UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

DELCINO PICININ JÚNIOR

Paralelizações de Métodos Numéricos em Clusters Empregando as Bibliotecas MPICH, DECK e Pthread

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Tiarajú Asmuz Diverio
Orientador

Porto Alegre, fevereiro de 2003

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Júnior, Delcino Picinin

Paralelizações de Métodos Numéricos em Clusters Empregando as Bibliotecas MPICH, DECK e Pthread / Delcino Picinin Júnior. – Porto Alegre: Programa de Pós-Graduação em Computação, 2003.

91 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2003. Orientador: Tiarajú Asmuz Diverio.

1. Cluster. 2. MPI. 3. DECK. 4. GMRES. 5. GC. 6. Paralelização. I. Diverio, Tiarajú Asmuz. II. Título.

"A ciência avança através de respostas provisórias, conjeturais, em direção a uma série cada vez mais sutil de perguntas que penetram cada vez mais fundo na essência dos fenômenos naturais."

— LOUIS PASTEUR

AGRADECIMENTOS

Em primeiro lugar eu gostaria de agradecer a Deus que é o responsável por todas as coisas boas que acontecem na vida.

No que se refere a pessoas, eu tenho muitas pessoas a agradecer, mas entre elas há duas pessoas muito especiais, que são meus pais. Durante toda a minha vida eu recebi de meus pais muito amor, carinho, estímulo, ajuda, entre outras milhares de coisas, sem as quais eu jamais seria quem eu sou.

Outra pessoa responsável por este trabalho e a quem eu devo agradecimentos é o meu orientador Prof. Dr. Tiarajú A. Diverio, que foi quem me aceitou no mestrado como seu orientando e seguiu do meu lado nestes dois anos de trabalho.

Durante o mestrado eu conheci várias pessoas, mas há duas em especial a quem eu devo um grande agradecimento, que são meus colegas de trabalho Rogerio e Cadinho. A eles eu agradeço pelo grande apoio e ajuda sempre que precisei, não só como colegas mas também como amigos. Eu sempre vou me lembrar com saudades dos finais de semana que ficávamos na UFRGS depurando código, discutindo métodos numéricos e assuntos relacionados a paralelização e tomando café.

No que se refere a depurar código e conversar tem um outro colega de trabalho de quem também vou sentir saudades, que é o Martinotto. Ao Martinotto eu agradeço pelos momentos de trabalho e descontração nos congressos. Agradeço também à Mônica pela disposição em trocar idéias e me auxiliar no uso do LaTeX.

Além dos já citados, eu fiz outros novos amigos a quem devo agradecer pelos momentos passados juntos que são: Alex Ferreira, Diana Adamati, Luciana Foss, Luciana Schreder, Guilherme, Aline, Leonardo e Rafael.

Há também outras pessoas a quem eu gostaria de agradecer que são o pessoal do *cluster*, inicialmente ao Rafael e Pilla e em um segundo momento ao trio Sanger, Diego e Clarissa pela ajuda com o uso das máquinas entre outras coisas.

Eu também gostaria de agradecer ao pessoal do doutorado pelo convívio nestes anos, entre eles o Campani, Juliana, Marcia, Patricia, Leandro ...

No que se refere a descontração, eu agradeço a minhas irmãs que me deram sobrinhos com quem eu brinco sempre que tenho tempo.

Agradeço também pelo apoio dos meus demais amigos como por exemplo a Gi, que sempre me deu muito estímulo e sempre torceu por mim.

Agradeço aos funcionários da UFRGS, em especial a seu Astro que sempre me atendia quando eu vinha trabalhar nos finais de semana na universidade.

Para encerrar agradeço ao CNPq que me auxiliou através de uma bolsa de mestrado, sem a qual o desenvolvimento deste trabalho seria bem mais difícil.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Aplication Program Interface
BiCGStab	BiConjugate Gradient Stabilized
BLAS	Basic Linear Algebra Subprograms
CGS	Conjugate Gradient Squared
CND	Coordinate Nested Dissection
CPU	Central Processing Unit
CR	Conjugate Residual
DECK	Distributed Execution and Communication Kernel
DSM	Distributed Shared Memory
EDPs	Equações Diferenciais Parciais
GC	Gradiente Conjugado
GMCPAD	Grupo de Matemática da Computação e Processamento de Alto Desempenho
GMRES	Método do Resíduo Mínimo Generalizado
IC	Incomplete Cholesky
ILU	Incomplete LU
MPI	Message Passing Interface
ORB	Ortogonal Recursive Bisection
PETSC	Portable, Extensible Toolkit for Scientific Computation
POSIX	Portable Operating System Interface
QMR	Quasi-Minimum Residual
RCB	Recursive Cordenate Bisection
SCI	Scalable Coherent Interface
SELAs	Sistemas de Equações Lineares Algébricas
SOR	Successive Overrelaxation
SSOR	Symmetric Successive Overrelaxation
SPMD	Single Program Multiple Data

TFQMR Transpose-Free Quasi-Minimum Residual Method

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho tem como objetivo de desenvolver e empregar técnicas e estruturas de dados agrupadas visando paralelizar os métodos iterativos do subespaço de Krylov, fazendo-se uso de diversas ferramentas e abordagens. A partir dos resultados é feita uma análise comparativa de desempenho destas ferramentas e abordagens.

As paralelizações aqui desenvolvidas foram projetadas para serem executadas em uma arquitetura formada por um agregado de máquinas indepentes e multiprocessadas (*cluster*), ou seja, são considerados o paralelismo inter-nodos e intra-nodos.

Para auxiliar a programação paralela em *clusters* foram, e estão sendo, desenvolvidas diferentes ferramentas (bibliotecas) que visam a exploração dos dois níveis de paralelismo existentes neste tipo de arquitetura.

Neste trabalho emprega-se diferentes bibliotecas de troca de mensagens e de criação de *threads* para a exploração do paralelismo inter-nodos e intra-nodos. As bibliotecas adotadas são o DECK o MPICH e a Pthread.

Um dos itens a serem analisados neste trabalho é a comparação do desempenho obtido com essas bibliotecas. Outro item é a análise da influência no desempenho quando utilizadas múltiplas *threads* no paralelismo intra-nodos em *clusters* multiprocessados.

Os métodos paralelizados nesse trabalho são o Gradiente Conjugado (GC) e o Resíduo Mínimo Generalizado (GMRES), que podem ser adotados, respectivamente, para solução de sistemas de equações lineares simétricos positivos e definidos e não simétricas. Tais sistemas surgem da discretização, por exemplo, dos modelos da hidrodinâmica e do transporte de massa que estão sendo desenvolvidos no GMCPAD. A utilização desses métodos é justificada pelo fato de serem métodos iterativos, o que os torna adequados à solução de sistemas de equações esparsas e de grande porte.

Na solução desses sistemas através desses métodos iterativos paralelizados faz-se necessário o particionamento do domínio do problema, o qual deve ser feito visando um bom balanceamento de carga e a minimização das fronteiras entre os sub-domínios. A estrutura de dados desenvolvida para os métodos paralelizados nesse trabalho permite que eles sejam adotados para solução de sistemas de equações gerados a partir de qualquer tipo de particionamento, pois o formato de armazenamento de dados adotado supre qualquer tipo de dependência de dados.

Além disso, nesse trabalho são adotadas duas estratégias de ordenação para as comunicações, estratégias essas que podem ser importantes quando se considera a portabilidade das paralelizações para máquinas interligada por redes de interconexão com *buffer* de tamanho insuficiente para evitar a ocorrência de *deadlock*.

Os resultados obtidos nessa dissertação contribuem nos trabalhos do GMCPAD, pois as paralelizações são adotadas em aplicações que estão sendo desenvolvidas no grupo.

Palavras-chave: Cluster, MPI, DECK, GMRES, GC, Paralelização.

ABSTRACT

Parallelization of Numerical Methods in Clusters of PCs using libraries MPICH, DECK and Pthread

The objective of this work is the development and use of techniques and data structures to parallelize Krylov subspace iterative methods, making use of several tools and approaches. From the obtained results, a comparative analysis of the performance of these tools and approaches is made.

The parallelizations developed here were designed to be run in an architecture composed by a cluster of independent and multiprocessed machines, using intra-node and inter-node parallelism.

To make parallel programming in clusters easier, there are several tools (libraries) whose objective is to explore both levels os parallelism existent in this kind of architecture.

In this work different message passing and threads libraries are used to explore inter-node and intra-node parallelism. The libraries used are DECK, MPICH and Pthread.

One item analysed in this work is the comparison of the performance obtained with these libraries. Furthermore, an analysis of the influence of the use of multiple threads in performance ofintra-node parallelism in multiprocessed clusters is presented.

The methods parallelized in this work are Conjugate Gradient (CG) and Generalized Minimum Residual (GMRES), which can be used, respectively, to the solution of linear equations systems SDP (symmetric definite positive) and not symmetrical. Such systems arise from the discretization, for example, of the hydrodynamics and mass transportation models being developed in GMCPAD. The use of these methods is justified by the fact that, being iterative, they are adequate to the solution of sparse equations systems.

In the solution of these systems through parallelized iterative methods, it is necessary to partition the problem domain, what is done looking for a good load balancing and the minimization of the frontiers between the subdomains. The data structure developed for the parallelized methods in this work allow them to be adopted to the solution of equations systems generated from any kind of partitioning, as the format used to store the data supports any type of data dependency.

Furthermore, in this work two ordering strategies for communication are adopted. These strategies can be important when it is considered the portability to architectures without enough buffer to avoid the occurrence of deadlocks.

The results obtained in this work contribute to the works of GMCPAD, as the parallelizations are adopted in applications being developed in the group.

Keywords: Clusters, MPI, DECK, GMRES, GC, Parallelization.

1 INTRODUÇÃO

O objetivo principal deste trabalho é desenvolver e empregar técnicas e estruturas de dados eficiêntes para paralelizações de dois métodos iterativos de solução do subespaço de Krylov que são o GC e o GMRES. Essas paralelizações foram desenvolvidas para serem executadas em uma arquitetura formada por agregados de máquinas independentes (*cluster*), fazendo uso de diferentes bibliotecas de troca de mensagens e de criação de *threads*.

Neste capítulo serão apresentadas as motivações, seguidas pelas justificativas, que levaram ao desenvolvimento deste trabalho, descrevendo-o e pontuando-o capítulo a capítulo. Além disso, faz-se uma descrição de alguns trabalhos relacionados.

1.1 Motivação

Através de modelos computacionais é possível simular complexos processos naturais, pois a análise dos resultados de um modelo e de suas propriedades, viabilizada pelo avanço da tecnologia dos computadores, permite entendimentos e predições de situações reais (CUMINATO; MENEGETT, 2000).

Tais modelos são, geralmente, constituídos de um sistema de equações diferenciais parciais (EDPs), que são definidas em um espaço contínuo e infinito de pontos. Para resolver numericamente essas EDPs, faz-se necessário transformar esse espaço em um espaço discreto e finito de pontos, sendo o processo chamado de discretização, o responsável por essa transformação.

O processo de discretização gera um sistema de equações que deve ser resolvido a cada passo de tempo. Existem, basicamente, duas classes de métodos que os resolvem: a dos métodos diretos e a dos métodos iterativos. Nesse trabalho, emprega-se os métodos iterativos, pois as matrizes dos coeficientes a serem resolvidas são esparsas e de grande porte e, o emprego de métodos diretos destruiria essa esparsidade, dificultando a otimização no armazenamento. Além disso, os métodos diretos oferecem dificuldades adicionais no processo de paralelização, dada sua intrínseca dependência de dados.

Dada a necessidade de grande quantidade de memória e de velocidade de processamento para a solução de sistemas de equações com essas características, que geralmente têm milhares ou milhões de incógnitas, o emprego de processamento paralelo é uma alternativa viável para sua solução, quando se considera necessário que a simulação computacional seja feita em um tempo suficientemente rápido para ser útil.

Nesses casos, uma boa alternativa é o emprego de ambientes de alto desempenho. Entre os ambientes de alto desempenho os *clusters* têm se mostrado uma opção acessível e eficiente, com uma boa relação custo/benefício para obter um alto desempenho

computacional.

Um *cluster* é uma arquitetura baseada no agrupamento de máquinas independentes, interconectadas por uma rede de comunicação rápida. Tal agrupamento pode ser formado por máquinas monoprocessadas, multiprocessadas ou ambas. No caso de serem formados por máquinas multiprocessadas, além do paralelismo inter-nodos, deve-se explorar o paralelismo intra-nodos.

Para possibilitar a programação paralela inter-nodos em *clusters*, foram desenvolvidas diversas ferramentas, entre as quais estão o MPICH (GROPP; LUSK, 2002) e o DECK (BARRETO, 2000) . No caso de *clusters* formados por máquinas multiprocessadas, um bom recurso para exploração do paralelismo intra-nodos é o uso de múltiplas *threads*, pois compartilham código, dados e recursos.

Nesse trabalho são feitas paralelizações de métodos do subespaço de Krylov (SAAD, 1996) fazendo uso do MPICH-1.2.4 com e sem múltiplas *threads*. Essas mesmas paralelizações são feitas com o uso do DECK-2.1.2, objetivando avaliar o desempenho desta versão como uma ferramenta para a exploração do paralelismo em *clusters* de PCs e, também, efetuar uma comparação dessa com o MPICH.

1.2 Justificativa

O GMCPAD (Grupo de Matemática da Computação e Processamento de Alto Desempenho) da UFRGS vem trabalhando no desenvolvimento de modelos paralelos para a hidrodinâmica e para o transporte de substâncias 2D e 3D (DORNELES et al., 2002) (RIZZI, 2001) desde o início de 1998.

Entre os métodos que estão sendo adotados para resolver os sistemas de equações gerados pelo modelos, já discretos, citados acima, estão o GC (Gradiente Conjugado) e o GMRES (Resíduo Mínimo Generalizado). O Gradiente Conjugado está sendo adotado para resolver alguns dos sistemas de equações gerados pelo modelo discretizado de hidrodinâmica. Já para resolver as equações geradas pelo modelo discretizado de transporte de massa um dos métodos é o GMRES. As paralelizações desenvolvidas nesse trabalho foram incorporadas a esse modelo computacional (Modelo Computacional Paralelo para Hidrodinâmica e para o Transporte de Substâncias Bidimensional e Tridimensional).

1.3 Trabalhos Relacionados

No que se refere aos trabalhos relacionados, existem várias bibliotecas de métodos numéricos que implementam os métodos paralelizados neste trabalho, e além dessas no GMCPAD também já foram desenvolvidos trabalhos de paralelizações destes métodos.

Considerando as paralelizações já desenvolvidas no GMCPAD, no que se refere ao Gradiente Conjugado, foram desenvolvidas versões fazendo uso do DECK, da Pthread e, também, do MPI, versões essas que podem ser encontradas, respectivamente, em (CANAL, 2000) e (PICININ JUNIOR et al., 2001). Já no que se refere ao GMRES, foram desenvolvidas apenas versões fazendo uso do MPI e da Pthread que podem ser encontradas em (MARTINOTTO, 2001). A principal distinção das paralelizações do Gradiente Conjugado e do GMRES desenvolvidas nesse trabalho, em relação às já desenvolvidas anteriormente no GMCPAD refere-se ao fato dessas poderem trabalhar com matrizes irregulares, que necessitam uma estrutura de dados muito mais sofisticada,

resultantes de divisões de domínios em subdomínios de forma arbitrária não retangular.

Além do fato das paralelizações aqui desenvolvidas poderem trabalhar com matrizes irregulares, elas se diferem das já desenvolvidas no GMCPAD também por poderem usar duas estratégias de ordenação de mensagens, usarem primitivas não bloqueantes para sobreposição de processamento com comunicação e, por explorarem o paralelismo fazendo uso de múltiplas *threads* com diferentes estratégias. Estas opções e características aumentam consideravelmente a eficiência e a flexibilidade dos métodos de solução paralelos.

Considerando as bibliotecas de métodos numéricos, nesta seção, apresenta-se uma breve descrição das melhores bibliotecas com suas principais características. As bibliotecas apresentadas nesta seção são: PETSC (SMITH et al., 2002), Aztec (TUMINARO; SHADID; HEROUX, 2002) e IML++ (DONGARRA et al., 2002).

As paralelizações desenvolvidas neste trabalho se diferem das existentes nas bibliotecas citadas pelo fato de permitirem a exploração do paralelismo intra-nodos através do emprego da biblioteca Pthreads, que embora torne o processo de paralelização mais complexo permitem um melhor ganho de desempenho.

Além do uso da Pthreads as paralelizações desenvolvidas neste trabalho possuem versões que fazem uso da biblioteca DECK, que conforme pode ser visto nas conclusões deste trabalho apresentou bons resultados.

Outro item que diferencia as paralelizações deste trabalho é a possibilidade de adoção de estratégias de ordenação de mensagens, as quais visam efetuar um controle de fluxo em caso de falta de *buffer* por parte da rede de interconexão e, assim evitando ocorrer *deadlock*.

1.3.1 PETSC

A PETSC *(Portable, Extensible Toolkit for Scientific Computation)* foi desenvolvida no Argonne National Laboratory por Satish Balay, William Gropp, Lois Curfman McInnes e Barry Smith (SMITH et al., 2002).

Essa biblioteca foi escrita nas linguagens C, C++ e Fortran, todas fazendo uso do MPI como biblioteca de troca de mensagens.

A PETSC possui implementados os seguintes métodos iterativos: GMRES, CG, CGS, BiCGStab, TFQMR e CR.

Com o objetivo de acelerar a convergência dos sistemas, esta biblioteca também possui precondicionadores: Identidade, Jacobi, Jacobi em Bloco, Gauss-Seidel em Bloco (somente seqüencial), SOR e SSOR, IC, ILU e Aditivos de Schwarz.

O PETSC possui diferentes estruturas de armazenamento que visam se adequar aos problemas e suas soluções e permitem fazer uma distinção das células que geram dependência de dados, conhecidas como células de sobreposição.

A versão atual do PETSC faz uso de múltiplas *threads* na execução de subrotinas diferentes que não possuem dependência e, além disso, paraleliza algumas operações em laço com a biblioteca de *threads* OpenMP.

As operações de álgebra linear no PETSC foram implementadas fazendo uso do BLAS *(Basic Linear Algebra Subprograms)*, que é uma biblioteca de álgebra linear desenvolvida em linguagem de baixo nível.

1.3.2 Aztec

A Aztec foi desenvolvida no Sandia National Laboratories por Scott A. Hutchinson, John N. Shadid e Ray S. Tuminaro (TUMINARO; SHADID; HEROUX, 2002).

Semelhantemente ao PETSC, esta também faz uso do MPI como biblioteca de troca de mensagens, no entanto, possui implementação apenas na linguagem C.

Os métodos iterativos implementados na Aztec são: CG, CGS, BiCGStab e GMRES.

Além dos métodos, esta biblioteca também possui os precondicionadores: Jacobi em blocos com ILU em sub-blocos e Aditivos de Schwarz.

O Aztec também faz uso de uma estratégia de armazenamento de células que possuem uma dependência de localidade (células de sobreposição).

Outra importante característica do Aztec é o fato de incorporar a biblioteca de particionamento Chaco (TUMINARO; SHADID; HEROUX, 2002).

A implementação do Aztec faz uso de rotinas das bibliotecas: BLAS, Lapack, Linpack e Y12m.

1.3.3 IML++

A IML++ foi desenvolvida no National Institute of Standards and Technology por Jack Dongarra, Andrew Lumsdaine, Roldan Pozo e Karin A. Remington (DONGARRA et al., 2002).

Esta biblioteca foi escrita apenas na linguagem C++ e também faz uso do MPI.

Os métodos implemetados nesta biblioteca são: Iteração de Richardson, Iteração de Chebyshev, CG, CGS, BiCG, BiCGSTAB, GMRES e QMR.

Esta biblioteca, semelhantemente às demais também faz uso de precondicionadores. Os precondicionadores suportados por ela são: diagonal, ILU e IC.

A IML++ faz uso da Sparselib++, que é uma biblioteca de classes de operações algébricas desenvolvida em C++.

A Sparselib++ possui diversos formatos de armazenamento e possui funções para gerenciamento de matrizes esparsas, fazendo uso da BLAS para algumas operações, como por exemplo, multiplicação de uma matriz por um vetor.

Além dos trabalhos relacionados que já foram citados acima, uma revisão na literatura técnica nacional mostra que pesquisas em Computação Cientifica Paralela, particularmente aquelas que se referem a paralelização de métodos iterativos, não são ainda muito difundidas no Brasil. Pesquisas sob este escopo são restritas, aparentemente, às Instituições: COPPE/UFRJ, USP, LNCC, INPE, entre outras, além, obviamente, da UFRGS.

1.4 Apresentação do Trabalho

A estrutura desse trabalho está organizada de forma a proporcionar ao leitor um entendimento gradativo do conteúdo apresentado. Essa estrutura está dividida em seis capítulos. Nesse capítulo inicial, foi apresentada uma introdução contendo: motivações, justificativas e trabalhos relacionados.

Devido ao fato do escopo principal desse trabalho ser processamento paralelo, o segundo capítulo inicialmente apresenta o ambiente de programação paralela utilizado no desenvolvimento do mesmo. A descrição desse ambiente está dividida em duas partes: a primeira se refere à parte física (*hardware*) e a segunda se refere à parte lógica (*software*).

Nas paralelizações apresentadas nesse trabalho, o processamento paralelo é adotado na solução de sistemas lineares, os quais são gerados a partir da discretização de uma aplicação real. Assim, a segunda parte do capítulo dois tem por objetivo contextualizar o leitor sobre aplicações, modelos computacionais e, também, sobre as principais características dos métodos paralelizados neste trabalho.

Na paralelização de uma aplicação, após a obtenção de um modelo discreto, é necessário efetuar um particionamento do mesmo. Sendo assim, o capítulo três se refere a particionamento de domínios, onde são mostrados os algoritmos existentes, os tipos de sub-domínios resultantes dos particionamentos e os tipos de matrizes geradas a partir desses sub-domínios.

O objetivo principal deste trabalho que são as paralelizações está no capítulo quatro. Nesse capítulo inicialmente são discutidos e analisados as principais características e aspectos que são relevantes para desenvolver a paralelização dos dois métodos, as quais são: o tipo de domínio em que os métodos podem ser adotados, o tipo de armazenamento de dados aplicado, as estratégias de comunicação aplicadas e os pré-condicionadores aplicados.

As avaliações dos resultados obtidos com as paralelizações através de testes e comparações são apresentadas no capítulo cinco.

Por fim, no capítulo seis são discutidas conclusões e contribuições do trabalho, e apresentadas algumas propostas de trabalhos futuros.

2 ESCOPO DO TRABALHO

As paralelizações desenvolvidas neste trabalho foram projetadas para serem executadas em agrupamentos de máquinas independentes, que devem ser interligadas por rápidas redes de interconexão. Esses agrupamentos são conhecidos como *clusters*, sendo utilizado em português também os termos "agregado"ou "agrupamento".

Para o desenvolvimento de aplicações paralelas em *clusters* existem algumas bibliotecas - como por exemplo: MPI, DECK e Pthreads - que têm a função de proporcionar primitivas para a exploração dos paralelismos inter-nodos e intra-nodos.

Na aplicação desenvolvida o processamento paralelo é adotado na solução de sistemas lineares, os quais são gerados a partir da discretização de EDPs que modelam fenômenos naturais. Os métodos paralelizados para a solução destes sistemas são os métodos do subespaço de Krylov.

Neste capítulo é apresentado o escopo do trabalho contendo a arquitetura da máquina, os *softwares* adotados e as principais características da aplicação paralelizada.

Para encerrar o capítulo são apresentadas considerações finais referentes aos itens do presente capítulo.

2.1 Hardware: *Cluster*

Um *cluster* é uma arquitetura baseada na união de um conjunto de máquinas independentes, interconectadas por uma rede de comunicação rápida, formando uma plataforma de alto desempenho para execução de aplicações paralelas (BUYYA, 1999).

São muitas as características que podem ser empregadas para classificar a arquitetura de um *cluster*, entre elas estão: a rede de interconexão das máquinas, a homogeneidade ou não das máquinas que o formam, a homogeneidade ou não do sistema operacional e número de processadores em cada máquina.

O nível de paralelismo a ser explorado em um *cluster* depende, em parte, da arquitetura existente, ou seja, em *clusters* formados por máquinas multiprocessadas, existe a possibilidade da exploração do paralelismo intra-nodos em conjunto com a exploração do paralelismo inter-nodos. Já em *clusters* formados por máquinas monoprocessadas, somente o paralelismo inter-nodos pode ser explorado.

Na exploração do paralelismo inter-nodos, a rede de interconexão tem uma grande influência no desempenho. A capacidade dessa rede é geralmente medida em termos de latência e de largura de banda, onde latência é o tempo para enviar o primeiro *bit* de um computador para o outro, incluindo o custo da criação do pacote a ser enviado na rede, enquanto largura de banda é o número de *bits* que pode ser transmitido pela rede de interconexão por unidade de tempo (CARVALHO, 2002).

2.1.1 Redes de Interconexão

Existem vários tipos de redes de interconexão que podem ser adotadas em um *cluster*. O *ranking* publicado semestralmente pela Universidade de Mannheim, disponível em http://www.top500.org/clusters apresenta os *clusters* de maior desempenho da atualidade, onde as redes de interconexão adotadas são:

- *Fast Ethernet* - as placas de interconexão *Fast Ethernet* possuem uma vazão nominal de 100 Mbits/s. Essas placas surgiram em 1994 para solucionar o problema de congestionamento existente na Ethernet 10Mbits/s (RIGONI; DIVERIO; NAVAUX, 1999);

- *Myrinet* - essa arquitetura é baseada em comunicação de pacotes. As características que tornam a *Myrinet* uma rede de alto desempenho são: canais robustos, controle de fluxo, controle de pacotes, controle de erros, baixa latência, interface que pode mapear a rede em rotas selecionadas. Uma ligação *Myrinet* é composta por um par de canais *full-duplex* que permite uma taxa de transferência de cerca de 2.00 Gbits/s cada um (RIGONI; DIVERIO; NAVAUX, 1999);

- *SCI (Scalable Coherent Interface)* - o SCI é um padrão que especifica hardware e protocolos para conexão em uma rede de alta velocidade. O SCI possui um espaço de endereçamento físico de 64 bits entre os nodos que permite operações de leitura e escrita na área de memória compartilhada entre os nodos. O SCI especifica uma largura de banda inicial de 1 Gbits/s para ligações seriais e 8 Gbits/s usando um canal paralelo (RIGONI; DIVERIO; NAVAUX, 1999);

- *Gigabit Ethernet* - as redes Gigabit Ethernet permitem operações *half-duplex* e *full-duplex* a uma taxa de 1Gbits/s. A principal vantagem das redes Gigabit Ethernet é o custo relativamente baixo, no entanto, a padronização com a tecnologia Ethernet limita o desempenho que pode ser obtido (CARVALHO, 2002).

Foram feitos alguns testes nos *clusters* do Instituto de Informática da UFRGS para obtenção da largura de banda e latência das redes utilizadas pelos mesmos (CARVALHO, 2002). Os resultados obtidos nos testes podem ser vistos na tabela 2.1.

Tabela 2.1: Características das redes de interconexão

	Pico	Largura de Banda Máxima	Latência Mínima
Myrinet	Teórico	160 Mbits	3 ms
	Prático	110 Mbits	7 ms
SCI	Teórico	500 Mbits	2,3 ms
	Prático	80 Mbits	10 ms
Fast Ethernet	Teórico	100 Mbits	80 ms
	Prático	9,5 Mbits	150 ms

2.1.2 Tipo de Memória Segundo Localidade

O tipo de memória existente em um *cluster* influencia no desempenho obtido nas aplicações que nele são executadas e deve ser levado em consideração no desenvolvimento das mesmas.

Baseando-se no tipo de memória existente, os *clusters* podem ser divididos em duas categorias:

- *Memória Distribuída* - um *cluster* é um agregado de nodos independentes, tendo cada nodo sua memória local. Sendo assim, o espaço de endereçamento de um nodo em um *cluster* não pode ser acessado por outro de maneira direta, podendo-se dizer, então, que todos os *clusters* possuem memória distribuída. Existem implementações de camadas em *hardware* e *software* que permitem simular uma memória compartilhada em nodos com memória distribuída. Tais camadas são chamadas de DSM (*Distributed Shared Memory*) e adicionam um custo que pode diminuir o desempenho da aplicação (DREIER; MARKUS; THEO, 1998);

- *Memória Híbrida* - um *cluster* possui memória híbrida quando ele tem memória distribuída e compartilhada em conjunto. Como já foi visto, a memória distribuída está entre os nodos que compõem o agregado, já a memória compartilhada está entre os processadores de cada nodo, quando eles posuírem mais de um. Então *clusters* com memórias híbridas são formados por nodos independentes e multiprocessadas, como por exemplo *clusters* de PCs duais.

2.1.3 O *Cluster* do Instituto de Informática da UFRGS

O *cluster* adotado neste trabalho, é constituído por vinte e um PCs, onde, vinte nodos são dedicados exclusivamente para processamento e um nodo é servidor. Esse *cluster* faz parte do Laboratorio de Tecnologia em *Clusters* (LabTeC), que é um laboratorio de pesquisa desenvolvido no Instituto de Informática da UFRGS em conjunto com a Dell do Brasil. A topologia deste *cluster* está ilustrada na fig. 2.1.

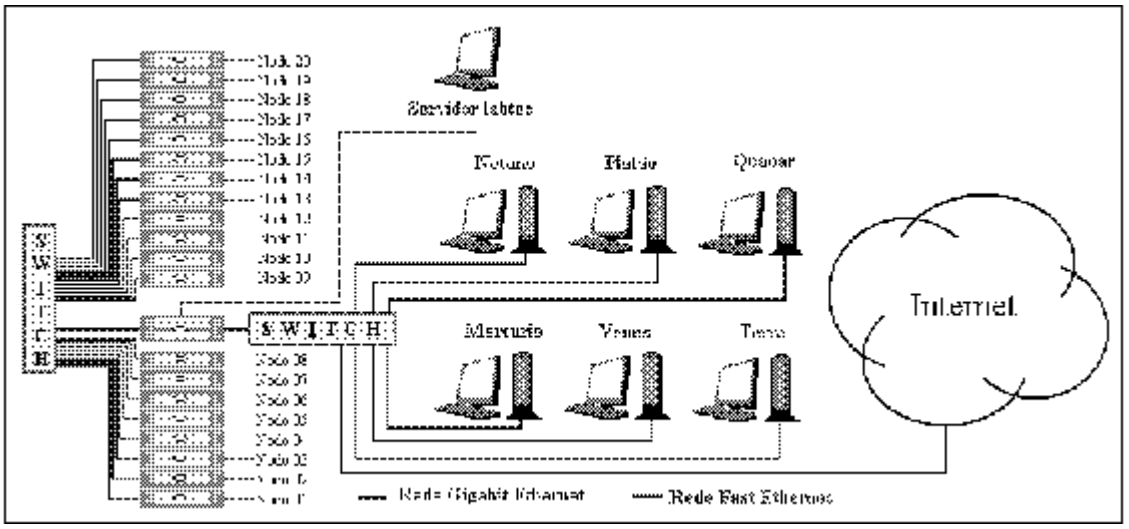


Figura 2.1: Topologia do *cluster* do Instituto de Informática (Labtec)

Os nodos de processamento deste *cluster* são interconectados através da rede *Fast Ethernet* por um *switch*, no qual também é interconectado o servidor através da rede *Gigabit Ethernet*.

Além do *switch* que interliga os nodos de processamento, o servidor também é interconectado a um segundo *switch*. Esse segundo *switch* tem a função de permitir o

acesso ao *cluster* pelas demais máquinas do LabTeC e, também, por máquinas externas ao LabTeC. As interconexões a este segundo *switch* são *Fast Ethernet*.

Na tab. 2.2 são apresentados mais detalhes referentes a arquitetura deste *cluster*.

Tabela 2.2: Arquitetura do *cluster* (LabTeC)

	NOME	CPU	RAM	Rede
0	labtec	Pentium IV Dual 1.8 GHz	1 GByte	*Gigabit Ethernet*
1	l1	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
2	l2	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
3	l3	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
4	l4	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
5	l5	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
6	l6	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
7	l7	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
8	l8	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
9	l9	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
10	l10	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
11	l11	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
12	l12	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
13	l13	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
14	l14	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
15	l15	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
16	l16	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
17	l17	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
18	l18	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
19	l19	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*
20	l20	Pentium III Dual 1.1 GHz	1 GByte	*Fast Ethernet*

No que se refere a este *cluster*, mais algumas características além das citadas acima são: todos os nodos possuem *cache* de 512 KBytes e discos rígidos SCSI, sendo que o servidor possui 36 GBytes e os nodos de processamento 18 GBytes.

No caso particular do *cluster* adotado neste trabalho, o acesso é feito através da máquina "labtec" que está conectada a todos os nodos de processamento. Quando deseja executar alguma aplicação no *cluster*, o usuário deve fazer o *login* na estação "labtec", e a partir dela, executar sua aplicação.

2.2 Software

Conforme já mencionado, dependendo da arquitetura do *cluster*, podem existir dois níveis de paralelismo a serem explorados. No caso específico do *cluster* do LabTec tem-se os dois níveis de paralelismo.

Para cada nível de paralelismo existem diferentes bibliotecas, de troca de mensagens ou de *threads*, visando sua exploração, onde cada biblioteca possui características diferentes das demais.

Nesta seção pretende-se fazer uma discussão e apresentação sobre as principais características de funcionamento e uso das bibliotecas adotadas neste trabalho.

2.2.1 Bibliotecas de Troca de Mensagens

As bibliotecas de troca de mensagens são ferramentas que possibilitam o desenvolvimento de aplicações paralelas em máquinas com memória distribuída.

A função de uma biblioteca de troca de mensagens é permitir que processos em diferentes máquinas possam trocar informações através de uma rede de interconexão.

As bibliotecas de troca de mensagens estão localizadas entre o sistema operacional e a aplicação. Essas bibliotecas são *softwares* que permitem o uso dos recursos do sistema operacional de maneira mais fácil. Uma ilustração da localização das bibliotecas de troca de mensagens pode ser vista na fig. 2.2.

Figura 2.2: Localização das bibliotecas de troca de mensagens

As primitivas da maioria das bibliotecas de troca de mensagens podem ser caracterizadas segundo três tipos:

- **primitivas de administração :** responsáveis entre outras funções por: criar grupos, informar aos processos suas identificações e informar o número total de processos existentes no grupo em que estão inseridos;
- **primitivas de comunicação ponto a ponto :** responsáveis por envio e recebimento de mensagens entre dois processos;
- **primitivas de comunicação em grupo :** responsáveis pela comunicação entre vários processos em um grupo.

A maioria das bibliotecas de troca de mensagens segue o paradigma SPMD *(Single Program Multiple Data)* (BAKER; SMITH, 1996), onde cada processador executa uma instância do mesmo programa ou algoritmo que, após ser colocado em execução, é chamado de processo. Cada instância desse programa recebe uma identificação única, baseada na qual os processos executam suas tarefas e identificam os demais processos em seu grupo de trabalho.

Embora no paradigma SPMD todos os processadores executem o mesmo algoritmo, estes não executam necessariamente as mesmas operações sobre os mesmos dados. Dependendo das identificações das instâncias dos processos e dos dados de entrada, cada instância do algoritmo pode efetuar operações diferentes das demais.

Neste trabalho foram utilizadas duas bibliotecas de troca de mensagens com o objetivo de efetuar comparações entre os desempenhos obtidos com cada uma delas. Essas

bibliotecas são o MPI - na implementação MPICH - e o DECK, que no decorrer deste capítulo serão apresentadas mais detalhadamente. .

2.2.1.1 MPI (Message Passing Interface)

O MPI é um padrão adotado para o desenvolvimento de biblioteca de comunicação através de troca de mensagens, sendo tal padrão largamente utilizado para exploração do paralelismo em arquiteturas caracterizadas por possuir memória distribuída. O MPI é portável para várias arquiteturas, tendo 125 primitivas para programação (PACHECO, 1997).

As primitivas de comunicação no MPI podem ser bloqueantes (espera confirmação do termino da comunicação) ou não bloqueantes (continua o processamento sem confirmação do termino da comunicação), com ou sem uso de *buffer*, o que torna o desenvolvimento de aplicações algo bastante flexível. Por exemplo, as primitivas de envio e recebimento padrão do MPI são respectivamente não bloqueante e bloqueante o que faz com que dois processos possam trocar dados primeiramente enviando e depois recebendo, sem ocorrer *deadlock* entre eles. Para isso basta que o *buffer* seja de tamanho suficiente para o armazenamento dos dados.

Existem algumas implementações do MPI que são de domínio público, entre as quais estão a MPICH(ANL/MSU)(GROPP; LUSK, 2002) e a LAN(*Ohio Supercomputer Center*)(LUMSDAINE et al., 2002). Embora essas implementações sejam muito semelhantes, a LAN tem a característica de permitir que diferentes processos façam parte de um grupo de trabalho, ou seja, não obriga a aplicação a seguir o paradigma SPMD.

Até o momento, as implementações existentes do MPI - que são do padrão MPI-1.0 - não dão suporte a múltiplas *threads*. Essa característica faz com que o MPI só disponibilize identificadores para processos e não para *threads*. Para solucionar essa limitação estão sendo desenvolvidas implementações do padrão MPI-2.0, onde se irá permitir o uso de primitivas MPI em múltiplas *threads*.

O MPI possui primitivas de comunicação ponto a ponto e, também, primitivas de comunicação em grupo, sendo possível a criação de vários grupos diferentes. No MPI existe um grupo de processos padrão chamado *MPI_COMM_WORLD* que contem todos os processos inicializados no MPI e permite a cada um deles se comunicar com os demais.

Utilização do MPI

No desenvolvimento de aplicações paralelas que fazem uso de alguma das implementações do MPI na exploração do paralelismo inter-nodos, as primeiras primitivas a serem usadas são:

- *MPI_Init(int argc, char argv[])* - primitiva que inicializa os processos;
- *MPI_Comm_rank(MPI_Comm comm, int *ident)* - primitiva que obtém o identificador do processo;
- *MPI_Comm_size(MPI_Comm comm, int *n)* - primitiva que obtém o número total de processos em um grupo.

Após a inicialização dos processos e obtenção dos identificadores e número total de elementos nos grupos, os processos podem iniciar a transmissão de dados. No MPI são vários os tipos de dados que podem ser trocados, conforme pode ser visto em (PACHECO, 1997).

Entre os diversos tipos de dados que podem ser transmitidos um que merece destaque é o *packet*. Um *packet* é formado pelo agrupamento de várias estruturas de dados, as quais podem ter diferentes tipos. Uma vantagem da comunicação de *packets* é o menor tempo de latência, pois o envio de um *packet* tem menor latência que o envio de várias estruturas de dados separadamente. Todavia vale ressaltar que existe um custo para empacotar e desempacotar os dados.

Nessas primitivas de empacotamento e desempacotamento é feito um controle de posicionamento dos dados através de um apontador, que é incrementado sempre que um novo dado for inserido no pacote e decrementado quando este for extraído do mesmo. Para facilitar o posicionamento deste apontador no desempacotamento dos dados, estes devem seguir a mesma ordem adotada nos empacotamentos.

No MPI as primitivas para efetuarem o empacotamento e o desempacotamento de dados são respectivamente:

- *MPI_Pack(void *buffer, int count, MPI_Datatype datatype, void *buffer_pack, int size_pack, int *apont, MPI_Comm comm)* - essa primitiva tem a função de empacotar vários dados em um pacote. O dado a ser empacotado está identificado pelo ponteiro *buffer*, esse dado possui tamanho equivalente a *count* e é do tipo *datatype*. O pacote que recebe os dados está identificado pelo ponteiro *buffer_pack* que pode armazenar no máximo o equivalente a *size_pack*. Nessa primitiva a posição dos dados é controlada pelo apontador *apont*;

- *MPI_Unpack(void *buffer_pack, int size_pack, int *apont, void *buffer, int count, MPI_Datatype datatype, MPI_Comm comm)* - essa primitiva efetua o desempacotamento dos dados. Seus parâmetros, embora estejam em uma ordem diferente, são os mesmos encontrados no *MPI_Pack*.

Para exemplificar o empacotamento de vários dados e o envio do pacote resultante para outro processo onde ocorrerá o desempacotamento é necessário descrever algumas primitivas de comunicação. As principais primitivas de comunicação ponto a ponto no MPI são:

- *MPI_Send(void * buffer, int count, MPI_Datatype datatype, int proc, int tag, MPI_Comm comm)* - essa primitiva tem como função o envio dos dados. O dado a ser enviado está identificado pelo ponteiro *buffer*, ele é do tipo *datatype* e de tamanho equivalente a *count*. O processo destino possui identificador igual a *proc*. Além dos dados já citados, nessa comunicação é permitido o envio de um rótulo através da variável *tag*;

- *MPI_Recv(void * buffer, int count, MPI_Datatype datatype, int proc, int tag, MPI_Comm comm, MPI_Status status)* - essa primitiva tem a função de receber dados. Seus parâmetros, embora estejam em uma ordem diferente, são basicamente os mesmos encontrados no *MPI_Send*, exceto pelo parâmetro *status* que informa se a operação foi bem sucedida e o número do processo que enviou a mensagem.

Nas figs. 2.3 e 2.4 estão sendo ilustrados respectivamente o envio e o recebimento de um pacote através da comunicação ponto a ponto.

No exemplo da fig. 2.3 é enviado um pacote contendo três tipos de dados diferentes, um valor *integer*, um vetor de *floats* e um vetor de dados do tipo *integer*. Esses dados são empacotados no *pack* e enviados para o nodo com identificador igual a *dest*.

Emissor

```
char pack[M_B];
int ttn,ap; int *vi; float *vf;

ap = 0;

MPI_Pack(&ttn,1,MPI_INT,&pack,M_B,&ap,MPI_COMM_WORLD);
MPI_Pack(vf,ttn,MPI_FLOAT,&pack,M_B,&ap,MPI_COMM_WORLD);
MPI_Pack(vi,ttn,MPI_INT,&pack,M_B,&ap,MPI_COMM_WORLD);

MPI_Send(&pack,ap,MPI_PACKED,dest,0,MPI_COMM_WORLD);
```

Figura 2.3: Envio de pacote no MPI

Receptor

```
char pack[M_B];
int ttn,ap; int *vi; float *vf;

ap = 0;

MPI_Recv(&pack,M_B,MPI_PACKED,org,0,MPI_COMM_WORLD,&Status);

MPI_Unpack(&pack,M_B,&ap,&ttn,1,MPI_INT,MPI_COMM_WORLD);
MPI_Unpack(&pack,M_B,&ap,vf,ttn,MPI_FLOAT,MPI_COMM_WORLD);
MPI_Unpack(&pack,M_B,&ap,vi,ttn,MPI_INT,MPI_COMM_WORLD);
```

Figura 2.4: Recebimento de pacote no MPI

Na fig. 2.4 está ilustrado um exemplo onde é recebido um pacote cujo o envio foi exemplificado na fig. 2.3.

Além da comunicação ponto a ponto, o MPI possui comunicação em grupo. A comunicação em grupo pode ser dividida em dois tipos: primitivas de comunicação em grupo para efetuar a movimentação de dados e primitivas de comunicação em grupo para computação global.

Entre as principais primitivas de comunicação em grupo para efetuar a movimentação de dados podemos destacar:

- *MPI_Broadcast(void *sendbuffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)* - essa primitiva tem a função de enviar dados para os demais processos. O dado a ser enviado é identificado pelo ponteiro *sendbuffer*, é do tipo *datatype* e ocupam um espaço equivalente a *count*. Nessa primitiva o emissor dos dados é identificado através do quarto parâmetro, que é representado pela variável *root*;

- *MPI_Scatter(void *sendbuffer, int sendcount, MPI_Datatypes sendtype, void *recvbuffer, int reccount, MPI_Datatypes rectype, int root, MPI_Comm comm)* - essa primitiva tem a função de particionar dados. O dado a ser particionado está identificado pelo ponteiro *sendbuffer*, esse dado é do tipo *sendtype* e ocupa um espaço equivalente a *sendcount*. O emissor é identificado pelo sétimo parâmetro. Cada uma das partições será atribuída à posição de memória apontada por

recvbuffer dos demais processos, esse dado é do tipo *rectype* e ocupa um espaço equivalente a *reccount*;

- *MPI_Gather(void *sendbuffer, int sendcount, MPI_Datatypes sendtype, void *recvbuffer, reccount, MPI_Datatypes rectype, int root, MPI_Comm comm)* - essa primitiva efetua a operação inversa ao *Scatter*, ou seja, o processo raiz recebe dados de todos os demais processos e os agrupa. Seus parâmetros, embora estejam em uma ordem diferente são, basicamente, os mesmos encontrados no *MPI_Scatter*.

Já entre as principais primitivas de comunicação em grupo para efetuar computações globais pode-se destacar:

- *MPI_Reduce(void *sendbuffer, void *recbuffer, int count, MPI_Datatype datatype, MPI_Op operação, int root, MPI_Comm comm)* - essa primitiva efetua uma operação pré-definida sobre dados de todos os processos do grupo. A operação é representada pelo quinto parâmetro. Os dados da operação estão identificados pelo ponteiro *sendbuffer* e o resultado é atribuído à posição de memória identificada pelo ponteiro *recbuffer*. Esses dados são do tipo *datatype* e de tamanho equivalente a *count*. O resultado da operação é enviado para o processo representado por *root*;

- *MPI_Allreduce(void *sendbuffer, void *recbuffer, int count, MPI_Datatype datatype, MPI_Op operação, MPI_Comm comm)* - essa primitiva é semelhante ao *Reduce*, mas o envio do resultado é para todos os processos do grupo;

- *MPI_Scan(void *sendbuffer, void *recbuffer, int count, MPI_Datatype datatype, MPI_Op operação, MPI_Comm comm)* - essa primitiva efetua uma operação pré-definida sobre os dados dos processos com identificação menor ou igual ao seu. Os parâmetros dessa primitiva são os mesmos parâmetros da primitiva *MPI_Reduce*.

Durante as comunicações, principalmente ponto a ponto, pode ocorrer a necessidade de um sincronismo entre todos os processos. Para isso o MPI fornece uma primitiva que bloqueia o processo corrente até que todos os demais processos do grupo também executem essa primitiva. Sua sintaxe é *"MPI_Barrier(MPI_Comm comm)"*.

Além dos possíveis sincronismos durante as execuções dos processos é necessário um sincronismo para finalizá-los. Para isso o MPI disponibiliza a primitiva *"MPI_Finalize()"* que bloqueia o processo até que essa primitiva seja executada por todos os demais processos.

2.2.1.2 DECK (Distributed Execution and Communication Kernel)

O DECK *(Distributed Execution and Communication Kernel)* é uma biblioteca de troca de mensagens desenvolvida pelo Grupo de Processamento Paralelo e Distribuído (GPPD) do Instituto de Informática da Universidade Federal do Rio Grande do Sul.

Semelhante ao MPI, o DECK possui primitivas de comunicação ponto a ponto e também primitivas de comunicação em grupo.

As primitivas de comunicação ponto a ponto no DECK para envio são não bloqueantes e as para recebimento são bloqueantes. Essas primitivas fazem uso de *buffer*, característica que permite evitar *deadlock* se todos os envios de mensagens forem efetuados antes dos recebimentos e se não ocorrer falta de *buffer*;

Para o uso das primitivas de comunicação em grupo, existe a necessidade de criação de um grupo, pois no DECK não existe nenhum grupo padrão.

Além das primitivas de troca de mensagens, o DECK possui primitivas para uso de *threads*, que permitem a criação, sincronização e outras operações envolvendo *threads* (BARRETO, 2000). Vale ressaltar que as primitivas de uso de *threads* do DECK são uma API (*Application Programming Interface*) da biblioteca *Pthreads*.

O DECK possui como funcionalidade extra em relação ao MPI a possibilidade de possuir várias *threads* enviando e recebendo mensagens simultaneamente dentro de um mesmo processo, ou seja, ele identifica diferentes *threads* em um processo (*thread safe*).

Utilização do DECK

Na utilização do DECK é necessário inicializar os processos com suas identificações, estratégia semelhante à que ocorre no MPI. Isso é feito através das seguintes primitivas:

- *deck_init(int argc, char argv[])* - primitiva que inicializa os processos;
- *int ident = deck_node()* - primitiva que obtém o identificador do processo;
- *int n = deck_numnodes()* - primitiva que obtém o total de processos em um grupo.

Diferentemente do MPI, além da inicialização dos processos, no DECK existe a necessidade de criação de caixas de correio para a comunicação ponto a ponto e a criação de grupos para as comunicações coletivas. No decorrer deste capítulo será explicado e exemplificado como este processo deve ser feito.

No que se refere a tipos de dados, semelhantemente ao MPI, o DECK possui vários tipos de dados que podem ser transmitidos, no entando um que merece uma maior atenção é o dado do tipo *packet*.

Os *packets* no DECK são semelhantes aos do MPI e, além disso, no DECK a ordem dos desempacotamentos também deve ser a mesma adotada nos empacotamentos. As primitivas de empacotamento e desempacotamento no DECK são, respectivamente:

- *deck_msg_pack(void *buffer, DECK_Datatype Datatype, void *buffer_date, int count)* - essa primitiva tem a função de empacotar vários dados em apenas uma estrutura. O dado a ser empacotado é do tipo *datatype*, sua posição está identificada pelo ponteiro *buffer_date* e tem tamanho equivalente a *count*. O pacote que recebe esses dados é identificado pelo ponteiro *buffer_pack*;
- *deck_msg_unpack(void *buffer, DECK_Datatype Datatype, void *buffer_date, int count)* - essa primitiva efetua o desempacotamento dos dados. Seus parâmetros são os mesmos encontrados no *deck_msg_pack*.

Conforme mencionado, para efetuar trocas de mensagens no DECK é necessário criar caixas de correio. Existem basicamente dois tipos de caixas de correio que são as caixas de saída e as caixas de entrada. As caixas de correio de saída de dados devem ser clonadas com as caixas de correio de entrada que foram criadas pelos outros processos. A primitiva de clonagem é a responsável por informar que os dados de uma caixa de saída devem ser transferidos para uma devida caixa de entrada, esse primitiva é efetuada baseando-se nos rótulos definidos na criação das caixas.

Exemplos de programas ilustrando envio e recebimento de pacotes através da comunicação ponto a ponto podem ser vistos respectivamente nas figs. 2.5 e 2.6.

Nas figs. 2.5 e 2.6 pode-se observar que além da criação de caixas de correio, também é necessário a criação de uma mensagem para armazenar os dados (pacote). Após o

Emissor

```
int tm, tc;  deck_mbox_t *vbox;  deck_mbox_t mbr;  deck_msg_t m;

deck_init(&argc, &argv);  size = deck_numnodes();  my = deck_node();

vbox=malloc(size*sizeof(deck_mbox_t));
sprintf(cid,"%d",my);  strcat(cid,-r");
deck_mbox_create(&mbr,cid);
for (tc=0;tc < size;tc++)
if (tc != my) {
   sprintf(cid,"%d",tc);  strcat(cid,-r");
   deck_mbox_clone(&vbox[tc],cid); }

deck_msg_create(&m,250);
deck_msg_pack(&m,DECK_SHORT, &tm,1);
deck_msg_pack(&m,DECK_FLOAT, vf,tm);
deck_msg_pack(&m,DECK_SHORT, vi,tm);
deck_mbox_post(&vbox[tc], &m);
deck_msg_destroy(&m);
```

Figura 2.5: Envio de pacote no DECK

Receptor

```
int tm, tc;  deck_mbox_t *vbox;  deck_mbox_t mbr;  deck_msg_t m;

deck_init(&argc, &argv);  size = deck_numnodes();  my = deck_node();

vbox=malloc(size*sizeof(deck_mbox_t));
sprintf(cid,"%d",my);  strcat(cid,-r");
deck_mbox_create(&mbr,cid);
for (tc=0;tc < size;tc++)
if (tc != my) {
   sprintf(cid,"%d",tc);  strcat(cid,-r");
   deck_mbox_clone(&vbox[tc],cid); }

deck_msg_create(&m,250);
deck_mbox_retrv(&mbr, &m);
deck_msg_unpack(&m, DECK_SHORT, &tm, 1);
deck_msg_unpack(&m, DECK_FLOAT, vf,tm);
deck_msg_unpack(&m, DECK_FLOAT, vi,tm);
deck_msg_destroy(&m);
```

Figura 2.6: Recebimento de pacote no DECK

envio dos pacotes ou recebimento e desempacotamento dos pacotes, a mensagem deve ser zerada ou destruída.

Algumas características existentes na comunicação ponto a ponto como criação de caixas de correio e clonagem e criação/destruição de mensagens não permanecem na comunicação em grupo.

Na comunicação coletiva no DECK não existe nenhum grupo padrão, então deve-se criar um grupo que contenha os processos envolvidos no trabalho, para que esses possam fazer uso das primitivas de comunicação coletiva.

Para criar um grupo de processos, inicialmente uma estrutura de nomes tipo *deck_name_t* é declarada, então é alocado espaço para essa entrutura onde deve ser feito cadastro das *threads* pertencentes ao grupo. Após isso, é criado um grupo do tipo *deck_group_t*, sendo que esse grupo já deve ter sido declarado e sua criação faz uso da estrutura *deck_name_t*.

As primitivas de comunicação coletivas disponíveis no DECK para movimentação de dados são:

- *deck_group_bcast(deck_group_t *group, void *sendbuffer, int count, deck_name_t name_thread)* - essa primitiva tem a função de enviar dados para os demais processos. O dado a ser enviado está identificado pelo ponteiro *sendbuffer* e ocupa um espaço equivalente a *count*. Nessa primitiva o grupo é identificado pelo primeiro parâmetro e o emissor dos dados através do último parâmetro;

- *deck_group_scatter(deck_group_t *group, void *recbuffer, void *sendbuffer, DECK_Datatype datatype, int count, deck_name_t name_thread)* - essa primitiva tem a função de particionar dados. O dado a ser particionado é identificado pelo ponteiro *sendbuffer*, seu tipo é *datatype* e ocupam um espaço equivalente a *count*. As partições serão atribuídas à posição de memória identificada pelo ponteiro *recvbuffer* dos demais processos. O emissor dos dados é identificado pelo último parâmetro;

- *deck_group_gather(deck_group_t *group, void *recbuffer, void *sendbuffer, DECK_Datatype datatype, int total_items, deck_name_t name_thread)* - essa primitiva efetua a operação inversa ao *Scatter*, ou seja, o processo raiz recebe dados de todos os demais processos e os agrupa.

Já para computação global as primitivas disponíveis no DECK são:

- *deck_group_reduce(deck_group_t *group, void *recbuffer, void *sendbuffer, DECK_Datatype datatype, int count, deck_name_t name_thread, DECK_Op operação)* - essa primitiva efetua uma operação pré-definida sobre dados de todos os processos do grupo. A operação é representada pelo último parâmetro. O dado da operação está identificado pelo ponteiro *sendbuffer* e o resultado dessa é colocado na posição de memória identificada pelo ponteiro *recbuffer*. Esses dados são do tipo *datatype* e de tamanho equivalente a *count*. O resultado da operação é enviado para o processo representado por *name_thread*;

- *deck_group_allreduce(deck_group_t *group, void *recbuffer, void *sendbuffer, DECK_Datatype datatype, int count, DECK_Op operação)* - essa primitiva é semelhante ao *Reduce*, mas o envio do resultado é para todos os processos do grupo.

Além das primitivas de criação dos grupos e de comunicação já apresentadas, o DECK disponibiliza uma primitiva para sincronização das *threads* do grupo *"deck_group_barrier(deck_group_t *g)"* e uma primitiva para destruir os grupos já existentes *"deck_group_destroy(deck_group_t *g)"*.

Um exemplo de um programa em DECK que faz uso de primitivas de comunicação coletiva pode ser visto na fig. 2.7.

Comunicação em Grupo no DECK

```
long v[K];
long w[K];

deck_group_t g;
deck_name_t *names;

deck_init (&argc, &argv);

total_items = deck_numnodes() * K;

names = (deck_name_t *) malloc(sizeof(deck_name_t) * deck_numnodes());

for (i=0; i < deck_numnodes(); i++) {
   sprintf(names[i], "#%i_main", i); }

deck_group_create(&g, "foo", names, deck_numnodes());

deck_group_reduce(&g, w, v, DECK_LONG, K, "#0_main", DECK_CC_MEAN);

deck_group_barrier(&g);

deck_group_destroy(&g);

deck_done();
```

Figura 2.7: Comunicação em grupo no DECK

Conforme já mencionado, o DECK permite que diferentes *threads* em um mesmo processo recebam e enviem mensagens. As primitivas para criação e gerenciamento de *threads* no DECK são:

- *deck_thread_create(deck_thread_t *t, void *start, void *parm)* - primitiva que cria e coloca uma *thread* em estado de pronta para ser escalonada;
- *deck_thread_yield()* - primitiva que faz com que a *thread* libere o processador e volte para o estado de pronta na fila do escalonador;
- *deck_thread_sleep(unsigned long time)* - primitiva que faz com que a *thread* fique em estado de espera por um determinado evento por um determinado tempo em segundos;
- *deck_thread_usleep(unsigned long usec)* - primitiva com função muito semelhante a *deck_thread_sleep* mas diferindo pelo fato do tempo de espera ser em microsegundos;
- *deck_thread_join(deck_thread_t t)* - primitiva que faz com que o processo fique em espera até que a *thread* identificada termine sua execução.

2.2.2 Bibliotecas de *threads*

As bibliotecas de *threads* são utilizadas em programação paralela com o objetivo de aumentar o desempenho das aplicações. Esse aumento de desempenho se deve ao fato de múltiplas threads terem um custo de criação e de comunicação menor que o custo existente em múltiplos processos.

Essas bibliotecas são mais adequadas para ambientes multiprocessados e proporcionam uma maior flexibilidade no gerenciamento, escalonamento e sincronismo se comparadas com processos.

Um fluxo de execução *(thread)*, algumas vezes chamado de processo leve *(lightweight process)*, é uma unidade básica de utilização da CPU e consiste em um contador de instruções, um conjunto de registradores e um espaço de pilha. Ela compartilha com *threads* irmãs suas seções de código, a seção de dados e os recursos do sistema operacional. Um processo tradicional é igual a uma tarefa com uma única *thread* (SILBERSCHATZ; GALVIN, 2000).

Segundo (KLEIMAN; SHAH; SMAALDERS, 1996) o uso de múltiplas *threads* é um eficiente recurso para aplicações que fazem uso de paralelismo pois, em processos com múltiplas *threads*, o sistema operacional pode escaloná-las para processadores diferentes, o que resulta em ganho de desempenho em algumas aplicações.

2.2.2.1 Pthread

Desenvolvido pelo comitê POSIX*(Portable Operating System Interface)*, a *Pthread* é um padrão adotado para o desenvolvimento de bibliotecas de *threads*.

Segundo (BARRETO, 2000) as primitivas de *threads* existentes no DECK são uma API das primitivas *Pthreads*.

Na programação com a *Pthread* na linguagem C, uma *thread* é um procedimento que ao ser invocado executa paralelamente aos outros fluxos já existentes. Esta nova *thread* ou fluxo, pode ter acesso a todas as variáveis globais ou declaradas dentro de seu escopo.

De acordo com (SILBERSCHATZ; GALVIN, 2000) existem dois tipos de *Pthreads*. O primeiro tipo é conhecida como *thread* de sistema, pois é escalonada pelo sistema operacional e recebe as mesmas proporções de tempo de processador que os processos. Já o segundo tipo, conhecida como *thread* de aplicação, é escalonada em nível de aplicação, recebendo uma parte de tempo de processador que é atribuído ao processo a que ela pertence.

Múltiplas *threads* de sistema têm como vantagem o fato de poderem estar executando paralelamente em diferentes processadores, mas seus custos de escalonamento são equivalentes aos custos de escalonamento de um processo (SILBERSCHATZ; GALVIN, 2000).

Múltiplas *threads* de aplicação têm como vantagem um menor custo de escalonamento em relação ao custo de escalonamento de um processo (SILBERSCHATZ; GALVIN, 2000).

O padrão de criação de *threads* da biblioteca *Pthreads* e da API para *threads* existente no DECK é de *threads* de sistema, pois essas visam o ganho em desempenho de aplicações em máquinas multiprocessadas.

As primitivas da biblioteca *Pthread* utilizadas para criar, gerenciar e destruir uma *thread* são:

- *pthread_create(pthread_t * thread, pthread_attr_t *attr, void * (*start_routine)*

*(void *), void * arg)* - primitiva que cria uma *thread*;

- *pthread_join(pthread_t th, void **thread_return)* - primitiva que faz com que o processo fique em espera até que a *thread* identificada termine sua execução;

- *pthread_exit(void *retval)* - primitiva que destrói uma *thread*.

Devido ao fato de múltiplas *threads* poderem acessar concorrentemente as mesmas variáveis, podem ocorrer casos onde múltiplas *threads* acessem e alterem o mesmo dado ao mesmo tempo causando "condições de corrida". Uma "condição de corrida"pode causar inconsistência nos dados e deve ser evitada.

Uma alternativa para evitar a ocorrência de condições de corrida é a criação de áreas de acesso, que devem permitir o acesso exclusivo às variáveis dentro dessa área por apenas uma *thread* em um dado momento. Essas áreas são chamadas de áreas de exclusão mútua e podem possuir diversas variáveis e operações. Uma alternativa para implementação de áreas de exlusão mútua é através dos recursos *mutex* e *cond* da biblioteca Pthreads, onde as principais primitivas são:

- *pthread_mutex_lock(pthread_mutex_t *mutex)* - indica o inicio de uma área de exclusão mútua identificada por um *mutex*;

- *pthread_mutex_unlock(pthread_mutex_t *mutex)* - indica o final de uma área de exclusão mútua identificada por um *mutex*;

- *pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)* - bloqueia a *thread* em uma área de exclusão mútua identificada por um *mutex* até que um sinal identificado por uma condição seja recebido;

- *pthread_cond_signal(pthread_cond_t *cond)* - envia um sinal identificado por uma condição para uma *thread* bloqueada;

- *pthread_cond_broadcast(pthread_cond_t *cond)* - envia um sinal identificado por uma condição para todas as *threads* bloqueadas.

2.2.2.2 Rthread

A *Rthread* é uma biblioteca de *threads* remotas que permite que múltiplas *threads* declaradas em um mesmo processo executem em diferentes máquinas com memória distribuída (DREIER; MARKUS; THEO, 1998).

Para o uso da *Rthread*, que é uma extensão da *Pthread*, os programas podem ser desenvolvidos com *Pthread* e passar por um processo de pre-compilação. Esse processo faz uma análise da granularidade das *threads*. As *threads* com maior carga de processamento serão transformadas em *Rthreads* (DREIER; MARKUS; THEO, 1998).

O acesso aos dados pelas *threads* remotas é feito através de troca de mensagens com as bibliotecas MPI e PVM, dependendo da implementação da *Rthread*.

Devido ao fato da *Rthread* fazer uso de troca de mensagens, após a pre-compilação é feito um processo de otimização que tem por função diminuir o número de mensagens.

A biblioteca *Rthread* pode ser vista como um mecanismo de DSM (*Distributed Shared Memory*) que simula dados compartilhados em memória distribuída.

2.2.2.3 *OpenMP*

A OpenMP é uma biblioteca de *threads* que permite que múltiplas *threads* executem partes diferentes de um programa simultaneamente em máquinas com memória compartilhada (OPENMP: SIMPLE, PORTABLE, SCALABLE SMP PROGRAMMING, 2002).

A principal caracteristica da biblioteca OpenMP é tornar mais fácil a programação, pois permite dividir as iterações de um laço entre diversos processadores, oferecendo quase que um paralelismo implícito.

A biblioteca OpenMP possui implementação em diversos sistemas operacionais como Linux e Windows sendo largamente portável. Essa biblioteca tem implementações para as linguagens C, C++ e Fortran (OPENMP: SIMPLE, PORTABLE, SCALABLE SMP PROGRAMMING, 2002).

2.3 Aplicação:Modelos Computacionais

Um modelo matemático tem por objetivo simular um fenômeno natural, através do cálculo de suas variáveis, essenciais, como por exemplo: velocidade, escoamento de um fluido, temperatura, direção do vento, entre outras. Para representar um fenômeno natural de maneira precisa, um modelo deve representar diversas variáveis independentes, as quais representam as propriedades deste processo natural.

Um modelo com várias variáveis envolve necessariamente derivadas com respeito a cada uma dessas variáveis ou derivadas parciais, sendo, portanto, necessario empregar equação diferencial "parcial" (EDPs) para representa-lo (CUMINATO; MENEGETT, 2000).

2.3.1 Discretização de EDPs

A solução das EDPs que compõem um modelo é fundamental para a simulação do fenômeno representado por estas. Devido ao fato de EDPs de modelos complexos não terem uma solução analítica, faz-se necessário a discretização destas EDPs.

O processo de discretização tem o objetivo de transformar as EDPs contínuas que são definidas em um domínio contínuo, isto é, com um número infinito de pontos, em EDPs discretas que são definidas em um domínio discreto (finito).

O processo de discretização é feito através da definição de uma malha e de um esquema de aproximação para os valores nodais sobre a região onde se busca a solução, chamado de domínio computacional. Com a criação da malha o domínio é dividido em células elementares e, em cada célula, as derivadas parciais e os outros termos presentes na EDP são aproximadas em função dos valores das variáveis dependentes.

As equações resultantes da discretização das EDPs recebem o nome de equações de diferenças. A união das equações diferenças de todos os pontos da malha gera um sistema de equações. Assim, solucionar o modelo matemático consiste em solucionar o sistema de equações gerado (TEODORO, 1998).

Desta forma, quanto mais refinada for a malha, isto é, quanto maior o número de pontos discretos, mais acurada será a simulação efetuada pelo modelo. Sendo necessario, também, que as EDPs tenham a mesma característica de projeção adotada na discretização.

A estrutura do sistema de equações resultante da discretização de um determinado

domínio depende da estratégia de discretização adotada e, também, de características intrínsecas das EDPs.

Além da estrutura dos sistemas, outro fator a ser considerado é a geometria da região onde o fenômeno é resolvido. Neste caso, na discretização, o formato do domínio discretizado irá, também, definir a disposição dos elementos no sistema gerado. A discretização em diferenças finitas sobre malhas cartesianas de domínios retangulares gera matrizes estruturadas e regulares, tendo forma de bloco diagonal principal. Já a discretização em domínios não retangulares resulta em matrizes estruturadas e irregulares (CUMINATO; MENEGETT, 2000).

Na seção abaixo é apresentado um exemplo para a geração de sistemas de equações lineares a partir da discretização da EDP da difusão bidimensional em um domínio irregular. A equação da difusão é utilizada na modelagem de fenômenos como, por exemplo: temperatura, velocidade ou concentração (CANAL, 2000).

2.3.2 EDP da Difusão

A equação diferencial da difusão pode ser uni ou multidimensional. Qualquer que seja sua dimensão ela sempre se refere ao espaço. Além disso, essa equação é dita não-estacionária pois evolui com o tempo (CANAL, 2000).

A EDP da difusão bidimensional é dada pela equação 2.1, onde os termos α_x e α_y são os termos de difusibilidade e dependem das propriedades físicas do material, que são obtidas através empiricamente (FLETCHER, 1988). A variável T pode representar velocidade, temperatura ou concentração. Isso depende do tipo de difusão que está sendo tratado: tempo, calor ou massa. essa EDP 2D é escrita como :

$$\frac{\partial T}{\partial t} = \alpha_x \frac{\partial^2 T}{\partial x^2} + \alpha_y \frac{\partial^2 T}{\partial y^2} \tag{2.1}$$

Dado um domínio irregular representado pela fig. 2.8, deseja-se encontrar a solução numérica da equação da difusão. Os valores em negrito correspondem aos valores nodais internos, enquanto os valores não negritos correspondem a valores nodais na fronteira do domínio.

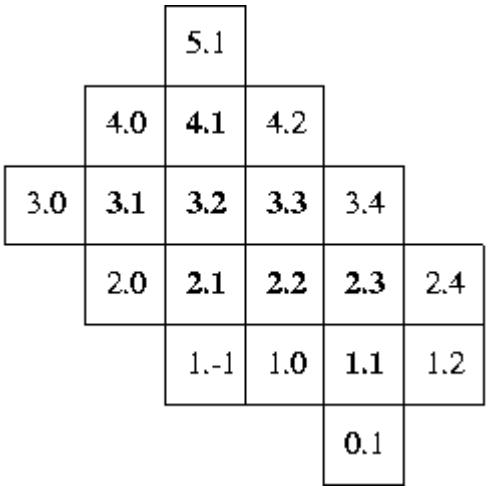


Figura 2.8: Domínio irregular (enumeração natural)

Para encontrar uma solução aproximada numérica da EDP da difusão sobre o domínio dado é necessário que essa seja discretizada. Considerando uma discretização em

diferenças finitas centradas no espaço e adiantada no tempo, usando estêncil de cinco pontos, a equação 2.1 e escrita como:

$$\frac{T_{(i,j)}^{m+1} - T_{(i,j)}^{m}}{\triangle t} + \alpha_x \left(\frac{T_{(i+1,j)}^{m+1} - 2T_{(i,j)}^{m+1} + T_{(i-1,j)}^{m+1}}{\triangle x^2} \right) + \alpha_y \left(\frac{T_{(i,j+1)}^{m+1} - 2T_{(i,j)}^{m+1} + T_{(i,j-1)}^{m+1}}{\triangle y^2} \right) \tag{2.2}$$

Tomando $S_x = \alpha_x \frac{\triangle t}{\triangle x^2}$ e $S_y = \alpha_y \frac{\triangle t}{\triangle y^2}$ em (2.2), tem-se o sistema de equações é dado por (2.3).

$$-S_x T_{(i-1,j)}^{m+1} + (1 + S_x + S_y) T_{(i,j)}^{m+1} - S_x T_{(i+1,j)}^{m+1} - S_y T_{(i,j+1)}^{m+1} - S_y T_{(i,j-1)}^{m+1} = T_{(i,j)}^{m} \tag{2.3}$$

A equação (2.3) pode ser escrita como

$$E T_{(i-1,j)}^{m+1} + C T_{(i,j)}^{m+1} + D T_{(i+1,j)}^{m+1} + F T_{(i,j+1)}^{m+1} + S T_{(i,j-1)}^{m+1} = T_{(i,j)}^{m} \tag{2.4}$$

onde $E = -S_x$, $D = -S_x$, $C = (1 + S_x + S_y)$, $F = -S_y$ e $S = -S_y$.

Assim, a discretização do domínio visto na fig. 2.8 resulta no sistema de equações lineares (2.4), e que é escrita na forma matricial $Ax = b$ mostrado na fig. 2.9. Esse pode ser resolvido através da utilização de um método numérico.

$$\begin{bmatrix}
C_{1,1} & & & S_{1,1} & & & & \\
 & C_{2,1} & D_{2,1} & & & S_{2,1} & & \\
 & E_{2,2} & C_{2,2} & D_{2,2} & & & S_{2,2} & \\
F_{2,8} & & E_{2,8} & C_{2,8} & & & & \\
 & & & & C_{8,1} & D_{8,1} & & \\
 & F_{8,2} & & & E_{8,2} & C_{8,2} & D_{8,2} & S_{8,2} \\
 & & F_{8,8} & & & E_{8,8} & C_{8,8} & \\
 & & & & & F_{4,1} & & C_{1,1}
\end{bmatrix} * \begin{bmatrix} T_{1,1}^{m+1} \\ T_{2,1}^{m+1} \\ T_{2,2}^{m+1} \\ T_{2,8}^{m+1} \\ T_{8,1}^{m+1} \\ T_{8,2}^{m+1} \\ T_{8,8}^{m+1} \\ T_{4,1}^{m+1} \end{bmatrix} = \begin{bmatrix}
T_{1,1}^{m} - E_{1,1}T_{1,0}^{m+1} - D_{1,1}T_{1,2}^{m+1} - F_{1,1}T_{0,1}^{m+1} \\
T_{2,1}^{m} - E_{2,1}T_{2,0}^{m+1} - F_{2,1}T_{1,-1}^{m+1} \\
T_{2,2}^{m} - F_{2,2}T_{1,0}^{m+1} \\
T_{2,8}^{m} - D_{2,8}T_{2,4}^{m+1} - S_{2,8}T_{8,4}^{m+1} \\
T_{8,1}^{m} - E_{8,1}T_{8,0}^{m+1} - F_{8,1}T_{2,0}^{m+1} - S_{8,1}T_{4,0}^{m+1} \\
T_{8,2}^{m} \\
T_{8,8}^{m} - D_{8,8}T_{8,4}^{m+1} - S_{8,8}T_{4,2}^{m+1} \\
T_{4,1}^{m} - E_{4,1}T_{4,0}^{m+1} - D_{4,1}T_{4,2}^{m+1} - S_{4,1}T_{8,1}^{m+1}
\end{bmatrix}$$

Figura 2.9: Representação matricial para o dominínio da fig. 2.8 discretizado com diferenças finitas, já considerando-se condições de contorno

2.4 Métodos do Subespaço de Krylov

Os métodos do subespaço de Krylov se tornaram muito populares nas duas últimas décadas. Essa popularidade deve-se ao fato desses métodos serem construídos empregando idéias sofisticadas e também por serem iterativos, característica que os torna mais apropriados à paralelização se comparados com métodos diretos (SAAD, 1996).

O objetivo principal desse capítulo é apresentar dois métodos do subespaço de Krylov que são o Gradiente Conjugado (GC) e o Resíduo Mínimo Generalizado (GMRES), inicialmente é apresentada uma visão geral dos métodos iterativos.

2.4.1 Métodos Iterativos

Na solução de sistemas de equações lineares existem basicamente duas classes de métodos que podem ser aplicados: a dos métodos diretos e a dos métodos iterativos.

Os métodos diretos têm a vantagem de serem robustos e de fornecerem uma solução numérica exata, exceto por erros de arredondamento. A grande desvantagem é que a seqüência de operações elementares entre linhas e/ou colunas pode causar o processo de *fill-in*, (que é o surgimento de elementos não nulos em posições onde não existiam), destruindo a esparsidade da matriz e, portanto, dificultando o armazenamento e a otimização. Além disso, os métodos diretos apresentam dependência global nas operações, o que torna complicada a sua paralelização (RIZZI, 2001).

Já a segunda classe, a dos métodos iterativos, se divide em duas subclasses, os estacionários e os não estacionarios. Nos métodos estacionários os passos da iteração atual não dependem das iterações anteriores, já nos não estacionários, alguns passos possuem essa dependência (AXELSSON, 1996).

Os métodos iterativos não estacionários são eficientes quando aplicados a matrizes grandes e esparsas que possuem determinadas características. Essa eficiência decorre do fato de que esses métodos têm boas propriedades de convergência sobretudo quando as matrizes são bem condicionadas. Além disso, são construídos sobre operações básicas em álgebra linear e, portanto, são altamente paralelizáveis.

Entre as principais características que diferem essas duas classes está o fato dos métodos diretos procurarem uma solução exata do sistema e os iterativos gerarem uma seqüencia de vetores $x^{(k)}$, em k iterações, cujo limite é a solução exata do sistema.

Considerando o sistema de equações lineares apresentado em (2.4), ele pode ser escrito como:

$$Ax = b \tag{2.5}$$

onde $A \in \mathbb{R}^{n \times n}$ é uma matriz não singular tal que $a_{ii} \neq 0, 1 \leq i \leq n$, os métodos iterativos têm a forma apresentada em (2.6) onde $x \in \mathbb{R}^n$, $d \in \mathbb{R}^n$ e $A \in \mathbb{R}^{n \times n}$

$$x^{(k+1)} = Ax^{(k)} + d \tag{2.6}$$

onde $x^{(k+1)}$ é o vetor de solução do passo atual, A é a matriz dos coeficientes, $x^{(k)}$ é o vetor de solução do passo anterior e d o vetor que indica a direção da nova iteração.

Como os métodos iterativos geram uma seqüência de vetores $x^{(k)}$, há a necessidade de se definir um critério de parada, que indica qual o grau de acurácia que deve ser obtido na solução, para que essa possa satisfazer o sistema.

A solução obtida $x^{(k)}$ deve ser aproximada da exata x^*, de forma que:

$$\|x^{(k)} - x^*\| \leq \varepsilon \tag{2.7}$$

onde ε representa o "erro"tolerado e $\|.\|$ representa a "norma dos vetores".

Embora os métodos iterativos tenham como desvantagem a falta de robustez, os métodos aqui empregados, que são o GC e o GMRES, têm uma sólida base matemática e suas propriedades são bem conhecidas.

No caso particular em que as matrizes esparsas são diagonalmente dominantes, simétricas ou não simétricas, os métodos iterativos tem, respectivamente, pelo menos dois métodos particularmente importantes, o GC e o GMRES.

Segundo (ARAUJO, 1997a), os métodos iterativos não estacionários mais bem sucedidos provêm de processos de projeção ou minimização de funcionais, geralmente sobre uma sucessão ou sequência de subespaços de dimensões crescentes, chamados de espaços de Krylov.

2.4.2 Subespaço de Krylov

Os métodos do subespaço de Krylov são iterativos não estacionários e podem ser aplicados na solução de sistemas de equações do tipo apresentado em (2.8).

$$Ax = b. \tag{2.8}$$

Os métodos do subespaço de Krylov encontram uma solução aproximada $x^{(k)}$ onde $Ax^{(k)} = b$, para um subespaço de dimensão m, tal que:

$$x^{(0)} + K^m, \tag{2.9}$$

, o que ocorre por imposição da condição de Petrov-Galerkin (SAAD, 1996)

$$b - Ax^m \perp L^m \tag{2.10}$$

onde L^m é um subespaço de dimensão m. A estrutura do subespaço de Krylov é dada por:

$$K^m(A, r^0) = span\{r^0, Ar^0, A^2r^0, ..., A^{m-1}r^0\} \tag{2.11}$$

Os métodos pertencentes ao subespaço de Krylov podem ser divididos em três diferentes classes, dependendo da escolha do subespaço L^m (CHOW, 1997):

1. $L^m = K^m$: é a classe dos métodos de projeção ortogonal ou de Galerkin. Quando A é SDP, essa escolha de L^m minimiza a A-norma de erro. O Gradiente Conjugado é um exemplo dessa classe para matrizes simétricas;

2. $L^m = AK^m$: é a classe dos métodos dos resíduos mínimos. Essa escolha de L^m minimiza a norma residual $\|b - Ax^m\|_2$. O GMRES é um exemplo dessa classe para matrizes não-simétricas;

3. $L^m = K^m(A^T, r_0^*)$ onde r_0^* é um vetor não paralelo a r_0. Essa escolha de L^m é projetada para A não simétrica e fornece relações de recorrência para bases do subespaço de Krylov. Métodos dessa classe, embora tenham a vantagem de necessitar de menos armazenagem se comparados com o GMRES, tem a desvantagem de serem mais propensos a terem problemas de gerarem novos elementos para a base de vetores ou problemas de aritmética finita *breakdown* (ARAUJO, 1997b). Exemplos são o BCG e o CGS.

2.4.3 Gradiente Conjugado

O método do GC (Gradiente Conjugado) é um método iterativo não estacionário do subespaço de Krylov e é considerado um dos métodos mais eficientes para resolução de sistemas lineares de grande porte e esparsos (SHEWCHUK, 1994).

O Gradiente Conjugado é aplicado na solução de sistemas de equações lineares do tipo

$$Ax = b \tag{2.12}$$

onde x é o vetor com as incógnitas, b é o vetor dos termos independentes, e A é uma matriz simétrica, definida e positiva.

Diferentemente dos métodos iterativos clássicos, o GC soluciona o sistema baseando-se na minimização da função quadrática, e não na intersecção de hiper-planos (SHEWCHUK, 1994).

A forma da função quadrática é

$$f(x) = \frac{1}{2}x^T Ax - bx + c, \tag{2.13}$$

onde A é a matriz, b e x são os vetores e c é um escalar. O GC é definido pela função (2.14), dada por:

$$\nabla f \equiv f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b, \tag{2.14}$$

que devido ao fato de A ser simétrica $(A = A^T)$ pode ser simplificado para obter (2.15).

$$\nabla f \equiv f'(x) = Ax - b = 0. \tag{2.15}$$

O Gradiente Conjugado parte do princípio de que o gradiente, que é um campo vetorial, aponta para a direção mais crescente da função quadrática. Assim se a matriz é simétrica definida e positiva (SDP), o gradiente da função quadrática resume-se à solução de um sistema de equações lineares.

Existem diferentes algoritmos do GC que apresentam algumas modificações em relação ao algoritmo original. Neste trabalho o algoritmo do GC adotado foi o apresentado em (SHEWCHUK, 1994), mostrado na fig. (2.10).

1. $n \Leftarrow 0$
2. $r \Leftarrow b - Ax$
3. $d \Leftarrow r$
4. $\delta_{novo} \Leftarrow r^T r$
5. $\delta_0 \Leftarrow \delta_{novo}$
6. enquanto $n < n_{max}$ e $\delta_{novo} > \varepsilon^2 \delta_0$
7. $\quad q \Leftarrow Ad$
8. $\quad \alpha \Leftarrow \delta_{novo}/d^T q$
9. $\quad x \Leftarrow x + \alpha d$
10. $\quad$ se (n é divisivel por 50) e (n é diferente de 0)
11. $\quad\quad r \Leftarrow b - Ax$
12. $\quad$ senão
13. $\quad\quad r \Leftarrow r - \alpha q$
14. $\quad \delta_{velho} \Leftarrow \delta_{novo}$
15. $\quad \delta_{novo} \Leftarrow r^T r$
16. $\quad \beta \Leftarrow \delta_{novo}/\delta_{velho}$
17. $\quad d \Leftarrow r + \beta d$
18. $\quad n \Leftarrow n + 1$

Figura 2.10: Algoritmo do Gradiente Conjugado

O método do gradiente conjugado inicia a partir de uma solução arbitrária x^0 e vai aproximando-se da solução do sistema através de uma seqüencia de passos $x^1, x^2, ...x^n$. Esse processo ocorre até que uma aproximação atinja a acurácia desejada ε.

Para que os passos não sejam na mesma direção o gradiente conjugado faz uso do resíduo da iteração anterior, pois o resíduo é ortogonal a direção anterior.

A definição das direções de pesquisa implica na construção de um subespaço pela combinação linear dos resíduos:

$$d^n = span\{r^0, r^1, r^2, ...r^{n-1}\}. \tag{2.16}$$

O fato do resíduo ser ortogonal à direção anterior, implica que ele seja também ortogonal ao resíduo anterior, então, $r^n r^{n-1} = 0$. Baseando-se nessa propriedade de

ortogonalidade dos resíduos, cada novo resíduo e direção do gradiente conjugado é ortogonal aos obtidos em iterações anteriores, ou seja $r^n r^{n-x} = 0$, onde $x > 0$. Aqui tem-se a herança de informações das iterações anteriores, o que torna o gradiente conjugado um método não estacionário (CANAL, 2000).

Assim, cada nova direção de pesquisa é construida, como em (2.17), a partir do subespaço dos resíduos, para ser A-ortogonal a todos os resíduos e direções anteriores. Esse passo pode ser visto na linha 17 do algoritmo da fig. (2.10).

$$d^n = r^n + \beta d^{n-1} \tag{2.17}$$

O β da equação (2.17), dado por (2.18), é a razão entre a norma do resíduo atual e o anterior. Seu cálculo no algoritmo pode ser visto na linha 16.

$$\beta = \frac{r^n r^n}{r^{n-1} r^{n-1}} \tag{2.18}$$

O resíduo r das equações (2.17) e (2.18) pode ser obtido de dois modos. No primeiro utiliza-se o valor corrente do vetor x para calcular o valor exato do residuo .(2.19).

$$r^n = bAx^n \tag{2.19}$$

Esse calculo pode ser visto nas linhas 2 e 11 do algoritmo. Ele requer uma multiplicação matriz-vetor, que é muito custosa computacionalmente. Para reduzir o número de operações matriz-vetor, calcula-se o valor exato do resíduo somente a cada k iterações, como pode ser visto na linha 10. No segundo modo, dada por:

$$r^n = r^{n-1} - \alpha^n A d^n \tag{2.20}$$

o resíduo é calculado a partir do resíduo anterior. Elimina-se uma multiplicação matriz vetor, mas tem a desvantagem do erro no cálculo do resíduo ir se acumulando a cada nova iteração. Esse calculo pode ser visto na linha 13 do algoritmo.

O tamanho do passo necessário na equação α (2.20) é calculado na linha 8 do algoritmo através da expressão (2.21).

$$\alpha = \frac{r^n r^n}{d^n A d^n} \tag{2.21}$$

O vetor solução x na próxima iteração, utilizado na equação (2.19), é calculado na linha 9 do algoritmo, através da expressão:

$$x^n = x^{n-1} + \alpha^n A d^{n-1} \tag{2.22}$$

Conforme já mencionado, o gradiente conjugado vai pesquisando novas direções até que a aproximação atinja uma acurácia desejada, acurácia essa que é verificada na linha 6 do algoritmo.

A idéia do Gradiente Conjugado é, então, ir efetuando novas iterações na direção oposta à do gradiente (campo vetorial), de tal forma que as direções já pesquisadas não seja repetidas, até encontrar o mínimo global. A minimização ocorre sobre certos espaços de vetores, chamados de subespaço de pesquisa (espaço de Krylov), gerados a partir dos resíduos de cada iteração (CANAL, 2000).

2.4.4 GMRES - Resíduo Mínimo Generalizado

O GMRES (Resíduo Mínimo Generalizado) é um método iterativo para a solução de sistemas de equações onde as matrizes de coeficientes são não simétricas, desenvolvido por Saad e Schultz no ano de 1986 (SAAD, 1996).

O GMRES é aplicado na resolução de sistemas de equações lineares do tipo:

$$Ax = b \tag{2.23}$$

onde x é o vetor com as incógnitas, b é o vetor dos termos independentes e A é uma matriz não simétrica.

O GMRES constrói a solução gerando uma seqüência (base do subespaço) de vetores ortogonais, usando, por exemplo, o processo de Arnoldi. Mas esse processo, quando usa a ortogonalização de Gram-Schmidt, define vetores ortogonais somente em aritmética real. Na aritmética de ponto flutuante os vetores não são ortogonais. Para resolver esse problema substitui-se o algoritmo de Gram-Schmidt clássico por outros algoritmos que são matematicamente equivalentes, mas computacionalmente superiores, como o algoritmo de Gram-Schmidt modificado (NOBLE; DANIE, 1986).

(CATABRIGA, 1998) O método iterativo GMRES tem por objetivo básico minimizar a norma residual do sistema de equações (2.23) e possui como principal característica a construção de uma base ortonormal no subespaço de Krylov.

O método do GMRES obtem uma solução aproximada $x^0 + z$ a partir de uma solução inicial x^0 e de um vetor z do subespaço de Krylov. Esse vetor z do subespaço de Krylov deve ser de tal forma que a norma do resíduo $\|b - A(x^0 + z)\|$ seja mínima.

Uma desvantagem do GMRES é o fato de seus produtos matriz-vetor aumentarem linearmente com as iterações, e todos os vetores da base do subespaço de Krylov terem que ser armazenados, o que é um problema quando a dimensão m do subespaço cresce. Para esse problema, a solução mais empregada é a reinicialização do algoritmo, fixando-se a dimensão m do subespaço. Essa estratégia gera o GMRES(m).

A versão do algoritmo do GMRES(m), como encontrado em (CATABRIGA, 1998), pode ser visto na fig. 2.11.

Conforme já mencionado, o GMRES constrói uma base ortonormal B, onde todos os vetores da base são ortogonais, ou seja $\forall u, v \in B, u \neq v, u.v = 0$, e unitários, ou seja $\forall v \in B, \|v\| = 1$.

Nesse trabalho a base ortonormal no subespaço de Krylov foi obtida através do processo de Gram-Schimidt modificado, cujo algoritmo pode ser visto das linha 13 à 16 da fig. 2.11.

O processo de Gram-Schimidt modificado além de uma base no subespaço de Krylov gera uma matriz denominada Matriz de Hessemberg. A matriz de Hessemberg, que é referida como matriz H, tem o formato quase triangular, necessitando apenas a eliminação da diagonal abaixo da principal.

A transformação da matriz H em uma matriz triangular superior pode ser feita pelo algoritmo QR, usando em cada passo o processo de rotação de Givens. O algoritmo QR pode ser visto da linha 19 à 28 na fig. 2.11.

Após o processo de QR, o sistema $Hy = e$ pode ser resolvido por retrosubstituição. Então uma nova aproximação do vetor x é calculada com a soma do valor atual de x com o vetor obtido pela multiplicação da base V - gerada pelo processo de Gram-Schimidt modificado - pelo vetor y obtido na solução do sistema $Hy = e$.

Após a obtenção da nova aproximação de x é calculado um novo resíduo baseando-se nessa nova aproximação, a convergência da solução é verificada. Se essa não for atingida a solução mais recente será utilizada para uma nova iteração (CATABRIGA, 1998).

1. Defina ϵ_{tol} → tolerância
2. m → número de vetores de Krylov
3. δ_{max} → número máximo de iterações
4. Dados A, b, m, ϵ_{tol}, δ_{max} -(Inicializações)-
5. $\epsilon \leftarrow \epsilon_{tol}$
6. $x = 0$
7. Para $\delta = 1, ..., \delta_{max}$ faça **(Ciclo GMRES)**
8. $u_i = b - Ax$
9. $\bar{e} = \|u_i\|$
10. $u_i = \frac{u_i}{\|u_i\|}$
11. Para $i = 1, ..., m$ faça ++(Iteração do GMRES)++
12. $u_{i+1} = Au_i$
13. Para $j = 1, ..., i$ faça [[(Ortogonalização Modificada do Gram-Schimidt)]]
14. $\beta_{i+1,j} = (u_{i+1}, u_j)$
15. $u_{i+1} = u_{i+1} u_j$
16. Fim do j [[(Ortogonalização Modificada do Gram-Schimidt)]]
17. $h^{(i)} = \{\beta_{i+1,1}, ..., \beta_{i+1,i}, \|u_{i+1}\|\}^T$
18. $u_{i+1} = \frac{u_{i+1}}{\|u_{i+1}\|}$
19. Para $j = 1, ..., i-1$ faça ###(Rotação de Givens)###
20. $$\begin{Bmatrix} h_j^{(i)} \\ h_{j+1}^{(i)} \end{Bmatrix} = \begin{bmatrix} c_j & s_j \\ -s_j & c_j \end{bmatrix} \begin{Bmatrix} h_j^{(i)} \\ h_{j+1}^{(i)} \end{Bmatrix}$$
21. Fim do j
22. $r = \sqrt{(h_i^{(i)})^2 + (h_{i+1}^{(i)})^2}$
23. $c_i = \frac{h_i^{(i)}}{r}$
24. $s_i = \frac{h_{i+1}^{(i)}}{r}$
25. $h_i^{(i)} = r$
26. $h_{i+1}^{(i)} = 0$
27. $\bar{e}_{i+1} = -s_i \bar{e}_i$
28. $\bar{e}_i = c_i \bar{e}_i$ ###(Rotação de Givens)###
29. Teste de Convergência: Se $|\bar{e}_{i+1}| \leq \epsilon$ sai do laço i
30. Fim do i ++(Iteração do GMRES)++
31. Resolver y: $$\begin{bmatrix} h_1^{(1)} & ... & h_1^{(i-1)} & h_1^{(i)} \\ 0 & ... & : & : \\ : & ... & h_{i-1}^{(i-1)} & h_{i-1}^{(i)} \\ 0 & ... & 0 & h_i^{(i)} \end{bmatrix} \begin{Bmatrix} y_1 \\ : \\ y_{i-1} \\ y_i \end{Bmatrix} = \begin{Bmatrix} \bar{e}_1 \\ : \\ \bar{e}_{i-1} \\ \bar{e}_i \end{Bmatrix}$$
32. Calculo da Solução $x \leftarrow x + \sum_{j=1}^{i} y_j u_j$
33. Teste de Convergência: Se $|\bar{e}_{i+1}| \leq \epsilon$ sai *do* laço δ
34. Fim do δ **(Ciclo GMRES)**
35. x é a solução aproximada.

Figura 2.11: Algoritmo do GMRES(m)

2.4.5 Pré-condicionadores

Para acelerar a convergência de um método iterativo, pode-se pré-condicionar a matriz, já que a taxa de convergência de um método iterativo depende da distribuição dos autovalores da matriz dos coeficientes do sistema de equações.

A taxa pode ser melhorada transformando o sistema original $Ax = b$ em um outro na forma $MAx = Mb$, onde M é a matriz pré-condicionadora. O sistema de equações resultante é equivalente ao original, no sentido de que tem mesma solução, mas tem melhor taxa de convergência por controlar os autovalores pequenos ou grandes (CHOW, 1997), resultando em melhor número de condição. Para matrizes SDP, isso significa que os seus autovalores devem estar agrupados próximos à unidade. O caso de matrizes não simétricas é mais complicado, e não há resultados gerais sobre a convergência a partir da análise dos autovalores.

O método de pré-condicionamento que explora somente as informações da matriz dos coeficientes para acelerar a convergência da solução do sistema de equações é dito pré-condicionador algébrico. Aquele método que considera as características físicas ou geométricas é dito pré-condicionador físico ou geométrico.

Nos trabalhos realizados no GMCPAD, utilizou-se somente a primeira categoria, visto que são de propósito geral e podem ser aplicados a uma grande classe de problemas. São robustos, mas não ótimos como os obtidos por técnicas específicas, os quais, porém, só podem ser aplicados aos problemas que os originaram (BENZI; TUMA, 1999).

Uma classificação para os pré-condicionadores é aquela que considera o lado do pré-condicionamento, de modo que se tem os tipos: à esquerda $M^{-1}Ax = M^{-1}b$; à direita $AMM^{-1}x = b$; e à esquerda e à direita $M^{-1}AN^{-1}Nx = M^{-1}b$, onde M e N representam transformações simples ou complexas da matriz original A. O tipo do pré-condicionador a ser usado depende da escolha do método iterativo.

Em métodos de minimização dos resíduos, como o GMRES, o pré-condicionamento à direita pode ser mais efetivo pois, em aritmética exata, os resíduos para sistemas de equações pré-condicionados à direita são idênticos aos resíduos verdadeiros. O lado do pré-condicionamento é importante para problemas não simétricos mal condicionados.

O pré-condicionamento pode, também, ser considerado ser implícito ou ser explícito. No implícito obtém-se uma matriz M de modo a aproximar a matriz A sem a necessidade de calcular M^{-1} diretamente. Na abordagem explícita calcula-se a matriz M^{-1} de modo que se possa aproximar A^{-1} diretamente.

Em qualquer caso o pré-condicionador deve ser obtido rapidamente e de modo que $M^{-1}A$ seja aproximadamente igual a matriz identidade I, com mínimo de custo computacional. Ou seja, deve ser uma boa aproximação de A, ser fácil de construir e de aplicar. A primeira propriedade assegura boa taxa de convergência, enquanto a segunda assegura que cada iteração não seja custosa. Essas duas exigências são mutuamente contrastantes, sendo necessário um equilíbrio entre elas. Como a eficiência do pré-condicionador é, geralmente, inversamente proporcional ao tempo se sua construção (CATABRIGA, 1998), deve-se conciliar o ganho da convergência com o tempo de pré-condicionamento, já que o objetivo é diminuir o tempo total de execução do algoritmo.

Os pré-condicionadores utilizados, nos trabalhos realizados no GMCPAD, para acelerar as soluções dos sistemas de equações lineares são o polinomial e o de Schwarz, pois os Métodos de Decomposição de Domínio (com sobreposição) de Schwarz podem ser vistos, também, como pré-condicionadores (ALONSO; FATICA, 2002), já que a composição das soluções (aproximadas) locais podem ser empregados para gerar um pré-condicionador global e distribuído. Nesse caso, utilizaram-se as fatorações incompletas de Cholesky (IC) e a diagonal Cholesky (DIC) para obter a aproximação para as inversas locais.

2.5 Considerações Finais

Os *clusters* são arquiteturas que estão sendo cada vez mais utilizadas para computação paralela, não só em grandes centros de pesquisa, mas também em pequenas universidades. Esse fato se deve ao custo de um *cluster* não ser o mesmo de um supercomputador, embora seu poder de processamento possa ser em algumas aplicações semelhante.

A maior disponibilidade de arquiteturas paralelas, no caso *clusters*, vem ocasionando um aumento das pesquisas e do desenvolvimento de aplicações e de ferramentas para esse tipo de arquitetura.

Neste trabalho para a exploração do paralelismo na arquitetura de *cluster* são adotadas três ferramentas que são: MPI, DECK e Pthreads. O MPI e o DECK são duas bibliotecas de troca de mensagens para a exploração do paralelismo inter-nodos e a Pthreads é uma biblioteca de *threads*.

No uso das biblitecas de troca de mensagens, como MPI e DECK, conforme já mencionado, se houver falta de *buffer*, na comunicação entre múltiplos nodos, pode ocorrer *deadlock*. A falta ou não de *buffer* depende de alguns fatores como por exemplo do número de dados a serem transmitidos, da implementação da biblioteca adotada e da rede de interconexão adotada no *cluster*.

Visando eliminar a possibilidade de ocorrer *deadlock*, são adotadas estratégias de ordenação de mensagens para evitar o *deadlock*, baseadas no algoritmo "Maximal" e no algoritmo "Par_ímpar Adaptativo". No capítulo quatro essas estratégias são detalhadas.

Outra consideração referentes às bibliotecas de comunicação é o uso de primitivas não bloqueantes para sobrepor comunicação com processamento.

As bibliotecas de comunicação e de *threads* estão sendo adotadas no desenvolvimento de uma paralelização de métodos numéricos para resolução de sistemas lineares como os gerados no modelo desenvolvido no GMCPAD.

No que se refere ao modelo que está sendo desenvolvido no GMCPAD, os métodos de discretização que estão sendo adotado para a hidrodinâmica e para o transporte de substâncias 2D e 3D (DORNELES et al., 2002) (RIZZI, 2001) são o UPWIND e os tipo TVD entre outros.

A aplicação destes métodos resulta em sistemas de equações lineares esparsos e de grande porte, característica que motivou o desenvolvimento deste trabalho, que é a paralelizações dos métodos do subespaço de Krylov apresentados.

3 PARTICIONAMENTO DE DOMÍNIO

Na solução de um determinado sistema de equações em paralelo se faz necessário a divisão do domínio deste sistema em subdomínios, quantos forem desejados, para que estes possam ser resolvidos em paralelo.

Nesse capítulo inicialmente serão contextualisadas as principais estratégias de solução paralela. Em um segundo momento serão apresentados diferentes tipos de particionamentos que podem ser adotados e o processo de geração dos sistemas de equações.

3.1 Solução Paralela

Segundo (SMITH; BJORSTAD; GROPP, 1996) uma solução paralela pode ser obtida seguindo duas estratégias que são: decomposição de dados e decomposição de dominios. Estas estratégias podem formar uma estratégia híbrida onde um método de decomposição de domínios podem ter a função de pré-condicionador para um método de decomposição de dados. Neste trabalho está sendo adotada a estratégia de decomposição de dados.

Decomposição de dados é uma estratégia onde gera-se um único sistema de equações que é distribuído entre os processadores de modo que possa ser resolvido independentemente e em paralelo. Na literatura essa estratégia também é chamada de particionamento de dados.

Decomposição de domínio é uma estratégia onde empregam-se métodos e técnicas para acoplar as soluções dos subproblemas gerados a partir da decomposições do domínio global. Desse modo, a solução do sistema de equações global é obtida pela combinação das soluções desses subproblemas

3.2 Particionamento

Na paralelização de uma determinada aplicação, como por exemplo o modelo que simula hidrodinamica e transporte de massa em um domínio como um lago, faz-se necessário particionar este domínio.

O particionamento de um domínio tem por objetivo possibilitar que cada parte gerada possa ser resolvida em paralelo com as demais, diminuindo o tempo total de solução do domínio global.

No particionamento do domínio de uma aplicação, o algoritmo adotado irá influênciar consideravelmente nos demais passos necessários para a paralelização da aplicação. Essa influência decorre do fato de que o particionamento irá definir não somente a carga de processamento para cada subdomínio, mas também a estrutura das matrizes geradas a

partir destes subdomínio e a geração de fronteiras artificiais.

Em um bom particionamento, deve existir uma boa relação entre a carga de processamento e a carga de comunicação, sendo que esta boa relação deve ser alcançada em todos os subdomínios devido a possível necessidade de sincronismo entre os mesmos. Uma aplicação física requer sincronismo entre as diferentes soluções dos diferentes subdomínios.

No que se refere à estrutura da matriz gerada a partir dos subdomínios, a implementação paralela deve empregar uma estrutura de armazenamento adequada de modo que os dados possam ser acessados da maneira mais rápida possível.

Já as fronteiras artificiais influenciam nas estratégias de comunicação, pois o número de células artificiais e suas posições irão definir o número de comunicações e os processos que devem se comunicar.

Embora o particionamento seja um fator de grande relevância na paralelização de uma aplicação, não existe um algoritmo que defina um particionamento bom pois este é um problema NP-difícil (GAREY; JOHNSON, 1979). Na falta de um algoritmo ótimo, são empregados algoritmos que baseando-se em heurísticas encontram uma solução aproximada em um tempo hábil.

3.2.1 Algoritmos

Um problema de particionamento de domínio pode ser visto como um problema de particionamento de grafos (?) (SCHOEGEL; KARYPIS; KUMAR, 2001). Existem várias classes de algoritmos para particionamento de grafo, cada classe possui diferentes algoritmos, cada um com suas vantagens e desvantagens.

Neste trabalho estão sendo apresentados dois tipos de particionamentos, que são o particionamento em faixa e o particionamento em subdomínios não retangulares, juntamente com alguns algoritmos que os geram. Esses particionamentos são apresentados fazendo uso do lago Guaíba como estudo de caso. Uma ilustração do domínio discreto do lago Guaíba pode ser vista na fig. 3.1.

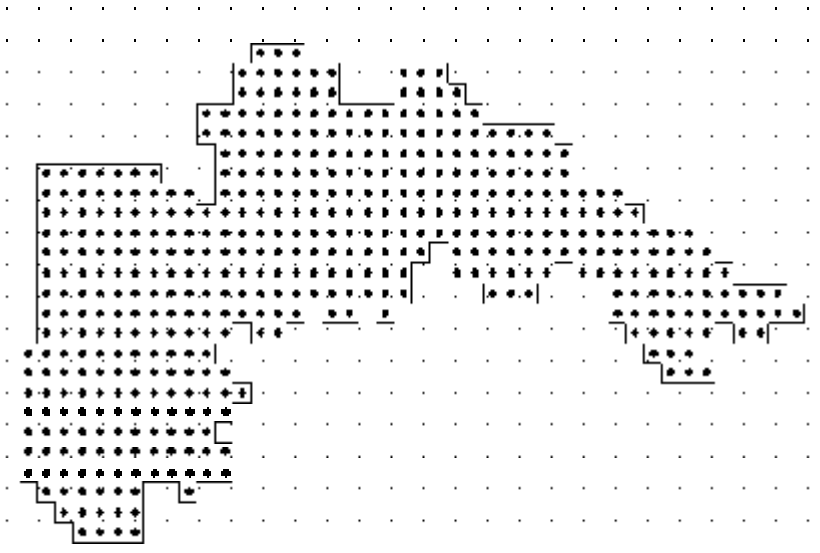


Figura 3.1: Domínio do lago Guaíba

Um estudo mais aprofundado sobre particionamento de grafos onde são apresentadas as diferentes classes de algoritmos, suas principais vantagens, desvantagens e

bibliotecas que as implementam (METIS, CHACO) pode ser encontrado em (SANTOS; DORNELES, 2001).

3.2.1.1 Particionamento em Faixas

Esta classe de algoritmos de particionamento é uma das mais simples e de menor custo computacional. Nessa classe de particionamento pode-se encontrar dois diferentes tipos de algoritmos que são: S-STRIP(*Straight Stripwise Partitioning*) e o STRIP(*Stripwise Partitioning*).

S-STRIP (*Straight Stripwise Partitioning*)

A idéia decorrente dessa abordagem consiste em dividir o domínio, através de faixas, em *n* subdomínios. Nesse particionamento, cada subdomínio resultante da divisão deve ter uma quantidade tal de linhas ou colunas (dependendo da direção das faixas) de modo que todos tenham uma carga computacional equivalente.

Embora as linhas ou colunas (dependendo da direção das faixas) sejam bem divididas entre os subdomínios, o balanceamento de carga obtido com este algoritmo não é bom, devido às "linhas de divisão" terem que ser retas.

Este tipo de particionamento tem a vantagem de diminuir o número de células artificiais e ,conseqüentemente, o número de comunicações. Embora o número de células artificiais seja menor, este algoritmo não diminui a extensão das fronteiras existentes, não diminuindo a carga de dados a serem trocados.

Outra vantagem deste algoritmo é que se a direção das faixas for a mesma direção adotada na numeração das células, os dados a serem trocados estão localizados de forma contínua.

A fig. 3.2 ilustra um exemplo de um particionamento em faixa com o algoritmo S-STRIP, onde o número de subdomínios resultante é 8.

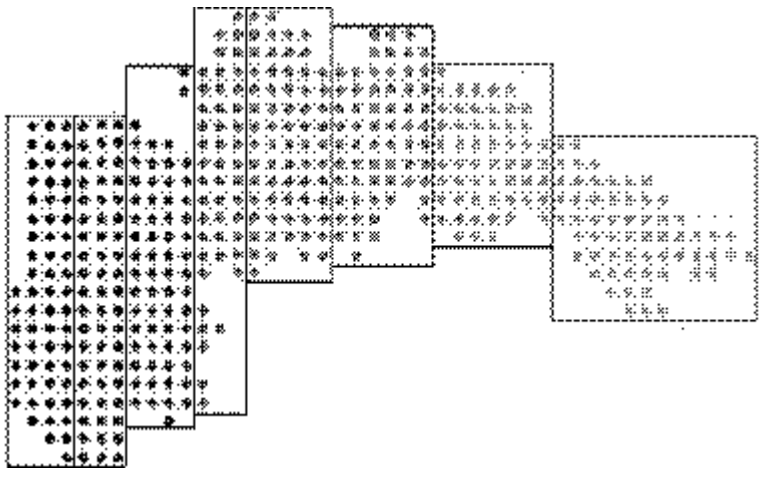


Figura 3.2: Particionamento em faixas não retangulares

Além das características já citadas, com este algoritmo existe a possibilidade de transformar os subdomínios não retangulares em retangulares. Esta transformação consiste em preencher os "espaços vazios" dos subdomínios com elementos de valor nulo.

A fig. 3.3 ilustra o resultado da transformação dos subdomínios ilustrados na fig. 3.2.

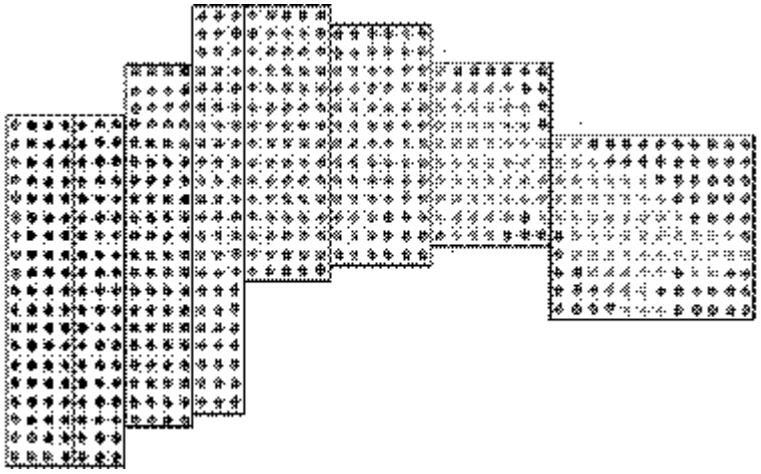


Figura 3.3: Particionamento em faixas retangulares

A transformação apresentada na fig. 3.3 facilita o armazenamento e as demais operações sobre a matriz resultante destes subdomínios, pois estas têm o formato estruturado em forma de diagonais. No entanto, além do custo de processamento desta transformação, ela possui a desvantagem de causar um aumento no tamanho da área de dados a serem computados, causando um maior custo computacional e de uso de memória, podendo introduzir erros adicionais.

STRIP (*Stripwise Partitioning*)

A sua concepção é muito semelhante ao do S-STRIP, tendo como unica diferença o fato de não exigir que as partições sejam em linhas retas.

Como vantagem sobre o algoritmo S-STRIP, esse permite um melhor balanceamento de carga entre os subdomínios fazendo com que o número de células entre subdomínios vizinhos difira no máximo de uma célula. Tem, no entanto, a desvantagem de não garantir que a enumeração das células das fronteiras sigam uma seqüência contínua e nem a possibilidade de transformação dos subdomínios não retangulares em retangulares.

A fig. 3.4 ilustra um exemplo de um particionamento em faixas com o algoritmo STRIP, onde o número de subdomínios resultante é 8.

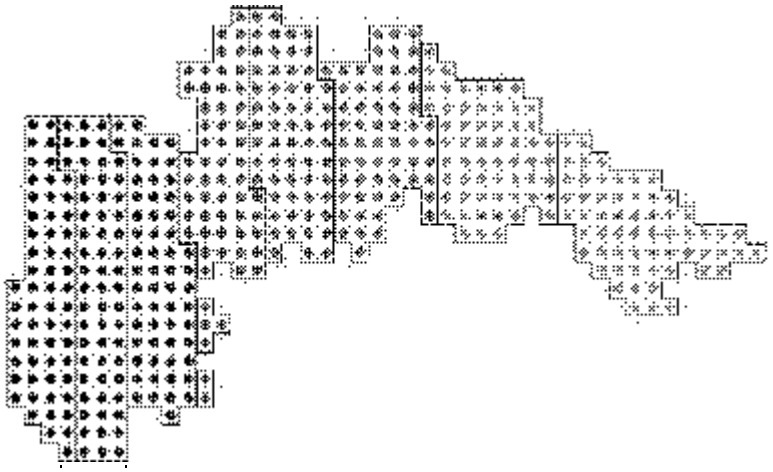


Figura 3.4: Particionamento em faixas não retas

3.2.1.2 Particionamento Irregular

Existem vários algoritmos que geram particionamentos em subdomínios não retangulares. Neste texto será apresentado o RCB (*Recursive Coordinate Bisection*). Esse algoritmo de particionamento aplicado ao domínio do lago Guaíba (estudo de caso apresentado), foi o que mostrou melhores resultados no balanceamento de carga e na minimização do número e tamanho das fronteiras geradas.

RCB (*Recursive Coordinate Bisection*)

O algoritmo RCB (*Recursive Coordenate Bisection*), também conhecido como CND (*Coordinate Nested Dissection*).

O RCB é uma evolução do ORB (*Orthogonal Recursive Bisection*), que exigia que cada corte fosse ortogonal ao corte anterior. Este algoritmo possui uma variante chamada de URB (*Unbalanced Recursive Bisection*) que, permitindo um pequeno desbalanceamento, apresenta uma melhor relação entre as áreas e perímetros dos subdomínios gerados, tendo porem um maior custo computacional.

O funcionamento do RCB consiste em fazer uma análise do domínio ou subdomínio e efetuar um corte na direção ortogonal à maior dimensão, assim reduzindo o tamanho das fronteiras. Esse corte, além de diminuir o tamanho das fronteiras, deve gerar dois subdomínios com carga aproximadamente equivalentes.

Na fig. 3.5 estão ilustradas as quatro fases de particionamento do domínio de 2 a 16 subdomínios através do algoritmo RCB. Nessa figura a ordem de geração do particionamento é da esquerda para a direita e de cima para baixo.

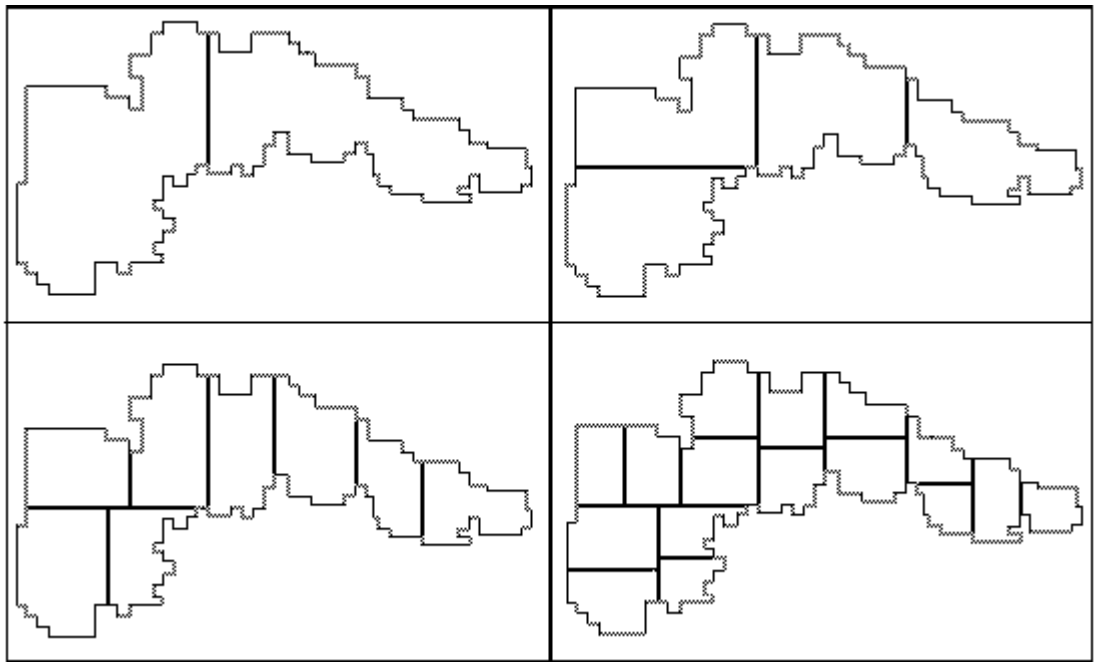


Figura 3.5: Fases da aplicação do algoritmo do RCB sobre o domínio do lago Guaíba

Uma ilustração mais ampliada do resultado final do particionamento apresentado na fig. 3.5 pode ser visto na fig. 3.6.

Segundo os testes desenvolvidos por (SANTOS; DORNELES, 2001), o algoritmo de particionamento RCB é suficientemente rápido para ser empregado sequencialmente para particionar domínios como aqueles usados no estudo de caso discutido neste trabalho.

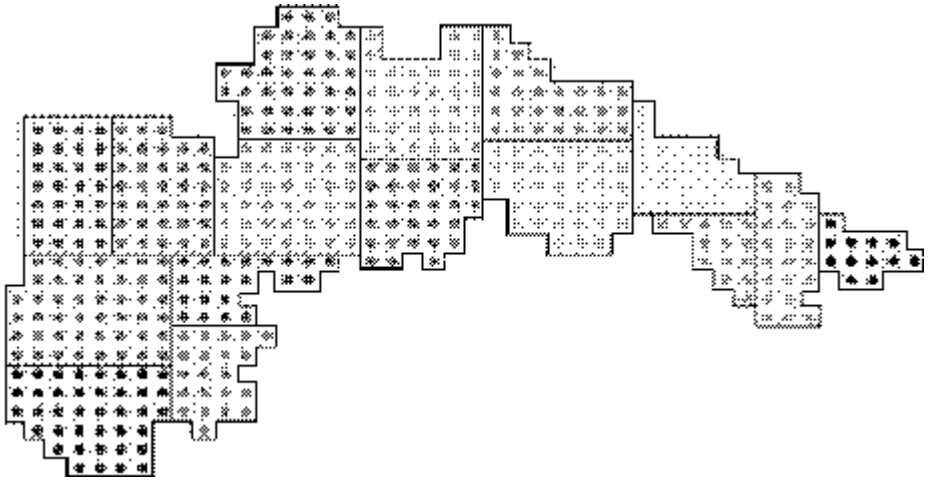


Figura 3.6: Particionamento em subdomínios irregulares

3.3 Geração de Sistemas de Equações

Em um modelo computacional, o processo de discretização é o responsável por gerar um sistema de equações a partir de um determinado domínio. Os algoritmos de solução paralelizados neste trabalho estão integrados ao modelo computacional desenvolvido no GMCPAD para simulação de hidrodinâmica e o transporte de substâncias em corpos de água.

As equações resultantes dos modelos referidos são no formato $\boldsymbol{Ax} = \boldsymbol{b}$, onde $\boldsymbol{x}$ é o vetor com as incógnitas, $\boldsymbol{b}$ é o vetor dos termos independentes, e $\boldsymbol{A}$ é a matriz dos coeficientes.

Existem dois momentos em que o processo de geração da matriz pode ser aplicado: antes do particionamento gerando um sistema de equações global a partir de um domínio global; depois do particionamento gerando sub-sistemas de equações a partir de subdomínios. No caso da discretização global, a geração é feita sequencialmente e o sistema de equações resultante é particionado entre os diversos processadores.

No entanto, se a discretização for aplicada aos subdomínios já distribuídos, então esse processo está sendo feito em paralelo e as partes do sistema de equações resultante já estarão distribuídas entre os processadores.

O nível de dependência de dados entre as células da malha após a discretização empregada para obter as equações discretas pode ser representado usando o estêncil de 5-pontos, visto na fig. 3.7.

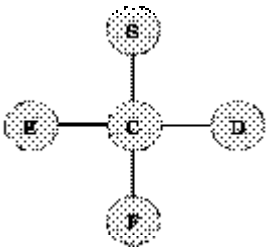


Figura 3.7: Discretização em 5 pontos (molécula computacional)

Com a discretização usando o estêncil de 5-pontos, cada célula depende da sua posição e dos quatro vizinhos. Ou seja, cada célula no domínio depende de seu valor central(C) e do valor de seus 4 vizinhos, que são: esquerdo (E), direito (D), superior (S) e inferior (F). Existem discretizações que empregam mais pontos, como por exemplo 9, onde os algoritmos aqui paralelizados também podem ser adotados.

O estêncil computacional define a posição e valor de cada elemento na matriz e a estrutura e regularidade da mesma.

3.3.1 Tipos de Matrizes

Usando-se uma malha cartesiana regular as matrizes geradas no processo de discretização são necessariamente estruturadas, porém podem ser regulares ou irregulares.

A regularidade, ou não, das matrizes depende da "geometria" dos domínios ou subdomínios e do algoritmo de particionamento adotado. A importância do formato do domínio ou dos subdomínios vem do fato de que é esse formato que define a localidade dos elementos não nulos na matriz.

No processo de geração da matriz, é a enumeração das células e de suas vizinhas que definirão quais colunas na matriz possuem valores não nulos. Nos casos em que se usa estêncil de 5-pontos, é a enumeração da célula e de suas quatro vizinhas mais próximas (esquerda (E), direita (D), superior (S) e inferior (F)) que definirão quais as posições que possuirão valores não nulos.

3.3.1.1 Matrizes Estruturadas Regulares

Entende-se por matrizes estruturadas regulares aquelas matrizes cujos elementos estão posicionados em forma de diagonais. Este tipo de matriz é gerada a partir de subdomínios retangulares. Essa característica, de os subdomínios retangulares gerarem matrizes neste formato, deve-se ao fato que a diferença entre a enumeração da célula central e suas quatro vizinhas serem equivalentes para todas as células deste subdomínio. Um exemplo de uma matriz estruturada regular pode ser visto na fig. 3.8, onde o subdomínio apresentado segue a enumeração tipo "ordenação natural"..

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

$$=\begin{Bmatrix}
C & D & 0 & 0 & S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
E & C & D & 0 & 0 & S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & E & C & D & 0 & 0 & S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & E & C & D & 0 & 0 & S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
F & 0 & 0 & E & C & D & 0 & 0 & S & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & F & 0 & 0 & E & C & D & 0 & 0 & S & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & F & 0 & 0 & E & C & D & 0 & 0 & S & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & F & 0 & 0 & E & C & D & 0 & 0 & S & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & F & 0 & 0 & E & C & D & 0 & 0 & S & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & F & 0 & 0 & E & C & D & 0 & 0 & S & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & F & 0 & 0 & E & C & D & 0 & 0 & S & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & F & 0 & 0 & E & C & D & 0 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & F & 0 & 0 & E & C & D & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & F & 0 & 0 & E & C & D & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & F & 0 & 0 & E & C & D \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & F & 0 & 0 & E & C
\end{Bmatrix}$$

Figura 3.8: Exemplo de uma matriz estruturada regular

Na fig. 3.8, o domínio contendo 16 células gerou uma matriz contendo 16 linhas e 16 colunas. Nessa matriz cada célula do domínio gerou uma linha e nesta linha um elemento

não nulo posicionado na coluna equivalente a enumeração da célula, este elemento está indicado pela cordenada "C"(central).

Além do elemento central, cada linha pode possuir no máximo mais quatro elementos não nulos, que representam os quatro vizinhos da célula que gerou o elemento central. O posicionamento da coluna destes quatro elementos é equivalente a enumeração dos quatro vizinhos que os geram e é indicado pelas cordenadas "E"(esquerda), "D"(direita), "S"(superior) e "F"(inferior).

3.3.1.2 Matrizes Estruturadas Irregulares

Matrizes estruturadas irregulares, que são geradas a partir de domínios não retangulares, têm a característica de não possuirem todos seus elementos distribuídos em forma de diagonais.

A distribuição dos elementos de forma irregular deve-se ao fato que a diferença entre a enumeração da célula central e suas quatro vizinhas não serem equivalentes para todas as células deste subdomínio.

As matrizes estruturadas irregulares exigem um formato de armazenamento mais sofisticado que o exigido pelas matrizes regulares. Um exemplo de uma matriz estruturada irregular e do domínio irregular a partir do qual ela foi gerada pode ser visto na fig. 3.9.

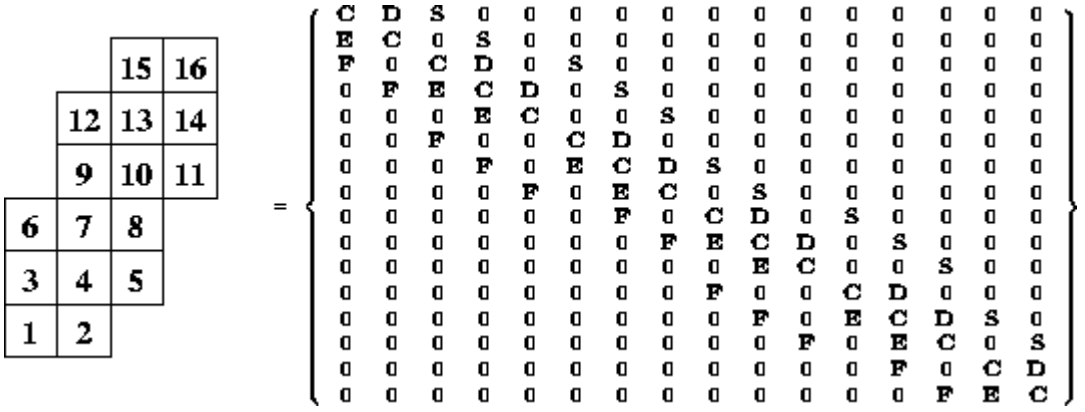


Figura 3.9: Exemplo de uma matriz estruturada irregular

3.3.2 Geração de Matrizes Após Particionamento

Na geração das matrizes locais, após o particionamento do domínio em subdomínios, cada célula dos subdomínios irá gerar uma linha nesta matriz. Cada linha da matriz corresponde a uma célula do domínio e os coeficientes da linha são definidos a partir da enumeração das células vizinhas.

Para garantir que as linhas das matrizes locais geradas pelas células centrais próximas às fronteiras com os outros subdomínios tenham os valores das células vizinhas necessários para o cálculo dos valores nodais, devido a sua dependência de dados, na distribuição dos subdomínios entre os processadores juntamente com os subdomínios devem ser enviadas todas as células das fronteira.

Na fig. 3.10 está ilustrado o particionamento de um domínio em dois subdomínios com as dependências de informações entre estes.

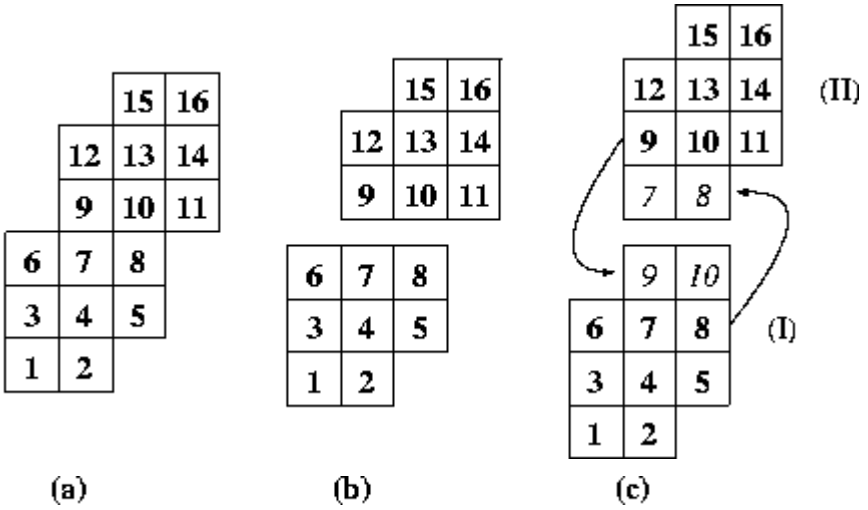


Figura 3.10: Exemplo para particionamento do domínio da fig. 3.9

Ao conjunto de células das fronteiras dos subdomínios, dá-se o nome de estêncil. Essas células podem ser vistas como sendo uma sobreposição de um subdomínio no outro, onde elas exercem a função de suprir a dependência de dados de um subdomínio no outro, devido a molécula computacional de 5-pontos apresentada na 3.7.

Na fig. 3.10 (a), a molécula computacional da célula 10 é formado pelas células 9, 13, 11 e 8. Na parte (b) da fig. 3.10 pode-se observar que após o particionamento a célula 8 não pertence ao mesmo subdomínio que a célula 10, ocasionando uma depêndencia na geração da matriz. Na parte (c) da fig. 3.10 as células em itálico representam os estênceis que são sobreposições dos domínios vizinhos utilizadas para suprir a dependência de dados.

Existem dois tipos diferentes de estênceis que são: estênceis internos e estênceis externos. Em dois subdomínios vizinhos as células de estêncil interno para um subdomínio são células de estêncil externo para o outro e vice versa, ou seja, as células de estêncil interno, internas a um subdomínio, são enviadas para o subdomínio vizinho, onde serão estêncil externo. Na fig. 3.10 as células "9 e 10"formam um estêncil interno no subdomínio "(II)"e um estêncil externo no subdomínio "(I)".

O número de estênceis que um subdomínio possui é equivalente ao número de vizinhos deste, e uma mesma célula pode pertencer a dois ou mais estênceis ao mesmo tempo.

Conforme já mencionado, as células de estênceis externos não geram linhas na montagem de uma matriz, geram apenas elementos nas colunas nas linhas geradas por suas vizinhas.

As figs. 3.12 e 3.11 ilustram a representação de uma "matrizes" a partir de dois subdomínios não retangular cujo particionamento foi apresentado na fig. 3.10.

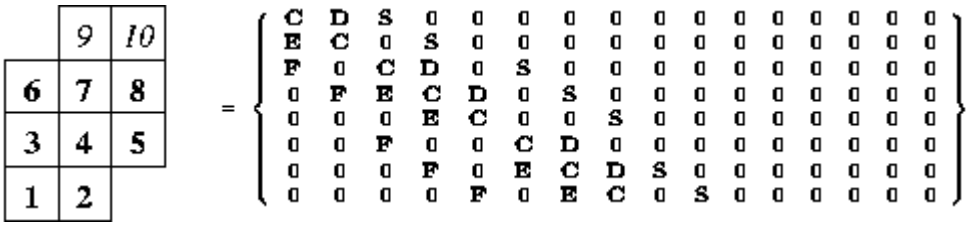


Figura 3.11: Exemplo da matriz (I) da fig. 3.10 (c)

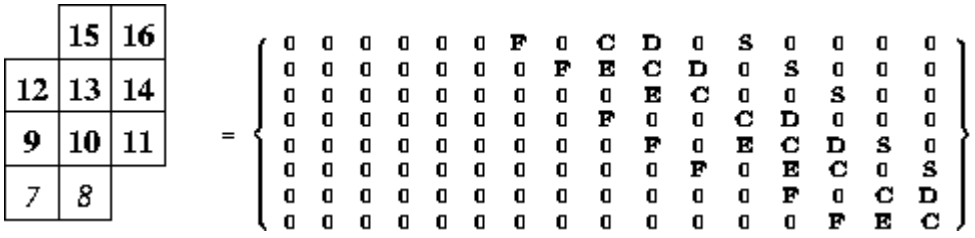


Figura 3.12: Exemplo da matriz (II) da fig. 3.10 (c)

Na fig. 3.12 a matriz gerada possui 8 linhas devido ao número de células internas do subdomínio, e 16 colunas devido ao número total de células no domínio global. O número de colunas é equivalente ao número de células do domínio global pois a enumeração das células adotada é global. Nessa matriz a primeira linha é gerada pela célula 9 a segunda pela célula 10 e assim sucessivamente.

3.3.3 Formato de Armazenamento

O formato de armazenamento de uma matriz influencia no desempenho de uma aplicação, pois ele deve proporcionar o acesso aos dados da maneira mais rápida e direta possível, com o mínimo uso de espaço em disco e em memória.

A escolha pelo uso de um ou outro formato depende da maneira como os dados válidos estão distribuídos na matriz e também da ordem em que os acessos a estes serão feitos.

3.3.3.1 Matrizes Estruturadas Regulares

Para o armazenamento de matrizes estruturadas regulares, um formato de armazenamento muito adotado é o baseado em vetores, uma ilustração desse formato pode ser vista na fig.3.13.

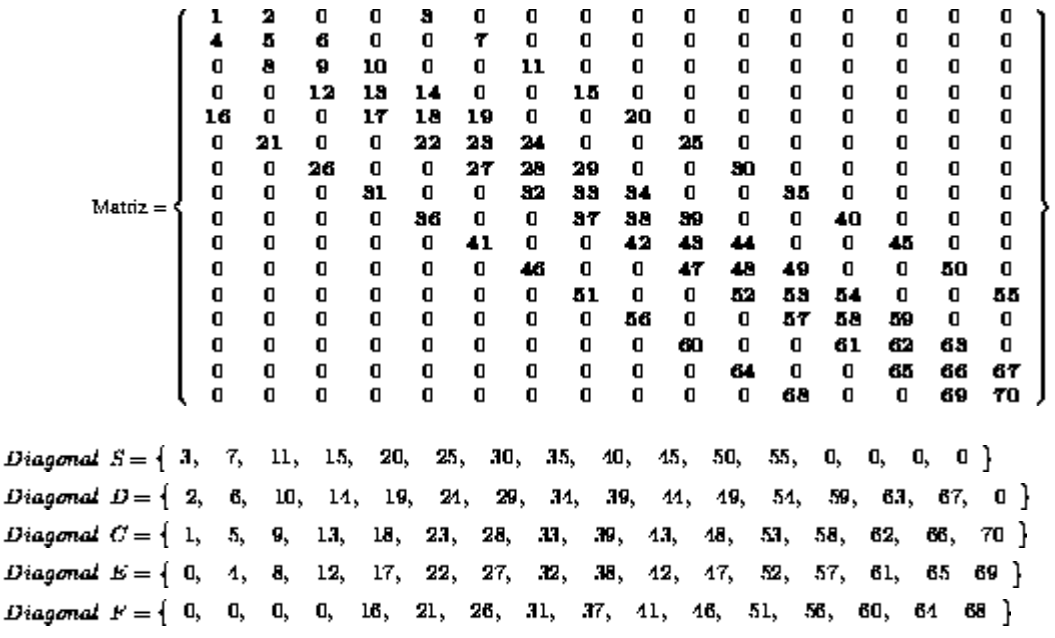


Figura 3.13: Armazenamento de uma matriz estruturada regular

Na fig. 3.13, que vem a ser uma representação do armazenamento do domínio

apresentado na fig. 3.8. Este formato é suficientemente bom para armazenas os elementos não nulos em sistemas de equações gerados pela discretização de domínior retangulares.

3.3.3.2 Matrizes Estruturadas Irregulares

Diferentemente das matrizes estruturadas regulares, para o armazenamento de uma matriz estruturada irregular é necessário uma estrutura de dados mais sofisticada, isso ocorre devido ao dados estarem dispostos em colunas de maneira não uniforme.

Segundo a literatura (JUDICE; PATRICIO, 1996), uma das estruturas mais usadas atualmente é a chamada de CSR (*Compressed Storage Rows*), que consiste no armazenamento dos dados e também de vetores de índices.

A estrutura CSR é baseada na criação de 3 vetores, um vetor contendo os dados (valores não nulos), o qual deve ter tipo equivalente ao tipo dos dados, um segundo vetor, do tipo inteiro, contendo as colunas onde os dados estão posicionados na matriz e um terceiro vetor, do tipo inteiro, contendo n + 1 posições, onde é armazenado o número de elementos da matriz até a linha anterior.

A identificação a qual linha cada valor armazenado no vetor de dados pertence é baseada neste terceiro vetor, onde a primeira posição armazena o valor zero, a segunda posição o total de elementos até o fim da primeira linha, a terceira posição é o total de elementos até o fim da segunda linha e assim sucessivamente.

Na fig. 3.14 está ilustrada uma matriz estruturada irregular com seu armazenamento em formato CSR, nesta matriz a posição dos dados é equivalente a já apresentada na fig. 3.9.

$$\text{Matriz} = \begin{Bmatrix}
1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 5 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 8 & 9 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 11 & 12 & 13 & 14 & 0 & 15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 16 & 17 & 0 & 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 19 & 0 & 0 & 20 & 21 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 22 & 0 & 23 & 24 & 25 & 26 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 27 & 0 & 28 & 29 & 0 & 30 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 31 & 0 & 32 & 33 & 0 & 34 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 35 & 36 & 37 & 38 & 0 & 39 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 40 & 41 & 0 & 0 & 42 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 43 & 0 & 0 & 44 & 45 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 46 & 0 & 47 & 48 & 49 & 50 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 51 & 0 & 52 & 53 & 0 & 54 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 55 & 0 & 56 & 57 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 58 & 59 & 60
\end{Bmatrix}$$

$$\textit{Valores} = \begin{Bmatrix}
1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10, & 11, & 12, & 13, & 14, & 15 \\
16, & 17, & 18, & 19, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & 28, & 29, & 30 \\
31, & 32, & 33, & 34, & 35, & 36, & 37, & 38, & 39, & 40, & 41, & 42, & 43, & 44, & 45 \\
46, & 47, & 48, & 49, & 50, & 51, & 52, & 53, & 54, & 55, & 56, & 57, & 58, & 59, & 60
\end{Bmatrix}$$

$$\textit{Colunas} = \begin{Bmatrix}
0, & 1, & 2, & 0, & 1, & 3, & 0, & 2, & 3, & 5, & 1, & 2, & 3, & 4, & 6 \\
3, & 4, & 7, & 2, & 5, & 6, & 3, & 5, & 6, & 7, & 8, & 4, & 6, & 7, & 9 \\
6, & 8, & 9, & 11, & 7, & 8, & 9, & 10, & 12, & 9, & 10, & 13, & 8, & 11, & 12 \\
9, & 11, & 12, & 13, & 14, & 10, & 12, & 13, & 15, & 12, & 14, & 15, & 13, & 14, & 15
\end{Bmatrix}$$

$$\textit{Sub Totais} = \{\ 0,\ 3,\ 6,\ 10,\ 15,\ 18,\ 21,\ 26,\ 30,\ 34,\ 39,\ 42,\ 45,\ 50,\ 54,\ 57,\ 60\ \}$$

Figura 3.14: Armazenamento de uma matriz irregular

Pode-se alterar o CSR colocando a diagonal principal em um vetor separado. Essa alteração diminui o total de armazenamentos no vetor de índices e também o total de acesso a este, o que pode resultar em um ganho de desempenho. Além disso, o armazenamento da diagonal principal em separado permite a implementação de pré-condicionadores de maneira mais fácil e com execução mais rápida.

3.4 Considerações Finais

Para que um modelo, como o de hidrodinâmica e transporte 2D e 3D, desenvolvido no GMCPAD, possa ser aplicado sobre vários domínios, de diferentes geometrias, este tem que ser flexível para permitir diferentes tipos de particionamento.

A importância de um bom particionamento decorre do fato que o particionamento do domínio pode influenciar na carga de processamento e de comunicações, e consequentemente no tempo de solução do problema a ser resolvido.

Além de um bom particionamento, uma aplicação paralela deve objetivar o menor custo de processamento seqüencial possível. Visando o máximo de paralelismo, uma alternativa é efetuar as operações de geração de matrizes paralelamente em subdomínios já distribuídos.

Conforme já mencionado, um algoritmo de particionamento robusto pode gerar subdomínios não retangulares e conseqüentemente esses irão gerar matrizes irregulares.

Para se obter um bom desempenho em uma aplicação, as operações sobre os dados devem ser otimizadas ao máximo e, para isso, os dados devem estar armazenados de modo a proporcionar a otimização destas operações.

Visando o máximo de otimização, neste trabalho está sendo adotado um formato de armazenamento que distingue dados necessárias para comunicação e processamento e dados necessários apenas para processamento. Este formato de armazenamento será apresentado no capítulo seis.

4 PARALELIZAÇÕES DOS MÉTODOS DO SUBESPAÇO DE KRYLOV

Na solução de um sistema de equações lineares através de uma abordagem paralela, os dados devem estar distribuídos entre os processos para que ocorra o paralelismo. Se esse sistema já tiver sido gerado em paralelo esses dados já estarão distribuídos, caso contrário a distribuição deve ser feita.

Conforme já mencionado, a abordagem adotada no modelo em que este trabalho está inserido é baseada na geração dos sistemas de equações paralelamente pelos processos. Essa abordagem além de explorar o processamento paralelo também na geração dos sistemas, elimina a necessidade de distribuição dos dado a cada nova etapa de uma simulação.

Nessa geração paralela, as partes dos sistemas resultantes dos estênceis externos irão gerar dependência de dados na solução. Essa dependência de dados torna necessária a comunicação entre os processos vizinhos durante a solução do sistema.

Baseando-se nessa necessidade de comunicação e visando proporcionar um melhor desempenho na execução dos métodos paralelizados, está sendo adotado um formato de armazenamento que identifica as informações a serem enviadas e recebidas de maneira mais direta.

Outro critério referente a comunicação entre os processos é a rede de interconexão, pois é esta que irá definir o tamanho do *buffer* disponível para as comunicações entre os processos. Com o objetivo de proporcionar uma maior portabilidade das paralelizações aqui apresentadas, foram desenvolvidas duas estratégias de ordenação de mensagens que visam evitar a ocorrência de *deadlock* mesmo na falta de *buffer*.

Neste capítulo são apresentados o formato de armazenamento desenvolvido e adotado e as duas estratégias de ordenação de mensagens desenvolvidas.

4.1 Estrutura de Dados para Domínios Não Retangulares

Conforme discutido na secão 3.2, uma estrutura de dados que vise armazenar uma matriz de maneira otimizada e que permita o acesso aos dados dessa de maneira menos custosa, deve levar em consideração o posicionamento dos elementos não nulos.

A posição e quantidade dos elementos não nulos em uma matriz depende do método de discretização adotado e do formato do domínio ou subdomínio a partir do qual a matriz está sendo gerada.

Além disso, conforme apresentado no na seção 3.1, o formato de um subdomínio gerado a partir de um algoritmo de particionamento depende não somente do domínio particionado mas também do algoritmo de particionamento adotado.

O algoritmo para particionamento de domínios não retangulares que mostrou melhor resultado para o domínio apresentado como estudo de caso foi o RCB, cujo exemplo do uso pode ser visto na fig. 4.1.

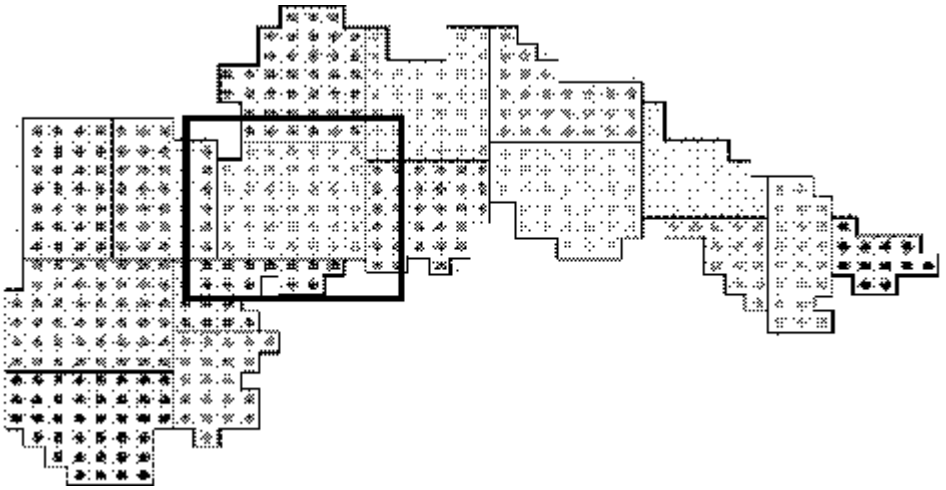


Figura 4.1: Particionamento em subdomínios irregulares destacando um subdomínio

Seguindo a abordagem adotada, após o particionamento do domínio o sistema de equações é gerado baseado na enumeração conforme ordenação natural das células dos subdomínio. Um exemplo de um subdomínio com seus vizinhos contendo as células enumeradas pode ser encontrado na fig. 4.2, que é uma ampliação da região em destaque na 4.1. A ilustração de todo o domínio do qual este subdomínio faz parte pode ser encontrada no anexo A3.

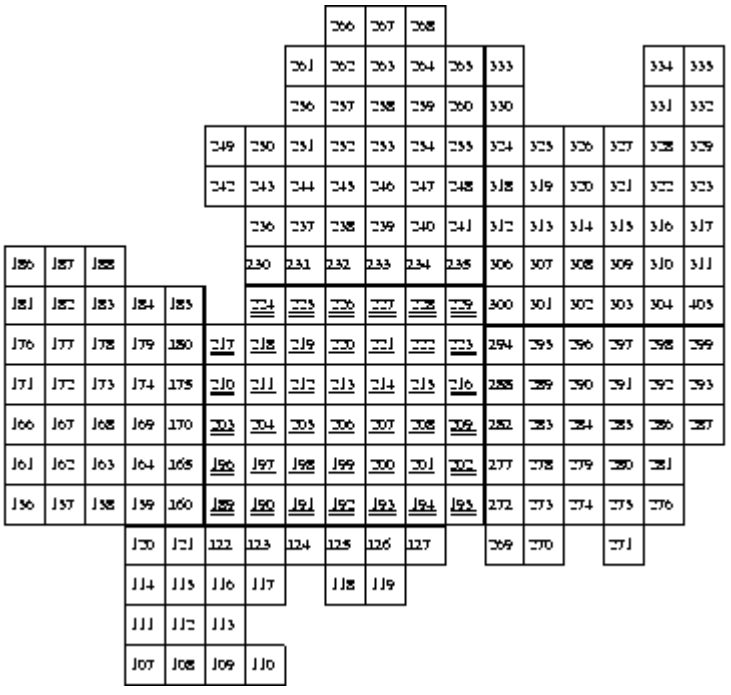


Figura 4.2: Enumeração de uma matriz irregular

Na fig. 4.2, as células internas do subdomínio estão sublinhadas, essas células geram linhas na matriz. São denotam, respectivamente, com (simples) para células internas e não adjacentes ao domínio computacional, e com (duplo) para células internas e adjacentes ao domínio computacional.

As células cuja enumeração é negritada são células externas adjacentes a este subdomínio e irão gerar valores não nulos, que são estênceis discutidos na seção 3.2.2. Na operação matriz-vetor, o resultado da multiplicação do vetor x pelos elementos não nulos gerados por estas células serão somados às linhas das células internas adjacentes que estão destacadas na forma sublinhada(duplo).

A enumeração das células segue a "ordem natural", ou seja, são numeradas da esquerda para a direita e de cima para baixo. Note-se na fig. 4.2 que a numeração das células dos subdomínios seguem o critério de enumeração global, porem de modo seqüêncial dentro de cada subdomínio.

Devido ao fato que os subdomínios não tem o formato retangular, as matrizes geradas a partir deles terão seus elementos distribuídos de maneira irregular. Essas matrizes, geradas de maneira já distribuída, serão usadas pelos algoritmos paralelizados em uma operação de produto matriz-vetor para obter a solução do sistema global.

Para a operação de produto matriz-vetor, cada processo possui um vetor local de tamanho equivalente ao número de células internas do subdomínio deste processo. Os índices das posições deste vetor são equivalentes às enumerações destas células internas.

Na multiplicação dos elementos não nulos gerados a partir das células dos estenceis externos, adjacentes ao subdomínio, não existem elementos no vetor local. Essa inexistencia de elementos ocorre devido ao fato destes elementos estarem armazenados nos vetores locais dos outros processos, processos nos quais estas células são internas.

A fig. 4.3, apresenta a matriz gerada pelo subdomínio (II) ilustrado na fig. 3.10. Além desta matriz esta figura apresenta dois vetores. O vetor maior, de X_9 até X_{16}, pertence ao processo local e tem índices equivalentes as células internas do subdomínio que gerou a matriz. O vetor menor, de X_7 até X_8, pertence ao subdomínio vizinho e tem índice equivalente a celulas internas deste vizinho.

$$
\begin{array}{c}
\\
\left\{\begin{array}{cccccccccccccccc}
00 & 00 & 00 & 00 & 00 & 00 & F_7 & 00 & C_9 & D_{10} & 00 & S_{11} & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & F_8 & E_9 & C_{10} & D_{11} & 00 & S_{13} & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & E_{10} & C_{11} & 00 & 00 & S_{14} & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_9 & 00 & 00 & C_{13} & D_{13} & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_{10} & 00 & E_{12} & C_{13} & D_{14} & S_{15} & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_{11} & 00 & E_{13} & C_{14} & 00 & S_{16} \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_{13} & 00 & C_{15} & D_{16} \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_{14} & E_{15} & C_{16}
\end{array}\right\}
\begin{array}{c}
\\
\left\{\begin{array}{c} X_7 \\ X_8 \end{array}\right\} \\
\left\{\begin{array}{c} X_9 \\ X_{10} \\ X_{11} \\ X_{12} \\ X_{13} \\ X_{14} \\ X_{15} \\ X_{16} \end{array}\right\}
\end{array}
$$

Figura 4.3: Matriz do subdomínio (II) da fig. 3.10

Para resolver este problema de disposição de dados, os processos devem enviar para seus vizinhos os elementos de seu vetor com enumeração equivalente a seus estênceis internos (que serão externos para seus vizinhos) e receber destes vizinhos os elementos equivalentes a seus estenceis externos (que são internos para seus vizinhos).

A localização dos dados a serem enviados no vetor é obtida através da enumeração dos estenceis internos. Para isso, cada processo deve ter uma lista contendo a identificação de seus estênceis internos.

Para armazenar os dados recebidos em posições com índices "equivalentes"a suas enumerações no subdomínio onde estes são estenceis internos, cada processo deveria

alocar um vetor com tamanho "equivalente"ao tamanho do domínio global, o que acarretaria um gasto desnecessário de memória. Por outro lado, para armazenar os dados seqüencialmente, ignorando sua enumeração, deve-se aplicar uma estratégia de indexar ou de busca dos mesmos, o que acarreta um custo de processamento.

Para resolver estes problemas, desenvolveu-se uma estrutura de armazenamento que proporciona o armazenamento e o acesso aos dados dos estênceis externos e dos elementos dos vetores a serem enviados e recebidos, de modo a não haver um custo desnecessário de memória nem de pesquisa.

Optou-se então por dividir os dados em duas estruturas de armazenamento separadas: um formato CSR para os dados do núcleo da matriz, e um formato distinto para cada fronteira entre os vizinhos.

Na montagem da parte interna da matriz no formato CSR considera-se como se o subdomínio não tivesse estenceis, ou seja, os elementos dos estênceis externos não entram na matriz conforme pode ser visto na fig. 4.4. O subdomínio é armazenado isoladamente dos demais subdomínios vizinhos.

$$\begin{Bmatrix}
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & C_0 & D_{10} & 00 & S_{11} & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & E_0 & C_{10} & D_{11} & 00 & S_{13} & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & E_{10} & C_{11} & 00 & 00 & S_{14} & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_0 & 00 & 00 & C_{12} & D_{13} & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_{10} & 00 & E_{12} & C_{13} & D_{14} & S_{15} & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_{11} & 00 & E_{13} & C_{14} & 00 & S_{16} \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_{13} & 00 & C_{15} & D_{16} \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & F_{14} & E_{15} & C_{16}
\end{Bmatrix}$$

Figura 4.4: CSR (parte interna do subdomínio (II) da fig. 3.10)

Já para armazenar os elementos dos estenceis considerou-se as características da operação matriz-vetor, onde os elementos do estêncil externo, que geram apenas colunas na matriz, devem ser multiplicados pelos elementos do vetor nas posições com índices equivalentes à enumeração destes estenceis externos, sendo o resultado adicionado ao vetor de resultados na posição com índice equivalente à enumeração da célula vizinha, que formam o estêncil interno.

Então considerando-se um estêncil de apenas uma célula criou-se um formato de armazenamento para cada vizinho baseada em três vetores: o primeiro contendo o valor do estêncil externo, o segundo a enumeração das células do estêncil externo e o terceiro contendo a enumeração das células do estêncil interno.

Uma ilustração destes vetores para o subdomínio (II) representado na fig. 3.10 pode ser visto na fig. 4.5.

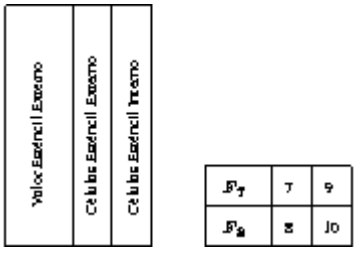


Figura 4.5: Parte externa do subdomínio II da fig. 3.10

Note-se que, para facilitar a operação de multiplicação do vetor pelos elementos do estêncil externo, deve-se enviar a enumeração do estêncil externo para o processo onde este é estêncil interno. Assim, quando este processo enviar os dados de seu estêncil

interno, estes dados já estarão ordenados de acordo com a ordem existente no vetor de valores do estêncil externo já armazenado no processo receptor.

Baseando-se neste formato de armazenamento a operação de multiplicação matriz-vetor, que é o passo computacional mais caro na solução de um sistema de equações, pode ser descrita em dois passos (SAAD, 2002):

1. O primeiro passo consiste em efetuar a operação de multiplicação matriz por vetor nas células internas da matriz armazenadas no formato CSR. Uma ilustração pode ser vista na fig. 4.6;

2. O segundo passo consiste em efetuar a operação de multiplicação matriz por vetor dos elementos dos estenceis e complementar os resultados obtidos no passo anterior. Uma ilustração pode ser vista na fig. 4.7.

$$\begin{bmatrix} T_9 \\ T_{10} \\ T_{11} \\ T_{12} \\ T_{13} \\ T_{14} \\ T_{15} \\ T_{16} \end{bmatrix} = \begin{bmatrix} X_9*C_9+X_{10}*D_{10}+X_{11}*S_{11} \\ X_9*E_9+X_{10}*C_{10}+X_{11}*D_{11}+X_{13}*S_{13} \\ X_{10}*E_{10}+X_{11}*C_{11}+X_{14}*S_{14} \\ X_9*F_9+X_{12}*C_{12}+X_{13}*D_{13} \\ X_{10}*F_{10}+X_{12}*E_{12}+X_{13}*C_{13}+X_{14}*D_{14}+X_{15}*S_{15} \\ X_{11}*F_{11}+X_{13}*E_{13}+X_{14}*C_{14}+X_{16}*S_{16} \\ X_{13}*F_{13}+X_{15}*C_{15}+X_{16}*D_{16} \\ X_{14}*F_{14}+X_{15}*E_{15}+X_{16}*C_{16} \end{bmatrix}$$

Figura 4.6: Parte interna do produto matriz-vetor da matriz da fig. 4.4

$$\begin{bmatrix} T_9 \\ T_{10} \end{bmatrix} = \begin{bmatrix} T_9 \\ T_{10} \end{bmatrix} + \begin{bmatrix} X_7*F_7 \\ X_8*F_8 \end{bmatrix}$$

Figura 4.7: Parte externa do produto matriz-vetor da matriz da fig. 4.4

Uma ilustração destes vetores para o subdomínio destacado e representado na fig. 4.8 pode ser visto na fig. 4.9.

		230	231	232	233	234	235	
		224	225	226	227	228	229	300
180	217	218	219	220	221	222	223	294
175	210	211	212	213	214	215	216	288
170	203	204	205	206	207	208	209	282
165	196	197	198	199	200	201	202	277
160	189	190	191	192	193	194	195	272
	122	123	124	125	126	127		

Figura 4.8: Enumeração de uma matriz irregular

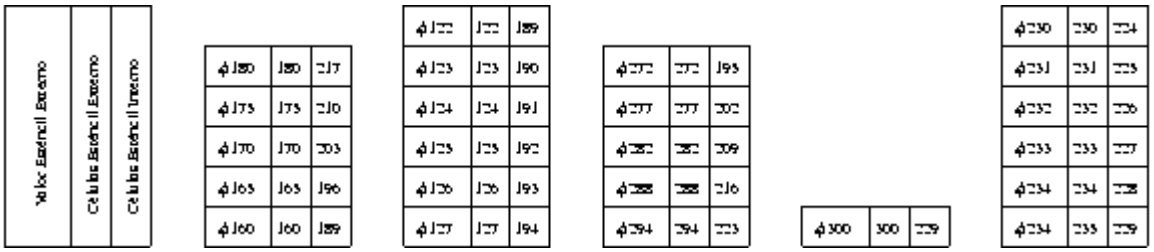


Figura 4.9: Armazenamento e enumeração dos dados dos estenceis

4.2 Ordenação de Mensagens

As paralelizações desenvolvidas neste trabalho são adequadas para solucionar sistemas de equações resultantes da discretização de subdomínios irregulares, os quais podem ter sido gerados por qualquer tipo de algoritmo de particionamento.

A fig. 4.10, gerada pelo algoritmo RCB, é um exemplo de um particionamento irregular, onde, os subdomínios estão enumerados e as fronteiras entre estes - que geram a necessidade de comunicação - estão marcadas com linhas negritadas.

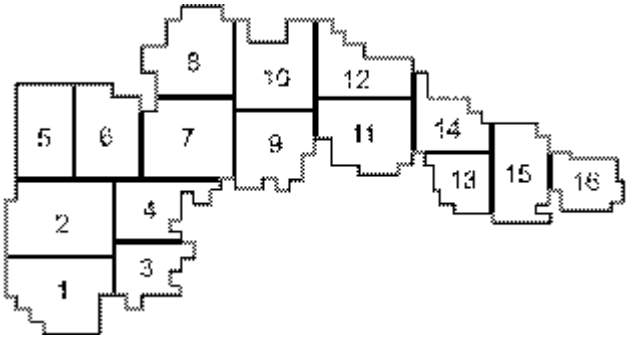


Figura 4.10: Subdomínios e suas fronteiras

Para solução de sistemas de equações gerados por subdomínios irregulares, os processos (subdomínios) devem ser capazes de se comunicar com seus vizinhos durante sua execução para a troca de informações referentes as suàs fronteiras.

Na comunicação através de troca de mensagens, cada primitiva de envio em um processo deve estar associada a uma primitiva de recebimento no processo destino. Se isso não ocorrer as informações enviadas serão armazenadas em um *buffer* ou os processos entram em estado de *deadlock*, dependendo do tamanho do *buffer* disponível.

Mais importante que o controle das primitivas de envio, é o controle das primitivas de recebimento, pois se dois processos quiserem se comunicar simultaneamente recebendo dados um do outro eles ficarão em estado *deadlock*, mesmo com a existência de *buffer*.

Neste trabalho, para a exploração do paralelismo inter e intra-nodos, estão sendo utilizadas duas bibliotecas de troca de mensagens que são o MPICH e o DECK.

Ambas as bibliotecas adotadas neste trabalho possuem implementações para diferentes redes de interconexão que são: *Fast Ethernet*, *Myrinet* e SCI.

As paralelizações desenvolvidas neste trabalho estão sendo avaliadas na rede *Fast Ethernet*, onde as implementações do MPICH e do DECK possuem um *buffer* de respectivamente 124 e 127 Kbytes, os quais se mostraram suficientes para as comunicações geradas na aplicação decorrente de nosso estudo de caso.

No entanto, para que as paralelizações desenvolvidas neste trabalho possam ser portáveis para outras máquinas cuja rede de interconexão não possua um *buffer* equivalente ao apresentado na rede *Fast Ethernet* foram desenvolvidas duas estratégias de ordenação de mensagens.

Uma ilustração de um caso que pode ocorrer *deadlock* está ilustrado na fig. 4.11, onde se os processos 7 e 9 quiserem comunicar-se entre si ambos enviando dados antes de receber, ambos ficarão bloqueados esperando uma mensagem que não chegara. Outra situação que pode causar *deadlock* é quando, todos os processos enviam seus dados antes de receber os dados dos vizinhos e, falta *buffer* suficiente para armazenar os dados enviados tornando a primitiva de envio bloqueante.

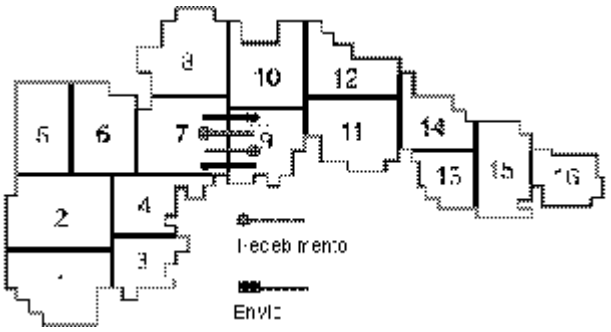


Figura 4.11: *Deadlock* na comunicação

Um exemplo de uma rede de interconexão com *buffer* menor que o da rede *Fast ethernet* é a *Myrinet*, onde os *buffers* das bibliotecas MPICH e DECK são 15 Kbytes.

4.2.1 Estratégia: Algoritmo Maximal

Essa estratégia é baseada na representação dos subdomínios através de um grafo gerado a partir do particionamento do domínio. Neste grafo, os vértices representam os subdomínios (ou processos) e as arestas a comunicação entre esses subdomínios.

A estratégia consiste em eliminar as arestas em n fases de modo que cada eliminação seja armazenada em ordem de ocorrência em uma tabela global de fases. Essa tabela deve conter a identificação dos dois vértices ligados pela aresta eliminada, formando uma tupla. A fig. 4.12 ilustra um exemplo da primeira fase, onde os processos e as comunicações são inicialmente identificadas.

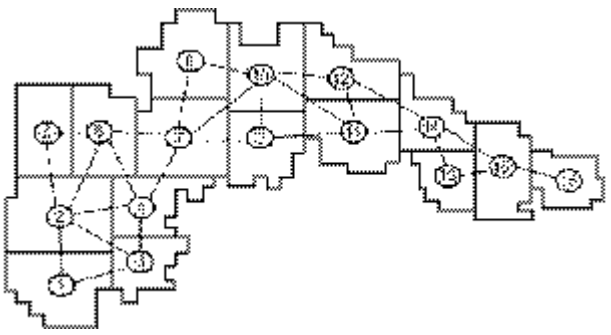


Figura 4.12: Domínio mapeado em um grafo

A decisão de qual a melhor ordem de eliminação das arestas adotada nessa estratégia é baseada em uma heurística que dá prioridade para eliminação das arestas ligadas a vértices que possuem mais vizinhos.

A tab. 4.1 mostra a seqüência completa de fases para a ordenação das mensagens considerando-se o domínio da fig. 4.12.

Tabela 4.1: Tabela global de fases da estratégia maximal

Ordem	Ciclo	Vértice	Vértice		Ordem	Ciclo	Vértice	Vértice
1	1	16	15		14	3	15	14
2	1	14	11		15	3	12	11
3	1	12	10		16	3	10	8
4	1	9	7		17	3	7	4
5	1	6	2		18	3	6	5
6	1	4	3		19	3	3	2
7	2	15	13		20	4	14	13
8	2	14	12		21	4	11	10
9	2	11	9		22	4	8	7
10	2	10	7		23	4	4	2
11	2	6	4		24	5	10	9
12	2	5	2		25	5	7	6
13	2	3	1		26	5	2	1

Devido ao fato dessa estratégia ser global, os dados obtidos na ordenação devem ser distribuídos entre os subdomínios de modo que eles recebam as tuplas em que sua identificação estiver contida e possam criar suas tabelas de fase locais. A tab. 4.2 ilustra como ficou a tabela local do processo 7, levando-se em consideração a tab. 4.1.

Tabela 4.2: Tabela local de fases da estratégia maximal (processo 7)

Ordem	Ciclo	Vértice	Vértice		Ordem	Ciclo	Vértice	Vértice
1	1	9	7		4	4	8	7
2	2	10	7		5	5	7	6
3	3	7	4					

Após a criação da tabela de fases locais os processos devem criar suas tabelas de ordenação. As tabelas de ordenação locais contêm duas vezes a identificação de cada vizinho, uma abaixo da outra, sendo uma associada à ação de envio e a outra à ação de recebimento.

A decisão de enviar ou receber mensagens para cada vizinho é baseada no valor de suas identificações dando prioridade de envio para os processos com identificador maior. Essa prioridade de envio para os processos com identificador maior elimina a possibilidade de criação de um ciclo de dependência e conseqüentemente de ocorrer um *deadlock*.

A tab. 4.3 ilustra como ficou a tabela de ordenação do processo 7. Nessa tabela a ação "1" representa envio e a "0" o recebimento.

Tabela 4.3: Tabela de ordenação maximal (exemplo para o processo 7 da fig. 4.12)

Ordem	Ação	Vértice		Ordem	Ação	Vértice
1	0	9		6	0	4
2	1	9		7	0	8
3	0	10		8	1	8
4	1	10		9	1	6
5	1	4		10	0	6

4.2.2 Comunicação: Par-Ímpar adaptativo

Em contraposição com a estratégia descrita acima (algoritmo Maximal), essa abordagem requer apenas informações locais, que são os identificadores dos subdomínios (processos) vizinhos.

Por necessitar de apenas informações locais, esta estratégia é mais adequada para ser adotada em algoritmos paralelos se comparada com a estratégia baseada no algoritmo Maximal.

O maior grau de paralelismo da estratégia Par-Impar adaptativo decorre do fato de que para a execução desta estratégia não há a necessidade de ocorrer um sincronismo e, além disso, esta estratégia não tem um custo computacional maior quando o domínio for dividido em um grande número de subdomínios.

Semelhante à estratégia baseado no algoritmo Maximal, essa segunda estratégia também é desenvolvida em n fases. Porém, esse algoritmo cria apenas a tabela de fases local e a utiliza para criar uma tabela de ordenação.

A fig. 4.13 ilustra um exemplo de um grafo representando o domínio global particionado, nesta figura são destacados o subdomínio 7 e os subdomínios adjacentes com suas respetivas comunicações identificadas.

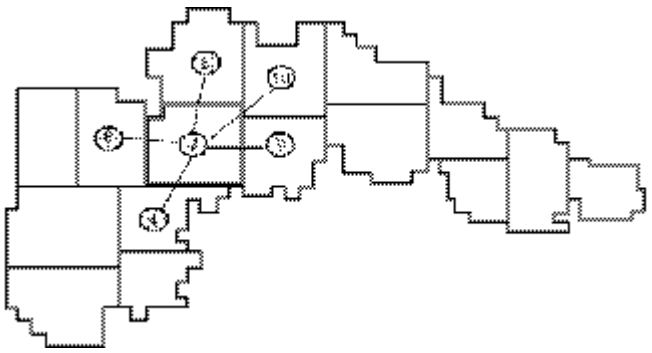


Figura 4.13: Domínio mapeado em um grafo local

A figura 4.13 que destaca o processo de número 7 é utilizada como exemplo para a criação de uma tabela de ordenação local para esta estratégia.

A decisão de qual a ordem em que o identificador dos vizinhos deve entrar na tabela de ordenação e qual a ação que deve ser associada (recebimento ou envio de mensagens) é baseada na criação de identificadores lógicos para cada processo vizinho e para o processo local. A tab. 4.4 contem os identificadores lógicos referente ao subdomínio 7 apresentado na fig. 4.13.

Tabela 4.4: Tabela de fases da estratégia Par-Ímpar adaptativo

	Vizinhos					**Local**
	Identificadores Reais					
	4	6	8	9	10	7
Fases	**Identificadores Lógicos**					
1	4	6	8	9	10	7
2	–	–	–	5	–	4
3	–	–	–	–	–	2

Na primeira fase, os identificadores lógicos possuem os mesmos valores dos identificadores reais. A alteração dos identificadores lógicos é baseada em sua paridade, os pares são divididos por dois e os ímpares são transformados em pares com a soma da unidade e, então, também são divididos por dois.

O algoritmo se baseia no fato de que cada identificador lógico de um subdomínio (processo) deve estar na tabela de seus vizinhos com o mesmo valor e que suas alterações sigam a mesma regra. Após possuir os identificadores lógicos é aplicado um algoritmo em *n* fases, sendo que em cada fase são efetuados 4 passos:

1. O primeiro passo é executar uma estratégia par-ímpar onde os nodos cujo identificador lógico tem paridade diferente da paridade do identificador lógico local são substituídos por um elemento nulo;

2. Os identificadores reais equivalentes aos identificadores lógicos substituídos por elementos nulos devem ser colocados na tabela de ordenação de mensagens. Se os identificadores lógicos são pares seus equivalentes reais devem ser associados com a ação de envio, caso contrário, com a ação de recebimento. Ou seja, os pares enviam primeiro;

3. Os elementos incluídos no passo acima devem ser incluídos novamente na mesma ordem mas com a ação oposta à já incluída no passo acima;

4. O passo final é alterar os identificadores lógicos restantes e voltar para o primeiro passo até que o único identificador lógico existente seja o do processo local.

A tab. 4.5 ilustra como ficou a tabela local do processo 7, levando-se em consideração a tab. 4.4.

Tabela 4.5: Tabela de ordenação par-ímpar adaptativo (processo 7)

Ordem	**Ação**	**Vértice**		**Ordem**	**Ação**	**Vértice**
1	0	4		6	1	6
2	0	6		7	1	8
3	0	8		8	1	10
4	0	10		9	1	9
5	1	4		10	0	9

5 AVALIAÇÃO E RESULTADOS

Neste capítulo são apresentados os resultados obtido com as paralelizações desenvolvidas dos métodos do GC e do GMRES fazendo uso das diferentes estratégias e ferramentas de paralelização. Os resultados estão dispostos em forma de gráficos, no entanto, para uma análise mais detalhada, as tabelas que geraram estes gráficos podem ser encontradas nos anexos deste trabalho. Com o objetivo de proporcinar uma melhor clareza, a apresentação dos resultados está dividida sobre três enfoques.

Inicialmente são apresentados e comparados os resultados das paralelizações que fazem uso do DECK com as que fazem uso do MPI, bloqueante e não bloqueante.

Em um segundo momento são apresentadas comparações de duas diferentes estratégias para uso de *clusters* multiprocessados, as quais são: uso de múltiplos processos em conjunto com múltiplas *threads* ou apenas uso de múltiplos processos. Ambas as abordagens fazem uso dos dois processadores das máquinas duais, ou seja, ambas são influenciadas pela contenção de memória.

Por fim, o último enfoque se refere à contenção de memória, onde é apresentada a contenção de memória encontrada na execução das paralelizações desenvolvidas.

5.1 Diferentes Bibliotecas de Troca de Mensagens

As figs. 5.1 e 5.2 apresentam, respectivamente, os gráficos de tempo de execução dos métodos do GC e do GMRES.

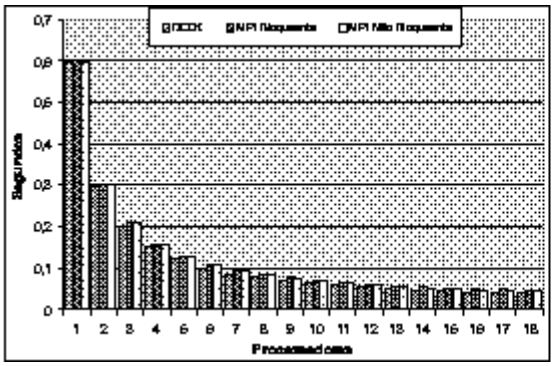


Figura 5.1: Tempo de execução do GC

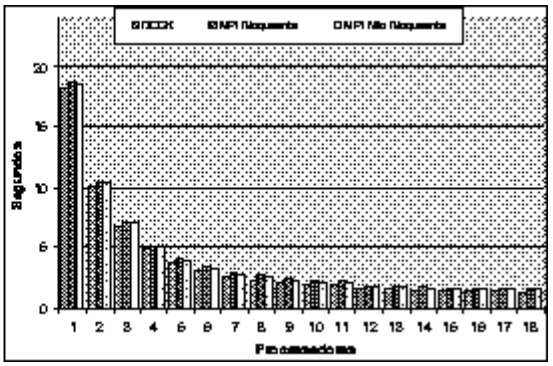


Figura 5.2: Tempo de execução do GMRES

Estes tempos de execução foram obtidos com a variação do número de máquinas de 1 até 18, embora as máquinas do *cluster* LabTec sejam duais, os testes que geraram estes tempos fizeram uso de apenas um processador de cada máquina. Nos resultados

pode ser observado que em ambos os métodos as paralelizações mantiveram o ganho de desempenho com até 18 nodos, que foi o número total de nodos disponíveis para os testes..

No que se refere à comparação de desempenho entre as diferentes bibliotecas de troca de mensagens, pode ser observado que a biblioteca DECK apresentou um melhor desempenho em relação à biblioteca MPICH. Com a biblioteca MPICH foram paralelizadas duas abordagens, com comunicação bloqueante e com comunicação não bloqueante, visando uma análise do uso de primitivas não bloqueantes na sobreposição de comunicação com processamento. Nas figs. 5.1 e 5.2 pode ser observado que a paralelização seguindo a segunda abordagem teve um melhor desempenho.

As figs. 5.3 e 5.4 apresentam, respectivamente, os gráficos de *speedup* dos tempos de execução apresentados acima.

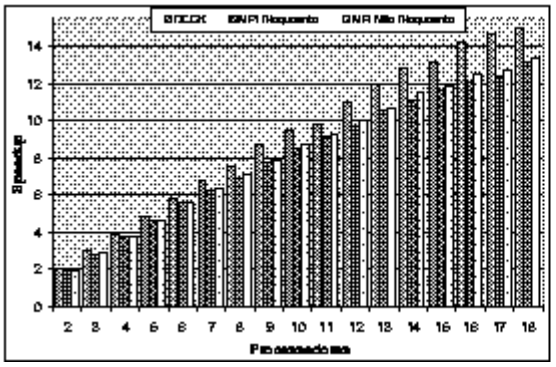


Figura 5.3: Speedup do GC

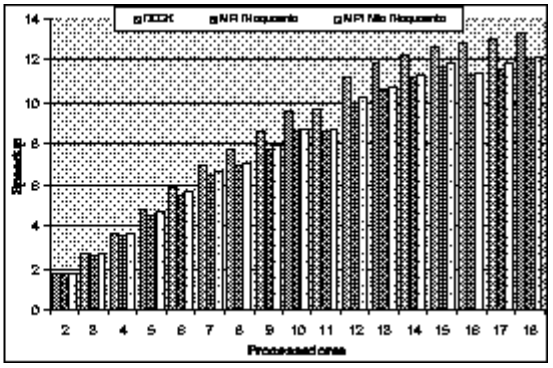


Figura 5.4: Speedup do GMRES

Nos gráficos de *speedup* apresentados acima pode ser observado que em ambos os métodos as paralelizações fazendo uso do DECK tiveram um aumento de desempenho em relação às que fazem uso do MPICH com o aumento do número de processadores.

Outro resultado observado foi que o método do GC teve um aumento de desempenho maior se comparado com o GMRES. Uma possível razão para isso é o fato do GMRES possuir maior *overhead* de comunicação. Esse *overhead* se deve ao fato de que no processo de Gram-Schimidt ocorre um grande número de operações de produtos escalares que necessitam de comunicação global.

As figs. 5.5 e 5.6 apresentam, respectivamente, os gráficos de eficiência dos tempos de execução apresentados acima.

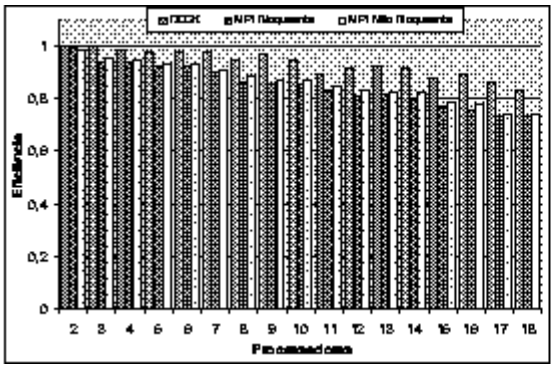


Figura 5.5: Eficiência do GC

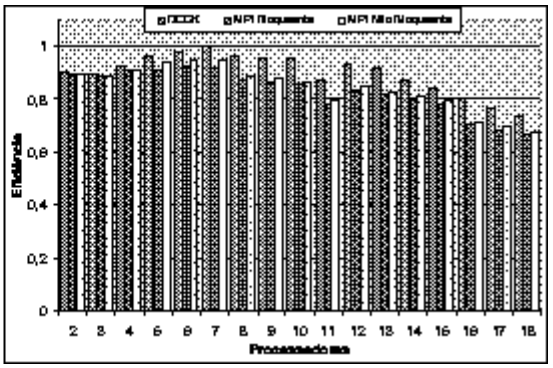


Figura 5.6: Eficiência do GMRES

Nos gráficos de eficiência apresentados acima pode ser observado que em ambos os métodos as paralelizações mantiveram uma boa eficiência com o aumento do número

de processadores. Baseando-se na eficiência concluiu-se que os algoritmos paralelisados com as estratégias adotadas mostraram-se bastante escaláveis considerando-se o número de nodos disponíveis para os testes.

No que se refere ao GMRES, pode ser observado que ocorreu uma melhor eficiência no aumento de 2 até 7 processadores, reação que não é comum devido ao aumento de *overhead* de comunicação. Uma possível justificativa para essa reação é o fato do método do GMRES usar uma grande quantidade de memória no armazenamento da base, e o aumento do número de processadores diminuí o tamanho da base. Essa justificativa decorreu do fato que foram feitos diferentes testes para se obter uma melhor estrutura de armazenamento para a base, e que esses testes mostraram que o armazenamento da base tem grande influência no desempenho do método.

5.2 Múltiplos Processos Vs. Múltiplas Threads

As figs. 5.7 e 5.8 apresentam uma comparação entre o uso de apenas múltiplos processos e o uso de múltiplos processos em conjunto com o uso de múltiplas *threads*. Essas figuras são respectivamente dos métodos do GC e do GMRES, os resultados são apresentados para as duas bibliotecas de troca de mensagens.

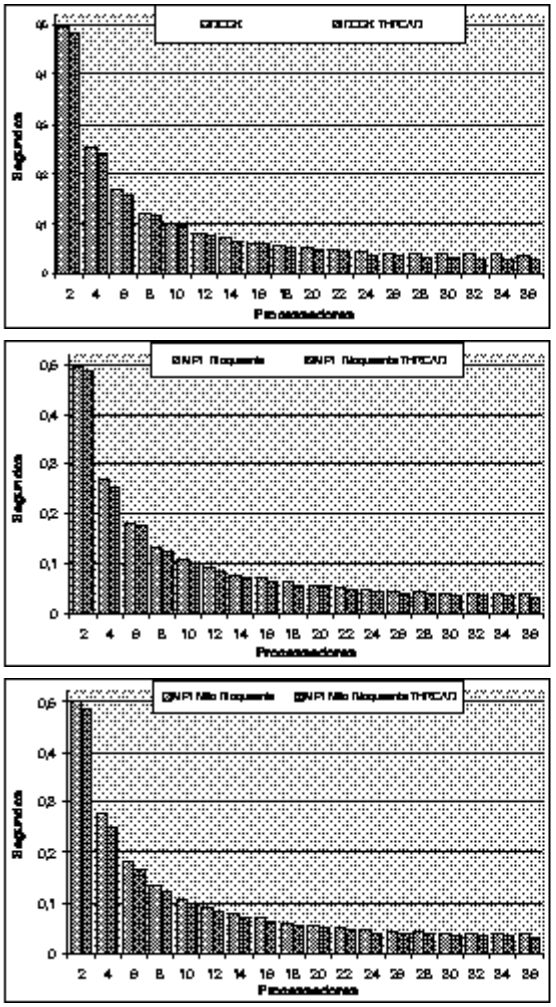


Figura 5.7: Processo vs. thread no GC

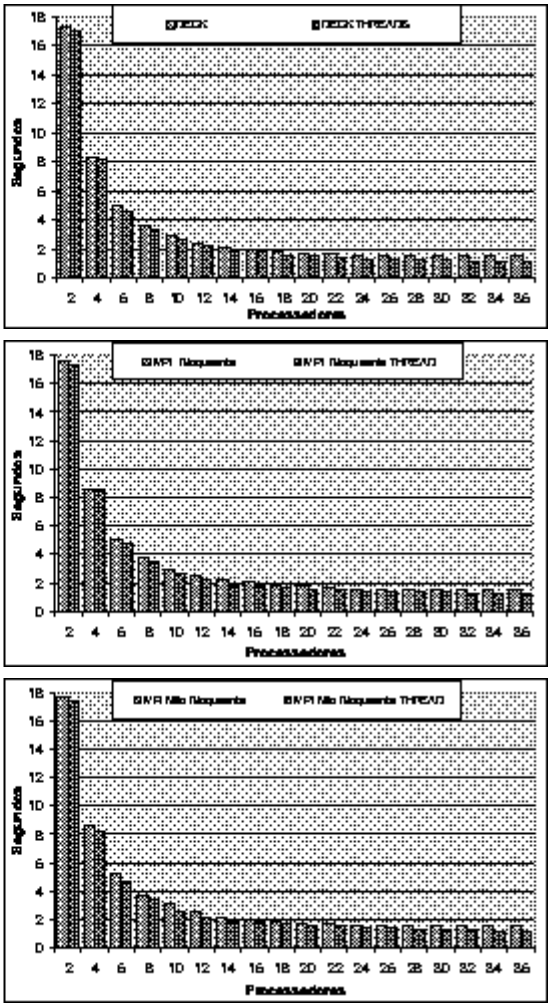


Figura 5.8: Processo vs. thread no GMRES

Nos resultados das figuras acima podemos observar que nas paralelizações dos dois métodos com ambas as bibliotecas e abordagens a exploração do paralelismo intra-nodal fazendo uso de múltiplas *threads* teve um melhor desempenho.

O ganho de desempenho no uso de múltiplas *threads* em relação ao com uso de apenas múltiplos processos é mais significativo de acordo com o aumento gradativo do número de processadores. Uma justificativa para esse aumento é a diminuição da carga de processamento e o aumento da comunicação quando se usa apenas múltiplos processos.

Na tab. 5.4 pode ser vista a porcentagem de ganho com o uso de múltiplas *threads* para 2 e 36 processos.

Tabela 5.1: Processos vs. threads GC

	2 Procs	**36 Procs**
DECK	2,6 %	26,3 %
MPI	1,9 %	17,3 %
IMPI	3,1 %	18,2 %

Tabela 5.2: Processos vs. threads GMRES

	2 Procs	**36 Procs**
DECK	1,46 %	24,0 %
MPI	1,3 %	20,6 %
IMPI	1,7 %	20,9 %

A abordagem adotada na paralelização com multiplas *threads* consiste na criação de uma segunda *thread*, semelhante a já existente, exceto pelo fato de não conter primitivas de comunicação. As duas *threads* sincronizam antes e depois de qualquer tipo de comunicação inter-nodos através de monitores.

As *threads* que estão sendo adotadas são do nível de sistema e possuem o mesmo custo de criação e escalonamento apresentado em um processo.

5.3 Análise da Contenção de Memória

Nesta seção estão sendo comparadas as execuções de processos comunicantes usando múltiplos processadores em uma mesma máquina ou em máquinas diferentes.

Os resultados obtidos sofrem influência de alguns fatores como por exemplo a contenção de memória existente em máquinas multiprocessadas, o custo de comunicação intra-nodos através de troca de mensagens e o custo de comunicação inter-nodos através de troca de mensagens.

As figs. 5.9 e 5.10 apresentam respectivamente os resultados obtidos com o GC e com o GMRES.

Nos graficos pode ser observado que a contenção existente no uso de dois processos em máquinas duais diminui gradativamente com o aumento do número de máquinas. Como justificativa tem-se o fato de que o aumento do número de máquinas diminui a carga de processamento das mesmas e consequentemente o uso de memória pelos processos.

No método do GMRES a contenção com poucas máquinas é maior mas à medida que o número de máquinas aumenta essa contenção diminui mais rapidamente.

Na tab. 5.3 pode ser vista a porcentagem de contenção apresentada na execução dos métodos para 2 e 18 processos.

Tabela 5.3: Processos vs. threads GC

	2 Procs	**18 Procs**
DECK	38,12 %	21,9 %
MPI	37,7 %	20,3 %
IMPI	37,3 %	19,1 %

Tabela 5.4: Processos vs. threads GMRES

	2 Procs	**18 Procs**
DECK	41,4 %	10,9 %
MPI	39,7 %	6,6 %
IMPI	39,8 %	6,6 %

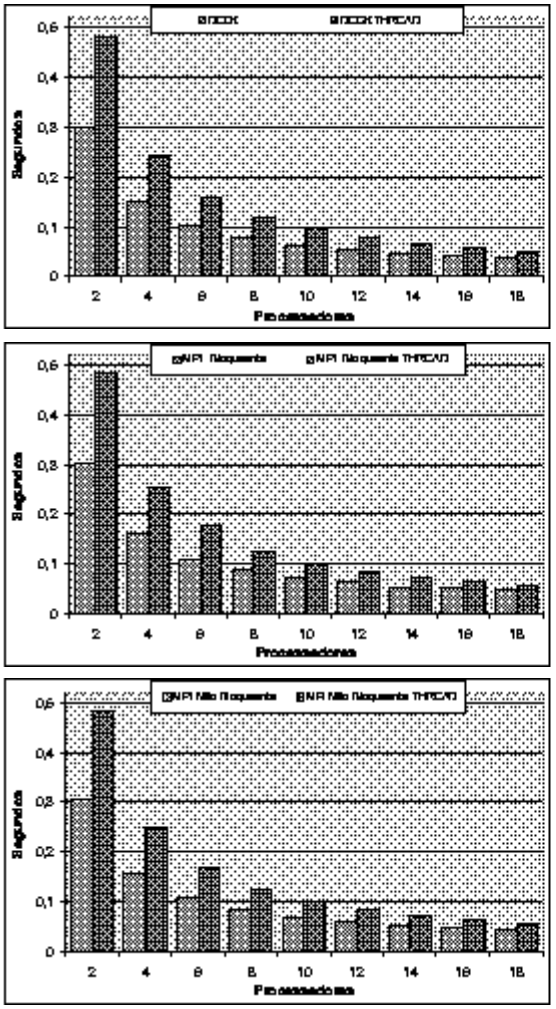


Figura 5.9: Contenção de memória no GC

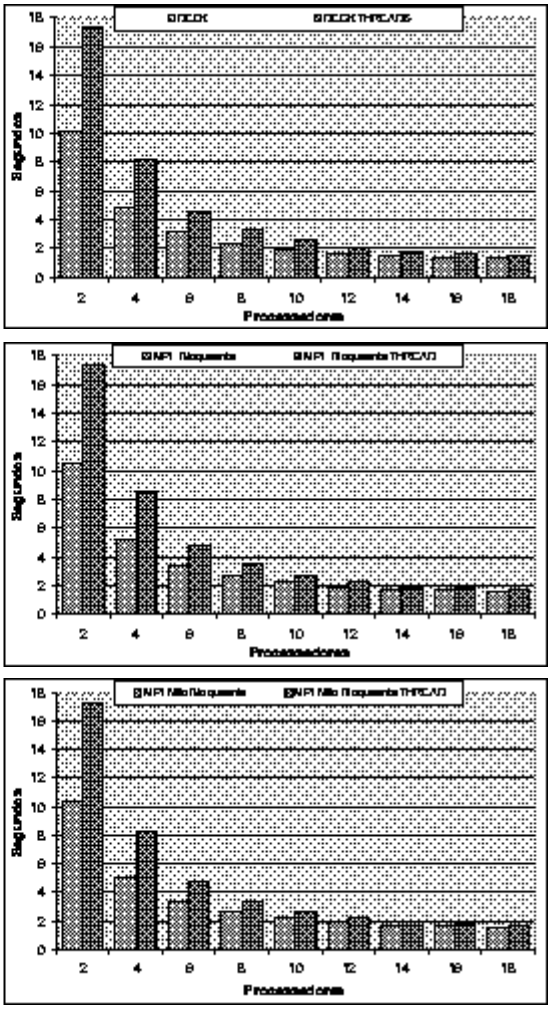


Figura 5.10: Contenção de memória no GMRES

Os resultados obtidos mostraram que o uso de máquinas monoprocessadas apresentam um melhor resultado se comparadas com o uso de máquinas multiprocessadas. Essa diferença tende a diminuir com o aumento do número de máquinas em processamento devido a diminuição do uso de memória por parte dos processos.

As versões do MPICH e do DECK que foram adotadas não possuem o recurso de comunicação intra-nodos através de memória compartilhada, característica que pode influenciar os resultados obtidos.

Os testes foa efetuados em um domínio que resultou 180.000 equações após a discretização. O método do GMRES foi executado com a base de tamanho 20. O número de iterações apresentado pelos métodos do GC e GMRES foram respectivamente 8 e 5 com o erro definido em 0.00000001.

6 CONCLUSÕES, CONTRIBUIÇÕES E TRABALHOS FUTUROS

Neste trabalho desenvolveu-se estratégias e estruturas de dados para a obtenção da solução paralela, via decomposição de dados, para os métodos iterativos do Gradiente Conjugado e do Resíduo Mínimo Generalizado. Esses *solvers* paralelos foram incorporados ao modelo computacional paralelo multifísica, que foi desenvolvido no GMCPAD objetivando simular a hidrodinâmica e o transporte escalar de substâncias em corpos de água bidimensionais e tridimensionais.

Sendo esse o capítulo das conclusões, contribuições e trabalhos futuros, assim ele foi organizado. A título de conclusões, nesta primeira parte do texto é feita uma síntese sobre os principais assuntos discutidos nos capítulos 2 a 5.

6.1 Conclusões

No capítulo dois foi apresentado o escopo do trabalho visando familiarizar o leitor ao ambiente onde as paralelizações foram desenvolvidas e conceitos relacionados aos métodos paralelizados. Este capítulo foi dividido em quatro seções, onde, inicialmente foi apresentado a arquitetura física da máquina em que o trabalho foi desenvolvido e avaliado, nesta seção foram descritas as principais características pertinentes a este tipo de arquitetura e as características existentes na máquina em que o trabalho foi desenvolvido. A segunda seção deste capítulo se refere aos recursos de software adotados neste trabalho. Nesta segunda seção apresentou-se as bibliotecas adotadas para a exploração do paralelismo inter-nodos e para a exploração do paralelismo intra-nodos, onde foram descritas suas principais características e também seu modo de uso. As próximas duas seções, três e quatro, contemplaram características da aplicação que foi desenvolvida. Na seção três foram apresentadas características de um modelo computacional que simula um evento físico. Na seção quatro foram apresentados os métodos paralelizados neste trabalho, onde se destacou as principais características dos métodos e seus algoritmos.

O capítulo três contemplou assuntos relativos ao particionamento do problema a ser resolvido através de uma abordagem paralela. Neste capítulo, inicialmente descreveu-se as principais características de diferentes tipos de particionamento com suas vantagens e desvantagens. Em um segundo momento abordou-se a geração do sistema de equações a partir de domínios particionados, onde se descreveu a maneira como as matrizes dos coeficientes é gerada e qual a melhor maneira para efetuar seu armazenamento.

No capítulo quatro foi feita uma abordagem mais aplicada, no sentido de descrever como as características da arquitetura, dos métodos paralelizados e do sistema de particionamento adotados, influenciaram no desenvolvimento do algoritmo paralelo.

Neste capítulo inicialmente foi descrita a estrutura de armazenamento e acesso dos dados adotada, onde é feita uma explanação sobre as informações necessárias para se efetuas as operações dos métodos de maneira mais otimizada possível e diminuir o máximo o custo de comunicação entre os diferentes subdomínios. Visando tornar o trabalho desenvolvido portável para diferentes redes de interconexão sem sofrer influência de parada ocasionada por falta de *buffer*, foram adotadas estratégias de ordenação de mensagens. Essas estratégias objetivam efetuar um controle de fluxo diminuindo a necessidade de *buffer* por parte da rede. Na segunda parte deste capítulo foram descritas duas estratégias de ordenação de mensagens desenvolvidas.

Por fim, no capítulo cinco foram apresentados os resultados obtidos no dois métodos paralelizados com as diferentes bibliotecas de comunicação, fazendo ou não o uso de múltiplas *threads* como recurso para exploração do paralelismo intra-nodos e fazendo ou não o uso de primitivas não bloqueantes como recurso para sobrepor comunicação com processamento. Além dos testes já citados, foi apresentado o nível de contenção de memória que ocorreu no uso de máquinas multiprocessadas. Neste capítulo todos os resultados foram discutidos e justificados levanto em consideração as características pertinentes a arquitetura e também aos métodos paralelizados.

Na exploração do paralelismo inter-nodos, pode-se concluir que a biblioteca DECK proporciona um bom desempenho, relativo a bibliotecas MPICH, e portanto, é uma boa ferramenta para exploração do paralelismo, não somente inter-nodos mas também intra-nodos, obtendo um maior desempenho quando comparada com o MPICH. No que se refere ao uso de primitivas bloqueantes ou não bloqueantes na biblioteca MPICH, concluiu-se que devido ao fato das primitivas não bloqueantes permitirem uma sobreposição de comunicação com processamento na operação de produto matriz-vetor, elas proporcionaram um maior ganho de desempenho.

Já no paralelismo intra-nodos, os testes mostraram que a abordagem que faz uso de múltiplas *threads* em conjunto com o uso de múltiplos processos em *clusters* multiprocessados tem um desempenho maior se comparados com a abordagem que faz uso de apenas múltiplos processos. Baseando-se nos testes concluiu-se que o uso de múltiplas *threads* em conjunto com múltiplos processos não apresentou uma diferença de desempenho muito significativa quando comparada com o uso de apenas múltiplos processos em situações com grande carga de trabalho, no entanto, apresentou uma diferença mais significativa com a diminuição da carga de trabalho, característica que permitiu aumentar a escalabilidade das paralelizações.

No decorrer do trabalho observou-se que o sincronismo tem uma grande influência no desempenho de uma aplicação paralela, e que nos métodos paralelizados existem operações que causam um sincronismo global. Esse sincronismo global mostrou que em situações em que a carga de processamento entre as maquinas é muito distinta, o desempenho da aplicação fica nivelado pelo processo com mais carga de trabalho. Baseando-se nestas observações concluiu-se que a estrutura de armazenamento desenvolvida proporcionou diminuir a influência do sincronismo global devido ao fato de proporcionar um melhor balanceamento de carga, sem aumentar o custo de comunicação. Além de diminuir o sincronismo concluiu-se que a estrutura de armazenamento adotada permite aumentar o desempenho no uso de primitivas não bloqueantes.

Ainda no que se refere a estruturas de armazenamento, concluiu-se que a maneira como os dados estão armazenados devem ser de acordo com o acesso a estes pelas operações aritméticas. Concluiu-se que na arquitetura de PCs a estrutura de armazenamento devem proporcionar as operações a fazer o máximo uso dos dados que já

estão alocados na memórias cache e RAM.

6.2 Contribuições

A contribuição deste trabalho foi o desenvolvimento de *solvers* iterativos paralelizados empregando duas bibliotecas de trocas de mensagens e uma outra biblioteca de *threads*. Esses *solvers* foram especificamente projetados para arquiteturas tipo *clusters* de PCs.

A escolha por esses *solvers* deveu-se ao fato de que a literatura técnica indica que o método do Gradiente Conjugado e o método do Resíduo Mínimo Generalizados são os mais efetivos para resolver os sistemas de equações gerados no modelo computacional desenvolvido no GMCPAD. Note-se, porém, que esses *solvers* podem ser empregados para a solução de qualquer sistemas de equações lineares, desde que esses sistemas obedeçam as características próprias de cada um dos métodos de solução.

Especificamente, as contribuições mais importantes para a área de Computação Científica Paralela foram:

1. O desenvolvimento e a implementação em paralelo de *solvers* que empregam estruturas de dados completamente arbitrárias;

2. O estudo e a análise comparativa do desempenho de duas bibliotecas de trocas de mensagens em arquiteturas tipo *cluster*;

3. O estudo e a análise comparativa do uso de primitivas bloqueantes e não bloqueantes;

4. O estudo e a análise comparativa do uso de apenas múltiplos processos ou o uso de múltiplos processos em conjunto com múltiplas *threads*;

5. O desenvolvimento e a implementação de duas estratégias de ordenação que podem ser empregadas para vários fins como, por exemplo, na ordenação de mensagens; na ordenação de acesso a áreas de dados, na ordenação de operações;

6. A integração dos *solvers* desenvolvidos no modelo HIDRA (Hidrodinâmica e Transporte 2D e 3D).

Desses estudos resultaram quatro (4) trabalhos, em autoria ou co-autoria, os quais foram publicados em eventos nacionais e/ou regionais. Dentre esses trabalhos estão artigos e o trabalho individual. As referências dos trabalhos são apresentados na bibliografia e foram: (PICININ JUNIOR et al., 2001), (PICININ JUNIOR, 2001), (RIZZI, 2002), (PICININ JUNIOR et al., 2003) e (PICININ JUNIOR et al., 2002). Além disso, outros artigos estão sendo preparados para submissão no período de 2003 para: WSDP e PatCo2003 (*Parallel Computing Conference*).

Este trabalho, que visou a obtenção de soluções paralelas para a solução dos sistemas de equações para o modelo computacional desenvolvido no GMCPAD recebeu, e ofereceu, contribuições do grupo de pesquisa, como pode ser visto nos artigos em co-autoria e autoria.

6.3 Trabalhos Futuros

Neste trabalho foram desenvolvidas atividades que visaram paralelização de métodos iterativos do subespaço de Krylov. Porém, alguns pontos importantes não puderam ser contemplados nesta dissertação. Em trabalhos futuros deve-se considerar a realização dessas atividades objetivando ao aprimoramento e ao complemento dessa dissertação.

Especificamente, as principais atividades a serem desenvolvidas podem ser pontuadas como:

1. O desenvolvimento e a implementação em paralelo de métodos e técnicas de pré-condicionamento para acelerar os métodos de solução (BENZI; TUMA, 1999);

2. O desenvolvimento de métodos de solução para sistemas de equações não-lineares, dado que essa abordagem é, algumas vezes, imprescindível às soluções de equações não-lineares (CRUZ, 1997) (KEYES; VENKATAKRISHNAN, 1997) (GROPP et al., 1998);

3. O desenvolvimento de estruturas de dados visando o uso efetivo das bibliotecas BLAS 1, 2 e 3 para a solução das operações matriciais que compõem qualquer método de solução interativo do subespaço de Krylov (DONGARRA, 2002);

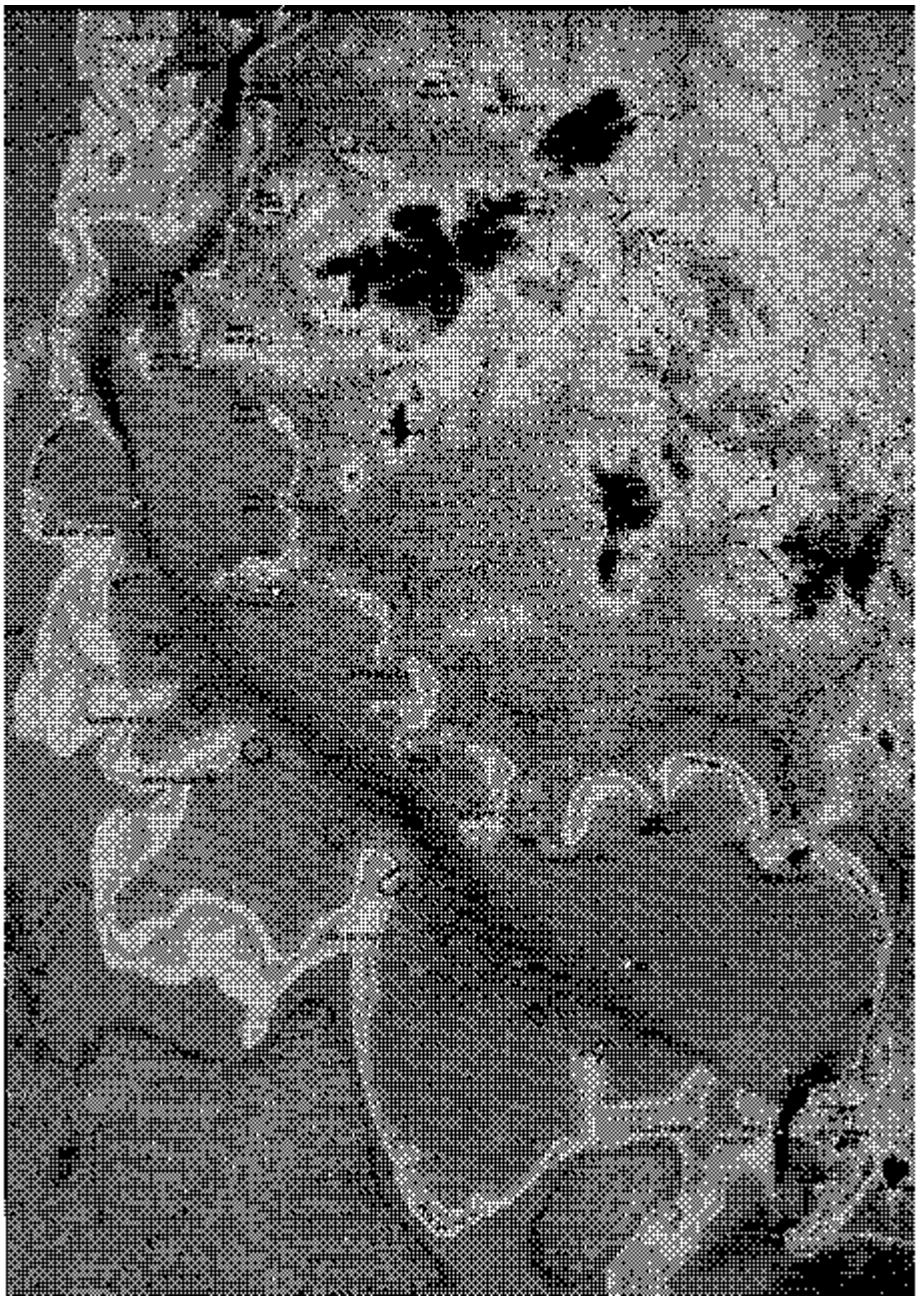
4. O uso e a análise de desempenho, comparativamente ao métodos já implementados, da bibliotecas PETSc, dado que a literatura mostra que essa é uma das mais eficientes (SMITH et al., 2002);

5. O uso, e a análise comparativa com outras, da biblioteca de *threads* OpenMP, visando a exploração do paralelismo intra-nodos (OPENMP: SIMPLE, PORTABLE, SCALABLE SMP PROGRAMMING, 2002) (LEWIS; BERG, 1998).

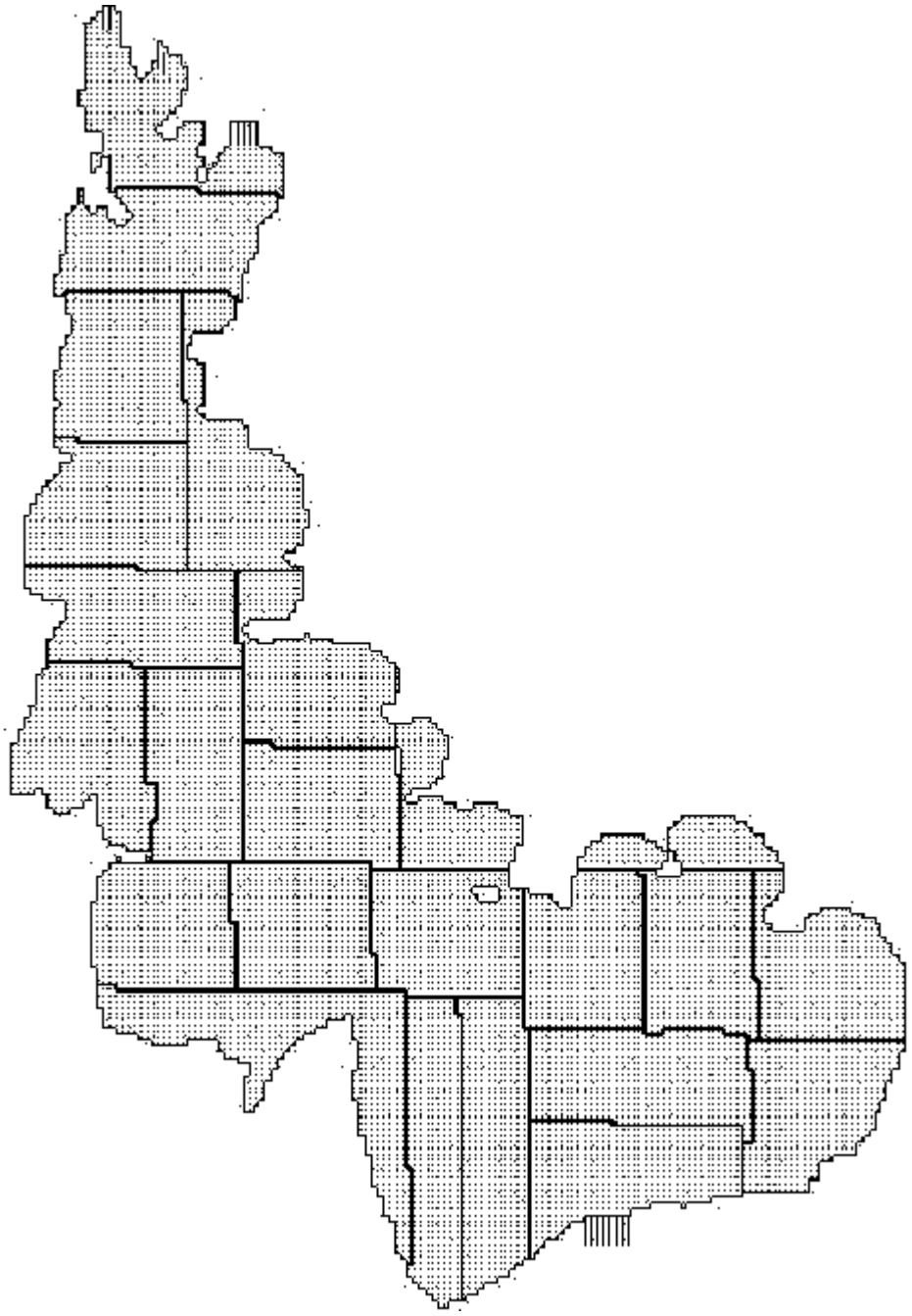
Todos os códigos gerados por esses trabalhos podem ser agregados aos códigos já desenvolvidos no grupo de modo a oferecer flexibilidade e eficiência adicionais ao modelo computacional multifísica.

APÊNDICE A ESTUDO DE CASO

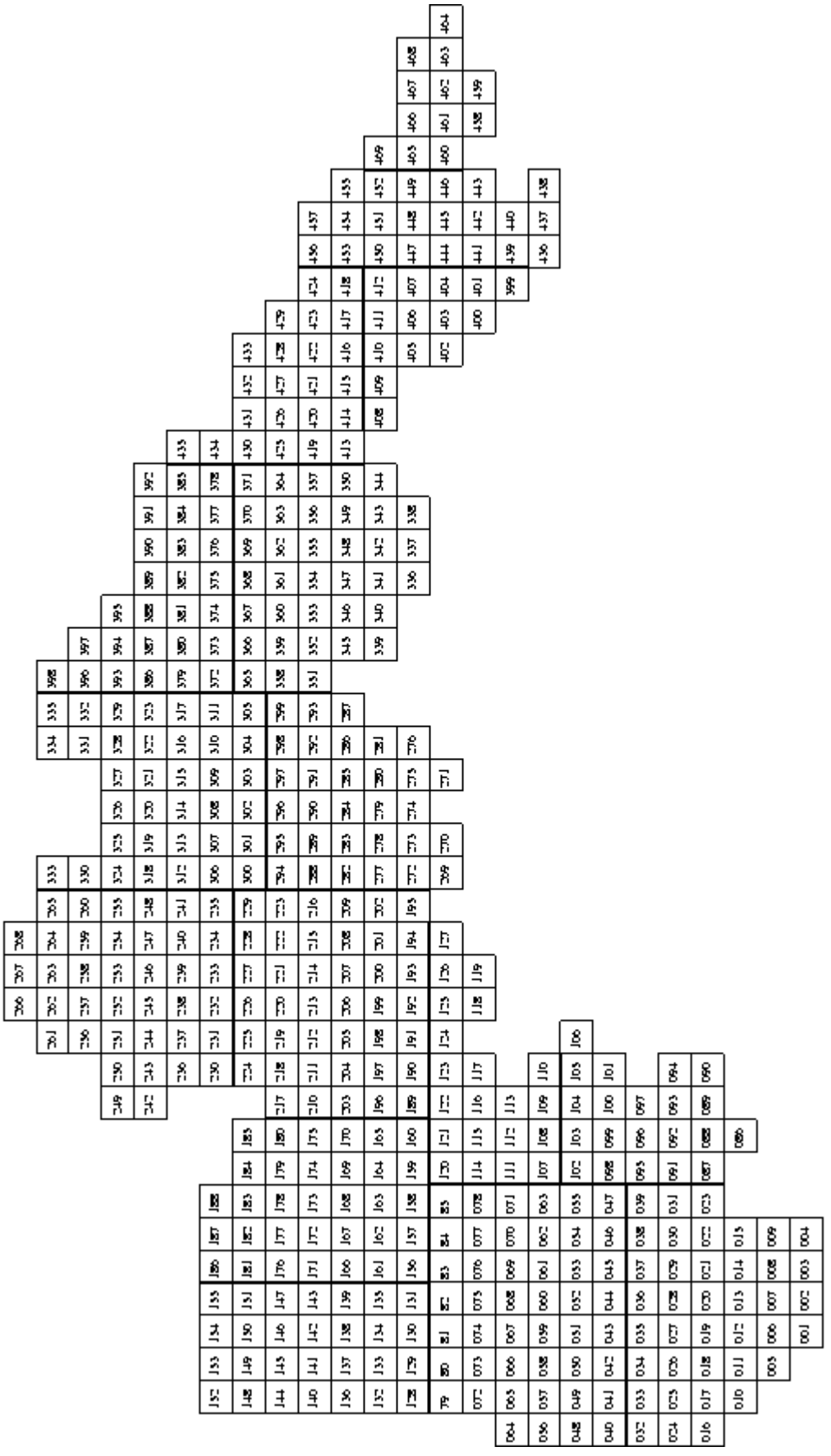
A.1 Domínio do Estudo de Caso



A.2 Domínio do Estudo de Caso



A.3 Domínio do Estudo de Caso Enumerado (simplificado)



A.4 Tabela de Resultados no Estudo de Caso

Tabela A.1: Tempos de execução do GC com diferentes bibliotecas

		DECK	MPI Bloqueante	MPI Não Bloqueante
		Segundos		
Processadores	1	0,594695	0,600326	0,596258
	2	0,298059	0,302652	0,303207
	3	0,198922	0,212886	0,209132
	4	0,151379	0,159313	0,157217
	5	0,121734	0,129753	0,128572
	6	0,101749	0,108215	0,106811
	7	0,087355	0,095196	0,093266
	8	0,078616	0,086831	0,084334
	9	0,068391	0,077901	0,075904
	10	0,062881	0,070431	0,068274
	11	0,060777	0,065435	0,064373
	12	0,053885	0,061925	0,059541
	13	0,049695	0,056718	0,055787
	14	0,046446	0,054193	0,051417
	15	0,045275	0,051736	0,050479
	16	0,041786	0,049554	0,047603
	17	0,040681	0,048461	0,047106
	18	0,039649	0,045659	0,044562

Tabela A.2: Tempos de execução do GMRES com diferentes bibliotecas

		DECK	MPI Bloqueante	MPI Não Bloqueante
		Segundos		
Processadores	1	18,185274	18,655447	18,645437
	2	10,120284	10,455817	10,414811
	3	6,7749670	7,022109	7,013409
	4	4,905890	5,146529	5,107904
	5	3,776493	4,079863	3,959194
	6	3,108244	3,368696	3,274876
	7	2,610033	2,899992	2,811381
	8	2,360241	2,681468	2,641399
	9	2,109874	2,410495	2,360929
	10	1,902954	2,175526	2,158164
	11	1,889304	2,163426	2,132818
	12	1,625078	1,869421	1,825616
	13	1,528513	1,756234	1,731741
	14	1,491281	1,674029	1,645235
	15	1,442435	1,593906	1,569051
	16	1,412044	1,658114	1,641561
	17	1,396849	1,606871	1,573509
	18	1,370951	1,547444	1,530833

Tabela A.3: Speedup do GC com diferentes bibliotecas

		DECK	MPI Bloqueante	MPI Não Bloqueante
		Speedup		
Processadores	2	1,995225	1,983552	1,966504
	3	2,989588	2,819941	2,851108
	4	3,928517	3,768217	3,792579
	5	4,885200	4,626683	4,637541
	6	5,844725	5,547530	5,582365
	7	6,807795	6,306210	6,393090
	8	7,564554	6,913728	7,070197
	9	8,695515	7,706268	7,855422
	10	9,457467	8,523604	8,733309
	11	9,784869	9,174386	9,262547
	12	11,036373	9,6944045	10,014242
	13	11,966898	10,584400	10,688117
	14	12,804008	11,077556	11,596514
	15	13,135173	11,603641	11,812001
	16	14,231919	12,114582	12,525639
	17	14,618495	12,387817	12,657793
	18	14,998991	13,148032	13,380413

Tabela A.4: Speedup do GMRES com diferentes bibliotecas

		DECK	MPI Bloqueante	MPI Não Bloqueante
		Speedup		
Processadores	2	1,796913	1,784217	1,790280
	3	2,684186	2,656672	2,658541
	4	3,706824	3,624859	3,650310
	5	4,815386	4,572567	4,709402
	6	5,850658	5,537883	5,693478
	7	6,967449	6,432930	6,632127
	8	7,704837	6,957176	7,058924
	9	8,619127	7,739259	7,897500
	10	9,556339	8,575143	8,639490
	11	9,625382	8,623103	8,742160
	12	11,190400	9,979264	10,213230
	13	11,897362	10,622415	10,766873
	14	12,194397	11,144040	11,332993
	15	12,607343	11,704232	11,883257
	16	12,878687	11,251003	11,358357
	17	13,018782	11,609797	11,849590
	18	13,264714	12,055652	12,179928

Tabela A.5: Eficiência do GC com diferentes bibliotecas

		DECK	MPI Bloqueante	MPI Não Bloqueante
		Eficiência		
Processadores	2	0,997612	0,991776	0,983252
	3	0,996529	0,939980	0,950369
	4	0,982129	0,942054	0,948144
	5	0,977040	0,925336	0,927508
	6	0,974120	0,924588	0,930394
	7	0,972542	0,900887	0,913298
	8	0,945569	0,864216	0,883774
	9	0,966168	0,856252	0,872824
	10	0,945746	0,852360	0,873330
	11	0,889533	0,834035	0,842049
	12	0,919697	0,807867	0,834520
	13	0,920530	0,814184	0,822162
	14	0,914572	0,791254	0,828322
	15	0,875678	0,773576	0,787466
	16	0,889494	0,757161	0,782852
	17	0,859911	0,728695	0,744576
	18	0,833277	0,730446	0,743356

Tabela A.6: Eficiência do GMRES com diferentes bibliotecas

		DECK	MPI Bloqueante	MPI Não Bloqueante
		Eficiência		
Processadores	2	0,898456	0,892108	0,895144
	3	0,894728	0,885557	0,886180
	4	0,926706	0,906214	0,912577
	5	0,963077	0,914513	0,941880
	6	0,975109	0,922980	0,948913
	7	0,995349	0,918990	0,947446
	8	0,963104	0,869647	0,882365
	9	0,957680	0,859917	0,877500
	10	0,955633	0,857514	0,863949
	11	0,875034	0,783918	0,794741
	12	0,932533	0,831605	0,851102
	13	0,915181	0,817108	0,828221
	14	0,871028	0,796002	0,809499
	15	0,840489	0,780282	0,792217
	16	0,804917	0,703187	0,709897
	17	0,765810	0,682929	0,697034
	18	0,736928	0,669758	0,676662

Tabela A.7: Tempo de execução do GC com DECK (processo vs. thread)

		DECK	DECK Threads
		Segundos	
Processadores	2	0,494569	0,481706
	4	0,253095	0,243067
	6	0,167668	0,159386
	8	0,123535	0,118108
	10	0,103077	0,097451
	12	0,081712	0,077664
	14	0,072283	0,067001
	16	0,061538	0,059089
	18	0,055834	0,050801
	20	0,050994	0,047461
	22	0,047419	0,044344
	24	0,044081	0,038224
	26	0,043084	0,035339
	28	0,042016	0,032168
	30	0,041073	0,031888
	32	0,040382	0,030393
	34	0,039804	0,029769
	36	0,038004	0,027971

Tabela A.8: Tempo de execução do GMRES com DECK (processo vs. thread)

		DECK	DECK Threads
		Segundos	
Processadores	2	17,335776	17,081271
	4	8,285444	8,141163
	6	4,893329	4,624905
	8	3,574176	3,289742
	10	2,897212	2,589377
	12	2,274214	2,142716
	14	2,058052	1,851935
	16	1,850062	1,711628
	18	1,732376	1,538987
	20	1,631024	1,478827
	22	1,609225	1,301159
	24	1,580891	1,261155
	26	1,572141	1,221817
	28	1,561243	1,190991
	30	1,549919	1,170441
	32	1,531412	1,159961
	34	1,524171	1,154807
	36	1,531216	1,162637

Tabela A.9: Tempo de execução do GC com MPI bloqueante (processo vs. thread)

		MPI	MPI Threads
		Segundos	
Processadores	2	0,495729	0,485854
	4	0,267515	0,254447
	6	0,179931	0,175113
	8	0,132219	0,125211
	10	0,106718	0,100433
	12	0,090973	0,084087
	14	0,078061	0,071752
	16	0,070962	0,064362
	18	0,061859	0,057307
	20	0,056251	0,054741
	22	0,051057	0,048245
	24	0,048367	0,042321
	26	0,044738	0,039703
	28	0,042801	0,038983
	30	0,041308	0,036673
	32	0,040785	0,036013
	34	0,040128	0,035621
	36	0,039223	0,032437

Tabela A.10: Tempo de execução do GMRES com MPI bloqueante (processo vs. thread)

		MPI	MPI Threads
		Segundos	
Processadores	2	17,574747	17,339666
	4	8,663096	8,493623
	6	5,132381	4,732021
	8	3,836033	3,471241
	10	2,899481	2,667973
	12	2,469446	2,188411
	14	2,172993	1,884432
	16	2,045235	1,809528
	18	1,891362	1,658242
	20	1,767079	1,607709
	22	1,659507	1,517523
	24	1,611128	1,472249
	26	1,592112	1,413419
	28	1,579628	1,381253
	30	1,560019	1,324382
	32	1,542212	1,284382
	34	1,529721	1,257326
	36	1,542312	1,223953

Tabela A.11: Tempo de execução do GC com MPI não bloqueante (processo vs. thread)

		MPI	MPI Threads
		Segundos	
Processadores	2	0,499875	0,483919
	4	0,278891	0,250571
	6	0,183429	0,168244
	8	0,133851	0,124239
	10	0,107329	0,100102
	12	0,090973	0,082407
	14	0,078222	0,071261
	16	0,070904	0,062997
	18	0,059241	0,055142
	20	0,054242	0,053434
	22	0,049515	0,047233
	24	0,047371	0,041324
	26	0,043978	0,037988
	28	0,042981	0,038003
	30	0,041143	0,034963
	32	0,041007	0,035501
	34	0,040098	0,034561
	36	0,039143	0,031996

Tabela A.12: Tempo de execução do GMRES com MPI não bloqueante (processo vs. thread)

		MPI	MPI Threads
		Segundos	
Processadores	2	17,607467	17,304478
	4	8,645211	8,217259
	6	5,15883	4,720881
	8	3,685252	3,364887
	10	3,146915	2,598108
	12	2,557447	2,153660
	14	2,174174	1,869887
	16	2,040909	1,800968
	18	1,882611	1,639984
	20	1,758729	1,582457
	22	1,650902	1,505553
	24	1,602331	1,463121
	26	1,580072	1,399432
	28	1,560135	1,369125
	30	1,550427	1,309741
	32	1,540132	1,260025
	34	1,526615	1,237435
	36	1,535495	1,213291

Tabela A.13: Tempo de execução do GC com DECK (contenção de memória)

		DECK	DECK Threads
		Segundos	
Processadores	2	0,298059	0,481706
	4	0,151379	0,243067
	6	0,101749	0,159386
	8	0,078616	0,118108
	10	0,062881	0,097451
	12	0,053885	0,077664
	14	0,046446	0,067001
	16	0,041786	0,059089
	18	0,039649	0,050801

Tabela A.14: Tempo de execução do GMRES com DECK (contenção de memória)

		DECK	DECK Threads
		Segundos	
Processadores	2	10,120284	17,291271
	4	4,905890	8,141163
	6	3,108244	4,624905
	8	2,360241	3,289742
	10	1,902954	2,589377
	12	1,625078	2,142716
	14	1,491281	1,851935
	16	1,412044	1,711628
	18	1,370951	1,538987

Tabela A.15: Tempo de execução do GC com MPI bloqueante (contenção de memória)

		MPI	MPI Threads
		Segundos	
Processadores	2	0,302652	0,485854
	4	0,159313	0,254447
	6	0,108215	0,175113
	8	0,086831	0,125211
	10	0,070431	0,100433
	12	0,061925	0,084087
	14	0,054193	0,071752
	16	0,049554	0,064362
	18	0,045659	0,057307

Tabela A.16: Tempo de execução do GMRES com MPI bloqueante (contenção de memória)

		MPI	MPI Threads
		Segundos	
Processadores	2	10,455817	17,339666
	4	5,146529	8,493623
	6	3,368696	4,732021
	8	2,681468	3,471241
	10	2,175526	2,667973
	12	1,869421	2,188411
	14	1,674029	1,884432
	16	1,658114	1,809528
	18	1,547444	1,658242

Tabela A.17: Tempo de execução do GC com MPI não bloqueante (contenção de memória)

		MPI	MPI Threads
		Segundos	
Processadores	2	0,303207	0,483919
	4	0,157217	0,250571
	6	0,106811	0,168244
	8	0,084334	0,124239
	10	0,068274	0,100102
	12	0,059541	0,082407
	14	0,051417	0,071261
	16	0,047603	0,062997
	18	0,044562	0,055142

Tabela A.18: Tempo de execução do GMRES com MPI não bloqueante (contenção de memória)

		MPI	MPI Threads
		Segundos	
Processadores	2	10,414811	17,304478
	4	5,107904	8,217259
	6	3,274876	4,720881
	8	2,641399	3,364887
	10	2,158164	2,598108
	12	1,825616	2,153661
	14	1,645235	1,869887
	16	1,641561	1,800968
	18	1,530833	1,639984

REFERÊNCIAS

ALONSO, J.; FATICA, M. **Parallel Methods in Numerical Analysis**: Domain Decomposition Methods. Stanford: Aerospace Computing Laboratoy, UMIST, Stanford University, 2002.

ARAUJO, E. R. **Métodos Iterativos em Álgebra Linear Computacional**. Petrópolis - RJ: Laboratório Nacional de Computação Científica - CNPq, 1997.

ARAUJO, E. R. **Métodos Iterativos para a Solução de Grandes Sistemas de Equações em Engenharia Estrutural**. 1997. Tese (Doutorado em Ciência da Computação) — COPPE/UFRJ-Departamento de Engenharia Civil, Rio de Janeiro.

AXELSSON, A. **Iterative Solution Methods**. Cambridge: Cambridge: University Press, 1996. v.1.

BAKER, L.; SMITH, B. J. **Parallel Programming**. New York: McGraw-Hill, 1996. 381p.

BARRETO, M. **DECK**: Um Ambiente para Programação Paralela em Agregados de Multiprocessadores. 2000. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

BENZI, M.; TUMA, M. A Comparative Study of Sparse Approximate Inverse Preconditioners. In: APPLIED NUMERICAL MATHEMATICS: TRANSACTIONS OF OF IMACS, 1999. **Proceedings...** [S.l.: s.n.], 1999. p.305–340.

BUYYA, R. **High Performance Cluster Computing**: Achitecture and Systems. [S.l.]: Prentice Hall, 1999. v.1.

CANAL, A. P. **Paralelização de Métodos de Resolução de Sistemas Lineares Esparsos com o DECK em Clusters de PCs**. 2000. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

CARVALHO, E. C. A. **Particionamento de Grafos de Aplicações e Mapeamento em Grafos de Arquiteturas Heterogêneas**. 2002. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

CATABRIGA, L. **Estudo de Pré-condicionadores para o Método GMRES usando Estrutura de Dados por Arestas**. Rio de Janeiro: Universidade Federal do Rio de Janeiro, 1998.

CHOW, E. T. **Robust Preconditioning for Sparse Linear Systems**. 1997. Tese (Doutorado em Ciência da Computação) — Department of Computer Science, University of Minnesota, Minneapolis - MN.

CRUZ, G. A. M. **Algoritmos Paralelos Iterativos do Tipo Quasi-Newton para a Minimização de Funções Multivariadas**. 1997. Mestrado em Matemática — Instituto de Matemática, UFRGS, Porto Alegre - RS.

CUMINATO, J. A.; MENEGETT, A. **Discretização de Equações Diferenciais Parciais**: Técnicas de Diferenças Finitas. São Paulo: Universidade de São Paulo, 2000.

DONGARRA, J. **Blas - Basic Linear Algebra Subprograms**. Disponível em: <http://www.netlib.org/blas/>. Acesso em: out. 2002.

DONGARRA, J.; LUMSDAINE, A.; POZO, R.; REMINGTON, K. **IML++ - Iterative Methods Library**. Disponível em: <http://math.nist.gov/iml++>. Acesso em: out. 2002.

DORNELES]Dor-01b DORNELES, R. V. **Particionamento de Domínio e Balanceamento de Carga**. [2003?]. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre - RS. (Em Andamento).

DORNELES, R. V.; RIZZI, R. L.; DIVERIO, T. A.; NAVAUX, P. O. A. Refined Schwarz-Krylov Solution for Hydrodynamics and Mass Transport in a PCs Cluster. In: SBAC - SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 13., 2002, Pirenopolis. **Proceedings...** Brasilia: UnB, 2002. v.1, p.17–24.

DREIER, B.; MARKUS, Z.; THEO, U. Parallel and Distributed Programming with Rthreads and Rthreads. In: INTERNATIONAL CONFERENCE ON MASSIVELY PARALLEL COMPUTING SYSTEM, 1998. **Proceedings...** [S.l.: s.n.], 1998. v.1, p.34–40.

FLETCHER, C. A. J. **Computational Techniques for Fluid Dynamics**. New York: Springer-Verlag, 1988. v.1.

GAREY, M. R.; JOHNSON, D. S. **Computer and Intractability**: a Guide to the Theory of NP-completeness. San Francisco: Freeman, 1979.

GROPP, W. D.; KEYES, D. E.; MCINNES, L. C.; TIDRINI, M. D. **Globalized Newton-Krylov-Schwarz Algorithms and Software for Parallel Implicit CFD**. Langley, Hampton: NASA, 1998.

GROPP, W. D.; LUSK, E. **MPICH - A Portable Implementation of MPI**. Disponível em: <http://www-unix.mcs.anl.gov/mpi/mpich/indexold.html>. Acesso em: out. 2002.

JUDICE, J.; PATRICIO, J. M. **Sistemas de Equações Lineares**. Coimbra: Universidade de Coimbra, 1996.

KEYES, D.; VENKATAKRISHNAN, V. **Newton-Krylov-Schwarz Methods**: Interfacing Sparse Linear Solvers with Nonlinear applications. Norfolk: Department of Computer Science, 1997.

KLEIMAN, S.; SHAH, D.; SMAALDERS, B. **Programming With Threads**. [S.l.]: Prentice Hall, 1996. 534p.

LEWIS, B.; BERG, D. J. **Multithreaded Programming With Pthreads**. [S.l.]: Sun Microsystems Press, 1998. 382p.

LUMSDAINE, A.; SQUYRES, J.; BARRETT, B.; SANKARAN, S.; CHABLANI, M.; SAHAY, V. **LAM / MPI Parallel Computin**. Disponível em: <http://www.lam-mpi.org/>. Acesso em: out. 2002.

MARTINOTTO, A. L. **Paralelização de Métodos Numéricos de Resolução de Sistemas Esparsos de Equações Utilizando MPI e de Pthreads**. 2001. Trabalho de Conclusão (Curso de Ciência da Computação) — Universidade de Caxias do Sul, Caxias do Sul.

NOBLE, B.; DANIE, J. W. **Álgebra Linear Aplicada**. Rio de Janeiro: Prentice-Hall, 1986. 378p.

OPENMP: Simple, Portable, Scalable SMP Programming. Disponível em: < http://www.openmp.org/>. Acesso em: out. 2002.

PACHECO, P. S. **Parallel Programming with MPI**. San Francisco: Morgan Kaufmann, 1997. 418p.

PICININ JUNIOR, D. **Paralelização do Algoritmo do Gradiente Conjugado Através da Biblioteca MPI e de Threads**. 2001. Trabalho Individual (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

PICININ JUNIOR, D.; MARTINOTTO, A. L.; RIZZI, R. L.; DORNELES, R. V.; DIVERIO, T. A.; NAVAUX, P. O. A. Parallelizing Conjugate Gradiente Method for Clusters Using MPI and Threads. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, 2002, Las Vegas. **Proceedings...** [Las Vegas]: CSREA, 2002.

PICININ JUNIOR, D.; MARTINOTTO, A. L.; RIZZI, R. L.; DORNELES, R. V.; DIVERIO, T. A.; NAVAUX, P. O. A. Ordenação de Mensagens e Pré-Condicionamento na Solução Paralela do Gradiente Conjugado em Clusters de PCs Multiprocessados. In: WSCAD - WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 3., 2003, Vitória - ES. **Anais...** [S.l.: s.n.], 2003.

PICININ JUNIOR, D.; RIZZI, R. L.; DORNELES, R. V.; DIVERIO, T. A.; NAVAUX, P. O. A. Paralelização de Métodos Iterativos do Subespaço de Krylov em Clusters Multiprocessados. In: CONGRESSO NACIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL, CNMAC, 2001, Belo Horizonte. **Anais...** [S.l.: s.n.], 2001.

RIGONI, E. H.; DIVERIO, T. A.; NAVAUX, P. O. A. **Introdução a Programação em Clusters de Alto Desempenho**. Porto Alegre: PPGC do Instituto de Informática, UFRGS, 1999. (RP-305).

RIZZI, R. L. **Modelo Computacional Paralelo para a Hidrodinâmica e para o Transporte de Massa 2-D e 3-D**. 2001. Proposta de Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

RIZZI, R. L. **Modelo Computacional Paralelo para a Hidrodinâmica e para o Transporte de Massa Bidimensional e Tridimensional**. 2002. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

SAAD, Y. **Iterative Methods for Sparse Linear Systems**. [S.l.]: PWS Publishing Company, 1996.

SAAD, Y. **Iterative Methods for Large Sparse Matrix Problems**: Parallel Implementations. [S.l.]: Brisbane, 2002.

SANTOS, L. L. P.; DORNELES, R. V. **Particionamento de Grafos**. 2001. Trabalho de Conclusão (Curso de Ciência da Computação) — Universidade de Caxias do Sul, Caxias do Sul.

SCHOEGEL, K.; KARYPIS, G.; KUMAR, V. **Graph Partitioning for High Performance Scientific Simulation**. Disponível em: <http://www-users.cs.umn.edu/ karypis/publications/partitioning.htm>. Acesso em: out. 2002.

SHEWCHUK, J. R. **An Introduction to the Conjugate Gradient Method without the Agonizing Pain**. Disponível em: <www.cs.cmu.edu/jrs/jrspapers>. Acesso em: abr. 2001.

SILBERSCHATZ, A.; GALVIN, P. B. **Operating System Concepts**. São Paulo: Prentice Hall, 2000. 696p.

SMITH, B.; BALAY, S.; KNEPLEY, M.; ZHANG, H.; BUSCHELMAN, K. **The Portable, Extensible Toolkit for Scientific Computation**. Disponível em: < http://www-fp.mcs.anl.gov/petsc>. Acesso em: ago. 2002.

SMITH, B.; BJORSTAD, P.; GROPP, W. **Domain Decomposition**: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge: Cambridge University Pres, 1996.

TEODORO, A. **Geração Automatizada de Malhas**. 1998. Dissertação (Mestrado em Matemática) — USP - São Carlos, São Carlos - SP.

TUMINARO, R. S.; SHADID, J. N.; HEROUX, M. **A Massively Parallel Iterative Solver Library for Solving Sparse Linear Systems**. Disponível em: <http://www.cs.sandia.gov/CRF/aztec1.html>. Acesso em: jul. 2002.