IMPLEMENTAÇÃO E AVALIAÇÃO DE DESEMPENHO DE SOFTWARE LIVRE PARA COMPUTAÇÃO CIENTÍFICA EM AMBIENTE PARALELO

Júnia Almeida Matos

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Aprovada por:

Prof. Amit Bhaya, Ph.D.

Prof. Eugenius Kaszkurewicz, D.Sc.

Prof. Evaldo Simões da Fonseca, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2008

MATOS, JÚNIA ALMEIDA

Implementação e Avaliação de Desempenho de Software Livre para Computação em Ambiente Paralelo [Rio de Janeiro] 2008

XVIII, 100 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia Elétrica, 2008)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Avaliação de Desempenho de Programas em SCILAB

2. Desempenho do SCILAB Paralelo

3. SCILAB - PVM

I. COPPE/UFRJ II. Título (Série)

Agradecimentos

Em primeiro lugar, quero agradecer a Deus por ter me dado a oportunidade de concluir mais um trabalho. Ao professor Amit Bhaya por ter me aceitado como sua orientanda e me dando o suporte necessário para que essa dissertação pudesse chegar até ao fim. À Capes pelo suporte financeiro.

Aos professores Evaldo Simões da Fonseca e Eugenius Kaszkurewicz por aceitarem participar da banca de avaliação desta dissertação.

Aos meus pais Antonino e Maria Aparecida que sempre acreditaram em mim e que mesmo distante soube entender que era para o meu bem conseguir chegar até aqui. Muito obrigado por tudo, meus amores. Aos meus sobrinhos Gustavo, Thays e Rayssa que mesmo pequenos foram motivos de alegria para mim.

Aos meus amigos do Nacad, em especial Oumar Diene e Leonardo Valente que estiveram sempre me ajudando neste trabalho.

Ao meu amigo Bruno Rodrigues que se prontificou a me ajudar a buscar moradia para a minha permanência ao Rio.

À Dona Marlene e Sr. Marcílio que estiveram ao meu lado quando eu precisei de um ombro amigo.

Ao Luis Alvãrino, Denise Nishimura e Margarita Olazar e aos amigos do LN pelo carinho e amizade.

Aos meus avós queridos paternos (Antonio Machado e Amélia Terezinha) e

maternos (Jaider Domingos e Maria da Conceição) que sempre acreditaram em mim e souberam entender a distância e meu esforço quando não pude estar com eles nos fins de semana.

Ao professor André Gustavo que foi a minha inspiração para que eu chegasse até aqui.

Ao meu amigo e futuro esposo, David Boechat, que sempre esteve ao meu lado em todos os momentos.

A todos o meu muito obrigado!

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

IMPLEMENTAÇÃO E AVALIAÇÃO DE DESEMPENHO DE SOFTWARE LIVRE PARA COMPUTAÇÃO CIENTÍFICA EM AMBIENTE PARALELO

Júnia Almeida Matos

Março/2008

Orientador: Amit Bhaya

Programa: Engenharia Elétrica

Esta dissertação apresenta a implementação do software para computação científica chamado SCILAB, cujo código fonte é aberto e gratuitamente disponibilizado pelo consórcio SCILAB, LAPACK e BLAS que são softwares de domínio público para computação paralela científica disponível pelo repositório de dados do NETLIB. SCILAB é uma linguagem interpretada que, em sua versão 4.0 utilizada neste trabalho, contém uma biblioteca para Parallel Virtual Machine*(PVM)*, que possibilita o uso de primitivos de comunicação em um ambiente de computação paralela, composto especificamente de um computador *SGI Altix 350* com 14 processadores, baseados em processadores *Intel Itanium II* executando *RedHat Enterprise Linux + SGI ProPack*.

Adicionalmente a integração de funções destes pacotes de software, SCILAB, LAPACK e BLAS, em um ambiente de computação paralela foram desenvolvidos estudos comparativos em problemas típicos de computação científica: multiplicação de matrizes, soluções de sistemas lineares esparsos e ordenação, de forma a aferir a viabilidade desta plataforma integrada com um instrumento para prototipagem rápida de programas paralelos.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

IMPLEMENTATION AND EVALUATION OF OPEN SOURCE SOFTWARE FOR SCIENTIFIC COMPUTATION IN A PARALLEL ENVIRONMENT

Júnia Almeida Matos

March/2008

Advisor: Amit Bhaya

Department: Electrical Engineering

This dissertation presents the implementation of the software for scientific computation called SCILAB, which is open source and freely available from the SCILAB consortium repository, and LAPACK and BLAS which are public domain software for parallel scientific computation available from the NETLIB repository. SCILAB is an interpreted language and in its version 4.0, used in this dissertation, contains a toolbox for the Parallel Virtual Machine*(PVM)*, which makes possible the use of communication primitives in a parallel computing environment, specifically an *SGI Altix 350*, with fourteen processors, based on the *Intel Itanium II* processor, running *RedHat Enterprise Linux + SGI ProPack*. In addition to integrating the functions in these software packages, SCILAB, LAPACK and BLAS, in a parallel computing environment, comparatives studies on typical problems of scientific computation, namely matrix multiplication, solution of sparse linear systems and sorting, are carried out in order to test the feasibility of integrated platform as an instrument for the rapid prototyping of parallel programs.

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrônimos e Notação utilizados

PVM $\Longrightarrow$ Parallel Virtual Machine.

MPI $\Longrightarrow$ Message Passing Interface.

BLAS $\Longrightarrow$ Basic Linear Algebra Subprograms.

LAPACK $\Longrightarrow$ Linear Algebra PACKage.

BLACS $\Longrightarrow$ Basic Linear Algebra Communication Subprograms.

PBLAS $\Longrightarrow$ Parallel BLAS.

ScaLAPACK $\Longrightarrow$ Scalable LAPACK.

NETLIB $\Longrightarrow$ é um repositório de softwares matemáticos, artigos e base de dados.

Fortran $\Longrightarrow$ FORmula TRANslation.

API $\Longrightarrow$ Aplication Programming Interfaces.

nprocs $\Longrightarrow$ Número de processadores.

n $\Longrightarrow$ Dimensão de uma matriz.

NUMA $\Longrightarrow$ Non Uniform Access Memory.

SMP $\Longrightarrow$ Symmetric MultiProcessors.

Speedup $\Longrightarrow$ Aceleração.

Tp $\Longrightarrow$ Tempo de processamento em paralelo.

Ts $\Longrightarrow$ Tempo de processamento de um algoritmo em seqüêncial.

EDP $\Longrightarrow$ Equação Diferencial Parcial.

tol $\Longrightarrow$ corresponde à tolerância

maxiter $\Longrightarrow$ corresponde ao máximo de iterações

$P_0 \quad ... \quad P_{n-1} \Longrightarrow$ corresponde a cada processador.

Capítulo 1

Introdução

Scilab é um ambiente para desenvolvimento e prototipação de software numérico de propósito geral, gratuito e distribuído com o código fonte. Além da distribuição com o código fonte, existem distribuições pré-compiladas para vários sistemas operacionais. Ele foi criado em 1990 por um grupo de pesquisadores do *INRIA-Institute Recherche en Informatique et en Automatique e do ENPC - École Nationale des Ponts et Chaussées* [1], [2].

A computação paralela tem como objetivo usar simultaneamente um conjunto de processadores interligados, de modo a resolver um problema conjuntamente, com o tempo de processamento menor do que um processo seqüencial (utilização de somente um processador). Um computador paralelo é uma coleção de processadores interconectados de maneira a permitir a coordenação de suas atividades e a troca de dados.

O PVM (*Parallel Virtual Machine*) é um pacote de software que permite que uma coleção heterogênea de computadores ligados em rede se apresente para o usuário como uma máquina virtual única oferecendo facilidades para criação e finalização de tarefas a serem executadas em paralelo. No PVM a configuração de máquina virtual permite a adição e a eliminação de hosts que são elementos de processamento, assim como o envio e a recepção de mensagens entre tarefas em diversos formatos.

O pacote Scilab inclui uma ferramenta (*toolbox*) que permite a utilização conjunta da linguagem PVM na paralelização dos algoritmos utilizados por ele.

1.1 Objetivos

Este trabalho visa implementar o software Scilab ao ambiente de uma máquina multiprocessador do NACAD (*Núcleo de Atendimento em Computação de Alto Desempenho*) com os seguintes objetivos:

- Criar um ambiente de programação de alto nível (Linguagem Interpretada Scilab) para facilitar prototipagem rápida de programas de computação científica;
- Permitir, ao mesmo tempo, que seja possível utilizar as primitivas de comunicação necessárias para computação paralela, dentro do mesmo ambiente de programação de alto nível, desta forma promovendo a portabilidade do código desenvolvido para a linguagem nativa (compilada) da máquina multiprocessador;
- Avaliar o desempenho paralelo da máquina trabalhando no modo subótimo utilizando a linguagem interpretada juntos às primitivas de comunicação para os efeitos de prototipagem rápida de códigos paralelos.

1.2 Organização do trabalho

Este trabalho está dividido em seis capítulos da seguinte forma:

O primeiro capítulo apresenta a introdução e os objetivos da presente dissertação, destacando alguns conceitos de computação paralela.

No segundo capítulo são apresentados aspectos gerais da computação paralela, composição e modos de utilização do sistema descrevendo as principais *API* e bib-

liotecas empregadas nos principais problemas típicos de computação científica aqui abordado, principalmente as respectivas funcionalidades.

No terceiro capítulo é apresentado o problema típico de computação científica, *multiplicação de matrizes*, juntamente com dois métodos de particionamento e utilização em paralelo e as análises de *Speedup* decorridos através do tempo de processamento para até oito processadores.

No quarto capítulo é discutido o método iterativo de Jacobi para resolução de sistemas lineares de grande porte fazendo menção às principais características do sistema. Adicionalmente é apresentado também, a resolução de sistemas lineares tanto sequencial quanto paralelo verificando o desempenho correspondente à implementação do método.

No quinto capítulo é discutido o método de ordenação *Quicksort* em paralelo, utilizando o modelo hipercúbico, que é uma forma de comunicação entre processadores utilizando topologias de rede. Os processadores são rotulados com números binários, e as comunicações utilizam esses rótulos (dos processadores), para a interação e reciprocidade no compartilhamento de mensagens e concatenação de vetores dos processadores vizinhos.

No sexto capítulo são apresentados as conclusões do trabalho e sugestões para trabalhos futuros.

Capítulo 2

Computação Paralela

2.1 Introdução

Este capítulo aborda alguns conceitos importantes na programação paralela, e apresenta uma breve explicação das bibliotecas que serão utilizadas nos próximos capítulos. Especificamente, a seção 2.2 introduz técnicas paralelas. A seção 2.3 recapitula resumidamente as questões relativas à arquitetura de memória e ao acesso a mesma, no contexto de computação paralela. A avaliação de desempenho de algoritmos paralelos é abordada na seção 2.4. Os interpretadores utilizados são introduzidos na seção 2.5. A seção 2.6 descreve a biblioteca *Parallel Virtual Machine (PVM)*, bem como os tipos de comunicação existentes no SCILAB paralelo. Finalmente, a seção 2.7 descreve as bibliotecas de computação científicas utilizadas nos problemas típicos de computação científica avaliados ao longo desta dissertação.

2.2 Programação Paralela

Os computadores paralelos surgiram na década de 80, quando os pesquisadores buscavam encontrar soluções cada vez mais rápidas e que resolvessem problemas com grande quantidade de dados. Em áreas como previsão do tempo, processamento de

imagens, inteligência artificial, modelagem e simulação de grandes sistemas é o que têm impulsionado o desenvolvimento de tecnologias na área da computação paralela.

Pensando em um melhor desempenho por unidade de custo, criou-se o *cluster*, que é um conjunto de máquinas conectadas através de uma rede de alta velocidade, onde é possível a execução de aplicações tanto seqüenciais quanto paralelas. Além disso o *cluster* é considerado como sendo um conjunto de estações de trabalho ou sistemas com múltiplas CPU, ou ainda um conjunto de nós em um computador paralelo.

O paralelismo em máquinas paralelas são gerenciados através de *API (Aplication Programming Interfaces)* como *PVM e MPI*. Dependendo da arquitetura da máquina, o paralelismo pode ser obtido através da criação de *processos ou threads*, definidos como se segue:

Processos são um conjunto de instruções que executam uma tarefa, e que não compartilham espaço de memória e, principalmente recursos alocados pelo sistema operacional, enquanto que, o processador é o dispositivo físico de hardware responsável por executar processos.

Threads são pequenas partes de um processo. Cada processo possui pelo menos um *thread*, que é chamado de thread mestre e diversos threads podem existir em um processo. Os *threads* compartilham entre eles os recursos alocados pelo sistema operacional para o processo, como o espaço de memória, o que os torna adequados à programação paralela em sistemas de memória compartilhada.

Nos sistemas de *memória compartilhada*, o paralelismo pode ser gerenciado pelo *OpenMP (Open Multi Processing)*, que consiste de especificações e interfaces para paralelização de um programa que funciona com a criação de um único processo com vários threads paralelos. Geralmente recomenda-se que o número de threads seja igual ao número de processadores disponíveis, a fim de evitar que mais de um thread seja executado em mais de um processador.

Em sistemas de memória distribuída, as API comumente utilizadas são as de

passagem por mensagem, como *PVM (Parallel Virtual Machine)* e *MPI (Message Passing Interface)*. Essas API são responsáveis por criar em cada processador processos que: executam as tarefas de forma independente; gerenciam a comunicação entre os processos criados e também realizam a finalização destes processos.

Para que um programa seja executado em paralelo é necessário paralelizar determinada tarefa onde seja possível a identificação de porções de código que podem ser executadas independentemente. Esses trechos de códigos são partes de um problema que podem ser processados independentemente e que trocam informações em tempo em tempo de acordo com a sincronização dos processos. A comunicação entre os processadores é avaliada pela granularidade.

A *granularidade* segundo Moretti [3], Freeman [4] é a quantidade de processamento realizado por cada processador, em relação à quantidade de comunicação entre os processadores. Quando os processadores executam poucas instruções e comunicam muito é dito que o programa é muito granular, caso contrário, é pouco granular.

As subseções seguintes 2.2.1 e 2.2.2 apresentam dois paradigmas de programação paralela que serão utilizados nesta dissertação: o paradigma mestre-escravo e o paradigma dividir-e-conquistar.

2.2.1 Modelo Mestre e Escravo

O modelo *Mestre-Escravo* de computação paralela consiste em duas entidades: processador mestre e múltiplos processadores escravos. O papel do mestre é realizar a decomposição do problema em pequenas tarefas, distribuir estas tarefas entre os escravos, e coletar os resultados parciais obtidos. Esses resultados quando coletados compõem a solução do problema. Os processos escravos recebem as mensagens com a tarefa, processam esta tarefa e enviam o resultado ao mestre. Geralmente a comunicação é efetuada apenas entre mestre e escravos. Uma representação deste modelo pode ser visualizada na Figura 2.1.

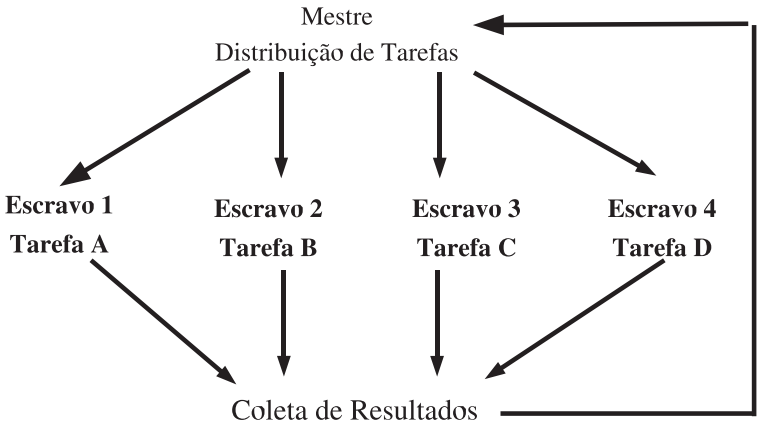

Figura 2.1: Representação Mestre Escravo.

2.2.2 Técnica da Divisão e Conquista

A técnica de Dividir para Conquistar consiste em distribuir uma grande tarefa em dois ou mais subproblemas para cada processador, de forma que, os subproblemas admitam resolução independente e seus resultados parciais possam ser combinados chegando ao resultado final. Esses subproblemas são instâncias do problema original dando origem a uma execução recursiva. A representação desse modelo pode ser visualizado na Figura 2.2.

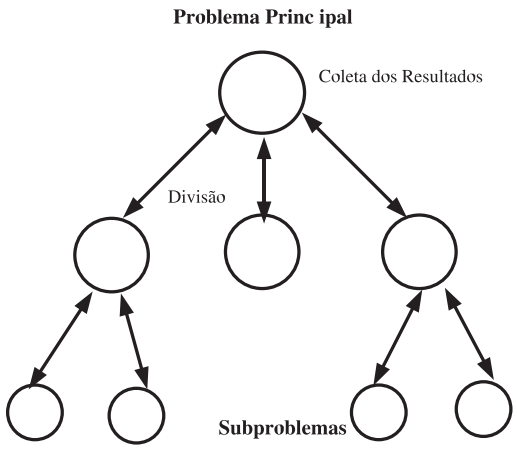

Figura 2.2: Representação do Modelo Divisão e Conquista

2.3 Arquitetura de Acesso à Memória

Para coordenar as tarefas dos vários processadores que trabalham na solução de um mesmo problema é necessária alguma forma de comunicação entre eles para transportar informações e dados entre os processadores, e para a sincronização de tarefas.

Em máquinas com múltiplos processadores, cada processador possui uma identificação que é um número, que inicia em *0* e finaliza em *p-1*, onde *p* é o número de processadores existentes na máquina. As máquinas que possuem múltiplos processadores, segundo a taxonomia de FLYNN, pertencem, à categoria *MIMD - Multiple Instruction Multiple Data.* As máquinas de múltiplos processadores, de acordo com a forma de acesso à memória, podem ser classificadas como:

- Sistemas de memória compartilhada;
- Sistemas de memória distribuída ou de memória local;
- Sistemas híbridos;

A forma de comunicação e sincronização entre os processadores depende do modo de acesso à memória e afeta diretamente a maneira como os programas paralelos são implementados.

Nas arquiteturas de memória compartilhada têm-se múltiplos processadores operando independentemente, mas que compartilham o mesmo espaço de endereçamento,ou seja, um único bloco de memória como mostrado na Figura 2.3.

Os sistemas de memória compartilhada podem ainda ser classificados como SMP *(Symmetric MultiProcessors)* quando possuem processadores idênticos e acesso uniforme à memória ou NUMA (*Non Uniform Access Memory*) quando o acesso à memória não é uniforme [5].

Nas arquiteturas com Memórias Distribuídas, têm-se múltiplos processadores independentes, mas cada um possui a sua própria memória que é acessada somente por

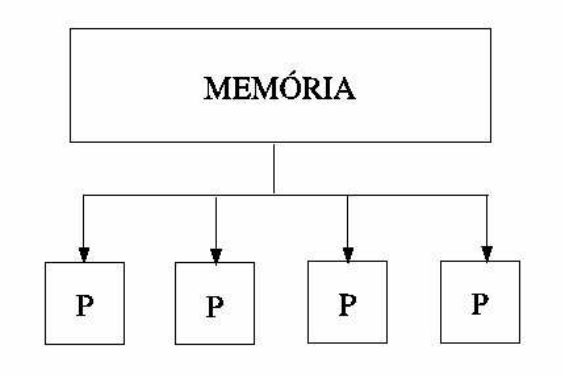

Figura 2.3: Memória Compartilhada.

esse processador como mostrado na Figura 2.4. Os dados podem ser compartilhados através de uma rede de comunicação utilizando a troca de mensagens. Neste modelo de acesso, deve haver a sincronização, realizada através da troca de mensagens.

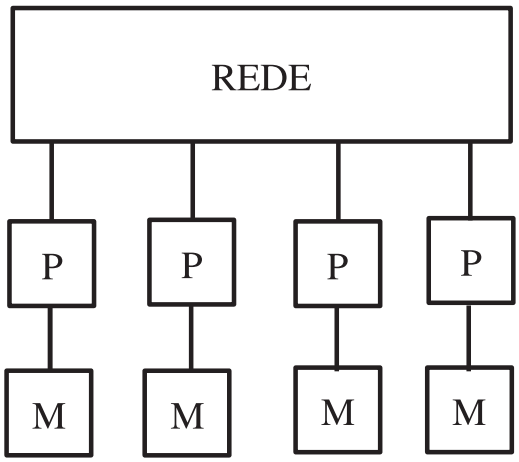

Figura 2.4: Memória Distribuída.

Os Sistemas Híbridos caracteriza-se por ter dois tipos de memórias: a memória distribuída e a memória compartilhada. Nesses sistemas, os processadores são divididos em grupos e, cada grupo de processadores possui seu próprio espaço de memória, sendo que estes não é acessível aos demais grupos.

2.4 Avaliação de Performance

2.4.1 Speedup

O Speedup é uma medida de desempenho de programas paralelos, onde se faz uma relação entre o tempo de execução do melhor algoritmo executado em um processador e o tempo de execução do programa em paralelo, ou seja, executando em mais de um processador. O desempenho de um algoritmo é medido pela *aceleração* e pela *eficiência* (seção 2.4.2) conforme abaixo:

- Seja Tp o tempo de execução de um programa paralelo em p processadores e Ts o tempo de execução do melhor algoritmo serial, a *aceleração* em p processadores é definida pela relação

$$Sp = \frac{Ts}{Tp}, \tag{2.1}$$

A aceleração Sp pode também medir o ganho obtido em utilizar um programa paralelo ao invés de um programa serial executado em apenas um processador. O item acima propõe a razão dos tempos como indicador da aceleração relativa, sem exigir que seja utilizado o mesmo algoritmo implementado seqüêncialmente e paralelamente. Para maiores detalhes e outras medidas de desempenho, *veja Grama et al. [6].*

2.4.2 Eficiência

A eficiência é um outro tipo de medida de desempenho paralelo. Ela é a relação entre o Speedup e o número de processadores conforme a equação abaixo:

$$Ep = 100\frac{Sp}{p}, \tag{2.2}$$

onde Sp é o Speedup apresentado na Seção 2.4.1 e p é o número de processadores utilizados para execução do algoritmo. A eficiência mede o percentual da fração de

tempo em que os recursos totais de computação são utilizados sem contabilizar o tempo total gasto com comunicação e sincronização. Quando a aceleração é linear, a eficiência é igual a 100% para todo p, o que raramente é verificado na prática. No desenvolvimento de algoritmos paralelos, deseja-se que a aceleração seja mantida o mais próximo possível de p e a eficiência o mais próximo possível de 100%.

As medidas de desempenho obtidos através do número de processadores utilizados no processamento, permitem avaliar o desempenho dos programas paralelos.

2.5 Interpretadores e Linguagens

2.5.1 Scilab

No software científico existem duas categorias em geral: uma delas são sistemas de álgebra computacional que executam computação simbólica e a outra categoria é sistema numérico executando computações númericas com objetivo específico de computação científica. Os mais conhecidos da primeira categoria são o Maple, Mathematica, Maxima, Axiom e Mupad. A segunda categoria representa um grande mercado dominado pelo Matlab. Scilab é um software com código fonte livre que faz parte da segunda categoria [7].

Scilab é uma linguagem interpretada com tipos de objetos dinâmicos que executa e é avaliado em formato binário. O Scilab foi criado com o principal objetivo do desenvolvimento na plataforma Linux, depois Windows e MacOSX. A compilação do Scilab para código fonte é completamente fácil de entender e fácil de instalar.

Scilab foi originalmente chamado de Basile e foi desenvolvido pelo *INRIA* como parte do projeto *Meta2*. O desenvolvimento continuou dentro do projeto do grupo *Scilab to Scilab*, que estava com um grupo de pesquisadores do *INRIA Metalau e ENPC*. Desde 2004, o desenvolvimento do Scilab têm sido coordenado por um consórcio.

O Scilab pode ser usado como uma linguagem de script para testar algoritmos ou executar computações numéricas. Também é uma linguagem de programação, cuja biblioteca padrão contém em torno de 2000 funções codificadas em Scilab. A sintaxe é simples, e o uso de matrizes (objetos fundamentais para o cálculo científico) é facilitado através de funções e operadores específicos. Essas matrizes podem ter diferentes tipos incluindo real, complexos, strings, polinomiais e racionais. Essa linguagem é dedicada principalmente para computação científica e fornece fácil acesso para uma grande biblioteca em áreas como álgebra linear, integração numérica e otimização.

Scilab pode facilmente importar novas facilidades de bibliotecas externas dentro do mesmo, usando links dinâmicos e estáticos [8], [9]. Além disso, esse interpretador fornece muitas funcionalidades visuais incluindo 2D, 3D, contorno e plotações paramétricas, e animações. Os gráficos podem ser exportados em vários formatos como Gif, PostScript, PostScript-Latex e Xfig. Além das funções de interface de usuário, o Scilab usa um sofisticado desenvolvimento como a interface TCL/Tk para GUI, *do inglês, Graphical User Interfaces*.

Scilab é um grande pacote de software contendo aproxidamente 13.000 arquivos, com mais de 400.000 linhas de código (em C e Fortran), 70.000 linhas de código Scilab *(bibliotecas específicas)*, 80.000 linhas de ajuda on-line e 18.000 linhas de configurações de arquivos. Esses arquivos incluem:

- Funções elementares de cálculo científico;
- Álgebra Linear, matrizes esparsas;
- Polinômios e funções racionais;
- Controle Clássico e robusto, Inequações Matriciais Lineares - LMI, otimização;
- Métodos não lineares (otimização, ODE e DAE solução, Scicos; que é um sistema dinâmico híbrido de modelagem e simulador);
- Processamento de Sinais;

- Amostragem aleatória e estatística;
- Grafos (algoritmos, visualização);
- Gráficos, animação;
- Paralelismo usando PVM *Parallel Virtual Machine*;
- Tradução de código do Matlab para Scilab;

Enfim, um grande número de contribuições para várias áreas.

2.5.2 Scilab Parallel

O Scilab Parallel possibilita a utilização do Scilab comum em ambientes multiprocessados, através do *Toolbox PVM*, além de se integrar a bibliotecas externas [10], [2] que serão mostradas na seção 2.7. Para que o Scilab funcione de maneira correta é necessário que haja a instalação primária do *PVM* que pode ser encontrado em [11] para download.

A figura 2.5 mostra o esquema de integração do *PVM com o Scilab* que gera o Scilab paralelo. Ao executar o *scilab paralelo* e habilitar o *pvm* existente, cria-se em background o daemon *pvmd* que por meio do *pvm_spawn* habilita cada *pvmd* às máquinas solicitadas no momento da execução da tarefa.

Por meio da execução do comando *top* no terminal do sistema operacional *Linux*, a tela da figura 2.6 nos fornece informações importantes como: as tarefas que estão sendo executadas *scilex*, o usuário, a *cpu* em trabalho, e o tempo em que estão em execução.

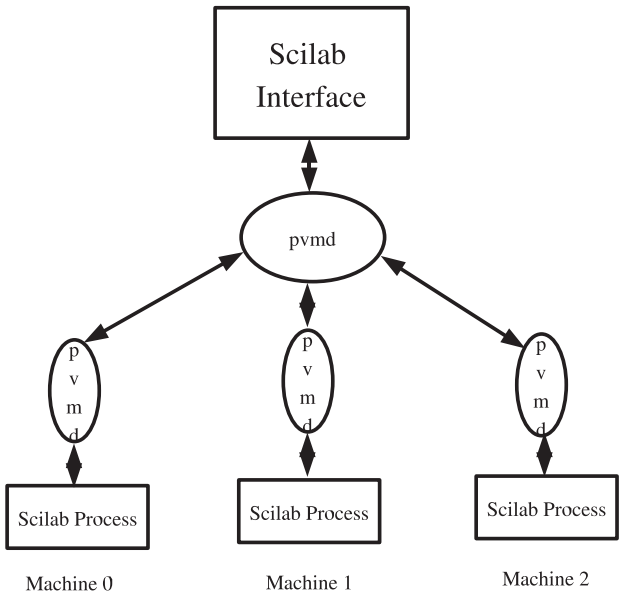

Figura 2.5: Interação entre vários processos Scilab.

Na programação do Scilab com o *PVM* existe um manual que pode ser visualizado em [12] contendo todas as funções necessárias para todos os toolbox do Scilab.

2.6 Distribuição de Dados

De acordo com a abordagem dos conceitos na Seção 3.1 que será apresentado a seguir, quando se utiliza vários processadores para executar uma tarefa é necessário que se distribuam os dados. A distribuição de dados depende do tipo de algoritmo e do tipo da arquitetura da máquina que está sendo utilizado. O processo de divisão de dados para diferentes processadores é denominado *decomposição*.

A técnica de divisão e conquista apresentado na seção 2.2.2 é uma técnica de resolver um problema grande que pode implicar em uma maneira de distribuir dados.

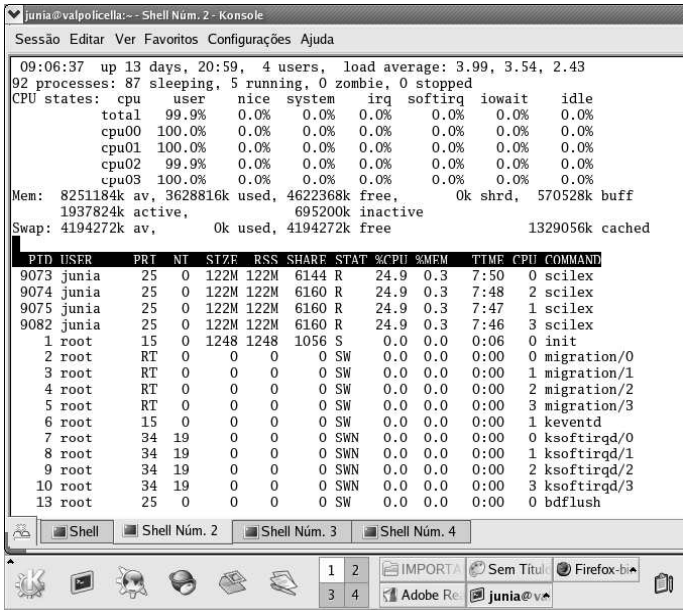

Figura 2.6: Mostra da tela dos processos através do comando do Linux top

2.6.1 PVM - Parallel Virtual Machine

O *PVM* teve início em 1989 no *Oak Ridge National Laboratory - ORNL* e seu protótipo (versão 1.0) foi implementado por *Sunderam e Geist (ORNL)* [11]. Com a versão 2.0 deu-se início à disponibilidade pública do código que era utilizada somente para arquiteturas que tinham processamento distribuído, mas, com o crescimento da tecnologia, os computadores foram se tornando cada vez mais robustos e novas arquiteturas foram criadas e com isso houve uma adaptabilidade para o *PVM* deixando de ser somente utilizado em máquinas com processamento distribuído para também ter processamento em sistemas multiprocessados. Os processadores físicos podem ser uma rede de estações de trabalho ou nós de um computador com multiprocessadores. Atualmente a versão mais recente e disponível é a versão 3.4.5 [11].

O modelo computacional do *PVM* é composto por duas partes: um daemon denominado *pvmd* e uma biblioteca de rotinas com a interface PVM denominado *libpvm*. O *pvmd* não faz nenhum processamento, mas funciona como um roteador para o controle de mensagens, ou seja, funciona como um ponto entre os hosts

autenticando as mensagens, e controlando os processos para detectar possíveis falhas.

Na *libpvm* encontram-se as primitivas necessárias à comunicação entre as tarefas de uma aplicação, funcionando como um elo entre uma tarefa e a máquina virtual, ou seja, são rotinas que podem ser chamadas para efetuar troca de mensagens, solicitar a geração de processos, coordenar tarefas e solicitar modificações na máquina virtual.

As tarefas executadas através do *PVM* utilizam um número inteiro como identificador, o *task identifier (TID)*, que são utilizados nas mensagens que são trocadas entre os computadores que formam a máquina virtual paralela. Como esse identificador deve ser único na máquina virtual, ele é definido pelo *daemon* local, e não pelo usuário. A identificação das tarefas pela aplicação do usuário é possível através de diversas rotinas, que o *PVM* oferece, que retornam valores de *TID*, possibilitando a identificação das tarefas pela aplicação do usuário.

Os programas *PVM* utilizam o modelo mestre-escravo, onde o processo mestre é o primeiro a ser executado e todos os outros processos são criados por ele. Para iniciar a execução de um ou mais processos idênticos em cada processador, é usada a rotina *pvm_spawn()*.

As aplicações paralelas possuem diversos processos que são executados concorrentemente. Todo processo precisa ser gerenciado pelo *daemon* e as funções do *PVM* são divididas entre funções de controle de processos e funções que gerenciam o envio de dados.

As funções mais importantes quanto ao controle de processos são:

- *pvm_start()*$\Longrightarrow$ é uma função que habilita o daemon da máquina virtual.
- *pvm_mytid()*$\Longrightarrow$ é uma função que devolve um valor inteiro. Este valor é usado pelos demais processos para comunicar-se com o processo chamador. Esta função é a primeira função do *PVM* chamada no programa e tem a função de registrar os processos e informar ao processo o identificador.
- *pvm_spawn()*$\Longrightarrow$ é usado para inicializar as tarefas. No modelo de progra-

mação mestre/escravo, esta função é usada no programa mestre para lançar processos escravos. Esta função é responsável também por espalhar as tarefas em uma máquina multiprocessada.

- *pvm_kill()*$\Longrightarrow$ é usado quando deseja-se terminar o processo.
- *pvm_halt()*$\Longrightarrow$ desabilita o daemon que está sendo executado.
- *pvm_parent()*$\Longrightarrow$ função útil em processos escravos onde esta função tem por objetivo retornar o identificador do processo que iniciou o processo atual, facilitando a troca de mensagens entre os escravos e o mestre.

Quanto ao envio e recebimento de dados, o *PVM* foi construído sobre um conjunto de diretivas de trocas de mensagens, assim como o *MPI- Message Passing Interface*. As aplicações funcionam atráves de trocas de informações e cada mensagem possui um identificador de tipo, uma codificação e os dados propriamente ditos. As principais funções de envio e recebimento de dados são:

- *pvm_send()*$\Longrightarrow$ responsável por realizar o envio de dados armazenados em buffer de envio para uma determinada tarefa.
- *pvm_recv()*$\Longrightarrow$ responsável por receber uma mensagem enviada por outro processo e colocar no buffer de recebimento.

2.6.2 Tipos de Comunicação

A comunicação ponto-a-ponto é realizada no *PVM* através de rotinas *pvm_send* e *pvm_recv*. Todas as rotinas de envio do *PVM* são não bloqueantes e as rotinas de recebimentos podem ser bloqueantes ou não-bloqueantes. As rotinas de envio e recebimento de mensagens utilizam um *buffer* de mensagens. Se os dados estiverem armazenados no *buffer*, a rotina de envio é usado para iniciar o envio do conteúdo do *buffer* através da rede até ao *buffer* de recebimento, preparado pela rotina de recebimento.

Para a comunicação coletiva, o *PVM* oferece rotinas que são utilizadas para a comunicação entre os membros de um grupo de processos. Em geral, os grupos envolvem mais que dois processos. Essas rotinas implementam as operações coletivas.

As rotinas utilizadas para esse tipo de comunicação são *pvm_ bcast() e pvm_ reduce()*. Essas duas funções possuem interface conjunta com o Scilab em [1] e rotinas específicas do *PVM* em [11] .

2.7 Software Científico

2.7.1 BLAS(Basic Linear Algebra Subprograms)

O BLAS é constituído de rotinas básicas de álgebra linear [13],Freeman [4] e oferece um bom desempenho em diversas arquiteturas, sendo que este pacote possui versões otimizadas, nas quais suas bibliotecas são escritas em código de máquina. O pacote do software BLAS concentra suas operações em matrizes densas e de bandas Blackford et al. [14]. Em Blackford et al. [14],Freeman [4], as funções (rotinas) existentes neste pacote possuem três níveis de operações:

- Nível 1: São rotinas que realizam operações vetor-vetor do tipo: $y = \alpha x + y$
- Nível 2: São rotinas que realizam operações matriz-vetor do tipo: $y = \alpha Ax + \beta y$
- Nível 3: São rotinas que realizam operações matriz-matriz do tipo: $C = \alpha AB + \beta C$

Neste trabalho, matrizes são denotadas por letras maiúsculas *(A,B,C)* , vetores por letras minúsculas $(x,y,z,)$ e escalares por letras gregas como α, β .

2.7.2 LAPACK(Linear Algebra PACKage)

O LAPACK é uma biblioteca matemática destinada na resolução de problemas de álgebra linear constituída de rotinas para resolução de equações lineares, resolução de mínimos quadrados, autovalores e autovetores [15]. Essas rotinas podem ser utilizadas nas operações sobre matrizes reais ou complexas, usando precisão simples ou dupla.

O LAPACK foi baseado na utilização das rotinas do EISPACK e LINPACK em computadores vetoriais ou de memória compartilhada de forma eficiente. Isso aconteceu porque as duas bibliotecas citadas anteriormente apresentam certo sucesso entre a comunidade científica, mas não ofereciam um bom desempenho nessas máquinas, uma vez que elas não consideram a hierarquia de memória nelas existente, gastando mais tempo no envio de dados do que nas operações de ponto flutuante.

No LAPACK, utilizam-se algoritmos baseados em divisão por blocos para efetuar operações com matrizes, de forma que a hierarquia de memória é aproveitada para otimizar os cálculos.

Com o desenvolvimento do LAPACK, ele superou o LINPACK e EISPACK [4] em algumas características, como velocidade, precisão, robustez e funcionalidade, através do uso das rotinas do nível três do BLAS.

2.7.3 BLACS(Basic Linear Algebra Communication Subprograms)

O BLACS é uma biblioteca de comunicação responsável por interconectar vários processadores através de passagem por mensagem, dedicado à álgebra linear. O objetivo do BLACS é utilizar o ScaLAPACK, uma biblioteca de comunicação independente da arquitetura. O BLACS é uma interface que encapsula as bibliotecas de comunicação com o MPI ou PVM.

As rotinas BLACS [16] compreendem:

1. As rotinas de comunicação são síncronas para o envio e recepção de uma matriz.
2. As funções de difusão*(broadcast, em inglês)*;
3. As funções de redução (soma, máximo e mínimo);

O pacote *PVM_BLACS* é um pacote paralelo contido dentro da estrutura BLACS.

2.7.4 ScaLAPACK

O ScaLAPACK é uma biblioteca que foi criado baseado na biblioteca LAPACK. Isto aconteceu porque os cientistas vinham tendo algumas dificuldades quanto ao seu desempenho em computadores de memória distribuída, o que se tornava cada vez mais utilizadas. Então, surgiu a necessidade de se criar uma biblioteca que fosse parecida com o LAPACK, mas que oferecesse bom desempenho em computadores de memória distribuída.

O ScaLAPACK é constituído de rotinas [17] que resolvem os mesmos problemas que o LAPACK, além de ter outras bibliotecas que realizam a comunicação e a divisão das matrizes entre os processos desejados como o BLACS(*Basic Linear Algebra Communication*) e PBLAS (*Parallel BLAS*).

Capítulo 3

Estudo Comparativo de Computação Científica

3.1 Introdução

Este capítulo apresenta as técnicas utilizadas para resolver problemas típicos de computação científica e avalia o desempenho das mesmas em um ambiente de computação paralela. A seção 3.2 recapitula conceitos de multiplicação de matrizes no contexto da computação seqüencial. Técnicas de decomposição são descritas resumidamente na seção 3.3, dando maior ênfase nos dois tipos de particionamentos mais utilizados nesta dissertação. Apresenta-se também a paralelização destas técnicas nas linguagens de programação C, FORTRAN e SCILAB. A seção 3.4 finaliza o capítulo apresentando os resultados numéricos obtidos nos problemas estudados neste capítulo.

3.2 Multiplicação de Matrizes

Na computação científica, uma das operações mais freqüentes é a multiplicação de duas matrizes de dimensões conformais, isto é, $A \in \mathbb{R}^{mxn}$, $B \in \mathbb{R}^{nxp}$, cuja

multiplicação resulta em $C = AB \in \mathbb{R}^{mxp}$, dada pela expressão 3.1:

$$\sum_{j=1}^{n} a_{ij} b_{ji} \tag{3.1}$$

Esta operação possui complexidade computacional da ordem de $O(n^3)$.

O pseudo-código da Figura 3.1 refere-se à multiplicação clássica de matrizes baseada no cálculo do produto interno das linhas da primeira matriz com as colunas da segunda matriz.

Figura 3.1: Pseudo-Código de Multiplicação de Matrizes

```
Dados: A, B
for i=1 to n do
   for j=1 to n do
      C(i,j) =0 inicializa a matriz com zeros
      for k=1 to n do
         C(i,j)= C(i,j) + A(i,k)*B(k,j)
      end for
   end for
end for
```

Veremos adiante que há diversas outras maneiras de organizar os cálculos de multiplicação matricial, e a maneira mais adequada dependerá de vários fatores, tais como a arquitetura da máquina, a linguagem de programação a ser utilizada, etc.

3.3 Técnicas de Decomposição

Técnicas de decomposição são utilizadas, em computação paralela, para subdividir problemas em subproblemas que possam ser resolvidos simultaneamente, isto

é, em paralelo, reunindo os resultados parciais para obter o resultado final. Com essa abordagem, obtêm-se uma economia de tempo proporcionada pela concorrência na execução dos subproblemas. Em geral, a etapa da decomposição é uma das primeiras a serem realizadas na maioria das computações paralelas.

Decomposição Recursiva, Decomposição de dados, Decomposição Exploratória e Decomposição Especulativa são alguns dos nomes dados às decomposições mais comuns [6]. As duas primeiras mencionadas, a recursiva e a de dados são as mais usadas para uma variedade de problemas. Por outro lado, a decomposição especulativa e exploratória são técnicas de natureza especial que são aplicadas para uma classe específica de problemas.

A *Decomposição Recursiva* é um método de indução de concorrência para resolver problemas com a estratégia de dividir para conquistar. O problema é primeiramente dividido em um conjunto de subproblemas, se possível com pouca interdependência entre eles, após o qual a técnica de divisão/decomposição é aplicada novamente a cada um dos subproblemas.

A *Decomposição de dados* é um método normalmente utilizado para produzir concorrência em algoritmos que operam sobre grandes estruturas de dados. No primeiro passo desta técnica, os dados a serem utilizados na computação realizada são particionados e, no segundo passo, a partição dos dados é utilizada para induzir uma partição da computação em tarefas. As operações que estas tarefas realizam sobre diferentes partições dos dados geralmente são similares.

A *Decomposição Exploratória* é utilizada para decompor problemas em que a computação consiste na busca de um espaço de soluções. A partição do espaço de soluções em subespaços permite a busca concorrente em cada um destes subespaços, reduzindo o tempo total de busca.

A *Decomposição Especulativa* é usada quando um programa pode seguir uma de várias ramificações computacionalmente significantes, dependendo do resultado de outras computações precedentes. Nesta situação, enquanto uma tarefa está reali-

zando a computação, cujo resultado será usado para decidir a próxima computação, outras tarefas podem começar concorrentemente a computação do próximo estágio. Nesta técnica, claramente podemos ter um desperdício de computação, correspondente aos ramos que não serão percorridos. Uma maneira de minimizar esse desperdício é computar somente as condições mais prováveis das soluções, dando origem ao nome decomposição especulativa.

3.3.1 Multiplicação matricial bloco-coluna vezes bloco-linha

O método de particionamento da matriz bloco-coluna por bloco-linha consiste em decompor uma matriz $A \in \mathbb{R}^{mxn}$ por colunas (figura 3.2) e a matriz $B \in \mathbb{R}^{nxp}$ por linhas (figura 3.3).

Esse particionamento segue o critério de dividir as matrizes na forma *n/nprocs* onde $< n =$ *dimensão da matriz* $>$ e $<$ *nprocs* $=$ *número de processadores* $>$. A forma resultante existente em cada processador se dá em uma matriz $C \in \mathbb{R}^{mxp}$ quadrada para cada processador que esteja em processamento ilustradas nas figuras 3.4 e 3.5.

Neste método utilizamos o paradigma de mestre-escravo mencionado na seção 2.2.1. O processador *mestre* primeiramente realiza a tarefa adicional de gerar aleatoriamente as matrizes, distribuir e receber todas as submatrizes de cada processador escravo. Ao fim, o processador mestre realiza a soma expressa na equação 3.2 de tais submatrizes

$$Cresultante = \sum C(Processador1) + C(Processador2) \tag{3.2}$$

obtendo assim, o resultado final.

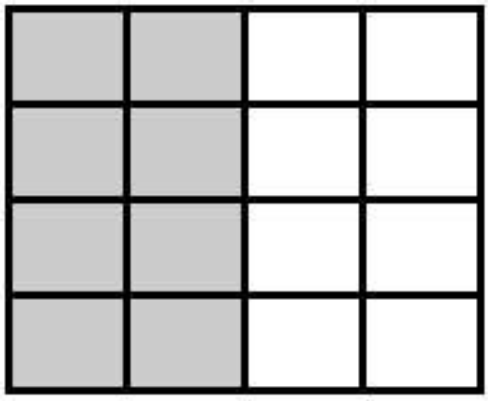

Figura 3.2: Visualização da matriz A particionadas por colunas.

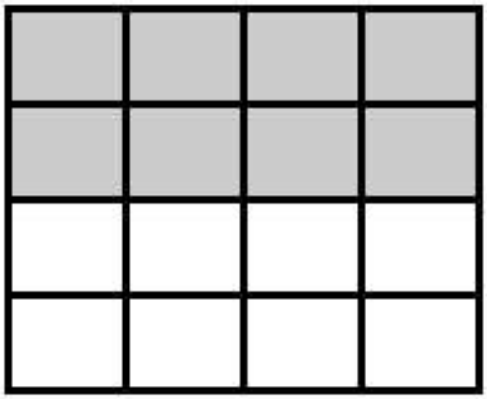

Figura 3.3: Visualização da matriz B particionadas por linhas.

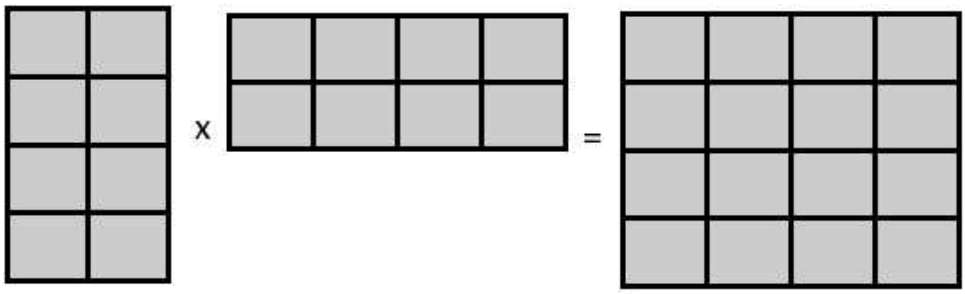

Figura 3.4: Multiplicação realizada no primeiro processador.

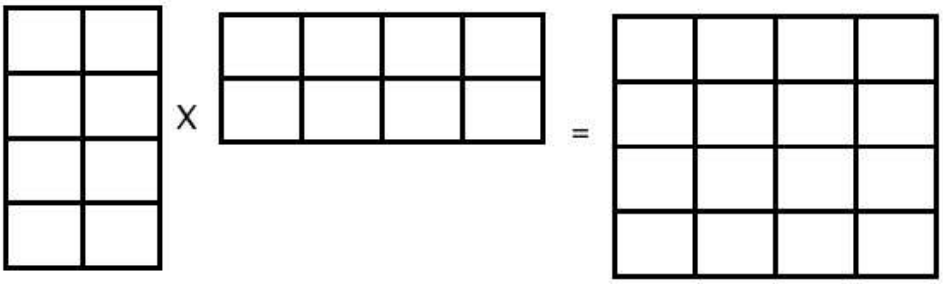

Figura 3.5: Multiplicação realizada no segundo processador.

Como forma de demonstrar esse tipo de particionamento, o código implementado na Figura 3.6 se refere ao processador *mestre*, que tem todas as informações necessárias, que serão distribuídas aos processos *escravos*. A Figura 3.7 contém as tarefas que os *escravos* executam na linguagem interpretada SCILAB utilizando o toolbox paralelo *PVM*. Para o cálculo matricial utilizamos função do BLAS demons-

Figura 3.6: Algoritmo Master.sci

```
stacksize(400000000)
ok     = pvm_start();
pvm_set_timer();
 arquivo = file('open','master1200v2_4.txt','unknown');
fim    = 0;
n      = 4000;
NPROCS = 4;
local  = n/NPROCS;
A = int(n*rand(n,n));
B = int(n*rand(n,n));
if ok<>0 then
   disp('pvm daemon already active')
end;
path = get_absolute_file_path('master.sci');
[ task_id, numt ] = pvm_spawn( path + '/slave.sci', NPROCS, 'nw' );
if numt<0 then
   disp(['pvm_spawn aborts to create a new process']);
end
 for i=1:NPROCS
    pvm_send ( task_id(i),A(:,local*i-(local-1):local*i),0);
    pvm_send ( task_id(i),B(local*i-(local-1):local*i,:),1);
end
[ rowA, colA ] = size(A);  [ rowB, colB ] = size(B);
RESULT  = zeros( rowA, colB );
for i=1:NPROCS
     RESULT = RESULT + pvm_recv( task_id( i ), 2);
end
resolv = (pvm_get_timer()/1000000);//tempo dado em segundos
RESULT;
fprintf (arquivo, "O tempo total para realizacao da soma",resolv);
file ('close',arquivo);
pvm_halt();
```

Figura 3.7: Código Slave.sci

```
stacksize(400000000)
parent = pvm_parent();
alpha  = 1 ;
beta   = 0 ;
A      = pvm_recv( parent, 0 );
B      = pvm_recv( parent, 1 );
RESULT = dgemm(alpha ,A ,B ,beta ,C );
pvm_send( parent, RESULT, 2 );
pvm_exit();
exit(0);
```

trando assim, a praticidade do uso de uma ferramenta numérica e colocando as aplicações a nível de prototipagem em paralelo.

Dgemm é uma função que se encontra no nível 3, apresentada na subseção 2.7.4, de operação da biblioteca Blas [13], [4] que realiza a operação matriz-matriz do tipo $C = \alpha AB + \beta C$. As funções das bibliotecas da seção 2.7 apresentam particularidades referentes à convenção na nomenclatura das rotinas. A nomenclatura das rotinas têm a seguinte estrutura:

<charater code> <name> <mod>

O ***<character code>*** é um caracter que indica o tipo de dados das matrizes e vetores e é expressa pela letra inicial da função. Então:

- **s** $\Longrightarrow$ corresponde ao tipo de dados real e de precisão simples;
- **c** $\Longrightarrow$ corresponde ao tipo de dados complexo e precisão simples;
- **d** $\Longrightarrow$ corresponde ao tipo de dados real e precisão dupla;

- **z** $\Longrightarrow$ corresponde ao tipo de dados complexo e precisão dupla.

Algumas funções do *Blas* combinam códigos de caracteres, como ***sc ou dz***. O <***name***> indica a operação do tipo e reflete no tipo de argumento da matriz.

- **ge** $\Longrightarrow$ matrizes em geral;
- **gb** $\Longrightarrow$ matriz banda em geral;
- **sy** $\Longrightarrow$ matriz simétrica;
- **sp** $\Longrightarrow$ matrix simétrica *(packet storage)*;
- **sb** $\Longrightarrow$ matriz banda simétrica;
- **he** $\Longrightarrow$ matriz hermitiana;
- **hp** $\Longrightarrow$ matrix hermitiana *(packet storage)*;
- **hb** $\Longrightarrow$ matrix banda hermitiana;
- **tr** $\Longrightarrow$ matrix triangular;
- **tp** $\Longrightarrow$ matrix triangular *(packet storage)*;
- **tb** $\Longrightarrow$ matrix banda triangular *(packet storage)*;

O campo <***mod***> faz o detalhamento da operação para o nível correspondente de operações.

- Nível 1
  - **c** $\Longrightarrow$ vetor conjugado;
  - **u** $\Longrightarrow$ vetor não conjugado;
  - **g** $\Longrightarrow$ rotação de Givens;
- Nível 2

  - **mv** $\Longrightarrow$ produto de matriz-vetor;
  - **sv** $\Longrightarrow$ resolução de sistemas de equações lineares com matriz-vetor;
  - **r** $\Longrightarrow$ atualização do rank-1 da matriz;
  - **r2** $\Longrightarrow$ atualização do rank-2 da matriz;

- Nível 3
  - **mm** $\Longrightarrow$ produto de matriz-matriz;
  - **sm** $\Longrightarrow$ resolução de sistemas de equações lineares com matriz-matriz;
  - **rk** $\Longrightarrow$ atualização do rank-k da matriz;
  - **r2k** $\Longrightarrow$ atualização do rank-2k da matriz;

De posse dessa informação, entende-se que a função *dgemm* significa *produto matriz-matriz, matriz geral, real ou precisão dupla.*

A utilização de qualquer biblioteca da seção 2.7 requer um conhecimento do tipo de dados em uso para uma correta escolha das funções.

3.3.2 Multiplicação matricial bloco-linha vezes matriz

A multiplicação matricial bloco-linha vezes matriz consiste na distribuição da matriz A por linhas, de acordo com a quantidade de processadores, e da difusão (broadcast) da *matriz B* inteira para todos os processadores. O processo de particionamento é feito pelo processador *mestre* que responsabiliza-se por gerar aleatoriamente as matrizes reais e realizar o particionamento da forma *linha X Matriz inteira* conforme figuras 3.8 e 3.9. O processador *mestre* divide a matriz da forma *n/nprocs* sendo *n* = <*a dimensão da matriz*> e *nprocs*=<*número de processadores*>. Matrizes são enviadas para os *processadores escravos* que serão responsáveis por realizar os cálculos da multiplicação de matrizes. Após os cálculos, cada processador *escravo* tem em sua memória submatrizes, que serão enviadas ao processador *mestre* para realizar a concatenação das submatrizes, utilizando uma estrutura já pré-definida

do Scilab que faz $C = [A; B]$. Essa concatenação das matrizes pode ser visualizada na Figura 3.10.

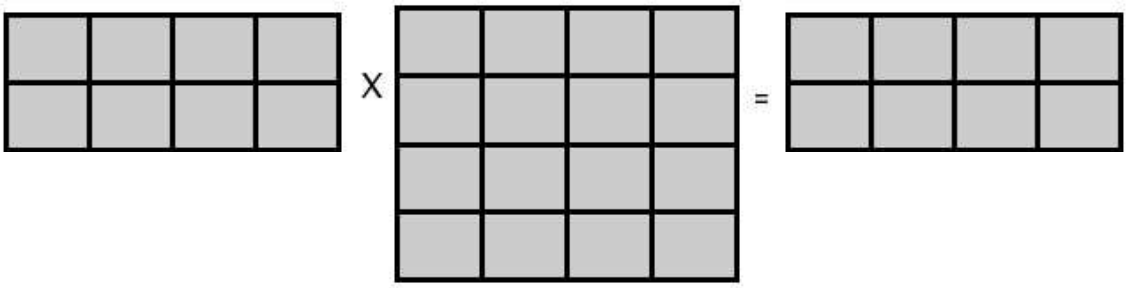

Figura 3.8: Multiplicação realizada pelo primeiro processador.

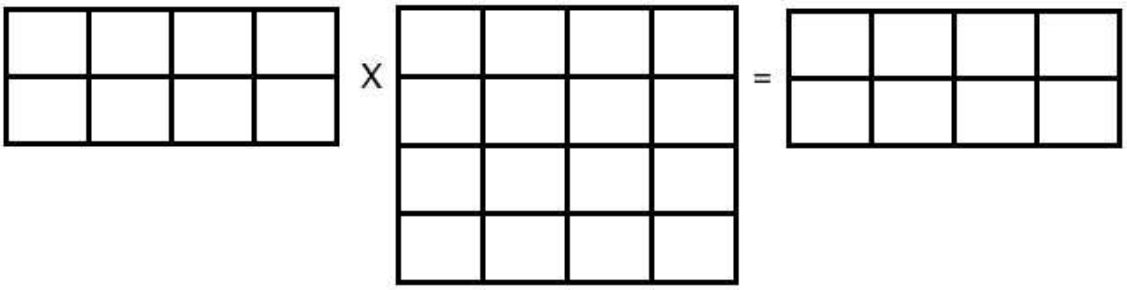

Figura 3.9: Multiplicação realizada pelo segundo processador.

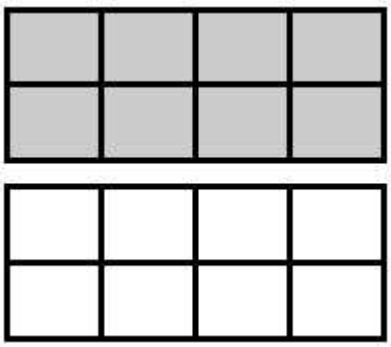

Figura 3.10: Concatenação das matrizes parciais.

O trecho dos códigos das Figuras 3.11 e 3.12 exemplificam o método de particionamento conforme Figura 3.10.

A linguagem interpretada Scilab é uma linguagem que possui como característica a facilidade de programação. Isto porque este interpretador de cálculo científico têm

Figura 3.11: Trecho do Algoritmo Master.sci

```
[ task_id, numt ] = pvm_spawn( path + '/slave.sci', NPROCS, 'nw' );
if numt<0 then
   disp(['pvm_spawn aborts to create a new process']);
end
for i=1:NPROCS
    pvm_send( task_id(i), A(local*i-(local-1):local*i,:), 0 );
    pvm_send( task_id(i), B, 1 );
end
[ rowA, colA ] = size(A);
[ rowB, colB ] = size(B);
rowpart = rowA/nprocs;
C = zeros(rowpart,[colB]);
D = zeros([rowA],[colB]);
E = zeros(rowpart,[colB]);
F = zeros(rowpart,[colB]);
C =  pvm_recv ( task_id( 1 ), 2 );
D =  pvm_recv ( task_id( 2 ), 3 );
E =  pvm_recv ( task_id( 3 ), 4 );
F =  pvm_recv ( task_id( 4 ), 5 );
RES =  [C;D;E;F];//concatena pedacos das matrizes
```

Figura 3.12: Trecho do Código Slave.sci

```
stacksize(400000000)
alpha  = 1;
beta   = 0;
parent = pvm_parent();
A = pvm_recv( parent, 0 )
B = pvm_recv( parent, 1 )
C = dgemm(alpha, A, B, beta, C);
pvm_send( parent, C, 2 );
pvm_send( parent, C, 3 );
pvm_send( parent, C, 4 );
pvm_send( parent, C, 5 );
pvm_exit();
exit(0);
```

em sua estrutura várias funções intrínsecas para diferentes abordagens numéricas. Como por exemplo, nesta mesma multiplicação de matrizes poderíamos realizá-lo utilizando o operador $*$ o que diminui muito o tempo em relação à implementação de código.

3.4 Simulações

A seguir, serão apresentados resultados de simulações computacionais, utilizando os dois métodos de particionamentos apresentados nas subseções 3.3.1 e 3.3.2.

As simulações foram realizadas com o objetivo de comparar os Speedups apresentado na seção 2.4 e o tempo gasto de processamento, utilizando a biblioteca do Netlib Blas [13] na linguagem interpretada *Scilab* e as nas linguagens compiladas *C e Fortran*.

3.4.1 Resultados

3.4.1.1 Avaliação do Speedup para a multiplicação matricial bloco-coluna vezes bloco-linha

As tabelas 3.1 e 3.2, e seus respectivos gráficos das figuras 3.13, 3.14 e 3.15, ilustram o tempo dispendido no processamento deste modo de particionamento apresentado na subseção 3.3.1, para as linguagens acima especificadas com dimensões de matrizes para *n=500, n=1000, n=2000, n=3000 e n=4000*. Estas dimensões foram escolhidas por causa da restrição em que o Scilab possui em utilizar dimensões maiores, chegando ao limite máximo de tamanho de pilha na memória. Existe no Scilab uma função chamada *Stacksize* que permite o aumento dessa pilha, mas para matrizes de dimensões maiores esta função foi ineficiente. Foram utilizados nos testes a quantidade máxima de 8 processadores.

	SCILAB - Tempo em segundos			
$N \downarrow Nprocs \rightarrow$	1 Proc	2 Proc	4 Proc	8 Proc
500	1.72	1.13	0.83	0.75
1000	20.22	9.31	5.61	4.47
2000	171.41	86.92	48.17	35.89
3000	573.02	287.45	154.08	102.7
4000	1355.68	674.02	357.05	231.45

Tabela 3.1: Tempo de processamento (em segundos) obtidos na execução do programa em até 8 processadores com a multiplicação matricial bloco-coluna vezes bloco-linha (SCILAB)

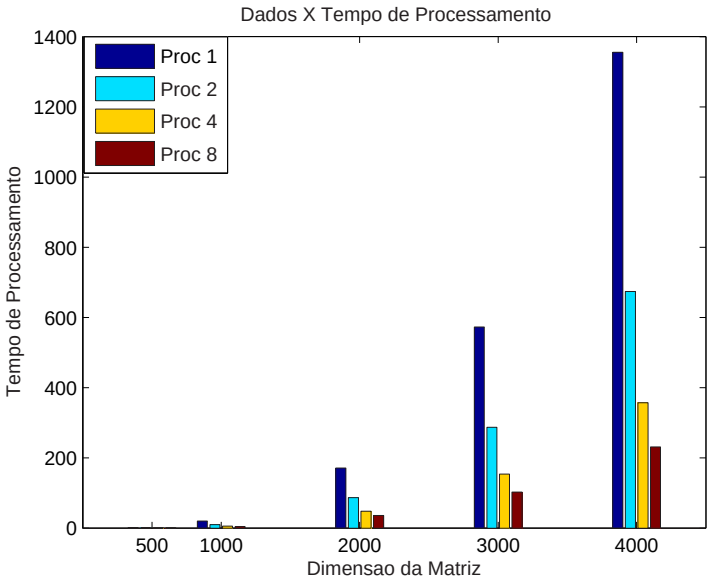

Figura 3.13: Tempo de Processamento obtido da tabela 3.1 dos dados do interpretador SCILAB

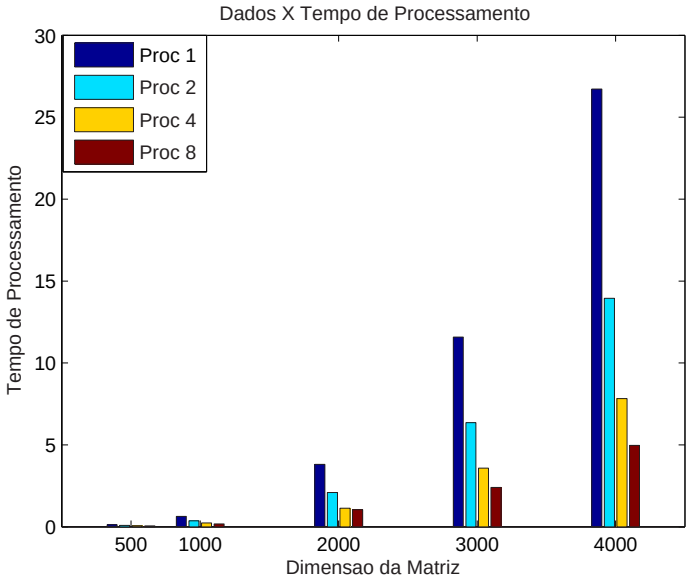

Figura 3.14: Tempo de Processamento obtido da tabela 3.2 com a linguagem C

	C - Tempo em segundos				Fortran - Tempo em segundos			
$N \downarrow Nprocs \rightarrow$	1 Proc	2 Proc	4 Proc	8 Proc	1 Proc	2 Proc	4 Proc	8 Proc
500	0.1211	0.0811	0.0643	0.0586	0.089	0.058	0.042	0.033
1000	0.6293	0.3608	0.2374	0.1755	0.503	0.322	0.264	0.182
2000	3.8092	2.932	1.1355	1.0484	3.381	1.951	1.432	1.131
3000	11.5813	6.3549	3.5844	2.4047	10.741	5.893	3.294	2.245
4000	26.7167	13.9422	7.8234	4.9652	24.696	12.741	6.945	4.624

Tabela 3.2: Tempo de processamento (em segundos) obtidos na execução do programa em até 8 processadores com a multiplicação matricial bloco-coluna vezes bloco-linha (C e FORTRAN)

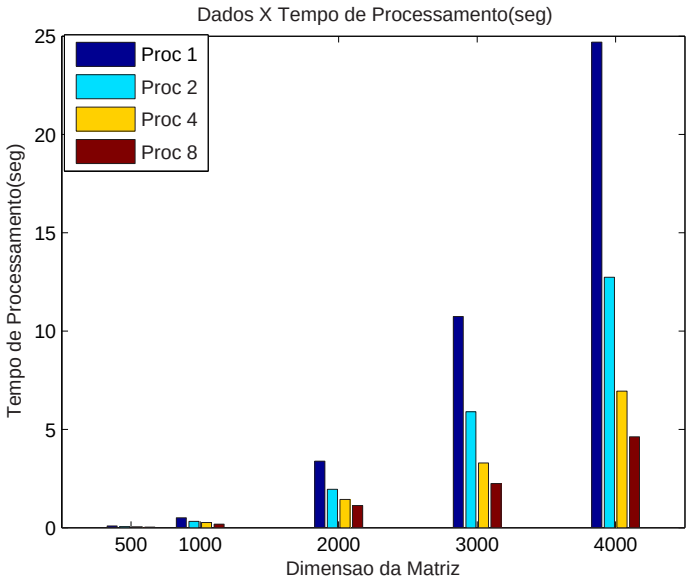

Figura 3.15: Tempo de Processamento obtido da tabela 3.2 com a linguagem FORTRAN

As tabelas 3.1 e 3.2 mostram que, em termos de tempo total de processamento, a linguagem SCILAB, apresentou-se um maior tempo de processamento pois, por ser uma linguagem interpretada, cada linha de código fonte é lida e executada em tempo real, não havendo compilação de código executável. Entretanto, como já dito no capítulo 1, a dissertação propõe a utilização do SCILAB apenas para prototipagem ou prova-de-conceito de algoritmos paralelos, e os resultados das Tabelas 3.1 e 3.2 enfatizam que, se o algoritmo paralelo passar nos testes feitos em SCILAB quanto a robustez e confiabilidade na obtenção do resultado correto, então ele deverá ser otimizado quanto à velocidade de execução em alguma outra linguagem compilada.

As tabelas 3.3 e 3.4 e seus respectivos gráficos nas figuras 3.16, 3.17 e 3.18 ilustram o Speedup pelo tempo de processamento de um processador em função da quantidade de processadores utilizados.

	Gráfico de Speedup - SCILAB			
	1 Proc	2 Proc	4 Proc	8 Proc
500	1.00	1.52	2.06	2.29
1000	1.00	1.97	3.47	4.50
2000	1.00	1.97	3.55	4.77
3000	1.00	1.99	3.71	5.57
4000	1.00	2.01	3.79	5.85

Tabela 3.3: Speedup obtido pela multiplicação matricial bloco-coluna vezes bloco-linha -SCILAB

	Gráfico de Speedup - C				Gráfico de Speedup -FORTRAN			
	1 Proc	2 Proc	4 Proc	8 Proc	1 Proc	2 Proc	4 Proc	8 Proc
500	1.00	1.493	1.883	2.065	1.00	1.534	1.854	2.696
1000	1.00	1.744	2.650	2.892	1.00	1.559	1.901	2.758
2000	1.00	1.474	2.713	2.938	1.00	1.732	2.361	2.989
3000	1.00	1.823	3.231	4.817	1.00	1.822	3.260	4.784
4000	1.00	1.916	3.414	5.380	1.00	1.938	3.559	5.340

Tabela 3.4: Speedup obtido pelo método multiplicação matricial bloco-coluna vezes bloco-linha- C e FORTRAN

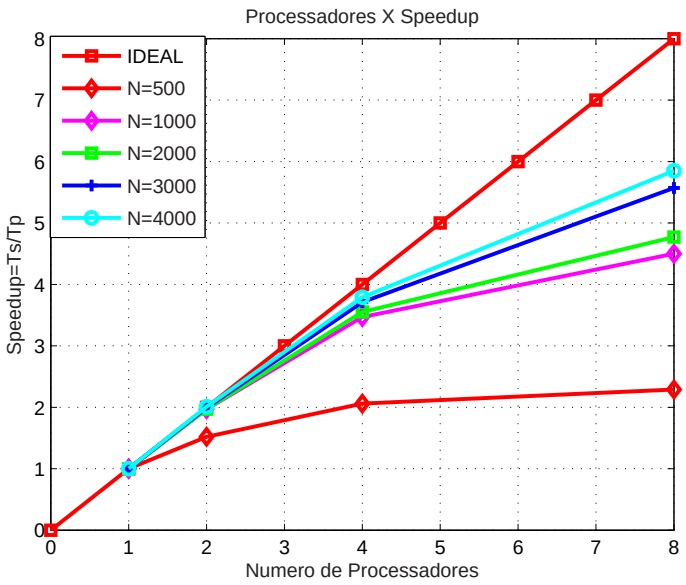

Figura 3.16: Gráfico do Speedup obtido pela multiplicação matricial bloco-coluna vezes bloco-linha de acordo com a tabela 3.3 do interpretador SCILAB

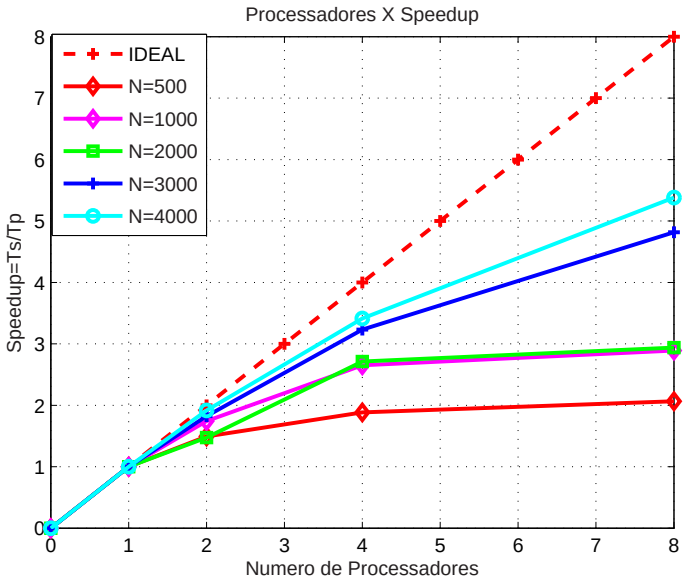

Figura 3.17: Gráfico do Speedup obtido pela multiplicação matricial bloco-coluna vezes bloco-linha de acordo com a tabela 3.4 com a linguagem C

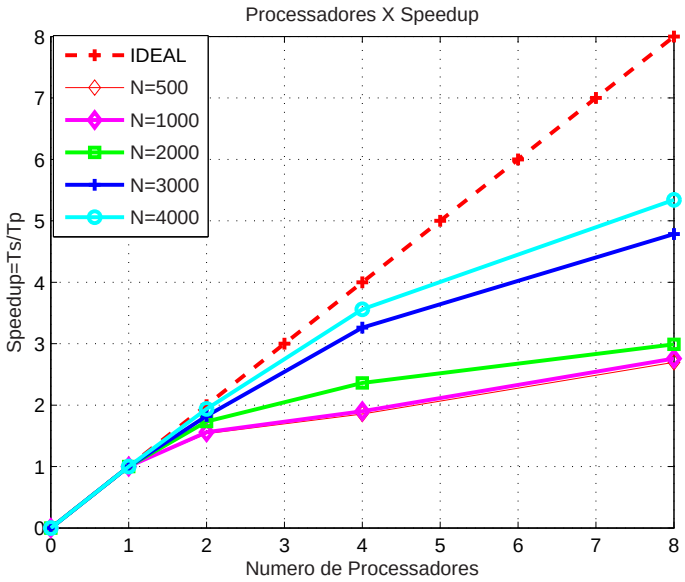

Figura 3.18: Gráfico do Speedup obtido pela multiplicação matricial bloco-coluna vezes bloco-linha de acordo com a tabela 3.4 com a linguagem FORTRAN

As tabelas 3.3 e 3.4 e os gráficos das figuras 3.16, 3.17 e 3.18 ilustram os ganhos de tempo obtidos por paralelismo independem da linguagem. Em outras palavras, seja a linguagem interpretada ou compilada, a curva de Speedup possui a mesma natureza, embora os tempos de computação sejam muito menores no segundo caso (da linguagem compilada).

3.4.1.2 Avaliação do Speedup para a multiplicação matricial bloco-linha vezes matriz

As tabelas 3.5 e 3.6 e os gráficos das figuras 3.19, 3.20 e 3.21, ilustram o tempo dispendido no processamento para a multiplicação matricial bloco-linha vezes matriz, para as linguagens com dimensões de matrizes para *n = 500, 1000, 2000, 3000 e 4000.*

	SCILAB - Tempo em segundos			
$N \downarrow Nprocs \rightarrow$	1 Proc	2 Proc	4 Proc	8 Proc
500	1.72	1.14	0.83	0.75
1000	20.22	10.20	6.30	4.21
2000	171.41	87.36	51.34	34.00
3000	573.02	288.73	242.89	108.69
4000	1355.68	676.64	371.30	251.45

Tabela 3.5: Tempo de processamento(em segundos) obtidos na execução do programa em até 8 processadores com a multiplicação matricial bloco-linha vezes matriz(SCILAB)

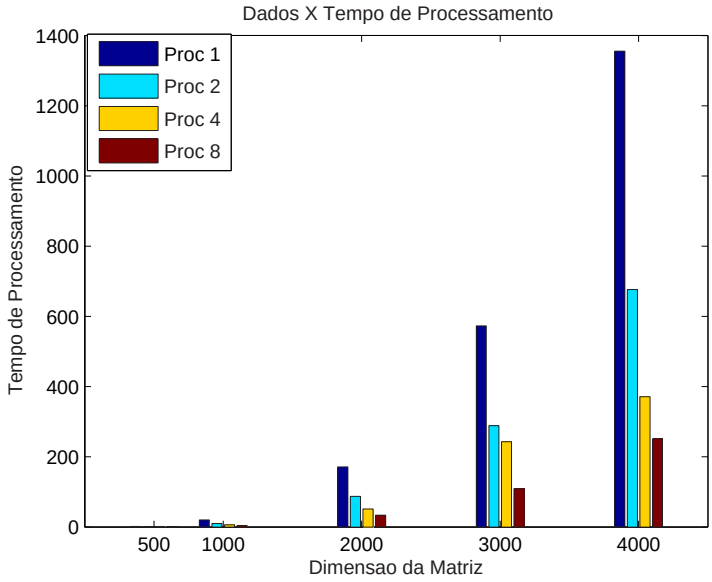

Figura 3.19: Tempo de processamento obtido da tabela 3.5 do interpretador SCILAB

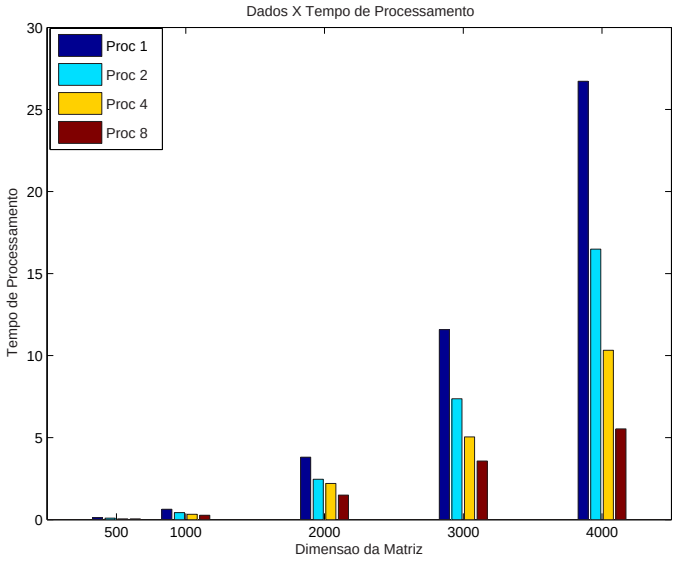

Figura 3.20: Tempo de Processamento obtido da tabela 3.6da Linguagem C

	C -Tempo em segundos				FORTRAN - Tempo em segundos			
$N \downarrow Nprocs \rightarrow$	1 Proc	2 Proc	4 Proc	8 Proc	1 Proc	2 Proc	4 Proc	8 Proc
500	0.1211	0.0932	0.0741	0.0502	0.089	0.060	0.053	0.037
1000	0.6293	0.4268	0.3247	0.2647	0.503	0.331	0.299	0.208
2000	3.8092	2.4610	2.2059	1.4950	3.381	2.152	1.651	1.232
3000	11.5813	7.3698	5.0441	3.5723	10.741	6.131	3.983	2.431
4000	26.7167	16.4849	10.3265	5.5310	24.696	13.963	7.498	5.231

Tabela 3.6: Tempo de processamento(em segundos) obtidos na execução do programa em até 8 processadores para a multiplicação matricial bloco-linha vezes matriz (C e FORTRAN)

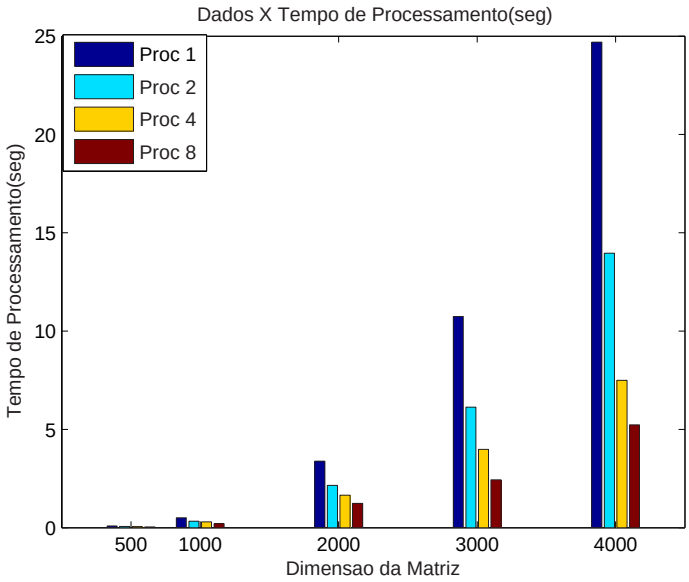

Figura 3.21: Tempo de Processamento obtido da tabela 3.6da Linguagem FORTRAN

Nos gráficos das figuras 3.19, 3.20 e 3.21, verificamos que à medida que aumentamos a dimensão das matrizes, os tempos de processamento aumentam, mas à medida que disponibilizamos processadores para execução das tarefas, há uma diminuição do tempo de processamento fazendo com que o Speedup experimental obtido se aproxime do Speedup ideal.

	Gráfico de Speedup - SCILAB			
	1 Proc	2 Proc	4 Proc	8 Proc
500	1.00	1.52	2.07	2.29
1000	1.00	1.97	3.20	4.77
2000	1.00	1.96	3.32	5.04
3000	1.00	1.98	3.35	5.39
4000	1.00	2.00	3.65	5.69

Tabela 3.7: Speedup obtido a multiplicação matricial bloco-linha vezes matriz

	Gráfico de Speedup - C				Gráfico de Speedup - FORTRAN			
	1 Proc	2 Proc	4 Proc	8 Proc	1 Proc	2 Proc	4 Proc	8 Proc
500	1.00	1.2990	1.6342	2.3930	1.00	1.483	1.679	2.400
1000	1.00	1.4740	1.6775	2.3774	1.00	1.519	1.680	2.418
2000	1.00	1.5478	1.7268	2.5479	1.00	1.571	2.047	2.744
3000	1.00	1.5714	2.2960	3.2419	1.00	1.751	2.696	4.418
4000	1.00	1.6206	2.5871	4.8303	1.00	1.768	3.293	4.721

Tabela 3.8: Speedup obtido para a multiplicação matricial bloco-linha vezes matriz(C e FORTRAN)

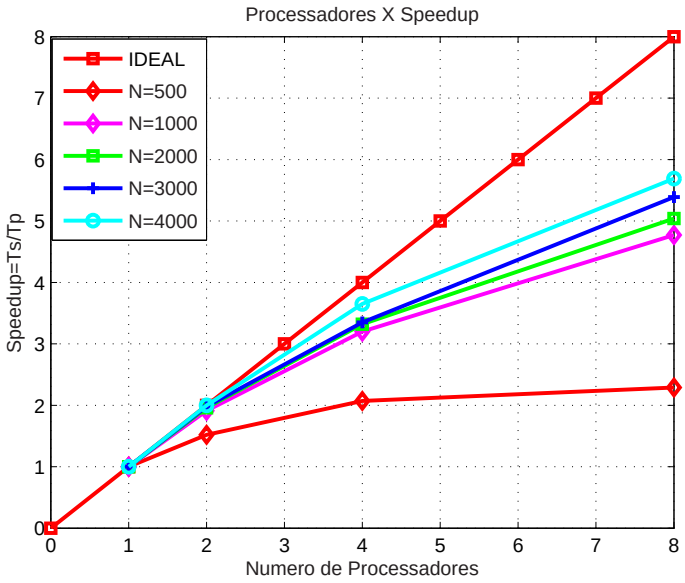

Figura 3.22: Gráfico do Speedup obtido para a multiplicação matricial bloco-linha vezes matriz de acordo com a tabela 3.7 do interpretador SCILAB

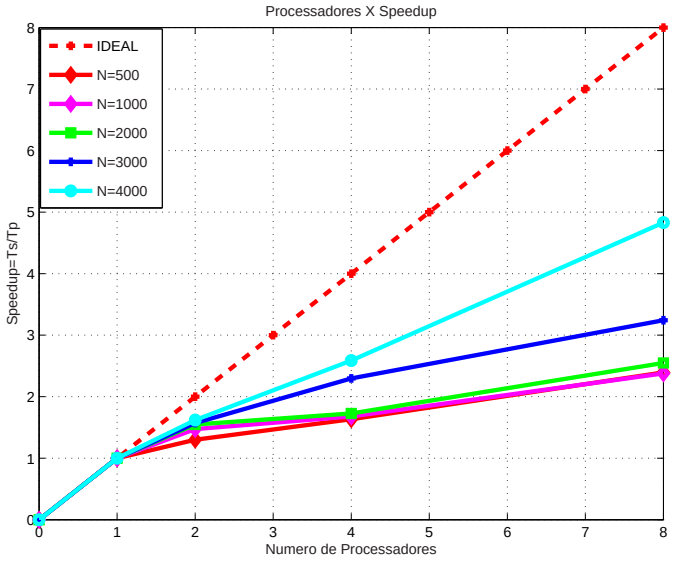

Figura 3.23: Gráfico do Speedup obtido pela multiplicação matricial bloco-linha vezes matriz de acordo com a tabela 3.8 da Linguagem C

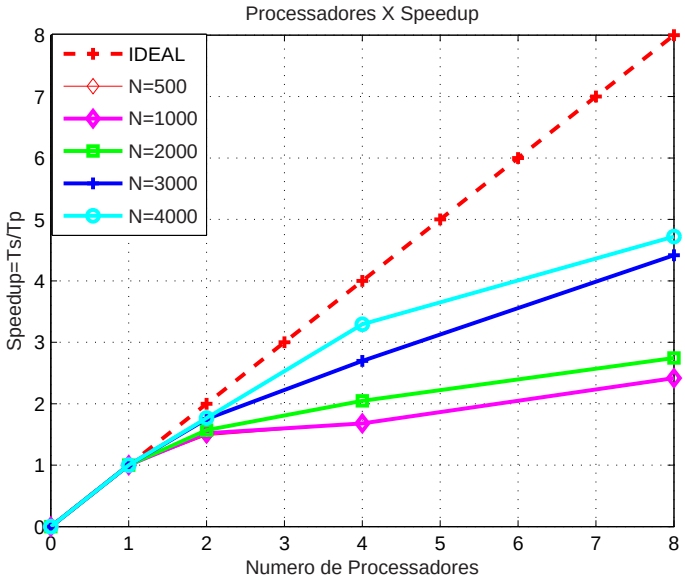

Figura 3.24: Gráfico do Speedup obtido pela a multiplicação matricial bloco-linha vezes matriz de acordo com a tabela 3.8 da Linguagem FORTRAN

Nos gráficos das figuras 3.23 e 3.24, o Speedup em ambas linguagens C e FORTRAN são muito parecidos, obtendo-se uma vantagem mínima na Linguagem Fortran. A partir dos resultados dos testes, algumas outras observações merecem destaque. Comparando as curvas de Speedup obtidas pelo SCILAB (figura 3.22) e pelas linguagens C e FORTRAN (figura 3.23 e 3.24), observa-se que a do SCILAB é que mais se aproxima ao caso ideal de Speedup linear, ao passo que as curvas do FORTRAN e, em especial do C, apresentam desvios maiores. As razões por trás deste comportamento têm a ver com a otimização de compiladores e bibliotecas, e, sem dúvida, se otimizarmos os códigos das três linguagens utilizadas, teríamos desempenhos melhores, possivelmente aproximando todas as curvas ao caso ideal de Speedup linear.

Alguns testes foram realizados à parte nesta dissertação e verificamos que a multiplicação matricial com o algoritmo 3.1, utilizando **laços iterativos** para as três linguagens, o interpretador SCILAB demonstrou tempo de processamento muito maior quando comparado com as linguagens C e FORTRAN. Por outro lado, o desempenho em paralelo apresentou-se ganho, muito semelhante aos dos gráficos das figuras anteriores presentes neste capítulo.

O SCILAB e Fortran possuem um grande número de funções intrínsecas ou pré-definidas. Nas funções intrínsecas para multiplicações de duas matrizes, podemos utilizar o operador *(*)* para SCILAB e *matmul* para FORTRAN. No entanto, foram realizados testes utilizando essas funções intrínsecas e constatamos que: o tempo de processamento para a simulação em FORTRAN foi mais lento que com o Scilab utilizando o operador * mas, o mesmo comportamento, conforme o paragráfo anterior, para a função de Speedup. Os gráficos de Speedup permaneceram semelhantes aos gráficos nas figuras 3.23 , 3.24 e figura 3.22.

Os resultados acima demonstram a viabilidade da utilização do SCILAB para testes preliminares de speedup, no sentido de prova-de-conceito, de algoritmos paralelos, processo usualmente denominado prototipagem.

Capítulo 4

Método para Resolução de Sistemas Lineares

4.1 Introdução

Neste capítulo é apresentado um método simples para resolução de sistemas lineares conhecido como *Método de Jacobi*, (veja Ortega [18], utilizado como referência principal, ou Freeman [4], Heath [19], Stoer e Bulirsch [20], Pacheco [21]). Este método é utilizado para resolução de sistemas lineares de dimensão elevada. Na seção 4.2 foram apresentados conceitos referentes ao método clássico de Jacobi. Na seção 4.3 foi apresentada a equação de Poisson e fundamentos usados nos testes discretizando a equação de Poisson para obter um sistema linear ao qual se aplica o método de Jacobi. Na seção 4.4 é apresentado o método de Jacobi em pa-ralelo, a forma de particionamento dos dados, detalhes da implementação do referido método e o pseudo-código em pa-ralelo. Em seguida na seção 4.5 são apresentados critérios de convergência para o método. E finalmente na seção 4.6 tratamos da análise das simulações para o método de Jacobi utilizando a *API PVM - Parallel Virtual Machine* implementadas nas linguagens compiladas C e FORTRAN e na linguagem interpretada SCILAB.

4.2 Método de Jacobi

Embora o método de Jacobi tenha uma convergência lenta para várias classes de sistemas lineares, este método fornece um ponto de partida conveniente para discussão de métodos iterativos e, mais ainda, métodos paralelizáveis [18].

Seja uma matriz $A^{n\times n}$ não singular e b um vetor em $\mathbb{R}^n$, e o sistema a ser solucionado dado por:

$$Ax = b \tag{4.1}$$

Supõe-se que os elementos diagonais a_{ii} da matriz A sejam não-nulos. O método de Jacobi é descrito pela equação a seguir:

$$x_i^{k+1} = \frac{1}{a_{i,i}}(-\sum_{j\neq i}^{n} a_{i,j}x_j^k + b_i), \qquad i = 1, ..., n, \qquad k = 0, 1, ... \tag{4.2}$$

onde o índice(k) indica o número de iterações. Espera-se, como em todos os métodos iterativos, que $x_1^0, ..., x_n^0$ seja uma seqüência de aproximações cada vez mais próximas à solução do sistema 4.1. É conveniente reescrever a equação 4.2 na forma matricial. Seja $D = diag(a_{11}, ..., a_{nn})$ uma matriz contendo os elementos da diagonal de A, e seja $B = D - A$. Podemos reescrever a equação 4.2 como:

$$x_{k+1} = Hx_k + d, \qquad k = 0, 1, ..., \qquad H = D^{-1}B, \qquad d = D^{-1}b \tag{4.3}$$

O método de Jacobi nem sempre converge, mas existem algumas condições suficientes para convergência, freqüentemente satisfeitas na prática. Nesse sentido, citam-se dois teoremas clássicos de convergência [18], [19]:

Teorema 4.1. Se a matriz A for estritamente diagonal dominante, isto é, $\mid a_{jj} \mid > \sum_{\substack{i=1\\ i\neq j}}^{n} a_{ij}, 1 \leq j \leq n$, ou irredutivelmente diagonal dominante (uma matriz é irredu-

tivelmente diagonal dominante se ela é estritamente diagonal dominante e irredutível [1]), a iteração de Jacobi converge para qualquer x^0 inicial .

Teorema 4.2. Seja $A = D - B$ uma matriz simétrica e positiva definida. A iteração do método de Jacobi converge para qualquer x^0 inicial, se e somente se $D + B$ for uma matriz positiva definida [18].

A partir da equação (4.3) verificamos que a operação básica utilizada nesta equação é uma multiplicação de uma matriz com um vetor. Segundo Ortega [18], Korfgen e I.Gutheil [22] a eficácia da implementação de uma multiplicação de uma matriz por um vetor, depende muito da estrutura da matriz H. Os métodos iterativos em geral, possuem vantagens quando A, e portanto H, é uma matriz esparsa de dimensão elevada. Matrizes deste tipo, por exemplo, surgem como resultados de discretização, por diferenças finitas ou métodos de elementos finitos, de problemas de equações diferenciais parciais elíticas.

4.3 Equação de Poisson

A EDP elítica [22], [18], [20] conhecida como *equação de Poisson* é dada abaixo pela equação

$$u_{xx} + u_{yy} = f(x, y) \tag{4.4}$$

onde (x, y) pertence à fronteira $\Omega = [0, 1] \times [0, 1]$ e u é dado pela fronteira do domínio Ω. f é uma determinada função de x e y e se $f \equiv 0$, a equação 4.4, é denominada equação de Laplace. Discretizamos um quadrado unitário como ilustrado na figura 4.2 com um espaçamento de malha h. Seja $u_{i,j}$ a solução da equação (4.4) no ponto de malha (ih, jh). Pela aproximação das derivadas da equação 4.4 utilizando diferenças centrais obtêm-se o sistema de equações 4.5.

[1] $A^{n \times n}$ é irredutível se e somente se $(I+ \mid A \mid)^{n-1} > 0$.

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = h^2 f_{i,j}, \qquad i,j = 1,...,n \tag{4.5}$$

Assumimos com a equação 4.5, que u é especificada na fronteira de seu domínio de definição.

Para essas aproximações deve-se considerar em geral, os valores das células adjacentes. E nesse caso, o nível de dependência de dados entre as células da malha, após a discretização, pode ser representada usando um estêncil ou *(molécula computacional)*. O estêncil empregado é o de 5-pontos, representado conforme a figura 4.1 e 4.2.

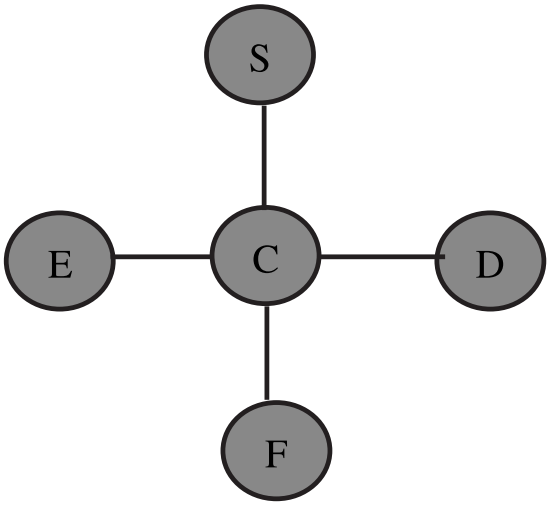

Figura 4.1: Estêncil de 5-pontos(molécula computacional).

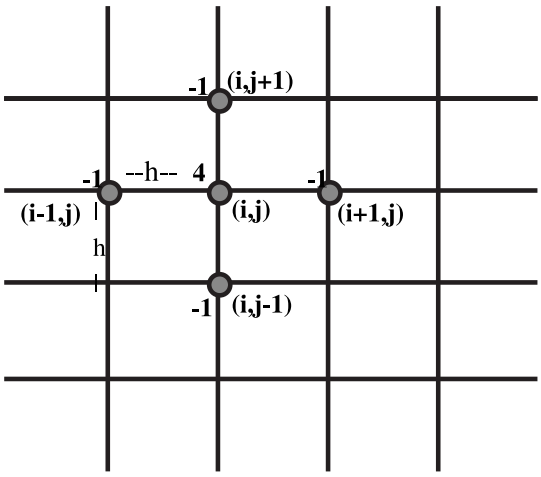

Figura 4.2: Representação da grade do Estêncil de 5-pontos

Com a discretização usando o estêncil de 5-pontos, o valor de uma célula no domínio depende do valor central(C) e do valor de seus quatro vizinhos, que são: esquerdo(E), direito(D), superior(S) e fundo (F). O estêncil computacional define a posição e valor de cada elemento na matriz.

Podemos escrever o sistema de equações 4.5 na forma $Ax = b$ sendo $x_k = u_{k,1}, k = 1,...,N$ incógnitas na primeira linha dos pontos de malha como $x_k = u_{k-N,2}, k = N+1,...,2N$ incógnitas na segunda linha da malha e assim por diante. Em outras

palavras:

$$x^t = (u_{1,1}, ..., u_{N,1}, u_{1,2}, ..., u_{N,2}, ..., u_{1,N}, ..., u_{N,N}) \tag{4.6}$$

A matriz A é bloco tridiagonal com as sub e super-diagonais compostas de matrizes identidade $(-I)$, e a diagonal principal de uma matriz banda T compostas de coeficientes do estêncil (veja equação 4.8).

$$\mathbf{A} = \begin{pmatrix} T & -I & & \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & T \end{pmatrix} \tag{4.7}$$

$$\mathbf{T} = \begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 4 \end{pmatrix} \tag{4.8}$$

O lado direito da igualdade no sistema $Ax = b$, consiste nas quantidades $-h^2 f_{i,j}$ também quanto os valores de fronteira conhecidos nas posições apropriadas.

Na matriz apresentada em (4.8) temos um exemplo de matriz esparsa do tipo banda, com cinco diagonais não nulas, para a dimensão $N = 9$. Aplicando o método de Jacobi à equação (4.5) obtêm-se:

$$u_{ij}^{k+1} = \frac{1}{4}[u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - h^2 f_{i,j}], \qquad i,j = 1, ..., N, \qquad K = 0, 1, ... \tag{4.9}$$

Para a equação de Laplace, ($f \equiv 0$), a equação (4.9) mostra que a atualização do valor do ponto da malha (i, j) é exatamente a média dos valores anteriores dos vizinhos do ponto da malha (veja figura 4.2).

O pseudo-código da figura 4.3 ilustra o método de Jacobi (descrito pela equação 4.5) na linha **8**.

Figura 4.3: Algoritmo do método de Jacobi

Algoritmo : Método de Jacobi

1 *Escolher o vetor x_ inicial* $\mathbb{R}^n$,

2 *Escolher o tol* //**tol= tolerância**

3 *Escolher o maxiter* //**maxiter máximo de iterações**

4 $iter\_num == 0$

5 $Enquanto((iter\_num < maxiter) and (residuo(x\_atual, x\_ant, Ax - b, n) \leq tol))$ *então*

6 *for k= 0 até max do*

7 *for i=1 até n do*

8 *Fazer* $x_i^{k+1} = \frac{1}{a_{i,i}}(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^k - \sum_{j=i+1}^{n} a_{i,j}x_j^k)$

9 *end for*

10 *end for*

11 *Fazer* $x^k = x^{k+1}$

12 *FIM*

13 *Apresentar x;*

A linha **5** do pseudo-código será detalhada na seção 4.5.

4.4 Método Paralelo Iterativo de Jacobi

Pela equação 4.9 vemos que para cada par (i, j), a variável correspondente $u_{i,j}$ pode ser computado ao mesmo tempo que se realiza o cálculo para outro par com (i, j) distinto. Por esta razão, o método de Jacobi é considerado um protótipo típico

de um método paralelo.

Pode-se estender esta idéia de paralelismo a uma matriz A e vetor $x\_inicial$ particionada em bloco-linhas, com a partição conformal do lado direito b. Explica-se esta idéia na implementação correspondente a seguir.

4.4.1 Implementação do Método Paralelo Iterativo de Jacobi

Em termos do modelo de computação paralela, utilizamos o modelo mestre-escravo (capítulo 2), no qual um dos processadores, o mestre, se encarrega de tarefas administrativas, tais como a distribuição de dados escalares, matrizes e vetores.

O processador mestre (P_0) é responsável pelo envio dos dados escalares n, tol, $maxiter$, da matriz A, do vetor b e do vetor x^0, dos quais a matriz e os vetores são divididos em bloco-linhas de tamanho $n/nprocs$ assumindo que $n \geq nprocs$ conforme a figura 4.4.

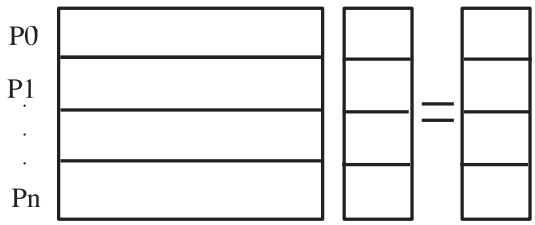

Figura 4.4: Divisão da Matriz

Após o envio dos dados pelo processador mestre (P_0), os processadores escravos $(P_1$ *a* $P_{n-1})$ recebem os dados alocados a cada um e em seguida iniciam os cálculos. A cada iteração é feita uma comunicação entre os processadores. São enviados os vetores $x\_parciais$, que são as soluções parciais pertencentes a cada processador, para que sejam atualizados. Cada processador, ao receber os subvetores procedentes de outros processadores concatenam os subvetores em um vetor x utilizado para o cálculo do próximo subvetor, conforme a linha 10 do pseudo-código apresentado em 4.5.

Na atualização dos vetores, para garantir o sincronismo entre os processadores,

a função do *PVM*, *pvm_barrier*, foi usada para garantir que todos os processadores escravos recomeçassem os cálculos no mesmo momento.

Depois de verificar os critérios referentes à seção 4.5, os processadores escravos enviam para o processador mestre o vetor x com resultado final.

Figura 4.5: Código Paralelo Jacobi

Algoritmo : Método Paralelo Jacobi

1 *Particionar por blocos de linhas a matriz A e enviar para os processadores*

2 *Escolher o vetor x_inicial* $\mathbb{R}^n$*, particionar por linhas e enviar para os processadores*

3 *Particionar por linhas e enviar o vetor b para os processadores*

4 *Escolher o tol* // **tol = tolerância**

5 *Escolher o maxiter* //**maxiter = máximo de iterações**

6 $iter\_num == 0$

7 $Enquanto((iter\_num < maxiter) and (residuo(x\_atual, x\_ant, Ax - b, n) \leq tol))$ *então*

8 *for k= 0 até max do*

9 *for i=1 até n/nprocs do*

10 *Resolva* $A_{ii}x_i^{k+1} = (b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} A_{i,j}x_j^k)$

11 *end for*

12 *end for*

13 *Fazer* $x^k = x^{k+1}$ *e atualizar com a concatenação de x entre todos os processadores*

14 *FIM*

15 *Apresentar x*

16 *Recomeçar os cálculos até que* $(residuo(x\_atual, x\_ant, Ax - b, n) \leq tol)$

Na figura 4.6 contém trecho do código do método de Jacobi onde visualizamos a implementação em Scilab.

Figura 4.6: Código Slave.sci

Algoritmo : Trecho de código do método de Jacobi-Slave.sci

```
1   stacksize(400000000);
2   parent = pvm_parent();
3   n = pvm_recv( parent, 0 );
4   tol = pvm_recv( parent, 1 );
5   max_iter= pvm_recv( parent, 2 );
6   A = pvm_recv( parent, 3 );
7   b = pvm_recv( parent, 4 );
8   xinicial = pvm_recv( parent, 5 );
9   x(:,1) = x0;
10    r = b - A*x(:,1);
11    n = size(A,1);
12       for i = 1:n,
13          D(i,i) = A(i,i);
14       end
15    iter_num = 1;
16       while((iter_num < maxiter)and(residuo(x_atual, x_ant, Ax - b, n) ≤ tol))
17          x(:,iter_num+1) = x(:,iter_num) + inv(D)*r(:,iter_num);
18          r(:,iter_num+1) = r(:,iter_num) - inv(D)*A*r(:,iter_num);
19         iter_num = iter_num + 1;
20       end
21    iter_num = iter_num - 1;
22    pvm_send( parent, x, 6 );
23    pvm_exit();
24    exit(0);
```

Na linguagem interpretada Scilab, as linhas *17 e 18* apresentada na figura 4.6 é uma multiplicação de uma matriz por um vetor da qual utiliza-se a função intrínseca *inv*, que realiza a inversão da matriz e, multiplica esta matriz por um vetor por meio do operador *(*)*. Verificamos que Scilab é uma ferramenta numérica que por meio de funções internas permite a praticidade e facilidade na programação economizando tempo para a implementação de um algoritmo.

4.5 Regra de parada do método iterativo

Seja x^k uma seqüência de iterações, gerada por um método iterativo. Um critério de parada indicando convergência da seqüência é:

$$\| x^{k+1} - x^k \| \leq \varepsilon \tag{4.10}$$

Outra possibilidade é verificar se a norma do resíduo está abaixo de uma tolerância ε especificada:

$$\| Ax^k - b \| \leq \varepsilon \tag{4.11}$$

Nesta dissertação, optou-se pelo uso conjunto dos dois critérios, além de não permitir que o número de iterações ultrapassasse um número máximo pré-especificado. Ou seja, o algoritmo detecta convergência através da satisfação simultânea de ambos os critérios (4.10) e (4.11).

4.6 Resultados Comparativos utilizando o método iterativo de Jacobi

Nesta seção apresentamos resultados de simulações computacionais para o método de Jacobi que têm por objetivo comparar o tempo de processamento, os speedups, utilizando as linguagens compiladas (C e FORTRAN) e a linguagem interpretada

(SCILAB). Em todos os testes, fizemos uso de matrizes de Poisson de dimensões *N=484, N=1024, N=1936, N=2916 e N=4096.*

Nas tabelas 4.1 e 4.2 podemos visualizar os tempos de processamento gasto durante as execuções do método de Jacobi. Para este problema típico de computação científica, as medições de tempo se referem à média aritmética dos tempos obtidos em 20 (vinte) execuções para cada tipo de linguagem.

	Scilab - Tempo em segundos			
$N \downarrow Nprocs \rightarrow$	1 Proc	2 Proc	4 Proc	8 Proc
484	2.497	1.367	0.89	0.68
1024	22.58	12.233	8.038	5.98
1936	106.66	57.187	37.99	27.67
2916	253.34	128.973	89.56	56.78
4096	617.19	331.18	193.00	125.48

Tabela 4.1: Tempo de processamento(dado em segundos) obtidos na execução do programa Jacobi pelo interpretador SCILAB

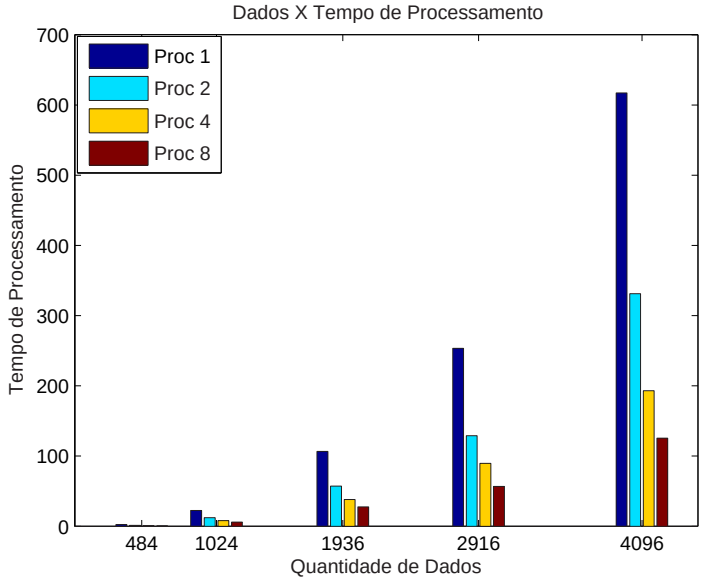

Figura 4.7: Comparação do tempo de execução para 1, 2, 4, 8 processadores do método de Jacobi(SCILAB)

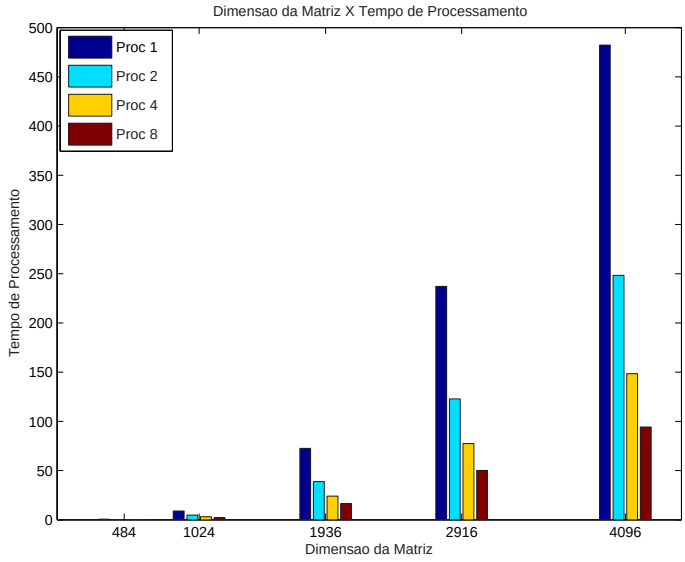

Figura 4.8: Comparação do tempo de execução para 1, 2, 4, 8 processadores do método de Jacobi em C

	C - Tempo em segundos				Fortran - Tempo em segundos			
$N \downarrow Nprocs \rightarrow$	1 Proc	2 Proc	4 Proc	8 Proc	1 Proc	2 Proc	4 Proc	8 Proc
484	0.667	0.359	0.228	0.149	0.513	0.276	0.176	0.127
1024	9.004	4.849	3.077	2.152	5.475	2.946	1.871	1.308
1936	72.646	38.807	24.127	16.552	61.062	32.274	20.253	13.893
2916	237.167	122.75	77.429	50.119	196.525	101.301	63.952	41.356
4096	482.317	248.36	148.45	94.326	397.11	204.38	118.47	77.350

Tabela 4.2: Tempo de processamento(dado em segundos) obtidos na execução do programa Jacobi - C e FORTRAN

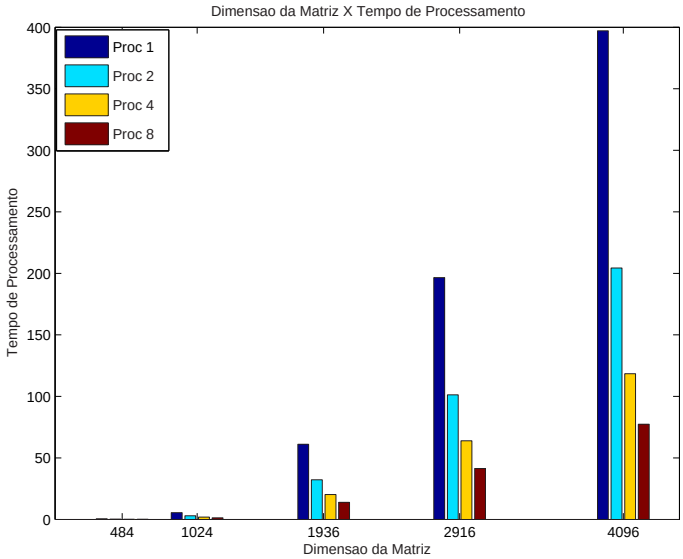

Figura 4.9: Comparação do tempo de execução para 1, 2, 4, 8 processadores do método de Jacobi(FORTRAN)

4.6 Resultados Comparativos utilizando o método iterativo de Jacobi

Por meio das tabelas 4.3 e 4.4 apresentamos as tabelas de Speedup de onde obtemos os gráficos das figuras 4.10, 4.11 e 4.12. Esses gráficos ilustram o Speedup pelo tempo de processamento de um processador em função da quantidade de processadores utilizados. Diante destes gráficos verificamos que independente da linguagem, o comportamento da curva de Speedup entre eles possuem características semelhantes. Entre os gráficos das figuras 4.11 e 4.12 verificamos que com a linguagem Fortran obtemos um ganho mínimo em relação à linguagem C. Por outro lado, o gráfico da figura 4.10 quase se aproxima da reta ideal de Speedup.

	Gráfico de Speedup - Scilab			
	1 Proc	2 Proc	4 Proc	8 Proc
484	1.00	1.826	2.800	3.672
1024	1.00	1.846	2.809	3.775
1936	1.00	1.865	2.807	3.854
2916	1.00	1.967	2.828	4.461
4096	1.00	1.863	3.190	4.918

Tabela 4.3: Speedup obtido Jacobi- Scilab

	Gráfico de Speedup - C				Gráfico de Speedup -Fortran			
	1 Proc	2 Proc	4 Proc	8 Proc	1 Proc	2 Proc	4 Proc	8 Proc
484	1.00	1.858	2.925	4.476	1.00	1.858	2.913	4.039
1024	1.00	1.856	2.926	4.184	1.00	1.858	2.926	4.186
1936	1.00	1.872	3.011	4.389	1.00	1.892	3.015	4.395
2916	1.00	1.932	3.063	4.732	1.00	1.940	3.073	4.752
4096	1.00	1.942	3.249	5.042	1.00	1.943	3.352	5.476

Tabela 4.4: Speedup obtido Jacobi-C e FORTRAN

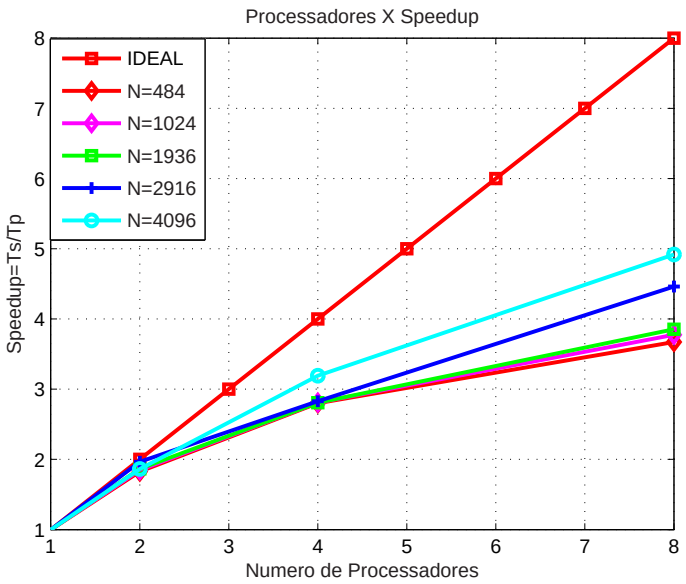

Figura 4.10: Speedup obtido conforme o tempo de execução da figura 4.10(SCILAB)

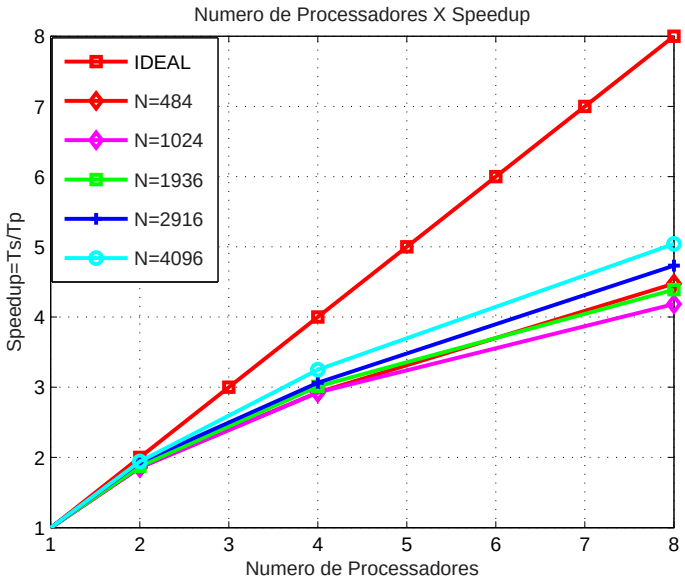

Figura 4.11: Speedup obtido conforme o tempo de execução da figura 4.10(C)

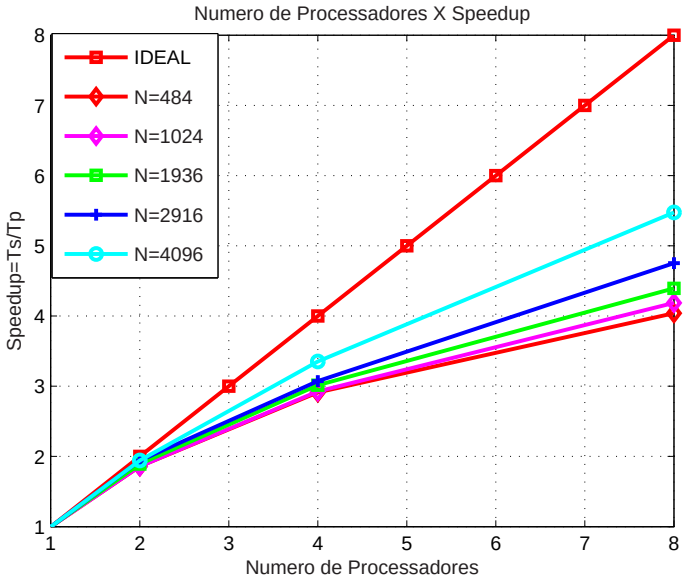

Figura 4.12: Speedup obtido conforme o tempo de execução da figura 4.12(em FORTRAN)

Verificamos que a linguagem interpretada SCILAB exige tempo de processamento um pouco maior (tabela 4.1) se comparado com as linguagens compiladas. Entretanto, do ponto de vista de prototipagem (ou prova de conceito) de um algo-

ritmo paralelo, no que concerne convergência e speedup, percebe-se que o comportamento qualitativo é o mesmo das linguagens compiladas. Conclui-se que é válido utilizar o ambiente interpretado SCILAB para o projeto e teste de protótipos de algoritmos, uma vez que SCILAB possui diversas funções intrínsecas que facilitam programação, permite a integração de funções externas de outras bibliotecas em sua interface e ainda permite que a programação de determinados problemas sejam programadas em um menor tempo de programação.

Capítulo 5

Ordenação Quicksort utilizando Topologia Hipercúbica

O método de ordenação Quicksort inventado por Hoare em 1962 [23], [24] é conhecido como o algoritmo mais rápido dentre os outros métodos de ordenação como *Bolha, Inserção, Seleção, Merge-Sort e ShelSort*[24], cuja finalidade é ordenar um vetor de números de forma ascendente ou descendente.

Este capítulo tem por finalidade avaliar o método de ordenação Quicksort em ambiente paralelo utilizando a linguagem interpretada SCILAB e as linguagens compiladas C e FORTRAN. Na seção 5.1 apresentamos o algoritmo Quicksort seqüencial detalhando os princípios básicos como a escolha do pivô, o algoritmo de partição e enfim o algoritmo completo. Na seção 5.2 apresentamos o método de ordenação Quicksort Paralelo usando a topologia hipercúbica: na subseção 5.2.1 descrevemos a topologia hipercúbica e o modo em que é realizada a comunicação entre os processadores; na subseção 5.2.2 apresenta o algoritmo Quicksort paralelo. Por fim, na seção 5.3 apresentamos os resultados obtidos nas implementações em SCILAB, C e FORTRAN.

5.1 Algoritmo Quicksort

O algoritmo Quicksort é um método de ordenação de um vetor de números. Este método tem como princípio o paradigma de dividir para conquistar, e portanto seu tempo de execução no pior caso é $O(n^2)$, na tarefa de ordenação de uma seqüência de entrada de n números. Embora um algoritmo cuja complexidade é $O(n^2)$ possa ser considerado ruim (Ziviani [24] e Prado [23]), o Quicksort se destaca de outros algoritmos porque seu tempo médio de execução é muito mais próximo do melhor caso do que do pior caso.

O processo de ordenação utiliza a função ***particiona*** apresentada no algoritmo da tabela 5.2. Dada uma seqüência de n itens e uma escolha de um **pivô**, a função particiona armazena em um subvetor os elementos menores que o pivô e em outro vetor os elementos maiores que o pivô.

O algoritmo Quicksort apresentado na figura 5.1 apresenta em sua estrutura uma característica recursiva, ou seja, após um processo de partição de um vetor em subvetores, cada subvetor por sua vez, é submetido ao mesmo procedimento realizado anteriormente. Claramente, o método *Quicksort* utiliza o paradigma da *Divisão e Conquista* através da função ***particiona*** apresentada na figura 5.2.

Figura 5.1: Algoritmo QuickSort

```
Quicksort(seqüência, inicio , fim)
1       se fim-inicio = 1 ;
2             retorna ;
3       fim se;
4       pivo = particiona(seqüência, inicio, fim);
5       Quicksort(seqüência, inicio, pivo-1);
6       Quicksort(seqüência, pivo+1, fim);
end algoritmo
```

A definição adequada da função de particionamento é determinante no sucesso e

na rapidez do algoritmo. Uma definição possível da função *particiona* é o rearranjo da seqüência de entrada $A[Esq, Dir]$ pela escolha aleatória de um item x chamado *pivô* da seqüência, de tal forma que ao final a seqüência A estará particionado em uma parte esquerda com chaves menores ou iguais a x, e uma parte direita com chaves maiores ou iguais a x. O particionamento é feito definindo duas variáveis que tenham a finalidade de guardar o deslocamento dos índices sendo um utilizado pela direita e outro utilizado pela esquerda de forma que essa varredura na seqüência se desloque comparando elementos com o *pivô* obtido até se encontrarem ao meio.

Figura 5.2: Algoritmo Particiona

Algoritmo : Particiona(seqüência, esq, dir)

1 ***Escolher aleatoriamente um item do vetor e colocar em x (pivô);***

2 ***Varrer o vetor a partir da esquerda até que encontrar um item do vetor[i]>=x;***

3 ***Da mesma forma percorrer o vetor a partir da direita até encontrar um item do vetor[j]<=x;***

4 ***Os itens do vetor[i] e do vetor[j] neste estágio encontra-se fora de lugar, então devemos trocá-los;***

5 ***Continuar o processo acima até que i e j se cruzem em algum ponto do vetor;***

6 ***Ao final o vetor[esq e dir] estará particionado na seguinte forma:***

7 ***Os itens vetor[esq], vetor[esq+1],..., vetor[j] serão menores ou iguais a x;***

8 ***Os itens vetor[i], vetor[i+1],..., vetor[dir] serão maiores que x;***

A escolha aleatória do pivô também é fundamental para que o método de ordenação Quicksort funcione como o melhor algoritmo de ordenação, pois, é o *pivô* quem define de onde começará a escolha do índice do particionamento *subrotina partição*. A forma de escolha do pivô *item* x é muito importante nesse algoritmo pois pode amenizar de alguma forma a ordenação facilitando o tempo de computação, ou seja

tornando-o mais eficaz.

Para a escolha do pivô pode-se aplicar um dos três critérios específicos. O primeiro critério consiste em escolher o pivô utilizando o primeiro elemento do vetor $A[0..n-1]$, que em algumas situações poderia levar ao pior caso se encontrássemos o *pivô* sendo o menor ou o maior elemento de todo vetor, pois o particionamento percorreria todo o vetor e não haveria em subdivisão dos vetores em esquerda e direita (*subrotina particiona*). O segundo critério seria escolher a posição do pivô aleatoriamente e posteriormente achando o elemento contido nesta posição. Segundo Prado [23], Ziviani [24] e Jaja [25], escolher o pivô aleatoriamente cairia no melhor caso assim como, o terceiro critério comumente conhecido como *mediana* que soma a primeira e a última posição da seqüência de entrada e divide por dois, que será o elemento chamado de pivô. Esse critério de escolha de pivô é chamado de mediana e será utilizado nesta dissertação.

5.2 QuickSort Paralelo utilizando a Topologia Hipercúbica

5.2.1 Topologia Hipercúbica

A palavra topologia se refere ao grafo de ligações entre processadores que permitam o tráfego de informações. Existem topologias que são comumente usadas nas redes de computadores que se baseiam na estrutura de conexão entre vários computadores. Essas topologias de rede são conhecidas como *(topologia em anel, estrela, mesh, hipercubo dentre outros)*.

A rede hipercúbica é uma topologia de rede que será utilizada no método de ordenação QuickSort. A topologia hipercúbica consiste em organizar os processadores em forma de um cubo de dimensão d tal que o *número de processadores* $= 2^d$. As figuras 5.3 e 5.4 são exemplos para d=0, d=1, ...,d=3.

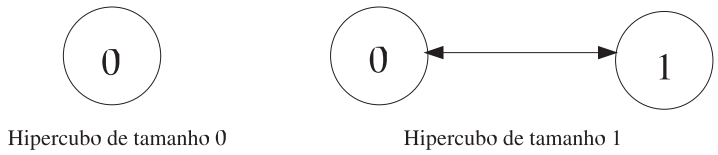

Figura 5.3: Comunicação no Hipercubo entre um e dois processadores cujo rótulo de comunicação são representações binárias dos inteiros 0 até 1

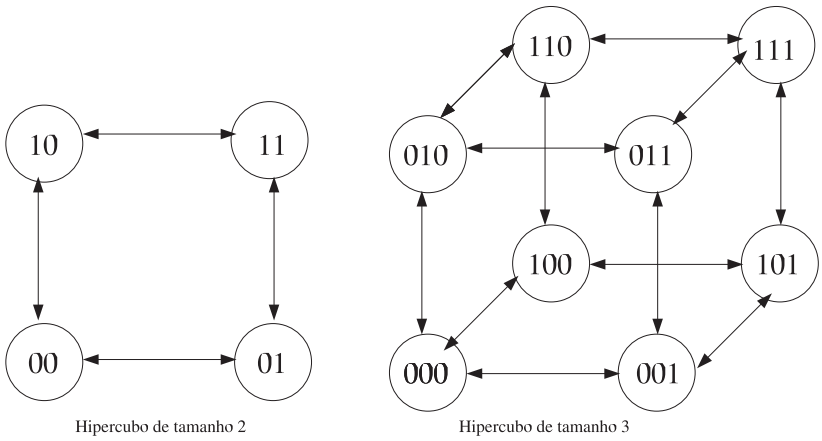

Figura 5.4: Comunicação no Hipercubo entre quatro e oito processadores cujo rótulo de comunicação são representações binárias dos inteiros 2 até 3

Na figura 5.5 visualizamos que a conexão dos processadores são realizados de acordo com a dimensão d do cubo e cada processador possui uma representação binária $labels = d$ que facilita a identificação dos processadores vizinhos que cada um pode iteragir na comunicação. Nesta mesma figura, visualizamos que cada pro-

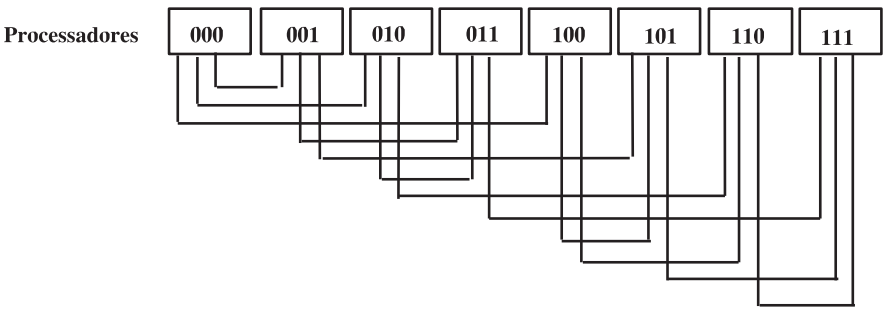

Figura 5.5: Conexão do Cubo - Código Gray

cessador podem comunicar com seus vizinhos desde que no mapeamento da repre-

sentação binária esse processador tenha a diferença de um único bit representado pela figura 5.5.

5.2.2 QuickSort Paralelo

O método de ordenação Quicksort possui várias formas de ser paralelizado. Na presente dissertação utilizaremos o algoritmo Quicksort na topologia Hipercúbica. O algoritmo da figura 5.6, foi proposto por Grama et al. [6].

Seja n o número de elementos a ser ordenado, $nprocs = 2^d$ onde $nprocs$ representa o número de processadores e d a dimensão do hipercubo.

O processador P_0 cuja representação binária é 000 inicia dividindo a seqüência n em $n/nprocs$ para todos os processadores assumindo que $n > nprocs$, e depois envia essas subseqüências para todos os processadores.

O processador P_0 cuja representação binária é 000 seleciona o pivô aplicando o critério da mediana que consiste em selecionar a primeira e a última posição da seqüência, somá-las, e dividir a posição resultante por dois, obtendo a posição e o elemento chamado pivô. Após esta fase da escolha do pivô cada processador, recebendo o pivô, particiona os elementos locais em duas subseqüências, um com elementos menores ou iguais ao pivô e outro com elementos maiores que o pivô. As subseqüências são enviadas para os processadores vizinhos, cuja representação binária de cada sub-cubo difere de apenas um bit.

O processador de menor índice envia seus valores maiores que o pivô para seu vizinho e recebe do seu vizinho os valores menores do que o pivô.

Especificamente, os processadores que têm 0 em seu bit mais significativo da representação binária no rótulo dos processadores, ficam com a subseqüência de menores elementos, e os processadores com o bit mais significativo 1 ficam com a subseqüência dos elementos maiores. O processo utilizando a *a função particiona*, é dividir e encontrar os elementos seguindo o critério de particionamento no hipercubo

Figura 5.6: Algoritmo Paralelo Quicksort Hipercúbico

```
inicio
1     tid = processos com rotulo ;
2     para i=1 até d faça;
3     inicio;
4           x= pivô;
5           particione B em B1 e B2 onde B1 ≤ x < B2;
6           se i^th bit é 0 então;
7           inicio;
8                 envie B2 para o processador ao longo do i^th link de
comunicação;
9                 C = subseqüência recebida ao longo de i^th do link de
comunicação;
10                B = B1 ∪ C;
11          fim do se;
12          senão;
13                envie B1 para o processador ao longo de i^th do link de
comunicação;
14                C = subseqüência recebida ao longo de i^th do link de
comunicação;
15                B = B2 ∪ C;
16          fim do senão;
17    fim;
18    ordena B usando o quicksort seqüencial;
fim
```

de modo que as subseqüências sigam a padronização da representação binária para cada processador.

Após este passo, cada processador em $(d-1)$ dimensão do hipercubo, os processadores d^{th} rotulado com o bit mais significativo 0 terão elementos menores que

o pivô e cada processador no outro ($d-1$ dimensão) hipercúbico terão elementos maiores que o pivô. Este procedimento é realizado recursivamente em cada sub-cubo, dividindo mais as subseqüências e acrescentando à lista de itens anterior remanescente. Depois de d semelhantes divisões, um ao longo de cada dimensão do cubo, a seqüência de números estará ordenada executando localmente o algoritmo 5.1 em seqüêncial e os processadores devolvem a lista dos valores ordenados contidos em cada processador.

5.3 Simulação do QuickSort na Topologia Hipercúbica

Nesta seção avaliamos o método de ordenação Quicksort, cujas seqüências para testes foram seqüências randômicas de tamanhos de $40000, 60000, 100000, 350000$ e 400000. Esses resultados que iremos apresentar, são provenientes da média aritmética entre 20 (vinte) execuções para cada seqüência sendo avaliados para até 8 processadores. Nestes mesmos testes, analisamos o comportamento deste método por meio do tempo de processamento e análise de Speedup.

Nas tabelas 5.1 e 5.3 apresentamos os resultados referentes ao tempo de processamento que o algoritmo Quicksort utiliza durante a execução. Nestas tabelas, podemos verificar que o tempo gasto na execução do Quicksort hipercúbico utilizando a linguagem interpretada SCILAB foi um pouco maior. Na tabela 5.3 verificamos que o tempo de processamento identificado para a linguagem C foi menor se comparado à linguagem compilada FORTRAN.

	SCILAB - Tempo em segundos			
$N \downarrow Nprocs \rightarrow$	1 Proc	2 Proc	4 Proc	8 Proc
40000	0.4335	0.3863	0.2438	0.1743
60000	0.5309	0.4587	0.2928	0.2111
100000	0.6957	0.5404	0.3580	0.2544
350000	1.8617	1.3651	0.8007	0.6325
400000	2.1050	1.3701	0.8313	0.6558

Tabela 5.1: Tempo de processamento(em segundos) obtidos na execução do programa com até 8 processadores com o método Hipercubo Quicksort(SCILAB)

	Gráfico de Speedup - SCILAB			
	1 Proc	2 Proc	4 Proc	8 Proc
40000	1.00	1.1221	1.7776	2.4868
60000	1.00	1.1573	1.8132	2.5149
100000	1.00	1.2873	1.9432	2.7341
350000	1.00	1.3637	2.3251	2.9431
400000	1.00	1.5363	2.5321	3.2109

Tabela 5.2: Speedup obtido pelo Quicksort Hipercubo-SCILAB

	C - Tempo em segundos				FORTRAN - Tempo em segundos			
$N \downarrow Nprocs \rightarrow$	1 Proc	2 Proc	4 Proc	8 Proc	1 Proc	2 Proc	4 Proc	8 Proc
40000	0.0400	0.0299	0.0230	0.0117	0.0430	0.0317	0.0245	0.0175
60000	0.0606	0.0440	0.0315	0.0166	0.0708	0.0559	0.0362	0.0254
100000	0.1033	0.0719	0.0478	0.0276	0.1944	0.1269	0.0825	0.0585
350000	0.3769	0.2321	0.1628	0.0834	0.4835	0.2962	0.1983	0.1302
400000	0.4341	0.2561	0.1647	0.0898	0.6532	0.3765	0.2337	0.1661

Tabela 5.3: Tempo de processamento(dado em segundos) obtidos na execução do programa em até 8 processadores para o Quicksort

Nos gráficos das figuras 5.7, 5.8 e 5.9 apresentamos a plotagem do Speedup relativo às tabelas 5.2 e 5.4. Através dos gráficos pode-se verificar que para a linguagem

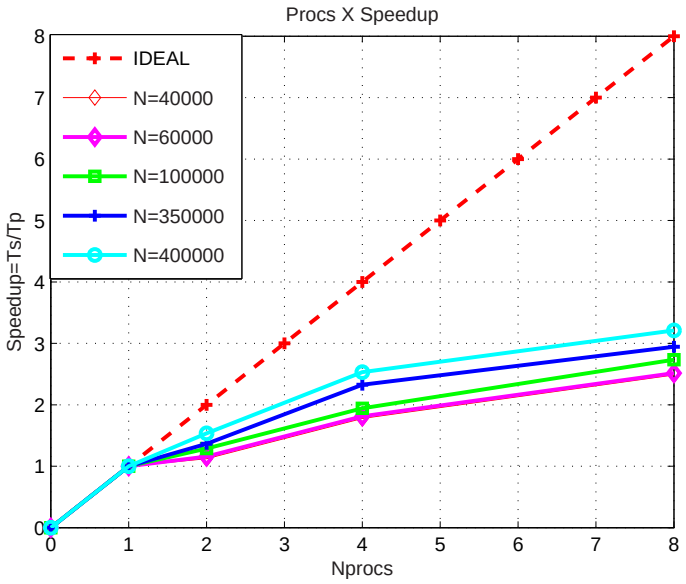

Figura 5.7: Gráfico do Speedup obtido pelo QuickSort Hipercubo- SCILAB

	Gráfico de Speedup - C				Gráfico de Speedup -FORTRAN			
	1 Proc	2 Proc	4 Proc	8 Proc	1 Proc	2 Proc	4 Proc	8 Proc
40000	1.00	1.3362	1.7322	3.4123	1.00	1.3542	1.7542	2.4531
60000	1.00	1.3742	1.9232	3.6291	1.00	1.6932	1.9531	2.7840
100000	1.00	1.4362	2.1573	3.9321	1.00	1.5312	2.3562	3.3230
350000	1.00	1.6235	2.3148	4.5170	1.00	1.6321	2.4372	3.7131
400000	1.00	1.6945	2.6351	4.8321	1.00	1.7348	2.7942	3.9321

Tabela 5.4: Speedup obtido pelo QuickSort nas linguagens - C e FORTRAN

interpretada SCILAB e para a linguagem FORTRAN, o desempenho apresentado é muito semelhante. Por outro lado, verifica-se que o ganho obtido está muito abaixo na linha ideal. Explica-se esse comportamento pelo fato da implementação do método Quicksort apresentar muita comunicação entre os processadores, prejudicando seu desempenho. Existem relatos na literatura sobre o algoritmo quicksort paralelo apresentar speedup muito abaixo do ideal [26].

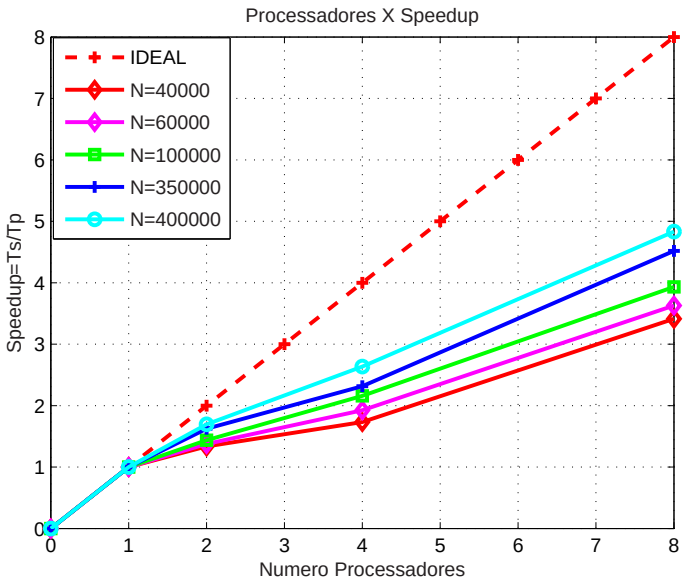

Figura 5.8: Gráfico do Speedup obtido pelo QuickSort Hipercubo- C

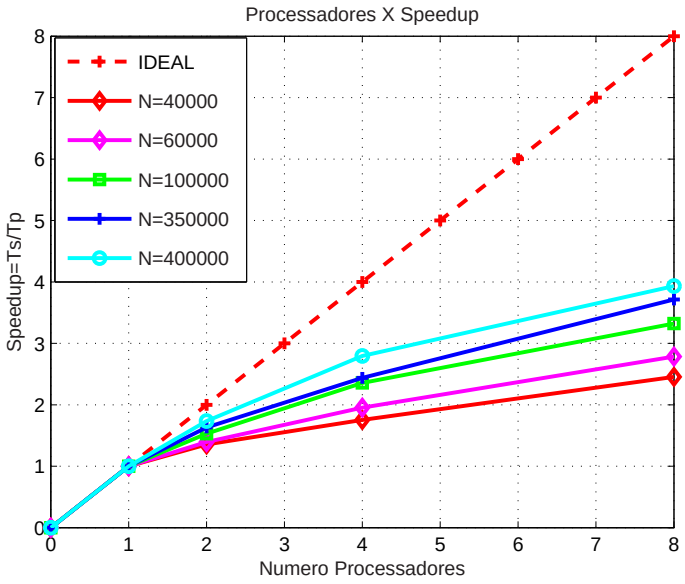

Figura 5.9: Gráfico do Speedup obtido pelo QuickSort Hipercubo- FORTRAN

O gráfico da figura 5.8 mostra um maior desempenho se comparado com os gráficos das tabelas 5.7 e 5.9 . Ou seja, para o número de processadores testado (oito) e as dimensões testadas (até 400000), os testes computacionais indicam um melhor desempenho da linguagem C em relação à escabilidade. Entretanto, esta conclusão parcial precisa ser testada com números de processadores e dimensão de vetores ainda mais elevados, para verificar se esta vantagem se mantém.

Como conclusão geral, pode-se dizer que embora o problema de ordenação de números seja de difícil paralelização, para todas as linguagens testadas mais uma vez observa-se o mesmo comportamento qualitativo da linguagem interpretada e compilada, confirmando a possibilidade de utilizar SCILAB para prototipagem rápida.

Capítulo 6

Conclusões e trabalhos futuros

O objetivo desse trabalho foi implementar, em ambiente paralelo e de forma integrada, o software SCILAB para computação científica juntamente com os softwares LAPACK e BLAS, e verificar a possibilidade de utilizar a plataforma integrada para prototipagem rápida de problemas de computação científica. Foram escolhidos alguns problemas típicos de computação científica: multiplicação de matrizes, solução de sistemas lineares e o problema de ordenação. Para estes problemas, constatou-se que o comportamento do conjunto SCILAB, LAPACK e BLAS (pacotes do NETLIB) em ambiente de computação paralela é qualitativamente semelhante ao comportamento de linguagens convencionais como C e FORTRAN. Por outro lado, por ser uma linguagem interpretada, o tempo computacional utilizado pelo SCILAB fica um pouco maior daquele obtido por uma linguagem compilada como C ou FORTRAN. Conclui-se que é apropriado utilizar o conjunto SCILAB, LAPACK e BLAS para prototipagem rápida de algoritmos, verificar a consistência da lógica delas e ter uma idéia qualitativa dos tipos de solução e ganho relativo de paralelização. Pode ser afirmado também que, em toda aplicação na qual o tempo total de computação é importante, essa fase de prototipagem deve ser seguida por outra linguagem de programação em alguma linguagem compilada.

Outra conclusão que merece destaque é que o conjunto de bibliotecas utilizado

SCILAB, LAPACK e BLAS, além de ser livre e de código aberto, mostrou-se fácil de usar, abrangente em termos de bibliotecas disponíveis e numericamente robusto. Este conjunto oferece uma alternativa viável e gratuita a softwares comerciais bastante difundidos, tanto no meio acadêmico quanto industrial. Além disso, a linguagem interpretada Scilab permite o uso de programação utilizando funções intrínsecas de álgebra linear permitindo assim uma programação rápida e consistente para diversos problemas.

Como proposta de trabalhos futuros, os seguintes temas se destacaram durante o trabalho realizado nesta dissertação:

1. Integrar SCILAB e a interface denominada Message Passing Interface *(MPI)*, pois verificou-se que a versão atual de SCILAB, que se integra apenas com a interface Parallel Virtual Machine *(PVM)*, possui diversas limitações em termos de primitivos de comunicação. Além disso, a interface PVM deixou de ser desenvolvido nos últimos anos, e talvez seja descontinuada. Com a integração SCILAB e MPI, espera-se também uma melhoria no desempenho do conjunto.

2. Estender a implementação integrada de SCILAB, LAPACK e BLAS, realizada nessa dissertação, a um ambiente de Grid.

3. Estudar a viabilidade de utilizar o conjunto SCILAB, LAPACK e BLAS em outras aplicações do tipo " embaraçosamente paralelas " nas quais a facilidade de programar as tarefas concorrentes é mais importante do que o desempenho temporal.

Apêndice A

Manual de Instalação das Bibliotecas

Apresentamos neste apêndice a configuração e a instalação dos "*software*" necessários para o desenvolvimento deste trabalho.

A.1 Plataforma de Testes

A plataforma de testes utilizada é a máquina Saturno, mostrada na figura A.1, que se encontra no Núcleo de Atendimento em Computação de Alto Desempenho *(NACAD)*. A figura A.1 mostra a máquina Saturno que possui as seguintes especificações:

- Cluster SGI Altix 350
- 14 processadores Intel Itanium II (palavra de 64 bits) de 1.5 Ghz.
- Pico Teórico de Performance de 6 GFlop/s por CPU
- Memória: 28 Gbytes RAM (compartilhada - NUMA)
- Armazenamento em disco: 360 Gbytes
- Sistema operacional: RedHat Enterprise Linux + SGI ProPack

Figura A.1: Máquina Saturno do NACAD.

- Compiladores Intel e GNU (Fortran-90 e C/C++) com suporte OpenMP e MPI
- Scilab 4.0

A.2 PVM*(Parallel Virtual Machine)*

Primeiramente, deve-se baixar o pacote de instalação em [11] e seguir os passos:

- Conectar-se na máquina como administrador do sistema *(root)*;
- Em seguida, escolher um diretório para instalação: *usr/local*
- Descompactar o pacote através do comando "$tar -zxvf < nomearquivo.tar.gz >$ ";
- Entrar no diretório criado depois da descompactação e em seguida por meio da linha de comando digitar: $export\ PVM\_ROOT = PWD$;
- Após o passo anterior, ainda na linha de comando digitar o comando "*make*";
- Em seguida, ainda como *root*, procurar no diretório */etc*, o arquivo $profile$ e editar o arquivo adicionando a linha:

1. $export\ PVM\_ROOT = /usr/local/pvm/pvm3$

A.3 SCILAB

Para instalar o Scilab, é necessário que baixe o "software"no site [1]. Em seguida deve-se verificar todas as dependências necessárias para a instalação. As dependências necessárias são os pacotes: java, TCL/TK e PVM.

O Visual Tcl é o ambiente de desenvolvimento utilizado para criação de aplicações gráficas na linguagem Tcl/Tk. O TK é um conjunto de ferramentas gráficas (Tool Kit) do Tcl. Com o Tk é possível construir rapidamente aplicativos gráficos para UNIX e outros sistemas operacionais. O TCL/TK existente no Linux deve ser superior à versão 8.4. Se no sistema Linux em uso não houver versão superior é necessário instalá-lo. Para instalar o TCL/TK faça o *download* no site *http* : $//www.tcl.tk/software/tcltk$ e siga as instruções abaixo:

1. Conectar-se na máquina como administrador do sistema *(root)*;
2. Descompactar os arquivos usando "$tar\ -zxvf < nomearquivo.tar.gz >$ ";
3. Entrar no diretório criado depois da descompactação e em seguida entrar no diretório "*unix*" digitando o seguinte comando:
   (a) cd unix;
4. Executar na linha de comando de dentro do diretório unix o comando "$./configure$";
5. Novamente na linha de comando executar:
   (a) *make*
   (b) *make install*

Depois da instalação do software TCL/TK deve-se identificar a localização dos diretórios java, Tcl, Tk e PVM. Isso é feito da seguinte forma:

1. Procurar a localização dos diretórios << *lib* >> e << *include* >> do pacote TCL através do seguinte procedimento:

   (a) Executar o comando *whereis tcl*. Este comando retorna a localização do diretório *tcl*.

   (b) Entrar no diretório *tcl* encontrado no passo anterior.

   (c) Verificar a existência dos diretórios << *lib* >> e << *include* >> executando o comando "*ls*".

   (d) Copiar a localização dos diretórios << *lib* >> e << *include* >>

2. Repetir todos os passos do ítem anterior substituindo o termo TCL destacado para *TK*.

3. Repetir todos os passos do ítem 1 substituindo o termo TCL destacado para *java*.

4. Repetir todos os passos do ítem 1 substituindo o termo TCL destacado para *PVM*.

   - Se o diretório PVM não existir, o pacote *PVM* deve ser instalado conforme descrito no item A.2 e em seguida deve-se encontrar e copiar a localização dos diretórios << *lib* >> e << *include* >> do PVM.
   - Em seguida, entrar no diretório << */lib* >> e verificar o nome do diretório << *Linuxversao* >>.

Após todos os passos anteriores, deve-se realizar a instalação do pacote *SCILAB*. Para isso, realiza-se as seguintes instruções:

1. Entrar no site do Scilab (*http* : *//www.scilab.org/download/index_download.php ?page = old_releases*) e baixar a versão 4.0. Deve-se ter uma atenção especial quanto à plataforma em uso e ao tipo do arquivo, podendo ser *source, binário, rpm*. Utilizamos neste trabalho a plataforma *Linux 64 bits e o tipo do arquivo source*;

2. Após o passo anterior, conectar-se na máquina como administrador do sistema *root*.

3. Escolher o diretório $/usr/local$ podendo ser outro do interesse do administrador;

4. Copiar o arquivo Scilab para o diretório do ítem acima indicado;

5. Descompactar o arquivo usando o comando "$tar -zxvf < nomearquivo.tar.gz >$ ";

6. Após a descompactação do um arquivo citado no ítem anterior, é criado um diretório chamado "$scilab-4.0$" sendo $<4.0>$ a versão do Scilab correspondente. Entrar no diretório por meio do comando "$cd\ scilab-4.0$";

7. Dentro do diretório do Scilab, executar na linha de comando :./$configure$ $--prefix = /usr/local/scilab-4.0$ $--with-java$ $--with-tcl-library = /usr/local/scilab-4.0/lib$ $--with-tcl-include = /usr/local/scilab-4.0/include$ $--with-tk-library = /usr/local/scilab-4.0/lib$ $--with-tk-include = /usr/local/scilab-4.0/include$ $--with-pvm-library = /usr/local/scilab-4.0/pvm3/lib/LINUX64$ $--with-pvm-include = /usr/local/scilab-4.0/pvm3/include/$

8. Novamente na linha de comando executar:

   (a) *make*

   (b) *make install*

Depois da instalação das dependências do Scilab, é necessário executar na linha de comando os seguintes passos:

- export LD_LIBRARY_PATH= /usr/local/scilab-4.0/lib
- export LD_LIBRARY_PATH= $LD_LIBRARY_PATH: /usr/local/tk/lib: /usr/local/tcl/lib

- export DISPLAY=146.164.31.42:0.0

Para que o Scilab funcione corretamente é necessário a criação de links simbólicos no diretório */usr/local/bin* onde o responsável (administrador do sistema) deverá executar no diretório citado anteriormente, os comandos abaixo:

- ln -s /usr/local/pvm/pvm3/lib/pvm
- ln -s /usr/local/pvm/pvm3/lib/pvmd
- ln -s /usr/local/scilab/bin/scilab
- ln -s /usr/local/scilab/bin/scilex

O arquivo de "*profile*" deve ser editado. Este arquivo encontra-se no diretório */etc*. Para isso:

1. Abrir o arquivo através do comando:*vi profile*
2. Adicionar as seguintes linhas:
   (a) export PVMROOT=/usr/local/pvm/pvm3
   (b) export LD_LIBRARY_PATH=/usr/local/scilab-4.0/lib
3. Fechar o arquivo e em seguida atualizá-lo através do comando *sourceprofile*

Depois de realizar os passos acima citados, o Scilab pode ser utilizado através do comando: $scilab - nw$

A.4 Script PBS

O PBS (Portable Batch System) é um sistema de filas que permite o melhor gerenciamento dos recursos disponíveis nos servidores do NACAD.

A.4 Script PBS

Para a execução de um programa através de um sistema de fila,o uso de scripts PBS faz-se necessário para garantir que as tarefas aloquem os processadores.

Os scripts PBS utilizado pelas máquinas do NACAD, seguem um padrão conforme especificado abaixo:

- As filas estão divididas em dois recursos como: número de CPUs e tempo de processamento;
- As filas que consomem menos recursos (CPU e tempo de processamento) terão prioridade sobre as demais filas;
- Linhas iniciadas com $\sharp PBS$ são diretivas de configuração do PBS;
- Linhas iniciadas com $\sharp +$ *espaço* são considerados comentários;
- Caso não sejam especificados os limites dos recursos (CPU e tempo de processamento) o PBS colocará a tarefa em execução na menor fila;
- Nos casos em que o tempo alocado para o programa for insuficiente para sua execução, o mesmo será abortado após o término do tempo estipulado pela fila;
- Observar nos programas paralelos que os parâmetro do PBS ($\sharp PBS - lncpus = y$ ou $\sharp PBS - lnodes = x$ ou $\sharp PBS - lnodes = x : ppn = y$) deve ser o mesmo solicitado pelo programa.

A seguir apresentamos o script do PBS para execução de tarefas do Scilab em ambiente paralelo criado na máquina Saturno.

```
#PBS -S /bin/csh

#PBS -N Scilab
#PBS -m be
```

```
#PBS -k oe
#PBS -m ae
#PBS -M <seu e_mail>
#PBS -l ncpus=4
#PBS -l walltime=1:00:00
#PBS -j oe
#PBS -N job  #arquivo de saída

date
cd $PBS_O_WORKDIR
/home/users/junia/bin/scilab -nw << EOF
exec master.sci;
exit
EOF
```

A.5 Instalação de Bibliotecas

Os pacotes de bibliotecas BLAS, LAPACK, ScaLAPACK e BLACS (PVM-BLACS) podem ser encontrados respectivamente nos endereços abaixo:

1. *www.netlib.org/blas*
2. *www.netlib.org/lapack*
3. *www.netlib.org/scalapack*
4. *www.netlib.org/blacs*

Daqui em diante, inicializaremos o processo de instalação destas bibliotecas.

A.6 BLAS*(Basic Linear Algebra Subprograms)*

Para compilar o pacote BLAS deve-se :

1. Baixar a biblioteca no site [13];
2. Em seguida descompactar a biblioteca através do comando "*tar zxvf < blaz.tgz >* ";
3. Entrar no diretório criado após a descompactação digitando o seguinte comando: "*cd < nome_do_arquivo_criado >* "
4. Em seguida digitar na linha de comando: *f77 -O -c *.f*. Após este comando serão criados códigos objetos "*arquivo.o*" a partir dos programas fontes de cada função relativa ao pacote. O uso do comando *f77* é necessário, por que todas as funções existentes destas bibliotecas foram implementadas em *Fortran*.
5. Após a criação de todos os *arquivos.o* novamente digitar na linha de comando: *ar cr blas_LINUX.a *.o*. Esse comando *ar cr* criará um arquivo referente à todas as funções do passo anterior. Porém, essas funções no pacote *blas*, estarão em um único arquivo chamado, *blas_LINUX.a*.

A.7 LAPACK *(Linear Algebra Package)*

Para compilar o pacote LAPACK deve-se :

1. Baixar a biblioteca no site [15];
2. Em seguida descompactar a biblioteca através do comando "*tar zxvf < lapack.tgz >* ";
3. Entrar no diretório criado após a descompactação digitando o seguinte comando: "*cd < nome_do_arquivo_criado >* "

4. Em seguida digitar na linha de comando: *f77 -O -c *.f*. Após este comando serão criados códigos objetos "*arquivo.o*" a partir dos programas fontes de cada função relativa ao pacote. O uso do comando *f77* é necessário, por que todas as funções existentes destas bibliotecas foram implementadas em *Fortran*.

5. Após a criação de todos os *arquivos.o* novamente digitar na linha de comando: *ar cr lapack_LINUX.a *.o*. Esse comando *ar cr* criará um arquivo referente à todas as funções do passo anterior. Porém, essas funções no pacote *lapack*, estarão em um único arquivo chamado, *lapack_LINUX.a*.

A.8 BLACS (PVMBLACS)

A biblioteca *PVMBLACS* é um pacote paralelo dentro da estrutura BLACS. Para instalar o PVMBLACS deve-se seguir os seguintes passos:

1. Baixar a biblioteca no site [16] e procurar o pacote *PVMBLACS*;

2. Em seguida, descompactar a biblioteca através do comando "*tar zxvf < pvmblacs.tgz >*";

3. Entrar no diretório criado após a descompactação, digitando o seguinte comando: "*cd < nome_do_arquivo_criado >*";

4. Visualizar os arquivos pertecentes aos pacotes através do comando "*ls*";

5. Entrar no diretório "*BMAKES*" através do seguinte comando: "*cd < BMAKES >*";

6. Escolher dentro do diretório BMAKES o arquivo compatível para a arquitetura de máquina que está sendo instalado. O arquivo encontrado deverá ser renomeado através do seguinte comando: "*cp < nome_do_arquivo_compativel > Bmake.inc*";

7. Em seguida, devemos editar o arquivo *BMAKE.inc* através dos seguintes passos:

   (a) Executar o comando *whereis lapack*. Este comando retorna a localização do diretório;

   (b) Executar o comando *whereis blacs*. Este comando retorna a localização do diretório;

   (c) Executar o comando *whereis pvm*. Este comando retorna a localização do diretório;

   (d) Abrir o arquivo através do comando:*vi BMAKE.inc*, podendo ser outro editor;

   (e) Acrescentar, no arquivo aberto, as localizações dos diretórios anteriormente encontrados.

8. Após a execução dos passos anteriores digitar na linha de comando:"*make pvm*"

A.9 ScaLAPACK

Para compilar a biblioteca ScaLAPACK deve-se :

1. Baixar a biblioteca no site [17];
2. Descompatar a biblioteca utilizando o comando: "$tar-zxvf scalapack.tar.gz$".
3. Entrar no diretório criado após a descompactação, digitando o seguinte comando: "$cd < nome\_do\_arquivo\_criado >$ "
4. Verificar a existência do arquivo SLmake.inc através do comando "$lsSLmake.inc$";
5. Procurar pelo arquivo SLmake.inc e abrir com algum editor de sua preferência.
6. Editar o arquivo *SLmake.inc* indicando a localização dos diretórios "PVM", "$Blas$", "$Pblas$", "$Lapack$", "$Blacs$". A localização desses diretórios podem ser encontrados através do comando "$whereis < nome\_do\_diretorio >$ ".

7. Após a realização dos passos anteriores, deve-se executar o comando "*make*".

A.10 BENCHMARKS

Um "*benchmark*" é um teste ou conjunto de testes projetado para comparar desempenhos de diferentes sistemas de computação ou diferentes algoritmos para um conjunto de problemas padrão.

Nesse caso, é realizado uma breve análise do tempo encontrado em diversos problemas a fim de encontrar o tempo computacional que a máquina Saturno obteve durante a execução e comparar com a tabela encontrada em [27] no site do Scilab.

Informamos que esta seção tem a finalidade somente de comparar ambas as as plataformas utilizando a versão 4 do Scilab.

A tabela abaixo descreve a performance a máquina Saturno encontrada no *NACAD*.

```
-------------------------------------------------------------------

I. Matrix calculation
Creation, transp., deformation of a 1500x1500 matrix (sec): 0.9986042
800x800 normal distributed random matrix ^1000 (sec):       0.5510886
Sorting of 2,000,000 random values (sec):                   1.214144
700x700 cross-product matrix (b = a' * a) (sec):            6.0313286
Linear regression over a 600x600 matrix (c = a \ b') (sec): 1.0368243
Trimmed geom. mean (2 extremes eliminated):                 1.0792521
II. Matrix functions
FFT over 800,000 random values (sec):                       1.0662214
Eigenvalues of a 320x320 random matrix (sec):               1.5233018
Determinant of a 650x650 random matrix (sec):               1.1855082
Cholesky decomposition of a 900x900 matrix (sec):           1.2645056
Inverse of a 400x400 random matrix (sec):                   0.9637610
```

```
Trimmed geom. mean (2 extremes eliminated):                 1.1692056
III. Programmation
750,000 Fibonacci numbers calculation (vector calc) (sec):  0.3318010
Creation of a 2250x2250 Hilbert matrix (matrix calc) (sec): 0.9600131
Grand common divisors of 70,000 pairs (recursion) (sec):    0.6126157
Creation of a 220x220 Toeplitz matrix (loops) (sec):        1.0084813
Escoufier's method on a 37x37 matrix (mixed) (sec):         2.325808
Trimmed geom. mean (2 extremes eliminated):                 0.8401904

Total time for all 15 tests (sec):                          21.074007
Overall mean (sum of I, II and III trimmed means/50)_(sec): 1.0196799
--------------------------------------------------------------------
```

Abaixo descrevemos a plataforma de testes que está no site [27] como referência.

- Testes com:
- Intel Core 2 Duo 6600 de 2.4 Ghz
- 3 Go Ram
- Ambiente Windows com sistema operacional Vista 32 bits 6.0 (build 6000)
- incluindo a biblioteca ATLAS

```
--------------------------------------------------------------------

I. Matrix calculation
Creation, transp., deformation of a 1500x1500 matrix (sec): 0.3506902
800x800 normal distributed random matrix ^1000 (sec):       0.1244888
Sorting of 2,000,000 random values (sec):                   0.6976365
700x700 cross-product matrix (b = a' * a) (sec):            0.3191780
```

```
Linear regression over a 600x600 matrix (c = a \ b') (sec): 0.0879846
Trimmed geom. mean (2 extremes eliminated):                 0.2406370
II. Matrix functions
FFT over 800,000 random values (sec):                       0.3422662
Eigenvalues of a 320x320 random matrix (sec):               0.2271375
Determinant of a 650x650 random matrix (sec):               0.0932886
Cholesky decomposition of a 900x900 matrix (sec):           0.1285448
Inverse of a 400x400 random matrix (sec):                   0.0730085
Trimmed geom. mean (2 extremes eliminated):                 0.1396553
III. Programmation
750,000 Fibonacci numbers calculation (vector calc) (sec):  0.2052973
Creation of a 2250x2250 Hilbert matrix (matrix calc) (sec): 0.2555296
Grand common divisors of 70,000 pairs (recursion) (sec):    0.2489776
Creation of a 220x220 Toeplitz matrix (loops) (sec):        0.3566183
Escoufier's method on a 37x37 matrix (mixed) (sec):         0.4992032
Trimmed geom. mean (2 extremes eliminated):                 0.2830969

Total time for all 15 tests (sec):                          4.0098497
Overall mean (sum of I, II and III trimmed means/50)_(sec): 0.2118938
--------------------------------------------------------------------
```

Apêndice B

Interfaces dentro do SCILAB

Scilab é uma linguagem interpretada, mas propõe uma alternativa oferecendo diferentes mecanismos para integrar funções externas compiladas. Esta funcionalidade permite igualmente aumentar as funcionalidades de Scilab integrando novas funções. Uma nova função integrada, são chamados primitivos para distinguir das funções interpretadas. Scilab propõe quatro abordagens diferentes para criar uma conversão. Mas nesta dissertação daremos ênfase somente em duas abordagens que foram utilizadas para a integração de algumas funções do Lapack utilizando o Scilab.

B.1 Interface via addinter

No caso da realização da conversão com *addinter*, o utilizador deve escrever duas funções. A função externa, efetuando o tratamento desejado e, uma função que permite vincular a chamada e os dados procedentes de Scilab com a primitiva.

Para isso, a função que efetua a conversão é escrita pelo utilizador utilizando as funções da biblioteca concebida para esse efeito como por exemplo: **CheckRhs, CheckLhs, GetRhsVar, CreateVar**. A função que efetua a conversão controla inicialmente o número de argumentos das entradas e saídas verificando se estão corretas, cujo papel são das funções **CheckLhs e CheckRhs**.

Conseqüentemente, a função da interface verifica o tipo e a dimensão dos parâmetros. A função **GetRhsVar** desempenha o papel necessário para recuperar um ponteiro para os dados e transmitir à função convertida. A função **CreateVar** reserva um espaço para armazenar o resultado antes de chamar a função. Por último o ficheiro da interface faz uma chamada à função primitiva.

Para realizar esta conversão devemos então compilar dois programas: a função que efetua a interface e a nova primitiva, e, utilizar o utilitário addinter que vai gerar um "script"Scilab a fim de ativar a informação da primitiva.

B.2 Interface via link

Este método permite a conversão de uma nova primitiva sem ter que programar a função que efetua a conversão. O procedimento é simplificado, mas a utilização da informação da primitiva em Scilab é menor. O princípio é o seguinte:

1. Compila a primitiva em .o
2. Com o auxílio do comando **link** carrega-se dinamicamente a primitiva em Scilab.

```
n=link(SCI+'/routines/blas/dgemm.o','dgemm')


linking files /routines/blas/dgemm.o
to create a share executable.
Linking dgemm (in fact dgemm_)
Link done
n=
0
```

3. O comando **call** ou **fort** permite chamar a primitiva assim como linká-la. Ela é utilizada para passar os valores das variáveis necessárias para a primitiva,

seu tipo de C ou Fortran respectivamente, e sua posição na lista é chamada de primitiva. A sintaxe é usada da seguinte forma:

```
[parametros dos resultados] = fort('dgemm', descrição de entradas,
'out', descrição de saídas)
```

O único inconveniente deste método é que a chamada via **call** ou **fort** não corresponde à uma chamada padrão do Scilab é possui uma sintaxe mais complexa.

Referências

[1] SCILAB(2006). http://www.scilab.org.

[2] DESPREZ, F., FLEURY, E., GRIGORI, L., "Scilab//: User interactive applications and high performances". *Multiconference on Systemics, Cibernetics and Informatics* v. 1, August 1999.

[3] MORETTI, C. O., *Análise de Estruturas Utilizando Técnicas de Processamento Paralelo Distribuído*. Master's Thesis, August 1997.

[4] FREEMAN, L., *Parallel Numerical Algorithms*. Prentice Hall International Series in Computer Science, 1992.

[5] CUNHA, M. T. F., DE F. TELLES, J. C., *Uma metodologia Portável para Paralelização de Programas de Elementos de Contorno*. Master's Thesis, August 2004.

[6] GRAMA, A., A.GUPTA, KARYPIS, G., KUMAR, V., *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[7] CAMPBELL, S., CHANCELIER, J. P., NIKOUKHAH, R., *Modeling and Simulation in Scilab/Scicos*. Springer Science, 2006.

[8] DESPREZ, F., FLEURY, E., GOMEZ, C., STEER, S., UBÉDA, S., "Bringing metacomputing to scilab". *Proceedings of the IEEE - International Symposium on Computer Aided Control System Design* v. 1, pp. 398–403, August 1999.

[9] CARON, E., *Calcul numérique sur données de grande taille.* Master's Thesis, Université de Picardie - Jules Verne, décembre 2000.

[10] ACKNOWLEDGMENTS(2007), S. http://www.scilab.org/index.php?page =acknowledgment.htm.

[11] PVM(2006). http://www.netlib.org/pvm3.

[12] DO SCILAB(2006), M. http://www.scilab.org/product/man-eng/index.html.

[13] BLAS(2006). http://www.netlib.org/blas.

[14] BLACKFORD, L. S., DEMMEL, J., ET AL, J. D., "An updated set of basic linear algebra subprogramns(blas)". *ACM Transactions on Mathematical Software* v. 28, pp. 135–151, June 2002.

[15] LAPACK(2006). http://www.netlib.org/lapack.

[16] BLACS(2006). http://www.netlib.org/blacs.

[17] SCALAPACK(2006). http://www.netlib.org/scalapack.

[18] ORTEGA, J. M., *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, 1988.

[19] HEATH, M. T., *Scientific Computing An Introductory Survey.* Second ed., SIAM, 1997.

[20] STOER, J., BULIRSCH, R., *Introduction to Numerical Analysis.* Springer-Verlag, 1980.

[21] PACHECO, P. S., *Parallel Programming with MPI.* Morgan Kaufmann Publishers,Inc.Editorial and Sales Office, 1997.

[22] KORFGEN, B., I.GUTHEIL, "Parallel linear algebra methods". *Computational Nanoscience : Do It Yourself!* v. 31, n. 1, pp. 507–522, February 2006.

[23] PRADO, J. A. S., *Análise Experimental do QUICKSORT Probabilístico com Gerador de Números Pseudo-Aleatórios Penta-Independente.* Master's Thesis, Universidade Federal do Paraná - Setor de Ciências Exatas - Programa de Pós-Graduação em Informática, Dezembro 2005.

[24] ZIVIANI, N., *Projeto de Algoritmos com Implementação em C e Pascal.* Pioneira, 2001.

[25] JAJA, J., "A perspective on quicksort". *Computing in Science e Engineering* v. 2, pp. 43–49, February 2000.

[26] ALBUQUERQUE, J., JUNIOR, C. J. N. C. Avaliando o hyperquicksort utilizando uma now, Fevereiro 1999.

[27] BENCH(2007). http://www.scilab.org/product/index_product.php?page=benchmarks_4.1.