UNIVERSIDADE FEDERAL DA BAHIA
INSTITUTO DE GEOCIÊNCIAS
CURSO DE GRADUAÇÃO EM GEOFÍSICA

GEO213 – TRABALHO DE GRADUAÇÃO

IMPLEMENTAÇÃO DE ALGORITMO PARALELO PARA INVERSÃO DE DADOS GEOFÍSICOS

LUCAS GONDIM MIRANDA

SALVADOR – BAHIA
DEZEMBRO – 2011

Implementação de algoritmo paralelo para inversão de dados geofísicos

por

Lucas Gondim Miranda

GEO213 – TRABALHO DE GRADUAÇÃO

Departamento de Geologia e Geofísica Aplicada

do

Instituto de Geociências

da

Universidade Federal da Bahia

Comissão Examinadora

_____ Dr. Milton José Porsani - Orientador

_____ Dr. Wilson Figueiró

_____ Dr. Amin Bassrei

Data da aprovação:

Dedico este trabalho à minha família,
em especial aos meus pais e irmãos,
e aos meus amigos.

RESUMO

Um problema frequentemente encontrado na inversão de dados geofísicos é o grande volume de informações, cujo processamento torna-se inviável dentro da ótica da computação convencional. Atualmente, em função do desenvolvimento da indústria de *hardwares*, é possível superar os empecilhos gerados pelo déficit de memória e pelo excesso de tempo despendido para o processamento dos dados pertinentes ao modelamento e à inversão geofísica. Para isso, as universidades e a indústria possuem uma excelente ferramenta em crescente desenvolvimento desde o início da década de 90, denominada processamento paralelo. Diante de tal contexto, Porsani et al. propuseram um algoritmo, paralelizável, tipo-Levinson para solucionar sistemas de equações normais matriz-particionada que resolvem entre outros problemas, o de mínimos quadrados, relacionados com a inversão de dados geofísicos.

A técnica dos mínimos quadrados é encontrada na solução de muitos problemas relacionados com a estimação e análises de dados geofísicos. Exemplos deste método na exploração sísmica incluem vários algoritmos de processamento de dados, inversão para encontrar parâmetros de reservatório e estimação do background de velocidades. Um impedimento constantemente encontrado nestas situações é que o volume de dados é tão grande que as matrizes requeridas para a solução mínimos quadrados não podem ser armazenadas na memória de um único computador. Torna-se imperativo, portanto, a implementação do algoritmo paralelo, desenvolvido por Porsani et al., para que haja a solução efetiva de sistemas de equações normais matriz-particionada.

O presente trabalho discute o processamento paralelo de uma particularização do algoritmo já mencionado. Foi desenvolvido um método para solucionar, através de computação paralela, um sistema de equações normais tipo-bloco de ordem $j = 4$, sendo j a ordem da partição do sistema. O desempenho deste programa paralelo foi avaliado de duas formas distintas: uma baseada na Lei de Amdahl e outra na Lei de Gustafson-Barsis. Por fim, para poder prever o desempenho deste programa paralelo na solução de sistemas de ordem j superiores a 4, foi utilizado juntamente com a Lei de Amdahl, a métrica de Karp-Flatt.

ABSTRACT

A problem often found in geophysics data inversion is the great volume of information whose processing is not viable within the perspective of conventional processing.Currently, according to industry development of hardware, you can overcome the obstacles created by the lack of memory and the excess of time spent to compute the data related to the modeling and geophysical inversion. The universities and industry have an excellent tool in increasing development since the early 90's, called parallel processing. Given such context, Porsani et al. proposed an algorithm, parallelizable, Levinson-type, that solve systems of normal equations matrix partitioned, that solve least squares problems related to the inversion of geophysical data.

The technique of least squares is found in the solution of many problems related to estimation and analysis of geophysical data. Examples of this method in seismic exploration include several algorithms for data processing, inversion to find the reservoir parameters and estimation of the background of speeds. A frequent impediment found in these situations is that the volume of data is such that the matrices required for the least squares solution can not be stored in the memory of a single computer. It is imperative, therefore, the implementation of parallel algorithm - developed by Polsani et al. â so that there is an effective solution of systems of normal equations matrix-partitioned.

This paper discusses the parallel processing of a special feature of the algorithm mentioned above. We developed a method to solve, by parallel computing, a system of normal order block-type equations $j = 4$, with j equal to the order of the matrix partitioned in blocks. The performance of this parallel program was evaluated in two different ways: one based on Amdahl's Law and the other on the Law of Gustafson-Bars. Finally, in order to predict the performance of this parallel program to solve systems of order j, above 4, it was used along with Amdahl's Law, the Karp-Flatt metric.

ÍNDICE

ÍNDICE DE FIGURAS

INTRODUÇÃO

A utilização da computação paralela parece ser o caminho para a resolução de problemas complexos da Geofísica que exigem grande poder computacional. O paralelismo possibilita uma significante redução no tempo gasto na análise de problemas complexos, dando chance ao geofísico de explorar um número maior de alternativas diferentes em um mesmo período de tempo, possibilitando-se a obtenção de uma estrutura mais otimizada.

Uma forma de explorar o paralelismo é utilizando-se os chamados supercomputadores, que são máquinas extremamente poderosas, possuindo, geralmente, vários processadores e grande capacidade de armazenamento de dados. Entretanto, essas máquinas apresentam um alto custo, além de serem de difícil acesso para o usuário comum. Outra dificuldade é que essas máquinas possuem características muito particulares, o que dificulta o desenvolvimento de algoritmos que aproveitam totalmente o potencial do computador.

Uma alternativa de baixo custo aos supercomputadores é a utilização de *clusters* para realizar o paralelismo. Um *cluster* pode ser visto como uma máquina paralela com multiprocessadores de memória distribuída. Programada para o paralelismo utilizando-se o paradigma de passagem de mensagens. Essa passagem de mensagens é feita com a utilização de bibliotecas de comunicação. Dessa forma, o paralelismo pode ser efetivado com custo reduzido, utilizando-se redes de computadores já implementadas, que podem estar disponíveis para a grande maioria dos usuários.

Para lidarmos com problemas geofísicos 2D/3D, representando situações geológicas realistas, precisamos de algoritmos rápidos, robustos e que possam, portanto, tirar proveito das facilidades de computação paralela, hoje disponíveis nos Clusters de PCs. Uma classe especial de algoritmos recursivos tipo-Levinson (Levinson, 1947) vem sendo desenvolvida nas últimas décadas. Mesmo quando a matriz dos coeficientes não possui nenhuma estrutura especial além da simetria, ainda assim o princípio utilizado na recursão de Levinson pode ser empregado (Porsani e Ulrych, 1991). Recentemente, Porsani et al. 2008, estendeu aquele método para a solução de sistemas particionados. A estrutura desse novo algoritmo é apropriada para obtenção da solução dos sistemas através de multiprocessadores ou computação paralela. Essa abordagem desperta grande interesse na inversão de dados geofísicos uma vez que permite resolver sistemas de grande porte de forma rápida.

Este trabalho está dividido em quatro capítulos. No capítulo 1 é discutido os fundamentos do processamento atentando aos principais conceitos que regem a computação paralela. As definições e os detalhamentos a respeito das métricas de desempenho estão presentes no

capítulo 2. E os resultados das aplicações destas métricas na avaliação da *performance* do programa paralelo desenvolvido no presente trabalho são apresentados no capítulo 3. Por fim, no capítulo 4 apresentamos as conclusões e sugestões para trabalhos futuros.

CAPÍTULO 1

Fundamentos do Processamento Paralelo

1.1 Terminologia

Para um melhor entendimento do conteúdo deste capítulo é necessário ter o conhecimento de alguns termos usuais do processamento paralelo. Segue abaixo alguns deles:

- *Processos*: são as principais unidades do processamento paralelo em um ambiente de computação distribuída; comunicam-se através de trocas de mensagens.
- *Execução sequencial*: execução de um programa em um único processador, com as instruções sendo processadas uma de cada vez.
- *Paralelização de código*: consiste na transformação de um programa sequencial em paralelo, com a identificação de partes de código que podem ser executadas independentemente. Exige mudanças no código do programa e, caso necessário, no algoritmo utilizado no programa sequencial.
- *Ganho de velocidade (speed-up)*: consiste na comparação entre o tempo de execução do programa em um único processador e o tempo de execução utilizando vários processadores. Serve como parâmetro para a avaliação do desempenho de algoritmos paralelos.
- *Sincronização*: coordenação necessária entre processos para a troca de informações. Pode ser um fator de decréscimo da eficiência do programa, uma vez que alguns processadores podem ficar inativos, esperando pelo término de outros processos.
- *Granularidade*: Quando os processos executam poucas instruções e necessitam se comunicar muito, diz-se que o programa é muito granular. Quando, ao contrário, os processos executam muitas instruções, com pouca troca de informação, diz-se que o programa é pouco granular. Um programa com granularidade alta necessita de uma maior sincronização que um programa com menor granularidade, o que afeta o tempo de execução do programa.

- *Escalabilidade*: um sistema computacional paralelo é dito escalável se o ganho de velocidade conseguido cresce proporcionalmente ao número de processadores utilizados.
- *Balanceamento de carga*: consiste na distribuição equilibrada de tarefas entre os processadores, de forma a garantir uma execução eficiente do programa paralelo.
- Concorrência: identificar as partes do processamento que podem ser executadas de forma simultânea.

1.2 Cluster

Todos os testes dos programas em paralelo para elaboração deste trabalho foram realizados no cluster Águia que pertence ao CPGG-UFBA. O termo cluster é uma denominação dada a um sistema montado com mais de um computador, cuja finalidade é fazer com que todo o processamento seja distribuído aos computadores, mas de forma que pareça com que eles sejam um computador só. Com isso, é possível realizar processamentos que até então somente computadores de alta performance seriam capazes de fazer. Cada computador de um cluster é denominado nó ou nodo. Todos devem ser interconectados, de maneira a formarem uma rede. Essa rede precisa ser criada de uma forma que permita o adicionamento ou o decréscimo de um nó (em casos de defeitos, por exemplo), mas sem cessar o funcionamento do cluster.

O sistema operacional usado nos computadores deve ser de um mesmo tipo, ou seja, ou somente Windows, ou somente Linux. Isso devido ao fato de haver peculiaridades em cada sistema operacional que poderiam fazer com que o cluster parasse de funcionar. Para que exista, um cluster necessita de no mínimo dois computadores. É óbvio que, quanto mais computadores existir no cluster, maiores serão os custos de implementação e manutenção. Isso não se deve apenas ao preço dos computadores, mas também pelos equipamentos (switches, cabos, hubs, nobreaks, etc). Mas ainda assim, os custos costumam ser menores do que a aquisição ou/e manutenção de computadores poderosos e algumas vezes a computação dos dados é até mais eficiente.

1.3 Modelo de Organização da Memória

Há dois modelos básicos de organização da memória: memória compartilhada e memória distribuída. Quando se utiliza o modelo de memória compartilhada, todos os processadores tem acesso total (escrita e leitura) a qualquer compartimento da memória.

Tem como vantagem a alta velocidade de acesso aos dados e, como desvantagem, a limitação da quantidade de caminhos entre os processadores e a memória, o que diminui a

escalabilidade do sistema. Um acréscimo exagerado no número de processadores provocaria um congestionamento na rede de interconexão.

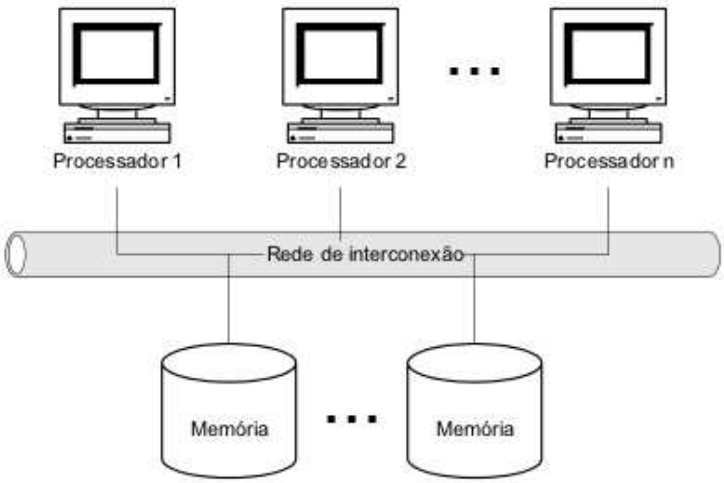

Figura 1.1: Arquitetura de memória compartilhada

O outro modelo básico de organização da memória é o modelo de memória distribuída. Neste caso, a memória é distribuída fisicamente entre os processadores, sendo de acesso exclusivo do processador ao qual está associada. As informações são trocadas entre os processadores por meio do envio de mensagens pela rede de comunicação.

As vantagens do modelo de memória distribuída são a escalabilidade do sistema, menor probabilidade de ocorrência de falhas, e também a facilidade de se incorporar novos processadores ao sistema, podendo-se utilizar conjuntamente processadores de arquiteturas diferentes. Neste trabalho, utilizou-se o modelo de memória distribuída, com a programação baseada na filosofia de troca de mensagens.

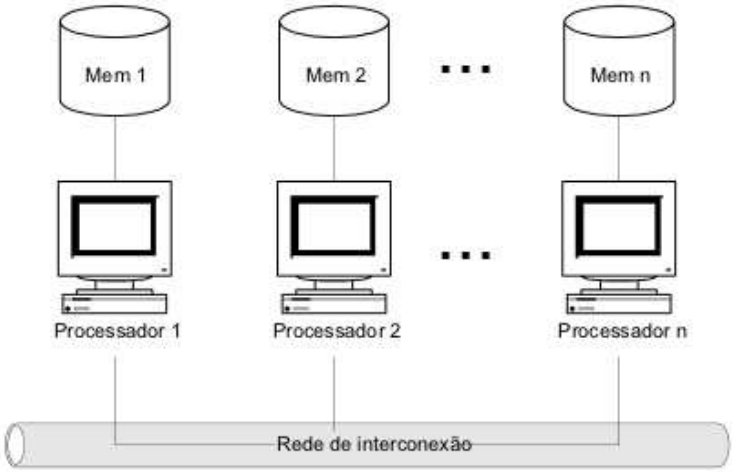

Figura 1.2: Arquitetura de memória distribuída

1.4 MPI

A troca de mensagens entre os processadores é realizada mediante um código padrão chamado MPI. A MPI (Message Passing Interface) é uma especificação para uma biblioteca de passagem de mensagens entre processadores, permitindo a comunicação entre os nós dessa rede. Hoje em dia, o MPI é o padrão para projetos de computação de alto desempenho, em máquinas paralelas e com memória distribuída. É interessante saber que o MPI não é uma linguagem de programação. O MPI é como se fosse uma linguagem de formatação como o *html* que é utilizada na construção de *homepages* na *web*.

O MPI é independente da linguagem, e existem interfaces específicas para C, C++ e FORTRAN, além de qualquer linguagem que possa usar o mesmo modelo de interface. Devido à sua estrutura e projeto mais avançado, o MPI tem ganhos substanciais em termos de velocidade e portabilidade. Logo, problemas que exijam cálculos em quantidade muito grande e computação de alto desempenho, podem ser modelados com uma linguagem de programação e o MPI como biblioteca de passagem de mensagens.

Existem diversas implementações do MPI, algumas de código aberto, e outras implementadas por empresas (como a HP-MPI, da HP). Entre elas, destacamos o Open MPI, que é uma implementação em código aberto da especificação MPI-2 e mantida por um grupo de pessoas formado por membros da indústria, academia e centros de pesquisa. O OpenMPI é liberado segundo a nova licença BSD, e está em constante desenvolvimento.

Mais informações sobre as rotinas MPI utilizadas neste trabalho encontram-se no Apêndice B.

1.4.1 Comunicação

A natureza do problema e o tipo de decomposição determinam o padrão de comunicação entre os diferentes processos. Para haver cooperação entre os processos é necessário definir algoritmos e estruturas de dados que permitam uma eficiente troca de informação. Alguns dos principais fatores que limitam essa eficiência são:

- Custo da comunicação: existe sempre um custo de tempo de processamento associado à troca de informação, e o tempo que um processador gasta para enviar uma mensagem a outro processador não contribui para melhorar a eficiência da computação dos dados.
- Necessidade de sincronização: o intervalo de tempo que os processos ficam à espera do instante de tempo ideal da sincronização também não contribui para a eficiência da computação dos dados.

- Latência (tempo mínimo de comunicação entre dois pontos) e Largura de Banda (quantidade de informação comunicada por unidade de tempo): para ter um processamento mais eficiente é recomendável enviar poucas mensagens grandes do que muitas mensagens pequenas.

Comunicação Síncrona

Os processos são executados de forma coordenada e sincronizada: a comunicação apenas se concretiza quando os nós estão sincronizados.

Comunicação Assíncrona

Os processos são executados de forma independente, não necessitando de sincronização para a transferência dos dados.

1.5 Modos de programação

De maneira geral, os modos de programação paralela podem ser classificados em dois modelos:

- *Paralelismo de dados*: uma mesma sequência de intruções é executada por cada processador sobre dados diferentes.
- *Paralelismo de processos*: o programa é particionado em processos cooperativos. Estes processos podem executar procedimentos bastante diferentes entre si, sem necessariamente haver uma sincronização entre os mesmos.

1.5.1 Principais Paradigmas da Programação Paralela

A depender de como é a estrutura do programa em paralelo que será feito para resolver algum problema, o programador poderá usufruir de alguns paradigmas já existentes. Abaixo estão descritos os principais paradigmas da programação paralela:

Master-Slave

Este paradigma divide a computação em dois componentes distintos: o processador *master* e o conjunto dos processadores *slaves*. O *master* é o responsável por decompor o problema em tarefas, distribuí-las pelos *slaves* e recolher os resultados parciais destes, de modo a calcular o

resultado final. O ciclo de execução dos *slaves* é muito simples: obter uma tarefa do *master*, processá-la e enviar o resultado de volta para o *master*.

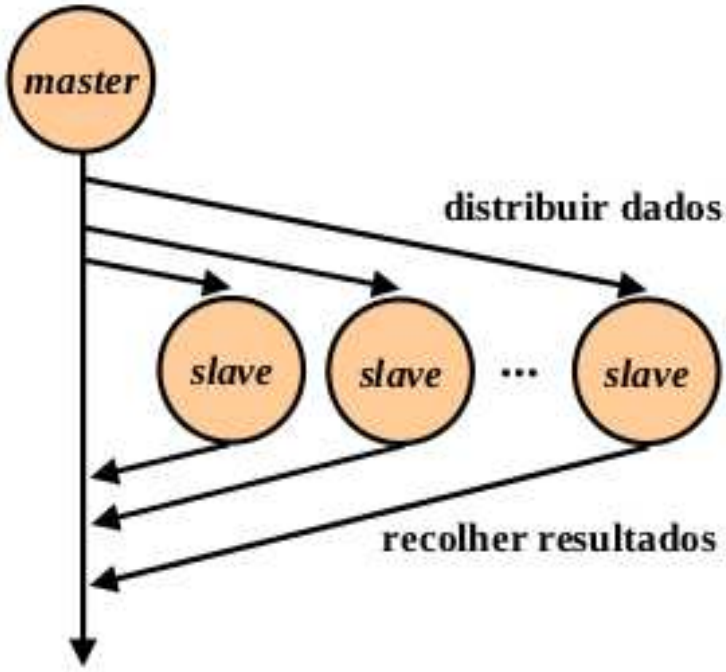

Figura 1.3: Esquema de um Paradigma Master-Slave

Single Program Multiple Data (SPMD)

Neste paradigma todos os processos executam o mesmo código fonte. Este paradigma é também conhecido como:

- *Geometric Parallelism*
- *Data Parallelism*

Tipicamente, os dados são bem distribuídos (mesma quantidade e regularidade) e o padrão de comunicação é bem definido: os dados ou são lidos individualmente por cada processador ou um dos processadores é o responsável por ler todos os dados e depois distribuí-los pelos restantes processadores.

Como os dados são bem distribuídos e o padrão de comunicação é bem definido, este paradigma consegue bons desempenhos e um elevado grau de escalabilidade. No entanto, este paradigma é muito sensível a falhas. A perda de um processador causa necessariamente uma diminuição da eficiênia do processamento.

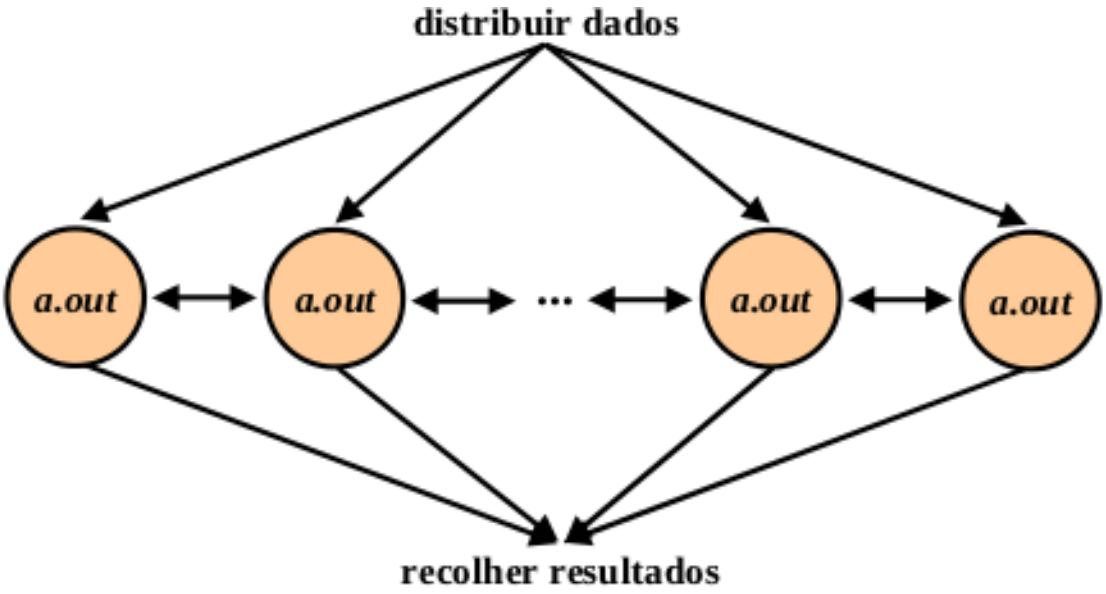

Figura 1.4: Esquema de um Paradigma SPMD

Divide and Conquer

A computação fica organizada numa espécie de árvore virtual: os processadores nos nós folha processam as subtarefas. Os restantes dos processadores são responsáveis por criar as subtarefas e por agregar os seus resultados parciais. O padrão de comunicação é bem definido e bastante simples: como as subtarefas são totalmente independentes, não é necessário qualquer tipo de comunicação durante o processamento das mesmas. Apenas existe comunicação entre o processador que cria as subtarefas e os processadores que as computam.

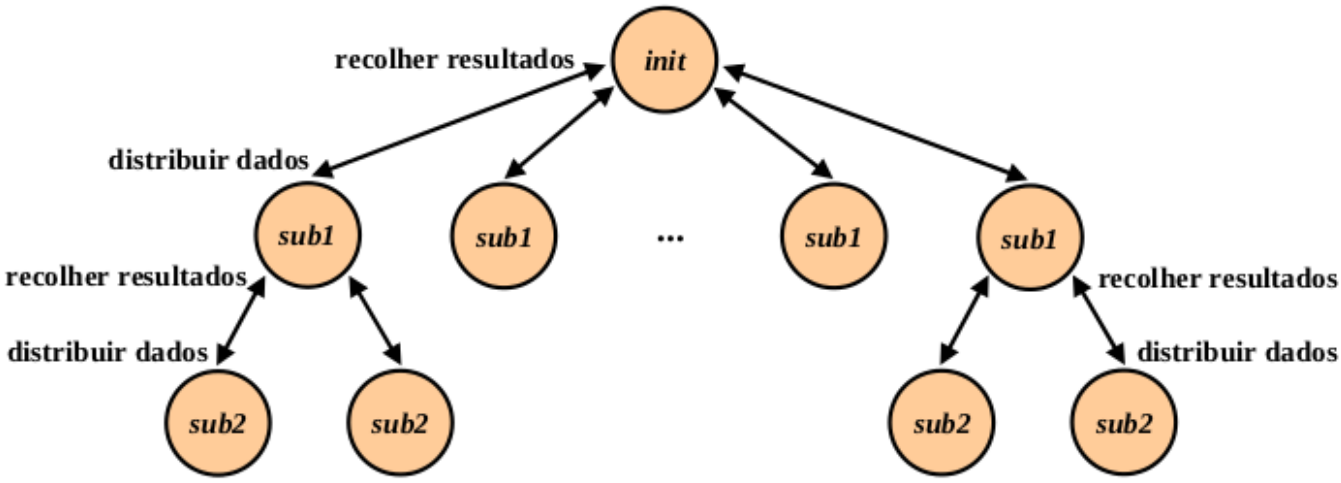

Figura 1.5: Esquema de um Paradigma Data Divide and Conquer

Speculative Parallelism

É utilizado quando as dependências entre os dados são tão complexas que tornam difícil explorar paralelismo usando os paradigmas anteriores. Este paradigma introduz paralelismo nos problemas através da execução de computações especulativas: a ideia é antecipar a execução de computações relacionadas com o processamento corrente na hipótese otimista de que essas computações serão necessariamente realizadas posteriormente. Quando isso não acontece, pode ser necessário repor partes do estado do processamento de modo a não violar a consistência do problema em resolução. Uma outra aplicação deste paradigma é quando se utiliza simultaneamente diversos algoritmos para resolver um determinado problema e se escolhe aquele que primeiro obtiver uma solução.

1.6 Metodologia da Programação de Foster

Um dos métodos mais conhecidos para desenhar algoritmos paralelos é a metodologia de Ian Foster (1996). O entendimento desta metodologia de programação paralela foi muito útil na execução deste trabalho. Esta metodologia permite que o programador se concentre inicialmente nos aspectos não dependentes da arquitetura, que são a concorrência e a escalabilidade, e só depois considere os aspectos dependentes da arquitetura, que são aumentar a localidade e diminuir a comunicação da computação.

A metodologia da programação de Foster divide-se em 4 etapas:

- Decomposição
- Aglomeração
- Granularidade
- Mapeamento

1.6.1 Decomposição

Uma forma de diminuir a complexidade de um problema é conseguir dividí-lo em tarefas ainda menores de modo a aumentar a concorrência e a localidade de referência de cada tarefa.

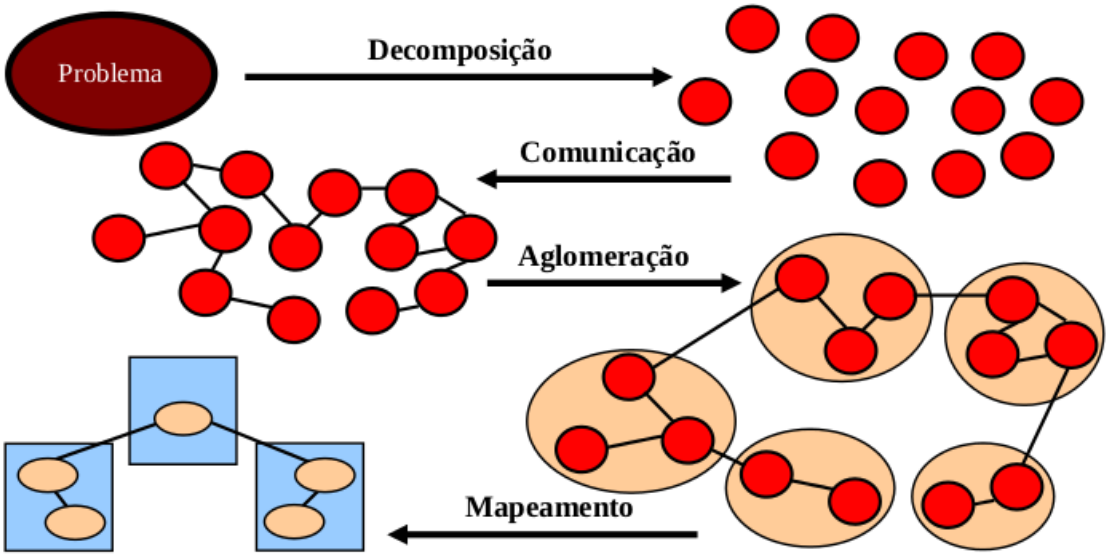

Figura 1.6: Estruturação da Metodologia Foster

1.6.2 Aglomeração

Aglomeração é o processo de agrupar tarefas de modo a diminuir os custos de implementação do algoritmo paralelo e também os custos de comunicação entre os processadores. O agrupamento em tarefas maiores permite uma maior reutilização do código do algoritmo sequencial na implementação do algoritmo paralelo. No entanto, o agrupamento deve garantir a escalabilidade do algoritmo paralelo de modo a evitar posteriores alterações.

1.6.3 Granularidade

Períodos de computação são tipicamente separados por períodos de comunicação entre os processadores. A granularidade é a medida qualitativa da razão entre processamento e comunicação. O número e o tamanho das tarefas em que a computação é agrupada determina a sua granularidade. Segue abaixo as vantagens e desvantagens de cada um dos tipos de granularidade:

Granularidade Fina

A computação é agrupada num grande número de pequenas tarefas.

Vantagem:

- Fácil de conseguir um balanceamento de carga eficiente.

Desvantagns:

- O tempo de computação de uma tarefa nem sempre compensa os custos de comunicação e sincronização.
- Difícil de se conseguir melhorar o desempenho.

Granularidade Grossa

A computação é agrupada num pequeno número de grandes tarefas.

Vantagens:

- O tempo de computação compensa os custos de comunicação e sincronização.
- Oportunidades para se conseguir melhorar o desempenho.

Desvantagem:

- Difícil de conseguir um balanceamento de carga eficiente.

1.6.4 Mapeamento

Mapeamento é o processo de atribuir tarefas a processadores de modo a maximizar a porcentagem de ocupação e minimizar a comunicação entre processadores. A porcentagem de ocupação é ótima quando a computação é balanceada de forma igual pelos processadores, permitindo que todos comecem e terminem as suas tarefas de forma simultânea. A porcentagem de ocupação decresce quando um ou mais processadores ficam suspensos enquanto os restantes continuam ocupados. A comunicação entre processadores é menor quando tarefas que comunicam entre si são atribuídas ao mesmo processador.

1.6.5 Fatores Limitativos do Desempenho

No processamento paralelo há um conjunto de fatores que influenciam a velocidade de processamento. Abaixo estão os pincipais:

- *Código Sequencial*: existem partes do código que são inerentemente sequenciais (isto é: iniciar/terminar a computação).
- *Comunicação*: existe sempre um custo associado à troca de informação entre os processadores.

- *Sincronização*: a partilha de dados entre os váris processadores pode levar a problemas de contenção no acesso à memória e enquanto os nós ficam à espera para sincronizar não contribuem para a computação.

- *Granularidade*: o número e o tamanho das tarefas é importante porque o tempo que demoram a ser executadas tem de compensar os custos da execução em paralelo (isto é custos de criação, comunicação e sincronização).

- *Balanceamento de Carga*: ter os processadores maioritariamente ocupados durante toda a execução é decisivo para o desempenho global do sistema.

CAPÍTULO 2

Métricas de Desempenho para Aplicações Paralelas

Dois dos principais objetivos das aplicações paralelas são melhorar o desempenho do processamento e da escalabilidade. Aumentar a performance do processamento significa reduzir o tempo de resolução do problema à medida que os recursos computacionais aumentam; e escalabilidade é a capacidade de aumentar o desempenho à medida que a complexidade do problema aumenta. Os fatores que condicionam o desempenho e a escalabilidade de uma aplicação paralela são os limites arquiteturais dos processadores e os limites algorítmicos.

As métricas de desempenho permitem avaliar a performance de uma aplicação paralela tendo por base a comparação entre a execução com múltiplos processadores e a execução com apenas um único processador. Esta comparação dos tempos de execução denomina-se *speedup*. Os dois principais métodos que avaliam a eficiência de um programa em paralelo são baseados na Lei de Amdahl e na Lei de Gustafson-Barsis, além duma medida secundária chamada Métrica de Karp-Flatt. Todos estes métodos de avaliação da performance de um programa em paralelo serão descritos a seguir.

2.1 Speedup

Seja T(p) o tempo gasto para a execução de uma dada tarefa utilizando um sistema de **p** processadores. O fator de *speedup* S(p) de um sistema de computação paralela com **p** processadores pode ser definido como a razão do tempo necessário para a execução serial T(1) pelo tempo gasto por um sistema paralelo T(p) na solução do mesmo problema. Este valor mostra o quanto mais rápido um algoritmo paralelo é em relação ao seu correspondente sequencial, ou seja, o quanto ele é escalável. Expressando na forma de equação tem-se:

$$S(p) = \frac{T(1)}{T(p)} \tag{2.1}$$

2.1.1 Lei de Amdahl

Para medir os benefícios que uma arquitetura paralela é capaz de oferecer, é necessário algum tipo de métrica que forneça em números o ganho obtido ao executar paralelamente uma determinada aplicação. Em outras palavras, obtém-se a razão entre o tempo de execução da aplicação rodando de forma puramente sequencial e o tempo de execução da mesma aplicação rodando de forma paralela.

A Lei de Amdahl é um dos métodos que tenta explicar o ganho obtido em uma aplicação ao adicionar mais processadores. O norueguês Gene Amdahl iniciou suas pesquisas seguindo a idéia de que um trecho do programa é puramente sequencial. Segundo Amdahl, apenas o desempenho do trecho que não é sequencial pode ser aumentado. Assim, o ganho de performance S (*speedup*) é dado pela seguinte expressão matemática:

$$S(p) \leq \frac{1}{f + \frac{1-f}{p}} \tag{2.2}$$

onde **f** é a fração serial do código em paralelo; **p** é a quantidade de processadores e **S(p)** é o valor do *speedup* em função da quantidade de processadores.

Por exemplo, suponha um programa que possua 60% do seu código puramente sequencial, e os 40% restantes do código podem ser executados paralelamente em dois processadores. Aplicando a Lei de Amdahl tem-se:

$$S(2) = \frac{1}{0.6 + \frac{1-0.4}{2}} = 1.25 \tag{2.3}$$

Portanto, o programa paralelo será 1.25 vezes mais rápido que o respectivo serial. Se o número de processadores tender ao infinito, o limite da fórmula de Amdahl tende a:

$$\frac{1}{f} \tag{2.4}$$

ou seja, o ganho máximo que se pode obter depende da porção do código que não pode ser executada paralelamente.

No exemplo acima, onde 60% do código do programa é puramente sequencial, o limite do ganho será de:

$$\frac{1}{0.6} = 1.67 \tag{2.5}$$

o que significa que o programa executará no máximo com um ganho de 67%, independentemente do número de processadores que forem adicionados. Observando a Lei de Amdahl, é fácil chegar à conclusão que é mais importante diminuir a parte sequencial do código do que aumentar o número de processadores.

Dedução da Lei de Amdahl

A computação realizada por uma aplicação paralela pode ser divididas em 3 classes:

C(seq): computações que só podem ser realizadas sequencialmente.

C(par): computações que podem ser realizadas em paralelo.

C(com): computações de comunicação/sincronização/iniciação.

Usando estas 3 classes, o *speedup* de uma aplicação pode ser definido do seguinte modo, de acordo com a Lei de Amdahl:

$$S(p) = \frac{T(1)}{T(p)} = \frac{C(seq) + C(par)}{C(seq) + \frac{C(par)}{p} + C(com)} \tag{2.6}$$

Como $C(com) \geq 0$ entao:

$$S(p) \leq \frac{C(seq) + C(par)}{C(seq) + \frac{C(par)}{p}} \tag{2.7}$$

Se **f** for a fração da computação que só pode ser realizada sequencialmente então:

$$f = \frac{C(seq)}{C(seq) + C(par)} \tag{2.8}$$

e

$$S(p) \leq \frac{\frac{C(seq)}{f}}{C(seq) + \frac{C(seq)*\left(\frac{1}{f}-1\right)}{p}} \tag{2.9}$$

Simplificando

$$S(p) \leq \frac{\frac{C(seq)}{f}}{C(seq) + \frac{C(seq)*\left(\frac{1}{f}-1\right)}{p}} \tag{2.10}$$

$$S(p) \leq \frac{\frac{1}{f}}{1 + \frac{\frac{1}{f}-1}{p}} \tag{2.11}$$

$$S(p) \leq \frac{1}{f + \frac{1-f}{p}} \tag{2.12}$$

Seja $f \geq 0$ e $f \leq 1$ a fração da computação que só pode ser realizada serialmente. A lei de Amdahl diz-nos que o *speedup* máximo que uma aplicação paralela com **p** processadores pode obter é:

$$S(p) \leq \frac{1}{f + \frac{1-f}{p}} \tag{2.13}$$

A lei de Amdahl fornece uma medida do *speedup* máximo que podemos obter ao resolver um determinado problema com **p** processadores. A lei de Amdahl também pode ser utilizada para determinar o limite máximo de *speedup* que uma determinada aplicação poderá alcançar independentemente do número de processadores a utilizar.

2.1.2 Lei de Gustafson-Barsis

A análise de Amdahl parte do princípio de que a fração serial (não-paralelizável) **f** do algoritmo é independente do número de processadores. Gustafson e Barsis argumentaram que, do ponto de vista prático, esta consideração não seria razoável em alguns casos, pois as tarefas aumentam com o aumento da capacidade computacional e a fração de código serial, na maioria dos casos, não cresce na mesma proporção. Isto levaria muitos algoritmos paralelos a terem sua fração serial reduzida com o aumento do número de processadores. A partir da observação de resultados práticos, Gustafson e Barsis sugeriram uma métrica alternativa à Lei de Amdahl, postulando assim a Lei de Gustafson-Barsis.

Dedução da Lei Gustafson-Barsis

Consideremos novamente a medida de *speedup* definida anteriormente:

$$S(p) \leq \frac{C(seq) + C(par)}{C(seq) + \frac{C(par)}{p}} \tag{2.14}$$

Se **f** for a fração da computação paralela realizada a executar computações sequenciais, então (1-f) é a fração do código que executa computações paralelas:

$$f = \frac{C(seq)}{C(seq) + \frac{C(par)}{p}} \tag{2.15}$$

e

$$(1 - f) = \frac{\frac{C(par)}{p}}{C(seq) + \frac{C(par)}{p}} \tag{2.16}$$

Logo:

$$C(seq) = f * \left(C(seq) + \frac{C(par)}{p} \right) \tag{2.17}$$

$$C(par) = p * (1 - f) * \left(C(seq) + \frac{C(par)}{p} \right) \tag{2.18}$$

Simplificando:

$$S(p) \leq \frac{(f + p * (1 - f)) * \left(C(seq) + \frac{C(par)}{p} \right)}{C(seq) + \frac{C(par)}{p}} \tag{2.19}$$

$$S(p) \leq f + p * (1 - f) \tag{2.20}$$

$$S(p) \leq p + f * (1 - p) \tag{2.21}$$

Seja $f \geq 0$ e $f \leq 1$ a fração da computação paralela realizada a executar computações sequenciais. A lei de Gustafson-Barsis diz-nos que o *speedup* máximo que uma aplicação paralela com p processadores pode obter é:

$$S(p) \leq p + f * (1 - p) \tag{2.22}$$

Ao usar o tempo de execução em paralelo como o ponto de partida, em lugar do tempo de execução sequencial, a lei de Gustafson-Barsis assume que a execução com um só processador é no pior dos casos **p** vezes mais lenta que a execução com **p** processadores. Isso pode não ser verdade se a memória disponível para a execução com um só processador for insuficiente face à computação realizada pelos **p** processadores. Por este motivo, o *speedup* estimado pela lei de Gustafson-Barsis é habitualmente designado por scaled *speedup*.

2.1.3 Métrica de Karp-Flatt

Uma outra métrica foi proposta por Karp e Flatt em 1990. A métrica de Karp-Flatt pode medir o grau de paralelização do código, o que é essencialmente útil para avaliar o limitante do fator de *speedup* num programa paralelo.

Considerando uma computação paralela com fator de *speedup* S usando **p** processadores ($p > 1$), a fração da tarefa serial determinada experimentalmente ϵ é dada por:

$$\epsilon = \frac{\frac{1}{S(p)} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2.23}$$

A fração serial determinada experimentalmente através da métrica de Karp-Flatt é uma função do fator de *speedup* observado e do número de processadores. Em sistemas que tenham um baixo fator de *speedup*, principalmente devido à limitante da fração serial do código paralelo, o aumento do número de processadores não refletirá diretamente sobre ϵ. Por outro lado, se o principal responsável pelo baixo fator de *speedup* for o aumento da comunicação entre os processadores, haverá um crescimento de ϵ com o aumento do número de processadores.

Dedução da Métrica da Karp-Flatt

$$T(1) = C(seq) + C(par) \tag{2.24}$$

$$T(p) = C(seq) + \frac{C(par)}{p} + C(com) \tag{2.25}$$

Seja ϵ a fração sequencial determinada experimentalmente duma computação paralela:

$$\epsilon = \frac{C(seq)}{T(1)} \tag{2.26}$$

Logo:

$$C(seq) = \epsilon * T(1) \tag{2.27}$$

$$C(par) = (1 - \epsilon) * T(1) \tag{2.28}$$

Se considerarmos que C(com) é desprezível, então:

$$T(p) = \epsilon * T(1) + \frac{(1 - \epsilon) * T(1)}{p} \tag{2.29}$$

Por outro lado:

$$S(p) = \frac{T(1)}{T(p)} \tag{2.30}$$

$$T(1) = S(p) * T(p) \tag{2.31}$$

Simplificando:

$$T(p) = \epsilon * S(p) * T(p) + \frac{(1-\epsilon) * S(p) * T(p)}{p} \tag{2.32}$$

$$1 = \epsilon * S(p) + \frac{(1-\epsilon) * S(p)}{p} \tag{2.33}$$

$$\frac{1}{S(p)} = \epsilon + \frac{(1-\epsilon)}{p} \tag{2.34}$$

$$\frac{1}{S(p)} = \epsilon + \frac{1}{p} - \frac{\epsilon}{p} \tag{2.35}$$

$$\frac{1}{S(p)} = \epsilon * \left(1 - \frac{1}{p}\right) + \frac{1}{p} \tag{2.36}$$

$$\epsilon = \frac{\frac{1}{S(p)} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2.37}$$

Seja S(p) o *speedup* duma aplicação paralela com $p > 1$ processadores. A métrica de Karp-Flatt diz-nos que a fração sequencial determinada experimentalmente é:

$$\epsilon = \frac{\frac{1}{S(p)} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2.38}$$

A métrica de Karp-Flatt é interessante porque ao desprezar o custo das operações de comunicação/sincronização/iniciação associadas à introdução de paralelismo numa aplicação, permite-nos determinar à posteriori qual a importância da componente C(com) no eventual decréscimo de eficiência da aplicação.

Por definição, a fração sequencial determinada experimentalmente é um valor constante que não depende do número de processadores.

$$\epsilon = \frac{C(seq)}{T(1)} \tag{2.39}$$

Por outro lado, a métrica de Karp-Flatt é uma função do número de processadores.

$$\epsilon = \frac{\frac{1}{S(p)} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2.40}$$

CAPÍTULO 3

Aplicação das métricas de desempenho do processamento paralelo

O princípio de Levinson foi originalmente aplicado para solucionar sistemas de Equações Normais simétricos Toeplitz (Levinson, 1947). A essência dos algoritmos tipo-Levinson é encontrar a solução de ordem j baseada na combinação linear das soluções direta e reversa de ordem $j-1$, as quais são linearmente independentes. Abaixo será mostrado como a recursão de Levinson pode ser extendida para solucionar um sistema real particionado de Equações Normais, não necessariamente bloco-Toeplitz. Este procedimento é uma generalização elaborada por Porsani et al(1991).

Seja um sistema sobredeterminado de equações lineares expresso por:

$$\mathbf{Gm} = \mathbf{d} \tag{3.1}$$

onde $\mathbf{m} = (\mathrm{m}_1,\ldots,m_N)^T$ é um vetor correspondente aos parâmetros do modelo com N elementos, $\mathbf{d} = (\mathrm{d}_1,\ldots,d_M)^T$ é um vetor correspondente aos dados observados com M elementos e $\mathbf{G}$ é uma matriz com M linhas e N colunas ($M > N$)

O sistema pode ser particionado da seguinte forma:

$$\left[\begin{array}{ccccc} \mathbf{G_1} & \cdots & \mathbf{G_j} & \cdots & \mathbf{G_L} \end{array}\right] \left[\begin{array}{c} \mathbf{m_1} \\ \vdots \\ \mathbf{m_j} \\ \vdots \\ \mathbf{m_L} \end{array}\right] = \mathbf{d} \tag{3.2}$$

onde as matrizes $\mathbf{G_j}$ tem M linhas e N_j colunas e os subvetores $\mathbf{m_j}$ tem N_j elementos. Para simplificar pode-se considerar $N_j = N_c$, sendo uma constante; $j = 1,\ldots, L$

Como consequencia do particionamento da matriz $\mathbf{G}$, o sistema de Equações Normais de ordem j correspondente pode ser mostrado abaixo:

$$\begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} -\mathbf{b_1} & \mathbf{A_{11}} & \cdots & \mathbf{A_{1j}} \\ \vdots & \vdots & \ddots & \vdots \\ -\mathbf{b_j} & \mathbf{A_{j1}} & \cdots & \mathbf{A_{jj}} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{m_{j1}} \\ \vdots \\ \mathbf{m_{jj}} \end{bmatrix} \tag{3.3}$$

onde $\mathbf{b_j} = \mathbf{G_j}^*\mathbf{d}$ são vetores com N_c elementos e $\mathbf{A_{i,j}} = \mathbf{G_i}^*\mathbf{G_j}$ são matrizes de ordem N_cxN_c .

O desenvolvimento de todas as operações algébricas deste método encontram-se no Apêndice A.

3.1 Particularização da solução de sistemas de ordem $j = 4$

Através do método geral para solucionar sistemas de equações normais tipo-bloco, foi particularizada uma situação de ordem $j = 4$ em que cada matriz $\mathbf{A_{ij}}$ é real, simétrica e tem ordem 10x10. Abaixo há a representação desta particularização, onde $\mathbf{b_j}$, $\mathbf{m_{4j}}$ (j variando de 1 até 4) são vetores que possuem 10 elementos cada:

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} -\mathbf{b_1} & \mathbf{A_{11}} & \mathbf{A_{12}} & \mathbf{A_{13}} & \mathbf{A_{14}} \\ -\mathbf{b_2} & \mathbf{A_{21}} & \mathbf{A_{22}} & \mathbf{A_{23}} & \mathbf{A_{24}} \\ -\mathbf{b_3} & \mathbf{A_{31}} & \mathbf{A_{32}} & \mathbf{A_{33}} & \mathbf{A_{34}} \\ -\mathbf{b_4} & \mathbf{A_{41}} & \mathbf{A_{42}} & \mathbf{A_{43}} & \mathbf{A_{44}} \end{bmatrix} \begin{bmatrix} \mathbf{1} \\ \mathbf{m_{41}} \\ \mathbf{m_{42}} \\ \mathbf{m_{43}} \\ \mathbf{m_{44}} \end{bmatrix} \tag{3.4}$$

Para análise desta particularização, desenvolveu-se uma implementação de código serial e outra de código paralelo para efeito de comparação dos tempos de execução e aplicação das leis que regem o processamento paralelo, as quais foram fundamentadas no capítulo anterior. Na figura 3.7 está representada a estratégia que foi adotada para resolver o problema utilizando multiprocessadores (cada retângulo presente nesta figura representa um processador ou nó).

Para o processamento paralelo foram utilizados 10 processadores do cluster Águia do CPGG-UFBA. Os processadores ou nós são identificados com os ranks (denominação comum em processamento paralelo) de 0 a 9. Abaixo está descrito como procede a troca de mensagens entre os processadores, utilizando-se para tal finalidade o padrão MPI:

- O processador 0 calcula $\mathbf{m_{11}}$ e o envia para o processador 4.
- O processador 1 computa os valores de ${}^{\mathbf{1}}\mathbf{H_{11}}$, ${}^{\mathbf{1}}\mathbf{E_{H1}}$ e os envia para o processador 5, o nó 1 também calcula os valores de ${}^{\mathbf{1}}\mathbf{F_{11}}$, ${}^{\mathbf{1}}\mathbf{E_{F1}}$ e os manda para o nó 4.

- O processador 2 calcula os valores de $\mathbf{^2H_{11}}$, $\mathbf{^2E_{H1}}$ e os envia para o processador 6, o nó 2 também calcula os valores de $\mathbf{^2F_{11}}$, $\mathbf{^2E_{F1}}$ e os manda para o nó 5.
- O nó 3 computa os valores de $\mathbf{^3F_{11}}$, $\mathbf{^3E_{F1}}$ e os manda para o nó 6.
- De acordo com os valores recebidos, o rank 4 calcula os valores de $\mathbf{m_{21}}$ e $\mathbf{m_{22}}$ e os envia para o processador 7.
- Conforme os valores recebidos dos ranks 1 e 2, o processador 5 computa os valores $\mathbf{^1H_{21}}$, $\mathbf{^1H_{22}}$, $\mathbf{^1E_{H2}}$ e os envia para o processador 8. O processador 5 também calcula os valores $\mathbf{^1F_{21}}$, $\mathbf{^1F_{22}}$, $\mathbf{^1E_{F2}}$ e os manda para o processador 7.
- Segundo os valores recebidos dos ranks 2 e 3, o processador 6 computa os valores $\mathbf{^2F_{21}}$, $\mathbf{^2F_{22}}$, $\mathbf{^2E_{F2}}$ e os manda para o processador 8.
- De acordo com valores recebidos, o rank 7 calcula os valores de $\mathbf{m_{31}}$, $\mathbf{m_{32}}$, $\mathbf{m_{33}}$ e os envia para o processador 9.
- Segundo os valores recebidos dos ranks 5 e 6, o processador 8 computa os valores $\mathbf{^1F_{31}}$, $\mathbf{^1F_{32}}$, $\mathbf{^1F_{33}}$ e $\mathbf{^1E_{F3}}$ e os manda para o processador 9.
- Por fim, o processador 9 calcula a solução do sistema de ordem $j = 4$ que é uma particularização do método desenvolvido por Porsani et al. A saber, a solução é $\mathbf{m_{41}}$, $\mathbf{m_{42}}$, $\mathbf{m_{43}}$ e $\mathbf{m_{44}}$.

3.2 Determinação do Speedup

Como visto no capítulo anterior, o valor do *speedup* mostra o quanto mais rápido um algoritmo paralelo é em relação ao seu correspondente sequencial, ou seja, o quanto ele é escalável. Expressando o *speedup* na forma de equação tem-se:

$$S = \frac{T(1)}{T(p)} \tag{3.5}$$

É de grande valia determinar o *speedup* do programa paralelo. Para isso, foram medidos os tempos de execução do programa serial como também do program paralelo.

O intervalo de tempo de execução do programa paralelo para o caso particular de ordem $j = 4$ é de 0.16s.

Já o intervalo de tempo de execução do programa serial para o mesmo caso particular de ordem $j = 4$ é de 0.36s

Então o *speedup* deste código paralelo é igual a:

```
real    0m0.017s
user    0m0.016s
sys     0m0.000s
```

Figura 3.1: Cronometragem do tempo de execução do programa paralelo

```
real    0m0.038s
user    0m0.036s
sys     0m0.000s
```

Figura 3.2: Cronometragem do tempo de execução do programa serial

$$S = \frac{0.36}{0.16} = 2.25 \tag{3.6}$$

Analisando este resultado, conclui-se que o programa paralelo resolve o mesmo problema do programa serial 2.25 vezes mais rápido.

3.3 Aplicação da Métrica de Karp-Flatt

Utilizando oportunamente a métrica de Karp-Flatt que foi explicada no capítulo anterior, pode-se determinar a fração do código que não é paralelizável. Para isso, utilizou-se o valor do *speedup* já calculado na seção anterior. Considerando um processamento paralelo com o fator de *speedup* S = 2.25, utilizando para isso 10 processadores, a fração do código paralelo que é estritamente serial, ou seja, não paralelizável é dada experimentalmente pelo valor de ϵ:

$$\epsilon = \frac{\frac{1}{S(p)} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{3.7}$$

$$\epsilon = \frac{\frac{1}{2.25} - \frac{1}{10}}{1 - \frac{1}{10}} = 0.3827 \tag{3.8}$$

Conclui-se que cerca de 38.27% do algoritmo não é paralelizável. A maneira representada na figura 3.7 de estruturar o problema dentro da ótica do processamento paralelo, apresenta 38.27% do seu código essencialmente serial. Isto significa que 38.27% do intervalo de tempo de execução do programa paralelo é gasto executando operações seriais. É um tanto quanto estranho ter operações seriais dentro dum código paralelo. Isto acontece devido às operações matemáticas que não podem ser executadas simultaneamente. Por exemplo, no processador 4 são computados os valores de $\mathbf{\Delta_{m1}}$ e $\mathbf{m_{22}}$. Mas para poder calcular $\mathbf{m_{22}}$

precisa-se do valor de $\mathbf{\Delta_{m1}}$, ou seja, estes dois valores não podem ser calculados de forma paralela (simultânea) e este fato contribui para a porcentagem de computações seriais que o programa paralelo efetua. É bastante oportuno ter a fração do código paralelo que é estritamente serial porque essa informação é imprescindível para poder aplicar a Lei de Amdahl.

3.4 Aplicação da Lei de Amdahl

Conforme a dedução e explicação no capítulo anterior, a Lei de Amdahl permite determinar o *speedup* máximo que um programa paralelo pode obter. A expressão matemática que define a Lei de Amdahl é a seguinte:

$$S(p) \leq \frac{1}{f + \frac{1-f}{p}} \tag{3.9}$$

sendo **f** a fração serial do programa paralelo e **p** a quantidade de processadores.

A aplicação da Lei de Amdahl ao programa paralelo em questão se dá do seguite modo:

$$S(p) \leq \frac{1}{0.3827 + \frac{1-0.3827}{p}} \tag{3.10}$$

A figura 3.3 representa o gráfico do valor do *speedup* máximo em função da quantidade de processadores.

Segundo o gráfico (figura 3.3) representativo da aplicação da Lei de Amdahl, para uma quantidade suficientemente grande de processadores, o ganho de desempenho obtido a partir do aumento desta quantidade não compensa. Observa-se que a curva aproxima-se de sua assíntota à medida que aumenta-se a quantidade de processadores. Conforme a Lei de Amdahl, o *speedup* máximo que este programa paralelo pode alcançar é de 2.6125, isto é, o modelo de estruturação deste programa paralelo que resolve um sistema de ordem $j = 4$ pode ser no máximo 2.6125 vezes mais rápido que o seu respectivo serial, segundo a Lei de Amdahl.

3.5 Aplicação da Lei de Gustafson-Barsis

A Lei de Amdahl parte do princípio de que por mais que aumente a quantidade de processadores, a fração serial (não-paralelizável) não diminui. Entretanto, a Lei de Gustafson-Barsis assume que a fração serial diminui com o aumento da quantidade de processadores. A expressão matemática que define a Lei de Gustafson-Barsis é a seguinte:

$$S(p) \leq p + f * (1 - p) \tag{3.11}$$

A aplicação da Lei de Gustafson-Barsis ao programa paralelo em questão se dá do seguite modo:

$$S(p) \leq 10 + 0.3827 * (1 - p) \tag{3.12}$$

A figura 3.4 representa o gráfico do valor do *speedup* máximo em função da quantidade de processadores, conforme a Lei de Gustafson-Barsis.

Através do gráfico da figura 3.4, percebe-se claramente uma dependência linear do *speedup* em relação à quantidade de processadores. Para que se possa visualizar as diferenças entre a Lei de Amdahl e a Lei de Gustafson-Barsis aplicado a este programa paralelo foi construído o gráfico que está representado na figura 3.5.

A estrutura do programa paralelo representado na figura 3.7 obedece a um determinado padrão. Este está relacionado com a quantidade total de processadores utilizados, a qual respeita uma série aritmética específica. A quantidade de processadores despendida para resolver este programa paralelo é $P_{total} = 4+3+2+1 = 10$. Ou seja, utilizando este algoritmo de programação paralela para resolver um sistema de ordem $j = 4$, são necessários 10 processadores. Percebe-se que para solucionar um sistema de ordem $j = 5$ precisa-se de $P_{total} = 5+4+3+2+1 = 15$ processadores. Um sistema de ordem $j = 6$ são necessários $P_{total} = 6+5+4+3+2+1 = 21$ processadores. Por indução, conclui-se que a quantidade de processadores P_{total} utilizada para executar o programa paralelo é $P_{total} = \sum_{\mathbf{k=0}}^{\mathbf{j-1}}(\mathbf{j} - \mathbf{k})$. Sendo j a ordem do sisema de blocos.

3.6 Previsão, através da Lei de Amdahl, dos tempos de execução do programa paralelo que soluciona sistemas de ordem supeior a $j = 4$

Surge a necessidade de obter o conhecimento do tempo de processamento paralelo não apenas de sistemas tipo-bloco de ordem $j = 4$, mas também de ordens superiores. Através da quantidade total de processadores necessários para executar o programa paralelo baseado no modo de programação ilustrado na figura 4.7 e na informação de sua parcela serial que é de 38.27%, pode-se prever teoricamente, por meio da Lei de Amdahl, o tempo de processamento do programa paralelo para solucionar um sistema de qualquer ordem j superior a 4. Para tal objetivo, é indispensável saber os valores dos tempos de execução do programa serial. Estes intervalos de tempo foram obtidos através do programa serial desenvolvido por Porsani et al. Este programa é capaz de fornecer a solução para um sistema de qualquer ordem j.

A figura 4.6 exibe os tempos de execução do programa serial e também a previsão dos tempos de execução do programa paralelo que soluciona sistemas com ordem que variam de $j = 75$ até $j = 120$.

A equação que representa o gráfico é a seguinte:

$$T(paralelo) = \frac{T(1) * (p * f - f + 1)}{p} \tag{3.13}$$

onde

- T(paralelo) é o tempo de execução do programa paralelo
- T(1) é o tempo de execução do programa serial
- f é a fração serial do código paralelo e
- p é a quantidade de processadores.

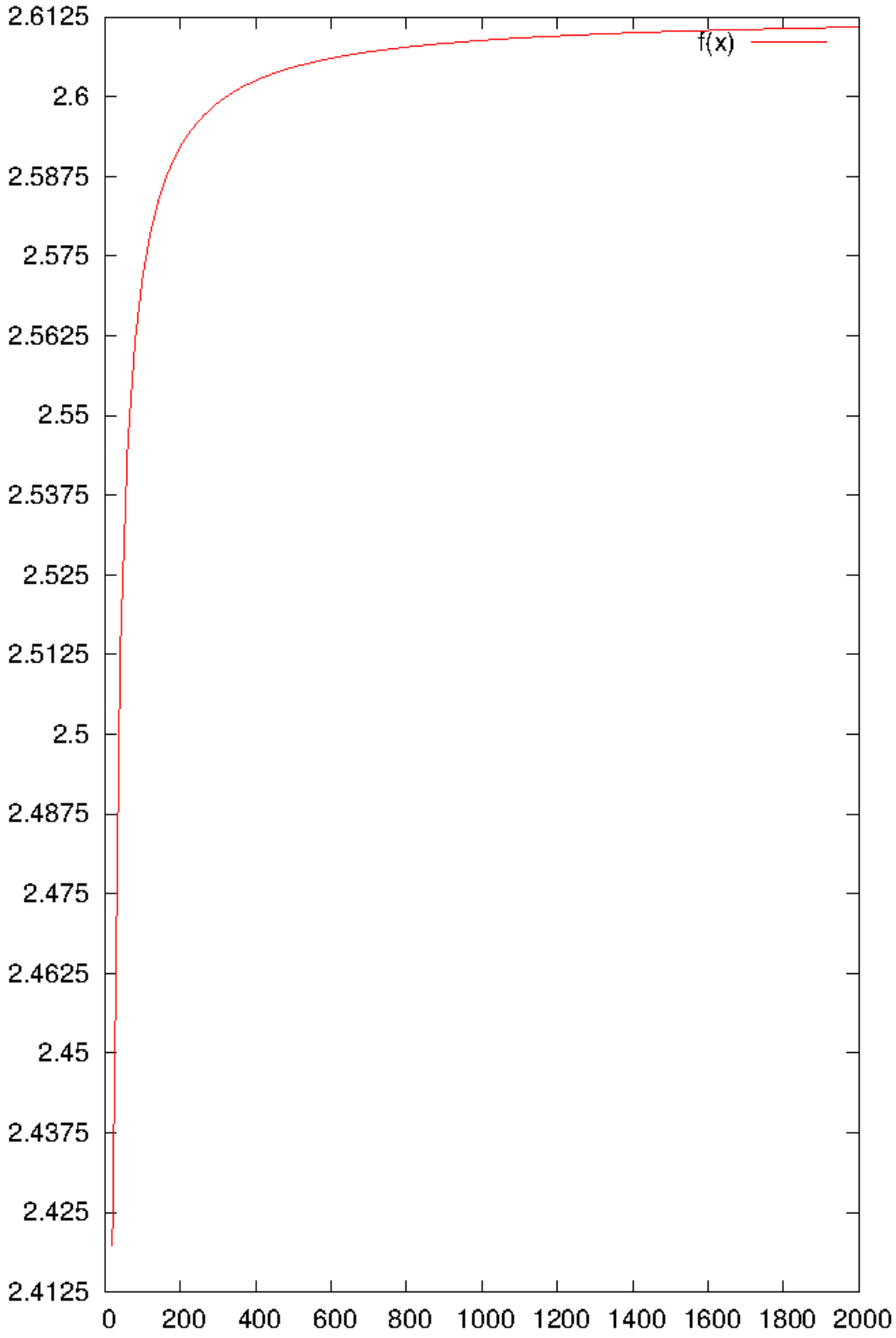

Figura 3.3: *Speedup* máximo do programa paralelo em função da quantidade de processadores.

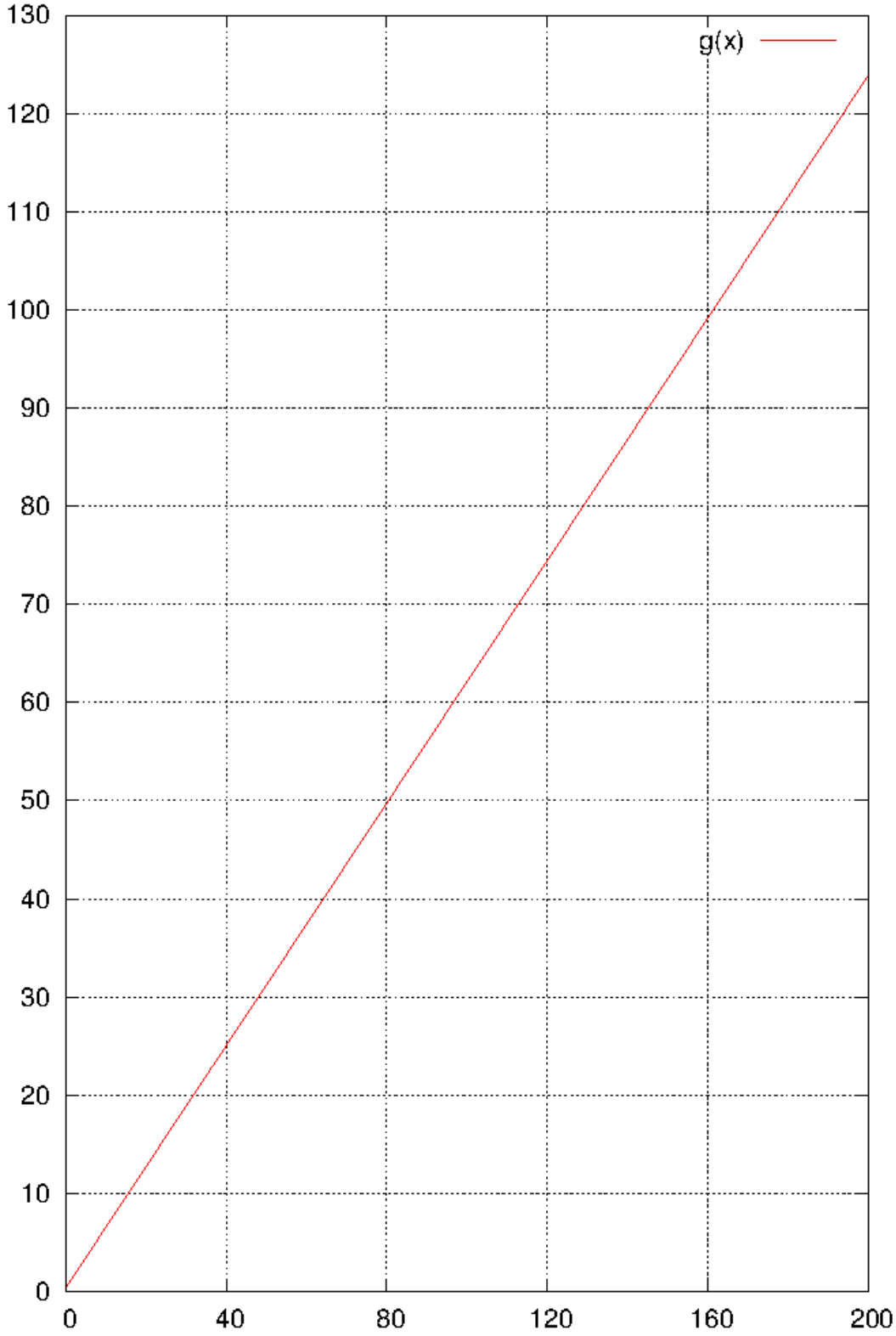

Figura 3.4: Gráfico que exibe a aplicação da Lei de Gustafson-Barsis em função da quantidade de processadores.

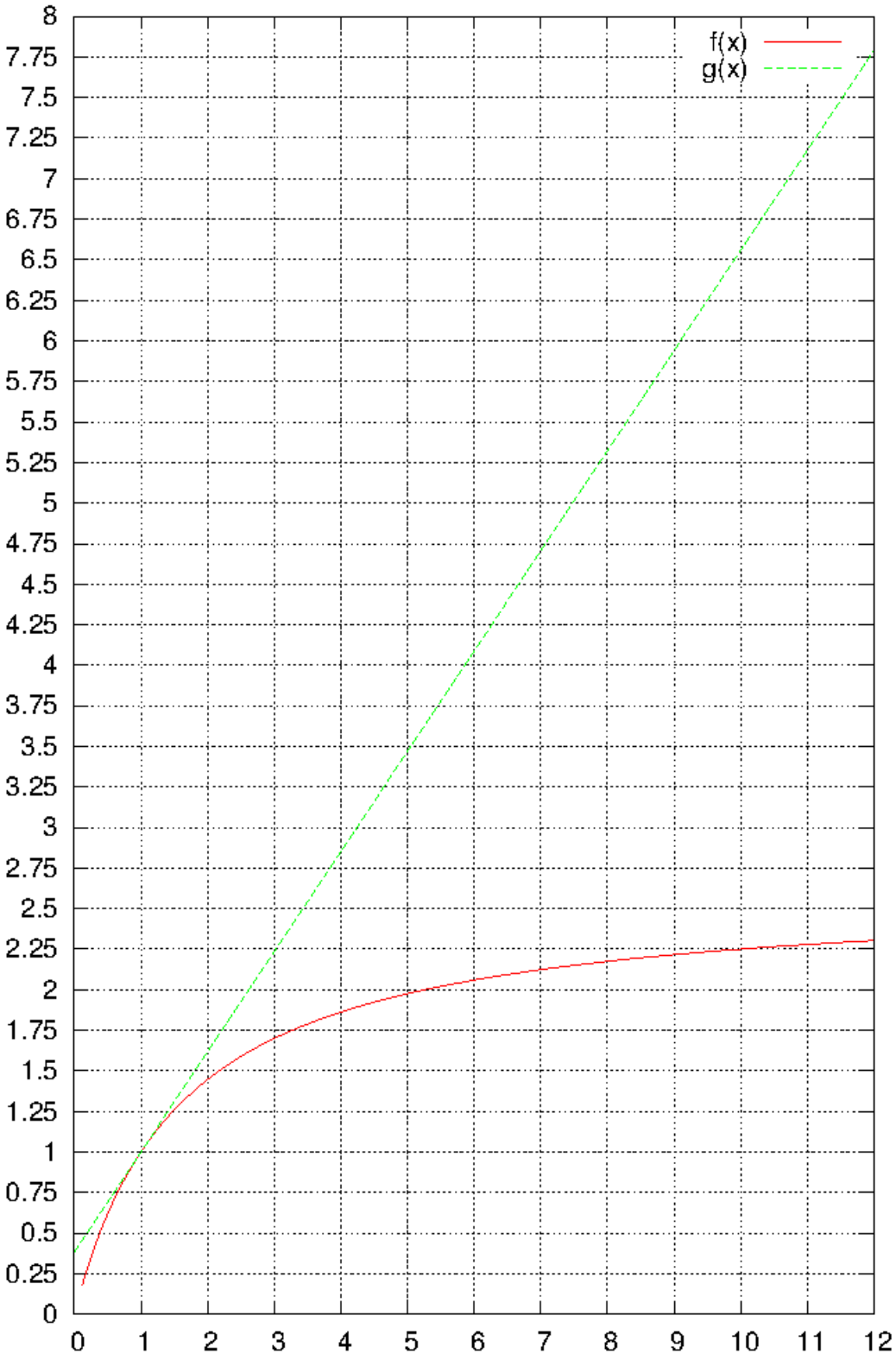

Figura 3.5: Gráfico que evidencia a comparação entre as curvas de Amdahl e de Gustafson-Barsis.

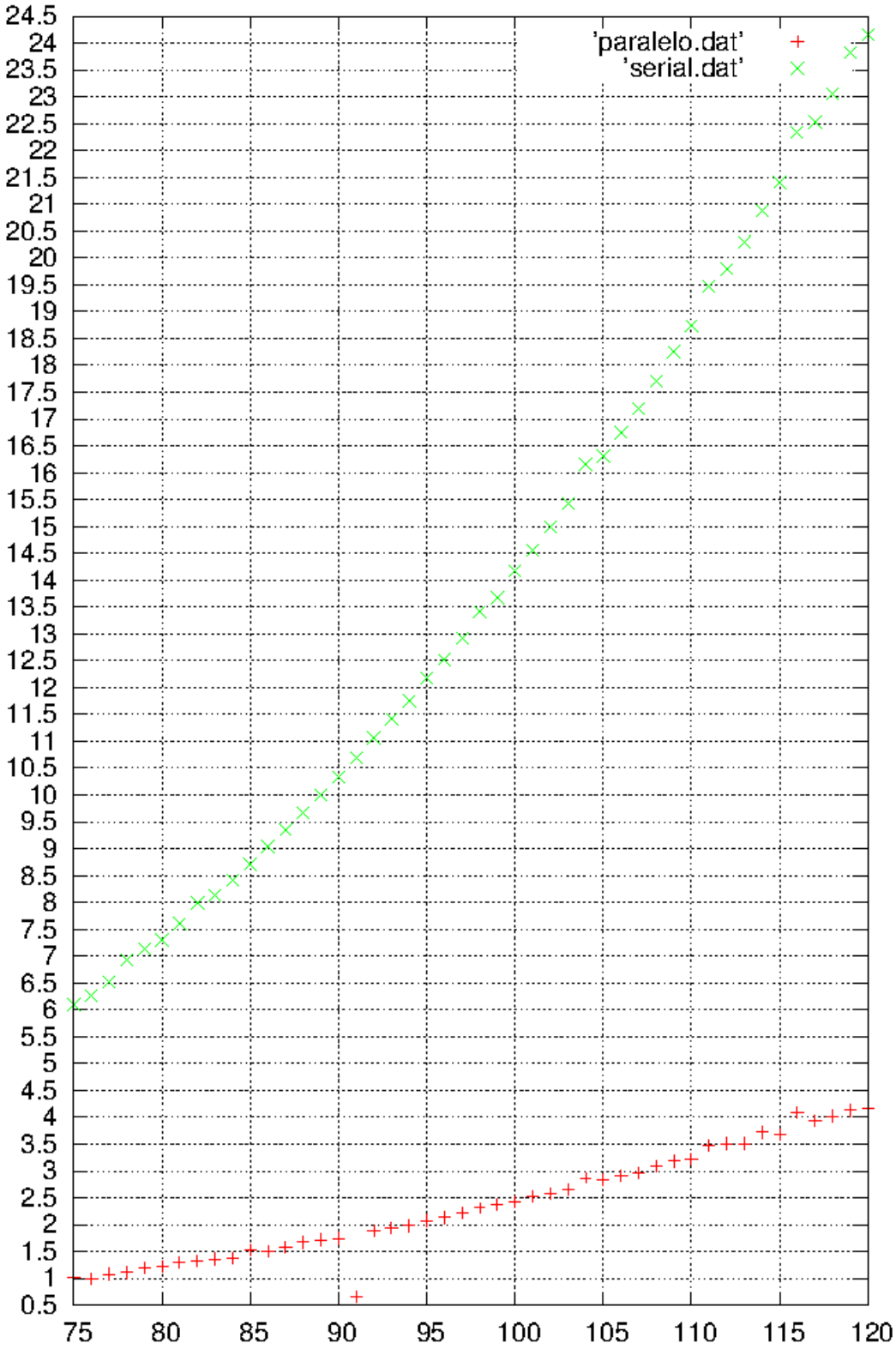

Figura 3.6: Gráfico que exibe a comparação dos tempos de execução do programa serial e do paralelo em função da ordem j.

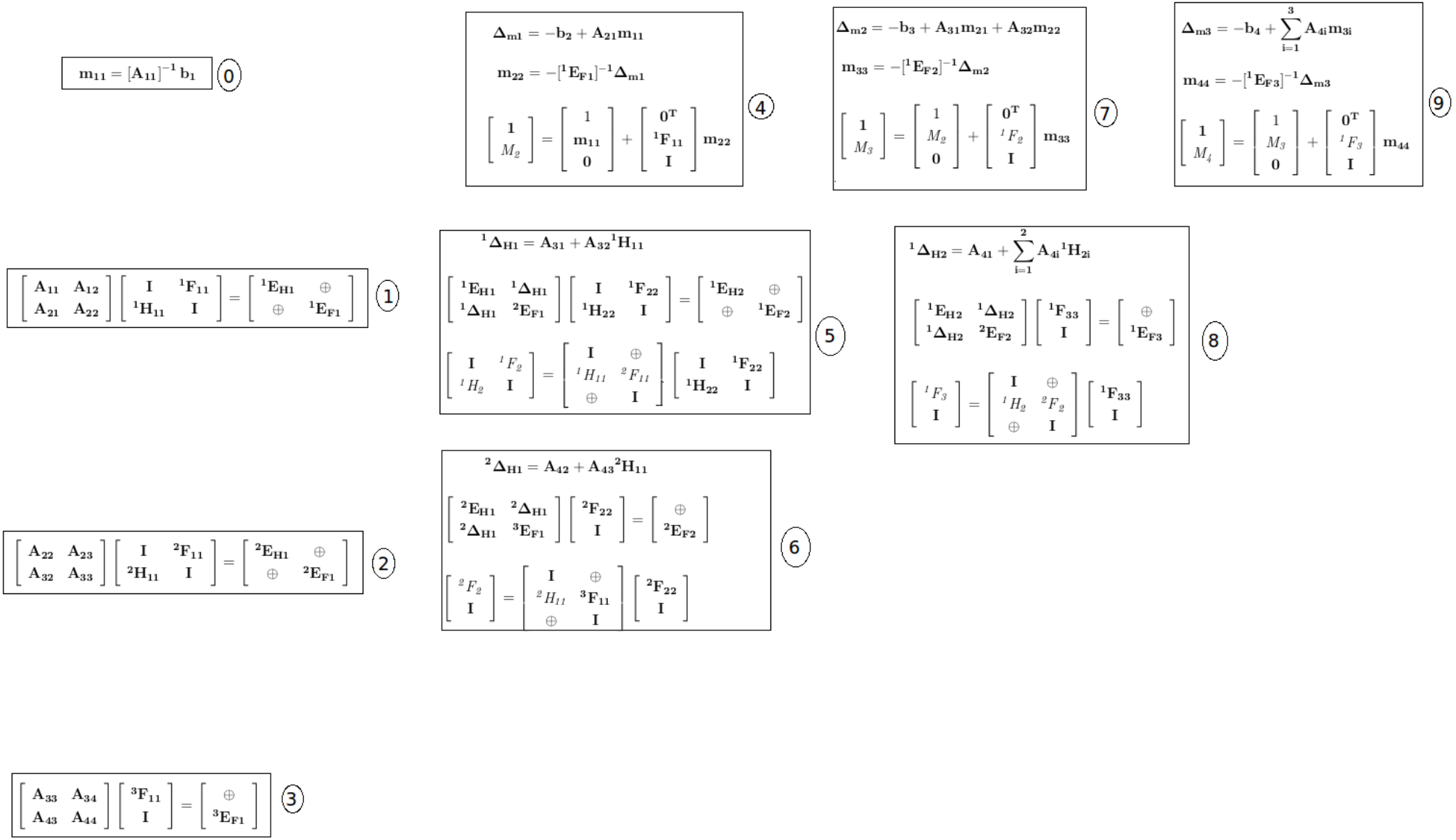

Figura 3.7: Estrutura do programa paralelo de ordem $j = 4$.

CAPÍTULO 4

Conclusões

A aplicação das métricas que avaliaram o desempenho do programa paralelo presente neste trabalho, baseadas na Lei de Amdahl e também na Lei de Gustafson-Barsis, forneceu resultados satisfatórios no que se refere à comparação dos tempos de execução entre o programa paralelo e o serial .

Entretanto, estes resultados seriam melhores se após o momento de envio das mensagens para outros processadores, o processador que fica responsável por enviar as mensagens não ficasse inativo. Por causa deste motivo, esta estrutura de programação paralela nao é ainda mais eficiente.

O programa paralelo desenvolvido neste trabalho resolve sistemas que apresentam matrizes $\mathbf{A_{ij}}$ de ordem 10, porque para ordens superiores houve uma limitação física de memória, pois este programa paralelo solicita muitos endereços de memória.

O recurso de poder efetuar a previsão dos tempos de execução do programa paralelo que efetua a computação das soluções dos sistemas de ordens superiores a $j = 4$ foi de grande valia para a compreensão do problema que envolve o processamento paralelo.

Sugere-se para eventuais trabalhos futuros que irão abordar este tema, a generalização do programa paralelo para qualquer ordem, afim de poder ter um meio de se comparar os resultados obtidos teoricamente através da Lei de Amdahl.

Agradecimentos

Gostaria de agradecer em primeiro lugar à JeovÃ¡ Deus por ter me dado forÇ§as durante todos esses anos fazendo com que eu chegasse até aqui.

Aos meu familiares, em especial, a meu pai Romildo, minha mãe LÃºcia e ao meu irmão Rodrigo, pelo amor, pelos ensinamentos, por estarem sempre presentes em todos os momentos, pela dedicaÇ§ão, empenho e pelo apoio em todoas as etapas de minha vida.

Ao meu orientador, Dr.Jos'e Milton Porsani, por todos os ensinamentos e orientaÃ§ões, que foram de fundamental importância no desenolvimento acadÃªmico e de todo o trabalho.

Aos professores do CPGG/UFBA pelo apoio e ensinamentos.

A Adriano, Joaquim Lago, MickelÂngelo pelo suporte em informática. E por estarem sempre dispostos à ajudar.

A Dona Ana e ao professor Alberto Brum por me ajudarem nas questÃ´es burocrÃ¡ticas envolvendo minha formatutra.

Aos meus colegas pela ajuda e incentivos durante toda minha formaÃ§Ã£o na Universidade.

Ao LAGEP-CPGG-UFBA pelo apoio técnico.

E por fim, à todos aqueles que colaboraram direta ou indiretamente para a conclusão deste trabalho.

APÊNDICE A

Extensão do Princípio de Levinson para solucionar um sistema particionado de Equações Normais

A.0.1 Método elaborado por Porsani et al. para solucionar um sistema de Equações Normais de ordem $j = 2$

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -\mathbf{b_1} & \mathbf{A_{11}} & \mathbf{A_{12}} \\ -\mathbf{b_2} & \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{m_{21}} \\ \mathbf{m_{22}} \end{bmatrix} \tag{A.1}$$

A recursão começa pela solução $\mathbf{m_{11}}$ e $^{\mathbf{1}}\mathbf{F_{11}}$ dos dois subsistemas de equações associados à primeira linha da equação acima:

$$\begin{bmatrix} -\mathbf{b_1} & \mathbf{A_{11}} & \mathbf{A_{12}} \end{bmatrix} \begin{bmatrix} 1 & \oplus \\ \mathbf{m_{11}} & {}^{\mathbf{1}}\mathbf{F_{11}} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \oplus \end{bmatrix} \tag{A.2}$$

onde $\oplus$ é uma matriz de ordem N_c N_c com elementos nulos e $\mathbf{I}$ é a matriz identidade de ordem N_c N_c. $\mathbf{m_{11}}$ e $^{\mathbf{1}}\mathbf{F_{11}}$ são as soluções direta e reversa de ordem 1. A recursão de Levinson de ordem 2 pode ser representada por:

$$\begin{bmatrix} 1 \\ \mathbf{m_{21}} \\ \mathbf{m_{22}} \end{bmatrix} = \begin{bmatrix} 1 & \oplus \\ \mathbf{m_{11}} & {}^{\mathbf{1}}\mathbf{F_{11}} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{m_{22}} \end{bmatrix} \tag{A.3}$$

Premultiplicando a equação acima pela matriz dos coeficientes de ordem 2 da equação 2.4 e considerando os resultados obtidos na equação 2.5, obtem-se:

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} -\mathbf{b_1} & \mathbf{A_{11}} & \mathbf{A_{12}} \\ -\mathbf{b_2} & \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix} \begin{bmatrix} 1 & \oplus \\ \mathbf{m_{11}} & {}^{\mathbf{1}}\mathbf{F_{11}} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{m_{22}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \oplus \\ \mathbf{\Delta_{m1}} & {}^{\mathbf{1}}\mathbf{E_{F1}} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{m_{22}} \end{bmatrix} \tag{A.4}$$

$$\mathbf{m_{22}} = -[^{\mathbf{1}}\mathbf{E_{F1}}]^{-\mathbf{1}}\mathbf{\Delta_{m1}} \tag{A.5}$$

Usando o resultado acima na equação 2.6, obtem-se a solução de ordem 2 para a equação 2.4.

A.0.2 Método elaborado por Porsani et al. para solucionar um sistema de Equações Normais de ordem *j + 1*

O sistema de ordem j+1 a ser resolvido é representado pela equação abaixo:

$$\begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{-b_1} & \mathbf{A_{11}} & \cdots & \mathbf{A_{1j}} & \mathbf{A_{1,j+1}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{-b_j} & \mathbf{A_{j1}} & \cdots & \mathbf{A_{jj}} & \mathbf{A_{j,j+1}} \\ \mathbf{-b_{j+1}} & \mathbf{A_{j+1,1}} & \cdots & \cdots & \mathbf{A_{j+1,j+1}} \end{bmatrix} \begin{bmatrix} \mathbf{1} \\ \mathbf{m_{j+1,1}} \\ \vdots \\ \mathbf{m_{j+1,j+1}} \end{bmatrix} \tag{A.6}$$

Dos j primeiros subsistemas da equação acima, pode-se formar dois sistemas de equações de ordem j,

$$\begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} \oplus \\ \vdots \\ \oplus \end{bmatrix} = \begin{bmatrix} \mathbf{-b_1} & \mathbf{A_{11}} & \cdots & \mathbf{A_{1j}} & \mathbf{A_{1,j+1}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{-b_j} & \mathbf{A_{j1}} & \cdots & \mathbf{A_{jj}} & \mathbf{A_{j,j+1}} \end{bmatrix} \begin{bmatrix} 1 \\ M_j \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{0^T} \\ {}^1F_j \\ \mathbf{I} \end{bmatrix} \tag{A.7}$$

onde $\mathbf{0^T}$ representa o vetor transposto de elementos nulos.

$$M_j = \begin{bmatrix} \mathbf{m_{j1}} \\ \vdots \\ \mathbf{m_{jj}} \end{bmatrix}, {}^1F_j = \begin{bmatrix} \mathbf{^1F_{jj}} \\ \vdots \\ \mathbf{^1F_{j1}} \end{bmatrix} \tag{A.8}$$

e $\mathbf{^1F_{ji}}$, $i = 1,\ldots,j$ são matrizes de ordem N_c N_c correspondentes à solução do subsistema reverso de ordem j representado na equação abaixo:

$$\begin{bmatrix} \oplus \\ \vdots \\ \oplus \end{bmatrix} = \begin{bmatrix} \mathbf{A_{11}} & \cdots & \mathbf{A_{1j}} & \mathbf{A_{1,j+1}} \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{A_{j1}} & \cdots & \mathbf{A_{jj}} & \mathbf{A_{j,j+1}} \end{bmatrix} \begin{bmatrix} {}^1F_j \\ \mathbf{I} \end{bmatrix} \tag{A.9}$$

O superscrito 1 em ${}^1F_j$ é usado para indicar a solução que tem $\mathbf{A_{11}}$ como o primeiro elemento da diagonal principal da matriz dos coeficientes. A solução da equação 2.9 pode ser obtida através da aplicação do princípio de Levinson: combinação linear das duas soluções independentes(direta e reversa) representadas na equação 2.10:

$$\begin{bmatrix} 1 \\ M_{j+1} \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0^T} \\ M_j & {}^1F_j \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{m_{j+1,j+1}} \end{bmatrix} \tag{A.10}$$

Substituindo a equação 2.13 na equação 2.9 e considerando os resultados obtidos na equação 2.10, obtem-se:

$$\mathbf{0} = \begin{bmatrix} -\mathbf{b_{j+1}} & \mathbf{A_{j+1,1}} & \cdots & \mathbf{A_{j+1,j+1}} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0^T} \\ M_j & {}^1F_j \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{m_{j+1,j+1}} \end{bmatrix} = \begin{bmatrix} \mathbf{\Delta_{mj}} & \mathbf{{}^1E_{Fj}} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{m_{j+1,j+1}} \end{bmatrix} \tag{A.11}$$

Analogamente à equação 2.7, os valores Δ_{mj} e $\mathbf{{}^1E_{Fj}}$ são obtidos pela multiplicação das soluções de ordem j com a últma linha da matriz,

$$\mathbf{\Delta_{mj}} = \begin{bmatrix} -\mathbf{b_{j+1}} & \mathbf{A_{j+1,1}} & \cdots & \mathbf{A_{j+1,j}} \end{bmatrix} \begin{bmatrix} 1 \\ M_j \end{bmatrix} \tag{A.12}$$

$$\mathbf{{}^1E_{Fj}} = \begin{bmatrix} \mathbf{A_{j+1,1}} & \cdots & \mathbf{A_{j+1,j+1}} \end{bmatrix} \begin{bmatrix} {}^1F_j \\ \mathbf{I} \end{bmatrix} \tag{A.13}$$

A solução da equação 2.14 pode ser representada pela equação abaixo:

$$\mathbf{m_{j+1,j+1}} = -[\mathbf{{}^1E_{Fj}}]^{-1}\mathbf{\Delta_{mj}} \tag{A.14}$$

Usando a equação acima na equação 2.13, obtem-se a solução da equação 2.9.

A.0.3 Método elaborado por Porsani et al. para obter as soluções direta e reversa dos subsistemas de Equações Normais

Inicia-se pela solução dos subsistemas direto e reverso de ordem 2X2 distribuídos ao longo da diagonal principal,

$$\begin{bmatrix} \mathbf{A_{ii}} & \mathbf{A_{i,i+1}} \\ \mathbf{A_{i+1,i}} & \mathbf{A_{i+1,i+1}} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{{}^iF_{11}} \\ \mathbf{{}^iH_{11}} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{{}^iE_{H1}} & \oplus \\ \oplus & \mathbf{{}^iE_{F1}} \end{bmatrix} \tag{A.15}$$

$\mathbf{{}^iE_{H1}}$ e $\mathbf{{}^iE_{F1}}$ são matrizes que representam as energias de erro do subsistema direto e reverso, respectivamente.

Os subsistemas direto e reverso de ordem j são representados abaixo:

$$\begin{bmatrix} \mathbf{A_{ii}} & \cdots & \mathbf{A_{i,i+j}} \\ \mathbf{A_{i+j,i}} & \cdots & \mathbf{A_{i+j,i+j}} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^{i}F_j \\ {}^{i}H_j & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{^{i}E_{Hj}} & \oplus_j \\ \oplus_j & \mathbf{^{i}E_{Fj}} \end{bmatrix} \tag{A.16}$$

onde

$${}^{i}H_j = \begin{bmatrix} \mathbf{^{i}H_{j1}} \\ \vdots \\ \mathbf{^{i}H_{jj}} \end{bmatrix}, \oplus_j = \begin{bmatrix} \oplus \\ \vdots \\ \oplus \end{bmatrix} \tag{A.17}$$

As soluções direta e reversa da equação 2.19 podem ser obtidas através da recursão de Levinson,

$$\begin{bmatrix} \mathbf{I} & {}^{i}F_j \\ {}^{i}H_j & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \oplus \\ {}^{i}H_{j-1} & {}^{i+1}F_{j-1} \\ \oplus & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{^{i}F_{jj}} \\ \mathbf{^{i}H_{jj}} & \mathbf{I} \end{bmatrix} \tag{A.18}$$

Premultiplicando equação 2.21 pela matriz dos coeficientes dado na quação 2.19 e considerando já conhecidos as soluções direta e reversa ${}^{i}H_{j-1}$ e ${}^{i+1}F_{j-1}$, obtem-se:

$$\begin{bmatrix} \mathbf{A_{ii}} & \cdots & \mathbf{A_{i,i+j}} \\ \mathbf{A_{i+j,i}} & \cdots & \mathbf{A_{i+j,i+j}} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^{i}F_j \\ {}^{i}H_j & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{^{i}E_{Hj-1}} & \mathbf{^{i+1}\Delta_{Fj-1}} \\ \oplus_{j-1} & \oplus_{j-1} \\ \mathbf{^{i}\Delta_{Hj-1}} & \mathbf{^{i+1}E_{Fj-1}} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{^{i}F_{jj}} \\ \mathbf{^{i}H_{jj}} & \mathbf{I} \end{bmatrix} \tag{A.19}$$

As matrizes $\mathbf{^{i}E_{Hj-1}}$, $\mathbf{^{i+1}\Delta_{Fj-1}}$, $\mathbf{^{i}\Delta_{Hj-1}}$, $\mathbf{^{i+1}E_{Fj-1}}$ são obtidas pela multiplicação das soluções ${}^{i}H_{j-1}$ e ${}^{i+1}F_{j-1}$ com a matriz dos coeficientes. Verifica-se que

$$\mathbf{^{i+1}\Delta_{Fj-1}} = \mathbf{^{i}\Delta_{Hj-1}} \tag{A.20}$$

onde

$$\mathbf{^{i}\Delta_{Hj-1}} = \mathbf{A_{i+j,i}} + \sum_{\mathbf{k=1}}^{\mathbf{j-1}} \mathbf{A_{i+j,i+k}} \mathbf{^{i}H_{j-1,k}} \tag{A.21}$$

Das equações 2.19 e 2.22, obtem-se uma forma compacta recursiva que é representada pela equação abaixo:

$$\begin{bmatrix} \mathbf{^{i}E_{Hj-1}} & \mathbf{^{i}\Delta_{Hj-1}} \\ \mathbf{^{i}\Delta_{Hj-1}} & \mathbf{^{i+1}E_{Fj-1}} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{^{i}F_{jj}} \\ \mathbf{^{i}H_{jj}} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{^{i}E_{Hj}} & \oplus \\ \oplus & \mathbf{^{i}E_{Fj}} \end{bmatrix} \tag{A.22}$$

onde $^{\mathbf{i}}\mathbf{E_{Hj}}$ e $^{\mathbf{i}}\mathbf{E_{Fj}}$ são matrizes que correspondem às energias do erro das soluções ${}^{i}H_{j}$ e ${}^{i}F_{j}$, respectivamente. A equação 2.25 tem que ser resolvida para $^{\mathbf{i}}\mathbf{H_{jj}}$ e $^{\mathbf{i}}\mathbf{F_{jj}}$:

$$\left[{}^{\mathbf{i+1}}\mathbf{E_{Fj-1}}\right] {}^{\mathbf{i}}\mathbf{H_{jj}} = -{}^{\mathbf{i}}\mathbf{\Delta_{Hj-1}}, \tag{A.23}$$

$$\left[{}^{\mathbf{i}}\mathbf{E_{Hj-1}}\right] {}^{\mathbf{i}}\mathbf{F_{jj}} = -{}^{\mathbf{i}}\mathbf{\Delta_{Hj-1}} \tag{A.24}$$

As soluções $^{\mathbf{i}}\mathbf{H_{jj}}$ e $^{\mathbf{i}}\mathbf{F_{jj}}$ são dadas abaixo:

$$^{\mathbf{i}}\mathbf{H_{jj}} = -\left[{}^{\mathbf{i+1}}\mathbf{E_{Fj-1}}\right]^{-\mathbf{1}} {}^{\mathbf{i}}\mathbf{\Delta_{Hj-1}} \tag{A.25}$$

$$^{\mathbf{i}}\mathbf{F_{jj}} = -\left[{}^{\mathbf{i}}\mathbf{E_{Hj-1}}\right]^{-\mathbf{1}} {}^{\mathbf{i}}\mathbf{\Delta_{Hj-1}} \tag{A.26}$$

E estas expressões podem ser utilizadas para atualizar as matrizes associadas às energias do erro

$$^{\mathbf{i}}\mathbf{E_{Hj}} = {}^{\mathbf{i}}\mathbf{E_{Hj-1}} + {}^{\mathbf{i}}\mathbf{\Delta_{Hj-1}}{}^{\mathbf{i}}\mathbf{H_{jj}} \tag{A.27}$$

$$^{\mathbf{i}}\mathbf{E_{Fj}} = {}^{\mathbf{i+1}}\mathbf{E_{Fj-1}} + {}^{\mathbf{i}}\mathbf{\Delta_{Hj-1}}{}^{\mathbf{i}}\mathbf{F_{jj}} \tag{A.28}$$

APÊNDICE B

Rotinas Básicas do MPI

Para um grande número de aplicações, um conjunto de apenas 6 subrotinas MPI serão suficientes para desenvolver uma aplicação no MPI.

Inicializar um processo MPI

MPI_INIT

- Primeira rotina do MPI a ser chamada por cada processo.

- Estabelece o ambiente necessário para executar o MPI.

- Sincroniza todos os processos na inicialização de uma aplicação MPI.

call MPI_INIT (mpierr)

mpierr: Variável inteira de retorno com o status da rotina.

mpierr=0, Sucesso

mpierr<0, Erro

Identificar processo do MPI

MPI_COMM_RANK

- Identifica o processo, dentro de um grupo de processos.

- Valor inteiro, entre 0 e n-1 processos.

call MPI_COMM_RANK (comm, rank, mpierr)

comm MPI communicator.

rank: Variável inteira de retorno com o número de identificação do processo.

mpierr: Variável inteira de retorno com o status da rotina.

Contar processos no MPI

MPI_COMM_SIZE

- Retorna o número de processos dentro de um grupo de processos.

call MPI_COMM_SIZE (comm, size, mpierr)

comm MPI Communicator.

size: Variável inteira de retorno com o número de processos inicializados durante uma aplicacação MPI.

mpierr: Variável inteira de retorno com o status da rotina

Enviar menssagens no MPI

MPI_SEND

- "Blocking send".

- A rotina retorna após o dado ter sido enviado.

- Após retorno, libera o "system buffer"e permite acesso ao "aplication buffer".

call MPI_SEND (sndbuf, count, datatype, dest, tag, comm, mpierr) sndbuf Endereço inicial do dados que será enviado. Endereço do "aplication buffer".

count: Número de elementos a serem enviados.

datatype: Tipo do dado.

dest: Identificação do processo destino.

tag: Rótulo da mensagem.

comm: MPI communicator.

mpierr: Variável inteira de retorno com o status da rotina.

Receber mensagens no MPI

MPI_RECV

- "Blocking receive".

- A rotina retorna após o dado ter sido recebido e armazenado.

- Após retorno, libera o "system buffer".

call MPI_RECV (recvbuf,count,datatype,source,tag,comm,status,mpierr)

recvbuf: Variável indicando o endereço do "aplication buffer".

count: Número de elementos a serem recebidos.

datatype: Tipo do dado.

source: Identificação da fonte.

tag: Rótulo da mensagem.

comm: MPI communicator.

status: Vetor com informações de source e tag.

mpierr: Variável inteira de retorno com o status da rotina.

Finalizar processos no MPI

MPI$_F INALIZE$

- Finaliza o processo para o MPI.

- Última rotina MPI a ser executada por uma aplicação MPI.

- Sincroniza todos os processos na finalização de uma aplicação MPI.

call MPI_FINALIZE (mpierr)

mpierr: Variável inteira de retorno com o status da rotina.

Referências Bibliográficas

Argonne National Laboratory; The Message Passing Interface (MPI) Standard. DisponÃvel em: ¡http://www-unix.mcs.anl.gov/mpi/¿.

Backus, M. M. (1959) Water reverberations - their nature and elimination, Geophysics, **24**(2):233-261.

Berryhill, J. R. e Kim, Y. C. (1986) Deep-water peg legs and multiples: emulation and suppression, Geophysics, **51**(12):2177-2184.

Bunch, A. W. H. e White, R. E. (1985) Least-squares filters without transient errors: an examination of the erors in least-squares filter design, Geophysical Prospecting, **33**:657-673.

Cohen, J. K. e Stockwell, J. W. (2008) The new SU user's manual, SEG & GRI, quarta edição.

Cruz, Ricardo Felipe Chartuni Cabral da (2010) Atenuação da reflexão múltipla do fundo marinho utilizando a deconvolução preditiva adaptativa, Tese de Mestrado, Universidade Federal da Bahia, Salvador-BA, Brasil.

Doicin, D. e Spitz, S. (1991) Multichannel extraction of water-bottom peg-legs pertraining to high-amplitude reflection, SEG Expanded Abstracts, **10**:1439-1442.

Introduction to Parallel Computing. DisponÃvel em: ¡http://www.llnl.gov/computing/tutorials/parallel$_c omp$/ > .

Kai Hwang, Zhiwei Xu; Scalable Parallel Computing, 1a EdiÃ§Ã£o, Mc Graw Hill, 1998.

Marple, S. L. (1980) A new autoregressive spectrum analusis algorithm, IEEE Transactions Acoustics, Speech and Signal Processing, **28**:441-454.

Mayne, W. H. (1962) Commom reflection point horizontal data stacking techniques, Geophysics, **27**(6):927-938.

Morf, M.; Dickson, B.; Kailath, T. e Vieira, A. (1977) Recursive solution of covariance equations for linear prediction, IEEE Transactions Acoustics, Speech and Signal Processing, **25**(12):429-433.

Parhami, B.; Introduction to Parallel Processing - Algorithms and Architectures. Plenum Series in Computer Science; Kluwer Academic Publishers; University of California at Santa Barbara; Santa Barbara, California 2002.

Park, I.; Parallel programming methodology and environment for the shared memory programming model, Dezembro 2000, Purdue University.

Peacock, K. L. e Treitel, S. (1969) Predictive deconvolution: Theory and practice, Geophysics, **34**:155-169.

Porsani, M. J. (1986) Desenvolvimento de algoritmos tipo-Levinson para o processamento de dados sísmicos, Tese de Doutorado, Universidade Federal da Bahia, Salvador-BA, Brasil.

Porsani, M. J. (1992) Efficient solution of covariance equations with applications to seismic trace extrapolation and preditive deconvolution, SEG Expanded Abstracts, **11**:1191-1194.

Porsani, M. J. e Ursin, B. (1991) Levinson-type extensions for non-toeplitz systems, IEEE Transactions Acoustics, Speech and Signal Processing, **39**:366-375.

Porsani, M. J. e Ursin, B. (2007) Direct multichannel predictive deconvolution, Geophysics, **72**(2):H11-H27.

Porsani, M. J. e Vetter, W. J. (1984) An optimal formulation for (levinson) recursive design of-lagged minimum energy filters, Proc. 54th Ann. Int. SEG Meeting, pp. 604-606.

Robinson, E. A. e Treitel, S. (1980) Geophysical signal analysis, Prentice-Hall, Englewood Cliffs.

Vershuur, D. J.; Berkhout, A. J. e apenaar, C. P. A. (1992) Adaptative surface-related multiple elimination, Geophysics, **57**(9):1166-1177.

Yilmaz, O. (2001) Seismic data analysis: processing, inversion and interpretation of seismic data, ol. 1 & 2, SEG, Tulsa-OK, USA, second eddition.

ANEXO I

Código do Programa Paralelo

```
      program firstmpi
!
! inclusao da biblioteca MPI
!
  use mpi
  integer mpierror,mpisize,mpirank
  integer mpistatus(3)
  REAL L(10,10),vfsol(10),m11(10),FSOL(10,10),F111(10,10),PROD(10,10),EF11(10,10)
  REAL H111(10,10),EH11(10,10),F211(10,10),EF21(10,10),vprod1(10),deltm1(10),Ndeltm1(1
  REAL m22(10),vprod2(10),m12(10),PROD1(10,10),DELTH11(10,10),F122(10,10),PROD2(10,10)
  REAL EF12(10,10),PROD3(10,10),F112(10,10),deltm2(10),m33(10),vprod3(10),m13(10),vpro
  REAL A11(10,10),b1(10),NA12(10,10),A22(10,10),A21(10,10),NA21(10,10),A12(10,10),NA23
  REAL A32(10,10),A33(10,10),Nb2(10),A31(10,10),Nb3(10),NDELTH11(10,10),Ndeltm2(10)
!
! inicializacao da MPI
!
      call MPI_Init(mpierror)
      call MPI_Comm_size(MPI_COMM_WORLD, mpisize, mpierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, mpirank, mpierror)
!
! processamento
!

!*****************************************************************
!
      if (mpirank.EQ.0) then

!-----------------------------------------------------------------
         call DECOMPL(10,A11,L)
         call VSOLF(10,L,b1,vfsol)
         call VSOLH(10,L,vfsol,m11)
         call MPI_Send(m11,10,MPI_REAL,4,1,MPI_COMM_WORLD,mpierror)
 !        do i=1,10
 !           print*,m11(i)
 !        enddo
 !       print*,'------------------------'
      endif
!
!*****************************************************************
!
      if (mpirank.EQ.1) then

!-----------------------------------------------------------------
         call DECOMPL(10,A11,L)
         call SOLF(10,L,NA12,FSOL)
         call SOLH(10,L,FSOL,F111)
         call GMPRD(A21,F111,PROD,10,10,10)
         do j=1,10
            do i=1,10
               EF11(i,j)=A22(i,j)+PROD(i,j)
```

```
            enddo
         enddo
         call MPI_Send(EF11,100,MPI_REAL,4,2,MPI_COMM_WORLD,mpierror)
         call MPI_Send(F111,100,MPI_REAL,4,3,MPI_COMM_WORLD,mpierror)

      endif
!
!*********************************************************************
!
      if (mpirank.EQ.2) then

!-------------------------------------------------------------
         call DECOMPL(10,A22,L)
         call SOLF(10,L,NA21,FSOL)
         call SOLH(10,L,FSOL,H111)
         call GMPRD(A12,H111,PROD,10,10,10)
         do j=1,10
            do i=1,10
               EH11(i,j)=A11(i,j)+PROD(i,j)
            enddo
         enddo
         call MPI_Send(EH11,100,MPI_REAL,5,4,MPI_COMM_WORLD,mpierror)
         call MPI_Send(H111,100,MPI_REAL,5,5,MPI_COMM_WORLD,mpierror)

      endif
!
!*********************************************************************
!
      if (mpirank.EQ.3) then

!-------------------------------------------------------------
         call DECOMPL(10,A22,L)
         call SOLF(10,L,NA23,FSOL)
         call SOLH(10,L,FSOL,F211)
         call GMPRD(A32,F211,PROD,10,10,10)
         do j=1,10
            do i=1,10
               EF21(i,j)=A33(i,j)+PROD(i,j)
            enddo
         enddo
         call MPI_Send(EF21,100,MPI_REAL,5,6,MPI_COMM_WORLD,mpierror)
         call MPI_Send(F211,100,MPI_REAL,5,7,MPI_COMM_WORLD,mpierror)

      endif
!
!*********************************************************************
!
      call MPI_Barrier(MPI_COMM_WORLD,mpierror)
!
!*********************************************************************
!
      if (mpirank.EQ.4) then

!-------------------------------------------------------------------------
         call MPI_Recv(m11,10,MPI_REAL,0,1,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(EF11,100,MPI_REAL,1,2,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F111,100,MPI_REAL,1,3,MPI_COMM_WORLD,mpistatus,mpierror)
         call GMPRD(A21,m11,vprod1,10,10,1)
         do i=1,10
            deltm1(i)=Nb2(i)+vprod1(i)
         enddo
         do i=1,10
```

```
            Ndeltm1(i)=(-1)*deltm1(i)
         enddo
         call DECOMPL(10,EF11,L)
         call VSOLF(10,L,Ndeltm1,vfsol)
         call VSOLH(10,L,vfsol,m22)
         call GMPRD(F111,m22,vprod2,10,10,1)
         do i=1,10
            m12(i)=m11(i)+vprod2(i)
         enddo
         call MPI_Send(m12,10,MPI_REAL,6,8,MPI_COMM_WORLD,mpierror)
         call MPI_Send(m22,10,MPI_REAL,6,9,MPI_COMM_WORLD,mpierror)

      endif
!
!*************************************************************************
!
      if (mpirank.EQ.5) then

!-------------------------------------------------------------------------
         call MPI_Recv(H111,100,MPI_REAL,2,5,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(EH11,100,MPI_REAL,2,4,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(EF21,100,MPI_REAL,3,6,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F211,100,MPI_REAL,3,7,MPI_COMM_WORLD,mpistatus,mpierror)
         call GMPRD(A32,H111,PROD1,10,10,10)
         do j=1,10
            do i=1,10
               DELTH11(i,j)=A31(i,j)+PROD1(i,j)
            enddo
         enddo
         do j=1,10
            do i=1,10
               NDELTH11(i,j)=(-1)*DELTH11(i,j)
            enddo
         enddo
         call DECOMPL(10,EH11,L)
         call SOLF(10,L,NDELTH11,FSOL)
         call SOLH(10,L,FSOL,F122)
         call GMPRD(DELTH11,F122,PROD2,10,10,10)
         do j=1,10
            do i=1,10
               EF12(i,j)=EF21(i,j)+PROD2(i,j)
            enddo
         enddo
         call GMPRD(H111,F122,PROD3,3,3,3)
         do j=1,10
            do i=1,10
               F112(i,j)=F211(i,j)+PROD3(i,j)
            enddo
         enddo
         call MPI_Send(EF12,100,MPI_REAL,6,10,MPI_COMM_WORLD,mpierror)
         call MPI_Send(F112,100,MPI_REAL,6,11,MPI_COMM_WORLD,mpierror)
         call MPI_Send(F122,100,MPI_REAL,6,12,MPI_COMM_WORLD,mpierror)

      endif
!
!*************************************************************************
!
 !     call MPI_Barrier(MPI_COMM_WORLD,mpierror)
!
!*************************************************************************
!
      if (mpirank.EQ.6) then
```

```
!-------------------------------------------------------------------------
         call MPI_Recv(m12,10,MPI_REAL,4,8,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(m22,10,MPI_REAL,4,9,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(EF12,10,MPI_REAL,5,10,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F122,10,MPI_REAL,5,12,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F112,10,MPI_REAL,5,11,MPI_COMM_WORLD,mpistatus,mpierror)
         call GMPRD(A31,m12,vprod1,10,10,1)
         call GMPRD(A32,m22,vprod2,10,10,1)
         do i=1,10
            deltm2(i)=Nb3(i)+vprod1(i)+vprod2(i)
         enddo
         do i=1,10
            Ndeltm2(i)=(-1)*deltm2(i)
         enddo
         call DECOMPL(10,EF12,L)
         call VSOLF(10,L,Ndeltm2,vfsol)
         call VSOLH(10,L,vfsol,m33)
         call GMPRD(F122,m33,vprod3,10,10,1)
         do i=1,10
            m13(i)=m12(i)+vprod3(i)
         enddo
         call GMPRD(F112,m33,vprod4,10,10,1)
         do i=1,10
            m23(i)=m22(i)+vprod4(i)
         enddo
         print*,'                                  '
         print*,'                                  '
         print*,'        PROCESSADOR DE NUMERO:',mpirank
         print*,'                                  '
         print*,'                                  '
         do i=1,10
            print*,m13(i)
         enddo
         print*,'-----------------------------------'
         do i=1,10
            print*,m23(i)
         enddo
         print*,'-----------------------------------'
         do i=1,10
            print*,m33(i)
         enddo
      endif
!
!*************************************************************************
      if (mpirank.EQ.7) then

!-------------------------------------------------------------------------
         call MPI_Recv(m12,10,MPI_REAL,4,8,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(m22,10,MPI_REAL,4,9,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(EF12,100,MPI_REAL,5,10,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F122,100,MPI_REAL,5,12,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F112,100,MPI_REAL,5,11,MPI_COMM_WORLD,mpistatus,mpierror)
         call GMPRD(A31,m12,vprod1,10,10,1)
         call GMPRD(A32,m22,vprod2,10,10,1)
         do i=1,10
            deltm2(i)=Nb3(i)+vprod1(i)+vprod2(i)
         enddo
         do i=1,10
            Ndeltm2(i)=(-1)*deltm2(i)
         enddo
         call DECOMPL(10,EF12,L)
```

```
         call VSOLF(10,L,Ndeltm2,vfsol)
         call VSOLH(10,L,vfsol,m33)
         call GMPRD(F122,m33,vprod3,10,10,1)
         do i=1,10
            m13(i)=m12(i)+vprod3(i)
         enddo
         call GMPRD(F112,m33,vprod4,10,10,1)
         do i=1,10
            m23(i)=m22(i)+vprod4(i)
         enddo
         print*,'                                  '
         print*,'                                  '
         print*,'        PROCESSADOR DE NUMERO:',mpirank
         print*,'                                  '
         print*,'                                  '
         do i=1,10
            print*,m13(i)
         enddo
         print*,'-----------------------------------'
         do i=1,10
            print*,m23(i)
         enddo
         print*,'-----------------------------------'
         do i=1,10
            print*,m33(i)
         enddo
      endif
!
!*********************************************************************
!
      if (mpirank.EQ.8) then

!---------------------------------------------------------------------------
         call MPI_Recv(m12,10,MPI_REAL,4,8,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(m22,10,MPI_REAL,4,9,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(EF12,100,MPI_REAL,5,10,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F122,100,MPI_REAL,5,12,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F112,100,MPI_REAL,5,11,MPI_COMM_WORLD,mpistatus,mpierror)
         call GMPRD(A31,m12,vprod1,10,10,1)
         call GMPRD(A32,m22,vprod2,10,10,1)
         do i=1,10
            deltm2(i)=Nb3(i)+vprod1(i)+vprod2(i)
         enddo
         do i=1,10
            Ndeltm2(i)=(-1)*deltm2(i)
         enddo
         call DECOMPL(10,EF12,L)
         call VSOLF(10,L,Ndeltm2,vfsol)
         call VSOLH(10,L,vfsol,m33)
         call GMPRD(F122,m33,vprod3,10,10,1)
         do i=1,10
            m13(i)=m12(i)+vprod3(i)
         enddo
         call GMPRD(F112,m33,vprod4,3,3,1)
         do i=1,10
            m23(i)=m22(i)+vprod4(i)
         enddo
         print*,'                                  '
         print*,'                                  '
         print*,'        PROCESSADOR DE NUMERO:',mpirank
         print*,'                                  '
         print*,'                                  '
```

```
         do i=1,10
            print*,m13(i)
         enddo
         print*,'-----------------------------------'
         do i=1,10
            print*,m23(i)
         enddo
         print*,'-----------------------------------'
         do i=1,10
            print*,m33(i)
         enddo
      endif
!
!*********************************************************************
      if (mpirank.EQ.9) then

!---------------------------------------------------------------------------
         call MPI_Recv(m12,10,MPI_REAL,4,8,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(m22,10,MPI_REAL,4,9,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(EF12,100,MPI_REAL,5,10,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F122,100,MPI_REAL,5,12,MPI_COMM_WORLD,mpistatus,mpierror)
         call MPI_Recv(F112,100,MPI_REAL,5,11,MPI_COMM_WORLD,mpistatus,mpierror)
         call GMPRD(A31,m12,vprod1,10,10,1)
         call GMPRD(A32,m22,vprod2,10,10,1)
         do i=1,10
            deltm2(i)=Nb3(i)+vprod1(i)+vprod2(i)
         enddo
         do i=1,10
            Ndeltm2(i)=(-1)*deltm2(i)
         enddo
         call DECOMPL(10,EF12,L)
         call VSOLF(10,L,Ndeltm2,vfsol)
         call VSOLH(10,L,vfsol,m33)
         call GMPRD(F122,m33,vprod3,10,10,1)
         do i=1,10
            m13(i)=m12(i)+vprod3(i)
         enddo
         call GMPRD(F112,m33,vprod4,10,10,1)
         do i=1,10
            m23(i)=m22(i)+vprod4(i)
         enddo
         print*,'                                  '
         print*,'                                  '
         print*,'        PROCESSADOR DE NUMERO:',mpirank
         print*,'                                  '
         print*,'                                  '
         do i=1,10
            print*,m13(i)
         enddo
         print*,'-----------------------------------'
         do i=1,10
            print*,m23(i)
         enddo
         print*,'-----------------------------------'
         do i=1,10
            print*,m33(i)
         enddo
      endif
!
!*********************************************************************
! finalizacao da MPI
!
```

```
      call MPI_Finalize(mpierror)
      end program firstmpi
!
!*********************************************************************
!
!     ###################################################################
!     #-----------------------------------------------------------------#
!     #--------------------SUBROTINAS REQUERIDAS----------------------#
!     #-----------------------------------------------------------------#
!     ###################################################################
      SUBROUTINE DECOMPL(n,a,l)
!
       integer n
       real l,a
       dimension l(n,n),a(n,n)
       integer i,j,s,t,k
       real c,d
!
      l(1,1)=sqrt(a(1,1))
      do i=2,n,1
         l(i,1)=a(i,1)/l(1,1)
      enddo
      c=0.0
      d=0.0
      do i=2,n,1
         do j=2,n,1
            do s=1,i-1,1
               c=c+l(i,s)*l(i,s)
            enddo
          l(i,i)=sqrt(a(i,i)-c)
          c=0.0
          do t=j+1,n,1
             do k=1,j-1,1
                d=d+l(t,k)*l(j,k)
             enddo
             l(t,j)=(1.0/l(j,j))*(a(t,j)-d)
             d=0.0
          enddo
          enddo
       enddo
      do i=1,n,1
         do j=1,n,1
            if (j.GT.i) then
            l(i,j)=0.0
            endif
         enddo
      enddo
      RETURN
      END
!---------------------------------------------------------------------
!---------------------------------------------------------------------
      SUBROUTINE SOLF(n,l,b,f)
!
      integer n
      real b,l,f
      dimension b(n,n),l(n,n),f(n,n)
      integer i,j,k
      real c
!
      do j=1,n,1
         f(1,j)=b(1,j)/l(1,1)
      enddo
```

```
      c=0.0
      do i=2,n,1
         do k=1,n,1
            do j=1,i-1,1
               c=c+l(i,j)*f(j,k)
             enddo
               f(i,k)=(b(i,k)-c)/l(i,i)
               c=0.0
         enddo
      enddo
      RETURN
      END
!-----------------------------------------------------------------------
!-----------------------------------------------------------------------
      SUBROUTINE SOLH(n,l,f,h)
!
      integer n
      real f,l,h
      dimension f(n,n),l(n,n),h(n,n)
      integer i,j,k
      real c
!
      do j=1,n,1
         h(n,j)=f(n,j)/l(n,n)
      enddo
      c=0.0
      do i=n-1,1,-1
         do k=1,n,1
            do j=n,i-1,-1
               c=c+l(j,i)*h(j,k)
            enddo
             h(i,k)=(f(i,k)-c)/l(i,i)
             c=0.0
         enddo
      enddo
      RETURN
      END
!-----------------------------------------------------------------------
!-----------------------------------------------------------------------
      SUBROUTINE VSOLF(n,l,b,f)
!
      integer n
      real b,l,f
      dimension b(n,1),l(n,n),f(n,1)
      integer i,j,k
      real c
!
      f(1,1)=b(1,1)/l(1,1)
      c=0.0
      do i=2,n,1
         do j=1,i-1,1
            c=c+l(i,j)*f(j,1)
         enddo
         f(i,1)=(b(i,1)-c)/l(i,i)
         c=0.0
      enddo
      RETURN
      END
!-----------------------------------------------------------------------
!-----------------------------------------------------------------------
      SUBROUTINE VSOLH(n,l,f,h)
!
```

```
      integer n
      real f,l,h
      dimension f(n,1),l(n,n),h(n,1)
      integer i,j,k
      real c
!
      h(n,1)=f(n,1)/l(n,n)
      c=0.0
      do i=n-1,1,-1
         do j=n,i-1,-1
            c=c+l(j,i)*h(j,1)
         enddo
      h(i,1)=(f(i,1)-c)/l(i,i)
      c=0.0
      enddo
      RETURN
      END
!-----------------------------------------------------------------------
!-----------------------------------------------------------------------
      SUBROUTINE GMPRD(A,B,R,N,M,L)
      DIMENSION A(1),B(1),R(1)
!
      IR=0
      IK=-M
      DO k=1,L
         IK=IK+M
         DO J=1,N
            IR=IR+1
            JI=J-N
            IB=IK
            R(IR)=0.0
            DO I=1,M
               JI=JI+N
               IB=IB+1
               R(IR)=R(IR)+A(JI)*B(IB)
            ENDDO
         ENDDO
      ENDDO
      RETURN
      END
```