de determinantes de matrizes densas de ordem n de forma puramente determinística, onde essa se apresenta em duas etapas, a forma analítica (**MDI – Método das Divisões Internas**) e uma forma gráfica prática (**MGP – Método Gráfico Prático**) que pode ser facilmente usada tanto para o cálculo manual quanto computadorizado. Também são apresentados algoritmos sequenciais e um paralelo utilizando este método. Ainda, nesse capítulo é apresentado o **Método das Divisões Internas Melhorado (MDIM)** e a comparação com seu antecessor MDI e com a Regra de Chió. O principal objetivo aqui é mostrar que é possível executar esta tarefa com um algoritmo de complexidade computacional $O(n)$. Entretanto, nenhuma implementação desse algoritmo é apresentada.

- **9. ESTUDOS DE CASO E RESULTADOS:** São apresentados estudos de casos com abordagem puramente matemática e envolvendo situações reais do dia a dia em engenharia elétrica. Os exemplos apresentados neste capítulo levam em consideração o uso da Metodologia de paralelização aqui proposta, desde as técnicas aplicadas para obtenção de paralelismo dos modelos matemáticos do problema em análise, sua representação sintetizada através da ESPG até a codificação na linguagem de programação Ψ . Os programas fonte em linguagem C++ equivalentes aos códigos em Ψ também são apresentados. Com exceção dos modelos pipeline.
- **10. CONCLUSÕES E TRABALHOS FUTUROS:** Apresenta argumentações a respeito dos procedimentos realizados e uma discussão sobre os resultados obtidos no decorrer dos ensaios da corrente metodologia de paralelização. Também, faz-se uma apresentação da etapa seguinte, em nível mais avançado, deste trabalho, visando uma melhoria na metodologia de paralelização aqui proposta, a fim de alcançar um público maior.
- **REFERÊNCIAS:** Lista das referências bibliográficas citadas ao longo do texto.
- **REFERÊNCIAS COMPLEMENTARES:** Lista das referências bibliográficas consultadas, mas não citadas ao longo do texto.
- **APÊNDICES:**
 - **A. LINGUAGEM Ψ (Psi) – PRÉVIA ESPECIFICAÇÃO:** Este apêndice apresenta a linguagem de programação proposta Ψ e sua especificação, de uma forma básica e suficiente para o adequado entendimento da mesma no estágio de desenvolvimento em que ela se encontra.
 - **B. RESUMO DA TEORIA DE DETERMINANTES:** Este apêndice apresenta um breve resumo do cálculo de determinantes.
 - **C. CÁLCULO DE CURTO CIRCUITO EM SISTEMAS DE POTÊNCIA:** Este apêndice apresenta um resumo da teoria de cálculo de curto circuito em sistemas de potência, que são aplicados no estudo de caso de Engenharia Elétrica.
- **ANEXOS:**
 - **A. Listagem do programa *sequencial* para o Cálculo de Determinante**

de matrizes densas usando a metodologia MDIM, proposta no capítulo 8 deste texto, em linguagem C++.

- *B. Listagem do programa **paralelo** para o Cálculo de Determinante de matrizes densas usando a metodologia MDIM, proposta no capítulo 8 deste texto, em linguagem C++.*
- *C. Listagem do programa **sequencial** para o Estudo de Caso 1: Multiplicação de Matrizes densas em linguagem C++.*
- *D. Listagem do programa **paralelo** para o Estudo de Caso 1: Multiplicação de Matrizes densas em linguagem C++.*
- *E. Listagem do programa **sequencial** para o Estudo de Caso 2: Cálculo de Curto Circuito em Sistemas de Potência em linguagem C++.*
- *F. Listagem do programa **paralelo** para o Estudo de Caso 2: Cálculo de Curto Circuito em Sistemas de Potência em linguagem C++.*

2 TRABALHOS RELACIONADOS

Alguns algoritmos são em sua essência altamente paralelos, mas projetados para serem computacionalmente sequenciais. Algoritmos que contêm paralelismo natural consistem de operadores mutuamente exclusivos, que requerem para sua execução os resultados de apenas alguns poucos outros operadores. Dessa forma, esses algoritmos foram transformados em processos sequenciais apenas para implementação em computadores sequenciais (MIKLOSKO; KOTOV, 1984).

O paralelismo natural está presente principalmente na álgebra linear e em algoritmos para computação numérica de equações diferenciais parciais. Eles estão presentes, também em algoritmos baseados na computação iterativa de um mesmo operador sobre diferentes dados, tais como o método das aproximações sucessivas, alguns métodos para a solução de sistemas de equações lineares, método Monte Carlo, integração numérica e em algoritmos complexos que se constituem de um grande número de operadores quase independentes, por exemplo, como a solução iterativa do problema de Diricklet (MIKLOSKO; KOTOV, 1984).

Um bom exemplo de processo computacional com paralelismo natural é descrito através da decomposição de uma matriz para a equação de Poisson (BUZBEE, 1973).

A teoria dos Operadores Autodestrutivos, proposta por (RICIERE, 1987) também consiste em outro exemplo de processo matemático com paralelismo natural, que

com poucas propriedades é possível se obter as derivadas primeira, segunda, terceira, etc., de polinômios, simultaneamente, sem dependência entre elas.

As operações básicas de adição, multiplicação e cálculo de determinantes de matrizes também apresentam alto grau de paralelismo natural e, é facilmente observável.

De forma geral, diante das tecnologias de hardware disponíveis atualmente, toda função polinomial pode ser paralelizável facilmente se aplicadas três das propriedades básicas da aritmética, que são a *associativa*, *comutativa* e *distributiva* de modo a se obter um reagrupamento de todas as operações do polinômio, para serem executadas com o mínimo possível de dependência.

2.1 Metodologias para Programação Paralela

É possível se encontrar diversos métodos para paralelização de algoritmos na literatura acadêmica, dos quais alguns serão apresentados nos próximos parágrafos.

Jamieson et al. (1987) propuseram uma metodologia de programação paralela chamada *THE FORCE*, que na verdade consiste em um macro pré-processador estendido para a linguagem de programação Fortran. Ela evoluiu no decorrer de um curso voltado para a implementação de algoritmos de natureza numérica em um sistema multiprocessador de memória compartilhada. A linguagem *The Force* suportava de 20 a 200 processos simultâneos e, incluía operações primitivas suportando *granularidade*¹ fina e grossa.

¹ Esta palavra, frequentemente usada em Computação de Alto Desempenho, Bancos de Dados e outras áreas, não existe no Dicionário da Língua Portuguesa – o Novo Dicionário Aurélio. Em seu lugar, foi encontrada a palavra **granulosidade**, com o mesmo significado usado pela comunidade para a palavra **granularidade**. Entretanto, os tradutores online reconhecem ambas as grafias e classifica-as como sinônimos. De acordo com o Novo Dicionário Aurélio da língua portuguesa: “**granulosidade** – [De *granuloso* + *-(i)dade*.] – **Substantivo feminino** 1. **Qualidade de granuloso**.”.

O trabalho sobre *The Force*, como uma extensão de linguagem de programação foi iniciada enquanto o autor estava em licença no Laboratório de Pesquisa Balística do Exército dos Estados Unidos da América do Norte (EUA) durante o ano acadêmico 1983-1984. Ela foi utilizada para cálculos de dinâmica dos fluidos no BRL (Ballistics Research Laboratory of the U. S. Army) por Nisheeth Patel durante a primavera de 1984.

Com a linguagem *The Force* o usuário não necessitava escrever comandos de comunicação tais como *send* e *receive* para a troca de mensagens entre os processos. A transferência de dados entre processos é baseada em coincidência no nome da variável de memória compartilhada. A eficiência existente no modelo de multiprocessadores com memória compartilhada também foi um fator importante no projeto da linguagem. Seu modelo computacional é baseado no conceito de fluxo de instruções.

O modelo computacional de *The Force* suprime todo o controle explícito de processos. Entretanto, um número não especificado, mas fixo de processos é assumido estar disponível no início da execução de um programa escrito em *The Force*. Por fim, a visão de um modelo computacional como um conjunto hierarquicamente estruturado de funções é bem estabelecida.

Kurfess (1991) apresenta uma metodologia de paralelismo envolvendo software e hardware através de uma linguagem chamada *UNITY*. Assim, quando se diz a frase “um programa *UNITY*” significa que está se dizendo “um programa escrito em notação *Unbounded Nondeterministic Iterative Transformation*”, que na verdade não é uma linguagem de programação completa, mas tão somente um mecanismo notacional para especificar alguns aspectos de programas tais como a declaração de variáveis, a especificação de valores iniciais e um conjunto de múltiplos comandos de atribuição, especialmente paralelos, de forma sistemática para várias arquiteturas e aplicações *apud* Kurfess (1991, p.34).

Um dos principais propósitos de *UNITY* é o desacoplamento das fases de projeto, especificação, codificação e implementação durante o desenvolvimento de

programas, diz o autor. Kurfess (1991), também apresenta outras metodologias de paralelismo utilizando linguagens com abordagens voltadas para o hardware. Entretanto, todas essas metodologias podem ser estendidas para linguagens de programação paralela de alto nível sem dificuldades, já que fazem uso extensivo de conceitos e formalidades para ambos os níveis de processamento computacional.

Pitas (1995) apresenta diversas técnicas de paralelização de dados e tarefas focando exclusivamente nas taxonomias SIMD (Single Instruction – Multiple Data stream) e MIMD (Multiple Instruction - Multiple Data stream) de Flynn. Ele, também explora a metodologia de pipeline no paralelismo de algoritmos para processamento digital de imagens, redes neurais artificiais e visão computacional.

Dentre as diversas abordagens apresentadas por Pitas (1995) há um pacote para processamento digital de imagens chamado *PARALLEL EIKONA*, que foi desenvolvido inteiramente em linguagem C no Department of Electrical Engineering na University of Thessaliniki, durante o evento ESPRIT Parallel Computing Action (1990-1992).

Esse software combina uma implementação paralela de alto desempenho com uma interface de usuário amigável e, foi desenvolvida para ser executado em computadores do tipo transputer sob o sistema operacional HELIOS que por sua vez possui uma interface de usuário suportada pelo sistema operacional MICROSOFT WINDOWS.

Na metodologia *PARALLEL EIKONA* usa-se em sua arquitetura o modelo de paralelismo de dados porque ele resolve alguns objetivos críticos de projeto que são: a) balanceamento da distribuição de carga de trabalho na rede de processadores; b) distribuição automática da aplicação sobre qualquer processador da rede e; c) o razoável nível de esforço de programação.

Wilkinson e Allen (1999) apresentam algumas técnicas de programação paralela para se resolver problemas de ordenação de conjuntos de dados, cálculos numéricos envolvendo operações com matrizes, sistemas de equações lineares e

métodos iterativos de otimização numérica, processamento de imagens envolvendo a obtenção de histogramas, redução de ruído, detecção de arestas, transformações no domínio da frequência e otimização em busca utilizando os métodos Branch-and-Bound, Algoritmos Genéticos e refinamentos sucessivos.

De um modo geral, os autores não apresentam inovações na área já que outros autores contemporâneos tratam dos mesmos assuntos de forma similar, entretanto são métodos propostos para se resolver e otimizar, de forma paralela, diversos problemas similares de matemática e engenharia.

Mattson et al. (2005) apresentam uma metodologia de paralelismo baseada em uma série de Padrões para Programação Paralela (Patterns for Parallel Programming) explorando suas construções através de micro e macro tarefas com as linguagens de programação C, C++, com OpenMP e MPI e a plataforma JAVA. Técnicas tais como Dividir e Conquistar, Recursão, Decomposição Geométrica, Pipeline e Coordenação Baseada em Eventos, são utilizadas ao longo do texto.

Os autores procuram sempre decompor os exemplos usados em pequenas partes que possam ser isoladas sequencialmente e paralelamente e depois de alguns passos de refinamento reagrupadas para formar a solução desejada. No decorrer do processo eles vão esclarecendo, através de diagramas de blocos que caminhos consideram mais adequados a serem seguidos até chegar à solução final.

Uma estratégia básica, porém muito útil que os autores usam é a de mostrar pequenas soluções sequenciais dos problemas em pseudocódigos, normalmente voltados para uma determinada linguagem de programação e, comparando essas pequenas rotinas com trechos de programas com as soluções em paralelo, usando C, C++, OpenMP, MPI e Java, conforme citado anteriormente.

Como produto final da abordagem seguida por Mattson et al. (2005) pode-se perceber, claramente, a existência de uma coletânea de sub-rotinas criadas ao longo do texto e, que podem ser facilmente adaptadas por usuários menos experientes. No entanto, os autores não deixam de expor ao leitor os diversos perigos e algumas

armadilhas que deverão evitar no decorrer do desenvolvimento de software paralelo, qualquer que seja o modelo de memória utilizado, ou seja, compartilhada, distribuída ou híbrida.

2.2 Considerações

De um modo geral, está claro que a computação paralela já foi exaustivamente explorada por um número considerável de pesquisadores e estudiosos desde a década 1960. Com isso, conclui-se que, atualmente é muito difícil de conseguir apresentar uma proposta inovadora, de uma metodologia para programação paralela otimizada.

Entretanto, todas as propostas observadas nos textos analisados se aplicam somente às arquiteturas de hardware de sua época, que felizmente, cujas arquiteturas são válidas e extensivamente utilizadas até os dias atuais.

Todavia, felizmente as possibilidades de desenvolvimento de novas arquiteturas de hardware paralelo ainda não se esgotaram, mantendo-se aberto um espaço para oportunidades de surgimento de novas metodologias para o projeto e implementação de software paralelo.

Dessa forma, esta Tese apresenta uma metodologia inovadora no âmbito da computação paralela e distribuída. Sendo seu principal objetivo contribuir para a otimização de modelos matemáticos e computacionais sequenciais, ou paralelos, ou mesmo híbridos. Essa metodologia está dividida em três partes, onde a primeira parte é apresentada em detalhes no **capítulo 5**, a segunda parte está no **capítulo 6** e, por fim a terceira e última parte trata de uma linguagem de programação paralela, apresentada no **capítulo 7**, mas cuja especificação é abordada de forma simplificada no **Apêndice A**.

Uma segunda metodologia, referente à resolução de determinantes de matrizes altamente densas é apresentada no capítulo 8 desta Tese.

3 RESUMO DA TEORIA DE COMPUTAÇÃO PARALELA

Este capítulo contém um breve resumo sobre computação paralela e algumas de suas arquiteturas que têm sido demonstradas promissoras desde sua apresentação ao mundo.

O conceito de paralelismo computacional é bem antigo, mas os altos custos e a elevada complexidade de projeto contribuíram para o uso exaustivo do paradigma de processamento sequencial, na solução dos problemas científicos e de engenharia ao longo das primeiras décadas até meados da década de 2000.

Os primeiros algoritmos paralelos foram desenvolvidos no início da década de 1960, apesar das arquiteturas paralelas ainda não existirem naquela época, já naquele período e nos anos seguintes muitos pesquisadores eram fascinados pela esperança de resolver problemas assumindo a existência de ambientes paralelos, sem saber sobre a aplicabilidade de seus estudos *apud* Codenotti e Leoncini (1993).

No entanto, com o avanço da microeletrônica o projeto de hardware paralelo se tornou viável levando os computadores a terem uma queda dramática nos preços, se tornando cada vez mais poderosos, de acordo com a Lei de Moore (CODENOTTI; LEONCINI, 1993), (HWANG, 1993), (TANENBAUM, 2012), (STALLINGS, 2012), (PATTERSON; HENNESSY, 2005).

Normalmente, o uso de paralelismo é motivado pela necessidade de desejo do aumento de velocidade na solução de um dado problema de engenharia ou

científico, através do uso de algum tipo de hardware computacional (CODENOTE; LEONCINI, 1993), (HWANG, 1993), (WILKINSON; ALLEN, 1999).

Algumas áreas que requerem grande velocidade computacional incluem a modelagem numérica, a simulação de modelos científicos e de engenharia, pois tais modelos, frequentemente, necessitam de enormes quantidades de cálculos repetitivos sobre grandes quantidades de dados (KRISHNAMURTHY, 1993), (WILKINSON; ALLEN, 1999). É de natureza humana inovar nas aplicações que excedam a capacidade dos sistemas computacionais e requerer mais e mais velocidade de processamento que a disponível na sua época (CHAPMAN *et al.*, 2008).

A maneira utilizada para se alcançar maior velocidade de processamento e que vem sendo considerada há várias décadas é o uso de diversos processadores sequenciais no mesmo hardware, para operarem juntos na solução de um único problema (STALLINGS, 2012). Esse modelo de hardware é chamado de computador paralelo e a arquitetura de software empregada no seu uso é chamada de programação paralela. Esta abordagem fornece um aumento significativo na velocidade de processamento (KURFESS, 1991); (CODENOTE; LEONCINI, 1993), (HWANG, 1993), (KRISHNAMURTHY, 1993), (MATTSON *et al.*, 2005), (CHAPMAN *et al.*, 2008), (SANDERS; KADROT, 2011), (STALLINGS, 2012).

A Figura 1 mostra a evolução das arquiteturas dos computadores desde o modelo original da Máquina de von Neumann, usado na construção de máquinas sequenciais até as arquiteturas paralelas modernas (HWANG, 1993), se considerando também as mais recentes usadas na construção das CPUs com múltiplos núcleos físicos (multicores) e as GPGPUs com muitos núcleos físicos (many-core).

Os itens que se seguem, vide Figura 1, focam em algumas características presentes nas arquiteturas de processadores e computadores ao longo de sua evolução. Esta ilustração foi compilada a partir de dados e imagens contidas em (HWANG, 1993, p.10) e (STALLINGS, 2012, p.516).

Figura 1 – Evolução das arquiteturas computacionais desde a invenção do computador até meados da década de 2000.

Fontes: (HWANG, 1993, p.10) e (STALLINGS, 2012, p.516).

3.1 Modelos Computacionais

Alguns modelos descrevem precisamente os recursos computacionais, enquanto outros se concentram em algoritmos em detrimento aos detalhes da máquina, considerados de pouca importância (MILLER; BOXER, 2005).

Os modelos computacionais mais largamente usados são de longe as *máquinas de acesso randômico* (RAM – Random-Access Machine) e as Máquinas de Turing (CODENOTTI; LEONCINI, 1993), (KRISHNAMURTHY, 1993), (MILLER; BOXER, 2005).

Os itens a seguir resumem os conceitos das máquinas de acesso randômico e das máquinas paralelas de acesso randômico (PRAM – Parallel Random-Access Machine). Uma abordagem mais completa sobre máquinas de Turing pode ser encontrada em (CODENOTTI; LEONCINI, 1993), (LEWIS; PAPADIMITRIOU, 1998) e muitos outros textos importantes na literatura científica.

3.1.1 RAM – Random-Access Machine

A *máquina de acesso randômico* (RAM) é uma máquina abstrata que representa essencialmente um *modelo computacional sequencial*. Ela é constituída de um processador, uma unidade de acesso à memória e uma memória que consiste de um número ilimitado de registradores, onde cada registrador pode armazenar um valor inteiro (HWANG, 1993), (CODENOTTI; LEONCINI, 1993), (MILLER; BOXER, 2005).

Assume-se que o tempo de acesso à memória é constante, unitário e igual a $\Theta(1)$. O único processador presente na máquina opera com algoritmos sequenciais. A unidade de acesso à memória faz uma conexão direta entre o processador e o endereço de memória desejado.

A Figura 2 mostra o esquema de uma máquina de acesso randômico.

Figura 2 – Diagrama de blocos de uma máquina de acesso randômico.

Fonte: (Miller e Boxer, 2005, p.92).

Em uma máquina de acesso randômico a execução de um programa ocorre em três passos: a) leitura; b) processamento e; c) escrita. Durante esses passos a RAM se comporta maneira muito similar a um computador de arquitetura von Neumann. Assim, o tempo de acesso à memória, considerando o tempo de leitura, é da ordem de $O(\log M)$, onde M é o total de posições existentes na memória. Para maiores

detalhes sobre o modelo RAM recomenda-se a leitura de (HWANG, 1993), (CODENOTTI; LEONCINI, 1993), (MILLER; BOXER, 2005).

3.1.2 PRAM – Parallel Random-Access Machine

A *máquina paralela de acesso randômico* (PRAM), como o próprio nome sugere, é utilizada em modelos computacionais paralelos. Uma característica muito importante da PRAM é que se pode ignorar os problemas de comunicação de modo que o usuário possa concentrar todo o seu tempo e esforço na solução de seu problema, utilizando todo o potencial da mesma. Sua estrutura é dividida em três blocos: a) processadores; b) unidade de acesso à memória e; c) a própria memória (HWANG, 1993), (CODENOTTI; LEONCINI, 1993), (MILLER; BOXER, 2005).

A Figura 3 mostra o diagrama de blocos de uma máquina de acesso randômico paralela. Está evidente que a PRAM contém n processadores, $P_1, P_2, \dots, P_n$, onde cada um deles é idêntico a um processador de uma máquina de acesso randômico, ou seja, a uma máquina de von Neumann. Geralmente, esses processadores são chamados de *elementos de processamento* (PE). No caso da PRAM a memória é chamada de memória global, pois a mesma é comum a todos os processadores.

A memória global é controlada pela unidade de acesso à memória tanto para leitura quanto para escrita pelos processadores e o tempo necessário para se ler ou escrever na mesma é considerado constante e igual a $\Theta(1)$, tal como na máquina de acesso randômico. Como toda arquitetura paralela, mesmo a PRAM sendo virtual ela também apresenta os problemas de conflitos no acesso à memória, ou seja, mais de um processador pode estar solicitando à unidade de acesso à memória para ler ou escrever simultaneamente no mesmo endereço.

Felizmente, os conflitos de leitura e escrita na memória estão previstos e podem ser contornados através de alguns comandos predefinidos. Para resolver conflitos de leitura a PRAM utiliza os comandos: *exclusive read* (ER) e *concurrent read* (CR).

Figura 3 – Diagrama de blocos de uma máquina de acesso randômico paralela.

Fonte: (MILLER; BOXER, 2005, p.94).

Entretanto, os comandos que resolvem os conflitos de escrita na memória são: *exclusive write (EW)* e *concurrent write (CW)*, mas a possibilidade da escrita de modo concorrente é mais complicada e, por este motivo, existe um conjunto de esquemas de arbitragem que deve ser usado para contornar o problema: *priority CW*, *common CW*, *arbitrary CW* e *combining CW*. Para maiores detalhes sobre o modelo PRAM se recomenda a leitura de (HWANG, 1993), (CODENOTTI; LEONCINI, 1993), (MILLER; BOXER, 2005).

3.1.3 BSP – Bulk Synchronous Parallel model

O modelo BSP foi desenvolvido por Valiant (1990) *apud* Wilkinson e Allen (1999, p.421) visando suprir algumas deficiências do modelo PRAM, deficiências essas relacionadas à falta de suporte aos efeitos da hierarquia de memória e aos sistemas de passagem de mensagens. Diversas variações e extensões do modelo PRAM

foram sugeridos para incluir memória local e passagem de mensagens, mas o mais promissor foi o BSP que é composto de três características vantajosas sobre as demais propostas: a) esse modelo mantém um grupo de p processadores, cada um com sua própria memória particular; b) ele, também, tem uma rede de comunicação ponto-a-ponto para a troca de mensagens entre os processadores e; c) ele possui um mecanismo para sincronização de todos ou alguns processadores dentro de um intervalo bem definido. A Figura 4 ilustra um diagrama da visão geral do modelo BSP.

Figura 4 – Visão geral do modelo BSP.
Fonte: (WILKINSON; ALLEN, 1999, p.422).

No modelo BSP, vide Figura 4, as mensagens são transmitidas pelos processadores para solicitar acesso à memória particular de outros processadores.

Ainda, o modelo BSP permite que mais de um processo ou thread seja executado por um processador (VALIANT, 1990); (WILKINSON; ALLEN, 1999). Esse modelo foi estendido para a linguagem de programação BSP-L (BSP Language) por Cheatham *et al.* (1995) *apud* Wilkinson e Allen (1999), a qual usa o comando *forall* para iniciar e sincronizar processos.

3.2 Arquiteturas Paralelas

Há diversas arquiteturas paralelas de computadores, entre elas estão redes de estações de trabalho, clusters de computadores pessoais, supercomputadores densamente (“massivamente”) paralelos, multiprocessadores simétricos fortemente acoplados, estações de trabalho com múltiplos processadores, supercomputadores vetoriais, pipeline, grade (grid), etc. (CODENOTTI; LEONCINI, 1993), (WILKINSON; ALLEN, 1999), (GRAMA *et al.*, 2003), (MATTSON *et al.*, 2005).

Nos itens hardware e software a seguir estão apresentadas algumas dessas arquiteturas de hardware. Entretanto, sem qualquer formalismo, pois excelentes textos sobre o assunto podem ser encontrados nas referências citadas.

3.2.1 Hardware

Em se tratando de processamento paralelo, a organização de computadores apresenta alguns pontos importantes, que de acordo com Codenotti e Leoncini (1993) e Stallings (2012) são:

- Aumentar o desempenho de sistemas computadorizados usando múltiplos processadores que executem uma mesma tarefa em paralelo, a fim de suportar uma carga de trabalho maior.
- Existem duas arquiteturas mais comuns de múltiplos processadores que são os multiprocessadores simétricos (SMP) e os clusters.
- O uso da arquitetura mais recente que permite o acesso não uniforme à memória (NUMA) também vem sendo introduzida no mercado com bastante sucesso.
- É importante deixar claro que um SMP consiste de vários processadores semelhantes, dentro de um mesmo computador, interconectados por um barramento apropriado ou algum outro tipo de arranjo de comutação. Entretanto, um sistema baseado em SMP apresenta um problema crítico a ser resolvido, que é a coerência de cache, isto é, como cada processador possui sua própria cache é possível que uma determinada informação esteja

presente em mais de uma cache simultaneamente. Por essa razão, se a informação de uma cache for alterada, então a mesma informação deverá ser alterada em todas as caches que mantêm sua cópia.

- O arranjo que contém mais de um processador implementado em um único chip é chamado de processador de multiprocessamento. Essa arquitetura é nova e remonta de meados da década de 2000. Seu esquema de projeto visa replicar alguns dos componentes de um único processador, a fim de que o processador possa executar várias tarefas ou threads concorrentemente, por esta razão ele é chamado de processador multithread.
- Um conjunto completo de computadores interconectados de modo a poderem trabalhar juntos em uma mesma tarefa é chamado de cluster. Esse arranjo dá a ilusão ao programador de que ele está programando apenas uma máquina.
- Um sistema multiprocessador do tipo NUMA utiliza memória compartilhada, no qual o tempo de acesso de qualquer processador a uma palavra na memória varia conforme a posição daquela palavra na memória.

As abordagens acima citadas permitem a otimização da unidade de lógica e aritmética para processamento de vetores e matrizes com números de ponto flutuante e, também são comuns nos sistemas conhecidos como supercomputadores (STALLINGS; 2012).

As taxonomias de Flynn (FLYNN, 1972) ainda são as categorias de sistemas computacionais mais comuns para categorizar sistemas com capacidade de processamento paralelo e as mesmas serão apresentadas nos tópicos que se seguem.

3.2.1.1 Taxonomias de Flynn

São utilizadas para se classificar arquiteturas de computadores baseadas nos conceitos de fluxos de instruções e de dados e foram propostas por Michael J. Flynn em 1966 (MILLER; BOXER, 2005). Elas estão agrupadas em quatro categorias que definem fluxos únicos e múltiplos (CODENOTTI; LEONCINI, 1993), (PITAS, 1995), (WILKINSON; ALLEN, 1999). Essas classificações baseiam-se no número de

instruções concorrentes e nos dados disponíveis na arquitetura, conforme apresentadas nas próximas seções.

Um *fluxo de instruções* é definido como sendo uma sequência de instruções executadas pelo computador, enquanto um fluxo de dados é definido como sendo uma sequência de itens de dados operados pelo fluxo de instruções.

Os diagramas de blocos representativos das taxonomias de Flynn apresentados nos próximos itens foram obtidos a partir dos originais apresentados pelo próprio Flynn (1966) e Flynn (1972), mas com diferentes variações em seus formatos ilustrados por diversos autores.

3.2.1.1.1 SISD – Single Instruction, Single Data Stream

Esta é a classificação dada a computadores de um único processador com um único conjunto de dados sendo operado pelo processador, já que o conjunto de instruções é executado de forma sequencial.

A Figura 5 apresenta de forma simplificada o esquema de uma máquina construída sob a abordagem da arquitetura SISD. Note que há um único fluxo de instruções operando sobre um único conjunto de dados.

Figura 5 – Arquitetura para a taxonomia de Flynn SISD.

Fontes: (MATTSON *et al.*, 2005), (STALLINGS, 2012, p.517) e outras.

O modelo de computação RAM (Random-Access Machine) é uma típica máquina virtual desta categoria, tal como muitos computadores sequenciais incluindo os computadores pessoais e as estações de trabalho.

A arquitetura von Neumann é essencialmente uma máquina da categoria SISD (CODENOTTI; LEONCINI, 1993), (WILKINSON; ALLEN, 1999), (MILLER; BOXER, 2005).

3.2.1.1.2 SIMD – Single Instruction, Multiple Data Stream

Esta é a classificação dada a computadores com mais de um processador, com memória local, uma unidade de controle e conectado a uma rede de interconexão (PITAS, 1995), (MILLER; BOXER, 2005).

Nesta arquitetura a unidade de controle armazena o programa e envia as instruções para todas as unidades de processamento simultaneamente, em um único ciclo de relógio (clock). A Figura 6 apresenta, de forma simplificada, o esquema de uma máquina construída sob a abordagem da arquitetura SIMD.

Observe que há um único fluxo de instruções operando sobre múltiplos conjuntos de dados (CODENOTTI; LEONCINI, 1993), (WILKINSON; ALLEN, 1999), (MILLER; BOXER, 2005).

Figura 6 – Arquitetura para a taxonomia de Flynn SIMD.

Fontes: (MATTSON *et al.*, 2005), (STALLINGS, 2012, p.517) e outras.

Pode-se ver claramente, nesta arquitetura, que todas as unidades de processamento executam a mesma instrução simultaneamente, porém sobre o mesmo conjunto de dados, na memória local. Esta arquitetura é usada na construção de topologias de redes tais como malha (mesh), pirâmide e hipercubo. O aumento de desempenho é ganho estritamente usando-se mais unidades de processamento no modelo (FLYNN, 1966), (WILKINSON; ALLEN, 1999), (MILLER; BOXER, 2005).

3.2.1.1.3 MISD – Multiple Instruction, Single Data Stream

Esta estrutura tem recebido muito pouca atenção, já que não faz muito sentido (FLYNN, 1966), (MILLER; BOXER, 2005). Pois ela emprega uma alta largura de banda dedicada à unidade de execução que é compartilhada por n máquinas virtuais sobre sequências do programa independentes umas das outras. Cada máquina virtual tem acesso à execução apenas uma vez por ciclo do hardware. Cada máquina virtual tem sua memória privada de instruções e a interação entre os fluxos de instrução ocorre somente através da memória de dados comum *apud* Flynn (1966).

A Figura 7 apresenta de forma simplificada o esquema de uma máquina construída sob a abordagem da arquitetura MISD, neste caso se tem múltiplos fluxos de instruções operando sobre um único conjunto de dados.

Figura 7 – Arquitetura para a taxonomia de Flynn MISD.

Fontes: (MATTSON *et al.*, 2005), (STALLINGS, 2012, p.517) e outras.

3.2.1.1.4 MIMD – Multiple Instruction, Multiple Data Stream

Conforme Flynn (1966) uma máquina de arquitetura MIMD pode ser obtida a partir de uma máquina com arquitetura MISD mantendo-se os fluxos de dados e de instruções juntos em memórias próprias. Dessa forma, uma máquina MIMD consiste de um conjunto de unidades de processamento com memória local e uma rede de interconexão.

Na verdade, é bastante comum que um algoritmo seja implementado de forma tal que todos os processadores executem o mesmo programa (PITAS, 1995). Nota-se que nesta arquitetura todos os módulos são independentes e capazes de execução concorrente. Assim, MIMD significa que existem múltiplos fluxos de instruções operando sobre múltiplos conjuntos de dados. A Figura 8 ilustra a arquitetura da taxonomia de Flynn MIMD.

Entretanto, há algumas dificuldades inerentes ao modelo MIMD, que geralmente incluem a interconexão entre as unidades causando sérios problemas de interferência e a natureza universal dos módulos individuais que limita sua eficiência, *apud* Flynn (1966).

Figura 8 – Arquitetura da taxonomia de Flynn MIMD.

Fontes: (MATTSON *et al.*, 2005), (STALLINGS, 2012, p.517) e outras.

3.2.1.2 Pipeline

A sobreposição de instruções na execução é uma técnica de implementação conhecida como **pipelining**. Essa técnica é fundamental no projeto dos processadores atuais, pois os tornam mais rápidos (PATTERSON; HENNESSY, 2005).

Uma boa maneira de se entender como a técnica de pipeline funciona é através da analogia com a linha de montagem em uma fábrica (GRAMA *et al.*, 2003). No entanto, Patterson e Hennessy (2005) fazem uma analogia com o processo aplicado em uma lavanderia de roupas. Se todos os estágios de execução necessitarem, aproximadamente, do mesmo tempo e tiver trabalho suficiente a ser realizado, então o ganho de velocidade (speedup) devido à técnica de pipelining será igual ao número de estágios do pipeline (CODENOTTI; LEONCINI, 1993) , (WILKINSON; ALLEN, 1999), (GRAMA *et al.*, 2003), (HENNESSY; PATTERSON, 2008), (TANENBAUM, 2012).

Ainda, conforme Codenotti e Leoncini (1993) e, Wilkinson e Allen (1999) na técnica de pipelining o problema é dividido em uma série de tarefas que têm de ser completadas uma após a outra, já que essa é baseada na programação sequencial. Assim, em pipelining cada tarefa será executada por um processo ou processador ou um thread como mostrado na Figura 9.

Figura 9 – Pipeline linear com n processos.

Fonte: (HWANG, 1993). Imagem semelhante à original, reproduzida apenas para efeitos didático.

Dessa forma, cada processo de pipeline pode ser referenciado como um *estágio* de pipeline. Logo, cada estágio contribuirá para o problema todo passando para os próximos estágios a informação necessária. Esse tipo de paralelismo pode ser visto como uma forma de decomposição funcional, como é o caso da técnica de Dividir e Conquistar.

De acordo com Hwang (1993) a técnica de pipelining pode ser linear, como visto na Figura 10, ou não linear e, dependendo do controle do fluxo dos dados ao longo do pipeline o modelo linear pode ser dividido em duas categorias: pipeline síncrono e pipeline assíncrono.

Ainda, conforme Hwang (1993) o modelo de pipeline síncrono, ilustrado na Figura 10, usa latches sincronizados como interfaces entre os estágios. Ao receber um pulso de controle de sincronismo, todos os latches transferem os dados, simultaneamente, para o estágio seguinte.

Figura 10 – Modelo de pipeline síncrono.

Fonte: (HWANG, 1993, p.266). Imagem semelhante à original, reproduzida apenas para efeito didático.

No modelo de pipeline assíncrono, vide Figura 11, o fluxo de dados entre os estágios adjacentes é controlado por um protocolo “aperto de mão” (*handshaking*), isto é, quando o estágio S_i está pronto para transmitir, ele envia um sinal de controle “pronto” (*ready*) para o estágio seguinte S_{i+1} .

Figura 11 – Modelo de pipeline assíncrono.

Fonte: (HWANG, 1993, p.266). Imagem semelhante à original, reproduzida apenas para efeito didático.

Após o estágio S_{i+1} receber os dados do estágio anterior, ele retorna um sinal “reconhecido” (*acknowledge*) para o estágio S_i . Os pipelines assíncronos são

potencialmente usados no desenvolvimento de canais de comunicação com passagem de mensagem (message-passing) de sistemas multicomputadores (HWANG, 1993).

A Figura 12 ilustra a estrutura de um pipeline não linear e a Figura 13 apresenta a ideia chave do pipeline baseado no padrão de linha de montagem usado como analogia, anteriormente para um pipeline de quatro estágios (CODENOTTI; LEONCINI, 1993), (HWANG, 1993), (WILKINSON; ALLEN, 1999), (MATTSON *et al.*, 2005), (GRAMA *et al.*, 2003), (HENNESSY; PATTERSON, 2008), (STALLINGS, 2012), (TANENBAUM, 2012).

Figura 12 – Processos em um pipeline não linear.

Fonte: (HWANG, 1993).

Cada bloco P_i na Figura 13 representa um multipasso da computação realizada sobre o i -ésimo elemento de dados. $P_i(j)$ é o j -ésimo passo do processamento. A ideia aqui é mapear os passos de processamento para os estágios do pipeline em que cada estágio de pipeline executa um passo.

Figura 13 – Pipeline de quatro estágios, onde cada i -ésimo estágio executa o j -ésimo passo do processo.

Fonte: (Imagem compilada de diversas ilustrações apresentadas em vários textos. Reproduzida aqui apenas para efeito didático.)

Inicialmente, o primeiro estágio do pipeline executa $P_0(0)$. Após esse estágio ser completado, o segundo estágio recebe os dados do primeiro estágio e executa $P_0(1)$, enquanto o primeiro estágio executa o primeiro passo do segundo item, $P_1(0)$. A seguir, o terceiro estágio executa $P_0(2)$, enquanto que o segundo estágio executa $P_1(1)$ e o primeiro estágio $P_2(0)$ e, assim sucessivamente, até que o último estágio $P_3(4)$ seja executado. Esta ideia pode ser aplicada tanto para o modelo de pipeline linear, quanto para o não linear.

Hwang (1993) aborda o tema nos níveis de hardware e instruções, enquanto que Hennessy e Patterson (2008), Tanenbaum (2012) e Stallings (2012) abordam o mesmo assunto no nível de instruções. Wilkinson e Allen (1999), Mattson *et al.* (2005) e Scott (2009) tratam o assunto na programação de alto nível apresentando pequenos fragmentos de código em MPI e JAVA. Já Codenotti e Leoncini (1993) referem-se apenas a alguns computadores de grande porte que adotaram essa técnica no nível do software no passado.

No geral, a técnica de pipelining é híbrida já que se beneficia dos modelos sequencial e paralelo quase que simultaneamente, elevando dessa forma o ganho de velocidade (speedup) dos processadores e computadores atuais.

Apesar das dificuldades inerentes à técnica de pipelining, sua exploração e aperfeiçoamento no nível de software de alto nível são perfeitamente factíveis.

3.2.1.3 Computação em Grade (*Grid Computing*)

A computação em grade é uma arquitetura de computação distribuída e compartilhamento de recursos. Um sistema de grade é composto por uma coleção heterogênea de recursos conectados por LAN e/ou WAN (geralmente a Internet).

Esses recursos individuais são gerais e incluem servidores de computação, armazenamento, servidores de aplicativos, serviços de informação, ou até mesmo instrumentos científicos. Grades são frequentemente implementadas em termos de serviços Web e componentes de middleware integrados, que fornecem uma interface consistente para a grade.

A grade é diferente de um cluster, pois os recursos em uma grade não são controlados através de um único ponto de administração, o middleware de grade administra o sistema para controle de recursos na grade e as políticas que regem o uso dos recursos permanecem com os proprietários dos recursos. Atualmente, três excelentes exemplos de computação em grade são o projeto *Globus Genomics*², o projeto *SETI@Home*³, o projeto *World Community Grid*⁴ e muitos outros, de onde é possível se baixar um programa executável para que após sua instalação no computador pessoal, o mesmo torna-se membro (ou nó) da grade, dessa forma contribuindo para o processamento dos dados referentes ao respectivo projeto.

3.2.1.4 Hipercubo

Hipercubo é essencialmente uma rede de interconexão, mas também é considerado uma estrutura de multicomputador em que os nós são colocados nos vértices de um cubo d -dimensional. A configuração mais frequentemente utilizada é um hipercubo binário em que cada um dos nós de 2^n está ligado a outros n nós. A Figura 14, a seguir mostra uma configuração básica de uma estrutura computacional em hipercubo de oito nós, onde as setas representam o sentido do fluxo de dados através da rede.

² Fornece aos pesquisadores de biociências uma plataforma avançada, para ajudar no combate aos crescentes desafios associados com a análise de dados genômicos. - <http://www.globus.org/genomics/>

³ Busca por Inteligência Extraterrestre (*Search for Extra-Terrestrial Intelligence*) através da análise de sinais de Radiofrequência obtidos por Radiotelescópios na Terra - <http://setiathome.ssl.berkeley.edu/>

⁴ Busca avançar os conhecimentos sobre doenças humanas (em especial o Câncer e a AIDS) - <http://www.worldcommunitygrid.org/>

Figura 14 – Hipercubo binário tridimensional de oito nós.

Fontes: (WILKINSON; ALLEN, 2003), (GRAMA, 2003) e outras.

Alguns excelentes textos a respeito deste assunto podem ser encontrados em (WILKINSON; ALLEN, 1999), (GRAMA *et al.*, 2003), (MATTSON *et al.*, 2005), (TANENBAUM, 2012), (STALLINGS, 2012).

3.3.2 Software

A programação paralela requer uma plataforma de computação adequada, ou seja, um sistema computacional constituído por um único computador com vários processadores internos ou vários computadores interligados através de uma rede de comunicação, ou nos casos mais atuais, computadores dotados de processadores com CPUs de múltiplos núcleos e até com GPGPUs com centenas ou milhares de núcleos. Com isso, o software paralelo deve solucionar um dado problema distribuindo os dados entre as diversas unidades de processamento internas ou entre os diversos computadores da rede (CODENOTTI; LEONCINI, 1993), (WILKINSON; ALLEN, 1999).

O desenvolvimento de software paralelo tem sido tradicionalmente considerado como tempo e esforço intensivos. Isso pode ser atribuído em grande parte à complexidade inerente na especificação e coordenação de tarefas simultâneas, a falta de algoritmos portáteis, ambientes padronizados e ferramentas de desenvolvimento de software (CODENOTTI; LEONCINI, 1993), (GRAMA *et al.*, 2003), (MATTSON *et al.*, 2005).

A computação paralela tem gerado grande impacto numa variedade de áreas do conhecimento, que vão desde simulações computacionais para aplicações científicas e de engenharia até aplicações comerciais em processamento de transações e mineração de dados (**data mining**) (GRAMA *et al.*, 2003), (MILLER; BOXER, 2005).

Entretanto, nos dias atuais, com a redução dos preços dos processadores com múltiplos núcleos e das placas gráficas com muitos núcleos (GPGPU) programáveis, os computadores pessoais de mesa tornaram-se *máquinas paralelas*⁵ de “alto desempenho”⁶ a um baixíssimo custo, possibilitando o uso intenso de software paralelo em todas as camadas da sociedade⁷. Todavia, a computação paralela vem alcançando cada vez mais áreas do conhecimento, principalmente a de games que é um mercado muito exigente quanto às tecnologias empregadas, falando-se de computadores pessoais de pequeno porte. Em computadores de grande porte essa prática⁸ é bastante comum há várias décadas.

3.3.2.1 Fator ganho de velocidade (*Speedup Factor*)

Há três modelos para se medir a velocidade do aumento de desempenho, que são a lei de Amdahl (1967) a qual se baseia no volume fixo de trabalho ou o tamanho fixo do problema. A lei de Gustafson (1987) que é aplicada a problemas de escala, onde o tamanho do problema aumenta com o porte da máquina. E, por fim a lei de Sun e

⁵ Apesar dos computadores pessoais atuais terem atingido um grau de sofisticação bastante elevado – falando-se da capacidade de computação paralela, a grande maioria dos aplicativos ainda continua sendo desenvolvida com base no antigo paradigma da programação sequencial, desperdiçando-se, assim grande parte desse potencial disponibilizado.

⁶ Alto desempenho foi colocado entre aspas para indicar que essa categoria de computadores tornou-se muito mais potencial em termos de capacidade de processamento em relação aos seus predecessores, que eram especificamente sequenciais desde seu lançamento na década de 1970, mas não em relação aos supercomputadores que normalmente recebem essa nomenclatura há décadas.

⁷ Devido o elevado grau de complexidade que a computação paralela oferece, infelizmente o desenvolvimento de software paralelo ainda não está difundido nas camadas menos especializadas da sociedade, embora esse cenário esteja mudando vagarosamente.

⁸ O uso do paradigma da programação paralela.

Ni (1993) onde se aplicam em problemas de escala limitada pela capacidade da memória (HWANG, 1993).

Entretanto, a medida relativa do desempenho entre um sistema monoprocessador e um sistema multiprocessador é o chamado **fator ganho de velocidade** ou (do inglês: *speedup factor*) $S(n)$, definido como:

$$S(n) = \frac{\text{tempo de execução usando um processador (sistema monoprocessador)}}{\text{tempo de execução usando um multiprocessador com } n \text{ CPUs}}$$

que nos leva à eq.(1).

$$S(n) = \frac{t_s}{t_p} \quad (1)$$

onde t_s é o tempo de execução sequencial e t_p é o tempo de execução paralela do mesmo algoritmo em um computador monoprocessador e multiprocessador, respectivamente.

O fator ganho de velocidade também pode ser expresso em termos do número de passos computacionais, como:

$$S(n) = \frac{\text{número passos computacionais usando um processador}}{\text{número de passos computacionais paralelos com } n \text{ CPUs}}$$

(CODENOTTI; LEONCINI, 1993), (WILKINSON; ALLEN, 1999), (MILLER; BOXER, 2005), (MATTSON *et al.*, 2005), (HENNESSY; PATTERSON, 2008), (KIRK; HWU, 2010), (FARBER, 2011).

3.3.2.2 Custo Computacional

O custo ou a complexidade computacional é um estudo muito importante no desenvolvimento de algoritmos, o qual visa medir o tempo de execução do algoritmo, independentemente do hardware utilizado para sua execução.

Conforme Wilkinson e Allen (1999) o produto *processador-tempo* ou *custo* da execução de um algoritmo pode ser definido por:

$$Custo = (\text{tempo de execução}) * (\text{número total de processadores usados}).$$

O custo de execução de um algoritmo sequencial é simplesmente seu tempo de execução t_s , enquanto que o custo de execução de um algoritmo paralelo é $t_p * n$, onde t_p é o tempo de execução do algoritmo paralelo dado por $t_s / S(n)$ e, n é o número de processadores no sistema multiprocessador. Assim, o custo de execução de um algoritmo paralelo é dado pela eq.(2).

$$Custo = \frac{t_s * n}{S(n)} = \frac{t_s}{E} \quad (2)$$

Na eq.(2) E é a eficiência do sistema, definida por:

$$E = \frac{\text{tempo de execução usando um processador}}{\text{tempo de execução usando um multiprocessador} * \text{nr. processadores}}$$

que, de acordo com (WILKINSON; ALLEN, 1999) é escrita conforme mostrado na eq.(3), a seguir:

$$E = \frac{t_s}{t_p * n} = \frac{S(n)}{n} * 100\% \quad (3)$$

e $S(n)$ é o fator de ganho de velocidade, conforme apresentado na eq.(1).

3.3.2.3 Lei de Amdahl

A lei de Amdahl normalmente é aplicada para medir a eficiência de um sistema quando a carga de trabalho é fixa. Em muitas aplicações práticas, que demandam resposta em tempo real, a carga de trabalho computacional é frequentemente fixa com problemas de tamanho fixo. Assim, quando se aumenta o número de processadores em um computador paralelo, a carga fixa é distribuída para mais processadores para ser executada em paralelo. Portanto, o principal objetivo é

produzir os resultados o mais rápido possível. Em outras palavras, minimizar o tempo de resposta é o objetivo principal *apud* Hwang (1993).

Gene Amdahl propôs em 1967 um método para medir o potencial do ganho de velocidade de um programa de computador, usando um único processador em comparação com o uso de múltiplos processadores (STALLINGS, 2012). É possível se obter um ganho no desempenho melhorando apenas alguns elementos de um computador, assim a métrica utilizada para medir esse aumento de desempenho pode ser a lei de Amdahl (HENNESSY; PATTERSON, 2008).

Ainda, conforme Hennessy e Patterson (2008) o speedup pode ser obtido se usando algum recurso em particular. Então, faz-se a seguinte pergunta: “O que é ganho de velocidade (*speedup*)?”. De acordo com Amdahl a resposta é a seguinte: O speedup é a razão:

$$speedup = \frac{\text{desempenho para a tarefa inteira usando a melhoria, quando possível}}{\text{desempenho para a tarefa inteira sem usar a melhoria}}$$

(HENNESSY; PATTERSON, 2008), (STALLINGS, 2012).

Apesar da lei de Amdahl ser um método rápido para se medir o speedup a partir de alguma melhoria, ela depende de dois fatores (HENNESSY; PATTERSON, 2008):

1. A fração do tempo de computação no computador original pode ser convertida para tirar proveito da melhoria.
2. A melhoria ganha pelo modo de execução melhorado, ou seja, o quanto mais rápido a tarefa seria executada se o modo melhorado fosse usado para o programa inteiro.

$$speedup = \frac{\text{tempo de execução de um programa num único processador}}{\text{tempo de execução de um programa em N processadores paralelos}} = \frac{t_s}{t_p} \quad (4)$$

Na eq.(4) tem-se que: t_s – é o tempo necessário para a execução do algoritmo em um computador sequencial e, t_p – é o tempo gasto por um computador paralelo, para executar o mesmo algoritmo (WILKINSON; ALLEN, 1999), (MILLER; BOXER, 2005),

(FARBER, 2011), (KIRK; HWU, 2010), (MATTSON *et al.*, 2005), (HENNESSY; PATTERSON, 2008).

A eq.(5) traduz a lei de Amdahl:

$$speedup = \frac{s + p}{s + \frac{p}{n}} = \frac{1}{s + \frac{p}{n}} = \frac{1}{s + \frac{(1-s)}{n}} = \frac{n}{s * n + (1-s)} \quad (5)$$

onde se tem: s é o tempo de execução da parte sequencial do algoritmo, p o tempo de execução da parte paralela do mesmo algoritmo e n o número de processadores (WILKINSON; ALLEN, 1999). A tabela 3.1 apresenta o *speedup*, conforme a lei de Amdahl para alguns valores de n e p .

Tabela 3.1: Speedup conforme lei de Amdahl para alguns valores de n (número de processadores).

n	$p = 0,10$	$p = 0,50$	$p = 0,90$	$p = 0,99$
10	1,10	1,82	5,26	9,17
100	1,11	1,98	9,17	50,25
1000	1,11	1,99	9,91	90,99
10000	1,11	1,99	9,99	99,02

Os valores de p são dados em porcentagem de processamento paralelo na aplicação, enquanto que n é o número de processadores que estão executando a aplicação, concorrentemente.

3.3.2.4 Lei de Gustafson

A lei de Gustafson está relacionada à escalabilidade de um sistema (HWANG, 1993), (WILKINSON; ALLEN, 1999). Entretanto, conforme apresentado em (WILKINSON; ALLEN, 1999) e (HWANG, 1993) Gustafson apresentou em 1988 um argumento baseado em conceitos de escalabilidade, para mostrar que a lei de Amdahl não foi tão significativa como suposta em um primeiro momento, para limitar o aumento potencial do *speedup* (Gustafson, 1988) apud Wilkinson e Allen (1999).

Gustafson observou que, na prática, um multiprocessador maior geralmente permite maior dimensão do problema a ser realizado num tempo de execução razoável.

Usando como restrição o tempo de execução paralela constante, o fator ganho de velocidade resultante será numericamente diferente do fator ganho de velocidade da lei de Amdahl e é chamado fator ganho de velocidade escalonado.

A eq.(6) é conhecida como lei de Gustafson:

$$S_s(n) = \frac{s + n * p}{s + p} = s + n * p = n + (1 - n) * s \quad (6)$$

onde s é o tempo de execução da parte serial do algoritmo, p é o tempo de execução da parte paralela do mesmo algoritmo e n é o número de processadores (WILKINSON; ALLEN, 1999).

A tabela 3.2 apresenta a comparação entre o *speedup* de acordo com a lei Amdahl e a lei de Gustafson, para alguns valores de n e p .

Tabela 3.2: Comparação do speedup conforme a lei de Amdahl $A(n)$ e a lei de Gustafson $G(n)$, para n processadores e p percentual de processamento paralelo em uma aplicação.

n	$p = 0,10$		$p = 0,50$		$p = 0,90$		$p = 0,99$	
	$A(n)$	$G(n)$	$A(n)$	$G(n)$	$A(n)$	$G(n)$	$A(n)$	$G(n)$
10	1,10	1,90	1,82	5,50	5,26	9,10	9,17	9,91
100	1,11	10,90	1,98	50,50	9,17	90,10	50,25	99,01
1000	1,11	100,90	1,99	500,50	9,91	900,10	90,99	990,01
10000	1,11	1000,90	1,99	5000,50	9,99	9000,10	99,02	9900,01

Os valores de p são dados em porcentagem de processamento paralelo na aplicação, enquanto que n é o número de processadores que estão executando a aplicação concorrentemente.

3.3 Técnicas de Computação Paralela

Na computação paralela podem ocorrer diversas situações inesperadas, levando o sistema ao caos e até mesmo ao seu travamento. Quando se trata de programação em memória compartilhada, ou seja, com o compartilhamento de dados, esses problemas se tornam mais evidentes e, na maioria das vezes difíceis de ser resolvidos.

Felizmente, existem diversas técnicas já consagradas no meio acadêmico desenvolvidas especificamente para se contornar essas situações. Algumas dessas técnicas são apresentadas, sem o devido formalismo, nos itens que se seguem, mas excelentes textos avançados podem ser consultados nas referências citadas.

Portanto, o aspecto-chave da programação usando memória compartilhada é que esse modelo de memória oferece a possibilidade da criação de variáveis e estruturas de dados que podem ser acessados diretamente por cada processador, em vez de ter que passar os dados por mensagens, como nos ambientes de transferência de mensagens (WILKINSON; ALLEN, 1999).

3.3.1 Sincronismo

Em um programa paralelo, devido ao não sincronismo de agendamento de tarefas e outros fatores, os eventos podem ocorrer em qualquer ordem não definida. Por exemplo, numa execução, uma tarefa pode ler a variável x antes de outra tarefa ler a variável y , na próxima execução de leitura os eventos podem ocorrer na ordem inversa. Em muitos casos, a ordem em que dois eventos ocorrem não importa. Entretanto, em situações de ordem, para garantir que o programa esteja correto, o programador deve introduzir sincronização para fazer valer as restrições de ordenação necessárias (MATTSON *et al.*, 2005).

3.3.2 Condições de Corrida (*Race Conditions*)

A condição de corrida é um tipo de erro peculiar que ocorre em sistemas operacionais e programas paralelos. Como o sistema operacional e não o programador controla o agendamento das UEs, condições de corrida resultam em programas que dão respostas diferentes potencialmente, mesmo quando executados no mesmo sistema com os mesmos dados. Condições de corrida são erros particularmente difíceis de depurar porque, por sua natureza, não podem ser reproduzidos de forma confiável. Os testes ajudam, mas não são tão eficazes como com programas sequenciais: por exemplo, um programa pode ser executado corretamente as primeiras mil vezes e depois falhar catastroficamente na milésima

primeira execução - e, em seguida, executa corretamente quando o programador tenta reproduzir o erro como primeiro passo na depuração (MATTSON *et al.*, 2005), (CHAPMAN *et al.*, 2008).

As condições de corrida também podem resultar de erros de sincronismo, por exemplo, se várias UEs lerem e escreverem em variáveis compartilhadas, o programador deve proteger o acesso a essas variáveis de modo que a leitura e escrita ocorram em uma ordem válida, independentemente de como as tarefas sejam intercaladas.

Quando muitas variáveis são compartilhadas ou quando elas são acessadas através de vários níveis indiretos, pode ser muito difícil verificar por inspeção se não existir condições de corrida.

A depuração de programas contendo condições de corrida é muito difícil e pouco conveniente, pois apesar de na maior parte das vezes os testes apresentarem bons resultados, de vez em quando algo inesperado e inexplicável acontece *apud* Tanenbaum e Woodhull (2002).

Para Chapman *et al.* (2008), um dos maiores inconvenientes da programação paralela usando memória compartilhada é a possibilidade disso levar à introdução de um certo tipo de erro, que se manifesta por meio da corrupção de dados em silêncio. Para piorar a situação, o comportamento de tempo de execução de código com esse tipo de erro também não é reproduzível, isto é, se o mesmo programa é executado pela segunda vez, o problema pode não aparecer.

OpenMP tem vários mecanismos de segurança para ajudar a evitar esse tipo de erro. Mas, OpenMP não pode impedir a sua inserção, uma vez que é tipicamente um resultado do uso inadequado das directivas da linguagem. Por exemplo, o erro pode surgir a partir da paralelização incorreta de um laço ou uma atualização sem proteção dos dados compartilhados (CHAPMAN *et al.*, 2008).

3.3.3 Impasse (*Deadlock*)

Os conflitos são outro tipo de erros inerentes a programas paralelos. Um impasse ocorre quando existe um ciclo de tarefas, no qual cada tarefa é bloqueada à espera de outra para prosseguir. Como todas estão à espera do término de alguma outra tarefa para fazer algo, todas elas serão bloqueadas indefinidamente. Por exemplo, considere duas tarefas em um ambiente de passagem de mensagens, onde a tarefa *A* tenta receber uma mensagem da tarefa *B*, após a qual essa irá responder enviando uma mensagem própria para a tarefa *B*. Enquanto isso, a tarefa *B* tenta receber uma mensagem da tarefa *A*, depois *B*, que irá enviar uma mensagem para *A*, pois cada tarefa está esperando pelo final da outra para enviar-lhe uma mensagem, assim ambas as tarefas serão bloqueadas para sempre. Felizmente, os impasses não são difíceis de descobrir, já que as tarefas vão parar no ponto do impasse.

Dessa forma, quando dois processos permanecem bloqueados indefinidamente e mais nenhum trabalho pode ser realizado por eles, diz-se que estão em impasse ou deadlock. O impasse pode ser definido formalmente como segue, *apud* Tanenbaum e Woodhull (2002):

“Um conjunto de processos está em um impasse se cada processo no conjunto está esperando um evento que somente outro processo no conjunto pode causar.”.

Ainda, de acordo com Tanenbaum e Woodhull (2002) há quatro condições⁹ que devem ser sustentadas para ocorrer um impasse, demonstradas por Coffman e outros colaboradores (1971), assim temos: **Condição de:**

1. **exclusão mútua:** todo recurso está disponível ou está atribuído unicamente a um processo.

⁹ As notas foram extraídas originalmente de (TANENBAUM; WOODHULL, 2002), mas sofreram algumas alterações sem perder o seu teor.

2. **segura e espera:** todo processo que estiver segurando algum recurso concedido anteriormente pode solicitar novos recursos.
3. **nenhuma preempção:** recursos previamente concedidos a um processo não podem ser tirados; eles devem ser explicitamente liberados pelo processo que os mantém.
4. **espera circular:** deve haver uma sequência de ao menos dois processos, em que cada qual esteja esperando um recurso segurado pelo membro seguinte da sequência.

Se ao menos uma dessas quatro condições não estiver presente, então nenhum impasse ocorrerá.

A Figura 15 mostra uma situação de impasse, onde **R1** e **R2** são os recursos e **P1** e **P2** são os processos.

Figura 15 – Situação de impasse envolvendo dois processos e dois recursos.

Fontes: (TANENBAUM; WOODHULL, 2002, p. 123) e outras.

O grafo orientado da Figura 15 ilustra um impasse e tem o seguinte significado: o processo **P1** está esperando o recurso **R1**, que está sendo segurado pelo processo **P2**, o qual por sua vez não o liberará, já que o mesmo está esperando o recurso **R2**, que está sendo segurado por **P1**. Dessa forma, ambos os processos **P1** e **P2** esperarão indefinidamente um pelo recurso do outro.

Ainda, em (TANENBAUM; WOODHULL, 2002) são apresentados algoritmos para detectar, recuperar, prevenir e impedir a ocorrência de impasses.

3.3.4 Semáforo

Semáforos são utilizados para a sincronização de processos que normalmente são implementados nos sistemas operacionais (WILKINSON; ALLEN, 1999), (TANENBAUM; WOODHULL, 2002), (BEN-ARI, 2006), (TANENBAUM, 2012) e podem ser usados na solução do problema da seção crítica para dois processos de forma trivial *apud* Ben-Ari (2006).

Um semáforo tem um valor do tipo inteiro não negativo e duas operações atômicas. As operações permitidas são *V* (às vezes chamado de **up**) e *P* (às vezes chamado de **down**). A operação *V* aumenta o valor do semáforo de uma unidade. A operação *P* diminui o valor do semáforo de uma unidade, desde que possa ser feito sem violar a restrição de negatividade do valor, propôs Dijkstra (1968) em seu novo tipo de variável de controle de processos chamado **semáforo** (WILKINSON; ALLEN, 1999), (WALMSLEY, 2000), (TANENBAUM; WOODHULL, 2002), (BEN-ARI, 2006), (TANENBAUM, 2012).

Com os semáforos a comunicação inter processo parece fácil, mas não é *apud* Tanenbaum e Woodhull (2002), pois se uma análise cuidadosa não for feita no decorrer dos eventos envolvidos nesse procedimento, pode-se incorrer em impasse (deadlock). A solução para tais situações é utilizar-se das variáveis de condição (WILKINSON; ALLEN, 1999), (WALMSLEY, 2000), (TANENBAUM; WOODHULL, 2002).

3.3.5 Monitor

Os monitores são mecanismos de sincronização originalmente proposto por Hoare (1974). Um monitor é uma implementação ADT (tipo abstrato de dados) que garante o acesso mutuamente exclusivo aos seus dados internos. A sincronização condicional é fornecida pelas variáveis de condição. Os Monitores são amplamente reconhecidos como semáforos, embora sejam capazes de implementar muitas aplicações de seção crítica, eles estão disponíveis para uso em erros humanos (WILKINSON; ALLEN, 1999), (MATTSON *et al.*, 2005).

Os monitores fornecem uma programação concorrente estruturada primitiva que concentra a responsabilidade na correção de módulos computacionais. Eles são uma generalização do kernel ou supervisor encontrado em sistemas operacionais, onde as seções críticas, como a alocação de memória são centralizadas em um programa privilegiado. Programas aplicativos solicitam serviços que são executados pelo kernel. Os kernels são executados em um modo de hardware que garante que eles não recebam interferências de programas aplicativos (TANENBAUM; WOODHULL, 2002), (BEN-ARI, 2006).

Os monitores tornaram-se um mecanismo de sincronização de extrema importância, pois eles são uma generalização natural do *objeto* da programação orientada a objetos, que encapsula os dados e declarações de operação dentro de uma classe (BEN-ARI, 2006).

3.3.6 Exclusão Mútua (*Mutex*)

Os comandos provenientes de seções críticas de dois ou mais processos não devem ser intercalados (BEN-ARI, 2006), (HERLIHY; SHAVIT, 2008).

O problema de acessar dados compartilhados pode ser generalizado considerando-se recursos compartilhados. Além disso, para dados compartilhados o recurso poderia também ser um dispositivo físico tal como um dispositivo de entrada / saída.

Uma vez que um processo tenha terminado a sua seção crítica, é permitido a outro processo entrar na seção crítica para o mesmo recurso. Em um bloqueio de exclusão mútua, o mutex tem a função de serializar a execução de múltiplos threads.

Quando o semáforo é do tipo binário, às vezes, ele é chamado de *mutex*. O termo mutex é muito usado em pthreads e JAVA (HERLIHY; SHAVIT, 2008).

3.3.7 Balanceamento de Carga

O balanceamento de carga é o processo de alocação de trabalho para elementos de processamento, seja estática ou dinâmica, de modo que o trabalho seja distribuído

mais uniformemente possível entre os diversos elementos de processamento do sistema computacional.

Para executar-se um programa paralelo, as tarefas devem ser mapeadas para as unidades de execução, e para as unidades de execução de elementos de processamento. A forma como o mapeamento é feito pode resultar em um impacto significativo no desempenho global de um algoritmo paralelo. É crucial evitar a situação em que um subconjunto de elementos de processamento esteja realizando a maior parte do trabalho, enquanto outros ficam inativos, ou seja, o desbalanceamento da carga no sistema.

3.3.8 Análise de dependência de dados

Em computação é muito comum a ocorrência de dependência de dados, tarefas e processos, entretanto há situações em que isso não ocorre. Normalmente, é muito difícil e até mesmo, em diversos casos, impossível a eliminação dessas dependências. No entanto, existem técnicas que podem nos ajudar a minimizá-las.

A análise de dependência é muito importante para a computação paralela, pois se bem feita ela poderá resultar em melhorias significativas no desempenho de um algoritmo. De acordo com Wilkinson e Allen (1999) um dos assuntos chave em toda a programação paralela é identificar quais processos poderiam ser executados juntos, pois processos que tiverem alguma dependência entre eles não podem ser executados juntos, já que precisam ser executados em sequência.

De acordo com Mattson *et al.* (2005) dependências entre tarefas têm um grande impacto sobre o projeto de algoritmos emergentes e há duas categorias de dependências, que são a ordenação de restrições e as dependências relacionadas aos dados compartilhados. Dependências de dados compartilhados são potencialmente mais complicadas. No caso mais simples, não há dependências entre as tarefas. Quando os dados são compartilhados entre várias tarefas, o algoritmo pode ser muito mais complicado, embora ainda existam alguns casos

comuns que podem ser tratados de forma relativamente fácil. As dependências podem ser classificadas da seguinte forma:

1. **Dependências removíveis:** Neste caso, a dependência não é uma verdadeira dependência entre tarefas, mas uma dependência aparente que pode ser removida por simples transformações no código. O caso mais simples ocorre quando se tem uma variável temporária cujo uso é completamente local para cada tarefa, dessa forma, com cada tarefa inicializando a variável sem referência a outras tarefas. Em casos mais complicados como em expressões iterativas, talvez seja necessário transformar essas expressões na sua forma fechada para remover a dependência de laço.
2. **Dependências "separáveis":** Quando as dependências envolvem acumulação numa estrutura de dados comum, elas podem ser separadas a partir das tarefas - "puxado para fora da computação concorrente" - por replicar a estrutura de dados no início da execução, executar as tarefas, e depois combinar as cópias numa estrutura de dados única. Muitas vezes, a acumulação é uma operação de redução em que um conjunto de elementos de dados é reduzido a um único elemento através da aplicação repetida de uma operação binária, tal como a adição ou multiplicação.

Em mais detalhes, estas dependências podem ser gerenciadas como segue: uma cópia da estrutura de dados usada na acumulação é criada em cada uma das UEs. Cada cópia é inicializada (no caso de uma redução, um elemento binário para identificar cada operação – por exemplo, zero para a adição e um para a multiplicação). Cada tarefa, em seguida, realiza a acumulação em sua estrutura de dados local, eliminando a dependência de dados compartilhados. Quando todas as tarefas estão concluídas, as estruturas de dados locais em todas as UEs são combinadas para produzir o resultado final global (no caso de uma redução, aplicando a operação binária novamente).

Para Aho *et al.* (1986, 2008) há três tipos de dependências de dados, que são:

1. **Dependência verdadeira:** ler após escrever. Para uma operação de escrita seguida de uma operação de leitura de um mesmo endereço, a leitura depende do valor escrito.
2. **Antidependência:** escrever após ler. Para uma leitura seguida de uma escrita no mesmo endereço significa que há uma antidependência da leitura para a escrita, ou seja, a escrita não depende da leitura em si, mas se a escrita acontecer antes da leitura, a operação de leitura obterá um valor errado. Por não ser uma dependência verdadeira, a antidependência pode ser facilmente eliminada armazenando-se os valores em diferentes endereços de memória.
3. **Dependência de saída:** escrever após escrever. Quando duas escritas ocorrem no mesmo endereço, diz-se que elas compartilham uma dependência de saída, entretanto se a dependência for violada, o valor do endereço de memória escrito terá o valor errado após as duas operações serem efetuadas.

Por serem muito comuns, tanto MPI quanto OpenMP fornecem suporte para as reduções como parte de suas APIs (WILKINSON; ALLEN, 1999), (MATTSON *et al.*, 2005), (CHAPMAN *et al.*, 2008). Maiores detalhes sobre análise de dependência de dados podem ser encontrados nas diversas referências citadas ao longo deste tópico, bem como em muitas outras não citadas, tornando o assunto bastante comum na literatura.

As dependências de dados impedem de se fazer paralelização de um laço quando um valor que é escrito em uma iteração do laço é escrito ou lido em outra iteração. Quanto mais complexa a malha, mais difícil pode ser para decidir sobre tais dependências. Assim, deve-se ter em mente que a colocação correta das diretivas OpenMP é de responsabilidade do usuário e que de vez em quando é necessário algum conhecimento sobre as aplicações para decidir se realmente existe uma dependência (CHAPMAN *et al.*, 2008).

Na extração de paralelismo de um dado problema, a análise de dependência tem grande impacto, pois se não for explorada adequadamente o resultado será um algoritmo pobre em otimização paralela.

3.3.9 Granularidade

O número e o tamanho das tarefas em que um problema pode ser decomposto determina a chamada **granularidade** da decomposição *apud* Grama et al. (2003).

A decomposição de um problema em um número *grande* de pequenas tarefas é chamada de *granulação fina (fine-grained)* e a decomposição de um problema em um número *pequeno* de grandes tarefas é chamada de *granulação grossa (coarse-grained)* (KURFESS, 1991), (WILKINSON; ALLEN, 1999), (GRAMA *et al.*, 2003). Por exemplo, a decomposição para a multiplicação de uma matriz vezes um vetor, conforme ilustrado na Figura 16 é usualmente considerada granulação fina, pois cada uma das n tarefas realiza um único produto interno $p = A \times b$.

Figura 16 – Decomposição da multiplicação de uma matriz com um vetor em n tarefas, onde n é a ordem da matriz. Granulação fina (**fine-grained**).

Fonte: (GRAMA *et al.*, 2003, p. 87), reproduzida aqui apenas para efeito didático.

A Figura 17 mostra a decomposição do mesmo problema em apenas quatro tarefas usando granulação grossa, onde cada tarefa calcula $n/4$ produtos internos de cada vez.

Figura 17 – Decomposição de uma multiplicação de uma matriz com um vetor em apenas quatro tarefas, onde n é a ordem da matriz. Granulação grossa (**coarse-grained**).

Fonte: (GRAMA *et al.*, 2003, p. 90), reproduzida aqui apenas para efeito didático.

Assim, um conceito relacionado à granularidade é a de **grau de concorrência**. Ou seja, o número máximo de tarefas que podem ser executadas simultaneamente em um programa paralelo em um determinado tempo é conhecido como o seu **grau máximo de concorrência**. Em muitos casos o grau máximo de concorrência é menor do que o número total de tarefas devido a dependências entre as tarefas (KURFESS, 1991), (CODENOTTI; LEONCINI, 1993), (WILKINSON; ALLEN, 1999), (GRAMA *et al.*, 2003).

3.3.10 Memória Compartilhada

Um termo aplicado a hardware e software que indica a presença de uma região de memória que é compartilhada entre os componentes do sistema. Para ambientes de programação, o termo significa que a memória é compartilhada entre processos ou threads. Aplicado ao hardware, isso significa que a característica arquitetônica de manter processadores juntos é a memória compartilhada. Ou seja, um bloco de memória endereçável que é compartilhado entre uma coleção de UEs (Unidades de Execução).

A arquitetura de memória compartilhada mostrada na Figura 18 é utilizada pelo modelo de programação paralela OpenMP. Note-se que todas as CPUs

compartilham a mesma memória, através de uma rede de interconexão interna ao sistema.

Figura 18 – Arquitetura de memória compartilhada.

Fonte: (Imagem padrão semelhante às ilustradas em vários textos, reproduzida aqui apenas para efeito didático.)

No modelo de programação com memória compartilhada muitas das variáveis são compartilhadas por padrão.

3.3.11 Memória Distribuída ou Computação Distribuída

A memória distribuída é um espaço de endereçamento compartilhado entre várias UEs que é construído a partir de subsistemas de memória que são distintos e distribuídos sobre o sistema. Poderá haver um sistema operacional e suporte de hardware para o sistema de memória compartilhada distribuída, ou a memória compartilhada poderá ser totalmente implementada em software como uma camada de middleware separada.

A computação distribuída é um tipo de computação em que uma tarefa computacional é dividida em subtarefas que são executadas em um conjunto de computadores ligados em rede. As redes são de propósitos gerais tais como LANs, WANs, ou a Internet em oposição ao grupo de interconexões dedicadas.

A Figura 19 mostra um esquema de arquitetura distribuída tradicional.

Figura 19 – Arquitetura de memória distribuída.

Fonte: (Imagem padrão semelhante às ilustradas em vários textos, reproduzida aqui apenas para efeito didático.)

Por sua vez, a memória compartilhada virtual é um sistema que fornece a abstração de memória compartilhada, permitindo aos programadores escreverem seus programas para a memória compartilhada, mesmo quando o hardware é baseado em uma arquitetura de memória distribuída. Sistemas de memória compartilhada virtual podem ser aplicados dentro do sistema operacional, ou como parte do ambiente de programação (GRAMA *et al.*, 2003).

3.3.12 Sistemas Híbridos

Em alguns casos, mais de um modelo de programação pode ser usado na solução de um problema, resultando em um algoritmo híbrido. Um modelo híbrido pode ser composto de múltiplos modelos aplicados hierarquicamente ou diversos modelos aplicados sequencialmente em diferentes fases do algoritmo paralelo.

Ainda, em muitos casos, a formulação de um determinado algoritmo pode ter características de mais de uma técnica de desenvolvimento de algoritmos. Por exemplo, os dados podem fluir através de uma arquitetura pipeline, sendo guiados por um padrão de grafo de dependência de tarefas.

Em outras situações, quantidades maiores de computação podem ser descritas, também por um grafo de dependência de tarefas, mas cada nó do grafo representará uma super tarefa compreendendo múltiplas subtarefas, que podem ser adequadas ao paralelismo de dados ou ao pipeline paralelo.

Um bom exemplo de sistema híbrido é o algoritmo de ordenação Quicksort apresentado nas seções 3.2.5 e 9.4.1 de (GRAMA *et al.*, 2003).

Mattson *et al.* (2005) concluem que os sistemas híbridos são clusters de nós com espaços de endereços separados, em que cada nó contém diversos processadores com memória compartilhada. Esses clusters são formados de diversos SMPs conectados por uma rede de interconexão rápida, que são atualmente uma tendência em computação de alto desempenho (HPC).

3.4 Ambientes de Programação Paralela

Este assunto já está consagrado na literatura acadêmica e científica, por esta razão é importante fazer um breve resumo de forma bastante simplificada, porém objetiva. Teremos a oportunidade de nos inteirar da existência de algumas linguagens de programação paralela. Lembrando que a programação paralela pode ser feita em ambientes com memória compartilhada local, distribuída ou híbrida.

Os ambientes de programação paralela nos proveem um conjunto de ferramentas, características de linguagens e interfaces de programação de aplicações (APIs) necessárias à construção de programas paralelos. Um ambiente de programação implica em uma abstração particular de um sistema computacional chamado Modelo de Programação *apud* Mattson *et al.* (2005, p.12). No caso de arquiteturas sequenciais tem-se a bem conhecida Máquina de von Neumann e para arquiteturas paralelas as mais conhecidas são as Taxonomias de Flynn.

A década de 1990 marcou a convergência dos ambientes de programação paralela para apenas dois modelos: OpenMP para memória compartilhada local e MPI para passagem de mensagem.

OpenMP é uma API que fornece um modelo portátil de programação para desenvolvedores de aplicações paralelas usando memória compartilhada. Suas implementações estão atualmente disponíveis para as linguagens de programação Fortran, C e C++. Ela é frequentemente usada para se acrescentar paralelismo em programas sequenciais. Entretanto, OpenMP não é ideal para arquiteturas híbridas que combinam vários nós, em que cada nó possui vários processos e uma memória compartilhada com espaços de endereços separados para cada um. Pois o modelo OpenMP não reconhece os tempos de acessos não uniformes de memória, fazendo com que a alocação de dados resulte em mau desempenho nos casos de máquinas que não sejam SMPs (CHAPMAN *et al.*, 2008), (MATTSON *et al.*, 2005).

Programas escritos em OpenMP tendem a trabalhar muito bem em máquinas com arquitetura SMP, mas OpenMP é menos ideal para máquinas com arquiteturas ccNUMA e de memória distribuída (MATTSON *et al.*, 2005). Na verdade, OpenMP pode ser implementada em um computador que não fornece coerência de cache, já que tem seu próprio conjunto de regras sobre como os dados são compartilhados entre as threads em execução em diferentes processadores (CHAPMAN *et al.*, 2008). Felizmente, o usuário de OpenMP não precisa entender o funcionamento da coerência de cache em um determinado computador. A Figura 20 mostra o modelo de programação suportado por OpenMP. O programa inicia com um simples thread de execução, a **Thread inicial**, região serial. Observa-se que um **Conjunto de threads** é criado na região paralela (entre **Fork** e **Join**) e juntado, novamente, após a chamada de **Join**, região serial (CHAPMAN *et al.*, 2008).

Figura 20 – Modelo de programação suportado por OpenMP (*fork-join*).

Fontes: (CHAPMAN *et al.*, 2008, p. 24) e vários outros textos.

O sucesso da linguagem OpenMP pode ser atribuído a uma série de fatores. Um deles é a sua forte ênfase em programação paralela estruturada e, a sua maior vantagem está no fato dela poder rodar em muitas plataformas diferentes. Mas, acima de tudo OpenMP é oportuna, pois possui um forte crescimento na implantação de SMPs pequenas, grandes e outros *hardwares multithreading*. Chapman *et al.* (2008) apresenta um tutorial completo sobre OpenMP, enquanto que Mattson *et al.* (2005) apenas explora alguns poucos recursos necessários de OpenMP em parte dos exemplos de seus padrões de programação paralela.

MPI é uma biblioteca com um conjunto de rotinas que fornecem gerenciamento de processos, passagem de mensagens e algumas operações de comunicação coletiva. Normalmente, é difícil de escrever programas com ela, pois é responsabilidade do programador garantir o gerenciamento adequado da distribuição dos dados e da comunicação explícita usando mensagens.

Utiliza-se MPI para desenvolvimento de programas que necessitem rodar em arquiteturas com modelos de memória distribuída, pois MPI trabalha com troca de mensagens entre processos (WILKINSON; ALLEN, 1999), (MATTSON *et al.*, 2005), através de um canal de comunicação. Como o modelo de programação de MPI foi

projetado para trabalhar com memória distribuída, ela é uma boa escolha para se trabalhar com processadores densamente paralelos (MPPs) e outras máquinas de memória distribuída. Entretanto, MPI não oferece suporte para gerenciamento de dados residentes em memória compartilhada (MATTSON *et al.*, 2005).

A Figura 21 mostra o modelo de programação resumido suportado por MPI, note que a **Aplicação 1** do **Processador 1** envia (*send*) um pacote de **DADOS** para o **Processador 2**, o qual recebe em seu buffer (MPI Buffer) e entrega para a **Aplicação B**, através de (*receive*), para o processamento local ser realizado.

A terminologia *Processador* foi utilizada no modelo, mas não significa que ambos processadores, nele presentes, estejam em um único chip multinúcleo ou mesmo que sejam independentes e instalados na mesma placa de sistema, isto é, podem ser interpretados como **computadores** conectados através de uma rede de comunicação.

Figura 21 – Modelo de programação suportado por MPI (*send-receive*).

Fonte: (Imagem semelhante às ilustradas em diversos textos, reproduzida aqui apenas para efeito didático)

Algumas das linguagens de programação paralela mais recentes, apresentadas ao público nos últimos quatro anos são OpenCL, CUDA, OpenACC e OpenHMPP, todas voltadas para computação em unidades de processamento gráfico de propósito geral (GPGPU). Destas, a especificação da OpenCL pode ser encontrada no site oficial do consórcio responsável por sua padronização em Khronos (2012). As especificações de CUDA podem ser encontradas no site oficial da empresa

desenvolvedora das mesmas, que também é fabricante de chips GPGPU em Nvidia (2012), também existem excelentes literaturas sobre o assunto tais como (FARBER, 2011), (SANDERS; KANDROT, 2011), (KIRK; HWU, 2010), (HWU, 2012). As especificações da OpenACC podem ser encontradas em OpenACC (2012). Por último, a OpenHMPP foi criada pela empresa CAPS em 2007 e suas especificações podem ser encontradas no site oficial da CAPS (2012).

Pode-se encontrar em (MATTSON *et al.*, 2005, p.14) uma lista com 218 ambientes de programação paralela existentes na década de 1990.

Modelo SPMD

Em um programa SPMD, todas as UEs executam o mesmo código (Single Program) em paralelo, mas cada uma tem seu próprio conjunto de dados (Multiple Data). Diferentes UEs podem seguir caminhos diferentes através do programa. No código fonte, a lógica para controlar isto é expressa usando um parâmetro único que rotula cada UE (por exemplo, o ID do processo) (MATTSON *et al.*, 2005) (GRAMA *et al.*, 2003).

O modelo SPMD é ideal onde cada processo executará exatamente o mesmo código, mas isso não impede uma abordagem mestre-escravo, onde ambos os códigos mestre e escravo devem ficar no mesmo programa (WILKINSON; ALLEN, 1999) (CHAPMAN *et al.*, 2008).

Ainda, conforme Wilkinson e Allen (1999) o modelo SPMD apresenta a vantagem dos argumentos de linha de comando poder ser transmitidos para cada processo. Assim, dado um modelo SPMD, qualquer declaração global de variáveis será duplicada em cada processo, já as variáveis que não podem ser repetidas devem ser declaradas localmente, isto é, declaradas no código executado apenas pelo thread na qual foi declarada.

CPAR

Durante a pesquisa que resultou na Tese de Livre Docência na Escola Politécnica da Universidade de São Paulo, no Departamento de Engenharia da Computação e Sistemas Digitais, a então recém Doutora em Engenharia Liria Matsumoto Sato desenvolveu, no final dos anos 1980, uma linguagem paralela baseada no modelo de memória compartilhada distribuída – *DSM: Distributed Shared Memory*, como uma extensão da linguagem C, para a plataforma UNIX, e a nomeou de CPAR. A CPAR, também está disponível para a plataforma Linux.

Desde seu desenvolvimento, a linguagem CPAR vem sendo utilizada, pela Profa. Dra. Liria M. Sato, em suas disciplinas de pós-graduação, para ensinar programação paralela aos seus alunos. CPAR é uma linguagem fácil de aprender e utilizar, portanto, muito boa no aprendizado de computação paralela e distribuída.

4 METODOLOGIAS PROPOSTAS

Como visto na revisão bibliográfica, há várias propostas de metodologias para paralelização de modelos computacionais. Entretanto, nenhuma delas oferece um conjunto de procedimentos que oriente o modelador como realizar esta tarefa que parece ser trivial, mas que é muito complexa e difícil, principalmente se tratando da fase de otimização, de forma simples como a Metodologia apresentada neste trabalho.

A proposta aqui apresentada envolve o desenvolvimento de uma metodologia para a implementação, bem como a conversão de modelos matemáticos e computacionais sequenciais ou híbridos em modelos computacionais paralelos otimizados, visando maior simplicidade em sua compreensão e *maior velocidade*¹⁰ de processamento, sem *perda de generalidade*¹¹.

Como proposta principal, este trabalho apresenta uma metodologia subdividida em três etapas que, se aplicadas de forma completa e sistemática conduzirá o usuário a

¹⁰ O termo *maior velocidade* aqui não significa que a velocidade de processamento obtida com a aplicação da metodologia proposta seja a máxima, mas pura e simplesmente que o algoritmo paralelo otimizado atingirá uma velocidade maior do que aquela alcançada pelo modelo original, antes de sua otimização.

¹¹ A expressão *sem perda de generalidade* significa que o modelo resultante da aplicação desta metodologia gerará resultados igualmente precisos e confiáveis tanto quanto aqueles gerados pelo modelo original.

alcançar um grau de qualidade satisfatório no resultado esperado para a solução de seu problema. Essas três etapas são mostradas em detalhes nos capítulos desta Tese, conforme indicados a seguir:

Capítulo 5: Extraíndo Paralelismo;

Capítulo 6: ESPG – Expressão Semântica de Processamento Geral e;

Capítulo 7: Ψ - Uma Nova Linguagem de Programação.

Uma segunda metodologia de paralelização é proposta no capítulo 8, que consiste em uma forma de calcular o determinante de uma matriz densa de ordem n , $n \in \mathbb{N} \mid n \geq 2$, fazendo uso de duas técnicas diferenciadas, sendo uma algébrica e outra gráfica. A técnica algébrica é muito parecida com a *Regra de Chió*, vide Apêndice B, e recebe o nome de Método das Divisões Internas (MDI), que segue o padrão determinado no Método Gráfico Prático (MGP), utilizado como ferramenta de apoio tanto para o cálculo manual, quanto para o cálculo computacional.

A maior parte deste trabalho foca a proposta que envolve os capítulos 5, 6 e 7, pois essa é muito mais complexa e, portanto, considerada a mais contributiva como trabalho científico, a qual poderá gerar resultados muito relevantes nos âmbitos Acadêmico e Científico. Também, com grande aplicabilidade no desenvolvimento de aplicações nos contextos Industrial e Comercial. Nos casos Industrial e Comercial é justificado seu uso, principalmente, pela programação do modelo resultante poder ser feita com a linguagem Ψ , vide capítulo 7 e Apêndice A deste texto.

A etapa de extração de paralelismo apresentada no capítulo 5 é uma das partes mais importantes da metodologia proposta, pois ela é totalmente independente de tecnologia de hardware e aplica-se diretamente a modelos matemáticos, bem como a grandes conjuntos de dados que possam ser interpretados matematicamente. Ela é constituída de um conjunto de regras bem definidas que devem ser obedecidas em sua totalidade, conforme apresentadas.

A etapa que trata da Expressão Semântica de Processamento Geral apresentada no capítulo 6, é outra parte muito importante e, também independente de tecnologia de hardware. Esta etapa é muito importante para se alcançar o resultado otimizado que desejado, pois ela exigirá do usuário um reagrupamento do modelo inicial, outrora fragmentado na etapa da extração de paralelismo, de forma coordenada e de modo a representar os modelos computacionais utilizados, bem como a ordem em que suas partes serão executadas. Entretanto, só faz sentido utilizar o potencial da ESPG se, e somente se, o usuário pretender programar sua solução final com a nova linguagem de programação Ψ (Psi).

Finalmente, a terceira e última etapa desta metodologia consiste na apresentação de uma nova linguagem de programação paralela intitulada Ψ , conforme mostrada no capítulo 7. Essa linguagem foi criada no decorrer deste trabalho com o principal objetivo de traduzir a ESPG de acordo com sua especificação, em um programa escrito em linguagem C++ com recursos de computação paralela, proporcionando ao usuário a capacidade de programar os computadores equipados com os atuais processadores de múltiplos núcleos e GPGPUs de muitos núcleos, de forma fácil e rápida. Recursos esses especificados no corpo da ESPG. Ψ é uma linguagem de programação orientada a objetos que usa a já consagrada linguagem C++ como suporte e é paralela. Acredita-se que ela seja clara, enxuta e muito fácil de ser aprendida e manipulada.

O modelo de programação paralela de Ψ é implícito, significando que o usuário fica dispensado e é poupado do árduo aprendizado das complicadas técnicas existentes na computação paralela e distribuída, bem como de seus jargões. Ψ possui outras características tais como ser orientada a padrões e a capacidade de utiliza diferentes modelos de programação paralela no mesmo programa, permitir ao programador escrever programas utilizando três técnicas diferentes em uma única linha de código, neste caso a ESPG, que são *sequencial*, *paralelo* e *pipeline*, dentre outras possibilidades. Ψ tem a capacidade de ocultar o trabalho “pesado” que as linguagens e APIs de suporte lhe proporcionam. Outra característica importante da linguagem Ψ é que ela está preparada para trabalhar com os modelos de memória compartilhada local e distribuída.

Os itens que se seguem mostram, de forma resumida, mas clara e objetiva, a metodologia proposta neste texto em um diagrama de blocos único, com suas respectivas descrições.

4.1 Escopo Completo da Metodologia de Paralelização Proposta

A metodologia aqui proposta visa facilitar o trabalho de conversão ou transformação de um dado modelo matemático ou computacional escrito para processamento sequencial ou paralelo, ou mesmo híbrido, em um modelo matemático ou computacional otimizado para processamento paralelo com ou sem pipeline. Sendo esta metodologia facilmente compreendida através da análise de suas etapas, conforme mostrado na Figura 22.

Uma explicação simplória para o fluxo de dados no modelo apresentado na Figura 22 é: dado um modelo matemático ou computacional qualquer, apresentado na ENTRADA, o qual sofrerá uma TRANSFORMAÇÃO através de uma série de análises e otimizações matemáticas e computacionais, sendo por fim entregue ao usuário um modelo, também matemático ou computacional paralelo, com ou sem pipeline, otimizado e robusto em sua SAÍDA.

Figura 22 – Etapas da metodologia de paralelização e otimização proposta.

Os itens a seguir possuem simples resumos da composição e do funcionamento de cada etapa da metodologia, conforme Figura 22.

Por se tratar de uma metodologia, todo o trabalho de paralelização e otimização do modelo de entrada, realizado na etapa TRANSFORMAÇÃO, bem como o resultado codificado em uma linguagem de programação na etapa SAÍDA é realizado, manualmente, pelo usuário, mediante execução de uma série de procedimentos bem definidos.

Portanto, essa tarefa exige conhecimentos abrangentes de matemática e da própria metodologia aqui apresentada, para se alcançar um grau satisfatório de ganho de qualidade e desempenho no resultado desejado.

4.1.1 Modelo a Paralelizar e Otimizar – ENTRADA

O módulo ENTRADA, na Figura 22, representa o modelo matemático ou computacional a ser transformado (paralelizado e otimizado). Esse modelo pode ser uma única fórmula ou um conjunto de equações que descreve matematicamente um fenômeno físico ou de outra natureza. Também, pode ser apenas o enunciado de um dado problema de Física, Matemática, Economia, Engenharia ou outro que resulte em um modelo matemático.

Mesmo que o modelo de entrada seja naturalmente paralelizável, ao passar pela metodologia de paralelização o mesmo será rearranjado de forma paralela otimizada.

4.1.2 Metodologia de Paralelização e Otimização – TRANSFORMAÇÃO

O módulo TRANSFORMAÇÃO, da Figura 22, compreende duas partes da metodologia de paralelização e otimização proposta neste texto, propriamente ditas e é baseada em diversas técnicas elementares de análise matemática, computacional e a estratégia de Dividir para Conquistar.

O objetivo deste módulo, como o próprio nome sugere, é transformar o modelo a paralelizar e otimizar apresentado no bloco ENTRADA, vide Figura 22, em um modelo matemático paralelo otimizado, sem perder a precisão numérica do modelo original de entrada, mas de forma mais clara, rápida e robusta. No decorrer da

extração de paralelismo, este bloco conduzirá a diversas análises e simplificações¹² do modelo original visando, como uma de suas tarefas mais importantes, a alteração sistemática do número de operações aritméticas simples ou fórmulas, bem como o número de tarefas complexas existentes no mesmo, de modo a otimizar o uso do hardware disponível. A segunda etapa, não menos importante do que a extração de paralelismo é a edição da expressão semântica de processamento geral – ESPG, que resultará em uma codificação mais legível do programa fonte resultante, a ser compilado.

4.1.3 Modelo Paralelizado e Otimizado – SAÍDA

O módulo SAÍDA, veja a Figura 22, apresentará ao usuário o modelo paralelizado e otimizado, que por sua vez poderá ser híbrido, obtido do módulo TRANSFORMAÇÃO. Após a finalização desta fase o modelo paralelo otimizado resultante poderá ser utilizado pelo usuário, da forma como desejar. Entretanto, mantendo-se em mente que todo este trabalho foi realizado objetivando o uso mais adequado e otimizado dos atuais computadores disponíveis no mercado, o usuário deve optar por codificá-lo na linguagem paralela Ψ , desenvolvida exclusivamente como parte integrante deste trabalho e que visa traduzir as ESPGs em código executável. Incluem-se na lista de hardware atual os processadores lançados no mercado mais recentemente, em meados do ano 2011, contendo uma GPGPU integrada no mesmo chip da CPU. A este tipo de GPGPU, normalmente dá-se o nome de iGPU – GPU Integrada (**integrated GPU**).

A tecnologia de integração de GPGPU no mesmo chip da CPU já está sendo usada em grande escala pelos fabricantes de processadores e, vê-se claramente que as

¹² No contexto dessa etapa da Metodologia aqui proposta, analisar um modelo matemático ou de outra natureza, visando a extração de paralelismo não significa a redução do número de expressões ou operações matemáticas ou mesmo de tarefas mais complexas, mas sim, na maioria dos casos, o aumento desses elementos, permitindo-se desta forma a sua distribuição entre as diversas unidades de execução (UEs) existentes na infraestrutura de hardware disponível.

próximas gerações de microprocessadores serão todas baseadas nesta nova arquitetura de hardware.

5 EXTRAINDO PARALELISMO

A extração de paralelismo é uma tarefa, normalmente, árdua para a maioria dos programadores. Entretanto, existem diversas situações em que esta tarefa é trivial, tais como em grandes volumes de cálculos envolvendo vetores e matrizes, e em situações nas quais possa existir pouca dependência entre os dados, ou mesmo nenhuma¹³.

Esta é a primeira das três etapas que constituem a metodologia de otimização e paralelização de modelos matemáticos e computacionais, a ser apresentada.

Considera-se esta etapa como sendo a mais importante da metodologia proposta, pois é durante o desenrolar de suas fases que será possível se obter um grau satisfatório na otimização do problema, bem como uma extração adequada do paralelismo existente no modelo.

5.1 Especificações da Extração de Paralelismo

A metodologia de extração de paralelismo aqui apresentada está descrita na forma de procedimentos, onde a principal técnica utilizada é a de Dividir e Conquistar, que

¹³ A quantidade de problemas de Matemática, Física, Engenharia e Simulação dentre outras atividades que envolvam cálculos matemáticos, que não possuem dependência de dados é muito pequena em relação à quantidade de problemas que contêm dependência de dados.

será executada passo a passo. Nem todos os procedimentos são obrigatórios, entretanto deve-se atentar para a execução de todos eles, uma vez que o objetivo é atingir um paralelismo com considerável grau de otimização para o problema em análise.

Esta primeira parte da metodologia aqui proposta independe de qualquer tecnologia de hardware que tenha existido até o momento, pois suas características são puramente baseadas em análises conceituais de problemas matemáticos e computacionais. Todavia, é importante frisar que o último item deste conjunto de procedimentos é o único que apresenta um conteúdo voltado para a atividade prática de alguma linguagem de programação de computadores, que possua recursos próprios para o auxílio em computação paralela, envolvendo algum tipo de hardware.

Os melhores resultados obtidos com a aplicação desta fase da metodologia estão relacionados aos problemas que envolvem processamento matemático de alguma natureza, entretanto espera-se que ela venha a ser utilizada com sucesso e sem grande dificuldade em problemas de qualquer área do Conhecimento Humano.

A qualidade do resultado obtido com a aplicação desta metodologia depende do bom entendimento de suas técnicas, onde se leva em maior consideração a qualidade do modelo matemático obtido.

Esta etapa é apresentada de forma simplificada, sem qualquer detalhe especial, da seguinte maneira:

- 1) Analisar o problema a ser resolvido do ponto de vista matemático;
- 2) Extrair o Modelo Matemático completo do problema;
- 3) Otimizar o modelo matemático obtido no procedimento (2) utilizando-se de algum método de otimização numérica, se houver necessidade;
- 4) Extrair os dados a partir do modelo matemático obtido;
- 5) Analisar o modelo matemático do ponto de vista computacional. Neste ponto, considera-se o modelo todo como sendo híbrido (mistura de Paralelo

com Sequencial). Neste momento deve ser feita uma pré-fragmentação do modelo, a fim de prepará-lo para a próxima etapa;

- 6) De posse dos resultados da análise feita no procedimento anterior, fragmentar o modelo resultante em três subgrupos de processamentos, se possível. Uma boa prática para se conseguir isto é através da construção de uma tabela constituída de três colunas, a saber: coluna 1) **Paralelo natural**; coluna 2) **Sequencial** e; coluna 3) **Híbrido**. Cada linha dessa tabela pode representar uma iteração, de um conjunto de iterações necessárias ao seu preenchimento adequado:

6.1) Paralelo natural;

6.2) Sequencial e;

6.3) Híbrido;

- 7) Enumerar os blocos obtidos no procedimento (6), como variáveis e/ou tarefas que possuam dependência entre si, se for o caso;
- 8) Enumerar os blocos obtidos no procedimento (6), como variáveis e/ou tarefas compartilhadas, se for o caso;
- 9) Se o subgrupo híbrido for não vazio, então retornar ao procedimento (5). Repetir este processo até que não seja mais possível fragmentá-lo em novos subgrupos;
- 10) Reagrupar todas as variáveis e/ou tarefas obtidas nos procedimentos anteriores, de modo a obter uma estrutura de múltiplos estágios, ou seja, o pipeline, se possível, ou ao menos reagrupá-las em blocos sequenciais, realizando diversos processamentos em paralelo, ou concorrentes;
- 11) Editar a Expressão Semântica de Processamento Geral – ESPG da solução obtida e;
- 12) Traduzir a ESPG obtida para uma linguagem de programação paralela, de preferência para a linguagem Ψ .

Todos os passos constituintes desta primeira etapa da metodologia, que corresponde à otimização do modelo matemático ou computacional inicial, visando a extração de paralelismo otimizado, são detalhados a seguir, de forma que o seu entendimento seja pleno e sua aplicação imediata:

- 1)** Analisar o problema a ser resolvido do ponto de vista matemático:
 - a.** Este procedimento consiste na análise inicial feita no problema proposto, de modo a identificar o conteúdo ao qual se refere;
 - b.** Extrair as grandezas (físicas, econômicas, matemáticas, etc.) envolvidas, para facilitar o modelamento matemático a ser realizado no procedimento (2);
- 2)** Extrair o Modelo Matemático completo do problema:
 - a.** Este procedimento não é tão simples como parece e deve ser realizado com cuidado a fim de se evitar erros de interpretação;
 - b.** Munido das grandezas extraídas no item (b) do procedimento (1), se deve elaborar, adequadamente, as prováveis equações que serão utilizadas na solução do problema;
 - c.** Dispor as fórmulas obtidas no item (b) deste procedimento, de forma cronológica da sua utilização;
 - d.** Neste momento, também é importante agrupar as fórmulas em função de suas dependências;
 - e.** Não é possível remover dependências em função do tempo¹⁴, entretanto, existem muitas dependências que podem ser minimizadas e até mesmo eliminadas;
- 3)** Otimizar o modelo matemático obtido no procedimento (2) utilizando-se de algum método de otimização numérica, se houver necessidade:
 - a.** Otimizar o modelo matemático extraído do problema, significa, basicamente, torná-lo linear se possível, se ainda não estiver nesse

¹⁴ Fica a critério do modelador decidir se mantém o modelo em função do tempo ou não. O fato é que se, originalmente, o problema foi modelado em função do tempo, assim deve permanecer até o final da sua otimização.

formato. Esta é a etapa mais complexa e difícil da metodologia e, sua qualidade depende exclusivamente dos conhecimentos e habilidades do usuário;

- b.** A linearização de modelos matemáticos em análise numérica pode ser feita com algumas técnicas puras ou, em muitos casos, através da combinação de dois ou mais métodos;
- c.** Algumas técnicas tais como Pesquisa Operacional¹⁵ e Método dos Elementos Finitos¹⁶ podem ser utilizadas para a linearização de modelos não lineares, com um pouco de trabalho;
- d.** As Séries de Taylor e de MacLaurin¹⁷ podem ser utilizadas para a obtenção de funções polinomiais equivalentes às funções trigonométricas, logarítmicas e exponenciais com a precisão numérica desejada, pois em sistemas microprocessados elas possibilitam resultados com menor complexidade e menor tempo de processamento, e o mais importante para esta etapa da metodologia, é sua elevada capacidade de paralelismo natural;
- e.** Caso não se consiga a linearização do modelo matemático através dos métodos sugeridos no item (d) acima, é possível se atingir um grau de desempenho satisfatório utilizando-se de métodos de otimização numérica, tais como Multiplicadores de Lagrange¹⁸ e métodos iterativos;

4) Extrair os dados a partir do modelo matemático obtido:

- a.** Consiste nos dados que constituem os valores iniciais, intermediários ou finais de variáveis que podem ser de natureza numérica ou

¹⁵ Para detalhes sobre o assunto e Álgebra Linear basta consultar (GOLDBARG; LUNA, 2005), (BRONSON, 1982), (CUNHA, 2011), (BURDEN; FAIRES, 2003), (HUMES *et al.*, 1984), (ANTON; RORRES, 2010) dentre muitos outros.

¹⁶ Detalhes sobre o assunto são encontrados em (ZIENKIEWICZ; MORGAN, 1983), (TEIXEIRA-DIAS *et al.*, 2010), (JIN, 2002), (ASSAN, 2010).

¹⁷ Maiores detalhes em (BURDEN; FAIRES, 2003), (LATHI, 2007), (EWEN; TOPPER, 1977).

¹⁸ Tópico apresentado em (BOUCHARA *et al.*, 1999), (STEEB, 2011).

alfabética e que, por sua vez, podem também indicar intervalos numéricos de um modo geral;

- b. Neste item deve ficar clara a diferença entre valores que permanecerão constantes e aqueles que poderão variar ao longo do processamento da solução do problema;

5) Analisar o modelo matemático do ponto de vista computacional. Neste ponto considera-se o modelo todo sendo híbrido (mistura de Paralelo com Sequencial), então neste momento deve ser feita uma pré-fragmentação¹⁹ do modelo, a fim de prepará-lo para a próxima etapa:

- a. Identificar os blocos candidatos a serialismo, paralelismo natural e híbrido;

6) De posse dos resultados da análise feita no procedimento anterior, fragmentar o modelo resultante em três subgrupos de processamento, se possível. Uma boa prática para se conseguir isto é através da construção de uma tabela constituída de três colunas, a saber: coluna 1) **Paralelo natural**; coluna 2) **Sequencial** e; coluna 3) **Híbrido**. Cada linha dessa tabela pode representar uma iteração, de um conjunto de iterações necessárias ao seu preenchimento adequado;

6.1) Paralelo natural: todos os elementos que constituem paralelismo natural devem ser transferidos para esta coluna;

6.2) Sequencial: todos os elementos que constituem serialismo trivial devem ser copiados para esta coluna e;

6.3) Híbrido: os elementos que não constituem nem paralelismo natural e nem serialismo trivial devem ser transcritos para esta coluna, para serem reanalisados na próxima iteração de fragmentação.

¹⁹ Este método é apresentado na Figura 23.

- 7) Enumerar os blocos obtidos no procedimento (6), como variáveis e/ou tarefas que possuam dependência entre si, se for o caso;
- 8) Enumerar os blocos obtidos no procedimento (6), como variáveis e/ou tarefas compartilhadas, se for o caso;
- 9) Se o subgrupo híbrido for não vazio, então retornar ao procedimento (5). Repetir este processo até que não seja mais possível fragmentá-lo em novos subgrupos;
- 10) Reagrupar todas as variáveis e/ou tarefas obtidas nos procedimentos anteriores, de modo a obter uma estrutura de múltiplos estágios, ou seja, o pipeline, se possível, ou ao menos reagrupá-las em blocos sequenciais, realizando diversos processamentos em paralelo, ou concorrentes;
- 11) Editar a Expressão Semântica de Processamento Geral – ESPG da solução obtida e;
- 12) Traduzir a ESPG obtida para uma linguagem de programação paralela, de preferência para a linguagem Ψ .

Esta primeira parte da metodologia pode ser muito bem compreendida através do diagrama de blocos mostrado na Figura 23.

Figura 23 – Processo de fragmentação do modelo computacional heterogêneo inicial em suas partes homogêneas separadas.

Inicialmente, supõe-se que o problema sendo resolvido apresente um modelo computacional híbrido, representando três blocos de modelos computacionais abstratos, o qual deverá ser fragmentado em três subgrupos, conforme

procedimento (6) e seus subitens, ou seja, paralelo natural, sequencial e híbrido. Então, aplicando-se um processo recursivo, espera-se obter no final um refinamento com apenas duas arquiteturas de processamento computacional: paralelismo natural e sequencial. Embora, provavelmente, em algumas situações se tenha um subgrupo híbrido que não possa ser fragmentado.

Mas, como saber se um bloco é paralelo, sequencial ou híbrido?

Para muitas situações é muito fácil identificar o paralelismo, bastando para isto verificar a existência ou não de dependência entre as tarefas envolvidas no modelo, ou mesmo a dependência de dados entre as tarefas. Por exemplo, imagine uma situação em que haja a necessidade de dois vetores com uma ordem de grandeza consideravelmente elevada, de números complexos, cujos dados deverão ser obtidos de dois arquivos em disco. Esta é uma situação típica de paralelismo natural, na qual se podem ter duas tarefas, uma para carregar os dados de cada arquivo no seu vetor correspondente, pois a carga de ambos os vetores é independente uma da outra. Ainda, no mesmo modelo encontram-se as seguintes situações onde haja a necessidade da multiplicação de um dos vetores por uma determinada constante, em seguida calcular o produto interno entre esses dois vetores. Bem, numa primeira análise são vistas apenas duas tarefas sendo executadas sequencialmente, mas, na realidade existe uma situação de pipeline, em que o produto interno entre os vetores pode ser iniciado antes mesmo do produto da constante com o vetor correspondente terminar.

Assim, para um modelador ou programador que pensa de forma estritamente sequencial este problema teria uma solução, também sequencial. Entretanto, para um conhecedor desta metodologia, seria relativamente fácil identificar a existência do modelo pipeline neste problema. Enfim, voltando ao problema, é fácil ver que ainda há a possibilidade de se ter, ao menos, mais duas situações em que ocorrem o paralelismo natural, as quais são na multiplicação da constante com um vetor e no produto interno dos dois vetores. Portanto, para finalizar este exemplo chega-se a conclusão de que o mesmo possui uma região paralela contendo duas tarefas sequenciais sendo executadas paralelamente, uma região pipeline contendo dois

estágios envolvendo duas tarefas paralelas internamente e, por fim ao menos uma região sequencial envolvendo a regiões paralela e pipeline, anteriores.

Observe que o exemplo anterior menciona apenas alguns dos diversos procedimentos descritos na metodologia de extração de paralelismo, apesar de fazer uso de todos eles, anteriormente detalhados e mostrados na forma de diagrama de blocos da Figura 23. Esta figura apresenta o modelo computacional heterogêneo inicial e suas etapas que ocorrem recursivamente.

Ainda na Figura 23, as partes do modelo original que podem ser paralelizadas ficarão no subgrupo Paralelo, as partes que contêm serialismo ficarão no subgrupo Sequencial e o subgrupo Híbrido, de preferência, deverá ficar vazio, significando que foi atingida a extração máxima de paralelismo; entretanto, se este último subgrupo não ficar vazio é porque a extração de paralelismo desejada foi alcançada ou o processo entrou em laço indefinidamente, não sendo possível continuar com sua fragmentação, levando o modelador a interrompê-lo forçadamente, aplicando a técnica que considerar conveniente.

5.2 Paralelizando Expressões Aritméticas

Nesta seção será apresentada uma técnica elementar, porém eficiente, para extrair o paralelismo de expressões aritméticas de qualquer complexidade. Para isto, deve-se apoiar em três propriedades básicas da Teoria dos Números²⁰. Esses procedimentos poderão ser aplicados em várias etapas do processo de transformação de um modelo de processamento sequencial ou híbrido em paralelo.

As propriedades básicas da Teoria dos Números mencionadas no parágrafo anterior são a Associativa, Comutativa e Distributiva. Com apenas estas três propriedades pode-se facilmente paralelizar muitas expressões aritméticas e diversas equações e

²⁰ Esta teoria é muito importante para a manipulação eficaz dos elementos que constituem expressões aritméticas elementares e equações matemáticas de qualquer complexidade e, pode ser consultada em diversos textos de Matemática.

fórmulas matemáticas que aparecem nos modelos matemáticos do dia a dia. Portanto, elas constituem um ferramental consideravelmente poderoso.

A Tabela 5.1 apresenta as três propriedades mencionadas.

Tabela 5.1: Propriedades básicas para operações aritméticas elementares.

1. ASSOCIATIVA	$A + (B + C) = (A + B) + C$
2. COMUTATIVA	$A + B = B + A$
3. DISTRIBUTIVA	$A * (B + C) = A * B + A * C$

O uso destas propriedades será apresentado através de exemplos simples em paralelismo analítico. A seguir são apresentados alguns exemplos de simplificação de expressões matemáticas, a fim de se obter um paralelismo otimizado das mesmas, simplesmente aplicando as três propriedades acima.

No exemplo 1 há uma expressão aritmética bastante simples, mas que mostra claramente a existência de paralelismo natural.

Exemplo 1: Dada a expressão aritmética $a+b+c+d$. Pede-se analisar e paralelizá-la aplicando as propriedades básicas da aritmética, que se fizerem necessárias.

Nota-se, claramente, que serão efetuadas três operações na expressão dada. Se o modelo for executado em um computador de arquitetura sequencial, o processo ocorrerá em três passos. Estes três passos podem ser vistos, agrupando-se a expressão original, através da inclusão de parênteses, tal como segue:

$$a+b+c+d = (((a+b)+c)+d).$$

Em primeiro lugar adiciona-se **a** com **b**, em seguida adiciona-se o resultado obtido com **c** e, por fim o novo resultado é adicionado a **d**, perfazendo, assim, um total de três operações realizadas por um único processador em três passos, já que este é tipicamente um modelo sequencial.

Entretanto, se aplicarmos a propriedade Associativa na expressão aritmética, a fim de reagrupá-la de forma otimizada, isto é, duas em duas, o seguinte resultado é obtido:

$$a+b+c+d = (a+b)+(c+d)$$

$$\text{expr1} = [(x = a + b).2 \parallel (y = c + d).2] \rightarrow (res = x + y).2 \rightarrow \text{Propriedade Associativa}$$

Agora está clara a possibilidade do uso de Arquitetura Computacional Paralela com dois processadores, em que cada um realizará o cálculo de um par de operandos entre parênteses, perfazendo-se, desta forma, um total de 2 operações em paralelo e uma operação sequencial. Portanto, o total de passos necessários para a solução do problema original foi

reduzido de uma unidade, apesar de ainda termos de realizar as três operações iniciais.

Assim, considerando-se que cada operação seja executada em uma unidade de tempo (ut), vê-se uma redução no tempo total de execução do problema de $3ut$ para apenas $2ut$. Todavia, o *speedup* para esta solução paralela é dada pela eq. (1), vide capítulo 3, e reproduzida abaixo:

$$S(n) = \frac{\text{tempo de execução usando um processador (sistema monoprocessador)}}{\text{tempo de execução usando um multiprocessador com n processadores}} = \frac{t_s}{t_p} \Rightarrow$$

$$S(n) = \frac{t_s}{t_p} = \frac{3ut}{2ut} = 1,5.$$

Isto significa que com um simples rearranjo e agrupamento dos operandos na expressão original faz com que o algoritmo otimizado seja executado ~33% mais rápido do que o algoritmo original.

O exemplo 2, a seguir é um pouco mais elaborado, pois utiliza as quatro operações aritméticas básicas, entretanto seu objetivo é o mesmo do anterior, ou seja, extrair o paralelismo otimizado aplicando somente as propriedades da Teoria dos Números apresentadas neste item.

Exemplo 2: Dada a expressão aritmética $a * (b + c \div d) - e$. Pede-se analisar e paralelizá-la aplicando as propriedades básicas da aritmética que se fizerem necessárias.

Se a expressão dada for executada em um computador sequencial, quatro operações aritméticas serão realizadas, em quatro passos, como segue: 1) divisão; 2) adição; 3) multiplicação e; 4) subtração. Porém, após sua paralelização o modelo terá apenas 5 operações, que serão executadas em apenas três passos. Para isto serão aplicadas as três propriedades da Teoria dos Números citadas anteriormente. Assim, mantendo-se a expressão original a esquerda da igualdade e rearranjando o lado direito, tem-se:

$$a * \left(b + \frac{c}{d} \right) - e = a * b + a * \frac{c}{d} - e \rightarrow \text{Propriedade Distributiva.}$$

$$a * \left(b + \frac{c}{d} \right) - e = a * b - e + a * \left(\frac{c}{d} \right) \rightarrow \text{Propriedade Comutativa}$$

$$a * \left(b + \frac{c}{d} \right) - e = ((a * b) - e) + \left(a * \left(\frac{c}{d} \right) \right) \rightarrow \text{Propriedade Associativa}$$

Está visivelmente claro que a expressão original foi expandida para se obter sua paralelização, pois de outra forma a sua execução continuaria sendo sequencial.

Observe que na expressão original do exemplo 2, existe uma cadeia de dependências que leva à realização de quatro operações com quatro operadores em quatro passos, enquanto que a sua expressão expandida, apesar de possuir cinco operadores, permite executá-la em apenas três passos, já que é possível realizar-se duas operações simultaneamente, em dois passos e uma operação, sequencial, no terceiro passo.

Representações gráficas, da expressão original e sua expandida para o paralelismo, são feitas na forma de árvore de parsing, conforme mostradas na Figura 24.

Figura 24 – Árvore de parsing para as expressões do exemplo 2: (a) expressão original, totalmente sequencial e, (b) expressão expandida para paralelização.

Na Figura 24(a), nota-se visivelmente a total dependência entre as operações, forçando a uma execução sequencial, enquanto que na Figura 24(b) o paralelismo é evidente, mesmo com uma quantidade maior de operações. Os números na coluna central da Figura 24 representam os passos necessários para a execução completa de cada modelo.

6 ESPG – EXPRESSÃO SEMÂNTICA DE PROCESSAMENTO GERAL

A técnica aqui apresentada, intitulada Expressão Semântica de Processamento Geral (**ESPG**), foi criada para dar suporte ao desenvolvimento da metodologia de paralelização inicialmente proposta, visando facilitar a representação e visualização sistemática de processos computacionais paralelos, sequencias ou híbridos de forma simples, enxuta e facilmente compreensível.

6.1 Especificações da ESPG

Normalmente, os processos computacionais são apresentados na forma de diagramas de blocos, vide Figura 25. Entretanto, esta forma é muito dispendiosa, apesar de ser muito fácil de implementar e entender, mas geralmente não descreve de forma clara quais operações aritméticas ou tarefas estão sendo executadas nas várias etapas dos diversos níveis do processamento. Ao contrário, a ESPG que é um tipo de **equação estrutural** consegue descrever sucintamente todas as micro operações e as macro operações que estão (serão) sendo executadas nas diversas etapas dos vários níveis de processamento, se assim o modelador desejar.

De acordo com Kline (2011, p. 78) a forma gráfica não é sempre a melhor opção no âmbito da Modelagem de Equações Estruturais (SEM – Structural Equations Modeling).

Figura 25 – Representação gráfica tradicional de processamento híbrido, envolvendo CPU multicore e GPGPU many-core.

A ESPG para o diagrama de processamento híbrido mostrado na Figura 25 é mostrada na express.(7):

$$ESPG := '(CPU) \rightarrow [['(GPU1) \rightarrow "(GPU1)] || ['(GPU2) \rightarrow "(GPU2)] || "(CPU)] \rightarrow '(CPU) \quad (7)$$

A estrutura da ESPG, até este momento, está apresentada na express.(8) e nas linhas que se seguem estão os significados explícitos de seus elementos.

$$\begin{aligned} <ESPG> := [^{opi}(tarefa).p <operador_proc> ^{opi}(tarefa).p <operador_proc> \dots <operador_proc> \\ &^{opi}(tarefa).p] <operador_proc> [^{opi}(tarefa).p <operador_proc> \dots <operador_proc> ^{opi}(tarefa).p] \\ &<operador_proc> \dots <operador_proc> [^{opi}(tarefa).p <operador_proc> \dots <operador_proc> \\ &^{opi}(tarefa).p] \end{aligned} \quad (8)$$

onde,

- `<ESPG>` : é o nome do “repositório” (apenas a abstração representativa de um recipiente) que contém a equação estrutural da ESPG definida pelo usuário.
- `:=` : “atribuição” da ESPG ao repositório, ou seja, é a relação que existe entre todas as operações/tarefas descritas do lado direito da expressão estrutural e o nome do repositório `<ESPG>`.
- `(tarefa).p` : representa uma micro ou macro tarefa qualquer, que pode ser uma operação aritmética ocorrendo no nível do processador ou mesmo uma tarefa complexa que realiza o processamento de uma ou mais subtarefas que, inclusive podem ser outras micro operações aritméticas, utilizando `p` processadores. O índice `p` ($p \in \mathbb{N}$) deste elemento é opcional e representa o número de processadores que foram alocados para executar o código interno ao bloco delimitado pelos parênteses “(...)”. Esta informação é útil no ganho de velocidade, pois uma vez informado, o compilador alocará exatamente essa quantidade de processadores para executar a tarefa, evitando a perda de tempo com o gerenciamento de alocação automática de recursos por parte da API utilizada como modelo de programação paralela. A omissão deste parâmetro em blocos paralelos fará com que o compilador aloque, automaticamente, dois processadores para a execução de sua tarefa. Se for alocado somente um processador, então um erro em tempo de compilação será gerado. Uma tarefa pipeline pode ser sequencial ou paralela internamente, assim, se o parâmetro `p` for omitido nessa situação, o compilador considerará o modelo sequencial, caso contrário será paralelo. Para blocos sequenciais, o parâmetro `p` deverá ser obrigatoriamente omitido ou será gerado um erro de compilação. O parâmetro `tarefa` pode ser uma ESPG definida anteriormente, neste caso os parênteses e o índice `p` devem ser obrigatoriamente omitidos.
- `<operador_proc>`: é o operador de tipo do fluxo de processamento e, há prioridade de precedência entre eles, da seguinte forma: o operador de processamento pipeline tem maior precedência e o sequencial tem menor precedência. Este operador pode ser um de três tipos:

- $\rightarrow$: (*operador de processamento sequencial*) significa que a operação ou o conjunto de operações seguinte é dependente da operação ou do conjunto de operações à esquerda do operador;
- $||$: (*operador de processamento paralelo*) significa que a operação ou o conjunto de operações seguinte é executado(a) em paralelo com a operação ou o conjunto de operações à esquerda do operador;
- $>||$: (*operador de processamento em pipeline*) significa que a operação ou o conjunto de operações seguinte é executado(a) em pipeline com a operação ou o conjunto de operações à esquerda do operador.
- opi : (*operador de processamento interno*) informa ao compilador que o processamento interno aos delimitadores “(. . .)” de bloco de operações ou tarefas deverá ser implementado com um de três padrões:
 - ``` (*caractere apóstrofo*): o processamento do conteúdo dentro dos delimitadores deve ser implementado seguindo o modelo de programação sequencial;
 - `"` (*caractere aspas*): o processamento do conteúdo dentro dos delimitadores deve ser implementado seguindo o modelo de programação paralela;
 - `..` (*caractere trema*): o processamento do conteúdo dentro dos delimitadores deve ser implementado seguindo o modelo de programação pipeline.
- $()$: representa blocos de processamento, que podem ser micro ou macro operações ou tarefas. O bloco delimitado por parênteses “()” é obrigatório e mantém o elemento que será representado por um conjunto de operações, gerado pela linguagem de programação Ψ .
- $[]$: utilizados para definir uma região de processamento, agrupando todos os blocos acoplados entre si através de operadores de processamento externo. Sua função é idêntica à dos parênteses em expressões aritméticas comuns, a fim de agrupar um conjunto de operações. São opcionais, mas

recomendáveis por motivos de boa prática de modelagem, programação e documentação.

À primeira vista, implementar uma ESPG parece ser complicado, mas com um pouco de convivência e prática o modelador conseguirá escrevê-la diretamente, para diversos modelos das mais variadas complexidades.

6.2 Exemplos Práticos de ESPGs

Neste item serão apresentados alguns exemplos práticos de paralelismo utilizando-se a ESPG como ferramenta de representação de processos computacionais. Esses exemplos não são muito extensos, pois o objetivo é mostrar a facilidade em se obter a ESPG de um modelo computacional dado e, apresentar os benefícios obtidos com a edição e interpretação da mesma.

O exemplo 3 trata da tradução de um diagrama de blocos usado para mostrar o fluxo de dados de um processamento paralelo utilizando diversas tarefas, extraído da disciplina *Programação Paralela e Distribuída*, ministrada pela Profa. Liria M. Sato nos cursos de Pós-Graduação da Escola Politécnica da Universidade de São Paulo.

Exemplo 3: Representar o diagrama de blocos para processamento de múltiplas tarefas, ilustrado a seguir na forma de uma ESPG.

Identificando os elementos de processamento presentes no diagrama: Há seis tarefas (*tar1*, ..., *tar6*) sendo executadas em paralelo, duas a duas e, mais duas representações de processamento sequencial ("*seq1*" e "*seq2*") neste diagrama. Ainda, existe uma tarefa sequencial "*seq3*" que não foi

rotulada no diagrama de blocos original. Cada interseção encontrada abaixo de um par de tarefas paralelas representa uma espera para sincronismo entre elas, antes de continuar o fluxo normal até o próximo elemento de processamento, que normalmente é um bloco sequencial (neste diagrama em particular), com exceção daquela que está embaixo das tarefas *tar5* e *tar6*, que aguarda o término da execução da tarefa "seq3" para sincronização. Note que cada uma dessas tarefas pode ser de qualquer tipo, ou seja, micro tarefa representando simples operação aritmética ou pequenas porções de processamento computacional ou, ainda podem representar macro tarefas de qualquer natureza. O diagrama de blocos não informa o modelo de programação das tarefas *tar1*, ..., *tar6*, entretanto, pelo fato da ESPG requerer previamente esta informação, considera-se que todas elas sejam programadas em paralelo, internamente.

Expressão Semântica de Processamento Geral (ESPG) equivalente:

O repositório para esta ESPG será intitulado *diagramaBlocos*.

$$\text{diagramaBlocos} := '(\text{seq1}) \rightarrow ["(\text{tar1}).2 \parallel "(\text{tar2}).2 \rightarrow '(\text{seq2}) \rightarrow "(\text{tar5}).2 \parallel "(\text{tar6}).2] \parallel ["(\text{tar3}).4 \parallel "(\text{tar4}).4 \rightarrow '(\text{seq3})]$$

Ao contrário do diagrama de blocos, esta expressão pode ser colocada diretamente dentro de um programa escrito em linguagem Ψ (Veja o capítulo 7).

Diferenças entre ambas as formas de representação: Na ESPG *diagramaBlocos* as tarefas *tar1*, ..., *tar6* aparecem entre parênteses precedidos do operador de paralelismo interno (" - aspas) especificando que todas essas tarefas devem ser executadas, internamente, em paralelo além de serem executadas em paralelo duas a duas; esta representação não é possível usando-se o tradicional diagrama de blocos. Ainda, as tarefas "seq1", "seq2" e "seq3", também recebem a especificação de tipo de processamento através o operador de serialismo interno (' - apóstrofo); isto também não é possível no diagrama de blocos tradicional.

Previsão do número de processadores necessários para executar todo o processo:

Informar o número de processadores a serem usados na execução de cada tarefa paralela é um diferencial muito importante para a ESPG, em relação aos Diagramas de Blocos. Isto também permite um controle maior sobre o uso de recursos do hardware por parte do programador.

Dessa forma, serão alocados dois processadores para as tarefas *tar1*, *tar2*, *tar5* e *tar6*, e quatro processadores para as tarefas *tar3* e *tar4*; via de regra, cada tarefa sequencial é executada por apenas um processador. Vide ESPG *diagramaBlocos*.

O exemplo 3 apresenta uma ideia precisa da simplicidade e versatilidade da ESPG, comparada com seu equivalente descrito através do tradicional diagrama de blocos.

O exemplo 4, a seguir, mostra a obtenção de ESPGs a partir das expressões aritméticas dos exemplos 1 e 2, respectivamente, apresentados no capítulo 5.

Exemplo 4: Dadas as expressões aritméticas $expr1 = a+b+c+d$, do exemplo 1 e $expr2 = a*(b+c\div d)-e$, do exemplo 2. Pede-se para editar as ESPGs para o modelo paralelo.

As ESPGs para as expressões expandidas dos exemplos 1 e 2 são, respectivamente:

$$expr1 := ["(x = a + b).2 \parallel "(y = c + d).2] \rightarrow "(res = x + y).2 \quad e$$

$$expr2 := ["(r = a * b) \parallel "(s = \frac{c}{d})] \rightarrow ["(x = r - e) \parallel "(y = s * a)] \rightarrow "(res = x + y).$$

No exemplo 5 será mostrada a obtenção direta, isto é, a partir do problema original sem a necessidade de qualquer diagrama de blocos, de uma ESPG objetivando mostrar a sua simplicidade.

Este exemplo consiste de cálculos envolvendo matrizes, tais como adição e multiplicação.

Exemplo 5: Dadas duas matrizes A e B quadradas de dimensões expressivamente grandes (da ordem de dezenas de milhares de linhas e colunas), deve-se calcular a soma e o produto de ambas e armazenar seus resultados nas matrizes S e P , respectivamente. O que interessa são apenas as ESPGs para processamento paralelo e pipeline, sem o auxílio de diagramas de blocos para processamento.

A solução para o modelo paralelo é mais natural do que para o pipeline, por esta razão o problema será resolvido nesta sequência. Entretanto, para ambos os casos os modelos serão, na realidade, híbridos, pois inevitavelmente haverá trechos com algoritmos sequenciais.

Em primeiro lugar deve-se criar e inicializar as matrizes A e B , randomicamente, com números reais. As matrizes S e P não precisam ser inicializadas antecipadamente. Assim, para uma melhor compreensão, a solução será apresentada de forma modular e juntado tudo no final, em uma ESPG única.

Expressões Semânticas de Processamento Geral (ESPGs):

Solução em Paralelo:

iniciaMatrizes := "(A =< RANDOM, FLOAT >) || (B =< RANDOM, FLOAT >)"

calcParalelo := "(S = A + B) || (P = A * B)"

solucaoParalela := *iniciaMatrizes* → *calcParalelo*.

Solução em Pipeline:

iniciaMatrizes := "(A =< RANDOM, REAL >).2 || (B =< RANDOM, REAL >).2"

calcPipeline := "(S = A + B).2 || (P = A * B).2"

solucaoPipeline := *iniciaMatrizes* >|| *calcPipeline*.

Na **Solução em Paralelo** a ESPG *iniciaMatrizes* é composta de tarefas totalmente paralelas, pois não possuem dependência de dados entre si. Já, na ESPG *calcParalelo* as tarefas que envolvem os cálculos de soma e produto dependem da inicialização das matrizes, mas também são independentes entre si. Por fim, na ESPG *solucaoParalela* ambas as tarefas são executadas em sequência, pois como citado anteriormente, *calcParalelo* possui dependência de dados em relação à tarefa *iniciaMatrizes*. Observe a existência do operador de processamento sequencial entre as tarefas *iniciaMatrizes* e *calcParalelo*. Todas as tarefas, por serem paralelas, são intrinsecamente executadas por dois processadores, cada uma.

Na **Solução em Pipeline** as tarefas *iniciaMatrizes* e *calcPipeline* são internamente paralelas, como na **Solução em Paralelo**, porém para ganhar tempo de execução a tarefa *iniciaMatrizes* sinaliza a tarefa *calcPipeline* quando houver dados suficientes para esta última iniciar seu processamento. Isto deve ocorrer até que ambas as matrizes estejam completamente inicializadas.

Como se vê no exemplo 5, a especificação de qualquer tipo e complexidade de processamento pode ser facilmente descrita utilizando-se Expressões Semânticas de Processamento Geral, de forma muito simples e rápida. Além de toda essa facilidade, as ESPGs podem ainda ser incluídas no código fonte do programa final, pois as mesmas são partes integrantes da linguagem Ψ .

A linguagem Ψ é a terceira parte da Metodologia de Paralelização Otimizada proposta neste trabalho de pesquisa e é o tópico do próximo capítulo.

7 Ψ – UMA NOVA LINGUAGEM DE PROGRAMAÇÃO

A linguagem de programação Ψ (em português lê-se: “*pissĩ*” e em inglês “/ps/” ou “*psy*”). É uma letra do alfabeto Grego, que tem o mesmo formato gráfico tanto para forma maiúscula quanto para a minúscula, e possui um significado especial para a linguagem aqui proposta. Psi é um importante complemento às duas primeiras etapas do núcleo desta pesquisa, e é considerada necessária para torná-la completa. Como terceira etapa desta pesquisa, acredita-se que ela seja um recurso de elevado potencial na solução e implementação computacional de um dado problema. Um resumo das especificações da linguagem Ψ encontra-se no **Apêndice A**.

Diante de uma proposta de metodologia de paralelização de processos computacionais, apresentada inicialmente em duas etapas, houve a necessidade de expandir de modo a torná-la mais completa, cuja terceira etapa deveria ser a possibilidade de se codificar o modelo computacional obtido em uma linguagem de programação apropriada, com a capacidade de traduzir as ESPGs para código executável pelo computador.

Ψ é uma linguagem ***implicitamente paralela, orientada a objetos e orientada a padrões***. Ela foi projetada para tornar possível o aproveitamento dos recursos computacionais disponibilizados pelas novas microarquiteturas dos processadores atuais, tanto CPUs quanto GPGPUs e iGPUs. Um compilador específico deve fazer a tradução dos programas fonte escritos em linguagem Ψ , particularmente os

comandos utilizados na codificação de ESPGs, em programas fonte para uma linguagem subjacente, de poder computacional já consagrado, em conjunto com alguns modelos de computação paralela, também já estabelecidos no meio acadêmico, científico e industrial.

Um programa escrito em linguagem Ψ possui uma estrutura dividida em quatro regiões, das quais a primeira e a segunda são opcionais e as demais obrigatórias. Essas regiões possuem características bem definidas e, se em algum momento houver violação de ao menos uma delas, um erro em tempo de compilação será gerado. A extensão adotada para os programas fonte escritos em linguagem Ψ é “**.ppsi**” – que significa **parallel Ψ** - dessa forma, por exemplo, um programa fonte de nome *somaMatrizes* será gravado em disco como “*somaMatrizes.ppsi*”, o qual será reconhecido, automaticamente, pelo compilador da linguagem.

Outra característica muito importante de Ψ é que ela não aceita declarações de classes e nem de métodos no corpo do programa principal, forçando o programador a manter um código limpo e claro. Na necessidade de se declarar classes, essas deverão ser declaradas e implementadas em arquivos fonte separados, chamados de cabeçalhos, para posterior uso (**#uses**) no arquivo fonte do programa principal, tal como ocorre com os arquivos de cabeçalho da linguagem C/C++.

Caso o programador necessite utilizar recursos específicos ou avançados em sua aplicação, poderá fazê-lo importando os módulos necessários da biblioteca padrão ou se desejar, poderá escrever sua própria biblioteca e importá-la como arquivos de cabeçalho. Tanto para a primeira situação, quanto para a segunda o programador deverá fazer essas importações em uma região específica da estrutura de um programa escrito em linguagem Ψ .

Devido o fato de Ψ ser uma linguagem implicitamente paralela, é importante que o programador adquira a habilidade de “**pensar em paralelo**”²¹, quando estiver escrevendo sua aplicação. Dessa forma, ficará mais natural a interação do mesmo com a linguagem.

A Figura 26 ilustra a estrutura de um programa escrito em Ψ e, a seguir são apresentadas algumas das suas características e regras, de forma resumida.

Figura 26 – Estrutura de um programa fonte escrito em linguagem Ψ .

A seguir os blocos constituintes da estrutura de um programa Ψ serão detalhados, conforme mostrados na Figura 26. Também, serão apresentados outros detalhes sobre algumas funcionalidades e alguns comportamentos importantes e inerentes à linguagem Ψ .

²¹ Os seres humanos foram condicionados a pensar de forma sequencial ao desenvolver programas para computadores, pois esse paradigma persiste desde os primórdios da civilização, porém é proposto aqui o **paradigma do raciocínio em paralelo**, pelo menos durante o desenvolvimento de aplicações com a nova linguagem de programação, proposta neste trabalho de pesquisa.

Os programas fonte dos problemas matemáticos mostrados nos exemplos anteriores, serão apresentados em linguagem Ψ .

7.1 Detalhamentos das Regiões de um Programa Ψ

Como visto anterior, um programa escrito na linguagem Ψ contém quatro regiões, a saber: 1. **#uses**; 2. **#start**; 3. **#control** e; 4. **#finish**. A seguir serão apresentados os detalhes de cada uma dessas regiões.

1. **#uses**: Esta região é OPCIONAL, mas deve ser utilizada para a declaração dos nomes das bibliotecas padrão, que são intituladas pacotes de recursos avançados da própria linguagem ou para anexar módulos externos, escritos em Ψ , como cabeçalhos escritos pelo próprio programador. Outros detalhes sobre os arquivos cabeçalhos do usuário serão apresentados mais a diante, neste capítulo. Resumindo, **#uses** é equivalente a **#include** de C/C++, mas diferentemente das linguagens C/C++ ela é escrita uma vez só para importar quantos arquivos da biblioteca padrão ou cabeçalhos se desejar, tal como na linguagem Pascal.
2. **#start**: Esta região é OPCIONAL e informa o início do programa escrito em Ψ . É aqui que o programador deverá trabalhar com parâmetros da linha de comandos e outros comandos pertinentes, tais como declarações globais.
3. **#control**: Esta região é OBRIGATÓRIA e é onde toda a lógica de programação deve ser implementada, principalmente os blocos especiais delimitados pela diretiva “**segp {...}**” que mantém os comandos contendo os repositórios das ESPGs.
4. **#finish**: Esta região do programa Ψ também é OBRIGATÓRIA e deve ser usada para apresentação dos resultados do processamento. Resultados parciais de processamento ou mesmo de dados de entrada implementados na região de controle. A região **#finish** foi incluída na estrutura da linguagem Ψ apenas visando uma melhor organização no fluxo dos dados ao longo do processamento do problema, bem como maior legibilidade na fase de codificação. Como esta é a última região de um programa Ψ , então ela deve, necessariamente, terminar com o comando “**#end.**”, vide Figura 26.

Na linguagem Ψ é permitida a declaração de objetos e variáveis no momento do seu uso, não sendo assim obrigatória a sua declaração no início do programa.

Devido ao fato de Ψ aceitar arquivos de cabeçalho, então se faz necessário um modelo com estrutura pré-determinada para constituir-los. Assim, o item seguinte deste capítulo refere-se a esse assunto.

7.2 Formato do Arquivo de Cabeçalho da Linguagem Ψ

Conforme mencionado anteriormente, qualquer classe que precise ser declarada, deverá ter sua declaração feita fora do arquivo fonte do programa principal. Portanto, foi definida uma estrutura padronizada para arquivos a serem incluídos no código fonte de um programa escrito em Ψ , chamado de cabeçalho, que deverá ser chamado através da sua referência com a diretiva *#uses*. Na chamada do arquivo de cabeçalho, esse deve ter seu nome completo, incluindo a extensão de arquivo, especificado e, se necessário, o caminho completo do diretório onde ele se encontra.

Tal como qualquer arquivo de computador, os arquivos de cabeçalho para Ψ também têm uma extensão padronizada. A extensão de arquivo definida para os cabeçalhos é “*.hpsi*”, que significa **header Ψ** . Por exemplo, um arquivo de cabeçalho contendo a declaração e implementação de uma classe chamada *TCar* poderá ser nomeado como “*car.hpsi*”.

A Figura 27 mostra a estrutura de um arquivo de cabeçalho para a linguagem Ψ , na forma de diagrama de blocos, onde cada um de seus blocos representa uma região de programação que deve ser respeitada, conforme suas especificações.

Todos os elementos da Figura 27 estão incluídos no bloco ***Biblioteca do Usuário*** que aparece na Figura 26 e, também de forma recursiva no bloco de mesmo nome da própria Figura 27.

Figura 27 – Estrutura de arquivos de cabeçalho para códigos fonte escritos em Ψ .

É permitida a declaração de uma, e somente uma classe por arquivo de cabeçalho em Ψ . Assim, se a aplicação necessitar de n classes, então n arquivos de cabeçalho deverão ser implementados, isto é, um arquivo para cada classe declarada e implementada. Esta política garante uma melhor organização e maior legibilidade na fase de codificação do aplicativo em desenvolvimento.

A seguir são feitos os detalhamentos dos blocos constituintes da estrutura mostrada na Figura 27.

1. **#uses**: Esta região segue as mesmas políticas da região *#uses* presente no arquivo fonte do programa principal escrito em Ψ .
2. **#start**: Esta região é OBRIGATÓRIA e informa o local onde a classe deve ser declarada. Aqui não são permitidos acessos à linha de comandos. Declarações globais para este módulo devem ser feitas nesta região.
3. **#implementation**: Esta região é OBRIGATÓRIA e é onde todos os métodos da classe declarada devem ser implementados. O comando “**#end.**” no final do arquivo indica fim da declaração e implementação da classe, vide Figura 27. A falta deste comando de finalização gera um erro em tempo de compilação.

A seguir serão mostrados alguns exemplos de programas fonte, com e sem a chamada de arquivos de cabeçalho, escritos em Ψ .

7.3 Exemplos de Programas Escritos em Linguagem Ψ

Neste item serão ilustrados exemplos de programas fonte, objetivando mostrar a simplicidade que é escrevê-lo em linguagem Ψ . O primeiro exemplo é voltado para cálculos matemáticos envolvendo matrizes, pois essas são as estruturas matemáticas que apresentam maior facilidade de se obter paralelismo.

O exemplo 6 mostra uma solução em linguagem Ψ para cálculos envolvendo matrizes, conforme enunciado no exemplo 5. Isto nos dá a oportunidade de explorar a capacidade de paralelismo implícito oferecida pela linguagem, inclusive implementando ESPGs.

Exemplo 6: Dadas duas matrizes A e B quadradas de dimensões expressivamente grandes (da ordem de dezenas de milhares de linhas e colunas), deve-se calcular a soma e o produto de ambas e armazenar seus resultados nas matrizes S e P , respectivamente. Aproveitando as ESPGs obtidas como solução para o exemplo 5, deve-se codificá-la em linguagem Ψ . NOTA: O intervalo numérico dos dados das matrizes deve ser $[-10,10]$ em ponto flutuante (FLOAT ou REAL).

Das Expressões Semânticas de Processamento Geral (ESPGs) obtidas no exemplo 5, será usada somente a solução em pipeline.

ESPGs para solução em pipeline:

```

iniciaMatrizes := `(A = <RANDOM,FLOAT,[-10,10]>).2 || `(B = <RANDOM,FLOAT,[-10,10]>).2;
calcPipeline   := `(S = A+B).2 || `(P = A*B).2;
solucaoPipeline := iniciaMatrizes >|| calcPipeline;

1. /*
2.  * exemplo6.ppsi
3.  * Por: Marlim Pereira Menezes - Programa de Doutorado da POLI - USP
4.  * Objetivo: Realizar diversos cálculos matriciais usando a linguagem Psi.
5.  */
6. #uses
7.     ppsiLAPACK; // Biblioteca padrão de Álgebra Linear da Psi.
8.
9. #start // Nenhum parâmetro da linha de comandos será usado.
10.    // O comando a seguir escreve uma string na tela e avança uma linha.
11.    display "Calculos matriciais em pipeline, usando Psi.", linefeed;
12.
13. #control
14.    // Variáveis de entrada.
15.    // Declaração das matrizes A, B, S e P no formato 2-D.
16.    A[[]], B[[]], S[[]], P[[]];
17.    /* Aguarda a digitação do número de linhas da matriz A,
18.     * o tipo da variável será definido automaticamente.
19.     */
20.    A.lines = ?; // O valor a ser digitado deverá ser grande, da ordem de 3 ou 4 dígitos
21.    A.columns = A.lines; // Idem anterior, forçando as matrizes serem quadradas.
22.    B.lines = S.lines = P.lines = A.lines;
23.    B.columns = S.columns = P.columns = A.columns;
24.

```

```

25.  /* Bloco paralelo otimizado invocando as diretrizes da
26.  * metodologia ESPG com as diretivas de paralelismo.
27.  */
28.  segp {
29.      model openmp; // Modelo de memória compartilhada, usando OpenMP.
30.      // ESPGs do modelo a ser executado, implementadas em cascata.
31.      iniciaMatrizes := `(A.data=<RANDOM,FLOAT,[-10,10]>).2 || `(B.data=<RANDOM,FLOAT,[-
10,10]>).2;
32.      calcPipeline := `(S.data=A.data+B.data).2 || `(P.data=A.data*B.data).2;
33.      solucaoPipeline := iniciaMatrizes >|| calcPipeline;
34.  } // Fim do bloco segp
35.
36. #finish
37.  display linefeed, "Salvando os resultados em disco:", linefeed;
38.  display "-----", linefeed;
39.  // Armazena as matrizes resultantes, no formato 2D, em arquivos de disco...
40.  display linefeed, "Gravando matriz A.";
41.  file "matrizA.txt", A.data, table, text, append;
42.  display linefeed, "Gravando matriz B.";
43.  file "matrizB.txt", B.data, table, text, append;
44.  display linefeed, "Gravando matriz S.";
45.  file "Soma.txt", S.data, table, text, append;
46.  display linefeed, "Gravando matriz P.";
47.  file "Produto.txt", P.data, table, text, append;
48.  // Mensagens de término de programa na tela do monitor.
49.  display linefeed, "Processamento das matrizes bem sucedido.";
50.  display linefeed, "Fim de programa!";
51.  pause; // Aguarda o usuário pressionar uma tecla, para continuar.
52.
53.  // Nota: Neste ponto o compilador libera toda a memória alocada de volta para o S.O.
54. #end.

```

O código apresentado na listagem do exemplo 6 parece muito simplório, mas este é o objetivo da linguagem Ψ , ou seja, prover solução de alto grau de complexidade com o mínimo de trabalho por parte do usuário.

Nota-se que este código não se parece muito com um programa de computador como de costume. Mas, isto é apenas uma ilusão, pois o seu poder computacional está oculto em sua simplicidade, como será visto nas linhas que se seguem. Para entender um pouco sobre os comandos que aparecem no código fonte do exemplo 6, é importante que se leia o **Apêndice A**.

Explicando o funcionamento do código fonte do exemplo escrito em Ψ :

1. Todas as linhas do programa que contêm texto *verde itálico* são comentários feitos pelo programador e não afetam os resultados, pois são descartados pelo compilador.
2. A região **#uses** abrange as linhas 6 à 8. Na linha 7 há somente uma chamada à biblioteca padrão, que é *ppsiLAPACK*, a qual contém recursos avançados para cálculos de Álgebra Linear. Esses recursos foram programados em

paralelo e pipeline na forma de padrões e estão disponíveis para os modelos de memória compartilhada e distribuída, usando o poder computacional das linguagens e APIs **openmp**, **mpi**, **opencl** e **nvgpu** (“CUDA”). Somente modelos não paralelizáveis são implementados sequencialmente.

3. Na região **#start**, linhas 9 até 12, existe apenas o comando *display*, cuja finalidade, neste caso, é somente mostrar a mensagem entre aspas e avançar uma linha na tela do monitor de vídeo, com o comando *linefeed*.
4. A região **#control**, abrangendo as linhas 13 até 35, é onde está escrito o programa propriamente dito. Ela contém diversos objetos (lembrando que Ψ é orientada a objetos e a padrões) e comandos especiais que tornam o trabalho pesado de codificação que o programador teria, em um código bastante leve e fácil de entender. Na linha 16 tem-se a declaração de diversas variáveis do tipo de uma classe especial que contém a estrutura de matrizes para uso geral, já implementada na forma de padrão. Todas essas matrizes são bidimensionais (observe os pares de colchetes). Na linha 20, o caractere “?” representa uma entrada de dados via teclado, cujo valor terá seu tipo definido na atribuição à propriedade *lines* do “objeto” *A*. A linha 21 informa que a matriz *A* é quadrada. Nas linhas 22 e 23 os valores das propriedades *A.lines* e *A.columns* serão atribuídos para todas as variáveis à sua esquerda. Da linha 28 à linha 34 existe um bloco especial que se acredita existir somente na linguagem Ψ , pois trata do reconhecimento das ESPGs, que estão codificadas nas linhas 31 a 33. O comando da linha 29 informa ao compilador que este bloco deverá ser implementado com os padrões existentes na biblioteca padrão *ppsiLAPACK* usando o modelo de memória compartilhada baseado em OpenMP.
5. A parte final do programa é a região **#finish** que abrange as linhas 36 até 54. Aí há uma novidade, que é o comando *file* contendo diversos parâmetros tais como o nome do arquivo a ser criado/escrito, entre aspas, os dados a serem gravados, a forma como os dados devem ser gravados na mídia, o formato do arquivo (se texto ou binário) e se os dados devem ser gravados em um novo arquivo ou anexados ao final de um arquivo existente. Por exemplo, na linha 41 do programa aparece “*matrizA.txt*” que é o nome do arquivo que será

criado e aberto para gravação se não existir ou somente aberto para gravação em caso contrário, *A.data* é a estrutura contendo os dados a serem gravados na mídia, o comando *table* informa ao compilador que os dados deverão ser gravados na forma de tabela, contendo linhas e colunas, o comando *text* informa ao compilador que os dados deverão ser gravados no formato texto e, por fim o comando *append* significa que é para acrescentar os dados ao final do arquivo especificado, se esse existir, caso contrário um novo arquivo deverá ser criado. Há também diversas ocorrências do comando *display* com *linefeed* aparecendo antes e depois da mensagem. Finalmente, um último comando é o *pause*, que interrompe a execução do programa naquele ponto até que uma tecla seja pressionada, quando, então o processamento continuará normalmente.

Observe que o bloco precedido do comando *segp* é delimitado por um par de chaves “{...}” e o comando “#end.” aparece somente no final do programa.

Exemplo 7: Soma dos n primeiros termos de uma PA em psi, sem usar ESPG.

```

1. // exemplo7.ppsi
2. // Soma dos n primeiros termos de uma PA usando o comando for... do
3. // Por: Marlim Pereira Menezes - Programa de Doutorado da POLI - USP
4. #control
5.   display "Soma dos n primeiros termos de uma PA usando o comando for de psi.", linefeed;
6.   n = 10; // define o tamanho da sequência
7.   for i = 1...n do // Conta de 1 até n, inclusive.
8.     soma += i; // Efetua a soma
9.   #finish
10.  display "soma = ", soma, linefeed; // O resultado impresso será 55.
11. #end.

```

Nota-se que na listagem do exemplo 7 não aparecem as regiões **#uses** e **#start**, vide capítulo 7. É importante lembrar, que até este momento o usuário teve de se inteirar sobre os detalhes da Metodologia proposta ao longo dos capítulos anteriores, que são a extração de paralelismo de um dado problema, a edição da ESPG para a solução obtida e conhecer a linguagem Ψ .

8 CÁLCULO DE DETERMINANTE

Esta metodologia de paralelização do cálculo de determinante de matrizes altamente densas foi elaborada para dar suporte à Metodologia proposta citada nos capítulos anteriores.

Na esfera acadêmica, há diversas propostas para se calcular o determinante de matrizes de ordem n , tanto do tipo completa (sem elementos nulos), quanto dos tipos tri diagonal e esparsa. O advento das CPUs de múltiplos núcleos (multicore) e, mais recentemente das GPGPUs de muitos núcleos (many-core) de baixo custo, têm contribuído para o surgimento de muitas propostas novas para a paralelização de processos e modelos matemáticos que eram de domínio exclusivamente da Computação de Alto Desempenho (HPC). A seguir são apresentados exemplos de estudos que têm alguma relação com a metodologia proposta neste capítulo.

Todos os métodos atualmente usados para se calcular, manualmente, o determinante de matrizes de ordem n são muito difíceis e demandam muito tempo. Normalmente, calcular o determinante de matrizes de ordem $n \geq 4$ requer o uso de muito lápis e papel, sem mencionar o grande cuidado e atenção que se deve ter para evitar erros básicos de aritmética.

A grande importância dos determinantes na *Engenharia Elétrica* foi o fator motivador que nos levou a desenvolver esta nova metodologia. A *Álgebra Linear* apresenta diferentes métodos para se calcular o determinante de matrizes de qualquer ordem de maneira determinística. Contudo, constatou-se que nenhum método apresentado

na álgebra linear nos permite executar este cálculo, para matrizes de ordem superior a três, de forma mais simples do que o proposto deste texto.

Como o número de operações aritméticas básicas necessárias ao cálculo do determinante de uma matriz de ordem n é superior a $(n+1)!$ – fatorial, então serão utilizados daqui em diante os termos baixa, média e alta granularidade de processamento. O **Apêndice B** apresenta um resumo da teoria de determinantes.

Esta etapa desta metodologia tem a finalidade de estabelecer uma técnica adequada à obtenção determinística e imediata do número de submatrizes de ordem 2, contidas em uma matriz de ordem n qualquer, onde $n \in \mathbb{N}, n \geq 2$. Esta regra certamente elevará o custo de processamento no cálculo do determinante de matrizes de ordem maior do que três, mas como benefício, ter-se-á um modelo matemático mais geral.

Inicialmente, este estudo levará em consideração o uso desta metodologia aplicada somente no cálculo de determinantes a partir da *Regra de Cramer*.

Pretende-se demonstrar a abordagem diferente citada no parágrafo anterior, entretanto a proposta é bastante similar à *Regra de Chió* em alguns aspectos.

Esta proposta tem complexidade computacional linear se usada no modo paralelo com hardware adequado, enquanto que se executada de modo sequencial ela é de complexidade cúbica. Neste texto, a metodologia proposta é intitulada *Método das Divisões Internas – MDI*.

O *Método da Eliminação de Gauss-Jordan* tem complexidade computacional cúbica para o modelo sequencial e quadrática para o paralelo. Enquanto que a *Regra de Chió* tem sua complexidade computacional sequencial e paralela próximas da estratégia obtida na metodologia proposta.

A metodologia proposta realiza menos processamento que a regra de Chió no cenário do pior caso, isto por que sua única restrição é $a_{11} \neq 0$, que ocorrerá mais frequentemente apenas se a matriz corrente não for esparsa. Outra contribuição que

se acredita ser interessante é a proposta gráfica chamada *Método Gráfico Prático – MGP*, que dá suporte à estratégia MDI para obter-se, efetivamente, as submatrizes de ordem 2 existentes na matriz de entrada de ordem $n \geq 3$, quando o determinante é calculado manualmente, no papel ou quando se está codificando o método em alguma linguagem de programação.

8.1 Método Proposto

Pretende-se apresentar esta metodologia passo a passo, por meio de uma linguagem matemática clara. Será mostrado como calcular o determinante de uma matriz de ordem n na forma analítica. Também, será apresentado um modelo computacional com um algoritmo sequencial e um paralelo usando trechos de programas escritos em MATLAB e C++, para outras técnicas além da metodologia proposta.

A metodologia aqui proposta é executada em $(n-1)$ iterações, pela redução da matriz, eliminando a primeira linha e primeira coluna da matriz corrente e atualizando todos os $(n-1)^2$ elementos de cada iteração interna, até que o elemento a_{nn} seja alcançado. Este último elemento atualizado é o determinante da submatriz de ordem 2 mais interna que foi dividida pelos produtos dos determinantes das submatrizes dos passos anteriores sucessivamente, resultando no valor do determinante procurado para a matriz de entrada.

Assim, o algoritmo matemático para a metodologia proposta pode ser descrito nos sete passos seguintes:

1. Considerando a matriz A_n de ordem $n > 2$ e $a_{11} \neq 0$;
2. Isolar a 1ª linha e 1ª coluna;
3. Agrupar as entradas na matriz A_n com $(n - 1)^2$ submatrizes de ordem 2, combinando a 1ª linha e 1ª coluna isolada com as demais linhas e colunas de A_n ;
4. Dividir cada submatriz de ordem 2 pelo elemento a_{11} da matriz A_n ;

5. Salvar o resultado obtido no passo 4 na posição do último elemento de cada submatriz de ordem 2 (a_{22});
6. Repetir o processo até que $n = 2$;
7. Multiplicar todos os elementos da diagonal principal da matriz A_n resultante. Este produto é o determinante da matriz de entrada original A_n .

Este método é conveniente, pois ele não requer quaisquer métodos auxiliares para calcular o determinante de matrizes de ordem n . Rotulou-se esta estratégia de **MDI – Método das Divisões Internas**.

8.1.1 MDI – Método das Divisões Internas

O determinante da matriz A_n , mostrado na eq.(9), será resolvido, analiticamente, pelo processo descrito nos passos de 1 a 7 da estratégia MDI. O índice n representa a ordem da matriz A_n corrente, como $n - 1, n - 2, \dots, 2$.

$$A_n = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2(n-1)} & a_{2n} \\ a_{31} & a_{32} & a_{33} & & a_{3(n-1)} & a_{3n} \\ & \vdots & & \ddots & & \vdots \\ a_{(n-1)1} & a_{(n-1)2} & a_{(n-1)3} & \dots & a_{(n-1)(n-1)} & a_{(n-1)n} \\ a_{n1} & a_{n2} & a_{n3} & & a_{n(n-1)} & a_{nn} \end{bmatrix} \quad (9)$$

A primeira redução da matriz A_n , de acordo com o método descrito acima, resulta na eq.(10).

$$|A_{n-1}| = \begin{vmatrix} \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} & \dots & \begin{vmatrix} a_{11} & a_{1n} \\ a_{21} & a_{2n} \end{vmatrix} \\ a_{11} & \ddots & a_{11} \\ \vdots & & \vdots \\ \begin{vmatrix} a_{11} & a_{12} \\ a_{n1} & a_{n2} \end{vmatrix} & \dots & \begin{vmatrix} a_{11} & a_{1n} \\ a_{n1} & a_{nn} \end{vmatrix} \\ a_{11} & & a_{11} \end{vmatrix} \quad (10)$$

Este desenvolvimento resulta na eq.(11).

$$|A_{n-1}| = \begin{vmatrix} a'_{22} & \dots & a'_{2n} \\ \vdots & \ddots & \vdots \\ a'_{n2} & \dots & a'_{nn} \end{vmatrix} \quad (11)$$

Na segunda iteração (ou redução na ordem de A_n), obtém-se a eq.(12) após executar todos os passos do processo.

$$|A_{n-2}| = \begin{vmatrix} \begin{vmatrix} a'_{22} & a'_{23} \\ a'_{32} & a'_{33} \end{vmatrix} & \dots & \begin{vmatrix} a'_{22} & a'_{2n} \\ a'_{32} & a'_{3n} \end{vmatrix} \\ a'_{22} & \ddots & a'_{22} \\ \vdots & & \vdots \\ \begin{vmatrix} a'_{22} & a'_{23} \\ a'_{n2} & a'_{n3} \end{vmatrix} & \dots & \begin{vmatrix} a'_{22} & a'_{2n} \\ a'_{n2} & a'_{nn} \end{vmatrix} \\ a'_{22} & & a'_{22} \end{vmatrix} \quad (12)$$

Finalmente, na $(n-1)$ -ésima iteração do processo, obtém-se $a'_{nn} = |A_2|$ de acordo com a eq.(13).

$$|A_2| = \begin{vmatrix} a'_{(n-1)(n-1)} & a'_{(n-1)n} \\ a'_{n(n-1)} & a'_{nn} \end{vmatrix} \quad (13)$$

Assim, o determinante da matriz A_n é calculado aplicando-se a eq.(14) à diagonal principal da matriz A_n resultante.

$$|A_n| = \det A_n = \prod_{j=1}^n a_{jj} \quad (14)$$

8.1.2 MGP – Método Gráfico Prático

A fim de facilitar e aperfeiçoar a metodologia proposta, decifrou-se o método gráfico envolvido no processo de resolução escrita à mão, por isso é possível calcular o determinante de matrizes de ordem $n \geq 3$ com maior facilidade e segurança, com respeito ao alto grau da qualidade do resultado.

Este método gráfico se aplica ao MDI apresentado anteriormente, bem como à regra de Chió, completamente. Para esta demonstração usou-se a matriz literal de ordem $n = 4$, mostrada na eq.(15).

$$A_4 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & \textcolor{red}{a}_{22} & \textcolor{red}{a}_{23} & \textcolor{red}{a}_{24} \\ a_{31} & \textcolor{green}{a}_{32} & \textcolor{green}{a}_{33} & \textcolor{green}{a}_{34} \\ a_{41} & \textcolor{blue}{a}_{42} & \textcolor{blue}{a}_{43} & \textcolor{blue}{a}_{44} \end{bmatrix} \quad (15)$$

A Figura 28 mostra o método gráfico prático sendo aplicado à primeira iteração na matriz original A_4 .

Figura 28 – Método gráfico prático aplicado à 1ª iteração.

A matriz original A_4 de ordem $n = 4$ tem $n^2 = 16$ elementos de entrada. Após a primeira iteração, esta matriz será reduzida para ordem $n = 3$ contendo apenas $(n - 1)^2 = 9$ elementos de entrada, que foram atualizados nesta iteração, como mostrado na eq.(16).

$$A_3 = \begin{bmatrix} a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{bmatrix} \quad (16)$$

A Figura 29 apresenta o método gráfico prático sendo aplicado à segunda iteração na nova matriz A_3 .

Figura 29 – Método gráfico prático aplicado à 2ª iteração.

Após a segunda iteração esta matriz terá sua ordem $n = 2$ com apenas quatro elementos, que foram atualizados nesta iteração, como mostrado na eq.(17).

$$A_2 = \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix} \quad (17)$$

A terceira iteração, Figura 30, resultará em um simples elemento (a'''_{44}), que é o determinante da última submatriz de ordem 2.

$$\begin{bmatrix} a''_{33} & a''_{34} \\ a''_{43} & a''_{44} \end{bmatrix}$$

Figura 30 – Método gráfico prático para a terceira iteração.

Conforme visto nesta apresentação, o método gráfico é simples de aplicar e descreve adequadamente o processo elementar da estratégia MDI. Vale lembrar que, durante cada iteração dividiu-se todas as submatrizes de ordem 2 pelo elemento a_{11} da matriz corrente, com este diferente de zero.

8.1.3 Modelos Computacionais

Os modelos computacionais apresentados aqui abrangem as metodologias: Eliminação de *Gauss-Jordan*, Regra de *Chió* e a estratégia *MDI*. Será discutida complexidade computacional de algoritmos sequenciais para os três métodos; entretanto, apenas as métricas do algoritmo paralelo do MDI serão apresentadas.

8.1.3.1 Modelos Sequenciais

Como mencionado anteriormente, o algoritmo da eliminação de Gauss-Jordan tem complexidade no tempo $O(n^3)$ para uma matriz de ordem n . Entretanto, para muitas aplicações esta complexidade no tempo é *conveniente* (PRESS *et al.*, 1990), (WILKINSON; ALLEN, 1999), (QUINN, 2004). A listagem 1 apresenta um algoritmo básico escrito para o software MATLAB.

Listagem 1: Modelo Sequencial para o Método da Eliminação de Gauss-Jordan.

```

1. ...
2. det = 1;
3. for i = 1 : n - 1 % laço externo
4.     p = M(i, i);
5.     for k = i + 1 : n
6.         for j = i + 1 : n % laço mais interno
7.             d = M(k, j);
8.             M(k, j) = d - M(k, i) / (p * M(i, j)); % 3 operações
9.         end
10.    M(k, i) = 0;

```

```

11. end
12. end
13. for i = 1 : n
14.     det = det * M(i, i); % 1 operação
15. end
16. ...

```

O algoritmo da regra de Chió tem complexidade computacional no tempo $O(n^3)$ para uma matriz de ordem n . A listagem 2 apresenta um algoritmo básico escrito para a ferramenta MATLAB.

Listagem 2: Modelo Sequencial Regra de Chió.

```

1. ...
2. det = 1;
3. for i = 1 : n - 1 % laço externo
4.     if (M(i, i) ~= 1)
5.         det = det * M(i, i); % 1 operação
6.         p = M(i, i);
7.         for x = i : n
8.             M(i, x) = M(i, x) / p; % 1 operação
9.         end
10.    end
11.    for j = i + 1 : n
12.        for k = i + 1 : n % laço mais interno
13.            M(j, k) = M(j, k) - M(j, i) * M(i, k); % 2 operações
14.        end
15.    end
16. end
17. det = det * M(n, n); % 1 operação
18. ...

```

O algoritmo da estratégia MDI tem complexidade computacional no tempo $O(n^3)$ para uma matriz de ordem n , se executado de forma sequencial. A listagem 3 apresenta um simples algoritmo escrito para a ferramenta MATLAB.

Listagem 3: Modelo Sequencial para a Estratégia MDI.

```

1. ...
2. det = 1;
3. for i = 1 : n % laço externo
4.     det = det * M(i, i); % 1 operação
5.     for j = i + 1 : n
6.         for k = i+1:n % laço mais interno : 3 operações
7.             M(j, k) = M(j, k) - M(j, i) * M(i, k) / M(i, i);
8.         end
9.     end
10. end
11. ...

```

A estratégia MDI apresentada neste texto efetua três operações matemáticas no laço mais interno usado para computar os determinantes das submatrizes de ordem 2, que representam os novos elementos da matriz interna de ordem $(n - iter)$ e uma operação no laço externo. Assim, é possível se calcular o número total de operações

matemáticas elementares ($opIDM_n$) para esta estratégia substituindo os valores de n e $iter$ na eq.(18).

$$opIDM_n = n + 3 * \sum_{iter=1}^{n-1} (n - iter)^2 \quad (18)$$

onde n é a ordem da matriz inicial e $iter$ é o número da iteração do laço externo. Note que $(n - iter)$ é igual ao número da ordem da nova matriz reduzida. Dessa forma, obtém-se a equação geral para o cálculo do número de operações ($opIDM_n$) para a estratégia MDI com a eq.(19).

$$opIDM_n = n + k * \sum_{iter=1}^{n-1} (n - iter)^2 \quad (19)$$

onde k é o número de operações matemáticas no laço mais interno.

A equação implementada no laço mais interno apresentada na listagem 3 é a própria equação original da estratégia MDI, usada para o cálculo do determinante de cada submatriz de ordem 2 existente na matriz sendo reduzida de grau, vide detalhes internos das eq.(10) e eq.(12). Sua simplificação será mostrada mais adiante, neste capítulo.

Foram ignoradas as situações nas quais o elemento $a_{ii} = 0$ em todos os algoritmos apresentados anteriormente.

8.1.3.2 Modelo Paralelo Tradicional

A estratégia da metodologia proposta aqui possui complexidade computacional $O(n)$ usando o número máximo de processadores ou threads igual a $(n - 1)^2$. Este número decresce de uma unidade a cada iteração externa até alcançar o valor mínimo de recursos computacionais igual a 2 (última submatriz de ordem 2). O algoritmo efetua três operações aritméticas no laço mais interno para calcular os determinantes das submatrizes de ordem 2, que representam os novos elementos da nova matriz interna de ordem $(n - i)$ e uma operação no laço externo. Porém, as operações do

laço mais interno devem ocorrer simultaneamente em um único passo, mas para isto ocorrer é necessário que se tenha uma infraestrutura de hardware consistente com a metodologia proposta.

A listagem 4 apresenta um trecho do código escrito em linguagem C++ com as diretivas OpenMP para sua paralelização.

Listagem 4: Trecho de Modelo Paralelo com Memória Compartilhada – OpenMP para a Estratégia MDI.

```

1. ...
2. omp_set_num_threads(numberProcs);
3. // cria região paralela com shared e private
4. #pragma omp parallel shared(...) private(...)
5. {
6. ...
7. for(iter = 0; iter < nLin; iter++) { // laço externo
8.     if(thread_id == 0) {
9.         // checa se o elemento aiter, iter = 0
10.        if(M[iter * (nLin + 1)] == 0)
11.            sinal += swapPivotZero(iter, M);
12.        det *= M[iter * (nLin + 1)];
13.    }
14.    // distribui as linhas da matriz de (lin-iter)
15.    // entre as threads disponíveis
16.    size_lin = (nLin - iter) / n_threads;
17.    ...
18.    M_aux = M[iter * (nLin + 1)];
19.    for(i = start_i; ((i < end_i)&&(i < nLin)); i++) {
20.        ...
21.        // laço mais interno: calcula os determinantes
22.        // das matrizes de ordem 2 no laço controle
23.        for(j = iter + 1; j < nLin; j++) {
24.            M[i_lin + j] = M[i_lin + j] - M[i_inter] * M[iter_lin + j] / M_aux;
25.        } // for j
26.    } // for i
27.    #pragma omp barrier
28. } // for iter
29. } // omp parallel
30. ...
31. return (det);
32. ...

```

Neste pseudocódigo é verificado se o valor de a_{ij} é zero ou não. Este elemento foi considerado o pivô, que será usado para acumular o valor do determinante enquanto a rotina está rodando. Isto é realizado por meio da função chamada *swapPivotZero(iter, M)*. O balanceamento de carga das linhas da matriz corrente é executado com a variável *size_lin*. A variável *M_aux* é usada para acelerar o processo de cálculo. Note-se que o acesso aos elementos da matriz é feito utilizando-se da técnica de linearização, que permite alcançar um ganho de desempenho de cerca de 10% (resultado obtido empiricamente).

No laço mais interno, o cálculo do determinante é feito aplicando-se a eq.(20), que é a equação original da metodologia corrente, conforme citado anteriormente:

$$M[i_lin + j] = M[i_lin + j] - \frac{M[i_iter] * M[iter_lin + j]}{M_aux} \quad (20)$$

onde esta equação é equivalente à eq.(21):

$$\frac{\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}}{a_{11}} = \frac{a_{11}a_{22} - a_{21}a_{12}}{a_{11}} = a_{22} - \frac{a_{21}a_{12}}{a_{11}} \quad (21)$$

com seus índices apropriados.

A eq.(21) já antecipa a melhoria no desempenho do algoritmo MDI, a qual será explorada, futuramente, como aperfeiçoamento desta metodologia aqui proposta.

A Figura 31 mostra o *speedup* entre o código sequencial e o paralelo, executado com dois e quatro processadores.

Figura 31 – Speedup para 2 e 4 processadores.

A Figura 32 apresenta a comparação do *speedup* para dois e quatro processadores em relação ao tempo de processamento para o modelo sequencial, com matrizes de ordens na faixa entre $n = 300$ até $n = 5000$. Os intervalos entre as dimensões das matrizes testadas é o apresentado na figura 32.

Figura 32 – Resultados dos tempos de processamento.

Os tempos obtidos nos testes para as matrizes de ordens mostradas na figura 32 estão apresentados no quadro 8.1 abaixo. Estes valores constituem a média aritmética de duas medições efetuadas com matrizes de ordens conforme apresentadas, porém com dados diferentes.

Quadro 8.1: Tempos obtidos nos testes das matrizes com dimensões de 300 até 5000.

Dimensão	Tempo Serial	Tempo 2 CPUs	Tempo 4 CPUs
300	0,062	0,031	0,023
400	0,078	0,078	0,046
500	0,296	0,148	0,078
600	0,514	0,249	0,140
700	0,803	0,405	0,210
800	1,224	0,601	0,328
900	1,747	0,850	0,437
1000	2,379	1,170	0,842
2000	19,273	9,492	4,937
3000	64,973	31,941	16,450
4000	153,929	75,590	38,961
5000	300,074	147,412	75,473

Os dados utilizados para os testes usando uma matriz de ordem 300 foram os mesmos para os algoritmos sequencial e paralelo, tanto com 2 CPUs quanto com 4 CPUs em cada bateria de testes independente, assim como para todas as demais dimensões das matrizes. Ou seja, na primeira bateria de testes cada matriz teve seus valores gerados aleatoriamente e usada nos três modos de processamento, com 1, 2 e 4 CPUs sem alterações. Para a segunda bateria de testes cada matriz teve seus valores gerados aleatoriamente, assim como ocorreu na primeira bateria de testes. Em seguida se calculou a média aritmética dos tempos obtidos nas duas baterias de testes, gerando, assim, os resultados apresentados no quadro 8.1.

Todos os testes foram executados com as configurações de hardware e software listados na Tabela 8.1.

Tabela 8.1: Configurações de hardware e software usados nos testes.

Modelo da CPU	Intel i7-2600 – 3.4GHz – 4 cores – 8 threads
Modo Hyper-threading	Recurso desativado (apenas 4 núcleos físicos ativos)
Memória	12 GBytes – DDR3 – 1600MHz
Compilador C++	g++-4.exe
IDE	NetBeans 7.0.1
OpenMP	Versão 3.0
Plataforma Computacional	Windows 7 Professional 64 bits

8.2 MDIM – Método das Divisões Internas Melhorado

Considerando-se uma matriz quadrada A de ordem n ($n \in \mathbb{N} \mid n > 1$), o cálculo do seu determinante utilizando o algoritmo padrão do MDI consome mais tempo para ser realizado do que usando o algoritmo da Regra de Chió, nas mesmas condições, significando, claramente, que o algoritmo MDI original, se executado sequencialmente é mais lento do que a Regra de Chió. Entretanto, após as melhorias feitas no algoritmo MDI, esse passou a ser mais rápido do que a Regra de Chió, com n operações de vantagem.

Dizer que o MDIM supera a Regra de Chió em n operações, significa dizer que a Regra de Chió realiza n operações matemáticas a mais do que o novo algoritmo do Método das Divisões Internas. A eq.(21) é que torna esta melhoria significativa possível na estratégia MDI. Assim, considerando-se que x_{mdi} , x_{rc} , x_{mdim} sejam os

números totais de operações realizadas pelos algoritmos MDI, Regra de Chió e MDIM, respectivamente, pôde-se escrever as equações a seguir.

$$X_{mdi(n)} = X_{rc(n)} + y \quad (22)$$

Onde, na eq.(22) y é o número de operações realizadas a mais pelo algoritmo do MDI em relação ao da Regra de Chió ($y > n$).

$$X_{mdim(n)} = X_{rc(n)} - n \quad (23)$$

Onde, na eq.(23) n é a ordem da matriz.

O gráfico mostrado na Figura 33, a seguir, ilustra as diferenças entre o desempenho dos algoritmos MDI, MDIM e Regra de Chió, conforme citado anteriormente.

Figura 33 – Comparação entre os números de operações aritméticas efetuadas pelos algoritmos MDI, Regra de Chió e MDIM.

Analisando-se a Figura 33, é fácil notar que a curva do algoritmo MDIM é a que apresenta melhor desempenho, enquanto que a curva do algoritmo MDI apresenta o pior desempenho. Assim, quanto mais para direita, ou seja, mais distante do eixo

vertical, menor é o número de operações aritméticas a serem realizadas pelo método. Entretanto, isto não significa que o algoritmo computacional para o método de menor número de operações aritméticas seja o mais rápido, pois isso depende das técnicas aplicadas em seu desenvolvimento. Portanto, existe o risco do melhor método ter a menor velocidade de execução.

As equações utilizadas para a demonstração de desempenho dos algoritmos são todas de terceiro grau, pois elas representam o número de operações aritméticas realizadas para o processamento total do cálculo do determinante, ao invés do número de iterações realizadas pelos laços existentes nos algoritmos. Essas equações são mostradas abaixo, onde n representa a ordem da matriz.

A eq.(24) nos permite obter o número total de operações aritméticas realizadas pelo algoritmo do Método das Divisões Internas (MDI).

$$x_{mdi(n)} \cong \frac{2n^3 - 3n^2 + 3n}{2} \quad (24)$$

A eq.(25) permite-nos obter o número total de operações aritméticas realizadas pelo algoritmo da Regra de Chió.

$$x_{rc(n)} \cong \frac{4n^3 - 3n^2 + 11n}{6} \quad (25)$$

A eq.(26) permite-nos obter o número total de operações aritméticas realizadas pelo algoritmo do MDIM.

$$x_{mdim(n)} \cong \frac{4n^3 - 3n^2 + 5n}{6} \quad (26)$$

Analisando cuidadosamente as equações anteriores, nota-se que a eq.(26) é a que apresenta melhor resultado numérico. É importante lembrar que, apesar das equações anteriores serem todas de terceiro grau, os resultados que nos interessam são apenas aqueles referentes ao primeiro quadrante do gráfico de cada função,

pois é a região onde ambas as coordenadas do plano cartesiano são positivas. Isto é, uma matriz não pode ter ordem $n \leq 0$ e portanto, o número de operações aritméticas realizadas pelo computador não pode ser negativo.

Assim, com a melhoria feita no algoritmo MDI, a nova sequência de passos para o modelo matemático da estratégia MDIM – Método das Divisões Internas Melhorado é:

1. Dada a matriz A_n de ordem $n > 2$ e $a_{11} \neq 0$;
2. Isolar a 1ª linha e 1ª coluna;
3. Multiplicar todos os elementos da diagonal principal da matriz A_n resultante. Este produto é o determinante da matriz de entrada original A_n .
4. Agrupar as entradas na matriz A_n com $(n-1)^2$ submatrizes de ordem 2, combinando a 1ª linha e 1ª coluna isolada com as demais linhas e colunas de A_n ;
5. Dividir o elemento a_{ji} pelo elemento a_{ij} e guardar o resultado em uma variável, por exemplo, $x_{ji} = a_{21} / a_{11}$. Este processo deve se repetir para cada nova submatriz de ordem $(n-1)$;
6. Subtrair o produto $a_{12} * x_{ji}$ de a_{22} e guardar o resultado em a_{22} , considerando cada submatriz de ordem 2;
7. Repetir o processo até que $n = 2$;

A seguir a eq.(21) será reescrita, para análise:

$$\frac{\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}}{a_{11}} = \frac{a_{11}a_{22} - a_{12}a_{21}}{a_{11}} = a_{22} - \frac{a_{12}a_{21}}{a_{11}}$$

Vê-se claramente que não é possível continuar a simplificação da equação, nos restando somente rearranjar os elementos da sua parte fracionária, vide eq.(27).

$$a_{22} - \frac{a_{12}a_{21}}{a_{11}} = a_{22} - a_{12} \frac{a_{21}}{a_{11}} \quad (27)$$

Separando a eq.(27) em duas partes se obtêm as eq.(28) e eq.(29):

$$X_{ji} = \frac{a_{21}}{a_{11}} \quad (28)$$

e

$$a_{22} - \frac{a_{12}a_{21}}{a_{11}} = a_{22} - a_{12} * X_{ji} \quad (29)$$

Como se vê, claramente, no pseudo algoritmo MDIM, tal como no pseudo algoritmo MDI não é necessário atualizar os elementos da primeira linha e primeira coluna de cada submatriz de ordem $(n-1)$, inclusive da própria matriz de ordem n , durante o cálculo dos elementos da submatriz seguinte, como acontece na Regra de Chió, vide Apêndice B. Com isto, já se ganha na velocidade do cálculo do determinante, pois os acessos desnecessários a essas posições de memória são evitados. Ainda, de acordo com as eq.(28) e eq.(29) ganham-se n iterações com uma operação aritmética cada – em relação à Regra de Chió –, dispondo-as conforme mostrado nas listagens 5 e 6, modelo sequencial e paralelo, respectivamente.

O algoritmo sequencial da estratégia MDIM é mostrado na listagem 5, em C++ e sua listagem completa encontra-se no **Anexo A**.

Listagem 5: Modelo sequencial para a Estratégia MDIM.
--

```

1. ...
2.   for(int i = 0; i < nLin; i++) {
3.       Xii = M->mat[i * (nLin + 1)];
4.       // Tratando pivô = 0.
5.       if(Xii == 0 && i != nLin) {
6.           sinal += swapPivotZero(i, nLin, M->mat);
7.           Xii = M->mat[i * (nLin + 1)];
8.       }
9.       detM *= Xii;
10.      for(j = i + 1; j < nLin; ++j) { // Percorre as colunas da linha i
11.          Xji = M->mat[j * nLin + i] / Xii; // MDIM
12.      } // Obtendo as submatrizes de M.

```

```

13.     for(k = i + 1; k < nLin; ++k) { // Percorre as linhas da coluna j
14.         M->mat[j * nLin + k] -= M->mat[i * nLin + k] * Xji; // MDIM
15.     } // for k
16. } // for j
17. } // for i
18. // Acerta o sinal do determinante...
19. if(sinal % 2 != 0)
20.     detM = -detM;
21. ...

```

O algoritmo paralelo da estratégia MDIM é mostrado na listagem 6, escrito em linguagem C++, usando o modelo de memória compartilhada com OpenMP.

Listagem 6: Trecho de programa paralelo com Memória Compartilhada usando OpenMP para a Estratégia MDIM.

```

1. ...
2.     omp_set_num_threads(numCPUs);
3.     // Início do cálculo
4.     #pragma omp parallel shared(nLin,sinal,detM) private(iter,i,j)
5.     {
6.         int n_threads, id_thread, inicio_i, fim_i, size_lin, iter_nLin, i_nLin, i_iter;
7.         float Xii, Xij;
8.         n_threads = omp_get_num_threads();
9.         id_thread = omp_get_thread_num();
10.        for(iter = 0; iter < nLin; iter++) {
11.            Xii = M->mat[iter * (nLin + 1)];
12.            // Tratando pivô = 0.
13.            if(id_thread == 0) {
14.                if(Xii == 0 && iter != nLin) {
15.                    sinal += swapPivotZero(iter, nLin, M->mat);
16.                    Xii = M->mat[iter * (nLin + 1)];
17.                }
18.                detM *= Xii;
19.            }
20.            size_lin = (nLin - iter) / n_threads;
21.            if(((nLin - iter) % n_threads) != 0)
22.                size_lin++;
23.            inicio_i = iter + 1 + (id_thread * size_lin);
24.            fim_i = inicio_i + size_lin;
25.            iter_nLin = iter * nLin;
26.            for(i = inicio_i; ((i < fim_i) && (i < nLin)); i++)
27.            {
28.                i_nLin = i * nLin;
29.                i_iter = i_nLin + iter;
30.                Xji = M->mat[i_iter] / Xii; // MDIM
31.                for(j = iter + 1; j < nLin; ++j) {
32.                    M->mat[i_nLin + j] -= M->mat[iter_nLin + j] * Xji; // MDIM
33.                } // for j
34.            } // for i
35.            #pragma omp barrier
36.        } // for iter
37.    } // omp parallel shared
38.    // Acerta o sinal do determinante...
39.    if(sinal % 2 != 0)
40.        detM = -detM;
41. ...

```

A listagem completa do programa pode ser consultada no Anexo B.

A Figura 34 apresenta comparações entre os tempos de execução dos algoritmos MDI versus MDIM para matrizes de ordem $n = 300$ até $n = 6000$, com números inteiros versus números reais.

Figura 34 – Comparação dos tempos de execução do algoritmo MDI versus MDIM para matrizes de ordem $n = 300$ até $n = 6000$, com números inteiros versus números reais.

Observando-se a Figura 34 vê-se nitidamente que a estratégia MDIM é mais eficiente que a MDI. Ainda, a partir de uma análise mais apurada dos dados no gráfico, pôde-se concluir que os tempos de execução de ambos os algoritmos, MDI e MDIM, para números inteiros e reais são muito próximos, principalmente para as matrizes de ordens mais elevadas. Isto significa que não há relevância na classificação de algoritmos, do ponto de vista qualitativo, em função do tipo dos dados numéricos utilizados no processamento ser inteiro ou real (float).

A Figura 35 mostra o speedup para o algoritmo MDIM Sequencial, em relação ao algoritmo MDIM Paralelo, para as mesmas matrizes utilizadas nos testes mostrados na Figura 34. O algoritmo paralelo foi executado em 2 e 4 processadores.

Figura 35 – Ganho de velocidade (speedup) para MDIM Sequencial versus Paralelo com 2 e 4 processadores.

A Figura 36 mostra, de forma linear, a evolução do tempo de execução para a estratégia MDIM sequencial versus a mesma estratégia em paralelo sendo executada em 2 e 4 processadores.

Figura 36 – Análise linear da evolução do tempo de execução do MDIM Sequencial e Paralelo, executado com 2 e 4 processadores.

Como o gráfico da Figura 36 apresenta os resultados de forma linear, fica impossível se analisar o que ocorre com o desempenho dos algoritmos quando a matriz possui ordem $n \leq 1000$. Sendo assim, optou-se por mostrar esses detalhes na Figura 37, através do gráfico de escala logarítmica, de modo a facilitar a análise de desempenho da execução do MDIM, mantendo os dados originais.

Figura 37 – Análise logarítmica da evolução do tempo de execução do MDIM Sequencial e Paralelo, executado com 2 e 4 processadores.

Todos os testes foram executados com as configurações de hardware e software listados na Tabela 8.2.

Tabela 8.2: Configurações de hardware e software usados nos testes da estratégia MDIM.

Modelo da CPU	Intel i7-3770 – 3.4GHz – 4 cores – 8 threads
Modo Hyper-threading	Recurso desativado
Memória	16 GBytes – DDR3 – 1600MHz
Compilador C++	g++-4.exe
IDE	NetBeans 7.3
OpenMP	Versão 3.0
Plataforma Computacional	Windows 7 Professional 64 bits

9 ESTUDOS DE CASO E RESULTADOS

Os estudos de caso apresentados neste capítulo servirão para demonstrar a eficiência da Metodologia de Paralelização e Otimização proposta, considerando-se suas três etapas:

- a) Extração de paralelismo a partir do modelo dado.
- b) Obtenção da Expressão Semântica de Processamento Geral – ESPG.
- c) Codificação na linguagem Ψ .

Além das etapas citadas acima, ainda serão apresentados os algoritmos listados abaixo:

- d) Codificação Sequencial em C++.
- e) Codificação Paralela em C++ com OpenMP.

Em se tratando de metodologias para a Computação Paralela de Alto Desempenho, acredita-se que a melhor área do Conhecimento Humano para se testar algoritmos computacionais é a Matemática. Por esta razão, decidiu-se iniciar os estudos de caso com pequenas demonstrações, mas suficientes para corroborar as propostas aqui apresentadas, na área de Álgebra Linear, onde é apresentado um problema envolvendo operações com matrizes; problema esse já conhecido da comunidade acadêmica e um no campo de Sistemas de Potência da grande área da Engenharia Elétrica, onde serão utilizados os passos envolvidos no Cálculo de Curto Circuito em Sistemas de Potência.

Para todos os estudos de caso serão apresentados os seus algoritmos nos modelos sequencial, paralelo e pipeline em linguagem Ψ , entretanto serão apresentados somente os modelos sequencial e paralelo em linguagem C++, com a API OpenMP para as soluções paralelas. O objetivo da solução sequencial é a obtenção do “*speedup*” entre os métodos utilizados – sequencial e paralelo.

Todos os estudos de caso foram testados em um computador com as configurações de hardware e software apresentadas na tabela 9.1, a seguir.

Tabela 9.1: Configurações de hardware e software usados nos estudos de caso.

Modelo da CPU	Intel i7-3770 – 3.4GHz – 4 cores – 8 threads
Modo Hyper-threading	Recurso desativado ²² (apenas 4 núcleos físicos ativos)
Memória	16 GBytes – DDR3 – 1600MHz
GPGPU	NVIDIA GeForce GTX 460, 256 bits (336 CUDA cores)
Compilador C++	G++-4.exe (versão 4.6 – Windows 32 bits)
IDE	NetBeans 7.3 para Windows 32 bits
OpenMP	Versão 3.0
Plataformas Computacionais	Linux Ubuntu 12.04 LTS 64 bits e Windows 8 Professional 64 bits

9.1 Estudo de Caso 1: Multiplicação de Matrizes

Dadas as matrizes A e B , quadradas de ordem n , tal que $n = 6000$, calcular seu produto e armazenar o resultado na matriz C , eq.(30), de mesmas dimensões.

$$C = A * B \quad (30)$$

9.1.1 Solução Analítica

A eq.(31) mostra a forma geral para o cálculo do produto de matrizes, neste caso, especificamente as matrizes de entrada A e B , com matriz de saída C quadradas e de dimensões $n \times n$, com n expressivamente grande.

²² Em programação paralela é importante desativar-se este recurso de hardware, pois o mesmo pode causar perda real de desempenho na execução dos programas paralelos.

$$c_{ij} = \sum_{k=0}^{n-1} a_{ki} * b_{jk} \quad (31)$$

A eq.(31) é o modelo matemático original que nos fornece a solução desejada, sem dificuldades. Tradicionalmente, a solução algébrica desta equação consiste em se fazer os cálculos elemento a elemento, até esgotar todos eles. Todas as matrizes envolvidas devem ter, necessariamente, a mesma ordem, como exigência para este estudo de caso.

A representação gráfica para a solução da eq.(31) é mostrada na Figura 38. Note que o processo é muito simples, porém extremamente dispendioso.

Figura 38 – Processo gráfico para a multiplicação de duas matrizes densas, de ordem n .

A solução para o processo ilustrado na Figura 38 é mostrado de forma algébrica nas eq.(32) e eq.(33), para ordem $n = 2$.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} \quad (32)$$

Resolvendo a eq.(32), algebricamente, obtém-se a eq.(33).

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix} \quad (33)$$

9.1.2 Extração de Paralelismo

O principal objetivo deste item é obter aumento significativo na velocidade de processamento computacional, para a solução do problema dado, sem o comprometimento do resultado esperado e, que na maioria das vezes já é conhecido, pois em algum momento uma solução sequencial foi realizada.

Assim, a extração de paralelismo consiste, basicamente, em uma análise criteriosa do modelo inicial do problema, visando fragmentá-lo em partes independentes, de forma que essas partes possam ser executadas em diversos processadores, simultaneamente. Ou seja, este é um problema de otimização e pode ser realizado aplicando-se a Metodologia apresentada no capítulo 5 deste Texto.

Extrair paralelismo de operações envolvendo matrizes é relativamente simples, pois esses cálculos apresentam paralelismo natural, ou seja, a solução é trivial, bastando se escolher a melhor forma de fazê-lo. Assim, analisando-se o problema inicial do ponto de vista da Figura 38, vê-se claramente que há grande quantidade de independência de dados que pode ser transformada em diversas tarefas independentes.

A Figura 38 sugere que é possível se ter n^2 tarefas sendo executadas concorrentemente, onde cada uma realizará o cálculo de cada elemento da matriz C . Utilizar essa estratégia é bastante interessante, porém para a sua realização plena é necessário que se tenha um hardware com infraestrutura suficiente para suportar as dimensões das matrizes envolvidas no problema e, isto poderá elevar o seu custo.

Por outro lado, se for possível gerenciar o uso dos processadores disponíveis no hardware, ou seja, controlar a distribuição da carga de dados a ser processada, essa solução pode, perfeitamente, vir a ser uma boa proposta. Esta parte da solução se chama balanceamento de carga. Decidiu-se adotar uma abordagem um pouco diferente da citada no parágrafo anterior, que é fazer com que cada tarefa realize os cálculos de todos os elementos de cada linha da matriz C , dessa forma tem-se

apenas n tarefas, pois por si só ela já garante um aumento considerável de velocidade na computação do problema em análise.

Assim, a Figura 39 mostra o esquema gráfico da solução adotada.

Figura 39 – Estratégia de multitarefa adotada para a multiplicação de duas matrizes densas, de ordem n .

As tarefas mostradas na Figura 39 estão relacionadas diretamente apenas à matriz C . Cada tarefa efetuará os cálculos dos n elementos da matriz C .

9.1.3 Obtenção das ESPGs

As expressões semânticas para o modelo matemático otimizado resultante da fase de extração de paralelismo são apresentadas a seguir. Neste estudo de caso, são apresentadas três expressões semânticas.

a) ESPG sequencial:

$$\begin{aligned} seqMatrixMul := & ' (A = < RANDOM, REAL, [-30, 30] >) \rightarrow \\ & ' (B = < RANDOM, REAL, [-30, 30] >) \rightarrow ' (C = A * B) \end{aligned} \quad (34)$$

b) ESPG paralela:

$$\begin{aligned} parMatrixMul := & [' (A = < RANDOM, REAL, [-30, 30] >) || \\ & ' (B = < RANDOM, REAL, [-30, 30] >)] \rightarrow ' (C = A * B).4 \end{aligned} \quad (35)$$

c) ESPG pipeline:

$$\text{pipeMatrixMult} := [\text{ `` } (A = \langle \text{RANDOM, REAL, } [-30, 30] \rangle) . 2 | | \text{ `` } (B = \langle \text{RANDOM, REAL, } [-30, 30] \rangle) . 2 > | | \text{ `` } (C = A * B) . 2 \text{ `` }] \quad (36)$$

Cada expressão obtida nesta etapa será codificada em um programa fonte em linguagem Ψ , posteriormente codificada em linguagem C++ com o uso de OpenMP, quando necessário. Como dito anteriormente, somente as soluções sequencial e paralela serão codificadas em C++.

9.1.4 Codificação em Ψ

A solução sequencial para este tipo de operação é trivial e já bem difundida no meio acadêmico, portanto, com exceção da sua expressão semântica nada há de novidade. A eq.(33) apresenta o modelo matemático já otimizado para este problema e pode ser usado dessa forma, tanto para o modelo computacional sequencial quanto para o paralelo e o pipeline.

A ESPG na express.(34) representa o modelo computacional sequencial para a solução do problema dado. Essa expressão indica que os cálculos serão realizados completamente de forma sequencial. Assim, primeiro inicializa-se a matriz A com valores reais gerados, randomicamente, pelo sistema de forma sequencial, em seguida inicializa-se a matriz B da mesma maneira e, por fim calcula-se o valor da variável c_{ij} um de cada vez, para toda a matriz C . A listagem 7 mostra a solução codificada em Ψ , sequencial para a solução obtida.

Listagem 7: Modelo sequencial de programa em Ψ para multiplicação de matrizes de ordem n .

```

1. //
2. // multMatrixSeq.ppsi
3. // Multiplicação de matrizes usando a linguagem Psi.
4. //
5. #uses
6.   ppsilAPACK; // Usa a biblioteca padrão especial de Álgebra Linear.
7. #start
8.   display "Exemplo sequencial de multiplicação de matrizes em Psi.", linefeed;
9. #control
10.  A[[]], B[[]], C[[]]; // Declaração das matrizes A, B e C em 2-D.
11.  A.lines = 6000; // Inicia a ordem das matrizes A, B e C com o valor 6000.
12.  C.lines = C.columns = B.lines = B.columns = A.columns = A.lines;
13.  segp { // Início do bloco da ESPG (SEGP - Semantic Expression of General Processing)
14.    seqMatrixMul := `(A=<RANDOM,REAL,[-30,30]>)->`(B=<RANDOM,REAL,[-30,30]>)->`(C=A*B);
15.  }
16. #finish
17.   display "C = A * B = ", linefeed; // Mostra mensagem e pula uma linha.
18.   display C.data, table, linefeed.2; // Mostra a matriz C na tela em 2-D e pula 2 linhas.

```

```

19. display "Armazenando as matrizes em disco:", linefeed;
20. file "matrizA.txt", A.data, table, text, write; // Grava matriz 2D e formato ASCII.
21. file "matrizB.txt", B.data, table, text, write; // Idem para a matriz B.
22. file "matrizC.txt", C.data, table, text, write; // Idem para a matriz C.
23. display "Pressione uma tecla para terminar! ";
24. pause; // Aguarda uma tecla ser pressionada, para prosseguir.
25. #end.

```

O programa Ψ para o modelo paralelo é mostrado na listagem 8. Note que há duas diferenças, em relação à versão sequencial, que são o acréscimo do comando *model* com o modelo de memória que será usado no escopo da *segg* e a ESPG que deve representar a codificação em paralelo.

Listagem 8: Modelo paralelo de programa em Ψ para multiplicação de matrizes de ordem n .

```

1. //
2. // multMatrizPar.ppsi
3. // Multiplicação de matrizes usando a linguagem Psi.
4. //
5. #uses
6.   ppsiLAPACK; // Usa a biblioteca padrão especial de Álgebra Linear.
7. #start
8.   display "Exemplo paralelo de multiplicação de matrizes em Psi.", linefeed;
9. #control
10.  A[[]], B[[]], C[[]]; // Declaração das matrizes A, B e C em 2-D.
11.  A.lines = 6000; // Inicia a ordem das matrizes A, B e C com o valor 6000.
12.  C.lines = C.columns = B.lines = B.columns = A.columns = A.lines;
13.  segg { // Início do bloco da ESPG (SEGP - Semantic Expression of General Processing)
14.    model openmp; // Modelo de memória compartilhada usando OpenMP.
15.    parMatrixMul := ["(A=<RANDOM,REAL, [-30,30]>)" || "(B=<RANDOM,REAL, [-30,30]>)" -
>"(C=A*B).4;
16.  }
17. #finish
18.   display "C = A * B = ", linefeed; // Mostra mensagem e pula uma linha.
19.   display C.data, table, linefeed.2; // Mostra a matriz C na tela em 2D e pula 2 linhas.
20.   display "Armazenando as matrizes em disco:", linefeed;
21.   file "matrizA.txt", A.data, table, text, write; // Grava matriz em 2D e formato ASCII.
22.   file "matrizB.txt", B.data, table, text, write; // Idem para a matriz B.
23.   file "matrizC.txt", C.data, table, text, write; // Idem para a matriz C.
24.   display "Pressione uma tecla para terminar! ";
25.   pause; // Aguarda uma tecla ser pressionada, para prosseguir.
26. #end.

```

O programa Ψ para o modelo pipeline é mostrado na listagem 9. Note que há apenas uma diferença em relação à versão paralela, que é a ESPG que deve representar a codificação em pipeline.

Listagem 9: Modelo pipeline de programa em Ψ para multiplicação de matrizes de ordem n .

```

1. //
2. // multMatrizPipe.ppsi
3. // Multiplicação de matrizes usando a linguagem Psi.
4. //
5. #uses
6.   ppsiLAPACK; // Usa a biblioteca padrão especial de Álgebra Linear.
7. #start
8.   display "Exemplo pipeline de multiplicação de matrizes em Psi.", linefeed;
9. #control
10.  A[[]], B[[]], C[[]]; // Declaração das matrizes A, B e C em 2-D.
11.  A.lines = 6000; // Inicia a ordem das matrizes A, B e C com o valor 6000.
12.  C.lines = C.columns = B.lines = B.columns = A.columns = A.lines;
13.  segg { // Início do bloco da ESPG (SEGP - Semantic Expression of General Processing)

```

```

14.     model openmp; // Modelo de memória compartilhada usando OpenMP.
15.     pipeMatrixMul := [{"(A=<RANDOM,REAL,[-30,30]>).2|"}|(B=<RANDOM,REAL,[-
16.     }
17. #finish
18.     display "C = A * B = ", linefeed; // Mostra mensagem e pula uma linha.
19.     display C.data, table, linefeed.2; // Mostra a matriz C na tela em 2-D e pula 2 linhas.
20.     display "Armazenando as matrizes em disco:", linefeed;
21.     file "matrizA.txt", A.data, table, text, write; // Grava matriz em 2D e formato ASCII.
22.     file "matrizB.txt", B.data, table, text, write; // Idem para a matriz B.
23.     file "matrizC.txt", C.data, table, text, write; // Idem para a matriz C.
24.     display "Pressione uma tecla para terminar! ";
25.     pause; // Aguarda uma tecla ser pressionada, para prosseguir.
26. #end.

```

9.1.5 Codificação em C++

A listagem 10 mostra o equivalente em C++ do programa sequencial codificado em Ψ . O código completo encontra-se no **Anexo C**.

Listagem 10: Modelo sequencial de programa em C++ para multiplicação de matrizes de ordem n .

```

1. /*
2.  * File:    seqMatMul.cpp
3.  * Author:  Marlim P. Menezes
4.  * Progama: Doutorado em Ciências - Engenharia Elétrica
5.  * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
6.  * Objetivo: Efetuar a multiplicação de duas matrizes densas de modo sequencial,
7.  * para Estudo de Caso em Programação Paralela de Alto Desempenho.
8.  * Created on 29 de Março de 2013, 20:56
9.  */
10. #include <iostream>
11. ...
12. // Declaração das Constantes...
13. const int INTEGER = 0;
14. const int REAL    = 1;
15.
16. // Declara a estrutura para matriz...
17. struct TMatrix {
18.     int    lin;      // Linhas na matriz.
19.     int    col;      // Colunas na matriz.
20.     float *mat;      // Pseudo-Matriz linear.
21. };
22. // Programa principal
23. int main() {
24.     // Texto de tela
25.     ...
26.     // Inicia o gerador randômico...
27.     srand(time(NULL)); // Semente para iniciar o gerador randômico.
28.     // Declaração das matrizes
29.     TMatrix *A, *B, *C;
30.     ...
31.     // Alocação de memória para as três estruturas das matrizes.
32.     A = new TMatrix[sizeof(TMatrix)];
33.     ...
34.     A->lin = A->col = B->lin = B->col = C->lin = C->col = 6000; // Matrizes: 6000 x 6000
35.     // Alocação de memória para as três as matrizes propriamente ditas.
36.     ...
37.     int tipoDado = REAL;
38.     seqRandomMatrixGen(-30.0, +30.0, tipoDado, A); // Cria a matriz no intervalo [-30, 30].
39.     seqRandomMatrixGen(-30.0, +30.0, tipoDado, B); // Cria a matriz no intervalo [-30, 30].
40.     seqMatrixMul(A, B, C);
41.     // Salva a matriz P em formato 2D no disco.
42.     ...
43.     // Salva o relatório em disco...
44.     ...
45.     return 0;
46. }

```

47. ...

A listagem 11 mostra o equivalente em C++ com OpenMP do programa paralelo codificado em Ψ . O código completo encontra-se no **Anexo D**. As diferenças entre os programas *seqMatMul.cpp*, vide listagem 10, e *parMatMul.cpp* estão destacadas através de fonte bold na listagem 11.

Listagem 11: Modelo paralelo de programa em C++ para Multiplicação de matrizes de ordem n.
--

```

1. /*
2.  * File:    parMatMul.cpp
3.  * Author:  Marlim P. Menezes
4.  * Progama: Doutorado em Ciências - Engenharia Elétrica
5.  * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
6.  * Objetivo: Efetuar a multiplicação de duas matrizes densas de modo sequencial,
7.  * para Estudo de Caso em Programação Paralela de Alto Desempenho.
8.  * Created on 29 de Março de 2013, 20:56
9.  */
10.
11. #include <iostream>
12. ...
13. #include <omp.h>
14. ...
15. using namespace std;
16. // Declaração das Constantes...
17. const int INTEGER = 0;
18. const int REAL    = 1;
19. // Declara a estrutura para matriz...
20. struct TMatrix {
21.     int    lin;      // Linhas na matriz.
22.     int    col;      // Colunas na matriz.
23.     float *mat;      // Pseudo-Matriz linear.
24. };
25. // Programa principal
26. int main() {
27.     // Texto de tela
28.     ...
29.     // Inicia o gerador randômico...
30.     srand(time(NULL)); // Semente para iniciar o gerador randômico.
31.     // Declaração das matrizes
32.     TMatrix *A, *B, *C;
33.     ...
34.     // Alocação de memória para as três estruturas das matrizes.
35.     A = new TMatrix[sizeof(TMatrix)];
36.     ...
37.     A->lin = A->col = B->lin = B->col = C->lin = C->col = 6000; // Matrizes: 6000 x 6000
38.     // Alocação de memória para as três as matrizes propriamente ditas.
39.     ...
40.     int tipoDado = REAL;
41.     #pragma omp parallel sections
42.     {
43.         #pragma omp section // tarefa 1: inicializa a matriz A
44.         {
45.             parRandomMatrixGenOMP_A(-30.0, +30.0, tipoDado, A, 2); // Usa 2 CPUs
46.         }
47.         #pragma omp section // tarefa 2: inicializa a matriz A
48.         {
49.             parRandomMatrixGenOMP_B(-30.0, +30.0, tipoDado, B, 2); // Usa 2 CPUs
50.         }
51.     }
52.     parMatrixMulOMP(A, B, C, 4); // Usa 4 CPUs
53.     ...
54.     // Salva o relatório em disco...
55.     ...
56.     return 0;
57. }
58. ...

```

9.1.6 Resultados

A tabela 9.2 mostra um resumo dos tempos de processamento para cada um dos métodos sequencial e paralelo, e o *speedup* para os modelos sequencial e paralelo, considerando-se duas baterias de testes, sendo uma executando o modelo paralelo com 4 processadores e a outra com 8 processadores – neste último caso com recurso Hyper-threading ligado.

Tabela 9.2: Comparação dos tempos de processamento entre os modelos implementados.

Ordem da Matriz	Sequencial (segundos)	Paralelo 4 CPUs (segundos)	Paralelo 8 CPUs (segundos)	Speedup (ts/tp) 4 CPUs	Speedup (ts/tp) 8 CPUs
300	0,134128	0,046287	0,046287	2,897747	2,897747
400	0,339322	0,132438	0,125003	2,562120	2,714511
500	0,678644	0,231336	0,200014	2,933586	3,392982
1000	8,089683	2,76582	2,01577	2,924877	4,013197
2000	85,097809	28,3302	20,8609	3,003784	4,079297
3000	311,483787	105,539	74,646	2,951362	4,172813
4000	828,537782	274,723	183,982	3,015902	4,503363
5000	6385,313630	544,555	378,105	11,725746	16,887673
6000	7699,775845	946,084	693,269	8,138575	11,106476

A Figura 40 mostra a comparação entre os tempos de execução da tabela 9.2, de forma linear.

Figura 40 – Análise linear da evolução do tempo de execução dos algoritmos de multiplicação de matrizes densas, Sequencial e Paralelo executado com 4 e 8 CPUs (com Hyper-threading ligado).

A Figura 41 apresenta o gráfico comparativo para análise logarítmica dos mesmos dados do gráfico ilustrado na Figura 40.

Figura 41 – Análise logarítmica da evolução do tempo de execução dos algoritmos de multiplicação de matrizes densas Sequencial e Paralelo executado com 4 e 8 CPUs (com Hyper-threading ligado).

Nas Figuras 40 e 41, Paralelo 4p e Paralelo 8p significam que o mesmo algoritmo paralelo foi executado por 4 e 8 CPUs, respectivamente, com matrizes de mesmas dimensões, porém contendo dados numéricos diferentes.

A Figura 42 apresenta o gráfico do ganho de velocidade (speedup) do algoritmo sequencial em relação ao paralelo sendo executado em 4 e 8 processadores, para os mesmos testes que resultaram nos gráficos das Figuras 40 e 41, sobre os dados da tabela 9.2.

Como houve diferenças significativas nos ganhos de desempenho do algoritmo paralelo em relação ao sequencial, para as diversas dimensões das matrizes envolvidas nos testes, sendo executado em 4 e 8 processadores, não se chegou a determinar um fator speedup aproximado para todas as situações, então se decidiu apresentar os fatores obtidos, conforme apresentados na tabela 9.2, em um único gráfico de barras linear, visando uma análise mais precisa do que está ocorrendo.

Figura 42 – Análise de ganho de velocidade (speedup) dos algoritmos de multiplicação de matrizes Sequencial e Paralelo executado com 4 e 8 CPUs (com Hyper-threading ligado).

Nota-se, claramente, que o fator de ganho de desempenho obtido na execução do algoritmo paralelo, usando 4 processadores, para todas as dimensões de matrizes testadas é superior a 2. Entretanto, existem os casos das matrizes de ordem 5000 e 6000, onde o fator speedup é extremamente alto em relação ao esperado, sendo considerados casos especiais a serem explorados em um trabalho futuro.

Ainda, analisando a Figura 42 nota-se que o fator speedup, considerando modelo paralelo sendo executado com 8 processadores, é maior ou igual a 4 para matrizes de ordem $n \geq 1000$. Todavia, em nenhuma situação o ganho de velocidade do algoritmo paralelo sendo executado por 8 processadores, dobrou em relação ao ganho de velocidade obtido para 4 processadores. Isto ocorre devido à ativação do recurso Hyper-threading do chip multiprocessador utilizado. Pois, quando esse recurso está ativo o desempenho de paralelismo não dobra como esperado, por razões óbvias de projeto da microarquitetura do hardware.

Todos os algoritmos escritos em linguagem C++, testados anteriormente – sequencial e paralelo, seguem à risca as abordagens predefinidas nas ESPGs obtidas para os mesmos e, que são apresentadas nas expressões express.(34),

express.(35) e express.(36) para os modelos de programação sequencial, paralelo e pipeline da linguagem Ψ , respectivamente.

Um fator importante a ser considerado no aumento de velocidade é a forma como as matrizes foram dispostas na memória. Na realidade, a maneira como os dados são dispostos na memória faz diferença tanto no ganho de velocidade, quanto no tamanho da estrutura requisitante de endereços contíguos. Assim, todas as matrizes tiveram seus dados dispostos linearmente, de modo a ocuparem sempre posições contíguas de memória, garantindo, dessa forma o seu armazenamento completo na memória disponível, bem como o ganho de velocidade.

9.2 Estudo de Caso 2: Curto Circuito em Sistemas de Potência

Neste estudo de caso deve-se demonstrar como aplicar as três etapas da metodologia de paralelização otimizada desenvolvidas neste trabalho.

A ocorrência de curto circuito é um problema notório no cotidiano dos engenheiros eletricitas da área de sistemas de potência, e ocorre frequentemente nas linhas de transmissão devido principalmente a ocorrência de descargas atmosféricas. Em muitos casos também o curto circuito ocorre em subestações devido à falha em equipamentos por perda de isolamento, como por exemplo em transformadores.

O sistema elétrico a ser analisado, otimizado e, posteriormente, implementado com uma linguagem de programação é um subconjunto de uma rede elétrica maior obtida da base de dados da ONS (Operador Nacional do Sistema Elétrico), para o qual serão calculadas as correntes de curto circuito em diversas barras. Em seguida, ainda como parte deste estudo de caso, os programas implementados em linguagem C++, resultantes da otimização serão testados com um sistema elétrico contendo 5547 barras, a fim de verificar sua eficiência para as três técnicas de programação, isto é, sequencial, paralela e pipeline.

O sistema elétrico é fornecido na forma de arquivo de dados, formatado segundo um critério bem definido, pois a base de dados é feita para um programa em Fortran, o qual contém todos os dados relevantes aos cálculos de curto circuito que deverão ser efetuados. Também é utilizado um arquivo de dados contendo a lista de barras para as quais será calculado o curto circuito.

A listagem 12, a seguir, mostra a estrutura parcial do arquivo do sistema elétrico reduzido a ser analisado, apresentando apenas os dados necessários aos cálculos.

Listagem 12: Estrutura parcial do arquivo que constitui o sistema elétrico em análise.

1.	DBAR				
2.	(NB	CEM	BN	VBAS	DISJUN
3.	(----	====	-----	----	-----
4.	10839	MORRINHOS	69	69	
5.	10840	ROCHEDO	69	69	
6.	10844	VICENTIN	69	69	
7.	10846	GOIASA	69	69	
8.	11107	PLANALTO	69	69	12.5
9.	11150	PARANAIB	69	69	20

10.	9602	GOIATUBA	69	69
11.	9900	CEM_UTE---	69	69
12.	9901	CEM_UTE---	13	13.8
13.	99999			
14.	DCIR			
15.	(BF	CE	BT	NCT R1 X1 R0 X0 CN
16.	(-----	=====	-----	=====
17.	10839	10840	1	4912 4908 7351 16123CEL
18.	10844	11107	1	3365 6361 6691 24082CEL
19.	10846	10844	1	2556 4196 5222 15838CEL
20.	11107	10839	1	7.171 1333 1659 4977CEL
21.	9602	11150	1L	2252 4186 5210 15803CEL
22.	9602	9900	1	127624.19525.41691.521
23.	11107	9900	1	6.380212.09812.708 4576
24.	9900	9901	1T	333399999999999999
25.	0	9901	51T	999999999999999630120 3333
26.	9901	0	1G	16667630120 4667
27.	11150	0	1	.36108 1295.065953.2006EQUIV.
28.	11107	0	1	1.122439.2620.144213.158EQUIV.
29.	10846	0	1	16446 16000EQUIV.
30.	10840	0	1	61620 14820EQUIV.
31.	99999			

Na listagem 12, a seção **DBAR** informa os códigos das barras constituintes do sistema elétrico em estudo. A seção **DCIR**, do mesmo arquivo, contém as ligações entre as barras do sistema elétrico que formam o circuito da rede a ser analisado.

A listagem 13, a seguir, mostra o arquivo das barras do sistema elétrico onde deverão ser calculados os curtos circuitos.

Listagem 13: Estrutura do arquivo que constitui a lista de barras onde serão calculados os curtos circuitos trifásicos e fase-terra.

1. 10840
2. 10846
3. 9602
4. 9900
5. 9901

Ao final do processamento os resultados calculados deverão ser armazenados em um arquivo texto com o formato visto na listagem 14.

Listagem 14: Estrutura do arquivo contendo os resultados dos cálculos de curto circuito nas barras em que ocorreram as faltas, no sistema elétrico.

1.	num	barra	Vbase	Icc3f (kA)	fase	Iccft (kA)	fase
2.	----	-----	-----	-----	-----	-----	-----
3.	10840	ROCHEDO 69	69.0	0.915	-60.129	0.923	-67.360
4.	10846	GOIASA 69	69.0	1.060	-76.372	0.915	-80.314
5.	9602	GOIATUBA 69	69.0	2.719	-70.270	1.763	-72.344
6.	9900	CEM_UTE---69	69.0	3.188	-76.868	2.481	-76.734
7.	9901	CEM_UTE---13	13.8	9.152	-84.802	0.398	-2.105

Após análise dos dados contidos nos arquivos mostrados nas listagens (12, 13 e 14) acima, chegou-se a uma configuração para a estrutura do sistema elétrico descrito pelos mesmos.

A tabela 9.3 mostra os nomes das barras, suas ligações e as respectivas impedâncias de sequência zero (0: $Z^{(0)}$) e positiva (1: $Z^{(1)}$).

Tabela 9.3: Barras e Ligações com as respectivas impedâncias de sequência zero (0) e positiva (1), presentes no SE em estudo.

Barra (de)	Barra (para)	Impedância sequência (0)	Impedância sequência (1)
[1] 10839	[2] 10840	$Z^{(0)}_{12} = 73,51 + j161,23$	$Z^{(1)}_{12} = 49,12 + j49,08$
[3] 10844	[5] 11107	$Z^{(0)}_{35} = 66,91 + j240,82$	$Z^{(1)}_{35} = 33,65 + j63,61$
[4] 10846	[3] 10844	$Z^{(0)}_{43} = 52,22 + j158,38$	$Z^{(1)}_{43} = 25,56 + j41,96$
[5] 11107	[1] 10839	$Z^{(0)}_{51} = 16,59 + j49,77$	$Z^{(1)}_{51} = 0,07171 + j13,33$
[7] 9602	[6] 11150	$Z^{(0)}_{76} = 52,10 + j158,03$	$Z^{(1)}_{76} = 22,52 + j41,86$
[7] 9602	[8] 9900	$Z^{(0)}_{78} = 0,25416 + j0,91521$	$Z^{(1)}_{78} = 12,76 + j0,24195$
[5] 11107	[8] 9900	$Z^{(0)}_{58} = 0,12708 + j45,76$	$Z^{(1)}_{58} = 0,063802 + j0,12098$
[8] 9900	[9] 9901	$Z^{(0)}_{89} = 999,99 + j999,99$	$Z^{(1)}_{89} = j33,33$
[0] 0	[9] 9901	$Z^{(0)}_{09} = 6301,20 + j33,33$	$Z^{(1)}_{09} = 999,99 + j999,99$
[9] 9901	[0] 0	$Z^{(0)}_{90} = 6301,20 + j46,67$	$Z^{(1)}_{90} = j166,67$
[6] 11150	[0] 0	$Z^{(0)}_{60} = 0,0006595 + j0,032006$	$Z^{(1)}_{60} = 0,0036108 + j12,95$
[5] 11107	[0] 0	$Z^{(0)}_{50} = 0,001442 + j0,13158$	$Z^{(1)}_{50} = 0,011224 + j0,392620$
[4] 10846	[0] 0	$Z^{(0)}_{40} = j160,00$	$Z^{(1)}_{40} = j164,46$
[2] 10840	[0] 0	$Z^{(0)}_{20} = j148,20$	$Z^{(1)}_{20} = j616,20$

Na tabela 9.3, o número entre colchetes à esquerda do número da barra é o índice da mesma na lista de barras, lista essa obtida do arquivo do Sistema Elétrico. Os dados desta tabela juntamente com os dados das barras são necessários para a montagem das matrizes de admitâncias.

9.2.1 Solução Analítica

De modo geral, o modelo matemático para o tipo de problema abordado no presente estudo de caso costuma ser muito complexo, mas passível de paralelização.

Nas linhas que se seguem, serão descritos alguns passos para se calcular o curto circuito em sistemas de potência, de forma a minimizar a complexidade desta tarefa, assim, prepara-se o terreno para a extração do paralelismo que o mesmo tenha inerentemente.

É notório da comunidade acadêmica, na área de computação de alto desempenho, que cálculos matemáticos envolvendo matrizes possuem paralelismo natural e, que esse paralelismo pode ser obtido de forma relativamente fácil. Entretanto, esta tarefa será realizada no item 9.2.2 Extração de Paralelismo.

O modelo matemático para o cálculo de curto circuito em linhas de transmissão envolve um conjunto com uma quantidade razoável de equações, que devem ser

resolvidas em uma sequência predeterminada. Assim, na solução algébrica apresentada, esse conjunto de equações aparece na mesma sequência em que devem ser resolvidas, mas nenhuma solução numérica será mostrada, já que o caso em estudo apresenta uma grande base de dados envolvida na solução do mesmo, tornando humanamente indesejável ser resolvido com lápis e papel.

Um resumo da teoria de curto circuito em Sistemas de Potência é apresentado no **Apêndice C** deste texto, o qual poderá ajudar no entendimento do fenômeno, bem como da sua elevada complexidade matemática devido ao tratamento matricial do problema.

O roteiro a seguir descreve a solução analítica do problema de curto circuito em sistemas de potência apresentado neste Estudo de Caso. Para isto serão aplicados os procedimentos (1) a (4) da Metodologia desenvolvida ao longo desta Pesquisa, encontrada no **Capítulo 5** sob o título **Extraíndo Paralelismo**. Nos itens seguintes serão aplicados os demais procedimentos dessa Metodologia, de acordo com a necessidade:

1. Coletar os dados sobre as barras do Sistema Elétrico (SE). Encontram-se no arquivo fornecido (.ana), conforme mostrado na listagem 12.
2. Coletar os dados sobre os circuitos (ou as ligações) existentes no SE. Encontram-se no arquivo fornecido (.ana), vide listagem 12. Os valores fornecidos estão em % (por cento) e devem ser convertidos para pu, dividindo-os por 100 (considera-se: $S_{base} = 100 \text{ MVA}$). Vide tabela 9.3.
3. Obter a lista das barras a terem o curto calculado. Encontram-se no arquivo fornecido (.bar), conforme mostrado na listagem 13.
4. Montar a matriz de Admitâncias Nodais: A matriz de admitâncias, eq.(37), de um sistema elétrico com n -barras será quadrada e de ordem n . Assim, a matriz para o SE do estudo de caso presente terá ordem $n = 9$, pois o mesmo possui 9 barras. Vide listagem 12.

$$Y^{(0)(1)} = \begin{bmatrix} Y_{11}^{(0)(1)} & \dots & Y_{1j}^{(0)(1)} & \dots & Y_{1n}^{(0)(1)} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ Y_{i1}^{(0)(1)} & \vdots & Y_{ij}^{(0)(1)} & \vdots & Y_{in}^{(0)(1)} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ Y_{n1}^{(0)(1)} & \dots & Y_{nj}^{(0)(1)} & \dots & Y_{nn}^{(0)(1)} \end{bmatrix} = \begin{bmatrix} Z_{11}^{(0)(1)} & \dots & Z_{1j}^{(0)(1)} & \dots & Z_{1n}^{(0)(1)} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ Z_{i1}^{(0)(1)} & \vdots & Z_{ij}^{(0)(1)} & \vdots & Z_{in}^{(0)(1)} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ Z_{n1}^{(0)(1)} & \dots & Z_{nj}^{(0)(1)} & \dots & Z_{nn}^{(0)(1)} \end{bmatrix}^{-1} \quad (37)$$

Os símbolos (0) e (1) representam a sequência zero e positiva, respectivamente. Assim, ter-se-á duas matrizes de admitâncias para este SE, isto é, uma para a sequência zero e outra para a positiva, de acordo com a tabela 9.3 de impedâncias. A forma geral para a montagem da matriz de admitâncias é dada nos passos seguintes:

- a. Serão usadas letras maiúsculas (Y_{ij}) para descrever os termos da matriz de admitâncias, e letras minúsculas para descrever as admitâncias da linha ($y_{ij} = 1 / z_{ij}$).
- b. Os termos Y_{ii} são os somatórios de todas as admitâncias da linha, conectadas na i -ésima barra, por exemplo, a barra (1) da primeira linha da tabela 9.3 é dada pela eq.(38), para a sequência zero e positiva, respectivamente:

$$Y_{11}^{(0)(1)} = y_{12}^{(0)(1)} + y_{15}^{(0)(1)} = \frac{1}{z_{12}^{(0)(1)}} + \frac{1}{z_{15}^{(0)(1)}}. \quad (38)$$

- c. Os termos Y_{ij} fora da diagonal principal são negativos, entre a i -ésima e a j -ésima barras da linha de admitâncias, eq.(39), por exemplo da barra (1) para a barra (2), em ambas as sequências, zero e positiva:

$$Y_{12}^{(0)(1)} = -y_{12}^{(0)(1)} = -\frac{1}{z_{12}^{(0)(1)}}. \quad (39)$$

- d. Barras não conectadas têm admitância zero (= impedância infinita) entre elas – por exemplo, da barra (1) para a barra (3), eq.(40):

$$Y_{13}^{(0)(1)} = 0. \quad (40)$$

e. A matriz de admitâncias é simétrica, eq.(41):

$$Y_{ji}^{(0)(1)} = Y_{ij}^{(0)(1)}, \quad (41)$$

por exemplo, da barra (1) para a barra (2), eq.(42):

$$Y_{21}^{(0)(1)} = Y_{12}^{(0)(1)} = -\frac{1}{Z_{12}^{(0)(1)}}. \quad (42)$$

f. Por ser grande, 9 x 9, a matriz de admitâncias para o SE mostrado na tabela 9.3 será apresentada de forma bem simplificada, vide eq.(43).

$$Y^{(0)(1)} = \begin{bmatrix} Y_{12}^{(0)(1)} + Y_{15}^{(0)(1)} & -Y_{12}^{(0)(1)} & \cdots & 0 & -Y_{15}^{(0)(1)} & \cdots \\ -Y_{12}^{(0)(1)} & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & Y_{89}^{(0)(1)} \end{bmatrix} \quad (43)$$

$$\text{onde, } y_{12}^{(0)(1)} = \frac{1}{Z_{12}^{(0)(1)}}, y_{15}^{(0)(1)} = \frac{1}{Z_{15}^{(0)(1)}} \text{ e } y_{89}^{(0)(1)} = \frac{1}{Z_{89}^{(0)(1)}}.$$

5. Triangularizar as matrizes de admitâncias para as sequências zero e positiva, expandindo-as com a matriz identidade de mesma ordem à sua direita, eq.(44) e eq.(45):

$$Y_{\text{triang}}^{(0)(1)} = \left[\left[Y^{(0)(1)} \right] \left[I \right] \right] \quad (44)$$

$$Y_{\text{triang}}^{(0)(1)} = \begin{bmatrix} Y_{12}^{(0)(1)} + Y_{15}^{(0)(1)} & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & y_{89}^{(0)(1)} & 0 & \cdots & 1 \end{bmatrix}. \quad (45)$$

6. Calcular as matrizes de impedâncias a partir da inversão das matrizes de admitâncias, para as sequências zero e positiva. Esta inversão deve ser feita somente para as colunas necessárias, correspondentes às barras em curto. Para isto, será aplicado o método Back Substitution sobre as matrizes de

admitâncias do SE, $Y_{triang}^{(0)(1)}$. Esse método pode ser encontrado em bons livros de Álgebra Linear, vide referências. Como resultados dessas operações, têm-se as matrizes $Z^{(0)}$ e $Z^{(1)}$ preparadas para os cálculos seguintes.

7. Calcular o curto trifásico e o curto fase-terra para cada barra solicitada. Estes cálculos envolvem diversas equações, que serão apresentadas na ordem de uso, nos próximos subitens:

a. Cálculo da corrente de base para cada barra i em curto, vide eq.(46):

$$I_{base(i)} = \frac{S_{base}}{\sqrt{3} * V_{base(i)}} = \frac{100}{\sqrt{3} * V_{base(i)}}. \quad (46)$$

Como citado anteriormente, S_{base} tem seu valor adotado igual a 100MVA para todo o SE, mas o valor de $V_{base(i)}$ é fornecido no arquivo mostrado na listagem 12, e cada barra possui o seu.

b. Cálculo da impedância equivalente de Thévenin, para as sequências zero e positiva para cada barra i em curto circuito, equação eq.(47):

$$Z_{th(i)}^{(0)(1)} = Z_{ii}^{(0)(1)}. \quad (47)$$

Assim, a impedância equivalente de Thévenin para a sequência zero é calculada com a eq.(48):

$$Z_{th(i)}^{(0)} = Z_{ii}^{(0)} \quad (48)$$

e, para a sequência positiva é calculada com a eq.(49):

$$Z_{th(i)}^{(1)} = Z_{ii}^{(1)}. \quad (49)$$

Os valores das matrizes estão em pu.

c. A corrente de curto trifásico, conforme I_{base} , na barra i é dada pela eq.(50):

$$I_{cc3f(i)} = \frac{1}{Z_{th(i)}^{(1)}} I_{base(i)} . \quad (50)$$

d. Cálculo da corrente de curto fase-terra na barra i , eq.(51):

$$I_{ccft(i)} = \frac{3}{2Z_{th(i)}^{(1)} + Z_{th(i)}^{(0)}} I_{base(i)} . \quad (51)$$

As correntes de curto são dadas em A (Ampères) ou kA.

Dessa forma, conclui-se o modelo matemático, já simplificado, para o problema de cálculo de curto circuito em linhas de transmissão, conforme solicitado neste Estudo de Caso. O problema proposto considera apenas os cálculos de curto trifásico e curto fase terra para as sequências zero (0) e positiva (1), nas barras do Sistema Elétrico. Por estas razões seu modelo matemático ficou, relativamente, pouco complexo.

9.2.2 Extração de Paralelismo

A fim de se alcançar um paralelismo adequado para o problema proposto, os procedimentos (5) a (10) da metodologia de extração de paralelismo proposta no capítulo 5, serão plenamente explorados. Os procedimentos (1) a (4) foram aplicados no item 9.2.1 Solução Analítica.

Esta etapa será iniciada analisando-se o modelo matemático obtido no item anterior do ponto de vista computacional, onde se procura fragmentá-lo em três grupos computacionais, ou seja, sequencial, paralelo e, eventualmente, um híbrido, conforme recomendado no procedimento (6) da metodologia citada.

É fácil perceber-se a elevada heterogeneidade do modelo matemático, pois está visível o elevado grau de dependência de algumas de suas tarefas, e em outros casos a independência, neste último caso podendo ser executadas simultaneamente. Assim, de acordo com as instruções apresentadas nos procedimentos (6) a (9) da metodologia Extraíndo Paralelismo, construímos a tabela

9.4 contendo três colunas rotuladas de Sequencial, Paralelo e Híbrido, nas quais serão classificados os diversos blocos de processamento. Esse processo é recursivo, e deve ser interrompido quando atingido o que pode ser considerado um grau de eficiência desejado para a solução computacional.

Na tabela 9.4, a coluna **I** mostra as iterações do processo recursivo que existe naturalmente na extração de paralelismo, conforme mostrado na Figura 23, vide capítulo 5. Lembrando que este processo recursivo deve continuar até não se conseguir mais distinguir blocos Paralelos e Sequenciais de Híbridos.

Tabela 9.4: Modelos de programação: Subgrupos computacionais do modelo matemático obtido do estudo de caso para cálculo de curto circuito em sistemas de potência.

I	Paralelo	Sequencial	Híbrido
1	-	T1. Obtenção do número de barras no SE. T2. Obtenção do número de ligações no SE. T3. Obtenção do número de barras em curto no SE. T4. Carga da lista de barras do SE. T5. Carga da lista de ligações do SE. T6. Obtenção da matriz de admitâncias yBus0. T7. Obtenção da matriz de admitâncias yBus1. T8. Carga da lista de barras em curto. T9. Gravar yBus0 em disco. T10. Gravar yBus1 em disco.	O restante do Modelo.
2	T11. Criar matriz identidade iBus0. T12. Criar matriz identidade iBus1. T13. Triangularizar yBus0. T14. Triangularizar yBus1. T15. Obter zBus0 com inversão de yBus0 e iBus0. T16. Obter zBus1 com inversão de yBus1 e iBus1.	-	O restante do Modelo.
3	-	T17. Gravar zBus0 em disco. T18. Gravar zBus1 em disco. T19. Cálculo da corrente e fase de curto trifásico. T20. Cálculo da corrente e fase de curto de fase-terra. T21. Gravar os resultados em disco.	-

Ainda, com relação à tabela 9.4, nota-se a enumeração dos blocos de processamento que julgamos necessários à solução completa do problema proposto. Todavia, estes blocos serão tratados como tarefas, que deverão ser executadas atômicamente, mas agrupadas em pequenos blocos que poderão ser executados sequencialmente, ou em paralelo.

A tabela 9.4, também dispõe as tarefas em uma ordem tal que possam ser executadas sequencialmente, sem a necessidade de qualquer rearranjo.

A tabela 9.5 mostra o reagrupamento das tarefas enumeradas na tabela 9.4, de forma a se obter o paralelismo no sistema como um todo. Esta tabela será usada para auxiliar na edição da ESPG paralela.

Tabela 9.5: Modelos de programação: Rearranjo das tarefas de forma a obter-se o paralelismo e, portanto, aumento na velocidade de execução do sistema computacional.

Região	Modelo	Tarefa	Tarefa Executada	CPU
Paralela	Sequencial	- T1	1.1) Obtenção do número de barras no SE.	1
	Sequencial	- T2	1.2) Obtenção do número de ligações no SE.	1
	Sequencial	- T3	1.3) Obtenção do número de barras em curto no SE.	1
Sequencial			Intervalo para sincronização.	
Paralela	Sequencial	- T4	2.1) Carga da lista de barras do SE.	1
	Sequencial	- T5	2.2) Carga da lista de ligações do SE.	1
	Sequencial	- T8	2.3) Carga da lista de barras em curto.	1
Sequencial			Intervalo para sincronização.	
Paralela	Paralelo	- T11	3.1) Criar matriz identidade iBus0.	2
	Paralelo	- T12	3.2) Criar matriz identidade iBus1.	2
Sequencial			Intervalo para sincronização.	
Paralela	Paralelo	- T6	4.1) Obtenção da matriz de admitâncias yBus0.	2
	Paralelo	- T7	4.2) Obtenção da matriz de admitâncias yBus1.	2
Sequencial			Intervalo para sincronização.	
Paralela	Sequencial	- T9	5.1) Gravar yBus0 em disco.	1
	Sequencial	- T10	5.2) Gravar yBus1 em disco.	1
Sequencial			Intervalo para sincronização.	
	Paralelo	- T13	6.1) Triangularizar yBus0.	4
	Paralelo	- T14	7.1) Triangularizar yBus1.	4
Paralela			Intervalo para sincronização.	
	Paralelo	- T15	8.1) Obter zBus0 com inversão de yBus0 e iBus0.	2
Sequencial	Paralelo	- T16	8.2) Obter zBus1 com inversão de yBus1 e iBus1.	2
			Intervalo para sincronização.	
Sequencial	Paralelo	- T19	9.1) Cálculo das correntes e fases de curto trifásico e curto fase-terra.	4
Paralela			Intervalo para sincronização.	
	Sequencial	- T20	10.1) Gravar os resultados em disco.	1
	Sequencial	- T17	10.2) Gravar zBus0 em disco.	1
	Sequencial	- T18	10.3) Gravar zBus1 em disco.	1

O reagrupamento mostrado na tabela 9.5 não diminui a quantidade de processamento sequencial atômicamente, entretanto é possível se executar mais de uma tarefa ao mesmo tempo, obtendo-se, assim, algum grau de paralelismo. Esta atividade é um pré-requisito para se alcançar o procedimento (10) da metodologia

Extraindo Paralelismo, a partir do qual o caminho para se chegar ao pipeline estará aberto.

Com este rearranjo acrescentou-se a coluna **CPU** à tabela 9.5, cuja função é permitir uma antecipação da distribuição de recursos do hardware para a execução das tarefas do sistema computacional resultante. Esta coluna informa, previamente, o número de processadores alocados para cada tarefa. Entretanto, esses valores poderão sofrer alterações, de acordo com as necessidades do programador ou dos recursos de hardware disponíveis.

Como se vê, a tabela 9.5 é auto explicativa, porém há alguns pontos importantes a serem observados. A coluna **Região** descreve o tipo de cada região de processamento das tarefas a serem executadas, na análise corrente é possível ter somente dois tipos de regiões, que são sequencial ou paralela. A coluna **Modelo Tarefa** é importante para a visualização do modelo computacional interno de cada tarefa envolvida no processo, seguindo-se assim a enumeração original constante da tabela 9.4, e também ter-se uma visão geral da junção de tarefas, que antes eram atômicas, em uma única tarefa, como ocorre com as tarefas T19 e T20. As tarefas T19 e T20 foram juntadas em um único bloco de processamento, pois ambas são independentes entre si, mas dependentes do mesmo conjunto de dados buscados externamente. Mantê-las separadas seria desperdício de tempo e de recursos de hardware.

Na coluna **Tarefa Executada** vê-se que as tarefas foram renumeradas. Vê-se, também, através desta coluna que as tarefas foram reagrupadas em algumas regiões paralelas e outras sequenciais.

A construção de ambas as tabelas 9.4 e 9.5 é muito útil para se manter uma estruturação mais adequada e documentada da distribuição e atribuição das tarefas às várias regiões de processamento – regiões essas inerentes à programação paralela.

Após análise cuidadosa do modelo matemático obtido a partir do enunciado do problema proposto e da tabela 9.5, fica claro que serão necessárias diversas expressões semânticas de processamento geral, tanto para o modelo de programação sequencial quanto para o paralelo, entretanto, à primeira vista, ainda não está clara a existência de ESPGs para o modelo pipeline.

Porém, com uma análise mais cuidadosa dos dados da tabela 9.5, percebe-se a possibilidade de remanejamento na ordem de execução de algumas tarefas, de modo a se conseguir maior ganho de desempenho paralelo, bem como o pipeline buscado, visando aumentar ainda mais o ganho de velocidade de execução do programa como um todo.

A tabela 9.6 mostra a tabela 9.5 após o rearranjo da sequência de execução das tarefas, de modo que os resultados não sejam comprometidos e se mantenha compatível com a estrutura do hardware disponível. Esta tabela apresenta a organização das tarefas de acordo com as necessidades do modelo pipeline.

Tabela 9.6: Modelos de programação: Rearranjo das tarefas da tabela 9.5, a fim de obter pipeline e, ainda um pouco mais de ganho na velocidade de execução.

Região	Modelo	Tarefa	Tarefa executada	CPU
Paralela	Sequencial - T1	1.1)	Obtenção do número de barras no SE.	1
	Sequencial - T2	1.2)	Obtenção do número de ligações no SE.	1
	Sequencial - T3	1.3)	Obtenção do número de barras em curto no SE.	1
Sequencial			Intervalo para sincronização.	
Paralela	Sequencial - T4	2.1)	Carga da lista de barras do SE.	1
	Sequencial - T5	2.2)	Carga da lista de ligações do SE.	1
	Sequencial - T8	2.3)	Carga da lista de barras em curto.	1
Sequencial			Intervalo para sincronização.	
Paralela	Paralelo - T11	3.1)	Criar matriz identidade iBus0.	2
	Paralelo - T12	3.2)	Criar matriz identidade iBus1.	2
Sequencial			Intervalo para sincronização.	
Pipeline	Sequencial1 - T6	4.1)	Obtenção da matriz de admitâncias yBus0.	1
	Sequencial1 - T9	4.2)	Gravar yBus0 em disco.	1
	Sequencial2 - T7	4.3)	Obtenção da matriz de admitâncias yBus1.	1
	Sequencial2 - T10	4.4)	Gravar yBus1 em disco.	1
Sequencial			Intervalo para sincronização.	
Pipeline	Pipeline - T13	5.1)	Triangularizar yBus0.	2
	Pipeline - T15	5.2)	Obter zBus0 com inversão de yBus0 e iBus0.	2
Sequencial			Intervalo para sincronização.	
Pipeline	Pipeline - T14	6.1)	Triangularizar yBus1.	2
	Pipeline - T16	6.2)	Obter zBus1 com inversão de yBus1 e iBus1.	2
Sequencial			Intervalo para sincronização.	
Sequencial	Paralelo - T19	7.1)	Cálculo das correntes e fases de curto trifásico e curto fase-terra.	4
			Intervalo para sincronização.	
Paralela	Sequencial - T20	8.1)	Gravar os resultados em disco.	1
	Sequencial - T17	8.2)	Gravar zBus0 em disco.	1
	Sequencial - T18	8.3)	Gravar zBus1 em disco.	1

Como se pode ver, após análise criteriosa e rearranjo das tarefas conseguiu-se identificar três regiões de pipeline, as regiões 4, 5 e 6 onde foram alocados quatro processadores para todas elas. A região 4 contém dois pipelines de dois estágios cada um e teve dois processadores alocados para a execução sequencial de suas tarefas, são as tarefas T6 com T9, onde T6 gera os dados para a tarefa T9 e sinaliza quando esta última deverá iniciar o processamento através da flag Sequencial1, enquanto que a tarefa T7 gera os dados para a tarefa T10, a qual é sinalizada por T7 que deve iniciar seu processamento através da flag Sequencial2. Os pipelines Sequencial1 e Sequencial2 são sequenciais internamente, mas executados em paralelo. As tarefas T6, T7, T9 e T10 podem ser estritamente sequenciais, pois T6 e T7 realizam pouco processamento sendo muito rápidas e, as tarefas T9 e T10, apesar de serem muito lentas, serão mantidas como sequenciais por realizarem acesso a disco durante toda sua atividade – nenhuma leitura/escrita em disco é feita de forma paralela.

Os quatro processadores destinados aos pipelines da região 4 – tarefas T6, T9, T7 e T10, foram distribuídos uniformemente, sendo alocado um processador para cada tarefa, como se vê na tabela 9.6. Pelo fato de as tarefas T13 e T14 realizarem muito processamento elas foram separadas em duas regiões pipeline, onde a tarefa T13 faz pipeline com a tarefa T15 na região 5 e T14 faz pipeline com a tarefa T16 na região pipeline 6. As quatro tarefas anteriores possuem processamento paralelo internamente, por esta razão foram alocados dois processadores para cada uma. Assim, os quatro processadores disponíveis no computador utilizado foram alocados para a região pipeline 4, em seguida todos foram alocados para a região pipeline 5 e, finalmente para a região pipeline 6, visando um aumento no ganho de velocidade da execução, através desses pipelines de dois estágios cada um.

Verificou-se na análise anterior que há outras possibilidades de pipeline na tabela 9.5, entretanto ficou decidido estabelecer-se apenas essas três regiões, como opção por menor complexidade na codificação da solução do sistema.

É importante lembrar que o grau a ser alcançado tanto para paralelismo, quanto para pipeline depende muito da relação entre os dados, grau de dependência entre

eles, da infraestrutura do hardware disponível, e também, das necessidades do modelador. Portanto, acredita-se que o grau de paralelismo e pipeline alcançados nesta solução sejam satisfatórios para os propósitos deste estudo.

9.2.3 Obtenção das ESPGs

Munidos das tabelas anteriores é possível se editar as ESPGs para o modelo computacional obtido, de forma relativamente simples. Para facilitar a edição das expressões semânticas, serão utilizados os rótulos T1 a T20 dados às tarefas na tabela 9.5, mas no momento da programação esses rótulos serão substituídos pelos nomes dos métodos que devem ser coerentes com o que cada um faz, de fato.

- a) ESPG sequencial: Para esta solução pode-se usar a sequências das tarefas dispostas em qualquer uma das tabelas 9.4, 9.5 ou 9.6, no entanto a sequência disposta na tabela 9.5 será utilizada, pois dessa forma o trabalho de edição das ESPGs paralelas e pipelines será mais simples:

$$seqSC1 = '(T1) \rightarrow '(T2) \rightarrow '(T3) \rightarrow '(T4) \rightarrow '(T5) \rightarrow '(T8) \rightarrow '(T11) \rightarrow '(T12) \quad (52)$$

$$seqSC2 = '(T6) \rightarrow '(T7) \rightarrow '(T9) \rightarrow '(T10) \rightarrow '(T13) \rightarrow '(T14) \rightarrow '(T15) \rightarrow '(T16) \quad (53)$$

$$seqSC = seqSC1 \rightarrow seqSC2 \rightarrow '(T19) \rightarrow '(T20) \rightarrow '(T17) \rightarrow '(T18) \quad (54)$$

- b) ESPG paralela: A disposição das tarefas, conforme mostrada na tabela 9.5 será usada na edição desta ESPG, assim, as expressões a seguir compõem a solução desejada:

$$parSC1 := ['(T1) \parallel '(T2) \parallel '(T3)] \rightarrow ['(T4) \parallel '(T5) \parallel '(T8)] \quad (55)$$

$$parSC2 := ["(T11) \parallel "(T12)] \rightarrow ["(T6) \parallel "(T7)] \quad (56)$$

$$parSC3 := ['(T9) \parallel '(T10)] \rightarrow "(T13).4 \rightarrow "(T14).4 \quad (57)$$

$$parSC4 := ["(T15) \parallel "(T16)] \rightarrow "(T19).4 \quad (58)$$

$$parSC5 := ['(T20) \parallel '(T17) \parallel '(T18)] \quad (59)$$

$$parSC := parSC1 \rightarrow parSC2 \rightarrow parSC3 \rightarrow parSC4 \rightarrow parSC5 \quad (60)$$

c) ESPG pipeline: Esta solução é obtida a partir da tabela 9.6:

$$pipeSC1 := ['(T1) \parallel '(T2) \parallel '(T3)] \rightarrow ['(T4) \parallel '(T5) \parallel '(T8)] \quad (61)$$

$$pipeSC2 := ["(T11) \parallel "(T12)] \quad (62)$$

$$pipeSC3 := ["(T6).1 >|" "(T9).1 \parallel "(T7).1 >|" "(T10).1] \quad (63)$$

$$pipeSC4 := ["(T13).2 >|" "(T15).2 \rightarrow "(T14).2 >|" "(T16).2] \quad (64)$$

$$pipeSC5 := "(T19).4 \rightarrow ['(T20) \parallel '(T17) \parallel '(T18)] \quad (65)$$

$$pipeSC := pipeSC1 \rightarrow pipeSC2 \rightarrow pipeSC3 \rightarrow pipeSC4 \rightarrow pipeSC5 \quad (66)$$

Cada expressão obtida nesta etapa será codificada em um programa fonte na linguagem Ψ , porém somente as versões Sequencial e Paralela serão codificadas em C++.

9.2.4 Codificação em Ψ

O quadro 9.1 a seguir, obtido a partir da tabela 9.5, lista todas as tarefas que serão executadas, com seus respectivos nomes.

Quadro 9.1: Lista de tarefas com suas funções e seus nomes, obtida da tabela 9.5.

T1. Obtenção do número de barras no SE.	obtemNumBarrasSE(arquivo);
T2. Obtenção do número de ligações no SE.	obtemNumLigacoesSE(arquivo);
T3. Obtenção do número de barras em curto no SE.	obtemNumBarrasCC(arquivo);
T4. Carga da lista de barras do SE.	carregaBarrasSE(arquivo);
T5. Carga da lista de ligações do SE.	carregaLigacoesSE(arquivo);
T6. Obtenção da matriz de admitâncias yBus0.	criaMatrizAdmitancias(yBus0);
T7. Obtenção da matriz de admitâncias yBus1.	criaMatrizAdmitancias(yBus1);
T8. Carga da lista de barras em curto.	carregaBarrasCC(arquivo);
T9. Gravar yBus0 em disco.	gravaMatriz(arquivo, yBus0);
T10. Gravar yBus1 em disco.	gravaMatriz(arquivo, yBus1);
T11. Criar matriz identidade iBus0.	criaMatrizIdentidade(iBus0);
T12. Criar matriz identidade iBus1.	criaMatrizIdentidade(iBus1);
T13. Triangularizar yBus0.	triagMatriz(yBus0, iBus0);
T14. Triangularizar yBus1.	triagMatriz(yBus1, iBus1);
T15. Obter zBus0 com inversão de yBus0 e iBus0.	calcBackSubs(yBus0, iBus0, zBus0);
T16. Obter zBus1 com inversão de yBus1 e iBus1.	calcBackSubs(yBus1, iBus1, zBus1);
T17. Gravar zBus0 em disco.	gravaMatriz(arquivo, zBus0);
T18. Gravar zBus1 em disco.	gravaMatriz(arquivo, zBus1);
T19. Cálculo do curto trifásico e fase-terra.	calcCC();
T20. Gravar os resultados em disco.	gravaResultados(arquivo);

A solução sequencial não fará uso das ESPGs express.(52) à express.(54). Isto, porque se pretende mostrar como escrever, explicitamente, um programa extenso na linguagem Ψ . A listagem 15 mostra esta solução. A sequência de execução das tarefas nesta listagem segue a sequência da tabela 9.5.

Listagem 15: Modelo sequencial de programa em Ψ para Curto Circuito em Sistemas de Potência.

```

1. //
2. // seqShortCircuit.ppsi
3. // Cálculo de curto circuito em LTs usando a linguagem Psi, SEQUENCIAL.
4. //
5. #uses
6. // Nenhuma biblioteca padrão será usada explicitamente.
7. ccSE.hpsi; // Mantém a classe SE (Sistema Elétrico) e seus métodos.
8. #start
9. display "Exemplo de Calculo de Curto em LTs, sequencialmente com Psi.", linefeed;
10. display "Estudo de Caso desenvolvido no Programa de Doutorado em Ciencias. ",
linefeed;
11. display "Escola Politecnica da Universidade de São Paulo - Brasil.", linefeed;
12. display "=====", linefeed;
13. #control
14. // Obtenção de dados iniciais do SE.
15. ccSE.arqSE = "EPE_2011.ana"; // Arquivo com 5547 barras, para estudo de caso...
16. ccSE.arqBarraCC = "EPE_2011_5000.bar"; // Arquivo contendo 5000 barras em curto
17. // Obtém o número de barras e de ligações do SE, para alocação dinâmica de
18. // memória para as estruturas...
19. ccSE.obtemNumBarrasSE(ccSE.arqSE); // T1
20. ccSE.obtemNumLigacoesSE(ccSE.arqSE); // T2
21. ccSE.obtemNumBarrasCC(ccSE.arqBarraCC); // T3
22. display "Dimensoes do Sistema Eletrico(SE)", linefeed;
23. display "-----", linefeed.2;
24. display "numBarras : ", ccSE.numBarras, linefeed;
25. display "numLigacoes : ", ccSE.numLigacoes, linefeed;
26. display "numBarrasCC : ", ccSE.numBarras, linefeed;

```

```

27. // Define as dimensões das matrizes com os valores obtidos do arquivo do SE...
28. ccSE.yBus0.lines = ccSE.yBus0.columns = ccSE.numBarras;
29. ccSE.yBus1.lines = ccSE.yBus1.columns = ccSE.numBarras;
30. ccSE.zBus0.lines = ccSE.zBus0.columns = ccSE.numBarras;
31. ccSE.zBus1.lines = ccSE.zBus1.columns = ccSE.numBarras;
32. ccSE.barras.size = ccSE.numBarras + 1;
33. ccSE.ligacoes.size = ccSE.numLigacoes;
34. ccSE.barrasCC.size = ccSE.numBarrasCC;
35. ccSE.endBar.size = ccSE.nummax + 1; // nummax é uma constante definida em SE.
36. ccSE.endBar[0] = 0;
37. // Carrega dados do SE para os cálculos de curto.
38. ccSE.carregaBarrasSE(ccSE.argSE); // T4 - Carrega as barras do SE.
39. ccSE.carregaLigacoesSE(ccSE.argSE); // T5 - Carrega as ligacoes do SE.
40. ccSE.carregaBarrasCC(ccSE.argBarraCC); // T8 - Carrega as barras em curto.
41. ccSE.criaMatrizIdentidade(ccSE.iBus0); // T11
42. ccSE.criaMatrizIdentidade(ccSE.iBus1); // T12
43.
44. // Prepara-se para calcular o Curto Circuito em todas as barras solicitadas.
45. // *****
46. // Triangulariza as matrizes yBus...
47. ccSE.iBus0.lines = ccSE.iBus0.columns = ccSE.numBarras;
48. ccSE.iBus1.lines = ccSE.iBus1.columns = ccSE.numBarras;
49. ccSE.criaMatrizAdmitancias(ccSE.yBus0); // T6
50. ccSE.criaMatrizAdmitancias(ccSE.yBus1); // T7
51. // Grava as matrizes yBus0 e yBus1 em disco no formato 2D e texto.
52. display "Gravando yBus0...", linefeed;
53. file "yBus0.txt", ccSE.yBus0, table, text, write; // T9 - Grava yBus0 em 2D e texto
54. display "Gravando yBus1...", linefeed;
55. file "yBus1.txt", ccSE.yBus1, table, text, write; // T10 - Idem para yBus1
56. ccSE.triangMatriz(ccSE.yBus0, ccSE.iBus0); // T13
57. ccSE.triangMatriz(ccSE.yBus1, ccSE.iBus1); // T14
58. ccSE.calcBackSubs(ccSE.yBus0, ccSE.iBus0, ccSE.zBus0); // T15
59. ccSE.calcBackSubs(ccSE.yBus1, ccSE.iBus1, ccSE.zBus1); // T16
60.
61. // Calcula o curto...
62. ccSE.calcCC(); // T19 - O resultados dos cálculos são mantidos na estrutura relatCC...
63.
64. #finish
65. display "Gravando o relatorio de curto em disco", linefeed;
66. display "-----", linefeed;
67. file "relatorioCC.txt", ccSE.relatCC, text, write; // T20
68. display linefeed, "Gravando as matrizes resultantes em disco", linefeed;
69. display "-----", linefeed;
70. // Grava as matrizes zBus0 e zBus1 em disco no formato 2D e texto.
71. display "Gravando zBus0...", linefeed;
72. file "zBus0.txt", ccSE.zBus0, table, text, write; // T17 - Grava zBus0 em 2D e texto
73. display "Gravando zBus1...", linefeed;
74. file "zBus1.txt", ccSE.zBus1, table, text, write; // T18 - Idem para zBus1
75.
76. display "Fim...";
77. #end.

```

Conforme se vê na listagem 15, sem o uso de ESPGs o resultado é um programa fonte muito grande tal como nas linguagens de programação tradicionais. Escrever um programa como foi feito neste exemplo não é uma boa prática de programação quando se está usando a linguagem Ψ , pois por ela ser implicitamente paralela, o compilador irá tentar paralelizá-lo através da análise de dependência entre os dados e as tarefas, resultando, dessa forma, em maior tempo de compilação e, talvez um alcance do desempenho pobre. Portanto, o ideal é utilizar as ESPGs no código, pois o compilador está preparado para traduzi-la, já que a relação entre as tarefas é passada explicitamente.

O programa Ψ para o modelo paralelo é mostrado na listagem 16. Note que há, basicamente, três diferenças em relação à versão sequencial, que são o acréscimo do comando *model* que especifica o modelo de memória a ser usado no escopo da *segp* e as *ESPGs* que devem representar a codificação em paralelo.

Esta solução fará uso das *ESPGs* *express.(55)* à *express.(60)*. Como as *ESPGs* obtidas para este exemplo são muito grandes, pode-se se beneficiar de um recurso disponível na linguagem Ψ , que é a capacidade de prover sinônimos (em inglês: *aliases*) para elementos de programação quando seus nomes forem muito extensos.

Listagem 16: Modelo paralelo de programa em Ψ para Curto Circuito em Sistemas de Potência.

```

1. //
2. // parShortCircuit.ppsi
3. // Cálculo de curto circuito em LTs usando a linguagem Psi, PARALELO.
4. //
5. #uses
6. // Nenhuma biblioteca padrão será usada explicitamente.
7. ccSE.hpsi; // Mantém a classe SE (Sistema Elétrico) e seus métodos.
8. #start
9. display "Exemplo de Calculo de Curto em LTs, paralelamente com Psi.", linefeed;
10. display "Estudo de Caso desenvolvido no Programa de Doutorado em Ciencias.", linefeed;
11. display "Escola Politecnica da Universidade de São Paulo - Brasil.", linefeed;
12. display "=====", linefeed;
13. #control
14. // Arquivos de dados do SE relacionados.
15. ccSE.arqSE = "EPE_2011.ana"; // Arquivo com 5547 barras, para estudo de caso...
16. ccSE.arqBarraCC = "EPE_2011_5000.bar"; // Arquivo contendo 5000 barras em curto
17.
18. // Apelida os métodos a serem executados para facilitar na edição das ESPGs.
19. T1 <- ccSE.obtemNumBarrasSE(ccSE.arqSE); // T1
20. T2 <- ccSE.obtemNumLigacoesSE(ccSE.arqSE); // T2
21. T3 <- ccSE.obtemNumBarrasCC(ccSE.arqBarraCC); // T3
22. T4 <- ccSE.carregaBarrasSE(ccSE.arqSE); // T4
23. T5 <- ccSE.carregaLigacoesSE(ccSE.arqSE); // T5
24. T8 <- ccSE.carregaBarrasCC(ccSE.arqBarraCC); // T8
25. T11 <- ccSE.criaMatrizIdentidade(ccSE.iBus0); // T11
26. T12 <- ccSE.criaMatrizIdentidade(ccSE.iBus1); // T12
27. T6 <- ccSE.criaMatrizAdmitancias(ccSE.yBus0); // T6
28. T7 <- ccSE.criaMatrizAdmitancias(ccSE.yBus1); // T7
29. T9 <- ccSE.gravaMatriz("yBus0.txt", ccSE.yBus0); // T9
30. T10 <- ccSE.gravaMatriz("yBus0.txt", ccSE.yBus0); // T10
31. T13 <- ccSE.triangMatriz(ccSE.yBus0, ccSE.iBus0); // T13
32. T14 <- ccSE.triangMatriz(ccSE.yBus1, ccSE.iBus1); // T14
33. T15 <- ccSE.calcBackSubs(ccSE.yBus0, ccSE.iBus0, ccSE.zBus0); // T15
34. T16 <- ccSE.calcBackSubs(ccSE.yBus1, ccSE.iBus1, ccSE.zBus1); // T16
35. T19 <- ccSE.calcCC(); // T19
36. T20 <- ccSE.gravaResultados("ccRelatorio.txt"); // T20
37. T17 <- ccSE.gravaMatriz("zBus0.txt", ccSE.zBus0); // T17
38. T18 <- ccSE.gravaMatriz("zBus1.txt", ccSE.zBus1); // T18
39.
40. // ESPG...
41. segp
42. {
43.     model openmp;
44.     parSC1 := ['(T1) || '(T2) || '(T3)]->['(T4) || '(T5) || '(T8)];
45.     parSC2 := ['(T11) || '(T12)];
46.     parSC3 := ['(T9) || '(T10)] -> "(T13).4 -> "(T14).4;
47.     parSC4 := ['(T15) || '(T16)] -> "(T19).4;
48.     parSC5 := ['(T20) || '(T17) || '(T18)];
49.     //parSC := parSC1 -> parSC2 -> parSC3 -> parSC4 -> parSC5;
50. }

```

```

51.
52. #finish
53. display "Dimensoes do Sistema Eletrico(SE)", linefeed;
54. display "-----", linefeed.2;
55. display "numBarras : ", ccSE.numBarras, linefeed;
56. display "numLigacoes: ", ccSE.numLigacoes, linefeed;
57. display "numBarrasCC: ", ccSE.numBarras, linefeed;
58. display "Fim...";
59. #end.

```

Na região **#uses** da listagem 16, o programador informa ao compilador para usar a classe ccSE definida e implementada no arquivo fonte ccSE.hpsi, a qual possui todos os elementos pertinentes ao cálculo de curto circuito. É de responsabilidade do programador que não haja erros de lógica na implementação dos métodos da classe ccSE, pois o compilador não verifica este tipo de problema.

No código da listagem 16, todas as variáveis necessárias à configuração e execução dos métodos são inicializadas internamente, mas estas tarefas não precisam ser executadas pelo compilador, pois fica subentendido que elas foram implementadas, previamente, pelo programador. O trecho de código que vai da linha 19 à linha 38, onde temos os sinônimos das tarefas força o compilador a montar uma tabela de referências cruzadas entre os mesmos e os métodos que eles referenciam, então quando as ESPGs são traduzidas, o compilador fará a substituição dos sinônimos pelos respectivos métodos.

Da linha 41 a 50 há uma região *segp* (acrônimo em inglês para ESPG), onde o compilador traduzirá cada ESPG nela editada na sequência em que aparecem, de forma top-down dentro do corpo do programa fonte em C++ que está sendo montado. Note que o comando *model* informa ao compilador que o modelo de programação paralela é de memória compartilha baseado em OpenMP. Assim, o compilador irá paralelizar todo o programa fonte escrito em Ψ utilizando esse modelo de memória, não apenas os trechos referentes às ESPGs. Se algum método da classe ccSE não estiver paralelizado, então o compilador tentará paralelizar quando encontrá-lo em alguma das ESPGs, caso seja necessário.

O programa Ψ para o modelo pipeline é mostrado na listagem 17. Note que há apenas uma diferença em relação à versão paralela, que é a ESPG para codificação em pipeline. Esta solução fará uso das ESPGs express.(61) à express.(66).

Listagem 17: Modelo pipeline de programa em Ψ para Curto Circuito em Sistemas de Potência.

```

1. //
2. // pipeShortCircuit.ppsi
3. // Cálculo de curto circuito em LTs usando a linguagem Psi, PIPELINE.
4. //
5. #uses
6. // Nenhuma biblioteca padrão será usada explicitamente.
7. ccSE.hpsi; // Mantém a classe SE (Sistema Elétrico) e seus métodos.
8. #start
9. display "Exemplo de Calculo de Curto em LTs, em pipeline com Psi.", linefeed;
10. display "Estudo de Caso desenvolvido no Programa de Doutorado em Ciencias. ",
linefeed;
11. display "Escola Politecnica da Universidade de São Paulo - Brasil.", linefeed;
12. display "===== ", linefeed;
13. #control
14. // Arquivos de dados do SE relacionados.
15. ccSE.arqSE = "EPE_2011.ana"; // Arquivo com 5547 barras, para estudo de caso...
16. ccSE.arqBarraCC = "EPE_2011_5000.bar"; // Arquivo contendo 5000 barras em curto
17.
18. // Apelida os métodos a serem executados para facilitar na edição das ESPGs.
19. T1 <- ccSE.obtemNumBarrasSE(ccSE.arqSE); // T1
20. T2 <- ccSE.obtemNumLigacoesSE(ccSE.arqSE); // T2
21. T3 <- ccSE.obtemNumBarrasCC(ccSE.arqBarraCC); // T3
22. T4 <- ccSE.carregaBarrasSE(ccSE.arqSE); // T4
23. T5 <- ccSE.carregaLigacoesSE(ccSE.arqSE); // T5
24. T8 <- ccSE.carregaBarrasCC(ccSE.arqBarraCC); // T8
25. T11 <- ccSE.criaMatrizIdentidade(ccSE.iBus0); // T11
26. T12 <- ccSE.criaMatrizIdentidade(ccSE.iBus1); // T12
27. T6 <- ccSE.criaMatrizAdmitancias(ccSE.yBus0); // T6
28. T7 <- ccSE.criaMatrizAdmitancias(ccSE.yBus1); // T7
29. T9 <- ccSE.gravaMatriz("yBus0.txt", ccSE.yBus0); // T9
30. T10 <- ccSE.gravaMatriz("yBus0.txt", ccSE.yBus0); // T10
31. T13 <- ccSE.triangMatriz(ccSE.yBus0, ccSE.iBus0); // T13
32. T14 <- ccSE.triangMatriz(ccSE.yBus1, ccSE.iBus1); // T14
33. T15 <- ccSE.calcBackSubs(ccSE.yBus0, ccSE.iBus0, ccSE.zBus0); // T15
34. T16 <- ccSE.calcBackSubs(ccSE.yBus1, ccSE.iBus1, ccSE.zBus1); // T16
35. T19 <- ccSE.calcCC(); // T19
36. T20 <- ccSE.gravaResultados("ccRelatorio.txt"); // T20
37. T17 <- ccSE.gravaMatriz("zBus0.txt", ccSE.zBus0); // T17
38. T18 <- ccSE.gravaMatriz("zBus1.txt", ccSE.zBus1); // T18
39.
40. // ESPG...
41. segp
42. {
43.     model openmp;
44.     pipeSC1 := ['(T1) || '(T2) || '(T3)]->['(T4) || '(T5) || '(T8)];
45.     pipeSC2 := ['(T11) || '(T12)];
46.     pipeSC3 := ['(T6).1 >|| '(T9).1] || ['(T7).1 >|| '(T10).1];
47.     pipeSC4 := ['(T13).2 >|| '(T15).2] -> ['(T14).2 >|| '(T16).2];
48.     pipeSC5 := '(T19).4 -> ['(T20) || '(T17) || '(T18)];
49.     //pipeSC := pipeSC1 -> pipeSC2 -> pipeSC3 -> pipeSC4 -> pipeSC5;
50. }
51.
52. #finish
53. display "Dimensoes do Sistema Eletrico(SE)", linefeed;
54. display "-----", linefeed.2;
55. display "numBarras : ", ccSE.numBarras, linefeed;
56. display "numLigacoes: ", ccSE.numLigacoes, linefeed;
57. display "numBarrasCC: ", ccSE.numBarras, linefeed;
58. display "Fim...";
59. #end.

```

As linhas 46 e 47 são as únicas ocorrências de tarefas sendo executadas em pipeline. Quando o compilador encontra essas linhas ele verificará se os métodos que devem ser executados em pipeline já se encontram implementados dessa forma, caso algum deles ainda não esteja implementado em pipeline, então o

compilador se encarregará de convertê-lo de acordo com regras internas bem definidas.

Na linha 46 há duas regiões de pipeline com dois estágios cada. A primeira região envolve as tarefas sequenciais T6 e T9, onde T6 está no primeiro estágio e T9 no segundo estágio. A segunda região de pipeline envolve as tarefas T7 e T10, onde T10 inicia sua execução após ser sinalizada pela tarefa T7. As quatro tarefas referenciadas pela ESPG da linha 46 são sequenciais internamente, enquanto que as quatro tarefas na linha 47 são paralelas internamente, mas todas devem ser implementadas no modelo pipeline.

9.2.5 Codificação em C++

Os programas fonte equivalentes em linguagem C++, dos programas fonte mostrados anteriormente em linguagem Ψ , não seguem à risca a implementação usando classes. Em vez disso, foram implementados de forma estruturada para facilitar o entendimento por parte de leitores que não conheçam programação orientada a objetos. Foi interesse, também mostrar os programas fonte em sua íntegra, mas para isto eles não poderiam ser muito extensos.

A listagem 18 mostra o equivalente em C++ do programa sequencial codificado em Ψ . O programa fonte completo pode ser consultado no **Anexo E**. É importante se atentar para os nomes dos métodos (funções em C++ estruturado), os quais não são os mesmos sugeridos no quadro 9.1 e usados nas listagens em Ψ .

Listagem 18: Modelo sequencial de programa em C++ para Curto Circuito em Sistemas de Potência.

```

1. /*
2.  * File:    seqShortCircuit.cpp
3.  * Author:  Marlim P. Menezes
4.  * Progama: Doutorado em Ciências - Engenharia Elétrica
5.  * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
6.  * Objetivo: Efetuar o cálculo de curto circuito em linhas de transmissão de modo
7.  * sequencial, para Estudo de Caso em Programação Paralela de Alto Desempenho.
8.  * Created on 29 de Março de 2013, 20:56
9.  */
10.
11. // Biblioteca Padrão...
12. #include <iostream>
13. . . .
14. #include <cmath>
15. . . .
16. // Declaração da estrutura para as matrizes...

```

```

17. struct TMatrix {
18.     unsigned        lin; // Linhas na matriz.
19.     unsigned        col; // Colunas na matriz.
20.     complex<double> *mat; // Matriz.
21. };
22.
23. // Declaração da estrutura para as barras...
24. struct TBarra {
25.     unsigned num;    // Código de identificação da barra: (0..99999)
26.     string  nome;    // Nome da barra
27.     double  vBase;   // Tensão base da barra
28. };
29. // Declaração da estrutura para as ligações...
30. struct TLigacao {
31.     unsigned de;
32.     unsigned para;
33.     int      nc;
34.     string   estado;
35.     double   r1, x1;
36.     double   r0, x0;
37. };
38.
39. // Declaração de constantes...
40. const complex<double> zero = 0.0;
41. const complex<double> um = 1.0;
42. const double         sBase = 100.0; // MVA.
43. const unsigned       nummax = 99999; // Maior identificador de barra.
44. const double         pouco = 1e-10;
45. const double         pi = 3.141592653589793238462643383279; // ...
46.
47. // Funções protótipo...
48. //*****
49. . . .
50. void createIdentityMatrix(TMatrix *I);
51. void triangMatrix(TMatrix *M, TMatrix *I);
52. void backSubstitution(TMatrix *yBus, TMatrix *iBus, TMatrix *zBus, const unsigned col);
53. . . .
54. /*
55.  * Programa principal
56.  */
57. int main() {
58.     . . .
59.     TMatrix *yBus0;
60.     TMatrix *yBus1;
61.     TMatrix *zBus0;
62.     TMatrix *zBus1;
63.     . . .
64.     // Obtenção de dados iniciais do SE.
65.     string arqSE = "EPE_2011.ana"; // Arquivo com 5547 barras, para estudo de caso...
66.     string arqBarraCC = "EPE_2011_5000.bar"; // Arquivo contendo 5000 barras em curto
67.     string strDBAR = "DBAR"; // Identificador da seção das barras no arquivo do SE.
68.     string strDCIR = "DCIR"; // Identificador da seção das ligações no arquivo do SE.
69.     string strLim = "99999"; // Identificador do fim da seção procurada no arquivo do SE.
70.     . . .
71.     // Declara as estruturas para receber as barras e as ligações do SE...
72.     TBarra *barras; // Lista de barras do SE
73.     TLigacao *ligacoes; // Lista de ligações do SE
74.     unsigned *endBar; // Índices das barras...
75.     unsigned *barrasCC; // Barras em curto...
76.     /*
77.      * Calcula o Curto Circuito em todas as barras solicitadas
78.      * =====
79.      */
80.     TMatrix *iBus0, *iBus1; // Matrizes identidades para yBus0 e yBus1.
81.     iBus0 = new TMatrix[sizeof(TMatrix)];
82.     iBus1 = new TMatrix[sizeof(TMatrix)];
83.     iBus0->lin = iBus0->col = iBus1->lin = iBus1->col = numBarras;
84.
85.     // Triangulariza as matrizes yBus...
86.     triangMatrix(yBus0, iBus0);
87.     triangMatrix(yBus1, iBus1);
88.     . . .

```

```

89. // Inverte yBus0 para obter zBus0...
90. // Realiza esta tarefa somente para as colunas necessárias, indicadas
91. // na lista de barras em curto circuito...
92. for(unsigned bc = 0; bc < numBarrasCC; bc++) {
93.     backSubstitution(yBus0, iBus0, zBus0, endBar[barrasCC[bc]]); // inversão por coluna.
94. }
95. . . .
96. // Libera a memoria usada pelas matrizes: yBus0, yBus1, iBus0 e iBus1.
97. // Pois as mesmas não serão mais usadas....
98. delete [] yBus0->mat;
99. delete [] yBus1->mat;
100. delete [] iBus0->mat;
101. delete [] iBus1->mat;
102.
103. /*
104.  * Calcula o curto trifásico e fase terra nas barras selecionadas e
105.  * salva os resultados em disco.
106.  */
107. . . .
108. for(unsigned i = 0; i < numBarrasCC; i++) {
109.     indBar = endBar[barrasCC[i]];
110.     ind     = indBar - 1;
111.     iBase   = sBase / sqrt(3) / barras[indBar].vBase;
112.     zTh1    = kX(1.0/100.0, zBus1->mat[ind * (zBus1->lin +1)]); // Valores em %
113.     zTh0    = kX(1.0/100.0, zBus0->mat[ind * (zBus0->lin +1)]);
114.     iCC3F   = kX(iBase, um / zTh1);
115.     iCCFT   = kX(iBase, kX(3.0, um) / (kX(2.0, zTh1) + zTh0));
116.     iCC3Fpolar = polar(iCC3F);
117.     . . .
118. }
119.
120. // Salva os resultados em disco...
121. for(unsigned r = 0; r < resultado.size(); r++) {
122.     cout << resultado.at(r).c_str(); // Tela
123.     saveReport(arqRelatorio, resultado.at(r)); // Disco
124. }
125. . . .
126. return 0;
127.}

```

Na listagem 18, além dos métodos não terem os mesmos nomes usados na listagem 15, eles também não aparecem na mesma ordem, mas na ordem da tabela 9.4. Note que nesta codificação existem três estruturas declaradas explicitamente para comportar as matrizes, barras e ligações nos trechos que vão da linha 17 a 21, 23 a 27 e 30 a 37, respectivamente. As linhas 98 até 101 são usadas para liberar, explicitamente, a memória usada pelas matrizes yBus0, yBus1, iBus0 e iBus1 para o Sistema Operacional; isto é necessário quando se usa a linguagem C++, e fazer isto é responsabilidade do programador. Em Ψ toda a memória alocada para as diversas estruturas é liberada de forma automática, quando o programa executável termina, tirando essa responsabilidade do programador.

O cálculo do curto circuito em si é realizado no trecho de código que abrange as linhas 108 a 118 e, por fim os resultados são gravados em disco no código que vai da linha 121 até 124. Uma prática que foi bastante explorada na codificação em

linguagem C++, visando gerenciar a memória de forma mais adequada e, com isto ganhar um pouco de velocidade na execução do programa como um todo, foi o uso intenso de ponteiros.

A listagem 19 mostra o equivalente em C++ com OpenMP do programa paralelo codificado em Ψ . O programa fonte completo pode ser consultado no **Anexo F**.

Listagem 19: Modelo paralelo de programa em C++ para Curto Circuito em Sistemas de Potência.

```

1.  /*
2.  * File:   parShortCircuit.cpp
3.  * Author: Marlim P. Menezes
4.  * Programa: Doutorado em Ciências - Engenharia Elétrica
5.  * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
6.  * Objetivo: Efetuar o cálculo de curto circuito em linhas de transmissão de modo
7.  * paralelo, para Estudo de Caso em Programação Paralela de Alto Desempenho.
8.  * Created on 29 de Março de 2013, 20:56
9.  */
10.
11. // Biblioteca Padrão...
12. #include <iostream>
13. ...
14. #include <omp.h>
15.
16. // Declaração da estrutura para as matrizes...
17. struct TMatrix {
18.     unsigned        lin; // Linhas na matriz.
19.     unsigned        col; // Colunas na matriz.
20.     complex<double> *mat; // Matriz.
21. };
22.
23. // Declaração da estrutura para as barras...
24. struct TBarra {
25.     unsigned num; // Código de identificação da barra: (0..99999)
26.     string nome; // Nome da barra
27.     double vBase; // Tensão base da barra
28. };
29.
30. // Declaração da estrutura para as ligações...
31. struct TLigacao {
32.     unsigned de;
33.     unsigned para;
34.     int nc;
35.     string estado;
36.     double r1, x1;
37.     double r0, x0;
38. };
39.
40. // Declaração de constantes...
41. const complex<double> zero = 0.0;
42. const complex<double> um = 1.0;
43. const double sBase = 100.0; // MVA.
44. const unsigned nummax = 99999; // Maior identificador de barra.
45. const double pouco = 1e-10;
46. const double pi = 3.141592653589793238462643383279; // ...
47.
48. // Declaração de variáveis globais...
49. int numProcs; // Número de CPUs usadas em regiões paralelas...
50.
51. // Funções protótipo...
52. //*****
53. void loadSEBusData(TBarra *barras, unsigned *endBar,
54.     const string arq, const string str1, const string nm);
55. void loadSECircuitData(TLigacao *ligacoes, const string arq,
56.     const string str2, const string nm);
57. void loadSCDataBusses(unsigned *barCC, const string arq);

```

```

58. . . .
59. void   triangMatrix(TMatrix *M, TMatrix *I);
60. void   backSubstitution(TMatrix *yBus, TMatrix *iBus, TMatrix *zBus, const unsigned
col);
61.
62. /*
63.  * Programa principal
64.  */
65. int main() {
66.     . . .
67.     // Declaração das matrizes
68.     TMatrix *yBus0;
69.     TMatrix *yBus1;
70.     TMatrix *zBus0;
71.     TMatrix *zBus1;
72.
73.     . . .
74.     // Obtenção de dados iniciais do SE.
75.     string arqSE   = "EPE_2011.ana"; // Arquivo com 5547 barras, para estudo de caso...
76.     string arqBarraCC = "EPE_2011_300.bar"; // Arquivo contendo 300 barras em curto
77.     string strDBAR = "DBAR"; // Identificador da seção das barras no arquivo do SE.
78.     string strDCIR = "DCIR"; // Identificador da seção das ligações no arquivo do SE.
79.     string strLim  = "99999"; // Identificador do fim da seção procurada no arquivo do SE.
80.
81.     // Obtem o número de barras e de ligações do SE, para alocação dinâmica de
82.     // memória para as estruturas...
83.     unsigned numBarras = 0, numLigacoes = 0, numBarrasCC = 0;
84.     #pragma omp parallel sections num_threads(3) // 3 procs...
85.     {
86.         #pragma omp section // tarefa 1 => 1 proc
87.         {
88.             guessSEBusSize(numBarras, arqSE, strDBAR, strLim);
89.             . . .
90.             // Obtem o número de barras em curto...
91.             guessSCSize(numBarrasCC, arqBarraCC); // Original
92.         }
93.     }
94.
95.     // Define as dimensões das matrizes com os valores obtidos do arquivo do SE...
96.     yBus1->lin = yBus0->lin = numBarras;
97.     yBus1->col = yBus0->col = numBarras; // Matriz expandida p comporta a Identidade.
98.     zBus1->lin = zBus1->col = zBus0->lin = zBus0->col = numBarras; // Matrizes quadradas.
99.     . . .
100.    // Declara as estruturas para receber as barras e as ligações do SE...
101.    TBarra *barras; // Lista de barras do SE
102.    TLigacao *ligacoes; // Lista de ligações do SE
103.    unsigned *endBar; // Índices das barras...
104.    unsigned *barrasCC; // Barras em curto...
105.
106.    // Aloca memória para as barras, ligações, endbar e barrasCC...
107.    . . .
108.
109.    // Carrega os dados do SE: Barras e Ligações...
110.    #pragma omp parallel sections num_threads(3)
111.    {
112.        #pragma omp section // tarefa 1 => 1 proc
113.        {
114.            loadSEBusData(barras, endBar, arqSE, strDBAR, strLim);
115.        }
116.        #pragma omp section // tarefa 2 => 1 proc
117.        {
118.            loadSECircuitData(ligacoes, arqSE, strDCIR, strLim);
119.        }
120.        #pragma omp section // tarefa 3 => 1 proc
121.        {
122.            loadSCDataBusses(barrasCC, arqBarraCC); // Carrega a lista de barras em curto...
123.        }
124.    }
125.    . . .
126.    // Cria as matrizes yBus0 e yBus1...
127.    #pragma omp parallel sections //num_threads(numCPUs)
128.    {
129.        #pragma omp section // tarefa 1 => 2 procs

```

```

130.     {
131.         // Monta a matriz yBus0...
132.         createYBus0(yBus0, endBar, ligacoes, numLigacoes);
133.         . . .
134.     }
135. }
136. . . .
137. /*
138.  * Calcula o Curto Circuito em todas as barras solicitadas
139.  * =====
140.  */
141. TMatrix *iBus0, *iBus1; // Matrizes identidades para yBus0 e yBus1.
142. ...
143. iBus0->lin = iBus0->col = iBus1->lin = iBus1->col = numBarras;
144. . . .
145. // Triangulariza as matrizes yBus...
146. #pragma omp parallel sections //num_threads(numProcs)
147. {
148.     #pragma omp section // tarefa 1 => 2 procs
149.     {
150.         createIdentityMatrix(iBus0);
151.     }
152.     #pragma omp section // tarefa 2 => 2 procs
153.     {
154.         createIdentityMatrix(iBus1);
155.     }
156. } // parallel sections
157.
158. // Executa duas tarefas independentes em paralelo...
159. #pragma omp parallel sections num_threads(numProcs)
160. {
161.     #pragma omp section // tarefa 1 => numProcs/2 procs
162.     {
163.         triangMatrix(yBus0, iBus0);
164.     }
165.     #pragma omp section // tarefa 2 => numProcs/2 procs
166.     {
167.         triangMatrix(yBus1, iBus1);
168.     }
169. } // parallel sections
170.
171. // Inverte yBus0 para obter zBus0...
172. #pragma omp parallel sections //num_threads(numProcs)
173. {
174.     #pragma omp section // tarefa 1 =>
175.     {
176.         #pragma omp parallel for //num_threads(numProcs/2)
177.         for(unsigned bc = 0; bc < numBarrasCC; bc++) {
178.             backSubstitution(yBus0, iBus0, zBus0, endBar[barrasCC[bc]]); // inversão por
coluna...
179.         }
180.     } // tarefa 1
181.     #pragma omp section // tarefa 2
182.     {
183.         #pragma omp parallel for //num_threads(numProcs/2)
184.         for(unsigned bc = 0; bc < numBarrasCC; bc++) {
185.             backSubstitution(yBus1, iBus1, zBus1, endBar[barrasCC[bc]]);
186.         }
187.     } // tarefa 2
188. } // parallel sections
189. . . .
190. // Cálculos de curto e fase...
191. . . .
192. #pragma omp parallel for num_threads(numProcs)
193. for(unsigned i = 0; i < numBarrasCC; i++) {
194.     unsigned indBar = endBar[barrasCC[i]];
195.     unsigned ind = indBar - 1;
196.     double iBase = sBase / sqrt(3) / barras[indBar].vBase;
197.     complex<double> zTh1 = kX(1.0/100.0, zBus1->mat[ind * (zBus1->lin + 1)]); // Valores
em %
198.     complex<double> zTh0 = kX(1.0/100.0, zBus0->mat[ind * (zBus0->lin + 1)]); // Valores
em %
199.     complex<double> iCC3F = kX(iBase, um / zTh1);

```

```

200.     complex<double> iCCFT = kX(iBase, kX(3.0, um) / (kX(2.0, zTh1) + zTh0));
201.     . . .
202. }
203.
204.     . . .
205.     return 0;
206. }
```

A diferença básica entre as listagens 19 e 18 está na aplicação das diretivas de paralelismo da API OpenMP, que são usadas para delimitar regiões paralelas e paralelização, essencialmente, de laços. A paralelização das tarefas não ocorre neste programa fonte, necessariamente, na mesma ordem que é sugerida na tabela 9.5, entretanto com a disposição das tarefas como foi implementado em linguagem C++ e OpenMP permitiu-se atingir um grau de paralelismo satisfatório para os propósitos deste estudo, e isto pode ser observado na análise dos resultados obtidos a seguir, no item 9.2.6 Resultados.

Analisando as regiões paralelas do programa fonte da listagem 19, observa-se que: no trecho de código que vai da linha 84 a 93, existe uma região paralela com três seções, onde há três tarefas independentes, codificadas em sequencial, sendo executadas em paralelo, ou seja, concorrentemente. Essa região equivale à *ESPG parSC1* da expressão *express.(55)*, onde então sendo executadas as tarefas '(T1), '(T2) e '(T3); no trecho que vai da linha 110 a 124, há uma região paralela onde estão sendo executadas as tarefas sequencias '(T4), '(T5) e '(T8) paralelamente; da linha 127 até 135 existe uma região paralela executando, concorrentemente, as tarefas paralelas "(T6).2 e "(T7).2 em dois processadores cada uma delas; da linha 146 até a linha 156 tem-se uma região paralela executando, concorrentemente, as duas tarefas paralelas "(T11).2 e "(T12).2 em dois processadores cada; no trecho que vai da linha 159 até a linha 169 as tarefas sendo executadas em paralelo são a "(T13).2 e "(T14).2; da linha 172 a 188 existe outra região paralela executando, concorrentemente, as duas tarefas paralelas "(T15). E "(T16).2 e; por fim tem no trecho de código que vai da linha 192 até a linha 202 apenas a tarefa "(T19).4 sendo executada por quatro processadores, nesta última região paralela do programa.

Todas as demais tarefas que não apareceram nesta listagem são sequenciais e estão localizadas em pontos específicos do programa fonte, de modo a não comprometer o comportamento esperado quando o mesmo é executado.

O próximo item mostra uma análise estatística dos tempos de execução de ambos os modelos de programação, sequencial e paralelo, listados anteriormente, para as mesmas bases de dados.

9.2.6 Resultados

Apesar do estudo de caso disponibilizar uma pequena base de dados – com apenas 9 barras e 14 ligações no sistema elétrico mais 5 e 9 barras consideradas em curto circuito no mesmo SE –, para testes e corroboração da solução matemática do problema proposto, essa foi muito útil nas etapas da modelagem computacional e depuração dos algoritmos que compõem o programa desenvolvido. Os dados fornecidos nessa pequena base foram obtidos de um sistema elétrico real, obtido do Operador Nacional do Sistema Elétrico – ONS.

A tabela 9.7 mostra os tempos de processamento para os métodos sequencial e paralelo, e o ganho de velocidade (*speedup*) do modelo serial para o paralelo.

Tabela 9.7: Análise de ganho de velocidade (*speedup*). Número de processadores: 4 (2 por tarefa).

Número Barras no SE	Número Ligações no SE	Número de Barras a Calcular Curto	Sequencial (s)	Paralelo (s)	Speedup (t_s/t_p)
5547	9788	300	8975	4681	1,917325
5547	9788	400	9090	4756	1,911270
5547	9788	500	9209	4817	1,911771
5547	9788	1000	9702	5078	1,910595
5547	9788	2000	10799	5659	1,908288
5547	9788	3000	11914	6209	1,918828
5547	9788	4000	12980	6804	1,907702
5547	9788	5000	14057	7400	1,899595

Portanto, para verificar e comprovar-se a eficiência dos algoritmos computacionais paralelos, em relação aos mesmos algoritmos, porém sequenciais, foi necessário o uso de uma base de dados de um sistema elétrico real muito maior do que aquele usado nos testes iniciais da solução deste problema.

Na tabela 9.7, a primeira coluna mostra o número verdadeiro de barras e a segunda informa o número verdadeiro de ligações (ou circuitos) existentes no mesmo sistema elétrico. A terceira coluna, com números na cor vermelha, indica a quantidade de

barras a calcular o curto no SE, cujas correntes de curto trifásico e curto fase terra devem ser calculadas pelos dois algoritmos, utilizando a mesma base de dados original. Essa última base de dados do sistema elétrico estudado foi obtida da EPE – Empresa de Pesquisa Energética e contém as quantidades de barras e ligações apresentadas nas duas colunas à esquerda da tabela 9.7.

A tabela 9.8 oferece uma maneira de visualizar os dados de forma mais natural apresentando os tempos em horas, ao invés de milhares de segundos como mostrado na tabela 9.7.

Tabela 9.8: Análise de ganho de velocidade (speedup). Número de processadores: 4 (2 por tarefa).

Número Barras no SE	Número Ligações no SE	Número de Barras a Calcular Curto	Sequencial (h:m:s)	Paralelo (h:m:s)	Speedup (t_s/t_p)
5547	9788	300	2:29:35	1:18:01	1,917325
5547	9788	400	2:31:30	1:19:16	1,911270
5547	9788	500	2:33:29	1:20:17	1,911771
5547	9788	1000	2:41:42	1:24:38	1,910595
5547	9788	2000	2:59:59	1:34:19	1,908288
5547	9788	3000	3:18:34	1:43:29	1,918828
5547	9788	4000	3:36:20	1:53:24	1,907702
5547	9788	5000	3:54:17	2:03:20	1,899595

A Figura 43 mostra a comparação entre os tempos de execução da tabela 9.7, de forma linear, para análise de desempenho.

Figura 43 – Análise linear da evolução do tempo de execução dos algoritmos de cálculo de curto circuito Sequencial e Paralelo executado com 4 CPUs (2 por tarefa).

Na Figura 44 tem-se o gráfico comparativo de ganho de velocidade do algoritmo paralelo em relação ao seu equivalente sequencial, isto é, o speedup. Verifique os dados da tabela 9.7 (a tabela 9.8 também serve, mas será mais complicado, pois os tempos estão formatados com hora:minuto:segundo).

Figura 44 – Ganho de velocidade do algoritmo sequencial depois de paralelizado. Praticamente dobrou a velocidade de execução.

Todos os algoritmos escritos em linguagem C++, testados anteriormente – sequencial e paralelo, seguem à risca as abordagens predefinidas nas ESPGs obtidas para os mesmos e, que são apresentadas nas expressões **express.(52)** à **express.(54)** e expressões **express.(55)** à **express.(60)** para os modelos de programação sequencial e paralelo da linguagem Ψ , respectivamente.

Com estes resultados conclui-se que, apesar de ter atingido somente o dobro da velocidade de processamento, utilizando o algoritmo paralelo contra o sequencial, é compensador, uma vez que tal paralelismo foi realizado de forma elementar, sem muito aprofundamento nos recursos avançados da API paralela, o que nos possibilitou realizar este estudo, neste caso particular tratando-se da OpenMP.

Todas as matrizes foram construídas de forma linear, pois dessa forma obtém-se, de imediato, cerca de 10% no desempenho do programa como um todo. Assim, também é possível se criar matrizes muito maiores na memória, do que utilizando a forma matricial tradicional.

O estudo de caso não exigiu alocação dinâmica de memória na forma de lista, entretanto a memória alocada para cada matriz foi obtida em tempo de execução

utilizando-se o operador **new** da linguagem C++. Isto tornou possível gerenciar a alocação de memória para os gigantescos vetores utilizados como pseudo matrizes.

Devido à necessidade do uso de diversas estruturas de dados simples e quase totalmente independentes, porém muito grandes, foi possível se obter várias tarefas passíveis de paralelização interna, bem como de execução concorrente. Isto contribuiu significativamente para a melhoria no ganho de velocidade da execução do programa como um todo, não apenas isoladas. Todavia, apesar de não implementado em C++, foi possível alcançar-se o pipeline em três ocasiões.

O objetivo deste estudo de caso foi mostrar que com a aplicação da Metodologia de Paralelização e Otimização proposta neste trabalho de pesquisa pode-se obter ganhos de desempenho significativos quando se paraleliza uma aplicação computacional desse porte, permitindo o uso mais racional dos atuais computadores de mesa, de forma relativamente simples.

Apesar dos tempos de execução terem sido elevados, os resultados atingidos foram satisfatórios, pois é sabido que com as técnicas matemáticas rudimentares utilizadas esses tempos seriam esperados. Todavia, se técnicas avançadas de análise numérica e álgebra linear fossem utilizadas, certamente os resultados seriam comparáveis àqueles obtidos por sistemas computacionais comerciais, que fazem uso de APIs matemáticas desenvolvidas para computação de alto desempenho, e custam muito caro, além de serem executadas, normalmente em computadores de médio e grande porte. Existem algumas dessas APIs preparadas para serem executadas em computadores de mesa, tais como LAPACK, ScaLAPACK, LAPACK++, dentre outras.

Conclui-se, dessa forma, que a metodologia proposta é adequada para a paralelização de sistemas computacionais da área Sistemas de Potência da grande área de Engenharia Elétrica.

10 CONCLUSÕES E TRABALHOS FUTUROS

A Metodologia de Paralelização e Otimização aqui proposta provou funcionar adequadamente nos três modelos computacionais desejados: sequencial, paralelo e pipeline. No caso do modelo pipeline foi considerado apenas como proposta metodológica através do uso da nova linguagem de programação Ψ , que foi desenvolvida exclusivamente para completar a metodologia como um todo, dando suporte às Expressões Semânticas de Processamento Geral - ESPG, que por sua vez, também é parte integrante deste trabalho de pesquisa.

Ao longo da análise dos resultados obtidos nos Estudos de Caso apresentados chegou-se à conclusão de que estando em posse de uma infraestrutura de hardware adequada, é possível se atingir um ganho de desempenho consideravelmente elevado quando se utiliza essa metodologia. No entanto, ela pode ser melhorada ainda mais se exploradas técnicas avançadas inerentes ao uso da API paralela utilizada, juntamente com técnicas de otimização matemática adequadas para a solução do problema proposto.

A primeira parte da metodologia de paralelização e otimização, vide capítulo 5 – Extrair Paralelismo foi explorada em sua totalidade no estudo de caso número 2 – Cálculo de Curto Circuito em Sistemas de Potência, mostrando ser viável e eficiente no que diz respeito à forma como se obtém paralelismo de alto grau de qualidade na solução de um problema real de engenharia.

Apesar de se apresentar eficaz na solução de estudos de caso envolvendo apenas duas áreas do Conhecimento Humano, a Matemática e a Engenharia Elétrica, esta metodologia foi desenvolvida visando-se a extração de paralelismo de qualquer tipo de problema que venha gerar um modelo computacional consistente.

Uma vez que a metodologia ESPG venha possibilitar a tradução direta de diagramas de blocos que representam processamento computacional de, praticamente, qualquer arquitetura e grau de complexidade em uma expressão relativamente simples de se escrever e interpretar, é possível reduzir-se o suficiente a possibilidade de estudos de conceitos avançados e complicados sobre computação paralela, e até erros grosseiros que por ventura possam ser cometidos por usuários, mesmo que sejam especialistas na área de computação paralela. Já que a interpretação da ESPG deverá ser feita de forma automática pela linguagem de programação Ψ . No entanto, o usuário tem total liberdade de decidir se deixa a tradução da ESPG por conta do compilador de linguagem Ψ , ou se ele próprio, de posse de avançados conhecimentos de computação paralela e distribuída, a traduzirá para a linguagem de programação que desejar.

Psi realiza todo o trabalho de paralelismo através de diversos modelos de memória já consagrados, tais como OpenMP, MPI e outros, tornando os detalhes de implementação desses modelos **ocultos** para o programador ao traduzir a ESPG para o código escrito na API ou linguagem informada pelo usuário, pois essa tarefa é altamente onerosa para qualquer programador, já que o mesmo deverá conhecer a arquitetura de hardware escolhida, ainda que seja num nível básico.

Outro fator muito relevante a ser considerado na fase de codificação da ESPG, em uma linguagem de programação diferente da Ψ , é o fato do usuário perder tempo precioso que deveria ser investido na otimização dos modelos matemático e computacional, não tendo que se preocupar com detalhes de implementação tais como gerenciar a colisão no acesso à memória, deadlock, perda de sincronismo e outros problemas inerentes à programação paralela. Deve ficar claro que se o usuário decidir traduzir a ESPG para a linguagem de programação desejada, que não seja a

linguagem Ψ , por conta de suas habilidades com a programação paralela, já não será mais garantido um paralelismo otimizado do seu problema.

Acredita-se que, com esta proposta se contribui positivamente para a melhoria do uso mais racional dos recursos disponibilizados pelos atuais computadores pessoais, utilizando-se da computação paralela de alto desempenho em máquinas desse porte e baixo custo.

Pretende-se ainda, apresentar um passo a mais nas soluções da metodologia de paralelização e otimização aqui proposta, adotando uma abordagem computacional densamente (massivamente - *massively*) paralela, a fim de explorar plenamente o potencial dos recursos das tecnologias das modernas Unidades de Processamento Gráfico de Uso Geral (GPGPU), que já estão sendo construídas com milhares de núcleos físicos e disponíveis no mercado a preços baixíssimos frente a seu poder de processamento, comparado com os computadores de maior porte.

Como demonstrado ao longo deste trabalho, ficou claro que a metodologia para cálculo do determinante de matrizes densas nele proposta se encaixa perfeitamente no modelo de memória e microarquitetura das GPGPU atuais, uma vez que é possível se executar o algoritmo MDIM com $(n - 1)$ iterações, isto é, em termos de complexidade computacional linear $O(n)$. Isso, porque se vê a possibilidade de resolver o determinante de cada submatriz de ordem $n = 2$ por thread, levando-se a calcular os determinantes de todas as $(n - 1)^2$ submatrizes de ordem dois concorrentemente. Pretende-se, também fazer melhorias no modelo computacional desenvolvido para processamento paralelo usando memória compartilhada em *PThreads*, *OpenMP avançada*, *MPI*, *Thrust*, *CUDA* e *OpenCL* para integrar, futuramente, a biblioteca de funções matemáticas a ser disponibilizada para a linguagem Ψ .

Como principal trabalho futuro, pretende-se aperfeiçoar a metodologia de extração e otimização de paralelismo como um todo, de modo a permitir níveis de ganho de desempenho ainda mais elevados. Também, uma melhoria no formato da ESPG já está em vista, de modo a torná-la mais simples e funcional. As melhorias previstas

para a linguagem Ψ são muitas, começando pela sua estrutura que ainda pode se tornar mais simples, uma reestruturação na proposta de projeto do compilador, no qual se planeja implementar a capacidade de tomada de decisão sobre os níveis de otimização do código gerado utilizando IA – Inteligência Artificial; a implementação de uma biblioteca avançada de matemática e outras áreas também faz parte do conjunto de melhorias.

Entretanto, não menos importante que todas as melhorias citadas anteriormente, está o planejamento para transformar a linguagem Ψ , que por enquanto é apenas uma proposta acadêmica, em um projeto avançado de nível de Pós Doutorado, objetivando atingir um público não apenas acadêmico, mas científico, industrial, entretenimento e comercial.

Acredita-se que as metodologias propostas neste trabalho de pesquisa, venham contribuir positivamente para as mais diversas áreas do conhecimento que, de alguma forma, resumam a solução de seus problemas em modelos matemáticos potenciais a serem implementados, computacionalmente, em qualquer das linguagens de programação existentes, principalmente na linguagem de programação paralela aqui proposta, Ψ .

Os estudos de caso constantes deste texto serviram para corroborar as especificações apresentadas nas várias etapas do processo de paralelização e otimização de modelos matemáticos e computacionais, quando do uso das metodologias nele propostas. É importante lembrar que o objetivo dos estudos de caso foi, somente, de comprovar a eficiência, simplicidade e praticidade do uso da Metodologia de Paralelização e Otimização de Modelos Matemáticos e Computacionais aqui proposta, não o de competir com metodologias utilizadas comercialmente, hoje em dia. Entretanto, é objetivo chegar-se neste nível, mas neste momento é, tão somente, mostrar que de posse de uma metodologia adequada pode-se transformar modelos sequenciais em paralelos e pipeline de forma simples, levando-se ao uso mais racional dos atuais computadores pessoais.

REFERÊNCIAS

ADAMI, Christoph; **Introduction to Artificial Life**; Springer-Telos; USA; 1998.

ADELI, Hojjat, HUNG, Shin-Lin; **Machine Learning – Neural Networks, Genetic Algorithms, and Fuzzy Systems**; John Wiley & Sons, Inc.; USA; 1995.

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D.; **Compilers, Principles, Techniques and Tolls**; Addison-Wesley; USA; 1986.

AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi and ULLMAN, Jeffrey D.; **Compilers, Principles, Techniques and Tolls Second Edition**; PEARSON Addison-Wesley; USA; 2008.

ALBA, Enrique; **Parallel Metaheuristics – A New Class of Algorithms**; Published by John Wiley & Sons, Inc.; New Jersey; USA; 2005.

ANTON, Howard and RORRES, Chris; **Elementary Linear Algebra: Applications version – 10th Edition**; Canada; 2010.

BARBOSA, Helio J. C.; **Uma Introdução aos Algoritmos Genéticos – Minicurso**; LNCC – Laboratório Nacional de Computação Científica / CNPq; Rio de Janeiro; Brasil; 1997.

BEN-ARI, M.; **Principles of Concurrent and Distributed Programming – Second Edition**; Addison-Wesley; USA; 2006.

CANTO, Nilton C. F.; **Uma Abordagem Evolutiva para Identificação de Procedimentos de Raciocínio Humano**; Tese – Escola Politécnica da Universidade de São Paulo – EPUSP; São Paulo; Brasil; 2008.

CHAPMAN, Barbara; JOST, Gabrielle; PAS, Ruud van der and KUCK, David J.; **Using OpenMP – Portable Shared Memory Parallel Programming**; The MIT Press.; USA; 2008.

CODENOTTI, Bruno, LEONCINI, Mauro; **introduction to Parallel Processing**; International Computer Science Series – Addison-Wesley Publishing Company Inc.; 1993.

COLLET, Pierre, FONLUPT, Cyril, HAO, Jin-Kao, SCHOENAUER, Marc; **Artificial Evolution**; 5th International Conference – Evolution Artificielle EA 2001, Le Creusot, France, October 2001, Selected Papers, LNCS 2310; Springer; France; 2002.

COSNARD, Michel; ROBERT, Yves; QUINTON, Patrice and RAYNAL, Michel; **Parallel & Distributed Algorithms**; Elsevier Science Publishers; Amsterdam; The Netherlands; 1989.

COSNARD, Michel, TRYSTRAM, Denis; **Parallel Algorithms and Architectures**; International Thomson Computer Press.; London; UK; 1995.

FARBER, Rob; **CUDA – Application design and Development**; Morgan Kaufmann Publishers; USA; 2011.

FLYNN, Michael J.; **Very High-Speed Computing Systems**; Proceedings of the IEEE, vol. 54, p. 1901-1909; December, 1966.

FLYNN, Michael J.; **Some Computer Organizations and Their Effectiveness**; IEEE Transactions on Computers, vol. c-21, n^o 9; September, 1972.

GRAMA, Ananth and GUPTA, Anshul and KARYPIS, George and KUMAR, Vipin; **Introduction to Parallel Computing Second Edition**; PEARSON – Addison-Wesley; 2003.

HENNESSY, John L. e PATTERSON, David A.; **Arquitetura de Computadores – Uma abordagem quantitativa – 4^a ed.**; Elsevier Inc.; 2008.

HERLIHY, Maurice and SHAVIT, Nir; **The Art of Multiprocessor Programming**; Morgan Kaufmann Publishers; Elsevier Inc.; USA; 2008.

HUGHES, Cameron; HUGHES, Tracey; **Parallel and Distributed Programming Using C++**; Addison-Wesley; Boston – MA; USA; 2004.

HWU, Wen-mei W.; **GPU – Computing Gems: Jade Edition**; Morgan Kaufmann Publishers; USA; 2012.

JAMIESON, Leah H., GANNON, Dennis B. and DOUGLASS, Robert J.; **The Characteristics of Parallel Algorithms**; The MIT Press; Massachusetts Institute of Technology; USA; 1987.

KECMAN, Vojislav; **Learning and Soft Computing – Support Vector Machines, Neural Networks, and Fuzzy Logic Models**; The MIT Press; Massachusetts; USA; 2001.

KIRK, David B.; HWU, Wen-mei W.; **Programming Massively Parallel Processors – A Hands-on Approach**; Morgan Kaufmann Publishers; USA; 2010.

KLINE, Rex B.; **Principles and Practice of Structural Equation Modeling – Third Edition**; The Guilford Press; New York; USA; 2011.

KUNCHEVA, Ludmila I.; **Combining Pattern Classifiers – Methods and Algorithms**; John Wiley; New Jersey; USA; 2004.

LEWIS, Harry R.; PAPADIMITRIOU, Christos H.; **Elements of Theory of Computation, 2nd Edition**; Prentice-Hall, Inc.; USA; 1998.

MAHANTY, R. N., GUPTA, P. B. D.; **Application of RBF neural network to fault classification and location in transmission lines**; Generation, Transmission and Distribution, IEE Proceedings, Vol. 151, Issue 2 March 2004, p. 201-212.

MATTSON, Timothy G.; SANDERS, Beverly A.; MASSINGILL, Berna L.; **Patterns for Parallel Programming**; Software Patterns Series – Addison-Wesley; Boston – MA; USA; 2005.

MENEZES, M. P., PEREIRA, C. E. de M., SATO, L. M.; **IDM – A New Parallel Methodology to Calculate the Determinant of Matrices of the Order n , with Computational Complexity $O(n)$** ; IEEE Latim America Transactions – Vol. 10 No. 1; Jan 2012; p. 1357-1363.

MILLER, Russ, BOXER, Laurence; **Algorithms Sequential & Parallel : a unified approach – Second Edition**; Computer Engineering Series – Charles River Media, Inc.; Canada; 2005.

O'REILLY, Randall C., MUNAKATA, Yuko; **Computational Explorations in Cognitive Neuroscience – Understanding the mind by Simulating the Brain**; The MIT Press; Massachusetts; USA; 2000.

PAITHANKAR, Y. G.; **Transmission Network Protection – Theory and Practice**; Marcel Dekker, Inc.; New York; USA; 1998.

PATTERSON, David A. e HENNESSY, John L.; **Organização e Projeto de Computadores – A interface hardware / software – 3^a ed.**; Elsevier Inc.; 2005.

PEREIRA, Carlos E. de M. **Localizadores digitais de falta em linhas de transmissão**. Dissertação de Mestrado, ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO; São Paulo; Brasil; 1999.

PEREIRA, Carlos E. de M.; **Localização Digital de Faltas em Linhas de Transmissão: Desenvolvimento e Aperfeiçoamento de Algoritmos**; Tese de Doutorado, ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO; São Paulo; Brasil; 2003.

PIAZZI, Pierluigi; **Aprendendo Inteligência: manual de instruções do cérebro para alunos em geral**; (Coleção Neuropedagogia; vol. 1); Editora Aleph Ltda; São Paulo; Brasil; 2008.

PITAS, Ioannis; **Parallel Algorithms for Digital Image Processing Computer Vision and Neural Networks**; *Jhon Wiley & Sons Ltd*; Canada; 1995.

PRESS, Willian H.; FLANNERY, Brian P.; TEUKOLSKY, Saul A. and VETTERLING, Willian T.; **Numerical Recipes in C – The Art of Scientific Computing**; Published by the Press Syndicate of the University of Cambridge; USA; 1990.

QUINN, Michael J.; **Parallel Programming in C with MPI and OpenMP: 1st ed.**; McGraw-Hill; New York; USA; 2004.

RICIERE, Aguinaldo Prandini; **Foundations of the Self-Destructive Operator**; *Published by Barthô Publication*; São Paulo; Brazil; 1987.

RUSSELL, Stuart, NORVIG, Peter; **Inteligência Artificial – Tradução da Segunda Edição**; Elsevier Editora Ltda; Rio de Janeiro; Brasil; 2004.

SANDERS, Jason; KANDROT, Edward; **CUDA By Example – An Introduction to General-Purpose GPU Programming**; Addison-Wesley; USA; 2011.

SCOTT, Michael L.; **Programming Language Pragmatics: Third Edition**; Morgan Kaufmann Publishers; USA; 2009.

SIPSER, Michael; **Introduction to the Theory of Computacion – second edition – International Edition**; Course Technology, Cengage Learning; Boston; MA; USA; 2006.

STALLINGS, William; **Arquitetura e Organização de Computadores – 8^a ed.**; Pearson Prentice Hall; 2012.

STEEB, Willi-Hans; **The Nonlinear Workbook, 5th Edition**; World Scientific Publishing Co. Pte. Ltd.; USA; 2011.

TANENBAUM, Andrew S.; WOODHULL, Albert S.; **Sistemas Operacionais – projeto e Implementação – 2^a ed.**; Pearson Education Company – Bookman; 2002.

TANENBAUM, Andrew S.; **Organização Estruturada de Computadores – 4^a ed.**; Pearson Education Company; 2012.

VALIANT, Leslie G.; **A Bridging Model For Parallel Computation**; Communications of the ACM, Vol. 33, No. 8, pp.103-111; USA; 1990.

WILKINSON, Barry, ALLEN, Michael; **Parallel Programming – Techniques and applications Using Networked Workstations and Parallel Computers**; Prentice Hall; USA; 1999.

Endereços eletrônicos na Web:

CAPS; **OpenHMPP - HMPP for Hybrid Multicore Parallel Programming**; url: <http://www.caps-entreprise.com/wp-content/uploads/2012/11/OpenHMPP-Directives.pdf>; WEB; 2012.

KHRONOS; **OpenCL – Open Computing Language**; url: <http://www.khronos.org/opencl/>; WEB; 2012.

NVIDIA; **CUDA – Parallel Computing Platform and Programming Model**; url: <https://developer.nvidia.com/what-cuda>; WEB; 2012.

OPENACC; **OpenACC – Directives for Accelerators**; url: <http://openacc.org/>; WEB; 2012.

REFERÊNCIAS COMPLEMENTARES

ACHESON, David; **From Calculus to Chaos: *An Introduction to Dynamics***; Oxford University Press; New York – USA; 1997.

AKL, Selim G.; **Parallel Computation: Models And Methods**; Prentice-Hall, Inc.; USA; 1997.

ALBA, Enrique; **Parallel metaheuristics: a new class of algorithms**; *Published by* John Wiley & Sons, Inc.; New Jersey; USA; 2005.

ASHLOCK, Daniel; **Evolutionary Computation for Modeling and Optimization**; *Published by* Springer Science+Business Media, Inc.; New York; USA; 2006.

ASSAN, Aloisio Ernesto; **Método dos Elementos Finitos**; Editora Unicamp; Brasil; 2010.

BEAR, Mark F.; CONNORS, Barry W. e PARADISO, Michael A.; **Neurociências – Desvendando o Sistema Nervoso – 2ª Edição – 3ª Reimpressão – Traduzida para português**; Artmed Editora S.A.; São Paulo; Brasil; 2006.

BELLAC, Michel Le; **A Short Introduction to Quantum Information and Quantum Computation**; *Cambridge University Press*; Printed in the United Kingdom; 2006.

BOUCHARA, Jacques C.; CARRARA, Vera L.; HELLMEISTER, Ana Catarina P.; SALVITTI, Reinaldo; **Cálculo Integral Avançado**; Edusp – Editora da Universidade de São Paulo; Brasil; 1999.

BRONSON, Richard; **Schaum's Outline of Theory and Problems of Operations Research**; MacGraw-Hill, Inc.; USA; 1982.

BURDEN, Richard L.; FAIRES, J. Douglas; **Análise Numérica 7ª ed. (Numerical Analysis 7th ed.)**; Pioneira Thonson Learning, Ltda; Brasil; 2003.

BURKE, Edmund K., KENDALL, Graham; **Search Methodologies Introduction Tutorials in Optimization and Decision Support Techniques**; *Published by* Springer Science+Business Media, Inc.; New York; USA; 2005.

CHAUDHURI, Pranay; **Parallel Algorithms Design and Analysis**; *Printed by* Prentice Hall of Australia Pty Ltd; Australia; 1992.

CORMEN, Thomas H., LEISERSON, Charles E., RIVEST, Ronald L., STEIN, Clifford; **Algoritmos: Teoria e Prática (Introduction to algorithms – Second Edition)**; Editora Campus Ltda; Brasil; 2002.

CULLER, David E., SINGH, Jaswinder Pal, with GUPTA, Anoop; **Parallel Computer Architecture: a hardware/software approach**; Morgan Kaufmann Publishers, Inc.; USA; 1999.

CUNHA, M. Cristina C.; **Métodos Numéricos**; Editora Unicamp; Brasil; 2011.

DANTAS, Mario; **Computação Distribuída de Alto Desempenho: redes, clusters e GRIDs computacionais**; Axcel Books do Brasil Editora Ltda; Rio de Janeiro; 2005.

DEITEL, H. M.; DEITEL, P. J.; **C++: Como Programar 5ª Edição**; PEARSON Prentice-Hall; Brasil; 2006-2012.

EWEN, Dale; TOPPER, Michael A.; **Technical Calculus**; Prentice-Hall, Inc.; USA; 1977.

FAZZIO, Adalberto, WATARI, Kazunori; **Introdução à Teoria de Grupos aplicada em moléculas e sólidos**; *Editora da Universidade Federal de Santa Maria*; Santa Maria – RS; 2009.

FINKEL, Raphael A.; **Large-grain parallelism – Three case studies**; *Computer Science Department – University of Wisconsin – Madison*; Madison, WI; USA; 1987.

FORTUNA, Armando de Oliveira; **Técnicas Computacionais para Dinâmica dos Fluidos – Conceitos Básicos e Aplicações**; Edusp – Editora da Universidade de São Paulo; Brasil; 2012.

GIBBONS, Alan, RYTTER, Wojciech; **Efficient Parallel Algorithms**; Cambridge University Press; Cambridge; Great Britain; 1990.

GOLDBARG, Marco Cesar, LUNA, Henrique Pacca L.; **Otimização Combinatória e Programação Linear: Modelos e Algoritmos**; *Editora Campus Ltda*; Rio de Janeiro – Brasil; 2005.

GALOIS, Évarist; **Galois's Theory**; *Published by* VEDA, Publishing House of the Slovak Academy of Sciences; Czechoslovakia; 2005.

GARCIA, Claudio; **Modelagem e Simulação de Processos Industriais e de Sistemas Eletromecânicos**; Edusp – Editora da Universidade de São Paulo; Brasil; 2009.

HANSON, Andrew J.; **Visualizing Quaternions, series in interactive 3D technology**; *Printed by* Elsevier Inc.; China; 2006.

HOLUB, Allen I.; **Compiler Design in C**; Prentice-Hall International Editions; USA; 1990.

HROMKOVIČ, Juraj; **Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics – 2nd Edition**; Springer-Verlag Berlin Heidelberg; Germany; 2004.

HUMES, Ana Flora P. de Castro; MELO, Inês S. Homem de Melo; YOSHIDA, Luzia Kazuko; MARTINS, Wagner Tunis; **Noções de Cálculo Numérico**; McGraw-Hill do Brasil; Brasil; 1984.

HWANG, Kai; **Advanced Computer Architecture: Paralellism, Scalability, and Programmability**; McGraw-Hill, Inc.; International Editions 1993; USA; 1993.

JIN, Jianming; **The Finite element Method in Electromagnetics, Second Edition**; John Wiley & Sons, Inc.; USA; 2002.

KELAREV, Andrei; **Graph Algebras and Automata**; *Pure and Applied Mathematics – A Dekker series of Monographs and Textbooks*; Marcel Dekker, Inc.; New York; USA; 2003.

KITAEV, A. Yu., SHEN, A. H., VYALYI, M. N.; **Classical and Quantum Computation - 47**; *American Mathematical Society*; Rhode Island; USA; 2002.

KURFESS, Franz; **Parallelism in Logic: its potential for performance and program development**; Verlag Vieweg; Germany; 1991.

KRISHNAMURTHY, E. V.; **Parallel Processing: Principles and Practice**; International Computer Science Series – Addison-Wesley Publishing Company; New Zealand; 1993.

LANDER, Jeff; **Graphics Programming Methods**; *Published and Printed by* Charles River Media, Inc.; Massachusetts; USA; 2003.

LATHI, B. P.; **Linear Systems and Signals, Second Edition**; Oxford University Press; USA; 2004;

MIKLOSKO, J., KOTOV, V. E.; **Algorithms, software and hardware of parallel computers**; *Published by* VEDA, Publishing House of the Slovak Academy of Sciences; Czechoslovakia; 1984.

MONTICELLI, Alcir e GARCIA, Ariovaldo; **Introdução a Sistemas de Energia Elétrica**; Editora UNICAMP; Campinas; Brasil; 2003.

MORETTIN, Pedro A.; TOLOI, Clélia M. C.; **Análise de Séries Temporais, 2ª Edição**; Editora Blucher; Brasil; 2011.

NELSON, Philip A., SNYDER, Lawrence; **Programming Paradigms for Nonshared Memory Parallel Computers**; *Computer Science Department – University of Washington; Seattle, Washington; USA*; 1987.

NETO, João José; **Introdução à Compilação**; LTC – Livros Técnicos e Científicos Editora S.A.; Brasil; 1987.

NIELSEN, Michael A., CHUANG, Isaac L.; **Quantum Computation and Quantum Information**; *Published by Press Syndicate of the University of Cambridge*; Cambridge University Press; United Kingdom; 2000.

NYHOFF, Larry, LEESTMA, Sanford; **FORTRAN 77 and Numerical Methods for Engineers and Scientists**; *Prentice-Hall, Inc.*; New Jersey – USA; 1995.

OGATA, Katsuhiko; **Engenharia de Controle Moderno 5ª Edição**; PEARSON Prentice-Hall; Brasil; 2010.

ORSINI, Luiz de Queiroz; CIPPARRONE, Flavio A. M.; **Simulação Computacional de Circuitos Elétricos**; Edusp – Editora da Universidade de São Paulo; Brasil; 2011.

PINTO, Paulo Roberto Margutti; **Introdução à lógica simbólica**; Editora UFMG; Minas Gerais; Brasil; 2006;

PRIGOGINE, Ilya; **As Leis do Caos (Le leggi del caos - 1993)**; *Tradução para o português pela Fundação Editora da UNESP (FEU)*; 2002.

RAO, Valluru B., RAO, Hayagriva V.; **C++ Neural Networks and Fuzzy Logic – Second Edition**; MIS Press; New York; USA; 1995.

RICIERE, Aguinaldo Prandini; **Cálculo Variacional, Cipós e Bolhas de Sabão**; *Impresso por Prandiano Comércio de Livros LTDA*; São José dos Campos; São Paulo; Brasil; 1993.

RIELE, H. J. J. te, DECKKER, Th. J., VORST, H. A. van der; **Algorithms and applications on vector and parallel computers** (Special topics in supercomputing; v. 3); *Published and Printed by Elsevier Science Publishers B. V.*; Amsterdam; The Netherlands; 1987.

ROBERT, Yves; **The impact of vector and parallel architectures on the Gaussian elimination algorithm**; *Published by Manchester University Press; Printed by Oxford Road, Manchester M13 9PL, UK; John Wiley & Sons, Inc; USA; 1990.*

SUTTER, Herb; **Programação Avançada em C++: 40 novos quebra-cabeças de engenharia, problemas de programação e soluções**; PEARSON Makron Books; Brasil; 2006.

TAVARES, L. Valadares, CORREIA, F. Nunes; **Otimização Linear e Não Linear: Conceitos, Métodos e Algoritmos**; *Fundação Calouste Gulbenkian; Lisboa; Portugal; 1986.*

TEIXEIRA-DIAS, F.; PINHO-DA-CRUZ, J.; VALENTE, R. A. Fontes; SOUZA, R. J. Alves de; **Método dos Elementos Finitos – Técnicas de Simulação Numérica em Engenharia**; ETEP – EDIÇÕES TÉCNICAS E PROFISSIONAIS; Lisboa; Portugal; 2007.

TRAPPENBERG, Thomas P.; **Fundamentals of Computational Neuroscience**; OXFORD University Press; New York; USA; 2007.

WALMSLEY, Mark; **Multi-Threaded Programming in C++**; *Springer-Verlag London; Great Britain; 2000.*

YATES, Roy D. and GOODMAN, David J.; **Probability and Stochastic Processes – A Friendly Introduction for Electrical and Computer Engineers – Second Edition**; John Wiley & Sons, Inc.; USA; 2005.

ZANETTA JÚNIOR, Luiz Cera; **Transitórios Eletromagnéticos em Sistemas de Potência**; Edusp – Editora da Universidade de São Paulo; Brasil; 2003.

ZIENKIWICZ, O. C. and MORGAN, K.; **Finite Elements & Approximation**; John Wiley & Sons, Inc.; USA; 1983.

ZOMAYA, Albert Y.; **Parallel Computing for Bioinformatics and Computational Biology: models, enabling technologies, and case studies**; Wiley-Interscience – John Wiley & Sons, Inc.; New Jersey; USA; 2006.

APÊNDICES

APÊNDICE A

LINGUAGEM Ψ

Psi é uma letra grega (maiúscula: Ψ , minúscula: ψ). Ela tem como principal objetivo tornar a tarefa árdua de paralelização em uma tarefa mais amigável e agradável de ser realizada, de forma direta e objetiva para o programador, tornando mais racional o uso dos recursos computacionais disponibilizados pelas atuais tecnologias de hardware.

A linguagem Ψ foi desenvolvida para trabalhar no paradigma da programação paralela de alto desempenho, entretanto dando maior suporte às aplicações paralelas locais que fazem uso do modelo de memória compartilhada, sobre as mais modernas arquiteturas de processadores utilizados em computadores de baixo custo

atualmente. Isto se aplica, também aos processadores gráficos de uso geral (GPGPU).

Neste momento, Psi dá suporte somente à OpenMP com C++, entretanto ela foi planejada para suportar, também os modelos de memória Pthreads, MPI, Thrust, OpenCL e CUDA.

A.1 Especificação Simplificada

Ψ distingue letras minúsculas de maiúsculas, ou seja, ela é *case-sensitive*, somente para seu conjunto de palavras reservadas, sendo que os identificadores declarados pelo programador não possuem esta característica. Assim, um identificador declarado como “CASA” poderá sofrer variações em sua forma ao longo do programa fonte, como por exemplo para “Casa” ou “cASA” etc, não importando quais letras estão escritas em caixa baixa ou caixa alta. Em linguagem C++ essas variações representam identificadores distintos, já em Pascal elas têm o mesmo significado; para nós seres humanos, as três formas representam o mesmo objeto, tanto abstrato quanto real. Esta técnica pode tornar o programa fonte menos legível, entretanto do ponto de vista humano ela é mais consistente.

A listagem a seguir mostra as palavras reservadas existentes, atualmente, na linguagem Ψ .

Comando	Descrição	Exemplo de uso
<u>1. Operadores Matemáticos e de Atribuição</u>		
<code>+, ++, +=</code>	Adição (strings - concatenação, inteiros, reais e complexos).	<code>casa = "ca" + "sa";</code> <code>5 = 2 + 3;</code> <code>5.5 = 2.25 + 3.25;</code> <code>3+5.5j = (1+3j)+(2+2.5j);</code>
<code>-, --, -=</code>	Subtração (inteiros, reais e complexos).	<code>5 - 3 = 2</code>
<code>*, *=</code>	Multiplicação (inteiros, reais e complexos).	<code>2 * 3 = 6</code>
<code>/, /=</code>	Divisão (inteiros, reais e complexos).	<code>6 / 2 = 3</code>
<code>^ ou **</code>	Potenciação (inteiros, reais e complexos).	<code>2 ^ 3 = 2^3 = 2**3 = 2 ** 3 = 8</code>
<code>.</code>	Produto escalar ou interno unclidiano (Álgebra Linear), ou operador de acesso a elementos de um objeto (estrutura, classe,	<code>u = (-2,3) e v = (1,5) =></code> <code>pe = u.v = (2, 3) . (-1,</code> <code>5) = -2*1+3*5 = -2+15 = 13</code>

	etc).	(produto escalar dos vetores u e v). <code>corCasa = casa.getCor()</code> (operador de acesso ao método <code>getCor()</code> da classe <code>casa</code>).
<code>&</code>	Produto vetorial (Álgebra Linear).	$u = (1, 2, -2)$ e $v = (3, 0, 1) \Rightarrow pv = u \& v = (1, 2, -2) \& (3, 0, 1) = (\det[2, -2; 0, 1], -\det[1, -2; 3, 1], \det[1, 2; 3, 0]) = (2, -7, -6)$ (produto vetorial entre os vetores tridimensionais u e v)
<code>[...]</code> ou <code>[...,dif]</code>	Intervalo numérico de uma sequência finita. Sem <code>dif</code> =diferença os elementos da sequência são incrementados de um e um, com <code>dif</code> os elementos da sequência são incrementados ou decrementados desse valor. O valor de <code>dif</code> pode ser o resultado de uma função. O tipo numérico da sequência é definido internamente, em tempo de compilação, e pode ser inteiro ou real.	<code>[1...100]</code> (sequência numérica iniciada em 1 e terminada em 100 com incremento unitário; o tipo abstrato dos elementos da sequência é inteiro). <code>[1...100,2]</code> (sequência numérica iniciada em 1 e terminada em 99, pois este é o maior valor inteiro menor do que 100, com incremento de dois; ou seja, é a sequência dos números ímpares menores do que 100). <code>[1...100,0.5]</code> (sequência numérica de números reais de 1.0 até 100.0, incrementando de 0.5 em 0.5).
<code> ... </code>	Módulo (ou valor absoluto) de um número inteiro, real ou complexo.	$x = -3 = 3$; $x = -2.5 = 2.5$; $x = 3 + 4i = 5$.
<code> ... </code>	Norma de um vetor (Álgebra Linear).	Dado o vetor $u=(-2,1,2)$, sua norma é: $ u = (-2,1,2) = \sqrt{4+1+4} = 3$.
<code>=</code>	Atribuição de conteúdo a uma variável numérica ou alfanumérica.	<code>a = b = 5</code> ; // atribuição <code>A.lines = 1000</code> ; <code>B.columns = A.lines</code> ;
<code><-</code>	Operador de atribuição de apelido (<i>alias</i>) a elementos de programação de nomes extensos com nomes simples, para referenciá-los em ESPGs. Só pode ser usado dentro de blocos <code>sepg{ }</code> .	<code>cDA <- carragaDadosDoArquivo(...)</code> Neste caso <code>cDA</code> é um apelido (<i>alias</i>) que referenciará a função do lado direito do operador.
<code>:=</code>	Atribuição do modelo de processamento ao repositório ESPG.	<code>T1 <- A.randomMatrixGen(-100, +100)</code> ; <code>T2 <- B.randomMatrixGen(0, 100]</code> ; <code>T3 <- P.matrixMul(A,B)</code> ; <code>parESPG := ["(T1) </code>

		"(T2)] -> "(T3);
?	Leitura de dados via teclado (sem distinção entre numérico e texto). Também faz parte do operador lógico ternário.	A.lines = ?; A.columns = A.lines;
=> (then)	Operador de tomada de decisão: Equivalente ao comando "then" de uma operação de decisão "if".	if (<expr.>) then {} ≡ if(<expr.>) => {}
2. Operadores Lógicos		
==, !=	Operadores de avaliação entre dois valores, em expressões lógicas. Igualdade: ==; desigualdade: !=.	(a == b); // a é igual b? (a != b); // a é diferente de b?
>, >=	Operadores maior do que e maior do que ou igual a.	(a > b); // a é maior do que b? (b >= c); // b é maior do que ou igual a c?
<, <=	Operadores menor do que e menor do que ou igual a.	(a < b); // a é menor do que b? (b <= c); // b é menor do que ou igual a c?
<expr.> ? <val_1> : <val_2>;	Operador lógico ternário (idêntico ao da linguagem C++ e Java).	val = (a < 0) ? 0 : a; // val não pode ser negativo.
not ou !	Não (Negação ou Inversão).	if(not(a)) then {} ≡ if(!a) => {} b = not(a) = !a;
and	E.	if(a == b and b != c) => {} if(a and b) then {} c = a and b;
or	Ou.	if(a or b) then {} c = a or b;
xor	Ou Exclusivo.	if(a xor b) then {} ≡ if(a xor b) => {} c = a xor b;
nand	Negação do E.	if(a nand b) => {} c = a nand b;
nor	Negação do Ou.	if(a nor b) => {} c = a nor b;
nxor	Negação do Ou Exclusivo (Coincidência).	if(a nxor b) => {} c = a nxor b;
3. Operadores de processamento paralelo em Psi		
`(	Processamento sequencial interno ao bloco.	`(T1) // A tarefa T1 possui código sequencial.
"(	Processamento paralelo interno ao bloco.	"(T1) // A tarefa T1 possui código paralelo.
“(	Processamento pipeline interno ao bloco.	“(T1) // A tarefa T1 possui código interno pipeline ou paralelo, ou faz pipeline com alguma outra tarefa externa.
->	Processamento de fluxo de processamento	"(T1) -> `(T2); // A tarefa T2 possui código

	externo sequencial.	sequencial interno e é dependente da tarefa T1, que possui código paralelo interno.
<code> </code>	Processamento de fluxo de processamento externo em paralelo.	<code>"(T1) "(T2); //</code> A tarefa T2 possui código sequencial interno e é independente da tarefa T1, que possui código paralelo interno. Ambas as tarefas são executadas em paralelo.
<code>> </code>	Processamento de fluxo de processamento externo em pipeline.	
4. Delimitadores de comentários		
<code>//</code>	Comentário de uma linha.	<code>// Isto é um comentário.</code>
<code>/*...*/</code>	Comentário de múltiplas linhas.	<code>/* Isto * é um comentário. */</code>
5. Delimitadores Gerais		
<code>#</code>	Início de linha com diretiva de escopo (Obrigatoriamente na primeira coluna da linha de código e precedendo uma das diretivas de região, sem espaços em branco).	<code>#uses ... #start ... #control ... #finish ... #end.</code>
<code>;</code>	Fim de uma linha de código.	<code>display "Hello World! I am Psi";</code>
<code>.</code>	Precedido da diretiva #end, representa o fim do programa Psi.	<code>#end.</code>
<code>{}</code>	Delimitadores de blocos de comandos de programação. Não são permitidos em expressões matemáticas.	
<code>[]</code>	Delimitadores intermediários em ESPGs. Também, são usados nas representações de vetores e matrizes multidimensionais. Não é permitido em expressões matemáticas.	
<code>()</code>	Delimitadores internos em ESPGs. Também, são usados como delimitadores de métodos, ou seja, procedimentos e funções. São permitidos em expressões matemáticas, para definir a prioridade de cálculo na execução dessas expressões.	
6. Diretivas Psi		

control	Diretiva de região de controle em um programa em linguagem Ψ .	... #control ...
end	Diretiva de finalização de sub região e do programa principal da Ψ .	... #end.
finish	Diretiva de região final em um programa em linguagem Ψ .	... #finish ...
model	Modelo de linguagem paralela de destino (openmp, opencl, mpi, nvgpu ²³). Mais de um modelo de programação paralela pode ser especificado em um único bloco, resultando em programação paralela híbrida.	... model openmp, // OpenMP nvgpu; // CUDA ...
segp	Diretiva de bloco de processamento especial ESPG. Pode ter vários desses blocos na região #control.	... segp { model nvgpu; // CUDA ... } ...
shared	Diretiva de uso de memória compartilhada pela sub região da Ψ .	RESERVADA PARA USO FUTURO
start	Diretiva de região de início do programa em linguagem Ψ .	... #start ...
uses	Diretiva de região de importação de bibliotecas de padrões em linguagem Ψ . Informa ao compilador as bibliotecas externas que serão usadas.	#uses ... ppsiLAPACK, // biblioteca padrão de Álgebra Linear. TMatrix.hpsi; // Cabeçalho de usuário. ...
<u>7. Palavras reservadas de uso geral</u>		
abort	Aborta o processamento em qualquer ponto do programa.	
append	Informa ao comando file que os dados serão gravados no final do arquivo. Se o arquivo não existir, então um novo deve ser criado.	
break	Interrompe o processamento sendo executado. Somente em laços.	

²³ NVGPU (acrônimo para **NVidia GPU**) é um nome alternativo para a linguagem **CUDA** da NVIDIA.

case	Opção do comando switch. Idêntico ao comando case de Pascal.	
class	Declaração de um objeto virtual que modela um objeto do mundo real (equivalente a class de C++ e Java).	
continue	Continua o processamento interrompido pelo comando break Somente em laços..	
default	Parte do comando switch. Quando nenhuma das opções case for verdadeira, este será o valor aceito.	
display	Mostra conteúdo no dispositivo de saída padrão (atualmente é a tela do monitor).	
do	Comando de controle de laço terminado com o comando while. Também é o complemento do comando for.	
else	Comando auxiliar de decisão usado em conjunto com o comando "if".	
file	Grava ou lê dados de um arquivo externo.	
for	Comando de controle de laço.	
if	Comando de decisão e controle de fluxo de dados e desvio.	
line	Informa que o formato de impressão dos dados no dispositivo de saída deve ser linear (usado em conjunto com o comando "file").	
machine	Tipo binário ou "baixo nível" do arquivo externo.	
network	Comunicação bidirecional de dados através do modem ou da placa de rede padrão.	
pause	Aguarda uma tecla ser pressionada, para continuar o fluxo normal do programa.	
printer	Imprime variáveis e textos na impressora padrão.	
private	Especificador de escopo das regiões internas a uma classe. Os atributos declarados nesta região são privados à classe (segue o padrão da OOP).	
protected	Especificador de escopo das regiões internas	

	a uma classe. Os atributos declarados nesta região são privados à classe (segue o padrão da OOP).	
public	Especificador de escopo das regiões internas a uma classe. Os atributos declarados nesta região são públicos dentro e fora do escopo da classe (segue o padrão da OOP).	
read	Informa ao comando file que os dados serão lidos do disco ou de outro dispositivo de armazenamento externo.	
step	Passo incremental/decremental auxiliar para o comando for. Pode ser omitido somente se o incremento for unitário.	
switch	Comando chaveador de opções. É o mesmo que switch...case...default de C/C++.	
table	Informa que o formato de impressão dos dados no dispositivo de saída deve ser matricial (usado em conjunto com os comandos display, file e printer).	
text	Tipo ASCII ou "alto nível" do arquivo externo	
then (=>)	Executa o bloco de uma comparação "if".	
wait	Aguarda o termino de uma tarefa para efetuar o sincronismo.	RESERVADA PARA USO FUTURO.
while	Comando de controle de laço. Também é o comando de decisão com comando do.	
write	Informa ao comando file que o arquivo deve ser criado no disco ou em outro dispositivo de armazenamento externo. Se o arquivo já existir, os seus dados serão substituídos pelos novos dados.	
...	O conjunto de palavras reservadas ainda está em fase de desenvolvimento.	

A seguir estão ilustrados os formatos das instruções definidas até este momento, como parte do conjunto de palavras reservadas de Ψ .

1. Comando if...then...else:
 - 1.1 if(expr) then comando;

- 1.2 if(expr) => comando;
- 1.3 if (expr) then { comandos; }
- 1.4 if(expr) => { comandos; }
- 1.5 if(expr) then comando; else comando;
- 1.6 if(expr) => { comandos; } else { comandos; }
- 1.7 if(expr) => { comandos; } else if(expr) then { comando; } else { comandos; }
- 1.8 Assim por diante.
2. Comando for...step...do:
 - 2.1 for i = 1...10 do { comandos; }
 - 2.2 for i = 1...n step 2 do { comandos; } // Conta até o maior i <= n, de 2 em 2.
 - 2.3 for i = 0...10 step 2.5 do soma = soma + i;
 - 2.4 for i = 0...n step 2.5 do soma += i; // Conta até o maior i <= n, de 2.5 em 2.5.
3. Comando multiopção switch...case...default:
 - 3.1 switch(expr) { case 0: comandos; break; case 1,2: comandos; break; default: comandos; break; }
 - 3.2 switch(val) { case 0...9: { comandos; break; } case 10: { comandos; break; } }
 - 3.3 switch(simbolo) { case 'a'...'f','A'...'F': { comandos; break; } case '-': { comandos; break; } case '+': { comandos; break; } case '/': { comandos; break; } case '*': { comandos; break; } default: { comandos; } }
4. Comando while...do:
 - 4.1 while(expr) do comando;
 - 4.2 while(expr) do { comandos; }
5. Comando do...while:
 - 5.1 do { comandos; }while(expr);
6. Outros comandos serão apresentados oportunamente.

APÊNDICE B

RESUMO DA TEORIA DE DETERMINANTES

Este resumo foi extraído de (MENEZES et. al.; 2012), vide REFERÊNCIAS.

Estabelecida ao longo do século 17, a teoria de determinantes foi descoberta durante um estudo de processos para resolver sistemas de equações lineares. Embora atualmente os determinantes não sejam uma ferramenta tão prática para se resolver sistemas de equações, eles são usados, por exemplo, para sintetizar certas expressões matemáticas complicadas [1].

Neste apêndice apresentamos um breve resumo da metodologia mais básica para se calcular o determinante de matrizes de ordem n , já estabelecida no âmbito acadêmico.

Alguns métodos tradicionais são apresentados na álgebra linear: Regra de Sarrus (usada apenas para matrizes de ordem $n = 3$), Expansão em cofatores (para matrizes de ordem $n \geq 2$): Regra de Cramer, Teorema Fundamental de Laplace, o Método de Eliminação de Gauss-Jordan, a decomposição em LU e a Regra de Chió, além de outras [1], [2], [3], [4]. Há também abordagens baseadas em meta-heurísticas e métodos iterativos [5], [6], mas a metodologia proposta neste texto é baseada no método determinístico.

A eq.(B.1) representa uma matriz quadrada de ordem n , enquanto que a eq.(B.2) é sua forma compacta e a eq.(B.3) é uma das muitas maneiras de calcular seu determinante.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \quad (\text{B.1})$$

$$A = [a_{ij}]_{n \times n} = [a_{ij}] \quad (\text{B.2})$$

$$|A| = \det A = \prod_{i=1}^n a_{ii} \quad (\text{B.3})$$

A eq.(B.3) é usada para calcular o determinante da matriz A de ordem n , quando ela está escalonada na forma triangular ou mesmo quando decomposta em LU, e o determinante é simplesmente o produto dos elementos da diagonal principal [2], [3], [4].

B.1 Trabalhos Relacionados

Excelentes soluções computacionais relacionadas à solução de sistemas lineares e determinantes, escritos em linguagem C, são apresentadas em [4]. Embora otimizadas, suas abordagens são todas sequenciais. Os autores exploram o potencial dos ponteiros em C muito bem, a fim de alcançar um alto desempenho nos algoritmos mostrados, mas a complexidade computacional das metodologias utilizadas permanece inalterada; por exemplo, verifica-se que o cálculo de determinantes continua sendo $O(n^3)$.

O método de eliminação de Gauss-Jordan, com complexidade computacional $O(n^3)$, é proposto na forma de algoritmo sequencial escrito em linguagem C, no capítulo 12

de [5] e capítulo 10 de [6]. Ambos os livros também apresentam uma solução paralela com complexidade computacional $O(n^2)$.

Uma metodologia derivada da expansão da Regra de Chió é apresentada em [7], onde o autor afirma que são necessários $(n - 1)^2$ processadores para se obter o determinante da matriz fornecida, cuja complexidade computacional é $O(n)$. Existem muitos outros estudos muito importantes sobre paralelismo de metodologias para o cálculo de determinantes de matrizes de ordem n com complexidade computacional linear, as quais podem ser encontradas em diversas revistas e outras publicações.

B.2 Regra de Chió

A regra de Chió é usada para se calcular o determinante de uma matriz de ordem n , com $n \geq 2$, reduzindo-se a sua ordem de uma unidade a cada iteração, com a restrição do elemento $a_{11} = 1$. Por outro lado, se $a_{11} \neq 1$ será necessário se aplicar algumas propriedades dos determinantes para forçar $a_{11} = 1$.

A regra de Chió está resumida nos passos seguintes, conforme [1]:

1. Consideremos a matriz A de ordem $n \geq 2$, com $a_{11} = 1$;
2. Eliminar a primeira linha e primeira coluna de A ;
3. Para cada elemento remanescente na matriz, subtrair o produto dos elementos que são perpendiculares às extremidades do elemento considerado, a primeira linha e primeira coluna;
4. Com as diferenças obtidas, construímos a matriz de ordem $(n - 1)$, cujo determinante é igual ao determinante da matriz de entrada A_n .
5. Retornar ao passo 1, até que $n = 1$.

A seguir temos a forma analítica para a regra de Chió para uma matriz A_n , com $n = 4$. Forçamos o elemento $a_{11} = 1$ em todos os estágios, como descrito nos cinco passos da regra de Chió, apresentada anteriormente.

$$A_4 = \begin{bmatrix} 1 & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Solução:

$$|A_4| = \begin{vmatrix} 1 & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{vmatrix} \quad (\text{B.4})$$

Aplicando-se a regra de Chió à matriz A_4 da eq.(B.4) em um primeiro instante, sua ordem cairá de $n = 4$ para $n = 3$, conforme mostrado na eq.(B.5) e eq.(B.6).

$$|A_{4-1}| = \begin{vmatrix} a_{22} - a_{21} * a_{12} & a_{23} - a_{21} * a_{13} & a_{24} - a_{21} * a_{14} \\ a_{32} - a_{31} * a_{12} & a_{33} - a_{31} * a_{13} & a_{34} - a_{31} * a_{14} \\ a_{42} - a_{41} * a_{12} & a_{43} - a_{41} * a_{13} & a_{44} - a_{41} * a_{14} \end{vmatrix} \quad (\text{B.5})$$

$$|A_3| = \begin{vmatrix} 1 & a'_{12} & a'_{13} \\ a'_{21} & a'_{22} & a'_{23} \\ a'_{31} & a'_{32} & a'_{33} \end{vmatrix} \quad (\text{B.6})$$

No entanto, devemos continuar aplicando a regra de Chió à matriz A_3 , enquanto sua ordem for $n \geq 2$ (veja passo 1).

$$|A_{3-1}| = \begin{vmatrix} a'_{22} - a'_{21} * a'_{12} & a'_{23} - a'_{21} * a'_{13} \\ a'_{32} - a'_{31} * a'_{12} & a'_{33} - a'_{31} * a'_{13} \end{vmatrix}$$

$$|A_2| = \begin{vmatrix} 1 & a''_{12} \\ a''_{21} & a''_{22} \end{vmatrix} \quad (\text{B.7})$$

Como podemos ver no desenvolvimento acima, quando os seguintes passos de 1 até 5 são aplicados à matriz A_3 na eq.(B.6) obtemos a eq.(B.7), cuja matriz foi reduzida de uma linha e uma coluna, contendo uma nova ordem ($n = n - 1 = 2$).

$$\therefore \det A_4 = a''_{11} * |A_2| = a''_{11} * (a''_{22} - a''_{21} * a'''_{12}) \quad (\text{B.8})$$

onde,

$$a'''_{12} = a''_{12} / a''_{11}.$$

Conforme o passo 1 da regra de Chió, devemos continuar reduzindo a ordem da matriz, pelo simples fato da eq.(B.7) mostrar que a matriz já estar com ordem $n = 2$, podemos obter o seu determinante com a eq.(B.8).

B.3 Referências

- [1] Gelson Iezzi e Samuel Hazzan, Fundamentos de Matemática Elementar: Sequências, Matrizes, Determinantes e Sistemas - Vol. 4 2ª ed., Atual Editora Ltda, São Paulo, SP, Brasil, 1977.
- [2] Anton Howard e Chris Rorres, Álgebra Linear com Aplicações 8ª ed., Bookman, Porto Alegre, RS, Brasil, 2001.
- [3] Ralph Costa Teixeira, Álgebra Linear: Exercícios e Soluções, IMPA, Rio de Janeiro, RJ, Brasil, 2010.
- [4] William H. Press, and Brian P. Flannery, and Saul A. Teukolsky, and William T. Vetterling, Numerical Recipes in C – The Art of Scientific Computing, Published by the Press Syndicate of the University of Cambridge, Printed in USA, 1990.
- [5] Michael J. Quinn, PARALLEL PROGRAMMING in C with MPI and OpenMP: 1ª ed., McGraw-Hill, New York, USA, 2004.

- [6] Barry Wilkinson and Michael Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, Inc, New Jersey, USA, 1999.

APÊNDICE C

CÁLCULO DE CURTO CIRCUITO EM SISTEMAS DE POTÊNCIA

O objetivo do estudo de curto-circuito em sistemas de potência é dimensionar a proteção do mesmo a sobretensões e sobrecorrentes originadas por defeitos.

Através do estudo de curto-circuito, define-se o conjunto de pára-raios, de disjuntores, e a calibração dos relés, que promovem a proteção do sistema, de forma que mesmo em caso de defeitos, o sistema de potência seja o mais confiável possível, evitando degeneração de equipamentos e garantindo a continuidade no fornecimento.

O estudo de harmônicas é importante, pois fornece subsídios para que possa ser evitada a propagação excessiva de harmônicas pela rede elétrica, o que prejudica a qualidade da tensão fornecida.

A. Modelos de Sequência Utilizados para Simulação Computacional

Em sistemas de potência, a teoria das componentes simétricas é a base para estudos de curto circuito. Dessa forma, o modelamento dos elementos de sistema de potência será realizado em termos de sequência direta, inversa e zero (positiva, negativa e zero).

O objetivo final é definir modelos, que possam ser facilmente utilizáveis

computacionalmente, para a montagem da matriz Ybus, que representa o sistema de potência a ser estudado.

1) Linhas de Transmissão:

Figura C.1 – Modelo de linha de transmissão.

Na Figura C.1, temos:

l = comprimento da linha de transmissão (km)

$Z_c = \sqrt{Z/Y}$ = impedância característica (Ω)

γ = constante de propagação = $\sqrt{ZY}$ (km^{-1})

2) Transformador De Dois Enrolamentos:

Figura C.2 – Modelo de sequência direta do transformador de dois enrolamentos.

Para a sequência zero o modelo depende do tipo de ligação do primário e do secundário (delta, estrela solidamente aterrada, estrela aterrada por impedância e estrela não aterrada).

3) Transformador de Três Enrolamentos:

Figura C.3 – Modelo de sequência direta do transformador de três enrolamentos.

Na Figura C.3, temos:

$$\begin{cases} Z_p = (Z_{ps} + Z_{pt} - Z_{st}) / 2 \\ Z_s = (Z_{ps} + Z_{st} - Z_{pt}) / 2 \\ Z_t = (Z_{pt} + Z_{st} - Z_{ps}) / 2 \end{cases} \quad (C.1)$$

4) Gerador: Para gerador na barra B da Figura a seguir.

Figura C.4 – Modelos de sequência direta e sequência zero do gerador.

A tabela C.1 apresenta as variações da reatância de sequência direta de um gerador.

Tabela C.1: Variação da reatância de sequência direta do gerador.

Instante em Análise	Reatância
Imediatamente após defeito	X''_g (subtransitória)
Alguns instantes após defeito	X'_g (transitória)
Regime permanente	X_g (não saturada)

Na Figura C.4, temos:

E_g = tensão interna do gerador

X_g = reatância do gerador

Z_g = impedância de aterramento do gerador

Z_0 = impedância de sequência zero do gerador

5) *Motor*: Para motor na barra B da Figura a seguir.

Figura C.5 – Modelos de sequência direta e sequência zero do motor.

O comportamento do valor da reatância X_m é o mesmo que o da reatância X_g do gerador. E os demais parâmetros da Figura C.5, são equivalentes ao do gerador.

6) *Reator*:

Figura C.6 – Modelo do reator para as três seqüências.

Na Figura C.6, temos:

$$X_r = \frac{V_n^2}{Q_n} \quad (\text{C.2})$$

7) Banco De Capacitores:

Figura C.7 – Modelo do banco de capacitores para as três sequências.

Na Figura C.7, temos:

$$X_c = \frac{V_n^2}{Q_n} \quad (\text{C.3})$$

8) Cargas: São modeladas como uma impedância da barra para a referência, cujo valor é dado a seguir:

$$Z_{carga} = \left(\frac{V^2}{\sqrt{P^2 + Q^2}} \right) \angle \arctg\left(\frac{Q}{P}\right) \quad (\text{C.4})$$

Onde:

V = tensão na carga (pode tomar-se a nominal da rede)

P e Q : potências ativa e reativa da carga.

B. Matrizes de Rede

Como os sistemas de potência reais são compostos por um número muito grande de elementos, barras, linhas de transmissão, transformadores, etc, é necessário sistematizar o inter-relacionamento entre as grandezas elétricas do sistema de potência.

A base dessa sistematização é a representação conveniente do sistema de potência, que será feito usando-se a matriz de admitâncias nodais (matriz Y_{bus}) ou

a sua inversa, a matriz de impedâncias nodais (matriz Z_{bus}).

1) *Matriz de Admitâncias Nodais (Y_{bus}):*

A matriz de admitâncias nodais representa as ligações entre os elementos do sistema de potência e relaciona as correntes que entram em cada barra com as tensões medidas em cada barra em relação a alguma referência.

A relação entre tensão e corrente numa dada rede pode ser dada pelas seguintes equações:

$$[I] = [Y_{bus}][V] \quad (C.5)$$

$$[V] = [Z_{bus}][I] \quad (C.6)$$

Onde:

$[I]$ = vetor das correntes injetadas nas barras

$[V]$ = vetor das tensões nas barras

2) *Regras de Formação da Matriz de Admitâncias Nodais:* São as seguintes para redes sem mútuas.

I - Os elementos da diagonal principal correspondem à soma de todas as admitâncias ligadas à barra correspondente à linha da matriz Y_{bus} .

II - Cada elemento fora da diagonal principal corresponde ao negativo da soma de todas as admitâncias ligadas entre as barras correspondentes à linha e coluna da matriz Y_{bus} .

Ou seja: $Y_{bus\ ij} = \sum (\text{admitâncias entre } i \text{ e } j)$

C. Cálculos de Curto Circuito

O equacionamento para a obtenção das correntes de defeito é baseado nas seguintes hipóteses:

- 1) A rede é balanceada.
- 2) Geradores têm tensão equilibrada e só apresentam tensão de sequência direta.

Dessa forma, as correntes e tensões de falta podem ser obtidas resolvendo-se circuitos equivalentes formados por ligações convenientes das redes de sequência, que consistem em equivalentes de Thévenin vistos pela barra na qual está sendo equacionado o defeito.

Nos próximos itens serão mostrados os circuitos equivalentes para o cálculo dos quatro tipos de curto e as fórmulas para as correntes de curto circuito.

1) Curto Circuito Trifásico:

Figura C.8 – Circuito equivalente para o curto circuito trifásico.

Corrente de curto circuito trifásico:

$$I_{cc3f} = \frac{V_{th1}}{Z_{th1} + Z_p} \quad (C.7)$$

2) Curto Circuito Fase-Terra:

Figura C.9 – Circuito equivalente para o curto circuito fase-terra.

Corrente de curto-circuito fase-terra:

$$I_{CC}^{\phi-T} = \frac{3V_{th}^1}{2Z_{th}^1 + Z_{th}^0 + 3Z_g} \quad (C.8)$$

3) Curto-Circuito Dupla-Fase:

Figura C.10 – Circuito equivalente para o curto circuito dupla-fase.

Corrente de curto-circuito dupla-fase:

$$I_{CC}^{2\phi} = (\alpha^2 - \alpha) \frac{V_{th}^1}{2Z_{th}^1 + Z_F} \quad (C.9)$$

4) Curto-Circuito Dupla-Fase-Terra:

Figura C.11 - Circuito equivalente para o curto circuito dupla-fase-terra.

Corrente de curto-circuito dupla-fase-terra:

$$I_{CC}^{2\phi-T} = I_F^b = I_F^0 + \alpha^2 I_F^1 + \alpha I_F^2 \quad (C.10)$$

Onde:

$$I_F^1 = \frac{V_{th}^1}{Z_{th}^1 + Z_F + \left[(Z_F + 3Z_g + Z_{th}^0) // (Z_F + Z_{th}^1) \right]} \quad (C.11)$$

$$I_F^2 = -I_F^1 \frac{Z_F + 3Z_g + Z_{th}^0}{2Z_F + 3Z_g + Z_{th}^0 + Z_{th}^1} \quad (C.12)$$

$$I_F^2 = -I_F^1 \frac{Z_F + Z_{th}^1}{2Z_F + 3Z_g + Z_{th}^0 + Z_{th}^1} \quad (C.13)$$

ANEXOS

Anexo A: Código fonte seqMDIM.cpp

Programa em C++ para cálculo de determinante de matrizes densas de ordem n utilizando a estratégia MDIM: Modelo sequencial.

```

/*
 * File:    seqMDIM.cpp
 * Author:  Marlim P. Menezes
 * Programa: Doutorado em Ciências - Engenharia Elétrica
 * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
 * Objetivo: Calcular o determinante de matrizes densas de modo sequencial,
 * para Estudo de Caso em Programação Paralela de Alto Desempenho.
 * Created on 02 de Abril de 2013, 10:06
 */

#include <iostream>
#include <cstring>
#include <cstdlib> // Gerador de números randômicos.
#include <iomanip>
#include <cstdio>
#include <ctime>    // Para a semente do gerador randômico.
#include <cmath>

using namespace std;

// Declaração de constantes...
const int INTEGER = 0;
const int REAL    = 1;

// Declaração da estrutura para matriz...
struct TMatrix {
    int    lin;      // Linhas na matriz.
    int    col;      // Colunas na matriz.
    double *mat;     // pseudo-Matriz na forma linear.
};

// Funções protótipo...
//*****
void    randomMatrixGen(double, double, int, TMatrix *);
double randomNumberGen(double, double);
double seqMatrixDetMDI(TMatrix *); // MDI
double seqMatrixDetMDIM(TMatrix *); // MDI Melhorado
int    trataPivoZero(int, int, double *);
void    display2DMatrix(TMatrix *M);
void    save2DMatrix(const char fileName[], TMatrix *M);

/*
 * Programa principal
 */
int main() {
    // Texto de tela
    cout << "Estudo de Caso para calculo de determinante de matrizes densas sequencialmente "
    << endl;
    cout << "usando o algoritmo MDIM (MDI Melhorado) para analise de desempenho em Computacao
    Paralela." << endl;
    cout << "Por: Marlim P. Menezes - Doutorado em Ciencias - Engenharia Eletrica - POLI-
    USP" << endl;
    cout << "As matrizes utilizadas sao inteiras ou reais de ordem maxima 6000." << endl;
    cout << "*****" << endl;

    // Inicia o gerador randômico...
    srand(time(NULL)); // Semente para iniciar o gerador randômico.

```

```

// Declaração da matriz
TMatrix *A;

// Alocação de memória para a estrutura da matriz.
A = new TMatrix[sizeof(TMatrix)];
A->lin = A->col = 6000; // Dimensão da matriz corrente...lin x col

// Alocação de memória para a matriz propriamente dita.
A->mat = new (nothrow) double[A->lin * A->col];

// Verifica se alocou memória para a matriz A.
if(A->mat == NULL) {
    perror("\n\nERRO: Nao foi possivel alocar memoria para a matriz!\n");
    perror("Tente novamente com uma matriz menor.\n");
    perror("Programa interrompido.\n\n");
    exit(1);
}

// Gera a matriz A, randomicamente
int tipoDado = REAL;
randomMatrixGen(-30.0, +30.0, tipoDado, A); // Cria a matriz no intervalo [-30,30]

// Prepara para salvar a matriz em formato 2D no disco.
char matrixFileName[80]

// Salva a matriz A em formato 2D no disco.
strcpy(matrixFileName, "m2DA");
sprintf(matrixFileName, "%s%d.txt", matrixFileName, A->lin);
save2DMatrix(matrixFileName, A);

double det;
det = seqMatrixDetMDIM(A);
cout << "det = A = " << det << "    (calculado)" << endl;

return 0;
}

//*****
// Implementação das funções protótipo...
//*****

// Gera a matriz de dimensão col*lin com números randômicos no intervalo especificado.
void randomMatrixGen(double inicio, double fim, int tipo, TMatrix *M) {
    // Assim é mais rápido do que colocar o if dentro dos laços...
    if(tipo == INTEGER)
        for(int x = 0; x < M->lin * M->col; x++)
            M->mat[x] = floor(randomNumberGen(inicio, fim)); // Entradas inteiras...
    else if(tipo == REAL)
        for(int x = 0; x < M->lin * M->col; x++)
            M->mat[x] = randomNumberGen(inicio, fim); // Entradas reais...
}
//*****

// Gera e retorna um número pseudo randômico no intervalo [inicio, fim]...
double randomNumberGen(double inicio, double fim) {
    // Troca 'inicio' de lugar com 'fim', se necessário.
    if (inicio > fim)
        swap(inicio, fim);
    // Calcula o número randômico e o retorna.
    return ((rand() / (static_cast<double>(RAND_MAX) + 1.0)) * (fim - inicio + 1) + inicio);
}
//*****

// MDI Melhorado...
// Calcula o determinante de M sequencialmente usando a técnica MDIM.
double seqMatrixDetMDIM(TMatrix *M) {
    //=====
    // Agrupa em submatrizes de ordem 2 e calcula os seus respectivos
    // determinantes em matrizes de ordem (n-1), (n-2), (n-3), ..., (2).
    //=====
    // Percorre a matriz M, calcula as submatrizes de ordem 2 e guarda em M.
    int      nLin = M->lin; // Navega nas Linhas da matriz M.
    double   detM = 1.0,    // Determinante padrão de ordem 2.

```

```

        Xii,          // pivô principal
        Xij;          // pivô secundário
int    j,            // Contorle de linhas na solução das submatrizes 2x2...
        k,            // Contorle de colunoas na solução das submatrizes 2x2...
        sinal = 0;    // Controlador do sinal do determinante calculado.

// Início do cálculo
for(int i = 0; i < nLin; i++) { // Laço mais externo
    Xii = M->mat[i * (nLin + 1)]; // pivô principal

    // Tratando pivô = 0.
    if(Xii == 0 && i != nLin) {
        sinal += trataPivoZero(i, nLin, M->mat);
        Xii = M->mat[i * (nLin + 1)];
    }
    detM *= Xii; // cálculo do determinante
    for(j = i + 1; j < nLin; ++j) { // Percorre as colunas da linha i
        Xij = M->mat[j * nLin + i] / Xii; // pivô secundário: MDIM
        // Obtendo as submatrizes de M.
        for(k = i + 1; k < nLin; ++k) { // Percorre as linhas da coluna j
            M->mat[j * nLin + k] -= M->mat[i * nLin + k] * Xij; // MDIM
        } // for k
    } // for j
} // for i

// Acerta o sinal do determinante...
if(sinal % 2 != 0)
    detM = -detM;
return (detM);
}
//*****

// Tenta trocar uma linha ou coluna de pivô = zero e, retorna true se sucesso.
// Operação: Verifica se o elemento da linha subjacente ao pivô é diferente de zero,
// se sim, então troca as linhas de lugar; caso contrário, repete o processo para a
// linha adjacente à direita do pivô. Realiza esta operações até encontrar um elemento
// diferente de zero para ser o novo pivô ou, até chegar ao fim da matriz.
int trataPivoZero(int pp, int lin, double *m) {
    int trocou = 0;
    bool trocouLin, // Informa troca de linhas.
        trocouCol; // Informa troca de colunas.
    trocouLin = trocouCol = false;
    unsigned nLin = lin; // Número de linhas na matriz.

    int lc = pp + 1; // Avança nas linhas da coluna iter.
    // Busca um elemento diferente de zero primeiro nas linhas abaixo do pivô,
    // se o elemento também for igual a zero, então verifica o elemento à direita
    // do pivô apenas invertendo as coordenadas da linha e coluna na matriz.
    while((lc < nLin) && (!trocouLin && !trocouCol)) {
        if(m[lc * nLin + pp] != 0) // Verifica linha (elemento embaixo do pivô atual)...
            trocouLin = true;
        else if(m[pp * nLin + lc] != 0) // Verifica coluna (elemento à direita do pivô atual)...
            trocouCol = true;
        else
            lc++;
    }
    // Se encontrou, troca as linhas ou colunas de lugar, uma com a outra.
    double temp;
    int x;
    if(trocouLin) { // Troca as linhas de lugar entre elas...
        for(x = pp; x < nLin; x++) { // Avança nas colunas das linhas pp e lc.
            temp = m[pp * nLin + x];
            m[pp * nLin + x] = m[lc * nLin + x];
            m[lc * nLin + x] = temp;
        }
        trocou = 1; // Esta operação é mais rápida do que a soma.
    }
    else if(trocouCol) { // Troca as colunas de lugar entre elas...
        for(x = pp; x < nLin; x++) { // Avança nas colunas das linhas pp e lc.
            temp = m[x * nLin + pp];
            m[x * nLin + pp] = m[x * nLin + lc];
            m[x * nLin + lc] = temp;
        }
    }
}

```

```

        trocou = 1; // Esta operação é mais rápida do que a soma.
    }
    return trocou;
}
//*****

// Mostra a matriz na tela no formato de tabela (2-D).
void display2DMatrix(TMatrix *M) {
    for(int y = 0; y < M->lin; y++) { // pseudo-linhas
        for(int x = 0; x < M->col; x++) { // pseudo-colunas
            cout << setw(6) << M->mat[y * M->lin + x] << " ";
        }
        cout << endl;
    }
}
//*****

// Grava a matriz bidimensional no disco.
void save2DMatrix(const char fileName[], TMatrix *M) {
    FILE *ptrArq;
    ptrArq = fopen(fileName, "w"); // Abre arquivo p gravação.
    if(ptrArq == NULL) { // Falhou na criação do arquivo em disco
        perror("Erro ao criar o arquivo de dados em disco.");
        perror("Os dados não serão gravados no disco.");
    }
    else {
        for(int y = 0; y < M->lin; y++) { // lines
            for(int x = 0; x < M->col; x++) { // columns
                fprintf(ptrArq, "%.15f\t", M->mat[y * M->lin + x]);
            }
            fprintf(ptrArq, "\n");
        }
        fflush(ptrArq);
        fclose(ptrArq);
    }
}
//*****

```

Anexo B: Código fonte parMDIM.cpp

Programa em C++ para cálculo de determinante de matrizes densas de ordem n utilizando a estratégia MDIM: Modelo paralelo.

```

/*
 * File:   parMDIM.cpp
 * Author: Marlim P. Menezes
 * Programa: Doutorado em Ciências - Engenharia Elétrica
 * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
 * Objetivo: Calcular o determinante de matrizes densas de modo paralelo,
 * para Estudo de Caso em Programação Paralela de Alto Desempenho.
 * Created on 02 de Abril de 2013, 10:06
 */
#include <iostream>
#include <cstring>
#include <cstdlib> // Gerador de números randômicos.
#include <iomanip>
#include <cstdio>
#include <ctime>    // Para a semente do gerador randômico.
#include <cmath>
#include <omp.h>

using namespace std;

// Declaração de constantes...
const int INTEGER = 0;
const int REAL    = 1;

// Declaração da estrutura para matriz...
struct TMatrix {
    int    lin;      // Linhas na matriz.
    int    col;      // Colunas na matriz.
    double *mat;     // pseudo-Matriz na forma linear.
};

// Funções protótipo...
//*****
void  randomMatrixGenOMP(double, double, int, TMatrix *, int);
double randomNumberGen(double, double);
double parMatrixDetMDIMOMP(int, TMatrix *);
int    swapPivotZero(int, int, double *);
void    display2DMatrix(TMatrix *M);
void    save2DMatrix(const char fileName[], TMatrix *M);

/*
 * Programa principal
 */
int main() {
    // Texto de tela
    cout << "Estudo de Caso para cálculo de determinante de matrizes densas paralelamente " <<
endl;
    cout << "usando o algoritmo IIDM (Improved IDM) para analise de desempenho em Computacao
Paralela." << endl;
    cout << "Por: Marlim P. Menezes - Doutorado em Ciencias - Engenharia Eletrica - POLI-
USP" << endl;
    cout << "As matrizes utilizadas sao inteiras ou reais de ordem maxima 6000." << endl;
    cout << "*****" << endl;

    // Inicia o gerador randômico...
    srand(time(NULL)); // Semente para iniciar o gerador randômico.

    // Declaração da matriz
    TMatrix *A;

    // Alocação de memória para a estrutura da matriz.
    A = new TMatrix[sizeof(TMatrix)];
    A->lin = A->col = 6000; // Dimensão da matriz corrente...lin x col

    // Alocação de memória para a a matriz propriamente dita.
    A->mat = new (nothrow) double[A->lin * A->col];

```

```

// Verifica se alocou memória para a matriz A.
if(A->mat == NULL) {
    perror("\n\nERRO: Nao foi possivel alocar memoria para a matriz!\n");
    perror("Tente novamente com uma matriz menor.\n");
    perror("Programa interrompido.\n\n");
    exit(1);
}

// Gera a matriz A, randomicamente
int tipoDado = REAL;
int numProcs = 8; // Número de processadores sendo usados...
randomMatrixGenOMP(-30.0, +30.0, tipoDado, A, numProcs);

// Prepara para salvar as matrizes em formato 2D no disco.
char matrixFileName[80];

// Calcula o determinante em paralelo com numProcs processadores...
double det;
det = parMatrixDetMDIMOMP(numProcs, A);

// Salva a matriz A em formato 2D no disco.
strcpy(matrixFileName, "m2DA");
sprintf(matrixFileName, "%s%d.txt", matrixFileName, A->lin);
save2DMatrix(matrixFileName, A);

return 0;
}

//*****
// Implementação das funções protótipo...
//*****

// Gera a matriz de dimensão col*lin com números randômicos no intervalo especificado.
void randomMatrixGenOMP(double inicio, double fim, int tipo, TMatrix *M, int numCPUs) {
    // Assim é mais rápido do que colocar o if dentro dos laços...
    //omp_set_num_threads(numCPUs);
    if(tipo == INTEGER) {
#pragma omp parallel
        #pragma omp for
        for(int x = 0; x < M->lin * M->col; x++)
            M->mat[x] = floor(randomNumberGen(inicio, fim)); // Entradas inteiras...
    }
    else if(tipo == REAL) {
#pragma omp parallel
        #pragma omp for
        for(int x = 0; x < M->lin * M->col; x++)
            M->mat[x] = randomNumberGen(inicio, fim); // Entradas reais...
    }
}

//*****

// Gera e retorna um número pseudo randômico no intervalo [inicio, fim]...
double randomNumberGen(double inicio, double fim) {
    // Troca 'inicio' de lugar com 'fim', se necessário.
    if (inicio > fim)
        swap(inicio, fim);

    // Calcula o número randômico e o retorna.
    return ((rand() / (static_cast<double>(RAND_MAX) + 1.0))*(fim - inicio + 1) + inicio);
}

//*****

// MDI Melhorado Paralelo
// Calcula o determinante de M paralelamente usando a estratégia MDIM.
double parMatrixDetMDIMOMP(int numCPUs, TMatrix *M) {
    //=====
    // Agrupa em submatrizes de ordem 2 e calcula os seus respectivos
    // determinantes em matrizes de ordem (n-1), (n-2), (n-3), ..., (2).
    //=====
    // Percorre a matriz M, calcula as submatrizes de ordem 2 e guarda em M.
    int nLin = M->lin; // Navega nas Linhas da matriz M.
    double detM = 1.0; // Determinante padrão de ordem 2.

```

```

int      i,          // Contorle de linhas na solução das submatrizes 2x2...
        j,          // Contorle de colunas na solução das submatrizes 2x2...
sinal = 0, // Controlador do sinal do determinante calculado.
iter;      // Controle do laço mais externo do cálculo do determinante.

omp_set_num_threads(numCPUs);

// Início do cálculo
#pragma omp parallel shared(nLin,sinal,detM) private(iter,i,j)
{
    int n_threads, id_thread, inicio_i, fim_i, size_lin, iter_nLin, i_nLin, i_iter;
    double Xii, Xij;
    n_threads = omp_get_num_threads();
    id_thread = omp_get_thread_num();
    for(iter = 0; iter < nLin; iter++) {
        Xii = M->mat[iter * (nLin+1)];
        // Tratando pivô = 0.
        if (id_thread == 0) {
            if(Xii == 0 && iter != nLin) {
                sinal += swapPivotZero(iter, nLin, M->mat);
                Xii = M->mat[iter * (nLin + 1)];
            }
            detM *= Xii;
        }
        size_lin = (nLin - iter) / n_threads;
        if ((nLin - iter) % n_threads != 0)
            size_lin++;
        inicio_i = iter + 1 + (id_thread * size_lin);
        fim_i = inicio_i + size_lin;
        iter_nLin = iter * nLin;
        for(i = inicio_i; ((i < fim_i) && (i < nLin)); i++) { // Navega nas linhas...
            // Obtendo as submatrizes de M.
            i_nLin = i * nLin;
            i_iter = i_nLin + iter;
            Xij = M->mat[i_iter] / Xii; // DMIM
            for(j = iter + 1; j < nLin; ++j) { // Navega nas colunas...
                M->mat[i_nLin + j] -= M->mat[iter_nLin + j] * Xij; // MDIM
            } // for j
        } // for i
        #pragma omp barrier
    } // for iter
} // omp parallel shared
// Acerta o sinal do determinante...
if(sinal % 2 != 0)
    detM = -detM;
return (detM);
}
//*****

// Tenta trocar uma linha ou coluna de pivô = zero e, retorna true se sucesso.
// Operação: Verifica se o elemento da linha subjacente ao pivô é diferente de zero,
// se sim, então troca as linhas de lugar; caso contrário, repete o processo para a
// linha adjacente à direita do pivô. Realiza esta operações até encontrar um elemento
// diferente de zero para ser o novo pivô ou, até chegar ao fim da matriz.
int swapPivotZero(int pp, int lin, double *m) {
    int trocou = 0;
    bool trocouLin, // Informa troca de linhas.
        trocouCol; // Informa troca de colunas.
    trocouLin = trocouCol = false;
    unsigned nLin = lin; // Número de linhas na matriz.

    int lc = pp + 1; // Avança nas linhas da coluna iter.
    // Busca um elemento diferente de zero primeiro nas linhas abaixo do pivô,
    // se o elemento também for igual a zero, então verifica o elemento à direita
    // do pivô apenas invertendo as coordenadas da linha e coluna na matriz.
    while((lc < nLin) && (!trocouLin && !trocouCol)) {
        if(m[lc * nLin + pp] != 0) // Verifica linha (elemento embaixo do pivô atual)...
            trocouLin = true;
        else if(m[pp * nLin + lc] != 0) // Verifica coluna (elemento à direita do pivô atual)...
            trocouCol = true;
        else
            lc++;
    }
}

```

```

// Se encontrou, troca as linhas ou colunas de lugar, uma com a outra.
double temp;
int x;
if(trocouLin) { // Troca as linhas de lugar entre elas...
    for(x = pp; x < nLin; x++) { // Avança nas colunas das linhas pp e lc.
        temp = m[pp * nLin + x];
        m[pp * nLin + x] = m[lc * nLin + x];
        m[lc * nLin + x] = temp;
    }
    trocou = 1; // Esta operação é mais rápida do que a soma.
}
else if(trocouCol) { // Troca as colunas de lugar entre elas...
    for(x = pp; x < nLin; x++) { // Avança nas colunas das linhas pp e lc.
        temp = m[x * nLin + pp];
        m[x * nLin + pp] = m[x * nLin + lc];
        m[x * nLin + lc] = temp;
    }
    trocou = 1; // Ativa o sinalizador.
}
return trocou;
}
//*****

// Mostra a matriz na tela.
void display2DMatrix(TMMatrix *M) {
    for(int y = 0; y < M->lin; y++) {
        for(int x = 0; x < M->col; x++) {
            cout << setw(6) << M->mat[y * M->lin + x] << " ";
        }
        cout << endl;
    }
}
//*****

// Grava a matriz bidimensional no disco.
void save2DMatrix(const char fileName[], TMMatrix *M) {
    FILE *ptrArq;
    ptrArq = fopen(fileName, "w"); // Abre arquivo p gravação.
    if(ptrArq == NULL) { // Falhou na criação do arquivo em disco
        perror("Erro ao criar o arquivo de dados em disco.");
        perror("Os dados não serão gravados no disco.");
    }
    else {
        for(int y = 0; y < M->lin; y++) {
            for(int x = 0; x < M->col; x++) {
                fprintf(ptrArq, "%.15f\t", M->mat[y * M->lin + x]);
            }
            fprintf(ptrArq, "\n");
        }
        fflush(ptrArq);
        fclose(ptrArq);
    }
}
//*****

```

Anexo C: Código fonte seqMatMul.cpp

Programa em C++ para Multiplicação de matrizes de ordem mXn: Modelo sequencial.

```

/*
 * File:    seqMatMul.cpp
 * Author:  Marlim P. Menezes
 * Programa: Doutorado em Ciências - Engenharia Elétrica
 * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
 * Objetivo: Efetuar a multiplicação de duas matrizes densas de modo sequencial,
 * para Estudo de Caso em Programação Paralela de Alto Desempenho.
 * Created on 29 de Março de 2013, 20:56
 */

#include <iostream>
#include <complex>
#include <cstring>
#include <cstdlib> // Gerador de números randômicos.
#include <iomanip>
#include <cstdio>
#include <ctime>    // Para a semente do gerador randômico.
#include <cmath>

using namespace std;

// Definição das constantes...
const int INTEGER = 0;
const int REAL    = 1;

// Declaração da estrutura para matriz...
struct TMatrix {
    int    lin;    // Linhas na matriz.
    int    col;    // Colunas na matriz.
    double *mat;   // Pseudo-Matriz linear.
};

// Funções protótipo...
//*****
void seqRandomMatrixGen(double, double, int, TMatrix *);
void seqMatrixMul(TMatrix *, TMatrix *, TMatrix *);
double randomNumberGen(double, double);
void display2DMatrix(TMatrix *M);
void save2DMatrix(const char fileName[], TMatrix *M);

/*
 * Programa principal
 */
int main() {
    // Texto de tela
    cout << "Estudo de Caso para multiplicacao de matrizes densas em SEQUENCIAL " << endl;
    cout << "para analise de desempenho em Computacao Paralela." << endl;
    cout << "Por: Marlim P. Menezes - Doutorado em Ciencias - Engenharia Eletrica - POLI-USP" << endl;
    cout << "As matrizes utilizadas sao inteiras ou reais de ordem 6000." << endl;
    cout << "*****" << endl;

    // Inicia o gerador randômico...
    srand(time(NULL)); // Semente para iniciar o gerador randômico.

    // Declaração das matrizes
    TMatrix *A;
    TMatrix *B;
    TMatrix *C;

    // Alocação de memória para as três estruturas das matrizes.
    A = new TMatrix[sizeof(TMatrix)];
    B = new TMatrix[sizeof(TMatrix)];
    C = new TMatrix[sizeof(TMatrix)];

    // Definição da ordem das matrizes...
    A->lin = A->col = B->lin = B->col = 6000;

```

```

C->lin = A->lin;
C->col = B->col;

// Alocação de memória para as três as matrizes.
A->mat = new double[A->lin*A->col];
B->mat = new double[B->lin*B->col];
C->mat = new double[C->lin*C->col];

// Verifica se alocou memória para a matriz A.
if(A->mat == NULL) {
    perror("\n\nERRO: Nao foi possivel alocar memoria para a matriz!\n");
    perror("Tente novamente com uma matriz menor.\n");
    perror("Programa interrompido.\n\n");
    exit(1);
}
// Verifica se alocou memória para a matriz B.
if(B->mat == NULL) {
    perror("\n\nERRO: Nao foi possivel alocar memoria para a matriz!\n");
    perror("Tente novamente com uma matriz menor.\n");
    perror("Programa interrompido.\n\n");
    exit(1);
}
// Verifica se alocou memória para a matriz C.
if(C->mat == NULL) {
    perror("\n\nERRO: Nao foi possivel alocar memoria para a matriz!\n");
    perror("Tente novamente com uma matriz menor.\n");
    perror("Programa interrompido.\n\n");
    exit(1);
}

// Gera as matrizes A e B randomicamente, em seguida efetua a multiplicação...
int tipoDado = REAL; // Tipo de dado de entrada nas matrizes...
seqRandomMatrixGen(-30.0, +30.0, tipoDado, A); // Cria a matriz A no intervalo [-30, 30].
seqRandomMatrixGen(-30.0, +30.0, tipoDado, B); // Cria a matriz B no intervalo [-30, 30].
seqMatrixMul(A, B, C); // Multiplica as matrizes A e B e guarda os resultados em C.

// Prepara para salvar as matrizes em formato 2D no disco.
char matrixFileName[80];

// Grava a matriz A em formato 2D no disco.
strcpy(matrixFileName, "m2DA");
sprintf(matrixFileName, "%s%dxd.txt", matrixFileName, A->lin, A->col);
save2DMatrix(matrixFileName, A);

// Grava a matriz B em formato 2D no disco.
strcpy(matrixFileName, "m2DB");
sprintf(matrixFileName, "%s%dxd.txt", matrixFileName, B->lin, B->col);
save2DMatrix(matrixFileName, B);

// Grava a matriz C em formato 2D no disco.
strcpy(matrixFileName, "m2DC");
sprintf(matrixFileName, "%s%d.txt", matrixFileName, C->lin);
save2DMatrix(matrixFileName, C);

// Libera a memoria usada pelas matrizes para o SO
delete[] A->mat;
delete[] B->mat;
delete[] C->mat;

return 0;
}

//*****
// Implementação das funções protótipo...
//*****

// Gera a matriz de dimensão col*lin com números randômicos no intervalo especificado.
void seqRandomMatrixGen(double inicio, double fim, int tipo, TMatrix *M) {
    // Assim é mais rápido do que colocar o if dentro dos laços...
    if(tipo == INTEGER) {
        for(unsigned x = 0; x < M->lin*M->col; x++) {
            M->mat[x] = floor(randomNumberGen(inicio, fim)); // Entradas inteiras...
        }
    }
}

```

```

    }
    else if(tipo == REAL) {
        for(unsigned x = 0; x < M->lin*M->col; x++) {
            M->mat[x] = randomNumberGen(inicio, fim); // Entradas reais...
        }
    }
}
//*****

// Retorna P = A * B
void seqMatrixMul(TMatrix *A, TMatrix *B, TMatrix *P) {
    for(unsigned y = 0; y < A->lin; ++y) {
        for(unsigned x = 0; x < A->col; ++x) {
            double sum = 0;
            for(unsigned k = 0; k < B->lin; ++k) {
                double a = A->mat[y * A->lin + k];
                double b = B->mat[k * B->col + x];
                sum += a * b;
            }
            P->mat[y * A->lin + x] = sum;
        }
    }
}
//*****

// Gera e retorna um número pseudo randômico no intervalo [inicio, fim]...
double randomNumberGen(double inicio, double fim) {
    // Troca 'inicio' de lugar com 'fim', se necessário.
    if (inicio > fim)
        swap(inicio, fim);
    // Calcula o número randômico e o retorna.
    return ((rand()/(static_cast<double>(RAND_MAX)+1.0))*(fim-inicio+1)+inicio);
}
//*****

// Mostra a matriz na tela.
void display2DMatrix(TMatrix *M) {
    for(unsigned y = 0; y < M->lin; y++) {
        for(unsigned x = 0; x < M->col; x++) {
            cout << setw(6) << M->mat[y*M->lin+x] << " ";
        }
        cout << endl;
    }
}
//*****

// Grava a matriz bidimensional no disco.
void save2DMatrix(const char fileName[], TMatrix *M) {
    FILE *ptrArq;
    ptrArq = fopen(fileName, "w"); // Abre arquivo p gravação.
    if(ptrArq == NULL) { // Falhou na criação do arquivo em disco
        perror("Erro ao criar o arquivo de dados em disco.");
        perror("Os dados não serão gravados no disco.");
    }
    else {
        for(unsigned y = 0; y < M->lin; y++) {
            for(unsigned x = 0; x < M->col; x++) {
                fprintf(ptrArq, "%.15f\t", M->mat[y*M->lin+x]);
            }
            fprintf(ptrArq, "\n");
        }
        fflush(ptrArq);
        fclose(ptrArq);
    }
}

```

Anexo D: Código fonte parMatMul.cpp

Programa em C++ para Multiplicação de matrizes de ordem mXn: Modelo paralelo.

```

/*
 * File:   parMatMul.cpp
 * Author: Marlim P. Menezes
 * Programa: Doutorado em Ciências - Engenharia Elétrica
 * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
 * Objetivo: Efetuar a multiplicação de duas matrizes densas de modo sequencial,
 * para Estudo de Caso em Programação Paralela de Alto Desempenho.
 * Created on 30 de Março de 2013, 09:56
 */

#include <iostream>
#include <complex>
#include <cstring>
#include <cstdlib> // Gerador de números randômicos.
#include <iomanip>
#include <cstdio>
#include <ctime>    // Para a semente do gerador randômico.
#include <cmath>
#include <omp.h>

using namespace std;

// Declaração de constantes...
const int INTEGER = 0;
const int REAL    = 1;

// Declara da estrutura para matriz...
struct TMatrix {
    int    lin;      // Linhas na matriz.
    int    col;      // Colunas na matriz.
    double *mat;     // pseudo-matriz linear.
};

// Funções protótipo...
//*****
void  parRandomMatrixGenOMP_A(double, double, int, TMatrix *, int);
void  parRandomMatrixGenOMP_B(double, double, int, TMatrix *, int);
void  parMatrixMulOMP(TMatrix *, TMatrix *, TMatrix *, int);
double randomNumberGen(double, double);
void  display2DMatrix(TMatrix *M);
void  save2DMatrix(const char fileName[], TMatrix *M);

/*
 * Programa principal
 */
int main() {
    // Texto de tela
    cout << "Estudo de Caso para multiplicacao de matrizes densas em PARALELO " << endl;
    cout << "para analise de desempenho em Computacao Paralela." << endl;
    cout << "Por: Marlim P. Menezes - Doutorado em Ciencias - Engenharia Eletrica - POLI-USP" << endl;
    cout << "As matrizes utilizadas sao inteiras ou reais de ordem 6000." << endl;
    cout << "*****" << endl;

    // Inicia o gerador randômico...
    srand(time(NULL)); // Semente para iniciar o gerador randômico.

    // Declaração das matrizes
    TMatrix *A;
    TMatrix *B;
    TMatrix *C;

    // Alocação de memória para as três estruturas das matrizes.
    A = new TMatrix[sizeof(TMatrix)];
    B = new TMatrix[sizeof(TMatrix)];
    C = new TMatrix[sizeof(TMatrix)];

```

```

A->lin = A->col = B->lin = B->col = C->lin = C->col = 6000;

// Alocação de memória para as três as matrizes propriamente ditas.
A->mat = new (nothrow) double[A->lin*A->col];
B->mat = new (nothrow) double[B->lin*B->col];
C->mat = new (nothrow) double[C->lin*C->col];

// Verifica se alocou memória para a matriz A.
if(A->mat == NULL) {
    perror("\n\nERRO: Nao foi possivel alocar memoria para a matriz!\n");
    perror("Tente novamente com uma matriz menor.\n");
    perror("Programa interrompido.\n\n");
    exit(1);
}
// Verifica se alocou memória para a matriz B.
if(B->mat == NULL) {
    perror("\n\nERRO: Nao foi possivel alocar memoria para a matriz!\n");
    perror("Tente novamente com uma matriz menor.\n");
    perror("Programa interrompido.\n\n");
    exit(1);
}
// Verifica se alocou memória para a matriz C.
if(C->mat == NULL) {
    perror("\n\nERRO: Nao foi possivel alocar memoria para a matriz!\n");
    perror("Tente novamente com uma matriz menor.\n");
    perror("Programa interrompido.\n\n");
    exit(1);
}

// Inicializa as matrizes randomicamente, em paralelo, e as multiplica em paralelo
int tipoDado = REAL; // Tipo de dados de entrada nas matrizes...
#pragma omp parallel sections // Região paralela com três seções
{
    #pragma omp section // tarefa 1: inicializa a matriz A
    {
        parRandomMatrixGenOMP_A(-30.0, +30.0, tipoDado, A, 2); // Usa duas CPUs
    }
    #pragma omp section
    {
        parRandomMatrixGenOMP_B(-30.0, +30.0, tipoDado, B, 2); // Usa duas CPUs
    }
}
    parMatrixMulOMP(A, B, C, 4); // Usa quatro CPUs

// Prepara para salvar as matrizes em formato 2D no disco.
char matrixFileName[80];

// Grava a matriz A em formato 2D no disco.
strcpy(matrixFileName, "m2DA");
sprintf(matrixFileName, "%s%d.txt", matrixFileName, A->lin, A->col);
save2DMatrix(matrixFileName, A);

// Grava a matriz B em formato 2D no disco.
strcpy(matrixFileName, "m2DB");
sprintf(matrixFileName, "%s%d.txt", matrixFileName, B->lin, B->col);
save2DMatrix(matrixFileName, B);

// Grava a matriz C em formato 2D no disco.
strcpy(matrixFileName, "m2DC");
sprintf(matrixFileName, "%s%d.txt", matrixFileName, C->lin);
save2DMatrix(matrixFileName, C);

// Libera a memoria usada pelas matrizes para o SO
delete[] A->mat;
delete[] B->mat;
delete[] C->mat;

return 0;
}

//*****
// Implementação das funções protótipo...
//*****

```

```

// Gera a matriz de dimensão col*lin com números randômicos no intervalo especificado.
void parRandomMatrixGenOMP_A(double inicio, double fim, int tipo, TMatrix *A, int numCPUs) {
    // Assim é mais rápido do que colocar o if dentro dos laços...
    omp_set_num_threads(numCPUs);
    if(tipo == INTEGER) {
        #pragma omp parallel // Cria região paralela
        #pragma omp for // Paraleliza o comando for
        for(unsigned i = 0; i < A->lin * A->col; i++)
            A->mat[i] = floor((rand()/(static_cast<double>(RAND_MAX)+1.0))*(fim-inicio+1)+inicio);
    }
    else if(tipo == REAL) {
        #pragma omp parallel
        #pragma omp for
        for(unsigned i = 0; i < A->lin * A->col; i++)
            A->mat[i] = (rand()/(static_cast<double>(RAND_MAX)+1.0))*(fim-inicio+1)+inicio;
    }
}
//*****

// Gera a matriz de dimensão col*lin com números randômicos no intervalo especificado.
void parRandomMatrixGenOMP_B(double inicio, double fim, int tipo, TMatrix *B, int numCPUs) {
    // Assim é mais rápido do que colocar o if dentro dos laços...
    omp_set_num_threads(numCPUs);
    if(tipo == INTEGER) {
        #pragma omp parallel // Cria região paralela
        #pragma omp for // Paraleliza o comando for
        for(unsigned i = 0; i < B->lin; i++) // linhas
            for(unsigned j = 0; j < B->col; j++) // colunas
                B->mat[j*B->col+i] = floor((rand()/(static_cast<double>(RAND_MAX)+1.0))*(fim-
inicio+1)+inicio);
    }
    else if(tipo == REAL) {
        #pragma omp parallel
        #pragma omp for
        for(unsigned i = 0; i < B->lin; i++) // linhas
            for(unsigned j = 0; j < B->col; j++) // colunas
                B->mat[j*B->col+i] = (rand()/(static_cast<double>(RAND_MAX)+1.0))*(fim-
inicio+1)+inicio;
    }
}
//*****

// Retorna P = A * B
void parMatrixMulOMP(TMatrix *A, TMatrix *B, TMatrix *P, int numCPUs) {
    omp_set_num_threads(numCPUs);
    #pragma omp parallel // Cria região paralela
    for(unsigned y = 0; y < A->lin; ++y) {
        #pragma omp for // Paraleliza o comando for
        for(unsigned x = 0; x < A->col; ++x) {
            double sum = 0;
            for(unsigned k = 0; k < B->lin; ++k) {
                double a = A->mat[y * A->lin + k];
                double b = B->mat[k * B->col + x];
                sum += a * b;
            }
            P->mat[y * A->lin + x] = sum;
        }
    }
}
//*****

// Gera e retorna um número pseudo randômico no intervalo [inicio, fim]...
double randomNumberGen(double inicio, double fim) {
    // Troca 'inicio' de lugar com 'fim', se necessário.
    if (inicio > fim)
        swap(inicio, fim);
    // Calcula o número randômico e o retorna.
    return ((rand()/(static_cast<double>(RAND_MAX)+1.0))*(fim-inicio+1)+inicio);
}
//*****

// Mostra a matriz na tela.

```

```

void display2DMatrix(TMatrix *M) {
    for(int y = 0; y < M->lin; y++) {
        for(int x = 0; x < M->col; x++) {
            cout << setw(6) << M->mat[y*M->lin+x] << " ";
        }
        cout << endl;
    }
}
//*****

// Grava a matriz bidimensional no disco.
void save2DMatrix(const char fileName[], TMatrix *M) {
    FILE *ptrArq;
    ptrArq = fopen(fileName, "w"); // Abre arquivo p gravação.
    if(ptrArq == NULL) { // Falhou na criação do arquivo em disco
        perror("Erro ao criar o arquivo de dados em disco.");
        perror("Os dados não serão gravados no disco.");
    }
    else {
        for(int y = 0; y < M->lin; y++) {
            for(int x = 0; x < M->col; x++) {
                fprintf(ptrArq, "%.15f\t", M->mat[y*M->lin+x]);
            }
            fprintf(ptrArq, "\n");
        }
        fflush(ptrArq);
        fclose(ptrArq);
    }
}

```

Anexo E: Código fonte seqShortCircuit.cpp

Programa em C++ para o Estudo de Caso 2 – Cálculo de Curto Circuito em Sistemas de Potência: Modelo sequencial.
--

```

/*
 * File:    seqShortCircuit.cpp
 * Author:  Marlim P. Menezes
 * Progama: Doutorado em Ciências - Engenharia Elétrica
 * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
 * Objetivo: Efetuar o cálculo de curto circuito em linhas de transmissão de modo
 * sequencial, para Estudo de Caso em Programação Paralela de Alto Desempenho.
 * Created on 29 de Março de 2013, 20:56
 */

// Biblioteca Padrão...
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cstring>
#include <iomanip>
#include <vector>
#include <complex>
#include <cmath>

using namespace std;

// Declaração da estrutura para as matrizes...
struct TMatrix {
    unsigned    lin; // Linhas na matriz.
    unsigned    col; // Colunas na matriz.
    complex<double> *mat; // Matriz.
};

// Declaração da estrutura para as barras...
struct TBarra {
    unsigned num; // Código de identificação da barra: (0..99999)
    string nome; // Nome da barra
    double vBase; // Tensão base da barra
};

// Declaração da estrutura para as ligações...
struct TLigacao {
    unsigned de;
    unsigned para;
    int nc;
    string estado;
    double r1, x1;
    double r0, x0;
};

// Declaração de constantes...
const complex<double> zero = 0.0;
const complex<double> um = 1.0;
const double sBase = 100.0; // MVA.
const unsigned nummax = 99999; // Maior identificador de barra.
const double pouco = 1e-10;
const double pi = 3.141592653589793238462643383279; // ...

// Declaração de variáveis globais...
double timeSeq = 0.0; // Tempo total de processamento sequencial.

// Funções protótipo...
//*****
void memAllocTo2dMatrix(TMatrix *M);
void guessSESize(unsigned &nBar, unsigned &nLig, const string arq,
    const string str1, const string str2, const string nm);
void loadSEData(TBarra *barras, TLigacao *ligacoes, unsigned *endBar,
    const string arq, const string str1, const string str2, const string nm);
double str2double(const double n);
void guessSCSize(unsigned &nBarCC, const string arq);

```

```

void loadSCDataBusses(unsigned *barCC, const string arq);
void createYBusses(TMatrix *Bus1, TMatrix *Bus0, const unsigned *endBar,
    const TLigacao *Lig, const unsigned numLig);
void createIdentityMatrix(TMatrix *I);
void triangMatrix(TMatrix *M, TMatrix *I);
void backSubstitution(TMatrix *yBus, TMatrix *iBus, TMatrix *zBus, const unsigned col);
void displayVector(unsigned *vec, const unsigned size);
void display2dMatrix(TMatrix *M);
void displayBusses(TBarra *B, const unsigned size);
void displayCircuits(TLigacao *C, const unsigned size);
void save2dMatrixToFile(const string arq, TMatrix *M);
void saveReport(const string arq, const string reportLine);
complex<double> kX(const double k, complex<double> X);
complex<double> polar(complex<double> X);
inline int sign(const double v) { return ((v > 1.0) ? 1 : -1); }

/*
 * Programa principal
 */
int main() {
    // Mensagem de inicialização.
    cout << "Estudo de caso de Calculo de Curto Circuito em Linhas de Transmissao." << endl;
    cout << "Modelo de programacao: Sequencial" << endl;
    cout << "Escola Politecnica da Universidade de Sao Paulo - PEA - 2013." << endl;
    cout << "===== " << endl;

    // Declaração das matrizes
    TMatrix *yBus0;
    TMatrix *yBus1;
    TMatrix *zBus0;
    TMatrix *zBus1;

    // Alocação de memória para as estruturas das matrizes.
    yBus0 = new TMatrix[sizeof(TMatrix)]; // Matriz quadrada (n x n).
    yBus1 = new TMatrix[sizeof(TMatrix)]; // Matriz quadrada (n x n).
    zBus0 = new TMatrix[sizeof(TMatrix)]; // Matriz quadrada (n x n).
    zBus1 = new TMatrix[sizeof(TMatrix)]; // Matriz quadrada (n x n).

    // Obtenção de dados iniciais do SE.
    string arqSE = "EPE_2011.ANA"; // Arquivo com 5547 barras, para estudo de caso...
    string arqBarraCC = "EPE_2011_5000.bar"; // Arquivo contendo 5000 barras em curto
    string strDBAR = "DBAR"; // Identificador da seção das barras no arquivo do SE.
    string strDCIR = "DCIR"; // Identificador da seção das ligações no arquivo do SE.
    string strLim = "99999"; // Identificador do fim da seção procurada no arquivo do SE.

    // Obtem o número de barras e de ligações do SE, para alocação dinâmica de
    // memória para as estruturas...
    unsigned numBarras = 0, numLigacoes = 0, numBarrasCC = 0;
    guessSESize(numBarras, numLigacoes, arqSE, strDBAR, strDCIR, strLim);
    guessSCSize(numBarrasCC, arqBarraCC); // Obtem o número de barras em curto...
    printf("numBarras : %d\nnumLigacoes: %d\nnumBarrasCC: %d\n", numBarras,
        numLigacoes, numBarrasCC);

    // Define as dimensões das matrizes com os valores obtidos do arquivo do SE...
    yBus1->lin = yBus0->lin = numBarras;
    yBus1->col = yBus0->col = numBarras; // Matriz expandida p comporta a Identidade.
    zBus1->lin = zBus0->col = zBus0->lin = zBus0->col = numBarras; // Matrizes quadradas.

    // Aloca memória para as matrizes.
    memAllocTo2dMatrix(yBus0); // n x n
    memAllocTo2dMatrix(yBus1); // n x n
    memAllocTo2dMatrix(zBus0); // n x n
    memAllocTo2dMatrix(zBus1); // n x n

    // Declara as estruturas para receber as barras e as ligações do SE...
    TBarra *barras; // Lista de barras do SE
    TLigacao *ligacoes; // Lista de ligações do SE
    unsigned *endBar; // Índices das barras...
    unsigned *barrasCC; // Barras em curto...

    // Aloca memória para as barras, ligações, endbar e barrasCC...
    endBar = new unsigned[nummax+1]; // Inclui a barra TERRA
    endBar[0] = 0;

```

```

barras    = new TBarra[numBarras+1]; // Inclui a barra TERRA
ligacoes = new Tligacao[numLigacoes]; // original: [4*numBarras]
barrasCC  = new unsigned[numBarrasCC];

// Carrega os dados do SE: Barras e Ligações...
cout << "Vai carregar os dados do SE: Barras e Ligacoes: \n";
loadSEData(barras, ligacoes, endBar, arqSE, strDBAR, strDCIR, strLim);
cout << "Vai carregar os dados da Barras em Curto: \n";
loadSCDataBusses(barrasCC, arqBarraCC); // Carrega a lista de barras em curto...

// Monta as matrizes yBus...
cout << "Criando as matrizes yBus0 e yBus1: \n";
createYBusses(yBus1, yBus0, endBar, ligacoes, numLigacoes);

// Libera a memória usada pelas listas de barras e ligacoes...
delete [] ligacoes;

// Salva as matrizes yBus no disco...
string arqYBus0 = "yBus0.txt";
save2dMatrixToFile(arqYBus0, yBus0);
string arqYBus1 = "yBus1.txt";
save2dMatrixToFile(arqYBus1, yBus1);

/*
 * Calcula o Curto Circuito em todas as barras solicitadas
 * =====
 */
TMatrix *iBus0, *iBus1; // Matrizes identidades para yBus0 e yBus1.
iBus0 = new TMatrix[sizeof(TMatrix)];
iBus1 = new TMatrix[sizeof(TMatrix)];
iBus0->lin = iBus0->col = iBus1->lin = iBus1->col = numBarras;
iBus0->mat = new complex<double>[numBarras * numBarras];

// Triangulariza as matrizes yBus...
iBus1->mat = new complex<double>[numBarras * numBarras];
cout << "Criando a matriz Identidade iBus0: \n";
createIdentityMatrix(iBus0);
cout << "Criando a matriz Identidade iBus1: \n";
createIdentityMatrix(iBus1);
cout << "Triangularizando a matriz yBus0: \n";
triangMatrix(yBus0, iBus0);
cout << "Triangularizando a matriz yBus1: \n";
triangMatrix(yBus1, iBus1);

// Inverte yBus0 para obter zBus0...
// Realiza esta tarefa somente para as colunas necessárias, indicadas
// na lista de barras em curto circuito...
cout << "Resolvendo o sistema para yBus0 (backSubstitution): \n";
for(unsigned bc = 0; bc < numBarrasCC; bc++) {
    backSubstitution(yBus0, iBus0, zBus0, endBar[barrasCC[bc]]); // inversão por coluna...
}

// Inverte yBus1 para obter zBus1...
// Realiza esta tarefa somente para as colunas necessárias, indicadas
// na lista de barras em curto circuito...
cout << "Resolvendo o sistema para yBus1 (backSubstitution): \n";
for(unsigned bc = 0; bc < numBarrasCC; bc++) {
    backSubstitution(yBus1, iBus1, zBus1, endBar[barrasCC[bc]]); // inversão por coluna...
}

// Salva as matrizes zBus no disco...
string arqZBus0 = "zBus0.txt";
save2dMatrixToFile(arqZBus0, zBus0);
string arqZBus1 = "zBus1.txt";
save2dMatrixToFile(arqZBus1, zBus1);

// Libera a memoria usada pelas matrizes: yBus0, yBus1, iBus0 e iBus1.
// Pois as mesmas não serão mais usadas....
delete [] yBus0->mat;
delete [] yBus1->mat;
delete [] iBus0->mat;
delete [] iBus1->mat;

```

```

/*
 * Calcula o curto trifásico e fase terra nas barras selecionadas e
 * salva os resultados em disco.
 */

// Mostra os resultados na tela e salva em disco...
// Cálculo das correntes de curto circuito para as barras indicadas...
string arqRelatorio = "ccRelatorio.txt";
string txtTitulo = "Correntes e fases de curto, calculadas para as barras indicadas.\n";
string txtElementos = "num    barra          Vbase    Icc3f(kA)   fase          Iccft(kA)
fase\n";
string txtSublinha = "----- ===== - - - - - ===== - - - - - ===== - - - - -
\n";
cout << txtTitulo;
cout << txtElementos;
cout << txtSublinha;
saveReport(arqRelatorio, txtTitulo);
saveReport(arqRelatorio, txtElementos);
saveReport(arqRelatorio, txtSublinha);

unsigned indBar, ind;
double iBase;
char linhaRelatorio[256];
vector<string> resultado(numBarrasCC);
resultado.clear();
complex<double> zTh1, zTh0, iCC3F, iCCFT, iCC3Fpolar, iCCFTpolar;
cout << "Calculando as correntes e fases de curto circuito: \n";
for(unsigned i = 0; i < numBarrasCC; i++) {
    indBar = endBar[barrasCC[i]];
    ind = indBar - 1;
    iBase = sBase / sqrt(3) / barras[indBar].vBase;
    zTh1 = kX(1.0/100.0, zBus1->mat[ind * (zBus1->lin + 1)]); // Nas matrizes, valores em %
    zTh0 = kX(1.0/100.0, zBus0->mat[ind * (zBus0->lin + 1)]);
    iCC3F = kX(iBase, um / zTh1);
    iCCFT = kX(iBase, kX(3.0, um) / (kX(2.0, zTh1) + zTh0));
    iCC3Fpolar = polar(iCC3F);
    iCCFTpolar = polar(iCCFT);
    sprintf(linhaRelatorio, "%5d %12s %5.1f %9.3f %12.3f %9.3f %12.3f\n", barras[indBar].num,
        barras[indBar].nome.c_str(), barras[indBar].vBase,
        iCC3Fpolar.real(), iCC3Fpolar.imag(),
        iCCFTpolar.real(), iCCFTpolar.imag());
    resultado.push_back(linhaRelatorio);
}

// Mostra os resultados na tela e salva em disco...
resultado.begin();
for(unsigned r = 0; r < resultado.size(); r++) {
    cout << resultado.at(r).c_str(); // Tela
    saveReport(arqRelatorio, resultado.at(r)); // Disco
}

// Relatório final:
cout << endl << "RESULTADOS" << endl;
cout << "*****" << endl;
cout << "Numero de Barras no SE.....: " << numBarras << endl;
cout << "Numero de Barras em Curto no SE: " << numBarrasCC << endl;
cout << "Numero de Ligacoes no SE.....: " << numLigacoes << endl;
cout << "F I M ! ! !" << endl;

// Libera a memoria usada pelas matrizes para o SO
resultado.clear();
delete[] zBus0->mat;
delete[] zBus1->mat;
delete[] barras;
delete[] barrasCC;
delete[] endBar;

return 0;
} // main
//*****

```

Anexo F: Código fonte parShortCircuit.cpp

Programa em C++ para o Estudo de Caso 2 – Cálculo de Curto Circuito em Sistemas de Potência: Modelo paralelo.
--

```

/*
 * File:   parShortCircuit.cpp
 * Author: Marlim P. Menezes
 * Progama: Doutorado em Ciências - Engenharia Elétrica
 * Instituição: Escola Politécnica da Universidade de São Paulo - Capital
 * Objetivo: Efetuar o cálculo de curto circuito em linhas de transmissão de modo
 * paralelo, para Estudo de Caso em Programação Paralela de Alto Desempenho.
 * Created on 29 de Março de 2013, 20:56
 */

// Biblioteca Padrão...
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cstring>
#include <iomanip>
#include <vector>
#include <complex>
#include <cmath>
#include <omp.h>

using namespace std;

// Declaração da estrutura para as matrizes...
struct TMatrix {
    unsigned    lin; // Linhas na matriz.
    unsigned    col; // Colunas na matriz.
    complex<double> *mat; // Matriz.
};

// Declaração da estrutura para as barras...
struct TBarra {
    unsigned num; // Código de identificação da barra: (0..99999)
    string nome; // Nome da barra
    double vBase; // Tensão base da barra
};

// Declaração da estrutura para as ligações...
struct TLigacao {
    unsigned de;
    unsigned para;
    int nc;
    string estado;
    double r1, x1;
    double r0, x0;
};

// Declaração de constantes...
const complex<double> zero = 0.0;
const complex<double> um = 1.0;
const double sBase = 100.0; // MVA.
const unsigned nummax = 99999; // Maior identificador de barra.
const double pouco = 1e-10;
const double pi = 3.141592653589793238462643383279; // ...

// Declaração de variáveis globais...
int numProcs; // Número de CPUs usadas em regiões paralelas...

// Funções protótipo...
/*****
void memAllocTo2dMatrix(TMatrix *M);
void guessSEBusSize(unsigned &nBar, const string arq,
    const string str1, const string nm);
void guessSECircuitSize(unsigned &nLig, const string arq,
    const string str2, const string nm);
void loadSEBusData(TBarra *barras, unsigned *endBar,

```

```

        const string arq, const string str1, const string nm);
void    loadSECircuitData(TLigacao *ligacoes, const string arq,
        const string str2, const string nm);
double  str2double(const double n);
void    guessSCSize(unsigned &nBarCC, const string arq);
void    loadSCDataBusses(unsigned *barCC, const string arq);
void    createYBus0(TMatrix *Bus0, const unsigned *endBar,
        const TLigacao *Lig, const unsigned numLig);
void    createYBus1(TMatrix *Bus1, const unsigned *endBar,
        const TLigacao *Lig, const unsigned numLig);
void    createIdentityMatrix(TMatrix *I);
void    triangMatrix(TMatrix *M, TMatrix *I);
void    backSubstitution(TMatrix *yBus, TMatrix *iBus, TMatrix *zBus, const unsigned col);
void    displayVector(unsigned *vec, const unsigned size);
void    display2dMatrix(TMatrix *M);
void    displayBusses(TBarra *B, const unsigned size);
void    displayCircuits(TLigacao *C, const unsigned size);
void    save2dMatrixToFile(const string arq, TMatrix *M);
void    saveReport(const string arq, const string reportLine);
complex<double> kX(const double k, complex<double> X);
complex<double> polar(complex<double> X);
inline int    sign(const double v) { return ((v > 1.0) ? 1 : -1); }

/*
 * Programa principal
 */
int main() {
    // Mensagem de inicialização.
    cout << "Estudo de caso de Calculo de Curto Circuito em Linhas de Transmissao." << endl;
    cout << "Modelo de programacao: Paralelo" << endl;
    cout << "Escola Politecnica da Universidade de Sao Paulo - PEA - 2013." << endl;
    cout << "===== " << endl;

    // Declaração das matrizes
    TMatrix *yBus0;
    TMatrix *yBus1;
    TMatrix *zBus0;
    TMatrix *zBus1;

    // Alocação de memória para as estruturas das matrizes.
    yBus0 = new TMatrix[sizeof(TMatrix)]; // Matriz retangular (n x 2n).
    yBus1 = new TMatrix[sizeof(TMatrix)]; // Matriz retangular (n x 2n).
    zBus0 = new TMatrix[sizeof(TMatrix)]; // Matriz quadrada (n x n).
    zBus1 = new TMatrix[sizeof(TMatrix)]; // Matriz quadrada (n x n).

    // Obtenção de dados iniciais do SE.
    string arqSE = "EPE_2011.ana"; // Arquivo com 5547 barras, para estudo de caso...
    string arqBarraCC = "EPE_2011_5000.bar"; // Arquivo contendo 5000 barras em curto
    string strDBAR = "DBAR"; // Identificador da seção das barras no arquivo do SE.
    string strDCIR = "DCIR"; // Identificador da seção das ligações no arquivo do SE.
    string strLim = "99999"; // Identificador do fim da seção procurada no arquivo do SE.

    // Obtem o número de barras e de ligações do SE, para alocação dinâmica de
    // memória para as estruturas...
    unsigned numBarras = 0, numLigacoes = 0, numBarrasCC = 0;
#pragma omp parallel sections num_threads(3) // 3 procs...
    {
        #pragma omp section // tarefa 1 => 1 proc
        {
            guessSEBusSize(numBarras, arqSE, strDBAR, strLim);
        }
        #pragma omp section // tarefa 2 => 1 proc
        {
            guessSECircuitSize(numLigacoes, arqSE, strDCIR, strLim);
        }
        #pragma omp section // tarefa 3 => 1 proc
        {
            // Obtem o número de barras em curto...
            guessSCSize(numBarrasCC, arqBarraCC); // Original
        }
    }

    // Define as dimensões das matrizes com os valores obtidos do arquivo do SE...

```

```

yBus1->lin = yBus0->lin = numBarras;
yBus1->col = yBus0->col = numBarras; // Matriz expandida p comporta a Identidade.
zBus1->lin = zBus1->col = zBus0->lin = zBus0->col = numBarras; // Matrizes quadradas.

// Aloca memória para as matrizes.
memAllocTo2dMatrix(yBus0); // n x n
memAllocTo2dMatrix(yBus1); // n x n
memAllocTo2dMatrix(zBus0); // n x n
memAllocTo2dMatrix(zBus1); // n x n

// Declara as estruturas para receber as barras e as ligações do SE...
TBarra *barras; // Lista de barras do SE
TLigacao *ligacoes; // Lista de ligações do SE
unsigned *endBar; // Índices das barras...
unsigned *barrasCC; // Barras em curto...

// Aloca memória para as barras, ligações, endbar e barrasCC...
endBar = new unsigned[nummax+1]; // Inclui a barra TERRA
endBar[0] = 0;
barras = new TBarra[numBarras+1]; // Inclui a barra TERRA
ligacoes = new TLigacao[numLigacoes]; // original: [4*numBarras]
barrasCC = new unsigned[numBarrasCC];

// Carrega os dados do SE: Barras e Ligações...
#pragma omp parallel sections num_threads(3)
{
    #pragma omp section // tarefa 1 => 1 proc
    {
        cout << "Vai carregar os dados do SE: Barras: \n";
        loadSEBusData(barras, endBar, arqSE, strDBAR, strLim);
    }
    #pragma omp section // tarefa 2 => 1 proc
    {
        cout << "Vai carregar os dados do SE: Ligacoes: \n";
        loadSECircuitData(ligacoes, arqSE, strDCIR, strLim);
    }
    #pragma omp section // tarefa 3 => 1 proc
    {
        cout << "Vai carregar os dados da Barras em Curto: \n";
        loadSCDataBusses(barrasCC, arqBarraCC); // Carrega a lista de barras em curto...
    }
}

// Cria as matrizes yBus0 e yBus1...
numProcs = 4;
#pragma omp parallel sections //num_threads(numCPUs)
{
    #pragma omp section // tarefa 1 => 2 procs
    {
        // Monta a matriz yBus0...
        cout << "Criando as matrizes yBus0 e yBus1: \n";
        createYBus0(yBus0, endBar, ligacoes, numLigacoes);
    }
    #pragma omp section // tarefa 2 => 2 procs
    {
        // Monta a matriz yBus1...
        cout << "Criando as matrizes yBus0 e yBus1: \n";
        createYBus1(yBus1, endBar, ligacoes, numLigacoes);
    }
}

// Libera a memória usada pelas listas de barras e ligacoes...
delete [] ligacoes;

// Salva as matrizes yBus no disco...
string arqYBus0 = "yBus0.txt";
save2dMatrixToFile(arqYBus0, yBus0);
string arqYBus1 = "yBus1.txt";
save2dMatrixToFile(arqYBus1, yBus1);

/*
 * Calcula o Curto Circuito em todas as barras solicitadas
 * =====

```

```

*/
TMatrix *iBus0, *iBus1; // Matrizes identidades para yBus0 e yBus1.
iBus0 = new TMatrix[sizeof(TMatrix)];
iBus1 = new TMatrix[sizeof(TMatrix)];
iBus0->lin = iBus0->col = iBus1->lin = iBus1->col = numBarras;
iBus0->mat = new complex<double>[numBarras * numBarras];
iBus1->mat = new complex<double>[numBarras * numBarras];

// Triangulariza as matrizes yBus...
// Executa duas tarefas independentes em paralelo...
numProcs = 4;
#pragma omp parallel sections //num_threads(numProcs)
{
    #pragma omp section // tarefa 1 => 2 procs
    {
        cout << "Criando a matriz Identidade iBus0: \n";
        createIdentityMatrix(iBus0);
    }
    #pragma omp section // tarefa 2 => 2 procs
    {
        cout << "Criando a matriz Identidade iBus1: \n";
        createIdentityMatrix(iBus1);
    }
} // parallel sections

// Executa duas tarefas independentes em paralelo...
// As duas tarefas seguintes são dependentes das anteriores...
numProcs = 4;
#pragma omp parallel sections num_threads(numProcs)
{
    #pragma omp section // tarefa 1 => numProcs/2 procs
    {
        triangMatrix(yBus0, iBus0);
    }
    #pragma omp section // tarefa 2 => numProcs/2 procs
    {
        triangMatrix(yBus1, iBus1);
    }
} // parallel sections

// Inverte yBus0 para obter zBus0...
// Realiza esta tarefa somente para as colunas necessárias, indicadas
// na lista de barras em curto circuito...
// Executa duas tarefas independentes em paralelo...
// As duas tarefas seguintes são dependentes das anteriores...
#pragma omp parallel sections //num_threads(numProcs)
{
    #pragma omp section // tarefa 1 =>
    {
        #pragma omp parallel for schedule(static) num_threads(4) //num_threads(numProcs/2)
        for(unsigned bc = 0; bc < numBarrasCC; bc++) {
            backSubstitution(yBus0, iBus0, zBus0, endBar[barrasCC[bc]]); // inversão coluna...
        }
    } // tarefa 1
    #pragma omp section // tarefa 2
    {
        // Inverte yBus1 para obter zBus1...
        // Realiza esta tarefa somente para as colunas necessárias, indicadas
        // na lista de barras em curto circuito...
        #pragma omp parallel for schedule(static) num_threads(4) //num_threads(numProcs/2)
        for(unsigned bc = 0; bc < numBarrasCC; bc++) {
            backSubstitution(yBus1, iBus1, zBus1, endBar[barrasCC[bc]]); // inversão coluna...
        }
    } // tarefa 2
} // parallel sections

// Libera a memoria usada pelas matrizes: yBus0, yBus1, iBus0 e iBus1.
// Pois as mesmas não serão mais usadas....
delete [] yBus0->mat;
delete [] yBus1->mat;
delete [] iBus0->mat;
delete [] iBus1->mat;

```

```

/*
 * Calcula o curto trifásico e fase terra nas barras selecionadas e
 * salva os resultados em disco.
 */

// Cálculos de curto e fase...
vector<string> resultado(numBarrasCC);
resultado.clear();
numProcs = 4;
cout << "Calculando as correntes e fases de curto circuito: \n";
#pragma omp parallel num_threads(numProcs)
#pragma omp for schedule(static)
for(unsigned i = 0; i < numBarrasCC; i++) {
    unsigned indBar = endBar[barrasCC[i]];
    unsigned ind = indBar - 1;
    double iBase = sBase / sqrt(3) / barras[indBar].vBase;
    complex<double> zTh1 = kX(1.0/100.0, zBus1->mat[ind * (zBus1->lin + 1)]); // Valores em %
    complex<double> zTh0 = kX(1.0/100.0, zBus0->mat[ind * (zBus0->lin + 1)]); // Valores em %
    complex<double> iCC3F = kX(iBase, um / zTh1);
    complex<double> iCCFT = kX(iBase, kX(3.0, um) / (kX(2.0, zTh1) + zTh0));
    complex<double> iCC3Fpolar = polar(iCC3F);
    complex<double> iCCFTpolar = polar(iCCFT);
    char linhaRelatorio[256];
    sprintf(linhaRelatorio, "%5d %12s %5.1f %9.3f %12.3f %9.3f %12.3f\n", barras[indBar].num,
        barras[indBar].nome.c_str(), barras[indBar].vBase,
        iCC3Fpolar.real(), iCC3Fpolar.imag(),
        iCCFTpolar.real(), iCCFTpolar.imag());
    resultado.push_back(linhaRelatorio);
}

// Mostra os resultados na tela e salva em disco...
// Cálculo das correntes de curto circuito para as barras indicadas...
string arqRelatorio = "ccRelatorio.txt";
string txtTitulo = "Correntes e fases de curto, calculadas para as barras indicadas.\n";
string txtElementos = "num      barra          Vbase      Icc3f(kA)   fase          Iccft(kA)
fase\n";
string txtSublinha = "----- ===== - - - - - ===== - - - - - ===== - - - - -
-\n";
saveReport(arqRelatorio, txtTitulo);
saveReport(arqRelatorio, txtElementos);
saveReport(arqRelatorio, txtSublinha);
resultado.begin();
for(unsigned r = 0; r < resultado.size(); r++) {
    cout << resultado.at(r).c_str(); // Tela
    saveReport(arqRelatorio, resultado.at(r)); // Disco
}

// Relatório final:
cout << endl << "RESULTADOS" << endl;
cout << "*****" << endl;
cout << "Numero de Barras no SE.....: " << numBarras << endl;
cout << "Numero de Barras em Curto no SE: " << numBarrasCC << endl;
cout << "Numero de Ligacoes no SE.....: " << numLigacoes << endl;
cout << "F I M ! ! !" << endl;

// Salva as matrizes zBus no disco...
string arqZBus0 = "zBus0.txt";
save2dMatrixToFile(arqZBus0, zBus0);
string arqZBus1 = "zBus1.txt";
save2dMatrixToFile(arqZBus1, zBus1);

// Libera a memoria usada pelas matrizes para o SO
resultado.clear();
delete[] zBus0->mat;
delete[] zBus1->mat;
delete[] barras;
delete[] barrasCC;
delete[] endBar;

return 0;
}

//*****

```

```

// Implementação das funções protótipo...
//*****
//*****

// Alocação de memória para as matrizes propriamente ditas.
// A alocação é individual para tentar obter-se endereços contíguos nas matrizes.
void memAllocTo2dMatrix(TMatrix *M)
{
    M->mat = new complex<double>[M->lin * M->col];
    // Verifica se alocou memória para a matriz yBus0.
    if(M->mat == NULL) {
        perror("\n\nERRO: Nao foi possivel alocar memoria para a matriz!\n");
        perror("Tente novamente com uma matriz menor.\n");
        perror("Programa interrompido.\n\n");
        exit(1);
    }
    // Inicializa a matriz com zero...
    memset(M->mat, 0, sizeof(complex<double>) * M->lin * M->col);
}
//*****

// Abre o arquivo do SE e obtém o número de barras (*nBar)...
void guessSEBusSize(unsigned &nBar, const string arq,
                    const string str1, const string nm)
{
    // Verifica se arq contém o nome do arquivo a ser aberto para leitura...
    if(arq.empty()) {
        cout << "ERRO: Arquivo nao encontrado!" << endl;
        cout << "Certifique-se de que o nome do arquivo esteja correto." << endl;
        cout << "Fim de programa..." << endl;
        exit(0);
    }
    // Declaração manipulador de fluxo de arquivo
    ifstream inArquivoDados; // Prepara para abrir o arquivo de dados ANA...para leitura.

    // Percorre o arquivo sequencialmente, à procura da sequência 'pal_1'...
    // Para iniciar a leitura dos dados a partir dela...
    cout << "Obtendo os dados necessarios do SE: " << arq.data() << endl;

    inArquivoDados.open(arq.c_str());
    // Ignora todas as linhas até encontrar a palavra chave 'pal_1'...
    // Percorre o arquivo linha a linha, extraindo os dados do SE
    bool dbar = false; // se dbar = true conta as barras
    string linha; // Uma linha lida do arquivo...
    while(!inArquivoDados.eof()) {
        getline(inArquivoDados, linha); // Lê uma linha inteira do arquivo
        // Obtendo o número de Barras do SE...
        if(linha.substr(0, 4).compare(str1.data()) == 0) { // "DBAR"?
            dbar = true;
            // Avança duas linhas no arquivo e pega a terceira linha
            getline(inArquivoDados, linha); // Ignora esta linha inteira do arquivo
            getline(inArquivoDados, linha); // Ignora esta linha inteira do arquivo
            getline(inArquivoDados, linha); // Cinsidera esta linha inteira do arquivo
        }
        if (dbar && linha.substr(0, 5).compare(nm.data()) != 0) { // ainda é DBAR? Carrega barras
            nBar++;
        } // fim de DBAR
        if(dbar && linha.substr(0, 5).compare(nm.data()) == 0) // Acabaram as barras?
            dbar = false;
    }
    // Fecha o arquivo
    inArquivoDados.close();
} // guessSEBusSize
//*****

// Abre o arquivo do SE e obtém o número de ligações (*nLig)...
void guessSECircuitSize(unsigned &nLig, const string arq,
                        const string str2, const string nm)
{
    // Verifica se arq contém o nome do arquivo a ser aberto para leitura...
    if(arq.empty()) {
        cout << "ERRO: Arquivo nao encontrado!" << endl;
        cout << "Certifique-se de que o nome do arquivo esteja correto." << endl;
    }
}

```

```

    cout << "Fim de programa..." << endl;
    exit(0);
}
// Declaração manipulador de fluxo de arquivo
ifstream inArquivoDados; // Prepara para abrir o arquivo de dados ANA...para leitura.

// Percorre o arquivo sequencialmente, à procura da sequência 'pal_1'...
// Para iniciar a leitura dos dados a partir dela...
cout << "Obtendo os dados necessarios do SE: " << arq.data() << endl;

inArquivoDados.open(arq.c_str());
// Ignora todas as linhas até encontrar a palavra chave 'pal_1'...
// Percorre o arquivo linha a linha, extraíndo os dados do SE
bool dcir = false; // se dcir = true conta as ligações
string linha; // Uma linha lida do arquivo...
while(!inArquivoDados.eof()) {
    getline(inArquivoDados, linha); // Lê uma linha inteira do arquivo
    // Obtendo o número de Ligações do SE...
    if(linha.substr(0, 4).compare(str2.data()) == 0) { // "DCIR"?
        dcir = true;
        // Avança duas linhas no arquivo e pega a terceira linha
        getline(inArquivoDados, linha); // Ignora esta linha inteira do arquivo
        getline(inArquivoDados, linha); // Ignora esta linha inteira do arquivo
        getline(inArquivoDados, linha); // Cinsidera esta linha inteira do arquivo
    }
    if(dcir && linha.substr(0, 5).compare(nm.data()) != 0) { // ainda é DCIR? Carrega
    ligações
        nLig++;
    } // fim de DCIR
    if(dcir && linha.substr(0, 5).compare(nm.data()) == 0) // Acabaram as ligações?
        dcir = false;
}
// Fecha o arquivo
inArquivoDados.close();
} // guessSECircuitSize
//*****

// Carrega os dados do SE: Barras...
void loadSEBusData(TBarra *barras, unsigned *endBar,
    const string arq, const string str1, const string nm)
{
    // Verifica se arq contém o nome do arquivo a ser aberto para leitura...
    if(arq.empty()) {
        cout << "ERRO: Arquivo nao encontrado!" << endl;
        cout << "Certifique-se de que o nome do arquivo esteja correto." << endl;
        cout << "Fim de programa..." << endl;
        exit(0);
    }
    // Declaração manipulador de fluxo de arquivo
    ifstream inArquivoDados; // Prepara para abrir o arquivo de dados ANA...para leitura.

    // Percorre o arquivo sequencialmente, à procura da sequência 'pal_1'...
    // Para iniciar a leitura dos dados a partir dela...
    cout << "Carregando dados do arquivo: " << arq.data() << endl;

    inArquivoDados.open(arq.c_str());
    unsigned numBarra = 0;
    double vbas; // Temporária para vBase...
    // Inicia a barra 0 como padrão.
    barras[numBarra].num = 0; // NB
    barras[numBarra].nome = "TERRA"; // BN
    barras[numBarra++].vBase = 0.0; // VBAS
    // Percorre o arquivo linha a linha, extraíndo os dados do SE
    bool dbar = false; // se dbar = true carrega barras
    string linha; // Uma linha lida do arquivo...
    while(!inArquivoDados.eof()) {
        getline(inArquivoDados, linha); // Lê uma linha inteira do arquivo
        // Extraíndo os dados das Barras do SE...
        if(linha.substr(0, 4).compare(str1.data()) == 0) { // "DBAR"?
            dbar = true;
            // Avança duas linhas no arquivo e pega a terceira linha
            getline(inArquivoDados, linha); // Ignora esta linha inteira do arquivo
            getline(inArquivoDados, linha); // Ignora esta linha inteira do arquivo

```

```

    getline(inArquivoDados, linha); // Cinsidera esta linha inteira do arquivo
}
if (dbar && linha.substr(0, 5).compare(nm.data()) != 0) { // ainda é DBAR? Carrega barras
    barras[numBarra].num = atoi(linha.substr(0, 5).c_str()); // NB
    barras[numBarra].nome = linha.substr(9, 12); // BN
    vbas = atof(linha.substr(31, 4).c_str());
    if(vbas == 0.0) vbas = 1.0; // Para evitar divisão por zero...
    barras[numBarra].vBase = vbas; // VBAS
    endBar[barras[numBarra].num] = numBarra;
    numBarra++;
} // fim de DBAR
if(dbar && linha.substr(0, 5).compare(nm.data()) == 0) // Acabaram as barras?
    dbar = false;
}
// Fecha o arquivo
inArquivoDados.close();
} // loadSEBusData
//*****

// Carrega os dados do SE: Ligações...
void loadSECircuitData(TLigacao *ligacoes, const string arq,
    const string str2, const string nm)
{
    // Verifica se arq contém o nome do arquivo a ser aberto para leitura...
    if(arq.empty()) {
        cout << "ERRO: Arquivo nao encontrado!" << endl;
        cout << "Certifique-se de que o nome do arquivo esteja correto." << endl;
        cout << "Fim de programa..." << endl;
        exit(0);
    }
    // Declaração manipulador de fluxo de arquivo
    ifstream inArquivoDados; // Prepara para abrir o arquivo de dados ANA...para leitura.

    // Percorre o arquivo sequencialmente, à procura da sequência 'pal_1'...
    // Para iniciar a leitura dos dados a partir dela...
    cout << "Carregando dados do arquivo: " << arq.data() << endl;

    inArquivoDados.open(arq.c_str());
    unsigned numLigacao = 0;
    bool dcir = false; // se dcir = true carrega ligações
    string linha; // Uma linha lida do arquivo...
    while(!inArquivoDados.eof()) {
        getline(inArquivoDados, linha); // Lê uma linha inteira do arquivo

        // Extraindo os dados das Ligações do SE...
        if(linha.substr(0, 4).compare(str2.data()) == 0) { // "DCIR"?
            dcir = true;
            // Avança duas linhas no arquivo e pega a terceira linha
            getline(inArquivoDados, linha); // Ignora esta linha inteira do arquivo
            getline(inArquivoDados, linha); // Ignora esta linha inteira do arquivo
            getline(inArquivoDados, linha); // Cinsidera esta linha inteira do arquivo
        }
        if(dcir && linha.substr(0, 5).compare(nm.data()) != 0) { // ainda é DCIR? Carrega
        ligações
            ligacoes[numLigacao].de = atoi(linha.substr(0, 5).c_str()); // BF (DE)
            ligacoes[numLigacao].para = atoi(linha.substr(7, 5).c_str()); // BT (PARA)
            ligacoes[numLigacao].nc = atoi(linha.substr(14, 2).c_str()); // NC
            ligacoes[numLigacao].r1 = str2double(atof(linha.substr(17, 6).c_str())); // R1
            ligacoes[numLigacao].x1 = str2double(atof(linha.substr(23, 6).c_str())); // X1
            ligacoes[numLigacao].r0 = str2double(atof(linha.substr(29, 6).c_str())); // R0
            ligacoes[numLigacao].x0 = str2double(atof(linha.substr(35, 6).c_str())); // X0
            numLigacao++;
        } // fim de DCIR
        if(dcir && linha.substr(0, 5).compare(nm.data()) == 0) // Acabaram as ligações?
            dcir = false;
    }
    // Fecha o arquivo
    inArquivoDados.close();
} // loadSECircuitData
//*****

// Obtém o número de barras a ter o curto calculado no arquivo...
void guessSCSize(unsigned &nBarCC, const string arq)

```

```

{
    if(arq.empty()) {
        cout << "ERRO: Arquivo de dados não encontrado!" << endl;
        cout << "Certifique-se de que o nome do arquivo esteja correto." << endl;
        exit(1);
    }
    // Se preparando para abrir o arquivo
    ifstream inArquivoDados; // Prepara para abrir o arquivo de dados ANA...para leitura.

    // Verifica se o arquivo arq pode ser aberto para leitura...
    if(!inArquivoDados) {
        cout << "ERRO: Nao foi possivel abrir o arquivo de dados < " << arq.c_str() << " >." <<
endl;
        cout << "Forneca um nome de arquivo valido." << endl;
        exit(1);
    }

    // Percorre o arquivo sequencialmente, à procura da sequência 'pal_1'...
    // Para iniciar a leitura dos dados a partir dela...
    cout << "Obtendo o numero de barras a ter o curto calculado do arquivo: " << arq.data() <<
endl;

    inArquivoDados.open(arq.c_str());
    string linha;
    while(!inArquivoDados.eof()) {
        getline(inArquivoDados, linha); // Lê uma linha inteira do arquivo
        if(!linha.empty())
            nBarCC++;
    }
    // Fecha o arquivo
    inArquivoDados.close();
} // guessSCSize
/*****

// Carrega a lista das barras a terem o curto calculado.
void loadSCDataBusses(unsigned *barCC, const string arq)
{
    if(arq.empty()) {
        cout << "ERRO: Arquivo de dados não encontrado!" << endl;
        cout << "Certifique-se de que o nome do arquivo esteja correto." << endl;
        exit(1);
    }
    // Se preparando para abrir o arquivo
    ifstream inArquivoDados; // Prepara para abrir o arquivo de dados ANA...para leitura.

    // Verifica se o arquivo arq pode ser aberto para leitura...
    if(!inArquivoDados) {
        cout << "ERRO: Nao foi possivel abrir o arquivo de dados < " << arq.c_str() << " >." <<
endl;
        cout << "Forneca um nome de arquivo valido." << endl;
        exit(1);
    }

    // Para iniciar a leitura dos dados a partir dela...
    cout << "Carregando dados do arquivo: " << arq.data() << endl;
    unsigned bc = 0L;
    inArquivoDados.open(arq.c_str());
    string linha;
    while(!inArquivoDados.eof()) {
        getline(inArquivoDados, linha); // Lê uma linha inteira do arquivo
        if(!linha.empty())
            barCC[bc++] = atoi(linha.substr(0, 5).c_str());
    }
    // Fecha o arquivo
    inArquivoDados.close();
} // loadSCDataBusses
/*****

// Monta a matriz yBus0...
void createYBus0(TMatrix *Bus0, const unsigned *endBar,
    const TLigacao *Lig, const unsigned numLig)
{
    unsigned lin, col, ln, cl;

```

```

    complex<double> y0, z0;
#pragma omp parallel num_threads(numProcs/2)
#pragma omp for
for(unsigned i = 0; i < numLig; i++) {
    lin = endBar[Lig[i].de];
    col = endBar[Lig[i].para];
    z0.real(Lig[i].r0);
    z0.imag(Lig[i].x0);
    y0 = um / z0;

    // Se lin ou col é 0, então o elemento é da diagonal...
    if(lin == 0) lin = col;
    if(col == 0) col = lin;
    ln = lin - 1;
    cl = col - 1;
    Bus0->mat[ln * (Bus0->lin + 1)] += y0;
    if(lin != col) {
        Bus0->mat[cl * (Bus0->lin + 1)] += y0;
        Bus0->mat[ln * Bus0->lin + cl] -= y0;
        Bus0->mat[cl * Bus0->lin + ln] = Bus0->mat[ln * Bus0->lin + cl];
    }
}
} // createYBus0
//*****

// Monta a matriz yBus1...
void createYBus1(TMatrix *Bus1, const unsigned *endBar,
    const TLigacao *Lig, const unsigned numLig)
{
    unsigned lin, col, ln, cl;
    complex<double> y1, z1;
#pragma omp parallel num_threads(numProcs/2)
#pragma omp for
for(unsigned i = 0; i < numLig; i++) {
    lin = endBar[Lig[i].de];
    col = endBar[Lig[i].para];
    z1.real(Lig[i].r1);
    z1.imag(Lig[i].x1);
    y1 = um / z1;

    // Se lin ou col é 0, então o elemento é da diagonal...
    if(lin == 0) lin = col;
    if(col == 0) col = lin;
    ln = lin - 1;
    cl = col - 1;
    Bus1->mat[ln * (Bus1->lin + 1)] += y1;
    if(lin != col) {
        Bus1->mat[cl * (Bus1->lin + 1)] += y1;
        Bus1->mat[ln * Bus1->lin + cl] -= y1;
        Bus1->mat[cl * Bus1->lin + ln] = Bus1->mat[ln * Bus1->lin + cl];
    }
}
} // createYBus1
//*****

// Cria a matriz identidade...
void createIdentityMatrix(TMatrix *I)
{
    // Preenche a matriz com zero
    memset(I->mat, 0, sizeof(complex<double>) * I->lin * I->col);
    // Preenche a diagonal principal com um...
#pragma omp parallel for num_threads(numProcs/2)
for(unsigned i = 0; i < I->lin; i++)
    I->mat[i * (I->lin + 1)] = um;
} // createIdentityMatrix
//*****

// Triangulariza a matriz M (n x n)...
void triangMatrix(TMatrix *M, TMatrix *I)
{
    complex<double> m;
    unsigned n = M->lin;
    if(n > 1) {

```

```

#pragma omp parallel num_threads(numProcs)//num_threads(numProcs/2)
{
    for(unsigned i = 0; i < n; i++) {
        m = M->mat[i * (n+1)]; // pivô...externo
        if(abs(m) > pouco) { // compara com 1e-10.
            // Divide a linha i pelo seu pivô m...De ambas as matrizes M e I...
            for(unsigned j = i; j < n; j++) // Atualiza M
                M->mat[i*n+j] /= m;
            for(unsigned j = 0; j < n; j++) // Atualiza I
                I->mat[i*n+j] /= m;
            #pragma omp for schedule(static)
            for(unsigned k = i+1; k < n; k++) {
                m = M->mat[k*n+i]; // pivô...interno
                for(unsigned s = 0; s < n; s++) {
                    M->mat[k*n+s] -= m * M->mat[i*n+s];
                    I->mat[k*n+s] -= m * I->mat[i*n+s];
                } // for s
            } // for k
        } // if abs()
    } // for i
    m = M->mat[n*n-1];
    M->mat[n*n-1] = um;
    #pragma omp for schedule(static)
    for(unsigned u = 0; u < n; u++) {
        I->mat[(n-1)*n+u] /= m;
    } // for u
} // omp parallel
} // if lin
} // triangMatrix
//*****

// calcula as incógnitas do Sistema Linear pelo método Back Substitution...
void backSubstitution(TMatrix *yBus, TMatrix *iBus, TMatrix *zBus, const unsigned col)
{
    int lin = yBus->lin, llin = lin-1, coll = col-1;
    zBus->mat[llin*lin+coll] = yBus->mat[llin*lin+coll];
    if(lin == 1) {
        zBus->mat[0] /= yBus->mat[0] / yBus->mat[1];
    }
    #pragma omp parallel for schedule(static) num_threads(numProcs)
    for(int i = llin; i >= 0; i--) {
        zBus->mat[i*lin+coll] = iBus->mat[i*lin+coll];
        for(unsigned k = i+1; k < lin; k++) {
            zBus->mat[i*lin+coll] -= yBus->mat[i*lin+k] * zBus->mat[k*lin+coll];
        }
    }
}
//*****

// Mostra a matriz na tela.
void display2dMatrix(TMatrix *M) {
    for(unsigned y = 0; y < M->lin; y++) { // linhas
        for(unsigned x = 0; x < M->col; x++) { // colunas
            if(M->mat[y * M->col + x].imag() < 0.0)
                printf("%.2f%.2fj ", M->mat[y * M->col + x].real(), M->mat[y * M->col + x].imag());
            else
                printf("%.2f%.2fj ", M->mat[y * M->col + x].real(), M->mat[y * M->col + x].imag());
        }
        cout << endl;
    }
}
//*****

// Grava a matriz quadrada M no disco.
void save2dMatrixToFile(const string arq, TMatrix *M) {
    FILE *ptrArq;
    ptrArq = fopen(arq.c_str(), "w"); // Abre arquivo p gravação.
    if(ptrArq == NULL) { // Falhou na criação do arquivo em disco
        perror("Erro ao criar o arquivo de dados em disco.");
        perror("Os dados não serão gravados no disco.");
    }
    else {

```

```

        for(unsigned y = 0; y < M->lin; y++) { // linhas
            for(unsigned x = 0; x < M->col; x++) { // colunas
                if(M->mat[y * M->col + x].imag() < 0.0)
                    fprintf(ptrArq, "%9.5f%8.5fj ", M->mat[y * M->col + x].real(), M->mat[y * M->col + x].imag());
                else
                    fprintf(ptrArq, "%9.5f+%8.5fj ", M->mat[y * M->col + x].real(), M->mat[y * M->col + x].imag());
            }
            fprintf(ptrArq, "\n");
        }
        fflush(ptrArq);
        fclose(ptrArq);
    }
}
//*****

// Salva o relatório em disco, linha a linha.
void saveReport(const string arq, const string reportLine) {
    FILE *ptrArq;
    ptrArq = fopen(arq.c_str(), "a+"); // Abre arquivo p gravação.
    if(ptrArq == NULL) { // Falhou na criação/abertura do arquivo em disco
        perror("Erro ao criar/abrir o arquivo de dados em disco.");
        perror("Os dados não serão gravados em disco.");
    }
    else {
        fprintf(ptrArq, "%s", reportLine.c_str());
        fflush(ptrArq);
        fclose(ptrArq);
    }
}
//*****

```