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

RAFAEL ENNES SILVA

Escalonamento Estático de *Programas-MPI*

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

Prof. Dr. Tiarajú Diverio
Orientador

Prof. Dr. Nicolas Maillard
Co-orientador

Porto Alegre, Maio de 2006

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silva, Rafael Ennes

Escalonamento Estático de *Programas-MPI* / Rafael Ennes Silva. – Porto Alegre: PPGC da UFRGS, 2006.

79 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2006. Orientador: Tiarajú Diverio; Co-orientador: Nicolas Maillard.

1. Balanceamento de Carga. 2. Particionamento de Grafos. 3. Mapeamento de Processos. 4. Biblioteca β MPI. I. Diverio, Tiarajú. II. Maillard, Nicolas. III. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

AGRADECIMENTOS

Agradeço a todas as pessoas que contribuíram para a realização deste trabalho. Em especial meu orientador, Prof. Tiarajú Diverio e a meu coorientador, Prof. Nicolas Maillard pela confiança depositada em mim para desenvolver este trabalho, por acreditarem no meu potencial e pelo tempo que eles se dedicaram a minha orientação.

Aos meus pais, que desde o início me incentivam em todas as minhas escolhas. Gostaria de deixar registrada a enorme admiração que tenho por eles e o agradecimento pelo apoio durante todos esses anos.

À Sinara pelo amor, carinho, compreensão e apoio durante esse período. Deixo aqui também a minha gratidão pela sua paciência.

Aos professores do Instituto de Informática pelos ensinamentos e aos funcionarios pela colaboração e ótimo atendimento.

Gostaria de deixar registrado o agradecimento aos colegas do II pelo apoio nas horas de trabalho e pela amizade nos momentos de descontração. Em especial aos amigos e colegas do GMCPAD, CluMSSy e GPPD que tornaram esse momento muito agradável, tanto no Instituto quanto nas viagens a Congressos, churrascos, bares, padarias, lancherias e butekos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

APN	Arbitrary Processor Network
BLCR	Berkley Linux Checkpoint/Restart
BNP	Bounded Number of Processors
DAG	Directed Acyclic Graph
DCP	Dynamic Critical Path
DFG	Data Flow Graph
DSC	Dominant Sequence Algorithm
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
HPC	High Performance Computing
HPL	Highly Parallel Linpack
KL/FM	Kernighan-Lin/Fiduccia-Matteyses
LAM	Local Area Multicomputer
MIT	Massachussetts Institute of Technology
MPE	Multiprocessor Environment
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
PAD	Processamento de Alto Desempenho
RMA	Remote Memory Access
RPI	Request Progression Interface
SD	Sequence Dominant Algorithm
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multi-Processor
SPMD	Single Program Multiple Data
TP	Tempo Paralelo
UNC	Unbounded Number of Clusters

LISTA DE SÍMBOLOS

G	grafo
V	vértices
A	arestas
k	número de componentes da transformada de Fourier
L	índice da componente da transformada de Fourier
m	número de dimensões da transformada de Fourier
N	tamanho da transformada de Fourier
P	número de processos
Vol	volume de dados trafegados
Ω	subdomínio
ε	erro máximo para solução
u_i	solução número i de um problema
Φ	número pequeno diferente de zero
N_B	tamanho do subbloco de uma matriz de tamanho N no HPL
P$\times$Q	grade de processos utilizada no HPL

LISTA DE FIGURAS

RESUMO

O bom desempenho de uma aplicação paralela é obtido conforme o modo como as técnicas de paralelização são empregadas. Para utilizar essas técnicas, é preciso encontrar uma forma adequada de extrair o paralelismo. Esta extração pode ser feita através de um grafo representativo da aplicação.

Neste trabalho são aplicados métodos de particionamento de grafos para otimizar as comunicações entre os processos que fazem parte de uma computação paralela. Nesse contexto, a alocação dos processos almeja minimizar a quantidade de comunicações entre processadores. Esta técnica é frequentemente adotada em Processamento de Alto Desempenho - PAD. No entanto, a construção de grafo geralmente está embutida no programa, cujas estruturas de dados privadas são empregadas na contrução do grafo. A proposta é usar ferramentas diretamente em programas MPI, empregando, apenas, os recursos padrões da norma MPI 1.2. O objetivo é fornecer uma biblioteca (β-MPI) portável para o escalonamento estático de programas MPI. O escalonamento estático realizado pela biblioteca é feito através do mapeamento de processos. Esse mapeamento busca agrupar os processos que trocam muitas informações em um mesma máquina, o que nesse caso diminui o volume de dados trafegados pela rede. O mapeamento será realizado estaticamente após uma execução prévia do programa MPI.

As aplicações alvo para o uso da β-MPI são aquelas que mantêm o mesmo padrão de comunicação após execuções sucessivas. A validação da biblioteca foi realizada através da Transformada Rápida de Fourier disponível no pacote FFTW, da resolução do Problema de Transferência de Calor através do Método de Schwarz e Multigrid e da Fatoração LU implementada no *benchmark* HPL. Os resultados mostraram que a β-MPI pode ser utilizada para distribuir os processos eficientemente minimizando o volume de mensagens trafegadas pela rede.

Palavras-chave: Balanceamento de Carga, Particionamento de Grafos, Mapeamento de Processos, Biblioteca β MPI.

ABSTRACT

A good performance of a parallel application is obtained according to the mode as the parallelization techniques are applied. To make use of these techniques, is necessary to find an appropriate way to extract the parallelism. This extraction can be done through a representative graph of the application.

In this work, methods of partitioning graphs are applied to optimize the communication between processes that belong to a parallel computation. In this context, the processes allocation aims to minimize the communication amount between processors. This technique is frequently adopted in High Performance Computing - HPC. However, the graph building is generally inside the program, that has private data structures employed in the graph building. The proposal is to utilize tools directly in MPI programs, employing only standard resources of the MPI 1.2 norm. The goal is to provide a portable library (β-MPI) to static schedule MPI programs. The static scheduling realized by the library is done through the mapping of processes. This mapping seeks to cluster the processes that exchange a lot of information in the same machine that, in this case decreases the data volume passed through the net. The mapping will be done staticly after a previous execution of a MPI program.

The target applications to make use of β-MPI are those whose keep the same communication pattern after successives executions. The library validation is done through the available applications in the FFTW package, the solving of the problem of Heat Transference through the Additive Schwarz Method and Multigrid and the LU factorization implemented in the HPL benchmark. The results show that β-MPI can be utilized to distribute the processes efficiently minimizing the volume of messages exchanged through the network.

Keywords: Load Balancing, Graph Partitioning, Mapping of Processes, β-MPI Library.

1 INTRODUÇÃO

Existem problemas que precisam de uma grande quantidade de processamento para que suas soluções sejam encontradas. Este processamento, quando executado de forma sequencial pode gerar um atraso significativo para que as informações sejam obtidas, por exemplo, no caso da previsão meteorológica, na qual depois de o fenômeno ter acontecido, a solução não é mais útil. Para tratar este tipo de problema sugere-se a utilização do processamento de forma paralela ou distribuída. Isto consiste, conceitualmente, em dividir tarefas grandes e complexas em tarefas pequenas, as quais podem ser executadas independentemente. Esta divisão possibilita que uma dada aplicação tenha, em muitos casos, um tempo de execução em paralelo menor do que o tempo sequencial, melhorando o seu desempenho.

Muitas aplicações paralelas utilizam ferramentas de troca de mensagens em um contexto de arquitetura distribuída. O padrão mundialmente conhecido para troca de mensagens é o MPI (PACHECO, 2001; THE MPI MESSAGE PASSING INTERFACE STANDARD, 1995). Neste padrão, não é previsto um mecanismo de escalonamento de tarefas, o qual fica destinado ao programador definir como será realizada a distribuição das tarefas de acordo com os recursos disponíveis.

O bom desempenho de uma aplicação pode ser atingido conforme o modo como as técnicas de paralelização são empregadas. Para isto acontecer, é preciso encontrar uma forma de extrair o paralelismo. Esta extração pode ser feita usando-se um grafo representativo da aplicação. No grafo considera-se que os vértices representam o processamento e as arestas indicam as comunicações entre os vértices. Através das características do grafo, identificam-se as dependências e as simultaneidades das tarefas para a resolução da aplicação.

O desempenho do programa também depende da máquina paralela utilizada, ou seja, do tipo de *hardware* empregado. Algumas informações que podem ser obtidas a partir dele são: as características de rede; a comunicação entre os processadores e sobre o tempo de acesso e tipo da memória.

Tendo-se a modelagem da aplicação e do *hardware* empregado, é preciso mapear o grafo da aplicação sobre o grafo da arquitetura. Mapear um grafo significa informar onde cada tarefa será disparada como mostra a Figura 1.1. Para realizar o mapeamento deve-se levar em conta a escolha de uma estratégia eficiente para o particionamento do grafo da aplicação. Algumas ferramentas foram desenvolvidas para este fim como Jostle (WALSHAW, 2002) e Metis (KARYPIS; KUMAR, 1995).

A proposta deste trabalho é fornecer uma biblioteca que produza automaticamente o grafo de fluxo de dados de programas MPI e com base nas informações de comunicação entre os processos (arestas), sejam tomadas as decisões de escalonamento. Pelo escalonamento define-se onde e quando uma tarefa será executada. A ferramenta pro-

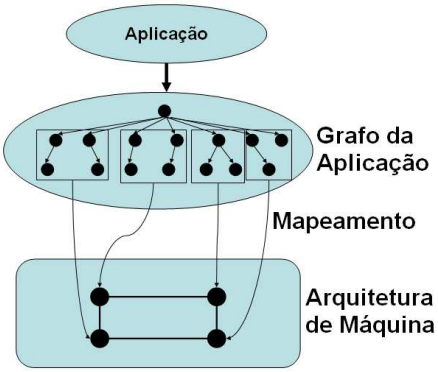

Figura 1.1: Extração e mapeamento do grafo.

posta vai tratar somente da localização de execução das tarefas, ou seja, do mapeamento destas sobre a arquitetura utilizada. Esse mapeamento será realizado estaticamente após uma execução prévia do programa MPI. O objetivo será encontrar o melhor mapeamento possível entre software e hardware de forma a diminuir a comunicação via rede . Após o mapeamento das tarefas, a aplicação pode ser executada novamente seguindo a ordem proposta pelo escalonador.

Como esta biblioteca para o remapeamento de tarefas de programas MPI será estática, o escopo de aplicações no qual ela poderá ser empregada são aquelas que mantêm o mesmo padrão de comunicação, após execuções sucessivas e, não tenham variações de entrada de dados. Exemplo de aplicações que seguem esta linha são: a fatoração LU (DONGARRA; LUSZCZEK; PETITET, 2001), Transformada Rápida de Fourier (FFT) (FASTEST FOURIER TRANSFORM IN THE WEST, 2003), cálculo do gradiente conjugado (CG) (BAILEY et al., 1991), etc.

1.1 Motivação da Pesquisa

A presente pesquisa é motivada pela larga utilização de troca de mensagens através do MPI em aplicações na área de PAD. Nesse contexto, vêm sendo desenvolvidas pesquisas por dois grupos no Instituto de Informática da UFRGS: pelo Grupo de Matemática da Computação de Alto Desempenho (GMC-PAD), através da modelagem de fenômenos físicos e desenvolvimento de aplicações, que utilizam MPI como o Modelo Unihidra (DORNELES, 2003); e do grupo *Cluster Management Scheduling System* (CluMSSy) no qual são feitas pesquisas na área de escalonamento de tarefas em programas MPI.

A concretização da proposta da biblioteca de escalonamento estático de programas MPI pode beneficiar alguns trabalhos do GMCPAD como o desenvolvido por Delcino entitulado Paralelização de Métodos Numéricos em Clusters Empregando as Bibliotecas MPICH, DECK e Pthreads", onde foi desenvolvida a paralelização do Método do Gradiente Conjugado GC e do Método do Resíduo Mínimo Generalizado GMRES, (PICININ, 2003). Para o grupo CluMSSy, este trabalho poderá ser integrado com a linha de pesquisa que trata da migração de processos usando a β-MPI, a qual permitirá aumentar a gama de aplicações que poderão utilizar a β-MPI.

Este trabalho tem como motivação auxiliar usuários não-especialistas no escalonamento de tarefas de uma aplicação. A biblioteca β-MPI vai auxiliar na adaptação de um programa MPI a máquinas, sem que o usuário precise reprogramar o código, mantendo o foco no principal interesse: o desempenho do programa.

1.2 Apresentação do Texto

Esta dissertação está organizada em sete capítulos. O capítulo inicial apresenta a introdução e motivação de pesquisa do tema proposto.

O segundo capítulo chama-se Representação de Programas Paralelos com Grafos e Escalonamento. O capítulo começa contextualizando os modelos em programação paralela - memória compartilhada e troca de mensagens e trata da modelagem de aplicações paralelas através do uso de grafos. Depois são apresentadas três ferramentas que utilizam grafos para representar tais aplicações. Essas ferramentas usam mecanismos de escalonamento que tomam suas decisões baseados nas informações do grafo.

O terceiro capítulo apresenta o MPI e suas características gerais. Esse capítulo começa abordando os conceitos básicos do MPI previstos na norma 1.2, modos de comunicação, depuração no MPI e migração de processos. Depois comenta-se sobre as distribuições MPICH e LAM/MPI. No resumo do capítulo menciona-se a implementação Open-MPI, a qual objetiva integrar implementações MPI, porém não possui uma versão estável atualmente.

O capítulo 4 trata da proposta da biblioteca de escalonamento estático para programas MPI - chamada β-MPI (Biblioteca de Escalonamento de TArefas). Essa biblioteca utiliza, como estratégia de redistribuição de tarefas, a análise do grafo de fluxo de dados (DFG) de uma aplicação e tem como objetivo diminuir o tempo de comunicação entre os processos. Essas características são apresentadas nas primeiras seções do capítulo. No final comparam-se as ferramentas apresentadas no segundo capítulo com a β-MPI em termos de características de funcionamento.

O capítulo 5 apresenta duas aplicações usadas para validar a β-MPI: a FFT e a resolução do problema de Transferência de Calor sobre uma placa plana utilizando o método de Schwarz e multigrid. A primeira parte do capítulo aborda os conceitos da transformada discreta de Fourier, apresenta a implementação paralela FFTW e uma avaliação experimental através dos programas de *benchmark* fornecidos pela FFTW. A segunda parte apresenta a solução do problema de Transferência de Calor sobre uma placa plana, a forma como foi montada a estratégia de solução e uma avaliação experimental com a β-MPI.

O capítulo 6 apresenta a fatoração LU tal como implementada no *Highly Parallel Linpack* (HPL). O capítulo começa abordando o algoritmo de fatoração LU sequencial. A segunda seção situa o leitor sobre o HPL mostrando as características do *benchmark*. Segue-se uma seção que trata da instalação e otimização de uso do HPL. As seções seguintes são aquelas que tratam do uso do HPL com a β-MPI e discutem desde a geração do grafo até o remapeamento de processos.

Finalmente, são apresentadas as conclusões. Esse capítulo também enfatiza as contribuições do trabalho assim como sugestões para trabalhos futuros.

2 REPRESENTAÇÃO DE PROGRAMAS PARALELOS COM GRAFOS E ESCALONAMENTO

A representação de programas paralelos através de grafos é frequentemente utilizada para identificar o comportamento de uma aplicação em termos de processamento e comunicação. Existem diversos tipos de grafos que podem ser utilizados na representação de um programa paralelo, tais como grafo acíclico, árvores, grafos orientados, etc.

O tipo de grafo empregado para representar uma aplicação também considera o modelo de programação adotado. Por exemplo, em uma aplicação que utiliza memória compartilhada, dividem-se tarefas entre os processos para que ao final um resultado seja alcançado - dividir para conquistar. Neste caso, pode-se usar um grafo orientado para representar os processos e suas tarefas dependentes que serão realizadas na execução. Em contexto de troca de mensagens, pode-se ter um grafo não-orientado acíclico para a representação da troca de mensagens entre os processos vizinhos.

A abordagem de representação de programas paralelos através de grafos tem como objetivo identificar os pontos críticos de uma aplicação, tanto de volume de processamento quanto de volume de troca de informações. Essa identificação é de fundamental importância para que o emprego de técnicas de escalonamento sejam aplicadas eficientemente. O escalonamento de um programa pode ser classificado em estático ou dinâmico, conforme o momento no qual ele é empregado. No escalonamento estático, as decisões são tomadas antes da execução da aplicação. Já no escalonamento dinâmico, tais decisões são feitas durante a execução (CASAVANT; KUHL, 1988).

No contexto de programas que utilizam troca de mensagens, ainda não está disponível um mecanismo de escalonamento estático para aplicações MPI. Por outro lado, existem ferramentas que usam o escalonamento estático para aplicações baseadas nos modelos de memória compartilhada e/ou troca de mensagens.

2.1 Modelos de Comunicação

Os modelos de comunicação utilizados em arquiteturas de alto desempenho têm sido alvo de pesquisas buscando explorar melhor tais arquiteturas. Há duas formas de explorar o paralelismo - a primeira é via memória compartilhada e a segunda é através de troca de mensagens.

O modelo de comunicação baseado em memória compartilhada é adequado para arquiteturas SMP. O aproveitamento do acesso à memória comum é realizado através da criação de fluxos concorrentes de execução. Esse modelo pode ser aplicado em agregados de computadores para uma exploração eficiente do paralelismo intra-nodo, sendo cada nodo composto por mais de um processador. O gargalo de comunicação nesta abordagem

está relacionado à quantidade de processos ou fluxos de execução que serão disparados, pois é necessário controlar o acesso aos dados compartilhados.

O modelo baseado em troca de mensagens é adequado para arquiteturas nas quais cada processador possui uma memória própria (memória distribuída) e a comunicação acontece através de uma rede dedicada. Nesse modelo cada processo disparado em um processador realiza as suas atividades e, quando necessário, envia ou recebe informações pela rede. O emprego deste modelo é conveniente para a exploração inter-nodos em um agregado. A utilização de troca de mensagens é limitada pela velocidade de comunicação da rede utilizada no agregado.

Estes dois tipos de modelos de comunicação são largamente utilizados no processamento de alto desempenho (PAD). Para o desenvolvimento de uma aplicação em PAD estes modelos podem ser expressos através de grafos, permitindo dessa forma a identificação dos pontos de comunicação e do processamento do programa.

2.2 Modelagem através de Grafos

Um grafo é uma estrutura G=(V,E) onde V é um conjunto de vértices ou nodos e E representa as arestas ou arcos do grafo (NETTO, 2001) os quais são interligados formando os caminhos. Um grafo pode ser orientado se as arestas têm um sentido definido entre os vértices. Por outro lado, se o sentido não for definido, diz-se que o grafo é não-orientado. Outra característica é ser cíclico ou não. Um grafo é dito cíclico quando ele possui ao menos um ciclo, ou seja, um mesmo vértice é a origem e o destino de um caminho - rota estabelecida entre vértices - onde cada caminho tem que ter no mínimo 3 arestas (NONATO, 2000). Um grafo acíclico caracteriza-se por não ter ciclos.

O conceito de tarefa também deve ser considerado quando se trata de modelagem com grafos. Uma tarefa pode ser definida como uma unidade de procedimento indivísivel, a ser executado seqüencialmente, definido por suas entradas, saídas e instruções

Na modelagem de aplicações paralelas através de grafos, existem três tipos de grafos que são os mais utilizados. Eles são: o grafo de tarefas, o grafo de dependências e o grafo de fluxo de dados. O grafo de tarefas (REWINI, ????), é definido como um DAG, onde um arco(i,j) entre tarefas Ti e Tj especifica que Ti deve ser terminada antes de Tj começar.

O grafo de dependências (ROOSTA, 2000), é uma noção que vem da área de compiladores, e enfatiza as dependências, seja de controle, seja de dados, entre as tarefas. Quando se trata de dependência de dados, um arco entre tarefas Ti e Tj significa que um dado produzido por Ti será necessário à execução de Tj.

O grafo de fluxo de dados (JAGANNATHAN, ????), enfatiza o volume de dados a serem trocados, além de sua fonte e destino: um grafo de fluxo de dados é um grafo direto no qual os vértices representam as operações e as arestas denotam as dependências entre operações. Valores produzidos e consumidos pelos vértices são carregados em tokens, os quais fluem ao longo das arestas

Neste trabalho, sera usado um modelo mais simples, porém adaptado aos programas MPI estáticos contemplados: o grafo de comunicações entre as tarefas. Ele será obtido a partir do Grafo de Fluxo de dados entre os processos (conforme explicado no capítulo 4) e considerado como uma representação simplificada do mesmo. Por esse motivo, será chamado no resto deste documento de DFG, apesar de ser conceitualmente diferente.

Em programas paralelos, o grafo precisa ser dividido para que as tarefas sejam distribuídas pela arquitetura adotada. Esta divisão (particionamento) tem que buscar um bom aproveitamento dos recursos alocados através de um balanceamento de carga eficiente.

2.2.1 Problema do Particionamento de Grafos

Segundo (SCHLOEGEL; KARYPIS; KUMAR, ????), dado um grafo $G = (V,E)$ valorado[1] e não orientado, para o qual cada vértice e aresta têm um peso associado, o problema de particionamento de grafo é dividir os vértices V em subdomínios ou partições, onde cada partição tem aproximadamente a mesma quantia de vértices. Os vértices representam o volume de dados a serem computados e as arestas representam o número de comunicações entre os subdomínios. O conjunto de arestas que conectam dois subdomínios diferentes é conhecido como corte de arestas. Uma aproximação é utilizar o termo diminuir o corte de arestas no sentido de minimizar o volume de comunicação entre processadores. Esta aproximação é muito utilizada em ferramentas de particionamento

O problema de particionamento de grafos é definido como sendo NP-completo. Portanto não é possível chegar a uma solução ótima (um algoritmo determinístico) para o cálculo do particionamento de grafos em um tempo razoável (polinomial), apenas se encontra uma boa aproximação da mesma. Este fato combinado com a natureza do problema, tem levado ao desenvolvimento de diversos algoritmos (SCHLOEGEL; KARYPIS; KUMAR, ????).

Um grafo pode ser usado para representar uma aplicação que utiliza memória compartilhada como meio de comunicação. Uma situação comum de grafos é o caso "dividir para conquistar"(ANDREWS, 1991), onde a partir de um fluxo de programa inicial criam-se fluxos de execução intermediários que vão sendo processados paralelamente para que eles converjam a um resultado final. O grafo de tarefas representa perfeitamente esta situação e um exemplo de ferramenta que adota este tipo de grafo, porém no contexto de memória compartilhada, é o Cilk (LEISERSON, 2001) .

Já em um ambiente de troca de mensagens, o grafo modela a aplicação, sendo o processamento representado pelos vértices e as mensagens trafegadas pela rede representadas pelas arestas. O peso das arestas indica o volume de mensagens trafegadas e o peso dos vértices é obtido pelo volume de processamento. Alguns ambientes que usam troca de mensagens e grafos são o Pyrros (YANG; GERASOULIS, 1992) e o CASCH (AHMAD et al., 2000).

2.3 Cilk

Cilk (LEISERSON, 2001) é um ambiente para programação com múltiplos fluxos concorrentes de execução (*multithreading*) baseado na linguagem ANSI C desenvolvido pelo Cilk Group no MIT. Ele foi projetado com propósitos gerais de programação paralela, mas foi especialmente concebido para explorar o paralelismo assíncrono e dinâmico em sistemas que suportam Posix Threads. A filosofia do Cilk é deixar o programador tratar a estruturação do programa para explorar o paralelismo e a localidade (grau de dependência das operações realizadas pelo processador sobre dados remotos) e o ambiente se encarrega do escalonamento para uma exploração eficiente da plataforma adotada. O pacote Cilk é composto por um ambiente de execução, um compilador e um conjunto de programas exemplo.

O ambiente de execução do Cilk tem como objetivo explorar eficientemente a plataforma SMP (com suporte à memória compartilhada) através do tratamento de questões relacionadas ao balanceamento de carga, paginação e comunicação. O Cilk destaca-se

[1] grafo é valorado quando existe ao menos uma função que relaciona os vértices e/ou as arestas com conjunto de números (CARVALHO, 2002).

em relação a outras linguagens baseadas em múltiplos fluxos de execução por garantir a predição e a eficiência de desempenho de um programa.

O Cilk utiliza um compilador chamado de *source-to-source* (cilk2c) que traduz um código em Cilk para um código em C. O código em C é compilado com o compilador gcc e o arquivo objeto gerado é ligado com o ambiente de execução Cilk e depois o executável é gerado.

A execução de programas Cilk pode ser visualizada através de um grafo acíclico dirigido (*DAG*), vide Figura 2.1 (LEISERSON, 2001). Um programa Cilk tem como característica o uso de 3 palavras-chave: *cilk*, *spawn* e *sync*. A palavra-chave *cilk* identifica um procedimento Cilk, o qual deve gerar sub-procedimentos em paralelo e sincronizá-los ao final. Os sub-procedimentos são threads não-bloqueantes disparadas pela palavra-chave *spawn* e sincronizadas pela palavra-chave *sync*.

Na execução de um programa Cilk, um procedimento será resolvido através de uma ou mais threads. Uma thread só poderá ser executada depois que todas as threads, da qual ela depende, tiverem sido completadas. As tomadas de decisões do escalonador serve como base para formar o *DAG* de um programa. O mecanismo de escalonamento utilizado é o roubo de tarefas *work-stealing*. Neste mecanismo, durante a execução, quando um processo acaba o seu trabalho ele pede a outro processo, escolhido randomicamente, mais tarefas para executar.

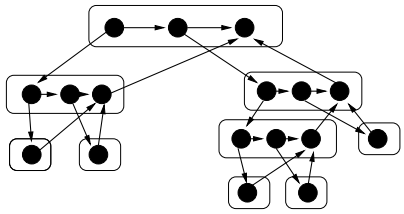

Figura 2.1: Modelo de processamento do Cilk.

Um programa Cilk pode ser escalonado seguindo um dos dois tipos de algoritmos - *nanoscheduling* ou *microscheduling*. O algoritmo *nanoscheduling* é utilizado para o escalonamento de procedimentos com um único processo. O algoritmo *microscheduling* trata do escalonamento através de um conjunto de processadores.

O algoritmo do *nanoscheduler* tem como característica a tomada de decisões rápida é o seguinte: ele executa uma tarefa e suas sub-tarefas na mesma ordem pela qual elas seriam executadas em C. O objetivo deste escalonador é garantir que, em um processo, o programa Cilk seja executado na mesma ordem que em um programa C.

O algoritmo *microscheduling* é usado para o escalonamento de procedimentos com um conjunto de processos. O *microscheduler* utiliza o escalonamento baseado através do roubo de tarefas *work-stealing*. Quando um processo acaba de processar a sua carga de trabalho, este rouba trabalho de um outro processo escolhido randomicamente. A vítima tem uma série de registros em uma fila para serem processados. O registro que estiver no topo da fila é roubado pelo processo ladrão. Depois o processo entrega a correspondente tarefa para o *nanoscheduler* executar.

Estes algoritmos de escalonamento foram utilizados em alguns protótipos de aplicações que foram desenvolvidos utilizando a linguagem Cilk. Dentre eles estão a renderização de imagens, desdobramento de proteínas (*protein folding*), procura por retrocesso (*backtracking search*), simulação de sistemas N-corpos e o processamento de matrizes densas e esparsas. Uma série de outras aplicações como a fatoração de Choleski, cálculo de Fibonacci e FFT estão disponibilizadas com o pacote Cilk-5.2.

O Cilk restringe o grafo de fluxo de dados da aplicação a um único tipo: o "divisão e conquista"por causa do algoritmo de escalonamento *microscheduling*. Apesar do uso de grafo na representação de suas aplicações, o ambiente Cilk foi concebido para aplicações no contexto de múltiplos fluxos concorrentes (memória compartilhada). Neste trabalho serão tratadas questões relativas ao emprego do DFG em ferramentas que utilizam o modelo de troca de mensagens para comunicação.

2.4 Pyrros

O Pyrros (YANG; GERASOULIS, 1992) é um sistema desenvolvido para atingir dois objetivos: fornecer um mecanismo de escalonamento estático automático e gerar um código para arquiteturas que utilizam troca de mensagens. Especificamente, este sistema foi concebido para arquiteturas nCUBE-1 nCUBE-2 e INTEL- iPSC/2. A motivação para o desenvolvimento deste sistema foi a falta de uma ferramenta para paralelização automática e geração de código otimizado para arquiteturas específicas. Isto é uma vantagem frente a um programa descrito manualmente, já que é preciso levar em conta as questões de escalonamento.

O Pyrros parte da descrição paralela de um grafo de fluxo de dados com suas restrições de precedências. Esta descrição do grafo é feita pelo usuário através de uma linguagem específica para isso. Depois o grafo passa por um escalonador que define o mapeamento das tarefas e por último produz, como saída, o código do programa baseado em troca de mensagens. A organização do sistema está ilustrada na Figura 2.2.

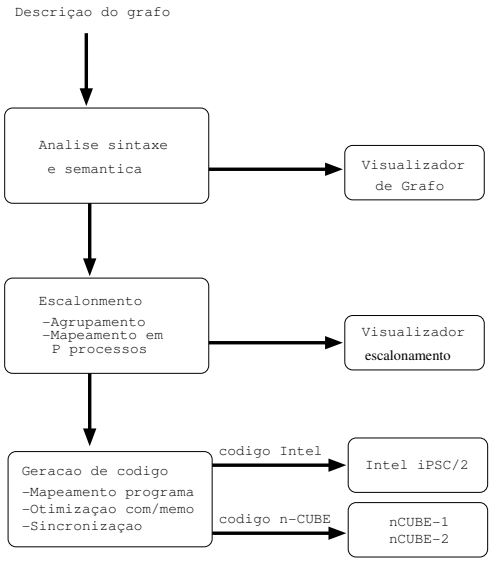

Figura 2.2: Organização Pyrros.

O sistema recebe a descrição de tarefas do programa que passa pelo analisador de sintaxe e semântica e permite a visualização do grafo de tarefas. O grafo é entregue ao módulo que aplica as políticas de escalonamento. Neste módulo, também, podem-se visualizar as tomadas de decisões do escalonador. Enfim, é realizada a geração do código e otimizações de comunicação e uso de memória. Como saída tem-se códigos de troca de mensagens para as arquiteturas nCUBE e INTEL iPSC/2.

2.4.1 Escalonamento

O escalonamento no Pyrros é realizado em dois passos: o (*Clustering*), definido como determinação de tarefas para um agrupamento, as quais serão executadas pelo mesmo processador e o escalonamento de um grafo agrupado em p processadores.

O agrupamento de tarefas para um mesmo processador é feito com base no algoritmo *Dominant Sequence Algorithm* (DSC)(YANG; GERASOULIS, 1992), o qual tem como objetivo reduzir o tempo paralelo (TP) em cada passo, refinando o agrupamento através de um método que zera as arestas [2] do caminho mais longo do grafo de precedência, dito seqüência dominante (SD). O algoritmo começa considerando todas as tarefas como sendo um agrupamento. Em seguida é identificada uma SD e todas as arestas do grafo são ditas não-examinadas. Enquanto houver arestas não examinadas na SD, será escolhida uma aresta para ser zerada desde que ela não aumente o TP. A aresta é marcada como examinada e uma nova SD é procurada.

Quanto à complexidade de tempo do algoritmo DSC, ela é praticamente linear $O(|v+e|\log v)$, onde v e e representam o número de vértices e de arestas do grafo, respectivamente. Para que a complexidade seja mínima a idéia é tentar encontrar as tarefas da SD, incrementalmente, de um passo para outro em um tempo $O(\log v)$.

O escalonamento de um grafo agrupado em p processadores segue três estágios: combinar os agrupamentos com os processos; mapear os processos em processadores; ordenar a execução das tarefas em cada processador. A distribuição de agrupamentos é feita verificando a carga de cada agrupamento, organizando-os em ordem crescente de carga e mapeando-os de forma a aproximar o balanceamento entre os processadores.

2.4.2 Geração de Código

A geração de um programa pelo Pyrros inclui a geração de processamento e de comunicação. O código gerado pela ferramenta Pyrros utiliza o modelo de troca de mensagens para comunicação. As primitivas são o *send* não-bloqueante e o *receive* bloqueante.

Dado o número de nodos que serão utilizados, um deles é denominado *host*, no qual o programa Pyrros *host* (principal) é executado. Este programa tem como objetivo alocar o número de processadores necessários, distribuir os programas, enviar os dados iniciais e coletar os resultados finais.

A qualidade do código gerado é dada pela otimização da comunicação e do uso de memória. Para otimizar a comunicação, o Pyrros precisa eliminar as operações de comunicação desnecessárias. Por exemplo, uma mensagem poderia ser enviada para muitos processadores através do envio repetitivo de um-para-um, causando uma contenção na rede. O Pyrros trata este problema utilizando algoritmos de *broadcasts* para otimizar o uso da topologia de rede adotada. Quanto ao uso da memória, à medida que os dados vão sendo utilizados, o espaço de memória vai sendo liberado.

Um exemplo de programa implementado, seguindo a sistemática do Pyrros para a geração de código, é o algoritmo de Gauss-Jordan sem pivoteamento apresentado em (YANG; GERASOULIS, 1992). Este programa é utilizado para a resolução de equações lineares e as rotinas da Blas nível 3 (na Blas nível 3 o alvo são operações de matriz-matriz) do Linpack foram usadas na implementação. Os testes realizados levaram em conta a comparação da geração do código automaticamente e da geração do código feita de forma manual. A arquitetura disponível foi a nCUBE-2. O desempenho final da aplicação foi similar para as duas situações.

[2] neste algoritmo uma aresta zerada entre duas tarefas significa que elas estão agrupadas

2.5 Computer Aided Scheduling - CASCH

A ferramenta CASCH (AHMAD et al., 2000) foi projetada para ser um ambiente completo para programação paralela, porque ela fornece mecanismos de paralelização de código, escalonamento, comunicação, sincronização e geração de código. Para utilizar esta ferramenta, o usuário primeiro escreve um programa sequencial, o qual será composto por um conjunto de funções. Essas funções no programa serial tratam problemas com tamanho definido pelo usuário, por exemplo, uma matriz de dados com tamanho $N = PN \times SN$, onde as constantes PN e SN definem a grade para o processamento paralelo e controlam a granularidade do particionamento. O programa sequencial serve de base para montagem do grafo de tarefas orientado (DAG), o qual através da ordem de chamada das funções tem a dependência de tarefas definida. A organização da ferramenta CASCH é composta pelos seguintes componentes:

- um analisador de sintaxe e semântica;
- gerador de grafos;
- um recurso de estimação de pesos;
- ferramenta de escalonamento;
- um recurso que insere comunicação e gera código paralelizado;
- uma interface gráfica;
- teste do programa paralelo e análise de desempenho.

O analisador de sintaxe e semântica verifica se existe algum erro e notifica ao usuário antes de passar para a geração do grafo. O usuário sabe previamente o número de rotinas existentes no programa, as quais serão usadas para montar o grafo de fluxo de dados. Para o gerador de grafos, as rotinas representam as tarefas e o tempo de execução delas representa o peso. A passagem de um parâmetro entre duas funções representa uma mensagem (aresta) e a duração dessa passagem indica o peso da aresta. Os pesos dos vértices e arestas do grafo são calculados por um recurso de estimação através de um *benchmarking*, que coleta informações de processamento e comunicação e armazena em um banco de dados. Esta base de dados contém informações de processamento e comunicação de diferentes máquinas.

Os algoritmos de escalonamento utilizados pelo CASCH são organizados em diferentes categorias podendo ser aplicados a diferentes arquiteturas. Estes algoritmos são classificados pelo sistema como os dedicados a um número ilimitado de clusters *Unbounded Number of Clusters* (UNC), a processadores totalmente conectados *Bounded Number of Processors* (BNP) e a processadores de redes arbitrárias *Arbitrary Processor Network* (APN). Nos casos dos algoritmos UNC e BNP, assume-se que a rede considerada é totalmente conectada e as estratégias de roteamento da rede são desconsideradas.

O escalonamento é feito *off-line* durante a compilação, o que caracteriza um escalonamento estático. Uma característica marcante desta ferramenta é uma interface gráfica que permite analisar diferentes algoritmos de escalonamento usando DFGs gerados dos programas. O melhor algoritmo de escalonamento é escolhido pelo usuário e passado para um gerador de códigos paralelos para uma dada máquina.

O gerador de código utiliza um recurso de inserção de comunicação entre as tarefas através de primitivas de troca de mensagens *sends* e *receives*. Depois do escalonamento, CASCH aloca cada tarefa para um processador conforme a ordem de prioridades destas tarefas. Se existe uma aresta de saída, de uma tarefa pertencente a um processador para uma outra que está em outro processador, então é inserida uma primitiva *send* após a tarefa. De maneira análoga, se a aresta estiver chegando de um outro processador, antes da tarefa é colocada uma primitiva *receive*. Para garantir que não vai ocorrer *deadlocks*, o sistema organiza a ordem de comunicação sempre mantendo uma primitiva *send* antes de um *receive*.

Depois do código paralelo ser gerado, o programa pode ser testado e as informações de tempo de execução analisadas. CASCH permite que esta analise seja feita por partes do programa. Ao final o usuário pode repetir todo o processo, caso o desempenho não tenha sido satisfatório.

Algumas aplicações foram paralelizadas pela CASCH. Algumas delas são a *Fast Fourier Transform* (FFT), eliminação de Gauss, aplicações N-corpos e a resolução de equações de Laplace (AHMAD et al., 2000). Essas aplicações foram executadas nas máquinas Intel Paragon e a IBM SP2. A Intel Paragon é formada por 140 processadores i386/XP com frequência de 50MHz enquanto que a IBM SP2 possui 48 processadores IBM P2SC de 160MHz. Os resultados mostraram a viabilidade e a utilidade da ferramenta CASCH.

A ferramenta CASCH utiliza troca de mensagens e algoritmos de escalonamento conhecidos. Porém esses algoritmos são compatíveis apenas com DAGs e na proposta desse trabalho, leva-se em conta DFGs conforme apresentado na seção 2.2.

2.6 Resumo

Neste capítulo foram apresentados alguns trabalhos que utilizam a representação de programas paralelos através de grafos para reconhecer o comportamento de uma aplicação em termos de processamento e comunicação. As 3 ferramentas consideradas mostram abordagens distintas para o emprego de grafos em conjunto com questões relativas ao escalonamento de tarefas.

O primeiro ambiente apresentado foi o Cilk, o qual restringe o grafo ao tipo divisão e conquista no contexto de múltiplos fluxos concorrentes. O escalonamento sobre um conjunto de processadores é baseado no roubo de tarefas. Para o uso de apenas um processador, o escalonador garante que a ordem de execução das tarefas seja mantida.

Já o sistema Pyrros parte de uma abordagem de descrição de tarefas que são escalonadas através de um método de agrupamento de tarefas para um processador. A distribuição de agrupamentos é feita verificando-se a carga de cada agrupamento, organizando-os em ordem crescente de carga e mapeando-os de forma a aproximar o balanceamento entre os processadores. O tipo de grafo utilizado no Pyrros é o DAG de tarefas. Este sistema caracteriza-se por ter como entrada uma descrição de grafo de tarefas e fornecer como saída um código que utiliza troca de mensagens para comunicação.

Por último, o CASCH é um ambiente completo para paralelização de códigos seqüenciais, através de grafos de tarefas e troca de mensagens, que permite testar diversas técnicas de escalonamento e escolher a que apresenta melhor desempenho. O usuário decide qual a melhor técnica, estaticamente, para ser empregada de acordo com a resposta obtida pelo sistema e depois o programa é executado.

As ferramentas apresentadas não tratam do escalonamento de programas que utilizam troca de mensagens através do MPI. Neste tipo de programas pode-se utilizar um grafo

de fluxo de dados DFG representativo de uma aplicação, no qual os vértices indicam o processamento enquanto que as arestas representam a comunicação. Com base nas informações do grafo, pode-se utilizar técnicas de particionamento para redistribuir as tarefas de forma a diminuir a comunicação. Existe uma ferramenta, chamada Metis (KARYPIS; KUMAR, 1995), que pode auxiliar nesta tarefa. Metis utiliza mecanismos que permitem o agrupamento de tarefas que têm uma alta intensidade de comunicação de forma que a troca de informações entre agrupamentos distintos seja reduzida. Nesse contexto, Metis poderia ser adotada como escalonador de tarefas.

Pode-se pensar em um mecanismo capaz de integrar as informações do grafo através técnicas de particionamento e redistribuição de tarefas automaticamente. Assim, obtém-se um mecanismo que extrai o grafo de uma aplicação e escalonam-se as tarefas de forma que elas utilizem menos a rede para comunicação - remapeamento. Por estar se tratando de um DFG, não há restrição quanto a forma do grafo, como no caso do grafo tipo divisão e conquista, por exemplo. Com o DFG pode-se representar qualquer um dos grafos utilizados nas ferramentas apresentadas. Uma abordagem de escalonamento estático contribui para que aplicações que tenham o mesmo padrão de comunicação possam ter seus processos distribuídos (remapeados) mais eficientemente.

O próximo capítulo apresenta o MPI mostrando as conceitos e características da norma. No capítulo seguinte ao próximo será proposto um mecanismo de escalonamento automático para programas MPI.

3 MPI - *MESSAGE PASSING INTERFACE*

O MPI é um padrão para troca de mensagens mundialmente conhecido e muito utilizado em aplicações de alto desempenho. A versão 1 da norma foi definida por um comitê de 40 especialistas em computação de alto desempenho nos anos de 1993 e de 1994. O objetivo dessa norma é facilitar o desenvolvimento de aplicações paralelas e bibliotecas através da definição de sintaxe, semântica de um conjunto de funções. Os seguintes aspectos fazem parte desta norma:

- comunicação ponto-a-ponto;
- operações de comunicação ou sincronização - entre processos de um mesmo grupo;
- grupos de processos - como devem ser manipulados;
- comunicadores - mecanismo que fornece um âmbito de execução diferente para os processos envolvidos;
- *Bindings* para Fortran e ANSI C - a definição da sintaxe das funções é próxima a estas linguagens.

A seguir, serão apresentados alguns conceitos do MPI, depois os serviços de comunicação e um serviço de rastreamento e depuração para programas MPI. Após essas 3 seções, são mostradas algumas características sobre migração de processos e as distribuições MPICH e LAM/MPI.

3.1 MPI - Conceitos

O MPI possui uma grande variedade de primitivas para o desenvolvimento de programas baseados em troca de mensagens. Existem alguns conceitos básicos necessários para a utilização do MPI descritos a seguir.

- Processo - é um programa em execução em uma máquina. O número de processos é determinado pelo usuário no disparo da execução da aplicação.
- Rank - todo processo possui um identificador único dado no momento de sua criação. Essa identificação começa de zero e vai até $N-1$, onde N é o número de processos no comunicador. O rank de um processo é usado para a localização de processo origem e destino durante a transmissão de mensagens.

- Comunicador - representa o conjunto de processos que podem ser contactados durante uma comunicação. Como padrão, o comunicador `MPI_COMM_WORLD` é predefinido e inclui todos os processos disparados no início da execução. Podem-se criar outros comunicadores na aplicação que incluam somente alguns processos, se necessário.

- Intercomunicador - é utilizado para permitir que processos de grupos distintos possam se comunicar.

- Grupo - é um conjunto ordenado de *N* processos. Cada grupo é associado a um comunicador e, como padrão, todos os processos são membros de um grupo em conjunto com um comunicador.

No MPI há basicamente duas formas de comunicação entre processos: ponto-a-ponto e grupo. Na comunicação ponto-a-ponto um processo se comunica apenas com um outro através de `send` ou `receive`. Já na comunicação em grupo, um ou mais processos lançam mensagens para os demais processos envolvidos.

3.2 Comunicação

Esta seção abrange os contextos de comunicação ponto-a-ponto e comunicação em grupo. O objetivo é apresentar as características das primitivas que são utilizadas neste âmbito.

3.2.1 Comunicação ponto-a-ponto

A comunicação ponto-a-ponto é caracterizada por 4 aspectos: quanto ao bloqueio; quanto ao modo de comunicação; quanto à persistência e quanto ao sentido de comunicação. Em relação ao bloqueio, as rotinas ditas bloqueantes são aquelas que garantem que o processo transmissor ou receptor ficará bloqueado até o término da transmissão de uma mensagem. Em contrapartida, os processos que usam as rotinas não bloqueantes continuam o seu processamento.

O MPI disponibiliza 4 modos de comunicação. O primeiro é o modo síncrono, no qual o receptor deve enviar uma confirmação de recebimento da mensagem, de forma que o transmissor tenha a garantia que a mensagem foi recebida. O segundo modo é o *bufferizado*, no qual a transmissão de uma mensagem utilizando *buffers* é completada rapidamente, pois, após a mensagem ser copiada para um *buffer*, o sistema decide quando transmiti- la. Há também a comunicação via modo *ready*, a qual é terminada rapidamente sem confirmações do processo receptor ou *buffers* com o objetivo de melhorar o desempenho da aplicação. E, por último, no modo padrão, é o sistema que decide se deve ou não bufferizar as mensagens. Por exemplo, o caso de mensagens pequenas, as quais podem ser bufferizadas.

O MPI oferece algumas variações das primitivas de comunicação ponto-a-ponto - `sends` `receives` bloqueantes padrão. Cada uma dessas primitivas é mostrada na Tabela 3.1.

Em relação à denominada persistência de comunicação, o MPI fornece algumas primitivas que têm como objetivo reduzir o atraso em programas que chamam as mesmas funções de comunicação ponto-a-ponto repetidamente. Essas primitivas são não-bloqueantes. Algumas das primitivas persistentes disponibilizadas pelo MPI são o `MPI_Send_init`

Tabela 3.1: Variações do `send` e do `receive`.

Função	Características
MPI_Send	envio padrão bloqueante
MPI_Isend	envio padrão não-bloqueante
MPI_Ssend	envio síncrono bloqueante
MPI_Issend	envio síncrono não-bloqueante
MPI_Recv	recepção padrão bloqueante
MPI_Irecv	recepção padrão não-bloqueante

e o `MPI_Recv_init`, os quais criam uma requisição a envio e recepção persistente, respectivamente. A transmissão de dados acontece efetivamente quando é feita a chamada às primitivas `MPI_start` ou a `MPI_Startall`. O que diferencia essas duas operações de transmissão de dados é que a MPI_Start ativa apenas uma operação, enquanto que a `MPI_Startall` ativa um conjunto de operações

Quanto ao sentido da comunicação, o MPI fornece primitivas que transferem dados nos dois sentidos - envio e recepção. Por exemplo, a primitiva `MPI_Sendrecv` que assim como as demais funções de envio nos dois sentidos é bloqueante.

Além das variações de primitivas para comunicação ponto-a-ponto, o MPI também oferece uma diversidade de funções para comunicação em grupo. Essas funções são explicadas na próxima seção

3.2.2 Comunicação em Grupo

A comunicação em grupo possui dois níveis: o primeiro relaciona-se à comunicação somente entre os processos do mesmo grupo dado por um mesmo comunicador, também chamada de comunicação coletiva; e o segundo diz respeito à comunicação entre processos de grupos distintos. Neste último caso, leva-se em conta também o conceito de inter-comunicadores, ou seja, comunicação entre processos associados a comunicadores distintos.

A comunicação coletiva é composta por primitivas que permitem a comunicação entre processos de um mesmo grupo, vide Figura 3.1. A descrição de tais primitivas está a seguir:

- *Broadcast* - os dados de um processo são enviados para os demais processos;
- *Scatter* - os dados de um processo são divididos e distribuídos entre todos os processos do grupo;
- *Gather* - os processos do grupo enviam seus dados para um único processo que ordena os pacotes conforme o número do processo;
- *Allgather* - são agrupados os dados dos processos do grupo e esse agrupamento é enviado para todos os membros do grupo;
- *Barrier* - garante que todos os processos do grupo cheguem no mesmo ponto de execução para, depois, continuar a aplicação;

- *Reduce* - faz uma operação (soma, produto, máximo, mínimo e outras) com os dados de todos os membros do grupo e o resultado é entregue a um processo;
- *Allreduce* - a função é a mesma que a anterior porém o resultado é passado para todos os membros do grupo.

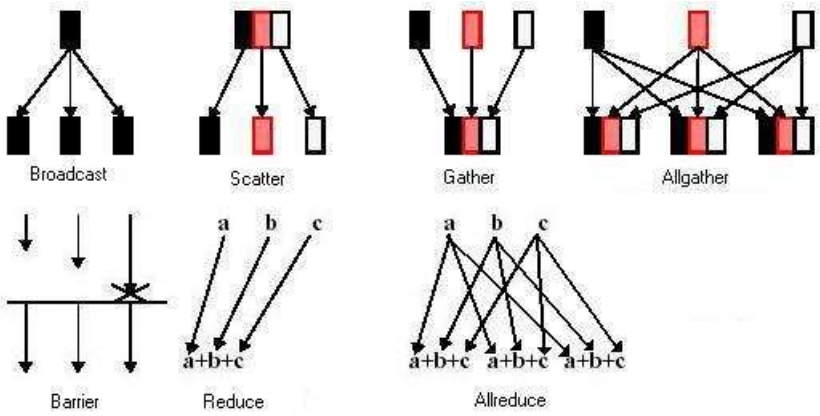

Figura 3.1: Funções de comunicação coletiva.

Na comunicação em grupo, a troca de mensagens ocorre entre os processos pertencentes a grupos distintos. Cada grupo tem um comunicador associado. O conceito de comunicador é importante para permitir que sub-tarefas sejam dadas para um grupo menor de processos os quais podem usar comunicação coletiva de modo seguro. O conceito de grupo e comunicadores é dinâmico no sentido que eles podem ser criados e destruídos durante a execução de uma aplicação. Para utilizar um grupo de processos seguem-se os seguintes passos:

- extração de um conjunto de processos do `MPI_COMM_WORLD` através da primitiva `MPI_Comm_group`;
- formação de um novo grupo com o conjunto de processos extraídos do grupo global através de `MPI_Group_incl`;
- criação de um novo comunicador para o novo grupo por `MPI_Comm_create`;
- atribuição de um novo rank para os processos associados ao novo comunicador através de `MPI_Comm_rank`;
- processamento e comunicação entre os processos do grupo;
- destruição do grupo e do comunicador através de `MPI_Group_free` e de `MPI_Comm_free`, respectivamente.

3.3 Depuração e Rastreamento

O MPI disponibiliza na sua distribuição uma ferramenta concebida para o rastreamento e depuração de programas MPI denominada MPE *Multiprocessing Environment* (GROPP; LUSK; SKJELLUM, 1999). O MPE permite que após a execução de um programa MPI se obtenha um arquivo de *log*. As informações obtidas no arquivo de *log*

```
1 #include <stdio.h>
2 #include <mpi.h>
3 #include "mpe.h"
4
5 int main(int argc, char **argv){
6    MPI_Comm row_comm;
7    int myrank, size, p,q, Q;
8    int event1a, event1b, event2a, event2b;
9    MPI_Init (&argc, &argv);
10   MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
11   MPI_Comm_size (MPI_COMM_WORLD, &size);
12   MPE_Init_log();
13
14   event1a = MPE_Log_get_event_number();
15   event1b = MPE_Log_get_event_number();
16   event2a = MPE_Log_get_event_number();
17   event2b = MPE_Log_get_event_number();
18
19   if(myrank==0){
20        MPE_Describe_state(event1a, event1b, "Mensagem", "red");
21        MPE_Describe_state(event2a, event2b, "Compute",   "blue");
22   }
23   MPE_Log_event(event2a, 0, "start compute");
24   Q = size /2;
25   p = 1;
26   q = myrank - Q;
27   if(myrank < Q) {
28           p = 0;
29           q = myrank;
30   }
31
32   MPE_Log_event(event2b, 0, "end compute");
33   MPE_Log_event(event1a, 0, "start Mensagem");
34   printf{"\n Mensagem Oi %d - %d - %d\n", myrank, q, Q};
35   MPE_Log_event(event1b, 0, "end Mensagem");
36   MPE_Finish_log;
37   MPI_Finalize();
38 }
```

Figura 3.2: código MPE

podem ser visualizadas graficamente, o que mostra o momento exato no qual uma determinada primitiva foi chamada por um processo.

Para a depuração e rastreamento de um programa MPI, através do MPE, é necessário fazer a instalação compilando o MPI com os *flags* de configuração do MPE. O rastreamento de programas MPI pode ser feito de duas formas: se o objetivo é depurar primitivas MPI, então compila-se passando como parâmetro um *flag* para geração do *log* e ao executar o programa, o arquivo de *log* será gerado automaticamente. Por outro lado, se for necessário depurar módulos ou funções do programa MPI, deve-se incluir o *header* do MPE e podem-se utilizar as primitivas MPE para monitoração de eventos, as quais indicam o início e o final da área de rastreamento, Figura 3.2. Para o caso de monitoração de primitivas MPI como *send* e *receive* não é necessário usar as primitivas MPE, bastando apenas incluir o *header* do MPE, Figura 3.3. As principais primitivas que o MPE disponibiliza são:

- `MPE_Init_log (void)` - é chamada por todos os processos para inicializar as estruturas de *log* do MPE.

- `MPE_Start_log (void)` - usada para dinamicamente habilitar a monitoração e por padrão é chamada após a primitiva MPE_Init_log;

```
1 #include <stdio.h>
2 #include <mpi.h>
3 #include "mpe.h"
4
5 int main(int argc, char **argv){
6    char mensagem[20];
7    int id, tam,i;
8    MPI_Datatype type;
9
10   MPI_Init(&argc,&argv);
11   MPI_Comm_rank(MPI_COMM_WORLD,&id);
12   MPI_Comm_size(MPI_COMM_WORLD,&tam);
13
14   if (id==0)
15   {
16        strcpy(mensagem,"Ola Mundo");
17        for (i=1;i<tam;i++)
18             MPI_Send(&mensagem,2,MPI_CHAR,i,20,MPI_COMM_WORLD);
19   }
20   else
21        MPI_Recv(&mensagem,2,MPI_CHAR,0,400,MPI_COMM_WORLD, &status);
22
23   MPI_Finalize();
24 }
```

Figura 3.3: Send/Receive

- `MPE_Stop_log (void)` - assim como `MPE_Start_log`, é chamada após `MPE_Init_log`, porém para desabilitar a monitoração;
- `MPE_Finish_log (char *logfilename)` - agrupa as informações de todos os processos, ajustando os tempos de duração para gerar o arquivo de *log* de acordo com o início e final da monitoração, dado pelas chamadas a `MPE_Init_log` e `MPE_Finish_log`, respectivamente;
- `MPE_Describe_state (int start, int end, char *name, char *color)` - esta primitiva é utilizada para descrever o estado de um evento - usado para a realização de análises de arquivos de *log* via visualizadores gráficos. O estado de um evento pode ser descrito através de cores, por exemplo;
- `MPE_Describe_event (int event, char *name)` - serve para descrever um evento definido pelo usuário - também utilizada para análises de arquivos de *log* através de visualização gráfica;
- `MPE_Log_event (int event, int intdata, char *chardata)` - utilizada para monitorar um determinado evento definido pelo usuário;
- `MPE_Log_get_event_number()` - atribui um número ao evento.

A monitoração das primitivas *send* e *receive* foi utilizada como exemplo de uso do MPE, como mostra a Figura 3.3. Essas duas primitivas foram visualizadas pela ferramenta *logviewer* onde pode ser visualizado o arquivo de *log* gerado pelo MPE . A primeira tela vista após a chamada da ferramenta de visualização está representada pela Figura 3.4, a qual mostra a relação entre número de eventos versus tempo, sendo que os eventos considerados são a transmissão e a recepção de uma mensagem e também a conexão entre os processos. Nessa Figura, à esquerda, em preto, é indicado o tempo para começar a transmissão da mensagem e no lado oposto, à direita, em cinza escuro, é o tempo de recepção da mensagem. Também é possível ver na Figura 3.4 que existe a opção de

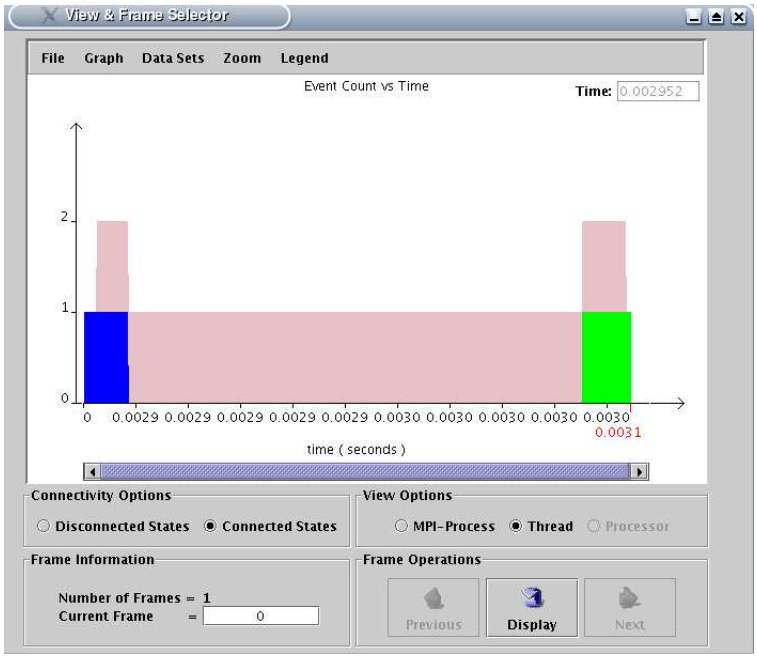

Figura 3.4: Visualização logviewer.

display em *frame operations*. Através dessa opção, a transmissão e a recepção de uma mensagem serão visualizadas conforme a Figura 3.5, a qual mostra que o processo 0 enviou uma mensagem para o processo 1 através de uma seta representativa.

O arquivo de log gerado pelo MPE possibilita que sejam filtradas as informações de identificadores de processos e volume de dados trocados entre eles. Estes são os dados necessários para compor um DFG de uma aplicação. Um exemplo de arquivo de *log* pode ser visto na Figura 3.6. Nessa Figura, as primitivas *send* e *receive* estão sendo monitoradas. E as informações necessárias para a construção do DFG estão nas linhas 8 e 14, que indicam o *send* e o *receive*, respectivamente. Na linha 8, o envio de uma mensagem é representado por `id=-101`, o processo origem por `pid=0`, o processo destino por `data=1` e o volume de dados enviados por `desc=100 1`. No volume de dados, o número 100 é o tamanho da mensagem e o número 1 indica que é realizado um *send*. A mesma idéia é válida para o *receive* na linha 14, porém a representação é dada por `id=-102`.

O arquivo de *log* fornecido após a execução do aplicativo com o MPE, pode ser utilizado para montar um grafo de fluxo de dados de uma aplicação. O ônus causado para montar este grafo é esperar a execução de toda a aplicação e executar um programa auxiliar que consiga filtrar as informações necessárias para a construção do grafo.

3.4 Distribuições MPI

Nesta seção são apresentadas as distribuições MPICH e LAM do MPI. O MPICH é uma distribuição livre que surgiu durante o processo de padronização do MPI com o objetivo de realimentar o Fórum de sua implementação. O LAM-MPI também é uma implementação livre que surgiu mais recentemente já com suporte à norma 2 do MPI (GROPP; LUSK; SKJELLUM, 1999).

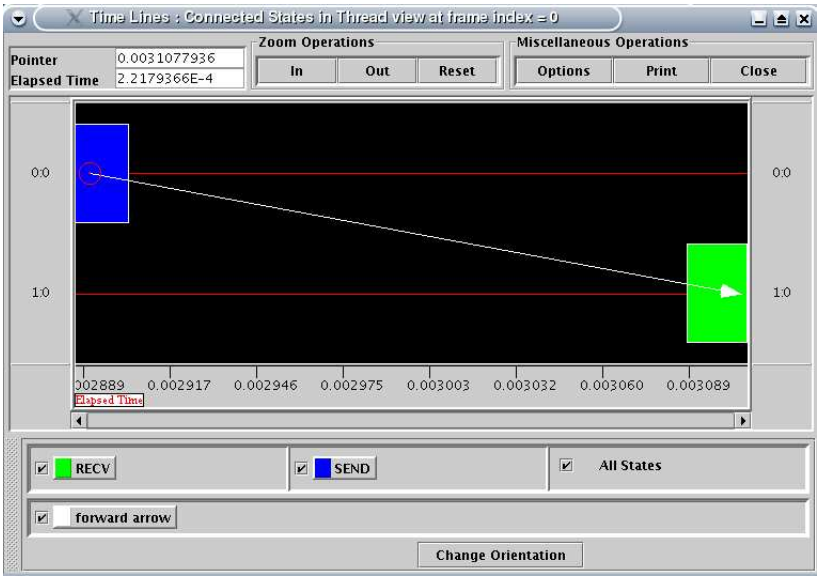

Figura 3.5: Visualização sendrecv.

```
ts=0.000163 type=loc  len=9, pid=0 srcid=900 line=341 file=clog.c
ts=0.000165 type=comm len=5, pid=0 et=init pt=-1 ncomm=91 srcid=900
ts=0.000166 type=loc  len=9, pid=0 srcid=901 line=286 file=clog.c
ts=0.000168 type=raw  len=7, pid=0 id=-201 data=-1 srcid=901 desc=MPI_PROC_NULL
ts=0.000169 type=raw  len=7, pid=0 id=-201 data=-2 srcid=901 desc=MPI_ANY_SOURCE
ts=0.000172 type=raw  len=7, pid=0 id=-201 data=-1 srcid=901 desc=MPI_ANY_TAG
ts=0.000215 type=raw  len=7, pid=0 id=161 data=1 srcid=901 desc=
ts=0.000218 type=raw  len=7, pid=0 id=-101 data=1 srcid=901 desc=100 1
ts=0.000240 type=raw  len=7, pid=0 id=162 data=2 srcid=901 desc=
ts=0.001041 type=loc  len=9, pid=1 srcid=900 line=341 file=clog.c
ts=0.001044 type=comm len=5, pid=1 et=init pt=-1 ncomm=91 srcid=900
ts=0.001079 type=loc  len=9, pid=1 srcid=901 line=286 file=clog.c
ts=0.001082 type=raw  len=7, pid=1 id=155 data=1 srcid=901 desc=
ts=0.001095 type=raw  len=7, pid=1 id=-102 data=0 srcid=901 desc=100 1
ts=0.001095 type=raw  len=7, pid=1 id=156 data=2 srcid=901 desc=
ts=0.001464 type=loc  len=9, pid=0 srcid=902 line=366 file=clog.c
ts=0.001468 type=sdef len=10, pid=0 id=200 start=155 end=156 color=green:light_gray desc=RECV
ts=0.001470 type=sdef len=10, pid=0 id=201 start=161 end=162 color=blue:gray3 desc=SEND
ts=1000000.000000 type=elog len=3, pid=0 end of log
```

Figura 3.6: Arquivo de log gerado pelo MPE.

3.4.1 MPICH

O MPICH (PACHECO, 2001; THE MPI MESSAGE PASSING INTERFACE STANDARD, 1995), foi desenvolvido pelo Argonne National Laboratory em 1993 em um esforço para fornecer uma implementação que poderia ser rapidamente portada para diferentes sistemas e que seguiria a norma MPI-1. O MPICH foi portado para diversos sistemas, como: clusters diferentes versões de UNIX/Linux; redes com WindowsNT e Windows 2000; MPPs como as Thinking Machines CM5, IBM, SP e intel Paragon, e SMPs. Atualmente a versão do MPICH é a 1.2.6.

Para a execução de uma aplicação utilizando o MPICH primeiramente compila-se o programa através do `mpicc` e depois executa-se a aplicação usando o `mpirun` . Mas antes de executar o programa é necessário criar um arquivo contendo uma lista de nodos, a qual vai informar em quais máquinas o programa será disparado.

3.4.2 LAM-MPI

A distribuição *Local Area Multicomputer*(LAM) foi inicialmente desenvolvida pelas Universidades de Ohio no Ohio Supercomputing Center em 1989. Depois, a partir da

versão 6.2b, o projeto passou para o domínio do Laboratory for Scientific Computing (LSC)da Universidade de Notre Dame. O LSC se mudou em 2001 para a Universidade de Indiana, onde o desenvolvimento do projeto está acontecendo atualmente.

A versão 6.5.6 da LAM-MPI suporta a norma MPI-1.2 e a norma MPI-2 incluindo a criação e gerenciamento dinâmico de processos. O propósito inicial da LAM era fornecer uma biblioteca MPI que seria executada eficientemente em redes heterogêneas de estações de trabalho e em clusters. Várias distribuições de Unix/Linux foram testadas em clusters e em estações de trabalho como: Solaris, OpenBSD, Linux Mac OS X, IRIX, HP-UX e AIX. O LAM ganhou grande popularidade em clusters devido ao fato de ter sido originalmente desenvolvido para arquiteturas de clusters.

Para a execução de programas, o LAM usa um aplicativo chamado `lamboot`, o qual carrega um daemon chamado `lamd`. O número de daemons carregados pelo `lamboot` formam a base de execução do LAM. O `lamd` é utilizado para auxiliar na carga da aplicação, realizada pelo `mpirun` e para controlar os processos em termos de comunicação.

O que um usuário precisa fazer, no LAM após escrever uma aplicação resume-se a 3 passos: primeiro compilar a aplicação através do `mpicc`; segundo, executar o aplicativo `lamboot` para a carga dos daemons; e por último executar `mpirun`. Uma outra possibilidade é usar o `mpiexec` para executar o programa. Uma característica importante que o diferencia do `mpirun` é configurar cada processo para ser executado em uma determinada máquina da lista de nodos. Isto é feito através da chamada a `mpiexec` seguido pelo número do processo, o parâmetro `-host` e a máquina destino conforme a chamada: `$ mpiexec -n 1 -host nodo programa`, vai lançar uma cópia de "programa"no *host* "nodo". Após a execução do programa é necessário finalizar os daemons, através do comando `lamhalt`.

3.5 Resumo

Este capítulo apresentou o MPI levando em conta a norma 1 do MPI. Primeiramente foram apresentados alguns conceitos básicos de MPI. Seguiram-se explicações sobre as características de comunicação seja ela ponto-a-ponto ou coletiva. Na terceira seção, foi mostrada a ferramenta MPE, concebida para a rastreabilidade e depuração de programas MPI. Na quarta seção foi apresentado o conceito de migração de processos e uma distribuição que possui suporte a funcionalidades desse tipo

Na seção distribuições, comentou-se sobre MPICH e LAM/MPI onde foram mostradas as diferenças fundamentais entre elas. O objetivo deste capítulo foi fornecer ao leitor subsídios necessários para a compreensão da proposta e das decisões do trabalho.

Além dessas distribuições MPI, existe o *Open Source High Performance Computing* (Open-MPI) (GRAHAM; WOODALL; SQUYRES, 2005), o qual é um consórcio de projetos que visa disponibilizar uma implementação MPI que suporte todas as funcionalidades do conjunto de implementações MPI em uma única. Os desenvolvedores do Open-MPI têm como intenção tornar pública uma versão que inclui o suporte ao padrão MPI-2, a tolerância a falhas e à migração de processos.

O capítulo seguinte apresenta a proposta de serviço de escalonamento estático para programas MPI baseado no grafo de fluxo de dados de uma aplicação. A distribuição LAM-MPI será utilizada, pois esta possui suporte a funcionalidades de migração de processos e em um trabalho futuro poderá ser usada na integração da β-MPI com este recurso.

4 ESCALONAMENTO AUTOMÁTICO DE PROGRAMAS MPI - β-MPI

Muitas aplicações paralelas utilizam ferramentas de troca de mensagens como recurso de comunicação. O padrão é o MPI, visto no capítulo anterior. Neste padrão, não é previsto um mecanismo de escalonamento de tarefas. O programador define como será realizada a distribuição das tarefas de acordo com a arquitetura utilizada.

Uma grande parte das técnicas de escalonamento existentes baseia-se na análise do grafo de fluxo de dados DFG da aplicação. A analogia entre uma aplicação que realiza troca de mensagens e um grafo de fluxo de dados é direta: os processos correspondem aos vértices do grafo e a comunicação entre eles corresponde às arestas do grafo.

A proposta deste trabalho é fornecer uma ferramenta que produza automaticamente o grafo de fluxo de dados de programas MPI - β-MPI (β- Biblioteca de Escalonamento de Tarefas) e que, com base nas informações das arestas do grafo, sejam tomadas as decisões de escalonamento. No escalonamento das tarefas é levado em conta onde e quando uma tarefa será executada. A ferramenta trata somente da localização de execução das tarefas, ou seja, do mapeamento destas sobre a arquitetura utilizada (vide Figura 4.1). Este mapeamento será realizado após uma execução de um programa MPI. O objetivo será encontrar o melhor mapeamento possível entre software e hardware. Após o mapeamento das tarefas, a aplicação deve ser executada novamente seguindo a ordem proposta pelo escalonador. Esta idéia é similar à encontrada na ferramenta CASCH, sob o ponto de vista de que o usuário executa uma aplicação previamente para optar entre um dos algoritmos de escalonamento que ela disponibiliza. Depois de definido o algoritmo, o usuário executa a aplicação novamente.

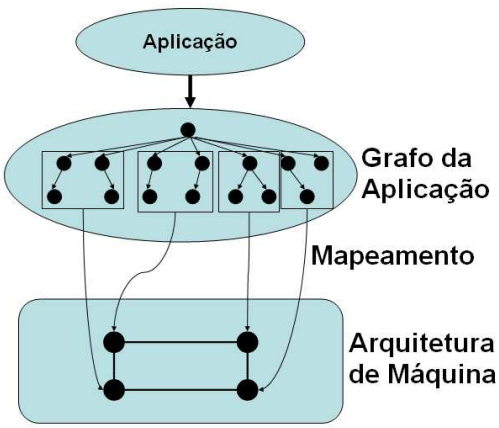

Figura 4.1: Extração e mapeamento do grafo.

As aplicações alvo para o emprego da β-MPI são aquelas que não podem ter variações de comportamento durante a sua execução. Uma vez que ela tenha sido executada, as execuções seguintes têm que manter o mesmo comportamento em termos de padrão de comunicação.

O projeto da biblioteca β-MPI é composto por três blocos: redefinição de primitivas; geração do DFG e ferramenta de particionamento. Estes blocos são ilustrados na Figura 4.2.

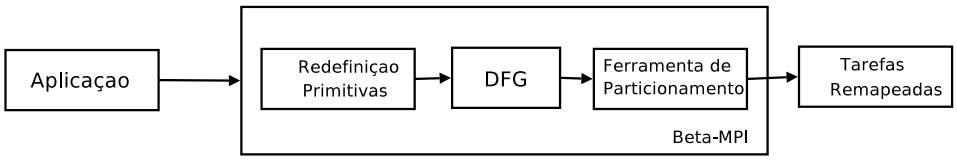

Figura 4.2: Estrutura da biblioteca β-MPI.

A estrutura da β-MPI é apresentada nas próximas 2 seções sendo que a primeira mostra a parte de geração do grafo de fluxo de dados de um programa MPI. E a segunda trata da ferramenta de particionamento e o mecanismo de redisparo de processos adotados.

4.1 A Geração do DFG pela β-MPI

O esquema de geração do DFG foi concebido através da redefinição das primitivas de troca de mensagens MPI. A seguir será explicado o mecanismo de redefinição de primitivas adotado na β-MPI e esclarecido como são geradas as informações do grafo da aplicação.

4.1.1 Redefinição das Primitivas

Para obter a representação de um programa através de um grafo é preciso encontrar uma forma de extrair informações de volume de processamento e de comunicação entre os processos envolvidos. Para resolver este problema, adotou-se o *rank* do processo como identificador de um vértice do grafo. As arestas são definidas pelas mensagens trocadas entre os processos e o volume de dados como sendo o peso da mesma. Como é permitido apenas uma aresta entre cada dois vértices, devido a ferramenta de particionamento de grafos contabilizar apenas uma aresta entre eles, a biblioteca deve identificar todas as mensagens trocadas e calcular a soma de todos os dados nelas contidos para construir o grafo.

Para rastrear os dados que cada mensagem contém, foi necessário redefinir as primitivas MPI utilizadas pela aplicação. A redefinição é realizada através da chamada de uma primitiva definida pela β-MPI que tem o mesmo comportamento da função original, porém, internamente, ela soma o volume de dados que está contido na mensagem e armazena o resultado em uma estrutura de dados. A estrutura formada é uma descrição do grafo de fluxo de dados da aplicação.

Especificamente, em se tratando do desenvolvimento, a biblioteca β-MPI é composta por um arquivo `beta-mpi.h` que redefine as primitivas MPI, utilizando o precompilador *C* para chamar as novas versões implementadas na `libbeta-mpi.a`, com a qual será realizada a ligação. No final da execução da primitiva redefinida é chamada a função MPI correspondente para que seja mantido o mesmo retorno.

Tabela 4.1: Primitivas β-MPI.

Primitivas que foram redefinidas disponíveis na biblioteca β-MPI
int bMPI_Init(int * argc,char *** argv)
int bMPI_Finalize()
int bMPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int bMPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
int bMPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
int bMPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
int bMPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int bMPI_Issend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
int bMPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
int bMPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
int bMPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)
int bMPI_Alltoallv(void *sendbuf, int *sendcnts, int *sdispls, MPI_Datatype sendtype, void *recvbuf, int *recvcnts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
int bMPI_Allreduce(void * sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

4.1.1.1 Primitivas Redefinidas

As primitivas MPI que foram redefinidas estão na Tabela 4.1. Em termos de comunicação, tanto primitivas de troca de mensagens ponto-a-ponto como aquelas de comunicação em grupo, disponibilizadas pelo MPI, foram redefinidas. Cabe ressaltar que existem primitivas que poderão ser redefinidas mediante o surgimento delas em uma aplicação que esteja usando a β-MPI.

A primitiva `MPI_Init` foi redefinida para que fosse criado um *buffer* de tamanho de n processos que depois é utilizado pela biblioteca β-MPI na contabilização dos dados. Este *buffer* é gerado na função __size() chamada na `MPI_Init`, vide Figura 4.3. Os `#define` colocados tanto na Figura 4.3 como nas demais figuras têm apenas o intuito elucidativo pois na verdade os #defines estão no arquivo *header* (*.h) da biblioteca.

A primitiva `MPI_Finalize` foi redefinida para que o grafo da aplicação fosse montado, vide Figura 4.4. A montagem do grafo é feita pela função __gatherGraph(), explicada em 4.1.2.

A forma como as primitivas de comunicação foram redefinidas é basicamente a mesma.

```
1 # define MPI_Init(argc, argv) bMPI_Init(argc, argv)
2
3 int bMPI_Init(int *argc, char argv){
4         int binit;
5         binit = MPI_Init(argc, argv);
6         void__size();
7         return binit;
8 }
9 int __size (){
10   if (s == -1) {
11     MPI_Comm_size (MPI_COMM_WORLD, &s);
12     __create_buffer (s);
13   }
14   return s;
15 }
16 void __create_buffer (int np){
17   int i;
18   msgSizes = (int *) malloc (np * sizeof(int));
19   for (i=0; i<np; i++)
20     msgSizes[i]=0;
21 }
```

Figura 4.3: Função init redefinida pela β-MPI.

```
1 # define MPI_Finalize() \ bMPI_Finalize()
2
3 int bMPI_Finalize(){
4         int bfina;
5         __gatherGraph();
6         bfina = MPI_Finalize();
7         return bfina;
8 }
```

Figura 4.4: Função finalize redefinida pela β-MPI.

A Figura 4.6 mostra a função `send` e a Figura 4.7 mostra a função `receive`. Em cada uma dessas primitivas, existe uma chamada a MPI_Type_size, utilizada para contabilizar o número de dados segundo o tipo (int, double, char, etc). Depois, há uma chamada para a função original MPI. Antes do retorno existe a primitiva __comm_force(), a qual é utilizada para contabilizar corretamente os dados de acordo com a representação do *rank* do processo no comunicador global.

Primeiro foram redefinidas as primitivas de troca de mensagens. O próximo passo é formar a estrutura de dados com as informações necessárias para gerar o DFG da aplicação.

4.1.2 Informações da Estrutura de Dados

O segundo bloco da composição da biblioteca β-MPI é a estrutura de dados com as informações para geração do DFG. O grafo será montado quando a execução de um programa MPI estiver chegando ao seu final, sendo chamada a primitiva `MPI_Finalize()`, a qual na sua redefinição tem uma chamada para a função `_gatherGraph()`, Figura 4.5. Esta função tem como objetivo percorrer a estrutura de dados gerada eliminando as arestas paralelas entre dois vértices e reunindo as informações locais de cada processo em um único *buffer*. Com o *buffer* contendo o grafo da aplicação, pode-se descrevê-lo de acordo com o formato requerido na utilização pela ferramenta de particionamento, como mostra a Tabela 4.2. Na primeira linha estão o número de vértices, o número de arestas e um identificador usado na escolha de associação de peso, o qual pode: não existir; ser

```
1  void __gatherGraph ()
2  {
3    // contadores
4    int i,j,k=0, cont_me=0;
5    int *theMsgs;
6    // tamanhos dos buffers de cada processo
7    if (__rank() == 0)
8      theMsgs = (int *) malloc (__size() * __size() * sizeof (int));
9    MPI_Gather (msgSizes, __size(), MPI_INT, theMsgs,  __size(), MPI_INT, 0, MPI_COMM_WORLD);
10
11   if (rk == 0) {  /* este é um trabalho para o rank 0!! */
12     /* imprimindo os valores recebidos */
13     k=0; // coloca numero de arestas em k
14     for (i = 0; i < __size(); i++){
15       for (j= 0; j < __size(); j++){
16         if (theMsgs[i + (j * __size() ) ] != 0)
17           k++;
18       }
19     }
20     if ((k % 2) == 0)
21       k = k / 2;
22     else
23       printf("warning: bad edges number (odd)\n");
24     printf("### graph info begins ###\n");
25     printf("%d %d 1\n",__size(), k);
26
27     for (i = 0; i < __size(); i++){
28       for (j= 0; j < __size(); j++){
29         if (theMsgs[j + (i * __size() ) ] != 0)
30           if(theMsgs[j + (i * __size() ) ] <= 1000)
31             cont_me++;
32       }
33     }
34     for (i = 0; i < __size(); i++){
35       for (j= 0 ; j < __size(); j++){
36         if (theMsgs[j + (i * __size() ) ] != 0)
37           printf("\ncont_me %d e size %d ",cont_me, __size());
38       }
39       printf("\n");
40     }
41     free (theMsgs);
42     printf("### graph info ends ###\n");
43   }
44   free (msgSizes);
45 }
```

Figura 4.5: Função __gatherGraph que monta o grafo.

associado às arestas; ser associado aos vértices; ou ser associado a ambos. Para a β-MPI o peso foi associado as arestas, pois o objetivo é diminuir o volume mensagens trocadas entre processos alocados em nodos distintos. Isto será feito através da alocação de processos que trocam muitas mensagens para um mesmo nodo.

Cada uma das linhas seguintes da Tabela 4.2 indicam um processo. Os pares vértice e aresta, em cada linha, representam o processo com o qual o processo linha se comunicou e a mensagem trocada entre eles. Isto significa que um processo 1 (Primeira linha) se comunica com um processo "vértice"e troca "aresta"mensagem.

O DFG da aplicação permite que se tenha uma descrição do volume de comunicação entre os processos. Isto pode ser identificado mais claramente através de uma visualização gráfica, como mostra a Figura 4.8. Essa figura tem a tabela e a ilustração do grafo com o volume de dados indicado entre os 4 processos. A visualização permite analisar os pontos críticos de comunicação diretamente, tendo-se uma do comportamento da aplicação. Aβ-MPI não fornece uma forma de visualização gráfica do DFG. Aqui ela foi usada apenas para mostrar os pontos de comunicação do DFG que o escalonador terá que tratar.

```
1   # define MPI_send(buf, num, datatype, dest, tag, comm, request) \
2    bMPI_send(buf, count, datatype, dest, tag, comm, request)
3
4  int bMPI_Send(void *buf, int count, MPI_Datatype datatype,
5  int dest, int tag, MPI_Comm comm)
6  {
7        int beta_return;
8        int datype_size;
9        MPI_Type_size(datatype, &datatype_size);
10       beta_return = MPI\_Send(buf, count, datatype, dest, tag, comm);
11       __comm_force(dest, count *datatype_size, comm);
12       return beta_return;
13   }
```

Figura 4.6: Função send redefinida pela β-MPI.

```
1   # define MPI_Recv(buf, num, datatype, src, tag, comm, stat) \
2    bMPI_Recv(buf, count, datatype, src, tag, comm, stat)
3
4  int bMPI_Recv(void *buf, int count, MPI_Datatype datatype,
5  int source, int tag, MPI_Comm comm, MPI_Status *status)
6  {
7      int beta_return;
8      int datype_size;
9      MPI_Type_size(datatype, &datatype_size);
10     beta_return = MPI_Recv(buf, count, datatype, source, tag, comm, status);
11     __comm_force(source, count *datatype_size, comm);
12     return beta_return;
13   }
```

Figura 4.7: Função receive redefinida β-MPI.

Tabela 4.2: Grafo β-MPI.

x vértices	*y* arestas	identificador	
vértice	aresta	vertice	aresta
vértice	aresta	vertice	aresta
vértice	aresta	vertice	aresta
vértice	aresta	vertice	aresta

4.1.3 Identificação de Processos

Na montagem da estrutura de dados através da chamada a `_add_msg` também foi considerada a questão de comunicação entre processos em grupos distintos. No MPI, quando uma aplicação é disparada, todos os processos envolvidos partem da utilização do comunicador `MPI_COMM_WORLD`, o qual indica que eles fazem parte de um grupo global onde cada um deles tem um identificador (*rank*). A partir disto, pode-se formar sub-grupos com alguns processos através da criação de comunicadores e identificadores locais. Porém para que seja possível a identificação de um processo global como um processo local em um sub-grupo é necessário traduzir os *ranks*. A Figura 4.9 mostra como isso é feito. Os parâmetros de entrada da função `comm_force` são o identificador do processo, com o qual o processo atual está se comunicando, o volume de dados que ele possui e a qual comunicador ele faz parte. Logo no primeiro teste é verificado se o processo faz parte do comunicador global `MPI_COMM_WORLD`. Se fizer parte do comunicador global então é contabilizado pela função `_add_msg` o `rank` do processo

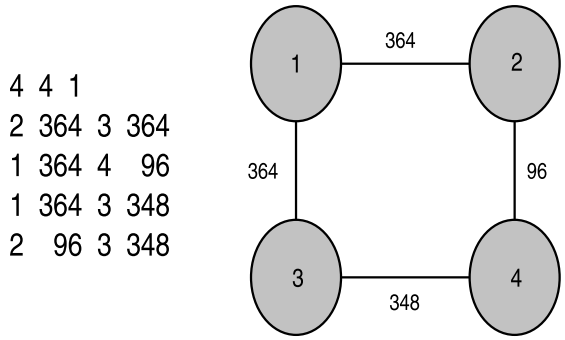

Figura 4.8: Formato de descrição do grafo e sua representação gráfica.

atual, o `rank` do processo com o qual ele está trocando informações e o volume de dados transferidos entre eles. Por outro lado, se o processo faz parte de um sub-grupo, ou seja, tem outro comunicador, consequentemente tal comunicador é identificado pela função `MPI_Comm_group`, assim como o `rank` local do processo. Depois os ranks do processo atual e do processo com o qual ele está se comunicando são traduzidos para os processos do comunicador global pela função `MPI_Group_translate_ranks` e seus dados são contabilizados pela chamada a `_add_msg`.

```
1
2  void __comm_force(int whoami, int countdata, MPI_Comm comm)
3  {
4          int rank, rank_1;
5          int rank_local;
6          MPI_Group worldgroup, localgroup;
7
8          if(comm!=MPI_COMM_WORLD)
9          {
10                 MPI_Comm_group(MPI_COMM_WORLD,&worldgroup);
11                 MPI_Comm_group(comm,&localgroup);
12                 //Global Group
13                 MPI_Comm_rank(comm, &rank_local);
14
15                 MPI_Group_translate_ranks(localgroup, 1, &rank_local, worldgroup, &rank);
16                 MPI_Group_translate_ranks(localgroup, 1, &whoami, worldgroup, &rank_1);
17
18                 __add_msg(rank, rank_1, countdata);
19         }
20         else
21         {
22                 MPI_Comm_rank(comm,&rank);
23                 __add_msg(rank, whoami, countdata);
24         }
25
26 }
```

Figura 4.9: Tradução de ranks realizada pela função `comm_force` na β-MPI.

Outro ponto tratado durante a implementação foi o cálculo do volume de dados. Os `MPI_Datatypes` estavam sendo contabilizados através da função `sizeof(type)` do *C*. Isto foi um equívoco que gerou a montagem de uma tabela com dados inconsistentes porque algumas informações não eram filtradas, ou seja, os tipos de dados não estavam sendo somados corretamente. A solução foi usar uma primitiva disponibilizada pelo MPI

(`MPI_Type_size`) para realizar o cálculo exato do tamanho de dados armazenado em cada mensagem.

O DFG de uma aplicação através da β-MPI é obtido após uma primeira execução. Com relação a utilização para a geração do grafo, o usuário vai precisar apenas colocar na aplicação o `#include beta-mpi.h` e compilar a aplicação. Quando a aplicação for executada o grafo será gerado automaticamente. Com isto, a próxima etapa é identificar os pontos críticos de comunicação para poder remapear os processos envolvidos eficientemente. Esta tarefa é realizada pela ferramenta de particionamento.

4.2 Ferramenta de Particionamento e Mapeamento dos Processos

De posse do grafo, uma estratégia de remapeamento de tarefas (processos) que pode ser usada é representar o programa através de um grafo e utilizar uma ferramenta de particionamento de grafos que forneça como saída as partições conforme a ordem em que os processos devem ser redisparados. Os processos remapeados são redisparados através de uma nova execução do programa MPI. Ao final, o usuário pode comparar as duas execuções e verificar se houve ganho no desempenho da aplicação com a redistribuição dos processos pelas máquinas utilizadas.

Este último bloco da biblioteca β-MPI vai receber como entrada o grafo de fluxo de dados da aplicação e vai particioná-lo buscando uma boa qualidade através do agrupamento de processos que trocam muitas mensagens a um mesmo nodo, o que gera menos tráfego através da rede. A eficiência de um particionamento relaciona-se à minimização do corte de arestas do grafo conforme apresentado no capítulo 2 e detalhado em (SCHLOEGEL; KARYPIS; KUMAR, ????). Para particionar o DFG foi escolhida a ferramenta Metis (KARYPIS; KUMAR, 1995) a qual oferece qualidade de particionamento e é pouco intrusiva no tempo de execução da aplicação.

A ferramenta Metis recebe um arquivo de entrada contendo uma descrição do DFG de uma aplicação pois para particionar o grafo é preciso passar como parâmetros o nome do arquivo com a descrição, o grafo e o número de partições desejadas. O comando a ser chamado será escolhido conforme o número de partições desejadas. Se o número de partições for maior do que 8, é preferível chamar o comando `$Kmetis`, pois este fornece melhor qualidade de particionamento. Por outro lado, se o número de partições for menor ou igual a 8, usando-se o `$pmetis` obtém-se melhor qualidade no particionamento.

Um exemplo de como é descrito o DFG está na Tabela 4.3. Esse DFG foi extraído da execução do *benchmark* HPL com 8 processos. A primeira linha é composta pelo número de vértices, número de arestas e por um identificador da ferramenta Metis, respectivamente. Logo, a primeira linha da Tabela 4.3 mostra que está sendo considerado um grafo com 8 vértices com 14 arestas e que está sendo levado em conta o peso nas arestas por causa do identificador 1 da ferramenta Metis.

Cada uma das linhas seguintes representam um processo: linha 1 - processo 1; linha 2 - processo 2; e assim sucessivamente. As colunas são compostas por pares: processo e volume de comunicação que representam esse processo com o qual o processo linha está se comunicando e o volume de dados trocados entre eles.

O grafo fornecido para a ferramenta de particionamento é remapeado segundo a disponibilidade de nodos. Por exemplo, para x processos que serão disparados por y máquinas, será estabelecida uma distribuição x/y deles de forma que a comunicação realizada via rede seja minimizada. A ferramenta vai gerar um arquivo de saída, o qual terá a ordem na qual eles devem ser redisparados para que isso aconteça.

Tabela 4.3: Grafo do HPL extraído com a β-MPI.

8-vértices	14-arestas	1-Metis					
proc.	dados	proc.	dados	proc.	dados	proc.	dados
2	1706	3	12	4	1691	5	1580
1	1706	3	1823	6	1776		
1	12	2	1823	4	1857	7	1966
1	1691	3	1857	8	1834		
1	1580	6	1830	7	11	8	1688
2	1776	5	1830	7	1820		
3	1966	5	11	6	1820	8	1969
4	1834	5	1688	7	1969		

Para realizar o disparo dos processos seguindo a ordem sugerida foi utilizado o comando `mpiexec` disponibilizado pelo `lam-mpi` (SQUYRES; LUMSDAINE, 2003; LAM/MPI PARALLEL COMPUTING, 2005). As funcionalidades desse comando são similares as do `mpirun`, porém existem algumas opções que estão disponíveis no `mpirun` que não estão disponíveis no `mpiexec` e vice-versa.

A sintaxe do `mpiexec` é:`$ mpiexec <opções><cmd1>:<cmd2>:...`. Por exemplo para rodar um programa mestre/escravo onde dois executáveis precisam ser lançados, o comando `$ mpiexec -n1 mestre:escravo` vai lançar uma cópia do mestre e uma cópia do escravo para cada uma das CPUs que estiverem disponíveis para o `lam-mpi`.

O argumento passado para a execução do aplicativo com `mpiexec`, conforme o remapeamento, foi o `-host` o qual permite que seja atribuído a uma máquina um determinado processo. Por exemplo, a chamada: `$ mpiexec -n 1 -host nodo programa`, vai lançar uma cópia de "programa"no *host* "nodo".

A execução da aplicação para a geração do grafo, o particionamento pela ferramenta Metis e a execução com os processos remapeados foram automatizados através de um *script* chamado `betarun`, vide Figuras 4.10, 4.11 e 4.12 (este *script* foi adaptado para a execução do HPL. Logo é necessário apenas executar a aplicação através da chamada do *script* passando como parâmetros o nome do aplicativo, o número de processos que serão utilizados e o número de partições. O *script* é composto por três partes:

- a primeira parte diz respeito ao disparo da primeira execução do programa MPI usando o lam-mpi. Os comandos usados são o `lamboot` para inicialização e o `mpirun` para o disparo da execução efetivamente. E a saída da execução do aplicativo, isto é o DFG, é armazenado em um arquivo de saída com o nome do aplicativo e extensão `.stdout`;

- a segunda parte trata do particionamento do DFG do programa. Primeiramente o arquivo de saída com a descrição do grafo é aberto para que apenas as informações do DFG do aplicativo sejam retiradas e armazenadas em um outro arquivo, agora no formato de arquivo de entrada da ferramenta Metis. Este procedimento é realizado porque o arquivo de saída gerado pela execução do `mpirun` possui algumas outras informações relativas ao aplicativo, as quais não são interpretadas pela ferramenta Metis. Depois o grafo é particionado através de um dos dois programas que o Metis

```
1  @listnodes=();
2  $cont=0;
3  $NODEFILE="/home/gppd/rennes/nodes";
4  $LAMHOME="/home/gppd/rennes/lam";
5  $PROG="$ARGV[0]";
6  $NPROC=$ARGV[1];
7  $NPARTICOES=$ARGV[2];
8  ETAPAa();
9  ETAPAb();
10 ETAPAc();
11 sub ETAPAa {
12         print "\nETAPA.a->>> lamboot:\nn### mpirun:\n";
13         lamboot();
14         system("$LAMHOME/bin/mpirun -ssi rpi tcp -v -np $NPROC ${PROG} > ${PROG}.stdout");
15 }
16 sub lamboot {
17         system("$LAMHOME/bin/lamboot -v $NODEFILE");
18 }
```

Figura 4.10: Script da β-MPI- parte 1

oferece `kmetis` e `pmetis` - o primeiro particiona grafos em até 8 partições com melhor qualidade enquanto o segundo particionamento é melhor para mais de 8 partições;

- e a terceira parte descreve como redisparar os processos segundo a ordem sugerida pela ferramenta Metis. Nessa etapa, primeiro a lista de nodos é recuperada, depois o arquivo particionado é aberto para o redisparo dos processos. Em seguida, da linha 19 à linha 25 há um controle para o caso de existirem mais processos do que máquinas para serem utilizadas. Após toda a lista de nodos ter sido percorrida, os argumentos terem sido organizados é feito o redisparo através de um outro comando de execução disponibilizado pelo LAM/MPI, chamado de `mpiexec`. Este comando foi utilizado porque ele permite que seja indicado qual processo será disparado em tal máquina. Por exemplo, na execução `mpiexec - n 1 -host nodo4 programa : -n 1 -host nodo2 programa`, será disparado o primeiro processo do programa no nodo4 e o segundo processo do programa no nodo2. Se fosse usado o disparo através `mpirun`, seria preciso obedecer a ordem do arquivo de máquinas. E para os dois comandos `mpiexec` e `mpirun`, se existirem mais processos do que máquinas os processos seguem uma fila circular após as *n* máquinas terem recebido um processo para execução.

Esta integração com a distribuição LAM/MPI foi realizada para que futuramente seja integrado um recurso de migração de processos na β-MPI, dado que a LAM/MPI possui funcionalidades de *checkpoint* porém ainda não tem oficialmente o suporte a migração de processos. O uso desse recurso tornará possível a disponibilização de um serviço de escalonamento dinâmico, dado que o redirecionamento de tarefas passa a acontecer durante a execução.

4.3 β-MPI versus Outras Ferramentas

O objetivo da β-MPI é gerar o DFG de uma aplicação e com base nessas informações distribuir os processos de forma eficiente. O programador executa a sua aplicação uma vez para que seja gerado o DFG e para que o escalonador, representado pela ferramenta Metis, use as informações do grafo para buscar diminuir a comunicação entre processos

```
sub ETAPAb {
        makeINPUTmetis();
        runmetis();
        backupmetis();
}
sub runmetis {
        while(!open(datas,'<',"$PROG.metis")){}
            if ($NPARTICOES > 8){
                print "\n kmetis partitioning\n";
                system( "kmetis ${PROG}.metis $NPARTICOES");
            }
            else{
                print "\n pmetis partitioning \n";
                system( "pmetis ${PROG}.metis $NPARTICOES");
            }
}
sub makeINPUTmetis {
  open(datas,'<',"$PROG.stdout");
  while(chop($line=<datas>)) {
        if ($line eq "### graph info begins ###") {
                open(outfile,'>',"$PROG.metis");
                while($graphline ne "### graph info ends ###") {
                        chop($graphline=<datas>);
                        if ($graphline ne "### graph info ends ###") {
                        printf outfile "$graphline\n";
                }
        }
   }
  }
  close(outfile);
}
```

Figura 4.11: Script da β-MPI- parte 2

alocados em máquinas distintas. Depois disso, a aplicação é disparada seguindo a ordem sugerida pelo escalonador e, ao final, pode-se comparar as execuções e verificar se houve melhora no desempenho do programa.

Existem algumas diferenças entre a β-MPI e as ferramentas apresentadas no Capítulo 2. A ferramenta Cilk restringe o grafo ao tipo divisão e conquista através do modelo de memória compartilhada e a técnica de escalonamento empregada é baseada no roubo de tarefas. Para a biblioteca β-MPI, a geração do DFG é para qualquer tipo de grafo no contexto de troca de mensagens, utilizando como técnica de escalonamento a minimização de comunicação.

O Pyrros é um sistema que parte de uma descrição de tarefas que são escalonadas primeiramente e fornece como saída um código que utiliza troca de mensagens para comunicação. A abordagem tratada pela β-MPI vem em direção oposta, parte-se de uma descrição de código que usa troca de mensagens, gera-se o DFG do programa e escalonam-se os processos.

No caso do ambiente CASCH, ele paraleliza códigos seqüenciais através de DFGs, usa troca de mensagem e permite ao usuário escolher qual técnica de escalonamento utilizar através de execuções sucessivas e comparações de desempenho. Esta abordagem é estática, pois as decisões de escalonamento são feitas *off-line*, para que depois de escolhida a técnica a aplicação seja executada. A β-MPI também usa o escalonamento estático, porém é focada para aplicações paralelas que utilizam troca de mensagens seguindo o padrão MPI.

```
1 sub ETAPAc { print "\nETAPA.c->>> Recupera Lista de Nos:\n";
2         getlist();
3         makeargs();
4         mpiexec();
5         lamhalt();
6 }
7 sub makeargs {
8         ### recupera particionamento do grafo por METIS
9         $endlist=`wc -l $PROG.metis.part.$NPARTICOES
10        open(datas1,'<',"$PROG.metis.part.$NPARTICOES");
11        $ct=0;
12        while(chop($nodemap=<datas1>)) {
13                if($nodemap >= $cont){
14                        $var_1= int $nodemap / $cont;
15                        while($var_1 >= 2){$var_1= int $var_1/$cont;}
16                        $mod_1=$nodemap % $cont;
17                        $ret_cont = $var_1 + $mod_1 -1;
18                }
19                else{ $ret_cont = $nodemap;}
20                $ct=$ct+1;
21                if ($endlist == $ct) {
22                        $ARGS="$ARGS -n 1 -host $listnodes[$ret_cont] ${PROG}";
23                } else {
24                        $ARGS="$ARGS -n 1 -host $listnodes[$ret_cont] ${PROG} :";
25                }
26        }
27 }
28 sub mpiexec {
29        lamboot();
30        system( "$LAMHOME/bin/mpiexec -ssi rpi tcp $ARGS ${PROG} > ${PROG}.RE.stdout");
31 }
32 sub lamhalt { system( "$LAMHOME/bin/lamhalt");}
33 #### recupera a lista de nodes ativos
34 sub getlist {
35        `lamnodes -n > /tmp/lamnodes`;
36        open(datas,'<',"/tmp/lamnodes");
37        while(chop($line=<datas>)) {
38                ($node) = split(/:/,$line);
39                $listnodes[$cont]=$node;
40                print "\t$listnodes[$cont]:";
41                $cont=$cont+1;}
42 }
```

Figura 4.12: Script da β-MPI- parte 3

4.4 Resumo

Este capítulo apresentou a proposta e a implementação da biblioteca de escalonamento estático para aplicações MPI: a β-MPI. O objetivo da β-MPI é utilizar as informações obtidas pelo DFG de um programa MPI para poder mapear os processos eficientemente otimizando o uso de recursos para a execução da aplicação.

Do ponto de vista do programador, será preciso apenas instalar a biblioteca, incluir o *header* `beta-mpi.h` na sua aplicação e compilá-la novamente. Após a execução do programa com o *script*, pode-se comparar as execuções com e sem o remapeamento.

Cabe ressaltar que o esquema de mapeamento de processos proposto reduz o escopo de programas adequados ao uso com a β-MPI. As aplicações têm que manter o mesmo padrão de comunicação quando executadas sucessivamente e não podem ter variações de dados de entrada durante a execução.

O uso do redisparo dos processos, utilizando o LAM/MPI, permite que futuramente haja uma integração de um sistema de escalonamento dinâmico na β-MPI. Isto porque o LAM/MPI possui funcionalidades de migração de processos, mas não tem, oficialmente, o suporte a migração de processos. Assim, será possível o uso de aplicações dinâmicas.

Nos próximos capítulos são apresentadas as aplicações que foram utilizadas para validar a biblioteca β-MPI. As primeiras aplicações, tratadas no capítulo seguinte, foram a FFT tal como implementada no pacote FFTW e a transferência de calor sobre uma placa plana utilizando a forma adotada na dissertação de Guilherme Galante(GALANTE, 2006) onde ele usou o método Aditivo de Schwarz e multigrid para resolvê-la. Na seqüência, foi feita uma abordagem detalhada sobre a fatoração LU implementada no *benchmark* HPL e a validação com a β-MPI.

5 VALIDAÇÃO DA β-MPI- DUAS APLICAÇÕES

Este capítulo e o próximo tratam da utilização da β-MPI em algumas aplicações. Inicialmente foram escolhidas duas: a FFT tal como disponibilizada na FFTW (FASTEST FOURIER TRANSFORM IN THE WEST, 2003) e o problema clássico de Transferência de Calor sobre uma placa plana utilizando o Método Aditivo de Schwarz resolvido através de multigrid (GALANTE, 2006). No capítulo seguinte será explorada a terceira aplicação - a fatoração LU implementada no HPL (SILVA et al., 2005). O objetivo dos capítulos é mostrar como a β-MPI pode auxiliar no mapeamento de tarefas para essas aplicações e comparar o desempenho antes e depois do remapeamento.

O capítulo está dividido em 3 seções. A primeira seção, a qual trata da Transformada Discreta de Fourier em termos de conceitos e características, aborda a biblioteca FFTW e suas otimizações e apresenta a avaliação experimental da FFTW e com β-MPI. A segunda seção trata do problema de transferência de calor, apresenta o Método Aditivo de Schwarz, explica sua solução através de multigrid e mostra a avaliação de desempenho das execuções do aplicativo com a β-MPI. A última seção resume o capítulo.

5.1 Validação da β-MPI com a Aplicação *Fast Fourier Transform* - FFT

Existem implementações da FFT tanto em aplicações (síntese e reconhecimento de voz) quanto em *benchmarks* (NAS, benchFFT) e bibliotecas como a FFTW. Este capítulo apresenta a Transformada Rápida de Fourier tal como implementada na biblioteca *Fastest Fourier Transform in the West* (FFTW) e sua utilização com a β-MPI. O objetivo é mostrar como a β-MPI pode auxiliar no mapeamento de tarefas para a FFTW e comparar o desempenho antes e depois do remapeamento.

5.1.1 Transformada Discreta de Fourier

Um sinal discreto no tempo é representado através da amostragem de um sinal contínuo no tempo em intervalos uniformes $t = nT$, onde T é o intervalo de tempo e n é um inteiro. Este sinal é uma sequência de números representados pela Eq. 5.1:

$$X(nT) = X(t)|_{t=nT} \qquad ,0 \leq n < \infty. \tag{5.1}$$

A Transformada Discreta de Fourier (DFT) (BRIGHAM, 1988) serve para obter seqüências em freqüência a partir de sequencias temporais de um sinal discreto. Seja um sinal discreto no tempo $x(n)$ composto por N amostras, a DFT desse sinal é dada pela Eq. 5.2 de onde se obtém o sinal discreto em freqüência dado por $X(k)$. A partir da DFT pode-se obter a Transformada Inversa Discreta de Fourier pela Eq. 5.3.

$$X(k) = \sum_{n=0}^{N-1} X(n) e^{-j(2\pi/N)kn} \tag{5.2}$$

$$\widetilde{X}(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j(2\pi/N)kn} \tag{5.3}$$

onde $e^{j2\pi/N}$ e $e^{-j2\pi/N}$ constituem os fatores cíclicos da DFT, os quais são periódicos e definem pontos no círculo unitário do plano complexo.

O desempenho do algoritmo de cálculo da DFT é fortemente afetado pelo volume de cálculo de senos e cossenos. Uma forma de otimização para este tipo de algoritmo, apesar de consumir mais memória, é a criação de uma tabela de senos e cossenos em uma matriz. Dessa forma, as informações são recuperadas sem a necessidade de novos cálculos.

Para evitar o enorme volume de processamento necessário para o cálculo de uma DFT, utiliza-se o chamado mapeamento de índices desenvolvido por Cooley-Tukey (BRIGHAM, 1974), ou troca de variáveis, o qual deu origem à FFT. O objetivo deste mapeamento é que uma DFT de uma dimensão seja transformada em uma DFT de duas ou mais dimensões. Dessa idéia, surgiram vários métodos como o desenvolvido por Danielson e Lanczos, o qual mostra que o cálculo de uma DFT de tamanho N pode ser feito através de duas amostras da DFT original de tamanho $N/2$, sendo que uma das DFTs é formada pelos pontos pares enquanto que a outra é formada pelos pontos ímpares, conforme demonstra a Eq. 5.4.

$$\begin{array}{rcl} F_k & = & \sum_{L=0}^{N-1} e^{j2\pi Lk/N} f_L \\ & = & \sum_{L=0}^{N/2-1} e^{j2\pi(2L)k/N} f_{2L} \quad + \quad \sum_{L=0}^{N/2-1} e^{j2\pi(2L+1)k/N} f_{2L+1} \\ & = & \sum_{L=0}^{N/2-1} e^{j2\pi Lk/(N/2)} f_{2L} \quad + \quad e^{j2\pi k/N} \sum_{L=0}^{N/2-1} e^{j2\pi Lk/(N/2)} f_{2L+1} \\ & = & F_k^{par} + e^{j2\pi k/N} F_k^{impar} \end{array} \tag{5.4}$$

onde F_k^{par} indica o k-ésimo componente da transformada de Fourier de tamanho $N/2$ formada pelos componentes pares da transformada original f_L, enquanto que F_k^{impar} corresponde a transformada de tamanho $N/2$ formada pelos componentes ímpares. A grande vantagem da Eq. 5.4 é que ela pode ser usada recursivamente até alcançar o ponto de ter uma transformada de apenas 2 termos. A aplicação de sucessivas divisões entre pares e ímpares gera uma transformada de 1 termo. A próxima etapa é descobrir o valor de n no conjunto original (par-ímpar). Pela substituição de p por 0 e i por 1 obtém-se um número binário. O valor de n é exatamente este número com a ordem dos bit invertida. Isto acontece porque as subdivisões sucessivas em pares e ímpares são testes sucessivos de bits de ordem menos significativa de n. Por exemplo, na transformada de Fourier de 8 pontos f(0), f(1), f(2), f(3), f(4), f(5), f(6), f(7), a primeira divisão entre pares e ímpares fornece dois conjuntos f(0), f(2), f(4), f(6) e f(1), f(3), f(5), f(7). A segunda divisão gera quatro conjuntos f(0), f(4), f(2), f(6), f(1), f(5) e f(3), f(7). Na Tabela 5.1 pode-se comparar os índices e a ordem dos bits.

O algoritmo da FFT é composto por duas etapas. Na primeira etapa, os dados são reordenados e na segunda, para cada potência de 2, os valores das transformadas parciais são calculados. A combinação dos valores em n pode ser expressa em um fluxograma denominado *Butterfly*, que representa a soma entre os sinais no domínio frequência. A estrutura resultante para o algoritmo da FFT é com 3 loops concatenados: sendo o primeiro um loop externo para os "m"(Para 8 amostras $m = 3$ pois $n = 2^3 = 8$) etágios; o

Tabela 5.1: Reordenação através da inversão de bits

original	índice	reordenado	índice
f(0)	000	f(0)	000
f(1)	001	f(4)	100
f(2)	010	f(2)	010
f(3)	011	f(6)	110
f(4)	100	f(1)	001
f(5)	101	f(5)	101
f(6)	110	f(3)	011
f(7)	111	f(7)	111

segundo um loop intermediário para realizar a combinação em cada par; e por último um loop interno para a combinação dos valores em um único componente de frequência.

Cada combinação só acessa uma vez cada par de amostras a cada iteração do loop externo. Isto permite que os resultados da combinação possam sobrescrever os valores que deram origem ao cálculo. Tal conceito chama-se cálculo *in-place*. Este conceito é relativo à utilização de memória, pois não é necessário que seja reservado um espaço para os dados de saída diferente dos dados de entrada. Os dados de saída sobrescrevem os dados de entrada.

A vantagem da FFT está na velocidade de processamento comparado ao método de cálculo da DFT diretamente, que caiu de N^2, para $Nlog_2N$ operações com o uso destes métodos. Existem outras variações do algoritmo da FFT, por exemplo, quando se realiza o cálculo antes da reordenação o algoritmo é chamado de Sande-Tukey ao invés de Cooley-Tukey. Outros algoritmos permitem que se façam subdivisões até um tamanho N especificado como base. Por exemplo problemas base-4 e base-8 subdividem um problema até atingerem $N = 4$ e $N = 8$, respectivamente. Existem ainda algoritmos onde o valor de N não precisa ser potência de 2, nos quais a subdivisão é feita através de números primos de N. Nestes dois últimos casos é preciso saber balancear a complexidade de implementação do algoritmo com o ganho de desempenho obtido. Esses algoritmos são utilizados em muitas aplicações e bibliotecas.

5.1.2 *Fastest Fourier Transform in the West* - FFTW

A FFTW (FASTEST FOURIER TRANSFORM IN THE WEST, 2003) é uma biblioteca formada por um conjunto de rotinas implementadas com a linguagem C de programação, para o cálculo da DFT de uma ou mais dimensões e utiliza-se como entrada tanto dados reais como complexos. A FFTW calcula as transformadas em paralelo, seja em sistemas de memória compartilhada, ou em sistemas de memória distribuída. Nesse trabalho será considerado apenas o cálculo da transformada em sistemas de memória distribuída.

A FFTW disponibiliza 4 modalidades de operação. A primeira é composta pelas transformadas de uma dimensão de dados complexos (FFTW), a segunda pelas transformadas de múltiplas dimensões de dados complexos (FFTWND), a terceira pelas transformadas de uma dimensão de dados reais (RFFTW) e a quarta pelas transformadas de múltiplas dimensões de dados reais (RFFTWND). Exceto a modalidade RFFTW, as demais também estão disponíveis para as rotinas de cálculo da transformada em sistemas de memória distribuída implementadas em MPI.

5.1.2.1 MPI FFTW

As rotinas da FFTW desenvolvidas para sistemas com memória distribuída são diferentes das versões originais da FFTW. Essas diferenças ocorreram pelo fato dos dados a serem usados na transformada estarem distribuídos pelos processos e cada um desses receber uma parte do vetor. Porém, a utilização das primitivas é similar a FFTW para sistemas monoprocessados.

As primitivas da MPI_FFTW devem ser chamadas na mesma ordem por todos os processos envolvidos no cálculo da transformada. Os dados das transformadas são distribuídos conforme o número de linhas (primeira dimensão) de dados. Cada processo recebe um subconjunto de linhas, por exemplo, no caso da transformada 3D $128 \times 128 \times 128$, o conjunto de dados pode ser distribuído em até 128 processadores. Para o cálculo da transformada em sistemas distribuídos, primeiro cada processo calcula a transformada dos seus dados locais - as linhas. Depois, cada processo realiza a transposição dos dados para pegar a dimensão local restante - as colunas. Nessa dimensão é feito o cálculo da transformada de Fourier e os dados podem ser transpostos de volta à forma original.

A transposição no cálculo da FFTW distribuída é a parte mais difícil. A transposição foi projetada assumindo que a comunicação entre processos poderia ser muito cara e, portanto, ela realiza um número de permutações com os dados locais para trocar o mínimo de número de blocos de dados entre os processos. Os autores consideram que a transformada é calculada mais eficientemente quando as primeiras duas dimensões de dados são divisíveis pelo número de processos.

Cada processo com o seu subconjunto de dados calcula a FFT local, depois realiza a transposição dos dados - comunicação. Após a transposição, os processos calculam a FFT das dimensões restantes e o volume de dados transpostos entre os processos.

Dois pontos relativos ao uso das primitivas MPI_FFTW devem ser considerados para se obter um bom desempenho. Primeiro, tentar escolher valores para a primeira e para a segunda dimensão que sejam divisíveis pelo número de processadores alocados - no caso de ser possível para apenas uma dimensão, prefere-se que ele seja atribuído à primeira dimensão. Essa escolha possibilita uma distribuição de carga mais homogênea e reduz a complexidade de comunicação e atraso. Segundo, o uso do formato de saída configurado pelo flag `FFTW_TRANSPOSED_ORDER` possibilita um maior ganho em termos de custo de comunicação.

5.1.3 Avaliação Experimental

A avaliação experimental teve como objetivo verificar a influência no desempenho do *benchmark* obtida com remapeamento de processos sugerido pela β-MPI. Nesses experimentos foi utilizado o aplicativo `fftw_mpi_test` no cluster labtec-UFRGS. O cluster é composto por um servidor com 2 processadores Xeon 1.8GHz -1GB de RAM e 20 nodos Dual Pentium III 1.3GHz - 1GB de RAM, do qual 4 nodos foram alocados. A rede utilizada foi a *Fast Ethernet*. O sistema operacional usado foi Linux Gentoo.

A execução do programa `fftw_mpi_test` em conjunto com a β-MPI possibilitou identificar o grafo de comunicação, o qual para essa aplicação possui como característica a regularidade nas trocas de mensagens devido ao algoritmo da FFT. Uma FFT bidimensional - 128×38 foi executada e com 8 processos que foram distribuídos em 4 nodos do cluster seguindo a fila circular de distribuição padrão do MPI. Após a segunda execução, os processos foram remapeados pela β-MPI e foram distribuídos conforme mostra a representação na Figura 5.1.

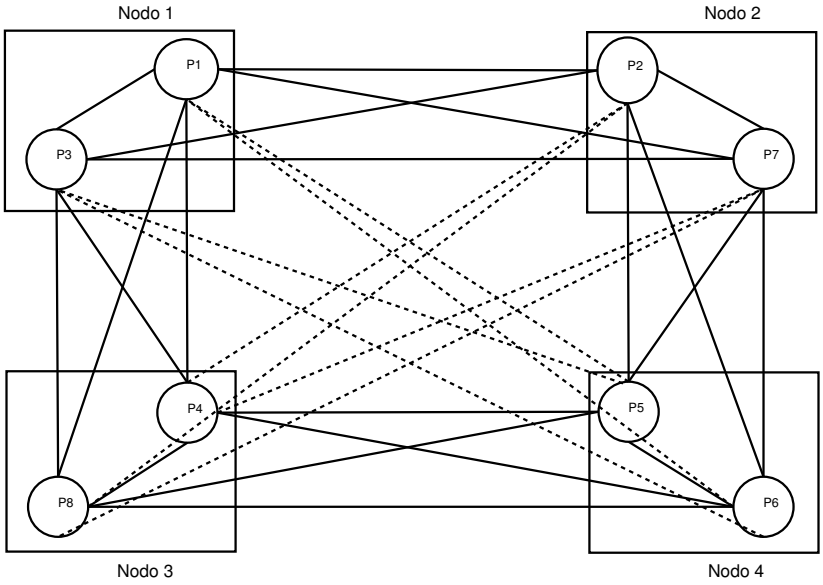

Figura 5.1: Grafo com os processos mapeados pela β-MPI- 2D 128×38.

No grafo da Figura 5.1 cada vértice está conectado aos demais por causa da primitiva `MPI_AlltoAll` utilizada na realização da transposição dos dados da FFT. Devido a segunda dimensão da FFT não ser múltipla do número de processos (8), o vértice representado pelo processo P8 troca menos informações com todos os demais como mostra a tabela 5.2. Essa tabela está no formato padrão Metis, descrito no capítulo 3.

Tabela 5.2: Grafo β-MPI.

8	28	1											
2	125kb	3	125kb	4	125kb	5	125kb	6	125kb	7	125kb	8	100kb
1	125kb	3	125kb	4	125kb	5	125kb	6	125kb	7	125kb	8	100kb
1	125kb	2	125kb	4	125kb	5	125kb	6	125kb	7	125kb	8	100kb
1	125kb	2	125kb	3	125kb	5	125kb	6	125kb	7	125kb	8	100kb
1	125kb	2	125kb	3	125kb	4	125kb	6	125kb	7	125kb	8	100kb
1	125kb	2	125kb	3	125kb	4	125kb	5	125kb	7	125kb	8	100kb
1	125kb	2	125kb	3	125kb	4	125kb	5	125kb	6	125kb	8	100kb
1	100kb	2	100kb	3	100kb	4	100kb	5	100kb	6	100kb	7	100kb

O grafo da FFT, por ser regular, todos os processos têm o mesmo número de comunicações com os demais, e isto é um caso em que o uso da β-MPI apenas serve para extrair o grafo do programa. O remapeamento não surtiu efeito em termos de minimização do corte de arestas, pois o número de arestas é o mesmo entre os processos e todos se comunicam com todos.

O passo seguinte foi escolher uma outra aplicação que tivesse um grafo de comunicação irregular. Essa característica é encontrada na abordagem feita por Guilherme Galante na sua dissertação de mestrado, onde ele utilizou Método Aditivo de Schwarz e Multigrid para resolver o clássico problema de transferência de calor sobre uma placa plana.

5.2 Validação da β-MPI com o Problema Transferência de Calor

O problema de transferência de calor sobre uma placa plana é uma aplicação tradicional de métodos numéricos. A transferência de calor em uma placa retangular, com os lados submetidos a temperaturas distintas T_1, T_2, T_3 e T_4, como ilustra a Figura 5.2, ocorre através da troca de calor entre partículas do material de um ponto com mais energia para outro com menos.

A temperatura inicial para cada uma das bordas é distinta e a cada instante de tempo a temperatura varia para os pontos da placa. Para este experimento foram atribuídas temperaturas de $T_1 = 1^oC$ e $T_2 = T_3 = T_4 = 0^oC$.

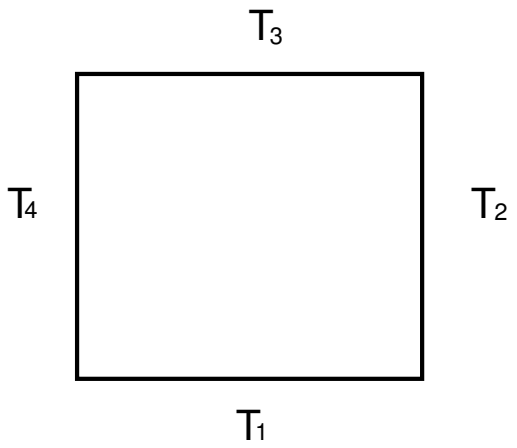

Figura 5.2: Placa plana submetida a diferentes temperaturas.

A abordagem adotada por Guilherme Galante em sua dissertação (GALANTE, 2006) para solucionar o problema foi utilizar o Método Aditivo de Schwarz resolvido por meio de Multigrid. O Metodo Aditivo de Schwarz é um método de decomposição de domínio o qual caracteriza-se pela necessidade de pouca comunicação, sendo, em geral, restrita às fronteiras dos subdomínios enquanto que o método Multigrid é um esquema para resolver um dado problema através do emprego de correção por refinamento de malhas. No processo de discretização de uma equação diferencial parcial, uma malha é o mapeamento de um domínio em uma estrutura composta por um número finito de pontos. Os passos adotados para a resolução do problema, como mostra a Figura 5.4 são divididos em quatro atividades:

- geração e particionamento de malhas;
- criação da hierarquia de malhas;
- montagem dos sistemas de equações lineares;
- resolução dos sistemas em paralelo.

Para a geração e o particionamento de malhas foram utilizadas malhas não-estruturadas triangulares, vide Figura 5.3. A geração da malha foi feita através dos pacotes *Triangle* e o *Easymesh* e como saída eles ofereceram: as coordenadas de cada vértice da malha; a conectividade dos triângulos (quais nodos formam um dado triângulo); e a vizinhança de

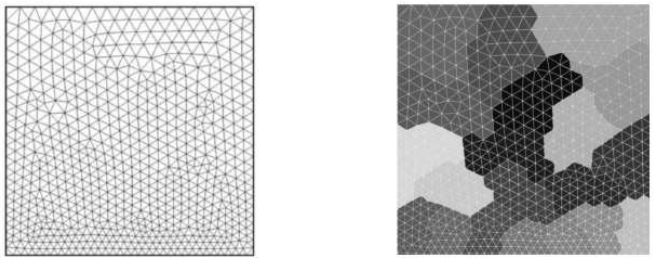

Figura 5.3: Malha original e malha particionada em 20 subdomínios (GALANTE, 2006).

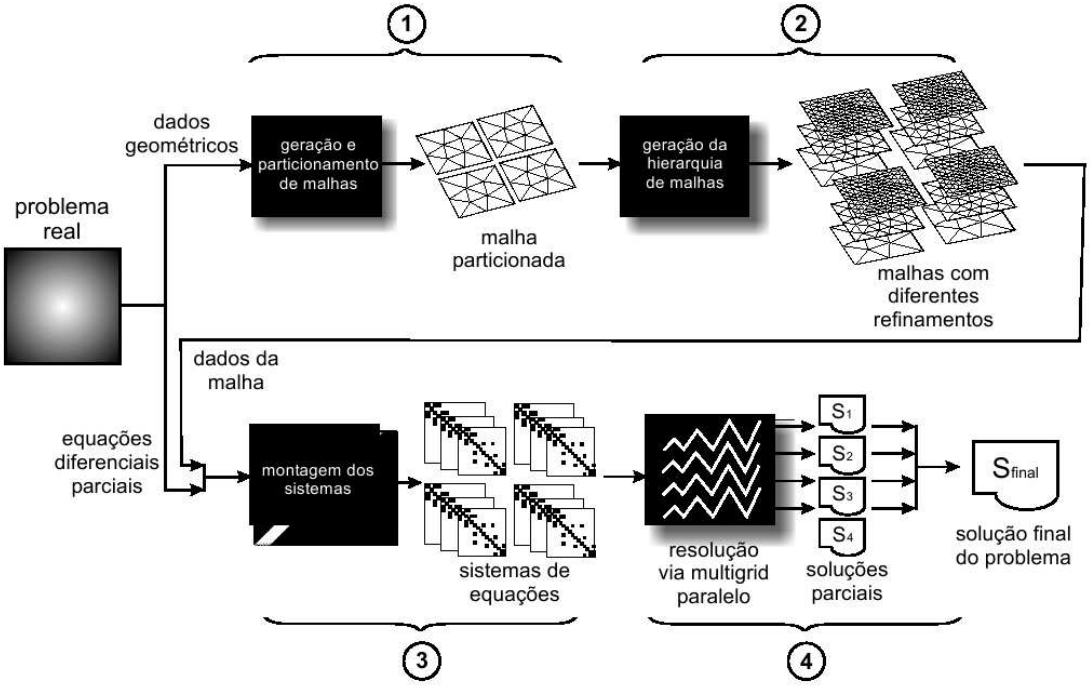

Figura 5.4: Passos para solucionar o problema (GALANTE, 2006).

cada triângulo. O particionamento da malha foi feito através dos programas disponíveis na ferramenta Metis, os quais geram uma lista das respectivas partições de cada triângulo.

A hierarquia de malhas foi definida em 4 níveis de refinamento. Para isso foi implementado um módulo chamado MGTool capaz de gerar malhas com diferentes refinamentos.

A montagem dos sistemas de equações lineares foi feita através da aplicação de um estêncil computacional que é utilizado para indicar a posição dos pontos presentes em uma equação diferencial parcial. O armazenamento da matriz esparsa foi feito através do formato CSR o qual caracteriza-se por armazenar os elementos da matriz em posições contíguas de memória.

Como este trabalho trata de otimizar aplicações MPI através do remapeamento de processos com vistas à diminuição de tráfego de dados pela rede, portanto será tratada apenas a atividade de resolução dos sistemas em paralelo. No tratamento dessa atividade será enfatizada a parte de comunicação.

5.2.1 Esquema de Comunicação da Aplicação

Depois do particionamento e da expansão dos subdomínios para criação da região de sobreposição, necessária para o método aditivo, cada processador fica responsável pela geração dos sistemas de equações locais. Para a geração desses sistemas foi adotada uma numeração local, sendo que as células pertencentes à região de sobreposição são numeradas depois das células internas.

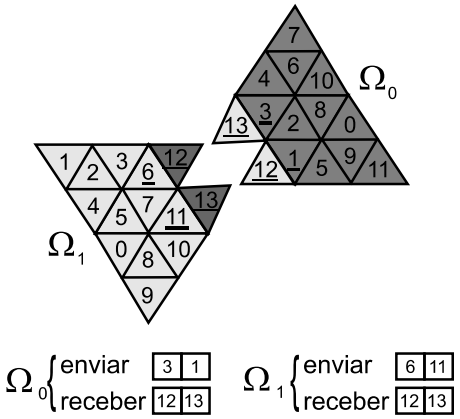

Figura 5.5: Estrutura de dados para a comunicação no aditivo de Schwarz (GALANTE, 2006).

O próximo passo é determinar as estruturas necessárias para a coordenação da comunicação entre os processos. Tais estruturas consistem em uma lista que armazena a identificação dos subdomínios vizinhos, e mais duas listas para cada vizinho. A primeira delas contém as posições do vetor solução a serem enviadas no fim de cada iteração. A segunda lista, contém as posições do vetor de termos independentes que receberão os valores calculados no subdomínio vizinho. A Figura 5.5 ilustra o domínio Ω particionado e expandido com uma célula de sobreposição. As posições dos elementos sublinhados duplamente são aquelas que serão enviadas para o subdomínio vizinho, e as que possuem sublinhados simples correspondem às posições que receberão os dados.

Com estas informações cada processo calcula a solução do sistema de equações referente ao seu subdomínio. O algoritmo está descrito na Figura 5.6.

Para a implementação do algoritmo mostrado foram utilizadas as seguintes primitivas MPI: de difusão `MPI_Bcast()` para que os processos recebessem as informações sobre as condições iniciais - níveis de refinamento, partições; primitivas de envio e recepção - `MPI_Send()` e `MPI_Receive()` referente a comunicação dos dados das fronteiras; e de redução `MPI_Allreduce()` para a soma das soluções u_i^k até o erro máximo definido e atualização de todos os processos com essa informação.

5.2.2 Avaliação Experimental

Nessa avaliação foi utilizado o cluster labtec tal como apresentado na seção anterior, com 8 nodos disponíveis. Os experimentos tiveram como cenários 2, 4 e 8 nodos. Para as execuções dos testes foram utilizados 4 níveis de malha com 1337, 5348 e 21392 e 85568 triângulos respectivamente. Para cada cenário, a aplicação foi executada 3 vezes. No primeiro cenário foram disparadas execuções com 4, 8 e 10 processos. Em todos eles houve uma significativa melhora no tempo de execução da aplicação e para esse cenário será discutido o caso com 4 processos disparados em 2 nodos. A Figura 5.7 ilustra o grafo

Inicialização: escolha uma solução inicial $u_i^{(0)}$ *para cada subdomínio* Ω_i*;*
defina o erro máximo para a solução ε*;*

1. resolva o sistema com a solução inicial $u_i^{(0)}$

2. $k = 1$*;*

do

3. enviar dados das fronteiras da iteração $u_i^{(k-1)}$ *para os vizinhos;*

4. receber dados das fronteiras da iteração $u_i^{(k-1)}$ *dos vizinhos;*

5. resolver o sistema utilizando os novos valores de fronteira recebidos;

6. obter $u_i^{(k)}$;

7. k *++;*

until $max\left(\left\|u_i^{(k)} - u_i^{(k-1)}\right\|_2 / \left\|u_i^{(k)}\right\|_2\right) \leq \varepsilon$

Figura 5.6: Algoritmo do método aditivo de Schwarz (GALANTE, 2006).

antes do mapeamento e mostra o mesmo grafo após a segunda execução do aplicativo. Esse grafo foi gerado pela comunicação de primitivas `MPI_Send`, `MPI_Receive` e `MPI_Allreduce`. Cabe notar que no primeiro grafo o volume de dados trafegado pela rede era de $Vol = 820 + 1058 + 2 + 1126 = 3006$ bytes e depois do mapeamento esse volume passou para $Vol = 680 + 2 + 750 + 1058 = 2490$ bytes o que fez reduzir para 17% o volume de mensagens trocadas. Isso significa que os subdomínios que têm uma fronteira mais extensa representados pelos processos P1 e P2, mapeados no nodo 1, e pelos Processos P3 e P4, mapeados no nodo 2, deixaram de usar a rede para se comunicar, mostrando que o corte de arestas diminui significativamente após a segunda execução da aplicação. O tempo de execução da aplicação passou de $396,65$ segundos para $370,70$ segundos, gerando um ganho de 7%. A tabela 5.3 mostra os tempos, os desvios padrões e o ganho das execuções com o cenário composto por 2 nodos.

Tabela 5.3: Mapeamento de tarefas com a β-MPI.

Nodos	Processsos	tempo sem a β-MPI	Desvio Padrão	tempo com a β-MPI	Ganho de tempo	Desvio Padrão
2	4	396,65s	0,679	370.70s	7 %	0,858
2	8	277,61s	0,631	274,92s	1 %	0,748
2	10	224,16s	0,461	217,64s	3%	0,206

No segundo cenário com 4 nodos, as execuções envolveram 8, 16, 22, 32 e 38 processos. Para caso com 8 processos o grafo gerado após o remapeamento da β-MPI está na Figura 5.8. Esse remapeamento fez o tempo de execução cair de 143,77 segundos para 135,57 segundos, gerando uma redução de tempo de 6%. Mais uma vez a diminuição no corte de arestas resultou em ganho para a aplicação. A tabela 5.4 mostra os resultados para os casos testados neste cenário.

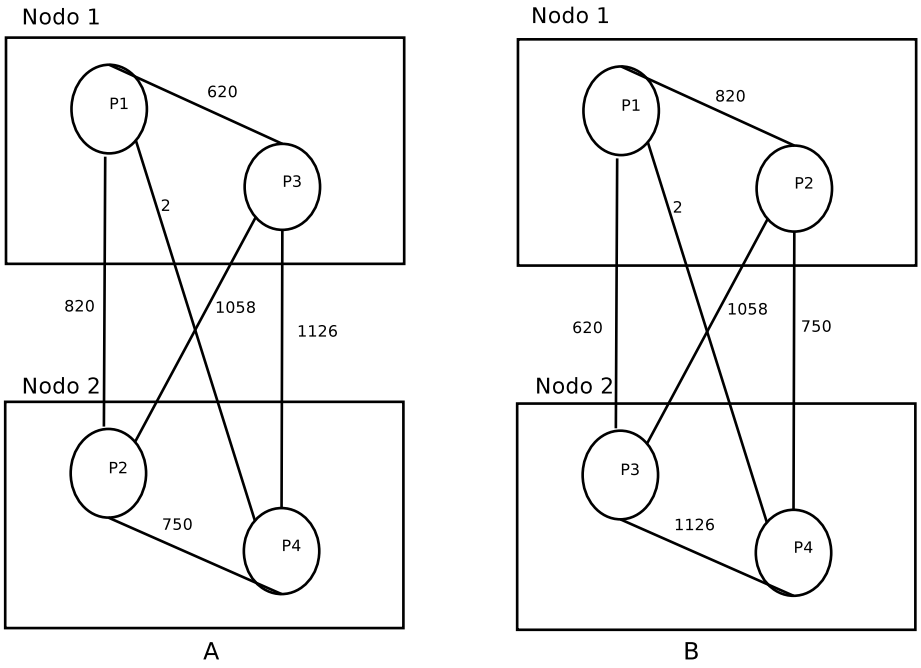

Figura 5.7: A - Grafo com os processos distribuidos seguindo a fila circular do MPI. B - Grafo com os processos distribuídos segundo a β-MPI.

Tabela 5.4: Mapeamento de tarefas com a β-MPI.

Nodos	Processsos	tempo sem a β-MPI	Desvio Padrão	tempo com a β-MPI	Ganho de tempo	Desvio Padrão
4	8	143,77s	0,373	135,57s	6%	0,382
4	16	73,66s	0,247	68,07s	8%	0,459
4	22	55,51s	0,199	55,06s	0,8%	0,252
4	32	42,80s	0,201	42,17s	1,5%	0,297
4	38	48,80s	0,297	48,45s	0,5%	0,203

Tabela 5.5: Mapeamento de tarefas com a β-MPI.

Nodos	Processsos	tempo sem a β-MPI	Desvio Padrão	tempo com a β-MPI	Ganho de tempo	Desvio Padrão
8	16	37,58s	0,245	37,30s	0,7%	0,267
8	32	22,83s	0,021	22,57s	1,1%	0,078
8	38	24,48s	0,043	24,05s	1,7%	0,057

No cenário com 8 nodos, as execuções foram disparadas com 16, 32 e 38 processos. O remapeamento gerou uma melhora menos significativa, vide a tabela 5.5. Isso aconteceu pelo fato de ter um aumento no volume de comunicação e pela redução do processamento necessário para cada subdomínio. Mas assim como nos casos anteriores houve uma diminuição no corte de arestas pelo remapeamento dos processos.

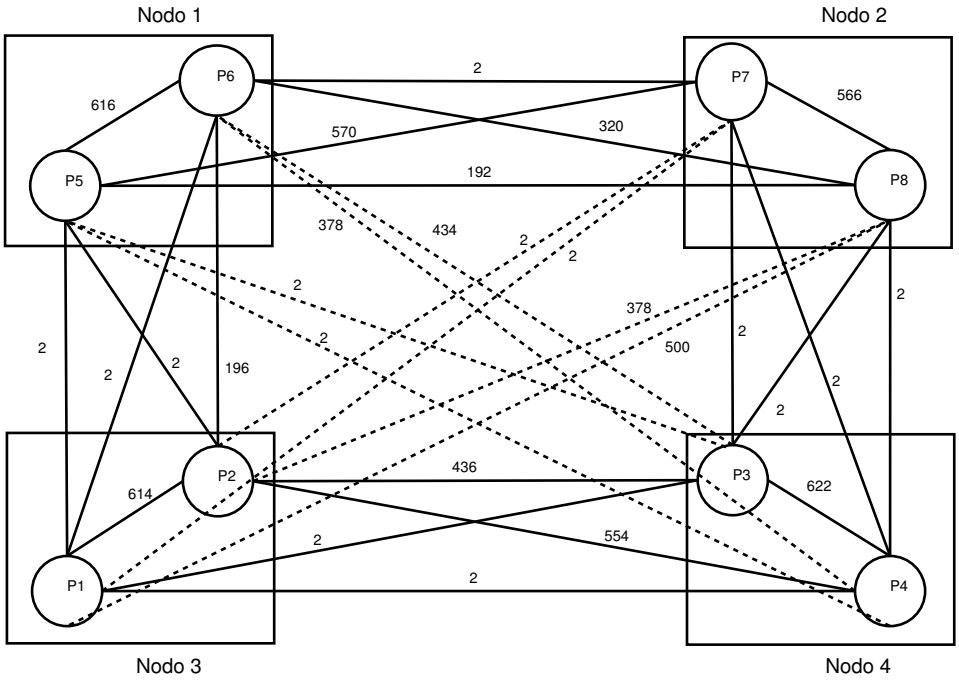

Figura 5.8: Grafo com os 8 processos mapeados em 4 nodos pela β-MPI.

5.3 Resumo

Este capítulo apresentou duas aplicações que foram usadas na validação da β-MPI. A primeira aplicação foi a FFT tal como implementada no pacote FFTW. Nesse caso, foi mostrado o conceito da FFT a seu algoritmo paralelo visando identificar os pontos de comunicação entre os processos. No final dessa parte, foi realizada uma avaliação com o intuito de validar a β-MPI, e nesse caso, percebeu-se que a biblioteca auxilia na extração do grafo de uma aplicação MPI para que este possa ser analisado e depois caso a comunicação entre os processos seja irregular, eles sejam mapeados de forma mais eficiente. Para a FFT, o mapeamento de processo não traz benefícios devido ao fato da comunicação ser regular.

Para suprir o problema do grafo irregular foi escolhida uma outra aplicação - a transferência de calor sobre uma placa plana resolvida pelo uso do Método Aditivo de Schwarz e Multigrid, discutido na dissertação de Guilherme Galante (GALANTE, 2006) onde ele eplora a eficácia dos métodos multigrid na aceleração de métodos iterativos. Essa aplicação teve um resultado interessante porque como cada processo recebe um subdomínio com tamanho distinto de fronteira com os demais, a comunicação pode se tornar um peso significativo no tempo de execução e isso foi visto claramente pela melhora depois do remapeamento com a β-MPI devido a minimização do corte de arestas.

O capítulo seguinte trata da última aplicação utilizada neste trabalho para validar a β-MPI- fatoração LU tal como implementada no HPL. Nesta abordagem, foram explorados os pontos: instalação otimizada do HPL; auxilio da β-MPI para extração do grafo e escolha de algoritmos de *broadcast* e para o mapeamento eficiente dos processos.

6 VALIDAÇÃO DA β-MPI COM A APLICAÇÃO LU - HPL

Neste capítulo é apresentada a fatoração LU disponibilizada pelo *Benchmark* Linpack. Essa aplicação foi escolhida para validar a β-MPI. Três formas de avaliação foram utilizadas: identificação do custo agregado pela intrusão da biblioteca; usando a β-MPI para auxiliar na escolha de algoritmos de roteamento de mensagens e; comparação do desempenho antes e depois do remapeamento de processos.

O capítulo está dividido em 5 seções. A primeira seção trata do algoritmo de fatoração LU. A segunda seção descreve as características do *benchmark Highly Parallel Linpack* (HPL). A seção seguinte, apresenta como foi instalado o HPL e otimização das BLAS no cluster labtec. Na quarta seção é feita a obtenção do DFG do HPL usando a β-MPI sob a perspectiva de diferentes algoritmos de *broadcast* e a seguir são mostrados os resultados dos experimentos realizados. A última seção resume o capítulo.

6.1 O Algoritmo de Fatoração LU

A fatoração LU é uma das técnicas mais utilizadas para resolver equações algébricas. Ela é uma variação do método de eliminação de Gauss. O método de eliminação de Gauss pode ser utilizado para decompor uma matriz dos coeficientes [A], em duas matrizes, a [L] sendo a triangular inferior e a [U] sendo a triangular superior com 1 na diagonal, tal que A=LU.

Dado um sistema A$x = b$, considera-se A uma matriz $n \times n$, x e b sendo vetores $n \times 1$. Os coeficientes da matriz A e do vetor b são $a_{ij}^{(k)}$ e $b_i^{(k)}$ sendo $i = 1...n$ e $j = 1...n$, e tais coeficientes foram obtidos ao final de k passos. O objetivo é encontrar o valor de x_i para $i = 1...n$ tal que

$$\begin{cases} a_{11}^{(0)}x_1 + a_{12}^{(0)}x_2 & \cdots & a_{1n}^{(0)}x_n & = b_1 \\ a_{21}^{(0)}x_1 + a_{22}^{(0)}x_2 & \cdots & a_{2n}^{(0)}x_n & = b_2 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1}^{(0)}x_1 + a_{n2}^{(0)}x_2 & \cdots & a_{nn}^{(0)}x_n & = b_n \end{cases}$$

Para realizar a eliminação de Gauss em cada passo k é escolhido um elemento pivô a_{ij} (sendo $i = j$) que será a base de cálculo para eliminar os elementos da coluna abaixo dele e obter a matriz |U|. Para realizar essa eliminação são utilizados fatores $l_{i1} = \frac{a_{i1}^{(0)}}{a_{11}^{(0)}}, \forall i = 2,...,n$. Estes fatores são definidos de modo que para um dado coeficiente da matriz $a_{11}^{(0)} \neq 0$, todos os coeficientes da linha 1 são multiplicados pelo fator e subtraídos dos coeficientes das colunas abaixo de $a_{11}^{(0)}$, conforme o sistema abaixo

$$\begin{cases} a_{ij}^{(1)} = a_{ij}^{(0)} - l_{i1}a_{1j}^{(0)}, \\ b_i^{(1)} = b_i^{(0)} - l_{i,1}b_1^{(0)} \end{cases}$$

para $i = 2,...n$ e $j = 1,...,n$. Desta forma, zera-se a coluna abaixo de $a_{11}^{(0)}$ conforme

$$\left[\begin{array}{ccc|c} a_{11}^{(0)} + a_{12}^{(0)} + a_{13}^{(0)} & \cdots & a_{1n}^{(0)} & b_1^{(0)} \\ 0 + a_{22}^{(1)} + a_{23}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ 0 + a_{32}^{(1)} + a_{33}^{(1)} & \cdots & a_{3n}^{(1)} & b_3^{(1)} \\ \vdots & \ddots & \vdots & \vdots \\ 0 + a_{n2}^{(1)} + a_{n3}^{(1)} & \cdots & a_{nn}^{(1)} & b_n^{(1)} \end{array}\right]$$

Nos passos seguintes, o mesmo procedimento acontece para os coeficientes da diagonal principal até chegar ao passo $k = n-1$ e que se obtenha:

$$\left[\begin{array}{ccc|c} a_{11}^{(0)} + a_{12}^{(0)} + a_{12}^{(0)} & \cdots & a_{1n}^{(0)} & b_1^{(0)} \\ 0 + a_{22}^{(1)} + a_{23}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ 0 + 0 + a_{33}^{(2)} & \cdots & a_{3n}^{(2)} & b_3^{(2)} \\ \vdots & \ddots & \vdots & \vdots \\ 0 + 0 + 0 & \cdots & a_{nn}^{(n-1)} & b_n^{(n-1)} \end{array}\right]$$

que permite a resolução direta do sistema para calcular o vetor x. Para $u_{ij} = a_{i,j}^{(n-1)}$, tem-se:

$$\left\{ x_n = \frac{b_n^{(n-1)}}{u_{nn}}; \forall i = n-1, n-2, ..., 1, x_i = \frac{(b_i^{(n-1)}) - \sum_{j=i+1}^{n} u_{ij}x_j}{u_{ii}} \right.$$

Dado o produto de duas matrizes, A=LU, sendo L= (l_{ij}) e U= (u_{ij}) para $i, j = 1,...,n$ definidas a partir dos coeficientes calculados acima e considerando que $l_{ij} = 0 \forall i < j, l_{ii} = 1$ e $u_{ij} = 0 \forall i > j$, então a matriz A se fatora por A=LU. A resolução do sistema A$x = b$, LU$x = b$. Isto é equivale a:

1. determinar y tal que L$y = b$;
2. determinar x (solução de A$x = b$) tal que U$x = y$

Este método de eliminação de Gauss simples é eficiente na maioria dos casos, porém ele não leva em consideração a ordem em que as equações são tratadas afetando assim a precisão do algoritmo de eliminação no computador. Por exemplo, dado o sistema abaixo:

$$\begin{cases} 0x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases}$$

Esse sistema gera resultados para $a_{11} \cong 0$ que não são confiáveis. Para testar essa afirmação, considera-se:

$$\begin{cases} \Phi x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases}$$

Sendo Φ um número pequeno diferente de zero. Resolvendo o sistema eliminando o x_1 da segunda equação (soma da segunda equação com a primeira multiplicada pelo fator $-\frac{1}{\Phi}$):

$$\begin{cases} kx_1 + x_2 = 1 \\ (1-\frac{1}{\Phi})x_2 = 2 - \frac{1}{\Phi} \end{cases}$$

Logo, $x_2 = \frac{2-\frac{1}{\Phi}}{1-\frac{1}{\Phi}}$ e $x_1 = \frac{1-x_2}{\Phi}$. Como $\frac{1}{\Phi}$ é grande, em cálculos com mantissa finita tanto $2-\frac{1}{\Phi}$ como $1-\frac{1}{\Phi}$ seriam computados como $(\frac{-1}{\Phi})$, o que geraria um $x_2 = 1$ e um $x_1 = 0$.

Para melhorar a precisão do algoritmo de eliminação de Gauss, as equações deveriam ser primeiro permutadas. Assim, após a etapa de eliminação, chega-se à:

$$\begin{cases} x_1 + x_2 = 2 \\ (1-k)x_2 = 1 - 2k \end{cases}$$

onde, $x_2 = \frac{1-2k}{1-\Phi} \cong 1$ e $x_1 = 2 - x_2 \cong 1$.

A técnica de permutação de linhas rearranja as equações de forma a colocar os maiores coeficientes na diagonal principal a cada passo. Este rearranjo é denominado pivoteamento parcial [1].

Dado um sistema $Ax = b$ ou $LUx = b$, a fatoração com pivoteamento leva a PA e não a A, onde P é a matriz permutação, no momento de utilizar L e U para resolver $Ax = b$ deve-se aplicar o procedimento sobre Pb e não sobre b: PAx =LUx =Pb. Após a fatoração com pivoteamento da matriz a obtenção do resultado do sistema é atingido através da retrossubstituição dos valores nas equações.

Este procedimento é adequado para a resolução de sistemas lineares com matrizes de tamanho limitado pela capacidade de processamento seqüencial da máquina utilizada. Para matrizes com ordem de grandeza superiores, torna-se necessária a paralelização do algoritmo LU. O HPL (Highly Parallel Linpack) é um *benchmark* que utiliza a versão paralela desse algoritmo.

6.2 Highly Parallel Linpack

O *benchmark* Linpack (DONGARRA; LUSZCZEK; PETITET, 2001) é um *benchmark* [2] composto por um pacote de funções utilizado para a solução de sistemas densos de equações lineares. É um dos *benchmarks* mais utilizados em processamento de alto desempenho.

Existem duas versões de *benchmarks* do Linpack anteriores ao HPL que podem ser usadas para avaliar o desempenho de computadores com sistemas densos de equações lineares segundo a capacidade das memórias: o primeiro, Linpack-100 desenvolvido em 1979, para matrizes 100×100; o segundo, Linpack-1000 desenvolvido nos anos 80, para matrizes 1000×1000.

Com o surgimento dos computadores com memória distribuída e seu potencial de solucionar problemas numéricos cada vez maiores, foi preciso pesquisar melhores soluções em *benchmarking*. Para isso, o *benchmark* deve ser escalável, para ter a capacidade de

[1] parcial: há apenas troca de linhas

[2] programa de teste de desempenho que analisa as características de processamento e de movimentação de dados de um sistema de computação com o objetivo de prever seu desempenho e revelar os pontos fracos e fortes de sua arquitetura (DEROSE; NAVAUX, 2002)

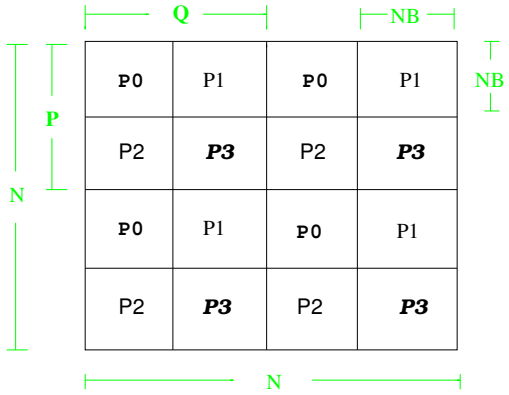

Figura 6.1: Representação dos parâmetros N,N_B,P×Q para 4 processos (P0-P3).

aproveitar as características de arquitetura de cada modelo de máquina na solução dos problemas tendo a mesma eficiência. Um novo programa baseado no Linpack que abrange estes problemas foi disponibilizado em 1993, chamado HPL (Highly Parallel Linpack).

Como o Linpack, o HPL também resolve sistemas lineares densos de equações, porém neste caso, com memória distribuída. O pacote utiliza aritmética de ponto flutuante de 64-bits e rotinas portáveis para operações de álgebra linear e passagem de mensagens.

Também é possível a seleção de um dos algoritmos de fatoração para poder estimar o tempo e a precisão da solução. Ele requer implementação em MPI, BLAS ou VSIPL (*Vector Signal Image Processing Library*).

6.2.1 Algoritmo

O HPL resolve sistemas densos de equações lineares. Essa resolução é baseada na fatoração LU com pivoteamento parcial (PA=LU) visto na seção 6.1.

Para o cálculo do LU em paralelo, é preciso definir a ordem de grandeza da matriz N×N de acordo com o tamanho total de memória das máquinas onde serão disparados os processos. Essa matriz será dividida em blocos menores de dimensão $N_B \times N_B$ que serão calculados por uma grade de duas dimensões P×Q de processos para garantir um bom balanceamento de carga assim como uma boa escalabilidade do algoritmo. Os parâmetros N, N_B e P×Q estão representados na Figura 6.1.

A fatoração do LU disponibilizada no HPL segue o procedimento mostrado nas etapas abaixo para um exemplo de contexto com 9 processos. A Tabela 6.1 mostra a distribuição dos processos. E a Figura 6.2 ilustra como acontece a comunicação entre eles.

P0	P1	P2
P3	P4	P5
P6	P7	P8

Tabela 6.1: Grade de Processos

- Fatoração dos painéis da matriz representado na Figura 6.3 pela parte escura onde estão L e U. Pela fatoração do bloco A_{00}, Figura 6.4 obtém-se diretamente L_{00} e U_{00}. Depois, L_{00} é enviado para os processos P1 e P2. E U_{00} é enviado para os processos P3 e P6.

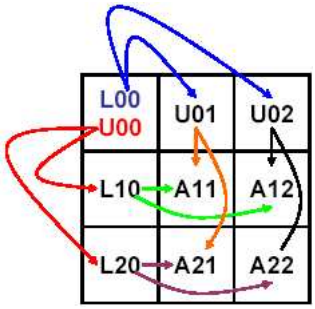

Figura 6.2: Comunicação entre os processos.

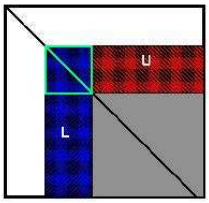

Figura 6.3: Bloco LU.

- Etapa de atualização que consiste na resolução do sistema linear pelos processos P1 e P2 que calculam U_{01} e U_{02}, respectivamente. E também pela resolução do sistema por P3 e P6, os quais encontram L_{10} e L_{20}, respectivamente.

- Uma etapa de atualização na qual os processos P4, P5 recebem de P3, L_{10},enquanto P7 recebe de P6, L_{20}. E os processos P4 e P7 recebem de P2, U_{01}, enquanto P5 recebe de P3, U_{02}.

- Depois o processo P4 fatora a matriz como P0 realizou e o procedimento se repete até que o último bloco seja fatorado.

Terminada a fatoração do LU, a distribuição final de L e U, do exemplo anterior está ilustrada na Figura Porém ainda resta a resolução do sistema por retrossubstituição. Para resolver $Ax = b$ por $y = \frac{b}{L}$ e $x = \frac{y}{U}$ conforme a Figura 6.5, os processos resolvem o sistema seguindo os passos abaixo:

- P0 - resolve $L_{00} * y_0 = b_0$ e envia y_0 para P3 e P6;

- P3 - envia $L_{10} * y_0$ para P4;

- P4 - resolve $L_{11}y_1 = b_1 - L_{10} * y_0$ e envia y_1 para P7;

- P7 - envia $L_{21} * y_1$ para P8;

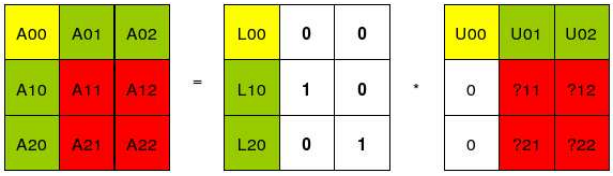

Figura 6.4: Fatoração LU em blocos.

U00 L00	U01	U02
L10	U11 L11	U12
L20	L21	U22 L22

Figura 6.5: Bloco de distribuição de L e U.

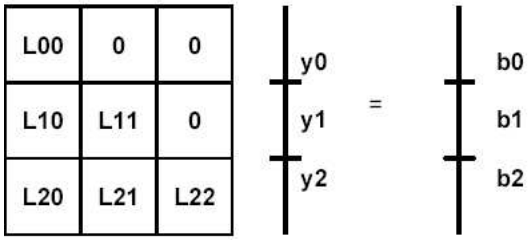

Figura 6.6: Solução L$y = b$.

- P8 - resolve $L_{22} * y_2 = b_2 - L_{20} * y_0 - L_{21} * y_1$.

O resultado é: y_0 em P0, y_1 em P4 e y_2 em P8.

6.2.2 Características de Implementação do Algoritmo

A cada iteração um painel de colunas N_B é fatorado. Existem três algoritmos de fatoração recursiva baseado na multiplicação matriz-matriz (*Crout, left- and right-looking*) que podem ser escolhidos para realizar esta tarefa. O programa também permite ao usuário escolher em quantos sub-painéis o painel atual deve ser dividido em cada nível de recursão. Além disso, é possível selecionar o critério de parada da recursão em termos do número de colunas para serem fatoradas. Quando este limite é alcançado, o sub-painel será fatorado utilizando um dos três algoritmos de fatoração baseados em operações matriz-vetor.

Finalmente, para cada coluna do painel, a procura do pivô e as operações de troca e difusão da linha pivô são combinadas em um único passo de comunicação. Após a fatoração do painel ter sido terminada, o painel de colunas fatoradas é distribuído para outras colunas de processos. Existem muitos tipos de algoritmos de *broadcast* e o programa oferece as seguintes opções que estão melhores detalhadas em (PETITET et al., 2004):

- anel em ordem crescente;
- anel em ordem crescente modificado;
- 2 anéis em ordem crescente;
- 2 anéis em ordem crescente modificado;
- redução de largura de banda;
- Redução de largura de banda modificada.

Com exceção do modo anel em ordem crescente, os demais modos de *broadcast* seguem um algoritmo de 2 fases como o apresentado por (BISSELING, ????). As opções

modificadas ajudam o próximo processo (aquele participaria da fatoração do painel antes do atual) na carga de mensagens a serem enviadas (por outro lado ele tem que receber tão bem quanto enviar a matriz de dados atualizada). As opções de anéis propagam os dados atualizados em um único modo *pipeline*, enquanto que as opções *two-ring* propagam dados em dois *pipelines* concorrentes. A redução de largura de banda divide uma mensagem em pedaços para ser enviada e distribui-los através de uma única linha da grade de processos assim que mais mensagens são trocadas, mas o volume total de comunicação é independente do número de processos.

6.3 Uso do HPL

Para poder testar o protótipo da biblioteca β-MPI, foi escolhido o benchmark Linpack (HPL). O objetivo desses testes iniciais foi medir o atraso causado na execução do benchmark pela intrusão da β-MPI e pelo DFG fornecido pela β-MPI, analisar padrão de comunicação da fatoração LU. As execuções com o HPL foram realizadas no cluster labtec-UFRGS. O sistema operacional usado foi Linux Debian com kernel 2.6 e foram utilizados 16 nodos do cluster.

O HPL faz uso extenso de operações matriciais. Para realizar tais operações, o HPL utiliza a Blas (*Basic Linear Algebra Subroutine*) (DONGARRA et al., 1990), a qual é implementada pela biblioteca Atlas (WHALEY; DONGARRA, 1999). Essa biblioteca foi concebida com o objetivo de fornecer uma implementação eficiente e portável da Blas.

Para extrair o máximo de desempenho da arquitetura do cluster labtec, a instalação da Atlas foi otimizada. Para essa otimização foram realizados testes em um nodo do cluster com o intuito de obter o poder de processamento sequencial. Estes testes foram feitos através do aplicativo `dgemm`.

Em um primeiro teste, foi instalada a biblioteca Atlas e executado o aplicativo `dgemm`, obtendo-se um pico de desempenho de 800MFlops, conforme mostra a Figura 6.7. Para um nodo bi-processado com 1,1GHz de freqüência de clock para cada processador, este resultado é considerado ruim, pois o aproveitamento não chega à metade da capacidade de processamento do nodo. Segundo (RICHARD et al., 2001), um resultado considerado bom é obter em torno de 80% de pico de desempenho da capacidade total de processamento. No caso dos nodos do cluster labtec isto equivale a 1,6GFlops de desempenho.

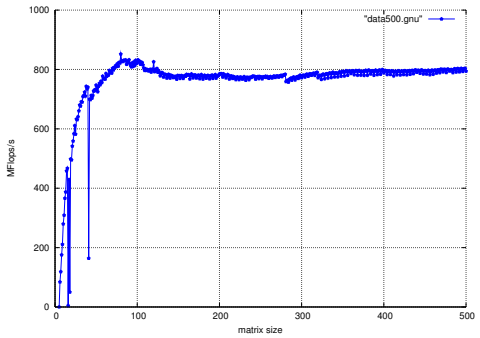

Figura 6.7: Desempenho da Atlas com o `dgemm` em um nodo Dual Pentium III.

Outro teste foi instalar a Blas com suporte a threads e o aplicativo `dgemm` foi usado novamente. O pico de desempenho alcançado foi de 1,68GFlops, como mostra a Figura 6.8. Este desempenho foi atingido porque com o supote a threads os dois processadores do nodo compartilham a memória e realizam os cálculos. O comportamento da curva

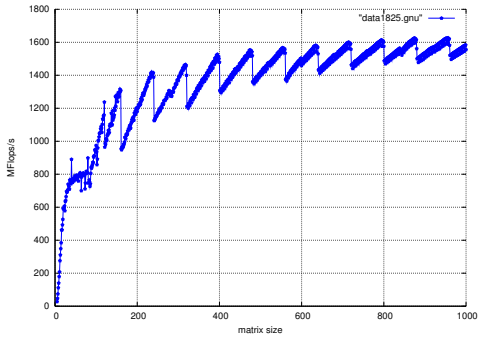

Figura 6.8: Desempenho da Blas com suporte a threads em nodo Dual Pentium III.

em formato de "dentes de serra"na Figura 6.8 é devido ao preenchimento das linhas da memória cache.

A instalação otimizada da Blas mostrou a real capacidade de processamento de um nodo do cluster. A utilização de suporte a threads vai levar a uma execução do *benchmark* mais eficiente.

O HPL tem um arquivo de configuração composto por uma série de parâmetros, localizado em ˜/hpl/bin/Linux_PIV_ATLAS/HPL.dat. Os parâmetros devem ser ajustados de acordo com a arquitetura onde o *benchmark* será executado. Os principais deles são:

- número de fatorações realizadas;
- N - tamanho da matriz que será calculada;
- número de diferentes tamanho de blocos;
- N_B - tamanho do bloco;
- número de grades de processos P $\times$Q;
- grade P $\times$ Q;
- algoritmos de fatoração;
- algoritmos de broadcast.

O arquivo HPL.dat foi configurado com o objetivo de extrair o máximo de desempenho do cluster labtec utilizando 16 dos seus nodos. Os parâmetros adotados foram grade de processos 4×4,bloco de tamanho $N_B = 100$, algoritmo de fatoração Crout e N=41000 [3].

6.4 Análise da Geração do DFG do HPL através da β-MPI

Os objetivos dessa análise são três. O primeiro é validar a β-MPI e mostrar como ela possibilita a escolha do melhor algoritmo de *broadcast* no linpack. O segundo objetivo é identificar quanto de atraso, na geração do DFG pelo β-MPI, será gerado na execução do *benchmark*. E o terceiro é decidir por um dos 6 tipos de algoritmos de *broadcasts* que o HPL fornece baseado no DFG gerado pela β-MPI.

[3]Este valor de N foi encontrado baseando-se na capacidade total de memória disponível do cluster, ou seja, $16Gb = P \times Q \times N^2 \times sizeof(double)$

Em termos de atraso causado pela geração do DFG, pode-se notar através das Figuras 6.9 e 6.10 que a intrusão da β-MPI no desempenho do HPL é mínima, chegando no máximo a 10MFlops. Este atraso chega a 0,02% comparado à execução do HPL sem a β-MPI, o que já era previsto pois a β-MPI apenas atualiza uma estrutura de dados com as informaçoes necessárias para a montagem do grafo da aplicação.

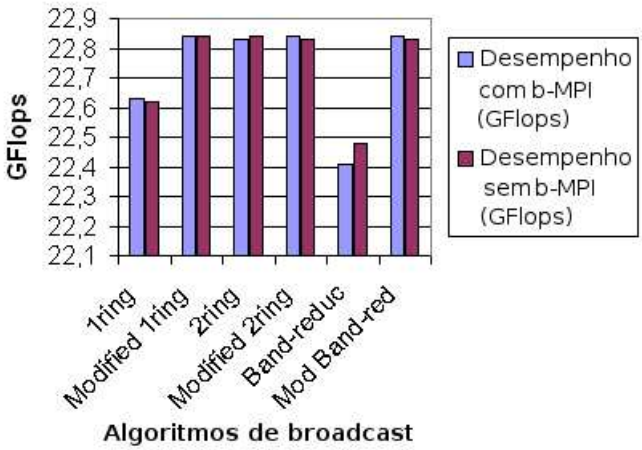

Figura 6.9: Desempenho do Linpack (GFlops) através dos 6 algoritmos de broadcast, com e sem β-MPI

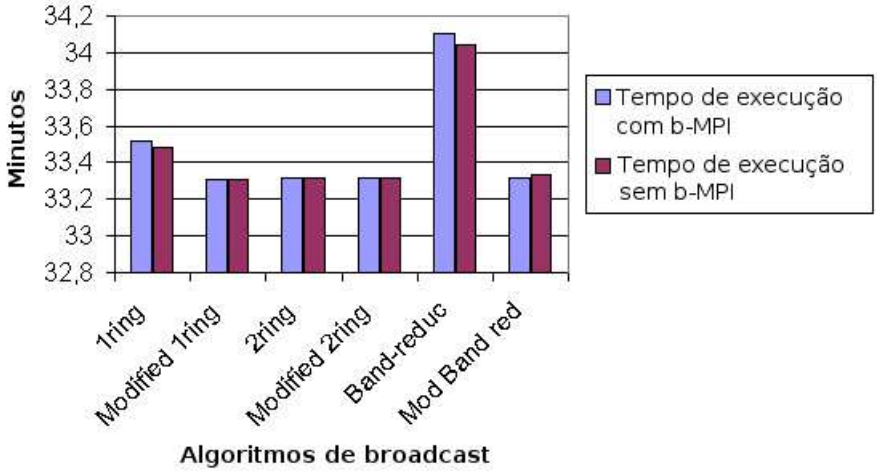

Figura 6.10: Tempo de execução com os 6 algoritmos de broadcast, com e sem β-MPI

Os 6 tipos de algoritmos de *broadcast* que o HPL disponibiliza para escolha pelo arquivo de configuração são: anel em ordem crescente (1-anel), anel em ordem crescente modificado (1-anelM), anel duplo em ordem crescente (2-anel), anel duplo em ordem crescente modificado (2-anelM), redução de largura de banda (BW-Redução) e redução de largura de banda modificada (BW-RedúçãoM).

Nos algoritmos em anel, a propagação da atualização dos dados ocorre em modo pipeline sendo que no 1-anel a passagem de informações acontece sequencialmente - processo 0 envia para o processo 1, o qual envia para o processo 2 e assim sucessivamente. Entretanto, no algoritmo 2-anel são gerados 2 pipelines concorrentes: os P processos são divididos em 2 partes: de 0 a $\frac{P}{2}-1$ e de $\frac{P}{2}$ até N. Os pipelines começam dos processos 1 e $\frac{P}{2}$ respectivamente. Nos algoritmos modificados, a diferença é que o processo 0 envia

duas mensagens: a primeira para o processo 1 e a segunda para o processo 2, a partir do qual a propagação segue o algoritmo padrão no caso do 1-anel; e no caso do 2-anelM, o processo 0 envia 3 mensagens, uma para o processo 1, uma para o processo $\frac{P}{2}$ e outra para o processo 2 e a partir daí o algoritmo continua o esquema original.

No algoritmo BW-Redução a mensagem é dividida pelo número de processos e cada pedaço dela é entregue a um processo. A versão modificada deste algoritmo tem como característica o fato que o processo 0 envia os pedaços da mensagem para o processo 1 e depois disso o algoritmo segue a versão original.

Cada um dos algoritmos de broadcast do HPL apresentados teve o seu respectivo DFG gerado pela β-MPI. Os grafos resultantes da geração são apresentados nas Figuras de 6.11 à 6.15. Nessas Figuras observa-se a presença de 3 tipos de linhas, as quais indicam o volume de dados trafegado entre os processos. Os tipos de linhas são as pontilhadas, que representam um volume de dados leve, as linhas normais, que indicam um volume de mensagens moderado e as linhas em negrito, as quais representam um volume de dados intenso. A quantidade de dados trafegados entre os processos usada para definir a qual tipo de linha pertence um dado *link* de comunicação estão na Tabela 6.2.

Tabela 6.2: Volume de dados trafegado para cada algoritmo de *broadcast*.

Algoritmo	negrito	normal	pontilhada
1-anel	1 Mb	0.5 Gb	1.3 Gb
1-anelM	0.5 Gb	0.8 Gb	1.4 Gb
2-anel	0.5 Gb	0.8 Gb	1.4 Gb
2-anelM	0.5 Gb	0.8 Gb	1.4 Gb
BW-Redução	0.4 Gb	———	1.4 Gb
BW-ReduçãoM	0.5 Gb	0.8 Gb	1.1 Gb

Um ponto em comum a todos DFGs que a β-MPI gerou é o traçado de rotas de linhas e colunas de comunicação em uma grade de processos virtual. Nenhum *link* de comunicação com rotas diferentes destas foi traçado, pois o algoritmo tem como característica a comunicação apenas entre linhas e coluna de processos, conforme descritona seção 6.2.1.

Os primeiros DFGs apresentados nas Figuras 6.11 e 6.12 utilizam o mesmo 2-anelM, porém para duas grades de processos distintas, no primeiro caso, 2×8 e no segundo caso, 4×4. Estes dois DFGs possuem, respectivamente, 16 arestas com tráfego intenso, 8 arestas com tráfego moderado e 24 arestas com tráfego leve e 16 arestas com tráfego intenso, 24 arestas com tráfego moderado e 8 arestas com tráfego leve. É importante notar que apesar de aparecerem *links* em diagonal no caso da grade 2×8, estes são devido aos *broadcasts* ao longo de linhas de processos, dado que uma linha, neste caso, é composta por 8 processos (P1 à P8 e P9 à P16). Comparando-se os 2 DFGs, é nítida a intensidade maior de tráfego de mensagens no caso da grade 2×8, que se explica devido ao fato dessa topologia possuir uma cadeia de processos muito longa para cada linha, a qual para fazer o *broadcast* de dados, e para rotear o conseqüente acúmulo de blocos, gera a sobrecarga de dados medida.

O DFG do algoritmo BW-Redução, Figura 6.13, apresenta como *links* de comunicação mais carregados aqueles no qual o *broadcast* foi ao longo das colunas, precisamente as arestas P1-P13, P2-P14, P3-P15 e P4-P16, todas elas com volume de tráfego de mensagens por volta de 1.4Gb. Isto também mostra mais tráfego ao longo de algumas rotas

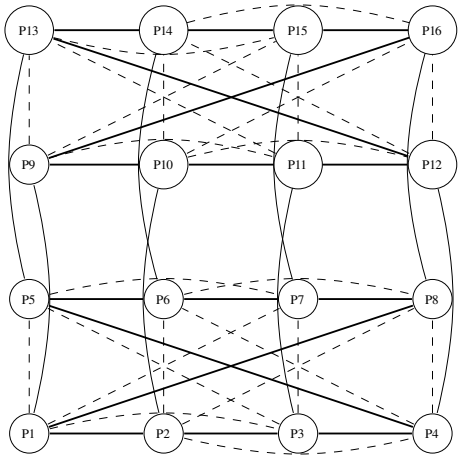

Figura 6.11: DFG para o *broadcast* em 2-anelM , com topologia 2×8.

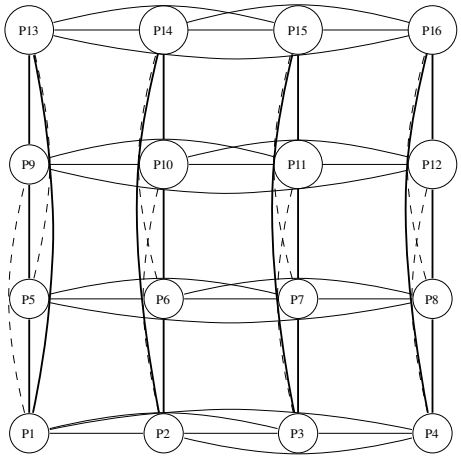

Figura 6.12: DFG o *broadcast* em 2-anelM , com topologia 4×4.

horizontais do que outros *broadcasts*, por exemplo P1-P4 com 1,4Gb versus os 0,8Gb no 2-anel, vide Figura 6.12. O algoritmo BW-redução é o pior dos 6 tipos que o HPL disponibiliza para essa execução. Confirma-se isto também pelos resultados de tempo de execução apresentados na Figura 6.10.

No entanto, a versão modificada do algoritmo BW-Redução, vide Figura 6.14 apresentou melhores resultados do que a sua versão original. As faixas de tráfego para a versão modificada são classificadas em três - 32 arestas para tráfego intenso, 8 arestas para tráfego moderado e 8 arestas para tráfego leve, enquanto que a versão original é classificada em apenas duas - 32 arestas de tráfego intenso e 16 arestas de tráfego leve, vide Tabela 6.2. A topologia utilizada nas duas versões foi a mesma. O que diferencia os dois tipos de algoritmos é a carga de mensagens trafegada na rede: pelos *links* de comunicação mais pesados na versão modificada trafegam 1,1Gb e no caso da versão original trafegam 1,4Gb. Dado que os links de tráfego intenso são os mais numerosos (32 *links*), eles contam consideravelmente no volume total de mensagens trocadas. Isto, portanto, justifica o melhor desempenho nos resultados da versão modificada do algoritmo frente a versão original.

No que diz respeito aos algoritmos baseados em anel, pode-se comparar o DFG do algoritmo 1-anel, vide Figura 6.15 com o grafo do algoritmo de 2-anelM apresentado na

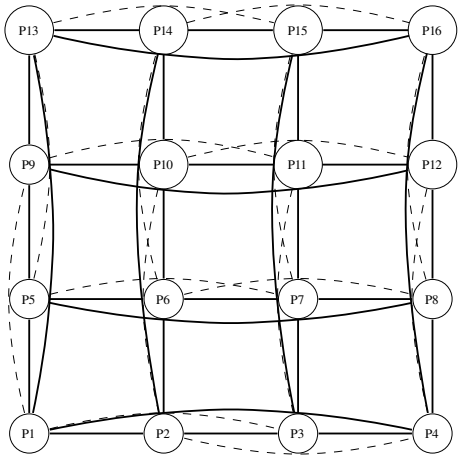

Figura 6.13: DFG do algoritmo BW-Redução com topologia 4×4.

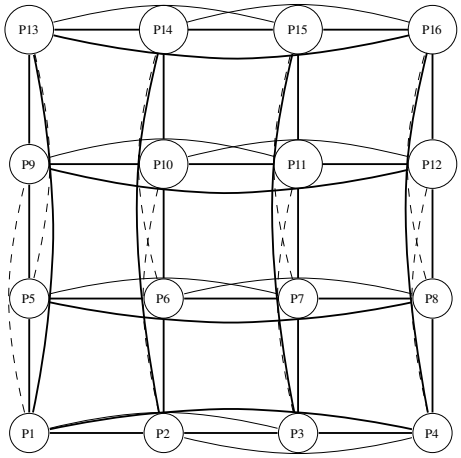

Figura 6.14: DFG do algoritmo BW-RedúçãoM com topologia 4×4.

Figura 6.12 para a mesma topologia 4×4. Os grafos dos algoritmos 1-anelM e 2-anel não foram apresentados porque têm o mesmo DFG da versão 2-anelM. Nota-se que para estes DFGs:

- as 12 arestas verticais (P1-P13, P1-P5, P2-P6, etc...) são da mesma categoria de volume de dados (intenso) para ambos algoritmos, sendo que no caso do 1-anel isto significa mais de 1,3Gb versus 1,4Gb no 2-anelM. Sendo assim, considera-se uma pequena vantagem para o algoritmo 1-anel;

- aparecem 4 arestas no 2-anelM que não estão no DFG do 1-anel (P14-P16, P10-P12, P6-P8 e P2-P4);

- existem mais arestas (P13-P15, P9-P11, P5-P6, P1-P3, P9-P1, P13-P9, P10-P2, etc) com tráfego por volta de 0,5Gb no caso do 2-anel do que no 1-anel, vide Tabela 6.2;

- as 12 arestas horizontais (P13-P14, P14-P15, P13-P16, etc) têm tráfego pesado de 1,3Gb no caso do 1-anel, enquanto no caso do 2-anelM estas arestas possuem tráfego moderado 0,8Gb.

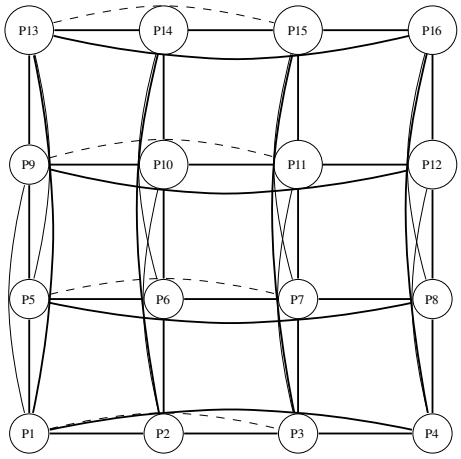

Figura 6.15: DFG do algoritmo de broadcast 1-anel e topologia 4×4.

A análise dos DFGs mostra um tráfego maior no caso do 1-anel, devido às 12 arestas horizontais somarem um atraso de 6Gb ($12 \times 1,3Gb - 12 \times 0,8Gb$) de dados extra para serem trafegados. Mesmo as 12 arestas verticais que somam 1,2Gb ($12 \times 1,4Gb - 12 \times 1,3Gb$) e as 4 arestas a menos que somam 3,2Gb ($4 \times 0,8$) não são suficientes para compensar os 6Gb de tráfego de dados extra. As medidas de desempenho apresentadas nas Figuras 6.9 e 6.10 confirmam esta análise. A diferença entre os algoritmos 1-anel e 2-anel é mínima e condiz com o balanço de vantagens e desvantagens analisado no DFG deste caso.

6.5 Mapeamento no HPL

Nesta etapa, o objetivo é validar mecanismo de remapeamento de tarefas da β-MPI. Este remapeamento de tarefas visa diminuir a comunicação entre os processos disparados no intuito de melhorar o desempenho da aplicação. Para realizar as simulações com o HPL, é preciso definir um número maior de processos do que o número de nodos disponíveis. Essa definição é dada pela escolha da grade de processos P×Q na configuração do HPL.

Haviam 16 nodos do cluster labtec dedicados para as simulações com o HPL e a β-MPI. As execuções foram realizadas uma vez em 4 cenários envolvendo 2, 4, 8 e 16 nodos do cluster em cada um deles. Foram definidas grades de processos distintas e tamanhos de matrizes N compatíveis com o número de processos e nodos alocados. O valor de N foi definido de tal forma que não fosse usado *swap* de memória durante o cálculo da matriz.

Na tabela 6.3 estão os resultados alcançados pelo remapeamento de processos realizado pela β-MPI. No cenário envolvendo dois nodos, foram apresentadas 2 situações. Na primeira, a grade considerada foi a 2×2 com tamanho de matriz dado por N=14000. O volume de dados trafegados entre os processos estão na Tabela 6.4.

Nas duas execuções do HPL, sem e com o remapeamento, a ordem de disparo dos processos foi a mesma. Isso aconteceu, porque sendo a comunicação mais intensa entre os processos 1 e 3 e entre os processo 2 e 4, a ferramenta de particionamento particionou o grafo seguindo o critério de mínimo corte de arestas corretamente. No MPI quando se disparam mais processos do que o número de máquinas disponíveis, a ordem de execução dos processos é uma fila circular, ou seja, processo 0 máquina 0, processo 1 máquina 1, processo 3 máquina 0 e assim por diante.

Tabela 6.3: Remapeamento de tarefas com a β-MPI.

Nodos	N	Grade	tempo s/ β-MPI	tempo c/ β-MPI	GFops s/ β-MPI	GFlops c/ β-MPI
2	14000	2x2	916,37s	914,78s	1,997	2,0
2	12000	2x4	1035,57	1035,27	1,113	1,113
4	16000	2x4	808,72s	775,37s	3,377	3,522
4	16000	4x2	729,56s	729,05s	3,743	3,746
4	12000	4x4	625,5s	632,27s	1,842	1,822
8	20000	2x8	968,07s	904,71s	5,510	5,896
8	14000	8x2	442,68s	438,52s	4,133	4,172
8	10000	4x4	333,58	329,43	1,999	2,024
16	20000	4x8	628,68s	611,2s	8,484	8,727
16	32000	4x4	1199,62s	1200,97s	18,21	18,19

Tabela 6.4: Volume de comunicação entre processos(2×2) - 2 nodos.

processos	volume de dados
1 e 2	398Mb
1 e 3	596Mb
3 e 4	396Mb
2 e 4	593Mb

Na segunda situação, foi considerada uma grade 2×4 e tamanho de matriz N=12000 - não houve ganho com o remapeamento de processos. Nesse caso, o volume de dados trocados entre os processos foi relativamente o mesmo e a ferramenta de particionamento sugeriu a seguinte ordem de disparo: processos 1 e 2 na máquina 0; processos 3 e 4 na máquina 1; processos 5 e 6 na máquina 0 e processos 7 e 8 na máquina 1. Cabe notar que o critério de mínimo corte de arestas também foi bem aplicado adequadamente, pois os processos que possuem muito tráfego foram agrupados. Porém, devido ao tráfego de dados ser muito pequeno entre as arestas que conectam os processos 1 e 3, e 5 e 7, frente às demais 12 arestas, não houve melhora no desempenho.

Tabela 6.5: Volume de comunicação entre processos (2×4) - 4 nodos.

processos	volume de dados	processos	volume de dados
1 e 2	386,3Mb	3 e 7	384,0Mb
1 e 3	192,2Kb	4 e 8	386,8Mb
1 e 4	386,1Mb	5 e 6	390,4Mb
1 e 5	389,9Mb	5 e 7	390,3Mb
2 e 3	388,2Mb	5 e 8	388,2Mb
2 e 6	393,0Mb	6 e 7	390,3Mb
3 e 4	388,4Mb	7 e 8	392,5Mb

O cenário composto por 4 nodos teve 3 situações: nas 2 primeiras, a matriz teve tamanho de N=16000. Para a situação com grade de 2×4 processos, a primeira execução (antes do remapeamento) teve como valor de desempenho alcançado 3,377GFlops e na segunda execução ele foi de 3,522GFlops. Na primeira execução, os processos foram distribuídos em fila circular-padrão MPI. Na segunda execução, o particionamento visando diminuir o corte de arestas teve um efeito mais notável do que nos casos envolvendo dois nodos. O volume de dados trocados entre os processos está na Tabela 6.5. A Figura 6.16 ilustra distribuição dos 8 processos pelos 4 nodos depois do remapeamento. Para este cenário, nota-se que o agrupamento dos processos 1 e 2, dos processos 3 e 4, dos processos 5 e 6 e dos processos 7 e 8 possibilitou o ganho de desempenho na segunda execução do HPL. Visto que eles trocam uma quantidade de informações alta, se eles usarem a rede para o tráfego de dados, o ônus será maior que se os processos 4 e 8 ou os processos 2 e 6 fizerem o mesmo. A diferença total em termos de comunicação foi de 14Mb a menos de dados trafegando pela rede depois do remapeamento dos processos.

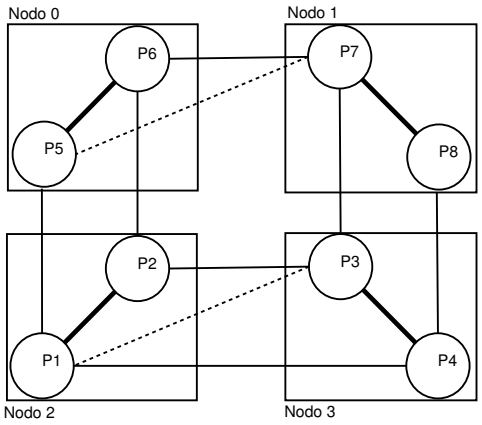

Figura 6.16: Grafo com os processos remapeados - grade 2×4.

Já no caso de uma grade 4×2 usando 4 nodos o efeito do remapeamento fez pouca diferença. A diminuição de dados trafegados pela rede foi apenas em torno de 348Kb de dados.

No terceiro cenário para 4 nodos, o tamanho da matriz foi de N=12000 e a grade de 4×4, o efeito do remapeamento foi nulo dado que o desempenho na primeira execução foi de 1,842GFlops versus 1,822GFlops na segunda execução. Isso aconteceu porque o número total de arestas cortadas foi aproximadamente o mesmo nos dois casos. Nesse caso a ferramenta Metis também sugeriu uma fila circular como ordem de redisparo dos processos. Como o algoritmo de corte de arestas é uma aproximação do verdadeiro custo de comunicação gerado pelo particionamento, podem acontecer algumas diferenças como essa.

Nos cenários 3 e 4 - com 8 e 16 nodos, respectivamente, aconteceram situações semelhantes às apresentadas no cenário 2 (situação 1). Na primeira situação do cenário 3, com grade de processos 2×8 e matriz com N=20000 e na primeira situação com grade de processos 4×8 e matriz com N=20000 obteve-se uma significativa melhora de desempenho devido ao fato de o corte de arestas ter sido grande na primeira execução e na segunda execução ele foi consideravelmente diminuído.

No terceiro cenário, para uma grade 8×2 e tamanhos de matriz N=14000, a melhora de desempenho foi menor porque o número de arestas cortadas não variou muito com o

remapeamento. No caso com uma grade de 4×4 praticamente não houve variação no corte de arestas sendo a diferença de desempenho, portanto, mínima.

Para o quarto cenário houve outras configurações de grade de processos e tamanhos de matriz, como 4×16 com N=18000, 8×4 com N=20000 e 16×4 com N=13000. Nessas configurações o remapeamento piorou o desempenho significativamente devido ao fato que o volume de comunicação começou a ficar muito grande frente ao volume de dados calculados.

6.6 Resumo

Este capítulo apresentou a fatoração LU implementada no *benchmark* HPL, uma análise detalhada sobre seu uso e uma avaliação levando em conta a execução dela com e sem a β-MPI. O objetivo foi mostrar como a β-MPI pode auxiliar os usuários de programas MPI a obter melhor aproveitamento dos recursos disponíveis através de uma distribuição eficiente de processos.

A primeira seção deste capítulo apresentou o algoritmo usado para a resolução de sistemas através da fatoração LU. A segunda seção mostrou no início o *benchmark* HPL e suas versões. Depois foram explicadas a versão paralela do algoritmo de fatoração LU e as características de implementação deste algoritmo no HPL.

Na terceira seção, foi tratada a utilização do HPL, desde a sua instalação até os parâmetros de execução que ele disponibiliza configurar. Em relação a instalação, foi considerada uma comparação e análise do uso das Blas com e sem o uso de threads através do aplicativo `dgemm`.

A quarta seção apresentou a execução do HPL com a β-MPI para verificar como a biblioteca pode ajudar na escolha do melhor algoritmo de *broadcast* que o *benchmark* disponibiliza, e medir o quanto de atraso na aplicação é agregado pelo uso da biblioteca.

Na quinta seção, foram apresentados os cenários de execução do HPL com a β-MPI. Muitas configurações foram feitas e pode-se notar que a eficiência do remapeamento de processos depende significativamente do número de arestas cortadas pelo particionamento e da diferença entre o volume de comunicação. Com a validação da β-MPI através dessa aplicação, percebeu-se que a β-MPI pode ser utilizada para realizar a extração do grafo da aplicação auxiliando na escolha de algoritmos e no mapeamento eficiente de processos. O assunto deste capítulo foi publicado no *17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)* (SILVA et al., 2005).

7 CONCLUSÕES

Este trabalho contemplou uma proposta para o escalonamento de programas MPI através da disponibilização de uma biblioteca - β-MPI. O escalonamento considerado foi estático, levando em conta apenas o remapeamento de processos em máquinas conforme o volume de mensagens trocadas entre eles em execução anterior.

Nos capítulos tratados, primeiramente foi feito um levantamento do estado da arte, o qual buscou identificar ferramentas ou bibliotecas com finalidades semelhantes à proposta. A ferramenta que mais se assemelha a β-MPI é a CASCH, por ela realizar um escalonamento estático, utilizar troca de mensagens e utilizar DFGs. Porém a abordagem que ela fornece é o uso de algoritmos para diferentes arquiteturas - UNC e BNP dedicados a redes totalmente conectadas, e o algoritmo APN, o qual considera a estratégia de roteamento da rede. Além disso, CASCH implementa rotinas próprias para troca de mensagens. Já no caso da β-MPI, o uso é definido para programas que usam primitivas de troca de mensagens MPI e é feito apenas o mapeamento das tarefas, sendo este baseado no grafo do programa e através do uso de uma ferramenta de particionamento.

A parte que tratou da validação de β-MPI teve 3 aplicações com características distintas, apesar de todas elas manterem um padrão de comunicação constante entre execuções sucessivas. A primeira aplicação tratada foi a FFT e o uso da β-MPI serviu apenas para extrair grafo do programa. Sendo assim, foi procurada uma outra aplicação: a transferência de calor, a qual devido à forma como a sua solução foi implementada apresentou um grafo irregular. Por essa característica foi possível ver que a β-MPI pode trazer um benefício otimizando a distribuição dos processos com o auxílio de técnicas de particionamento de grafos as quais têm como objetivo a minimização do corte de arestas.

No capítulo do HPL foi mostrado o uso da β-MPI na extração do grafo da aplicação e no auxilio a escolher o melhor algoritmo de *broadcast* para configuração de máquina utilizada. Para essa escolha foi considerado o algoritmo com menor número de mensagens sendo trafegadas na rede, o tempo de execução e o desempenho em GFlops. Depois disso foi feito o remapeamento de processos. Pode-se ver que a forma como os processos são distribuídos pelas máquinas afeta o desempenho da aplicação.

Desta forma validou-se a proposta em situações distintas e foi identificado como ela pode auxiliar um usuário que precise ter um programa MPI otimizado. O usuário não vai precisar modificar o código da sua aplicação, apenas incluir o header da biblioteca nos fontes e linkar com a biblioteca na compilação. Depois, o usuário roda o *script* `betarun` para que a aplicação seja executada 2 vezes, sendo que a primeira é para a extração do grafo e a segunda para que a distribuição dos processos seja mais eficiente.

Neste trabalho foi usada uma abordagem de otimização de aplicações MPI através da distribuição dos processos visando minimizar a comunicação via rede. Esta abordagem utilizando a representação por grafos e a ferramenta de particionamento Metis provou ser

eficiente pelos resultados alcançados com o remapeamento de processos. Além disso, esta proposta da β-MPI, também auxilia na escolha de algoritmos de comunicação. O usuário poderá partir de uma configuração otimizada de execução do programa MPI.

Os algoritmos utilizados nas ferramentas que estão nos trabalhos relacionados não são adequados para esta proposta. O motivo é que os algoritmos são aplicáveis em grafos do tipo DAG, enquanto que na β-MPI se utiliza o grafo de comunicação entre as tarefas.

Por outro lado, a proposta tem um escopo específico de aplicações que podem se beneficiar do tipo de escalonamento adotado por ele ser estático - são aplicações que precisam manter um mesmo padrão de comunicação quando executadas mais de uma vez. Uma idéia é utilizar o suporte a migração de processos da distribuição LAM-MPI, o qual permite que processos sejam transferidos de uma máquina para outra durante a execução de um programa, para viabilizar uma versão da β-MPI que realize escalonamento dinâmico. Esse tipo de mecanismo permite que sejam disponibilizadas funções como o balanceamento dinâmico de comunicação. Dessa forma, aumentaria a gama de aplicações que poderiam usar a biblioteca de escalonamento β-MPI.

7.1 Contribuições do Trabalho

A principal contribuição deste trabalho foi a disponibilização de uma biblioteca de escalonamento de tarefas para programas MPI. Com o protótipo da β-MPI vieram as seguintes contribuições:

- ferramenta que permite contabilizar o volume de troca de dados feito entre os processos;
- biblioteca de fácil integração com aplicações MPI;
- auxílio da β-MPI na escolha de algoritmos - como o de *broadcast* do HPL;
- a maioria das primitivas de comunicação da norma 1.2 do MPI já foram redefinidas na β-MPI;
- otimização de instalação e uso do HPL;
- validação da β-MPI através de 3 aplicações distintas;
- publicação em Congressos;

Os estudos realizados no desenvolvimento desse trabalho resultaram na publicação de um trabalho no *17th International Symposium on Computer Architecture and High Performance Computing* SBAC-PAD2005 (SILVA et al., 2005). Nessa publicação foi tratada a geração automática do grafo de uma aplicação - HPL, e o uso do grafo para auxiliar na escolha de um algoritmo de *broadcast*. Além dessa publicação foi aceita outra no Congresso PARA - 2006 (*WORKSHOP ON STATE-OF-THE-ART IN SCIENTIFIC AND PARALLEL COMPUTING*) que será em Junho de 2006 e o título do artigo é *Validation of a Mapping library based on Data Flow Graph Generation of MPI Programs*, onde é discutido o uso da β-MPI com as aplicações apresentadas nesta dissertação e o remapeamento produzido.

Houve também publicações regionais e internas do grupo. As regionais foram nos ERADs 2004 e 2005 e as internas do grupo foram no WSPPD 2004 e 2005.

7.2 Trabalhos Futuros

Neste trabalho foi desenvolvida uma proposta para o escalonamento estático de programas MPI através da extração do DFG de uma aplicação. Porém, a área que engloba o escalonamento de programas é muito vasta e existe uma série de pontos que não puderam ser contemplados durante o desenvolvimento desta dissertação. Existem algumas sugestões de trabalhos futuros, bem como atividades que possam vir a agregar os estudos realizados no decorrer deste trabalho.

Uma sugestão é executar aplicações integradas com a biblioteca de escalonamento estático em clusters heterogêneos. Em uma rede homogênea apenas consegue-se benefício com a distribuição dos processos pela comunicação entre eles. Já em uma rede heterogênea se teria mais uma variante que é o poder de processamento de cada máquina. Portanto, algumas máquinas receberiam maior ou menor quantidade de processos conforme a sua capacidade.

Para poder abranger aplicações reais seria necessário disponibilizar uma versão da β-MPI com suporte a funcionalidades de escalonamento dinâmico. Assim, aplicações com variações de dados durante a execução também poderiam ser beneficiadas. Uma aplicação que se poderia usar para validar um mecanismo de escalonamento dinâmico da β-MPI é a simulação do Modelo HIDRA (DORNELES, 2003) desenvolvido no GPPD.

Uma outra questão que pode ser tratada é utilizar outros algoritmos de escalonamento de grafos, tais como os utilizados na CASCH - APN, BPN e UNC. Isso implica em mudar a modelagem do grafo pois o grafo tratado seria orientado.

REFERÊNCIAS

AHMAD, I.; KWOK, Y.-K.; WU, M.-Y.; SHU, W. CASCH: a tool for computer-aided scheduling. **IEEE Concurrency**, Piscataway, NJ, USA, v.8, n.4, p.21–33, 2000.

ANDREWS, G. (Ed.). **Concurrent Programming - Principles and Practice**. [S.l.]: Addison-Wesley Publishing Company, 1991.

BAILEY, D. H.; BARSZCZ, E.; BARTON, J. T.; BROWNING, D. S.; CARTER, R. L.; DAGUM, D.; FATOOHI, R. A.; FREDERICKSON, P. O.; LASINSKI, T. A.; SCHREIBER, R. S.; SIMON, H. D.; VENKATAKRISHNAN, V.; WEERATUNGA, S. K. The NAS Parallel Benchmarks. **The International Journal of Supercomputer Applications**, [S.l.], v.5, n.3, p.63–73, Fall 1991.

BISSELING, R. H. Basic Techniques for Numerical Linear Algebra on Bulk Synchronous Parallel Computers. In: INTERNATIONAL WORKSHOP ON NUMERICAL ANALYSIS AND APPLICATIONS, 1996, ROUSSE, BULGARIA. **Proceedings...** Berlin: Springer-Verlag: 1997. p.46–57. (Lecture Notes in Computer Science, v.1196).

BRIGHAM, E. **The Fast Fourier Transform and its Applications**. [S.l.]: Prentice-Hall, 1988.

BRIGHAM, E. O. **The Fast Fourier Transform**. [S.l.]: Englewood Cliffs: Prentice-Hall, 1974.

CARVALHO, E. C. A. de. **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.

CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v.14, n.2, p.141–154, 1988.

DEROSE, C. A. F.; NAVAUX, P. O. A. Fundamentos de processamento de alto desempenho. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 2, 2002. **Anais...** [S.l.: s.n.], 2002. p.3–29.

DONGARRA, J. J.; CROZ, J. D.; HAMMARLING, S.; DUFF, I. S. A set of level 3 basic linear algebra subprograms. **ACM Trans. Math. Softw.**, New York, NY, USA, v.16, n.1, p.1–17, 1990.

DONGARRA, J.; LUSZCZEK, P.; PETITET, A. **The LINPACK Benchmark**: past, present, and future. www.cs.utk.edu/ luszczek/articles/hplpaper.pdf. Acesso em: Fev. 2004.

DORNELES, R. V. **Particionamento de Domínio e Balanceamento de Carga no Modelo HIDRA**. 2003. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

FASTEST Fourier Transform in the West. Disponível em: <http://www.fftw.org>, Acesso em: abr. 2004.

GALANTE, G. **Métodos Multigrid Paralelos em Malhas Não Estruturadas Aplicados à Simulação de Problemas de Dinâmica de Fluidos Computacional e Transferência de Calor**. 2006. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

GRAHAM, R. L.; WOODALL, T. S.; SQUYRES, J. M. Open MPI: a flexible high performance MPI. In: ANNUAL INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING AND APPLIED MATHEMATICS, 6., 2005, Poznan, Poland. **Proceedings...** [S.l.: s.n.], 2005.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI**: portable parallel programming with the message passing interface. 2nd ed. Cambridge, MA: MIT Press, 1999.

JAGANNATHAN, R. Dataflow Models. In: ZOMAYA, A.Y.H. (Ed.). **Parallel and Distributed Computing Handbook**. New York: McGraw-Hill, 1996. p.223-238.

KARYPIS, G.; KUMAR, V. **METIS**: unstructured graph partitioning and sparse matrix ordering system. Disponível em: <http://www-users.cs.umn.edu/ karypis/publications/-partitioning.html>. Acesso em: abr. 2004.

LAM/MPI Parallel Computing. Disponível em: <http://www.lam-mpi.org>. Acesso em: maio 2005.

LEISERSON, C. E. **Cilk 5.3.2 Reference Manual**. [S.l.]: MIT Laboratory for Computer Science, 2001. Disponível em: <http://supertech.lcs.mit.edu/cilk>. Acesso em: jun. 2004.

NETTO, P. O. B. **Grafos**: teoria, modelos, algoritmos. [S.l.]: E. Bucher, 2001.

NONATO, L. G. **Representação de Grafos**. Disponível em: <http://www.lcad.icmc.usp.br/ nonato/ED/Grafos/node74.html>. Acesso em: abr. 2004.

PACHECO, P. **Parallel Programming with MPI**. [S.l.]: Morgan Kaufmann, 2001.

PETITET, A.; WHALEY, R.; DONGARRA, J.; CLEARY, A. **HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers**. Disponível em: <http://www.netlib.org/benchmark/hpl/algorithm.html>. Acesso em: março 2004.

PICININ, D. **Paralelizações de Métodos Numéricos em Clusters Empregando as Bibliotecas MPICH, DECK e Pthread**. 2003. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REWINI, H. E. Partitioning and Scheduling. In: ZOMAYA, A.Y.H.(Ed.). **Parallel and Distributed Computing Handbook**. NewYork: McGraw-Hill, 1996. p.239-273.

RICHARD, B.; AUGERAT, P.; MAILLARD, N.; DERR, S.; MARTIN, S.; ROBERT, C. **I-cluster**: reaching top500 performance using maintream hardware. Grenoble: HP Laboratories, 2001. (Technical Report HPL HPL-2001-206 20010831).

ROOSTA, S. H. **Parallel Processing and Parallel Algorithms.** New York: Springer-Verlag, 2000.

SCHLOEGEL, K.; KARYPIS, G.; KUMAR, V. **Graph Partitioning for High Performance Scientific Simulations**. [S.l.]: Morgan Kaufmann, 2000.

SILVA, R. E.; PEZZI, G. P.; MAILLARD, N.; DIVERIO, T. Automatic Data-Flow Graph Generation of MPI Programs. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, SBAC-PAD, 17., 2005, Rio de Janeiro. **Proceedings...** Washington: IEEE, 2005. p.93–100.

SQUYRES, J. M.; LUMSDAINE, A. A Component Architecture for LAM/MPI. In: EUROPEAN PVM/MPI USERS' GROUP MEETING, 10., 2003, Venice, Italy. **Proceedings...** Berlin: Springer-Verlag, 2003. p.379–387.

THE MPI message passing interface standard. Disponível em: <http://www.mcs.anl.gov/mpi/standard.html>. Acesso em: mar. 2004.

WALSHAW, D. C. **Jostle - graph partitioning software**. Disponível em: <http://www.gre.ac.uk/ c.walshaw/jostle/>. Acesso em: mar. 2004.

WHALEY, R. C.; DONGARRA, J. Automatically Tuned Linear Algebra Software. In: SIAM CONFERENCE ON PARALLEL PROCESSING FOR SCIENTIFIC COMPUTING, 9., 1999. **Proceedings...** [S.l.: s.n.], 1999. 1 CD-ROM.

YANG, T.; GERASOULIS, A. PYRROS: static scheduling and code generation for message passing multiprocessors. In: ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 6., 1992, Washington D.C. **Proceedings...** [S.l.: s.n.], 1992. p.428–437.