JEAN MARCOS LAINE

Desenvolvimento de Modelos para Predição de Desempenho de Programas Paralelos MPI

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

São Paulo
2003

JEAN MARCOS LAINE

Desenvolvimento de Modelos para Predição de Desempenho de Programas Paralelos MPI

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

Área de Concentração:
Sistemas Digitais

Orientador:
Prof. Dr. Edson T. Midorikawa

São Paulo
2003

AGRADECIMENTOS

Primeiramente, gostaria de agradecer à Deus pela sabedoria, ajuda, proteção e outras infinitas provas de amor e presença.

Aos meus pais, Jacir e Maria, que sempre incentivaram e apoiaram minhas decisões. Agradeço ao carinho, à educação e aos conselhos que recebi durante toda a minha vida. Aos meus irmãos, Ivan e Alan, pelos momentos alegres de descontração que me ajudaram a renovar as energias e continuar o trabalho. Obrigado a todos vocês pela confiança e compreensão.

À minha namorada, Janaine, que sempre me apoiou e me incentivou a enfrentar os desafios e a superar os obstáculos que surgiram. Agradeço pela ajuda, paciência, companhia e amor doados em todo esse tempo que estamos juntos. Desejo muito sucesso e sorte!

Ao grande amigo Hélio, que inúmeras vezes ajudou e colaborou no desenvolvimento desse trabalho. À sabedoria e amizade provada nos momentos de reflexão e companheirismo.

Ao amigo e orientador, Prof. Dr. Edson T. Midorikawa, pelo conhecimento transmitido, orientação, disposição e permanente incentivo.

Aos amigos José Roberto (desde Jales-SP), Rodrigo (companheiro de república), Fábio, Gisele, Mário, Li, Augusto, Profa. Liria e demais colegas do LASB, pela amizade, companhia e troca de conhecimentos.

À CAPES, pelo apoio financeiro e incentivo dado à pesquisa.

À todos aqueles que de uma forma ou de outra colaboraram para a realização desse sonho.

RESUMO

Existem muitos fatores capazes de influenciar o desempenho de um programa paralelo MPI (*Message Passing Interface*). Dentre esses fatores, podemos citar a quantidade de dados processados, o número de nós envolvidos na solução do problema, as características da rede de interconexão, o tipo de *switch* utilizado, entre outros. Por isso, realizar predições de desempenho sobre programas paralelos que utilizam passagem de mensagem não é uma tarefa trivial. Com o intuito de modelar e predizer o comportamento dos programas citados anteriormente, nosso trabalho foi desenvolvido baseado em uma metodologia de análise e predição de desempenho de programas paralelos MPI. Inicialmente, propomos um modelo gráfico, denominado DP*Graph^{+}, para representar o código das aplicações. Em seguida, desenvolvemos modelos analíticos, utilizando técnicas de ajuste de curvas, para representar o comportamento das estruturas de repetição compostas por primitivas de comunicação e/ou computação local. Além disso, elaboramos modelos para predizer o comportamento de aplicações do tipo mestre/escravo. Durante o desenvolvimento das atividades de análise e predição de desempenho, implementamos algumas funções para automatizar tarefas e facilitar nosso trabalho. Por último, modelamos e estimamos o desempenho de duas versões diferentes de um programa de multiplicação de matrizes, a fim de validar os modelos propostos. Os resultados das predições realizadas sobre os programas de multiplicação de matrizes foram satisfatórios. Na maioria dos casos preditos, os erros ficaram abaixo de 6 %, confirmando a validade e a precisão dos modelos elaborados.

ABSTRACT

There are many factors able to influence the performance of a MPI (Message Passing Interface) parallel program. Within these factors, we may cite: amount of data, number of nodes, characteristics of the network and type of switch, among others. Then, performance prediction isn't a easy task. The work was developed based on a methodology of analysis and performance prediction of MPI parallel programs. First of all, we proposed a graphical model, named DP*Graph^{+}, to represent the code of applications. Next, we developed analytical models applying curve fitting techniques to represent the behavior of repetition structure compounds by comunication primitives and/or local computations. Besides, we elaborated models to predict aplications of type master/slave. For development of performance prediction activities, some functions was developed to automate tasks and make our work easy. Finally, we modeled and predicted the performance of two different programs of matrix multiplication to prove the accuracy of models. The results of predictions on the programs were good. In the majority of predicted cases, the errors were down 6 %. With these results, we proved the accuracy of developed models.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

INTRODUÇÃO

O avanço tecnológico das redes de interconexão tem viabilizado a utilização de *clusters* de PCs [Buyya99] em processamento de alto desempenho. Até pouco tempo atrás, esse processamento só podia ser obtido através de máquinas multiprocessadas. No entanto, com o surgimento de redes de alta largura de banda e com a fabricação de PCs cada vez mais velozes, a utilização dos *clusters* no processamento de alto desempenho tem sido cada vez mais viável. Além disso, nesse tipo de arquitetura conseguimos obter alto desempenho a um custo muito mais baixo, quando comparado às máquinas multiprocessadas.

Apesar da programação sobre ambientes de memória distribuída ser um pouco mais difícil e exigir maior atenção do programador, os *clusters* têm sido escolhidos devido ao baixo custo associado ao bom desempenho obtido. Muitas vezes, a programação sobre esse ambiente utiliza o paradigma de passagem de mensagem. Nesse paradigma, o programador deve especificar, explicitamente, todas as comunicações existentes entre os processos criados, enquanto que em um ambiente de memória compartilhada isso não é necessário. Além disso, em um programa paralelo muitos fatores que contribuem para a complexidade da programação devem ser considerados. Dentre esses fatores, podemos citar os problemas relacionados com a sincronização entre as tarefas de processos distintos e a possibilidade de dependência de dados entre processos. Estas e outras considerações fazem da programação paralela uma tarefa árdua, ainda mais quando estamos interessados em elaborar programas eficientes.

As duas principais "bibliotecas" de passagem de mensagem utilizadas na programação de sistemas de memória distribuída são: PVM (*Parallel Virtual Machine*)

[Oak02] e MPI (*Message Passing Interface*) [Mpi02]. No entanto, pelo fato do MPI ser um padrão para troca de mensagens, vem sendo cada vez mais preferido e utilizado por aqueles que buscam alto desempenho em ambientes de memória distribuída. Hoje em dia, não só aplicações científicas mas também comerciais, que demandam alto poder computacional, têm sido implementadas com MPI.

Em um programa paralelo muitos fatores interagem de forma complexa e contribuem no desempenho da aplicação, aumentando o tempo de execução do programa. Por esses e outros motivos muita pesquisa tem sido desenvolvida na área de análise e predição de desempenho, buscando identificar esses fatores e diminuir cada vez mais o tempo de execução das aplicações que exigem alto poder computacional. Com isso, várias metodologias e modelos têm sido propostos para guiar as atividades de análise e predição de desempenho. Através de uma análise detalhada do código do programa, podemos identificar os fatores que influenciam o desempenho do mesmo, bem como os possíveis gargalos do sistema. Já os modelos de predição nos permitem estimar o valor de algumas variáveis, como o tempo de execução do programa para algumas situações ainda não consideradas. Isso faz das atividades de análise e predição de desempenho uma importante atividade na construção de programas paralelos eficientes.

Dependendo da técnica adotada, podemos aplicá-la em diferentes etapas do ciclo de vida do *software*, melhorando e otimizando o programa desde o início de seu desenvolvimento. Uma técnica muito utilizada em análise e predição de desempenho é a técnica denominada medir-modificar [Crovella94]. No entanto, essa técnica contribui pouco no conhecimento dos fatores que realmente influenciam o desempenho da aplicação. Esse conhecimento pode ser adquirido através de outra técnica, que analisa com maior profundidade os fatores associados ao desempenho da aplicação. A modelagem analítica é uma técnica que permite essa análise mais detalhada. Essa técnica, além de permitir identificar os fatores que influenciam o desempenho dos programas, também pode mostrar como eles se relacionam. O resultado dessa modelagem é apresentado através de modelos matemáticos que representam o comportamento das aplicações. No entanto, dependendo dos fatores considerados os modelos analíticos tendem a se tornar complexos e pouco práticos. Para evitarmos esse problema, devemos reconhecer o nível de detalhe ideal ou desejado para que o modelo seja suficientemente preciso e, ao mesmo tempo, simples e prático.

Baseado em uma metodologia de análise e predição de desempenho de programas

paralelos MPI, proposta em [Li01a], nosso trabalho busca desenvolver modelos de predição capazes de representar o comportamento de estruturas de repetições, em especial aquelas constituídas de primitivas de comunicação MPI, e também modelar aplicações do tipo mestre/escravo, a fim de predizer o seu comportamento. Além disso, propomos um novo modelo gráfico, baseado em um conjunto de símbolos, para ser utilizado na representação do código dos programas paralelos MPI. A partir dos modelos elaborados, modelamos e predizemos o desempenho de duas versões diferentes de um programa de multiplicação de matrizes. Nesse estudo foi possível aplicar os modelos desenvolvidos a fim de verificar a precisão dos mesmos ao modelar uma aplicação real.

1.1 Objetivos

Este trabalho tem como objetivo principal estender a metodologia de análise e predição de desempenho de programas paralelos MPI proposta por Li [Li01a]. Para isso, modelamos algumas estruturas de programas paralelos não consideradas inicialmente em [Li01a] e propomos algumas extensões sobre os modelos analíticos desenvolvidos por ele. Com isso, esperamos que a metodologia inicial seja aperfeiçoada e se torne mais robusta e completa. Para que nosso objetivo fosse alcançado planejamos as seguintes atividades:

- propor um novo modelo gráfico para representar programas paralelos MPI;
- desenvolver modelos de predições para aplicações do tipo mestre/escravo. Através desses modelos é possível realizar predições variando não só o tamanho do problema, mas também o número de escravos envolvidos na solução do mesmo;
- modelar estruturas de repetição compostas por computação local e/ou primitivas de comunicação MPI;
- modelar e predizer o desempenho de aplicações reais para validar nossos modelos e verificar a sua precisão.

Com o cumprimento dos objetivos, a metodologia inicial se torna ainda mais interessante e novos modelos elaborados nos permitirá representar e estudar o comportamento de uma variedade maior de aplicações.

1.2 Motivação

A análise e predição de desempenho está se tornando cada vez mais importante na implementação eficiente de problemas da classe *Grand Challenge*. Esses problemas são de grande complexidade e exigem alto poder computacional para serem resolvidos. Como exemplo, podemos citar os cálculos atômicos feitos em pesquisas nucleares. As atividades envolvidas no processo de análise e predição de desempenho ajudam a melhorar a eficiência dos algoritmos, a descobrir potenciais pontos de paralelismo e a otimizar o desempenho dos sistemas paralelos.

Apesar de encontrarmos na literatura uma grande quantidade de modelos analíticos propostos para predição de desempenho de programas paralelos, a precisão, a eficiência e a praticidade dos mesmos ainda é uma questão aberta [Adve93, Moura99, Grove01].

As atividades de análise e predição de desempenho podem tornar viável a implementação de aplicações complexas que demandam alto poder computacional. Isso é possível, na medida em que o tempo de execução do programa pode ser reduzido ao identificarmos os fatores que influenciam nesse tempo e, conseqüentemente, minimizarmos seus efeitos com a implementação de programas mais eficientes. Para isso, podemos utilizar algoritmos e/ou implementações diferentes a fim de explorar com mais eficiência os pontos de paralelismo existentes nas aplicações. O fato dos *clusters* de PCs estarem sendo cada vez mais utilizados na busca do desempenho computacional desejado, também motiva o desenvolvimento do nosso trabalho.

1.3 Justificativas

Como resolvemos dar continuidade ao trabalho apresentado em [Li01a], ao verificarmos a possibilidade de realizar algumas extensões e aperfeiçoamentos sobre sua metodologia original a fim de torná-la mais completa e abrangente, muitas de nossas decisões foram influenciadas pelo trabalho desenvolvido por ele.

Escolhemos os programas paralelos MPI para serem modelados pois muitos detalhes envolvidos no processamento local e nas comunicações são explicitamente definidos através das primitivas utilizadas. Isso é importante e facilita o trabalho de

desenvolvimento dos modelos de predições, na medida em que alguns parâmetros envolvidos nas comunicações são declarados já nas primitivas. Além disso, modelos de predições desenvolvidos para programas que utilizam passagem de mensagem podem auxiliar a modelagem de programas escritos em qualquer outro paradigma de programação paralela [Grove01]. Ainda segundo Grove, programas paralelos de larga escala, que exigem alto desempenho, geralmente são escritos através de uma programação de passagem de mensagem e rodam sobre máquinas de memória distribuída.

Todas as implementações de passagem de mensagem oferecem um conjunto muito parecido de operações. No entanto, pelo fato do MPI ser um padrão de passagem de mensagem amplamente utilizado e oferecer mais primitivas de comunicação ponto-a-ponto e coletivas do que o PVM [Geist96], escolhemos os programas MPI para desenvolver nosso trabalho.

Para o desenvolvimento dos modelos de predições, escolhemos a modelagem analítica. Essa técnica de modelagem permite identificar os fatores que contribuem no desempenho das aplicações. Com isso, é possível realizar um estudo mais aprofundado desses fatores, favorecendo a construção de modelos mais precisos. No entanto, devemos tomar cuidado com o nível de detalhe e a quantidade de parâmetros considerados durante a modelagem para evitarmos o desenvolvimento de modelos muito complexos e pouco práticos de serem aplicados.

1.4 Metodologia

Como nosso projeto é uma continuidade do trabalho desenvolvido em [Li01a], nossas atividades foram iniciadas com o estudo de sua metodologia. Nesse estudo, nos familiarizamos com os conceitos envolvidos na área de análise e predição de desempenho. Além disso, conhecemos as estratégias adotadas por Li para o desenvolvimento dos modelos de predições, verificamos como proceder com os testes experimentais de avaliação e validação dos modelos elaborados e como realizar predições de desempenho a partir desses modelos. Posteriormente, observamos alguns aspectos que poderiam ser melhorados na metodologia e algumas potenciais extensões a serem desenvolvidas.

Após a definição das atividades que compreenderiam nosso trabalho de pesquisa, prosseguimos nosso estudo com uma pesquisa bibliográfica sobre os seguintes assuntos:

- modelos, paradigmas e ambientes de programação paralela;
- programação paralela com passagem de mensagem (MPI);
- técnicas de modelagem, avaliação e predição de desempenho.

O trabalho continuou com a modelagem das primitivas de comunicação ponto-a-ponto existentes no padrão MPI. Realizamos testes com os modelos elaborados e submetemos nossas idéias para avaliação nos artigos [Laine02] e [**?**]. Em seguida, estudamos o comportamento de estruturas de repetições e desenvolvemos modelos analíticos para representar o tempo gasto em laços determinísticos. Após a modelagem dessas estruturas, analisamos e modelamos o comportamento das aplicações do tipo mestre/escravo. Nossa intenção foi desenvolver modelos de predição em função do tamanho do problema (n) e da quantidade de processos utilizados (p). Com isso, estimamos o tempo de execução de aplicações paralelas MPI, variando os dois parâmetros citados e não apenas n como em [Li01a]. Por fim, testamos o que foi proposto através da modelagem e predição de duas versões de um programa de multiplicação de matrizes.

1.5 Organização do Trabalho

O próximo capítulo apresentará alguns modelos, paradigmas e ambientes de programação paralela, seguido por uma descrição detalhada das características e funcionalidades do padrão MPI. No capítulo 3, destacamos as principais técnicas de modelagem e avaliação de desempenho e alguns trabalhos relacionados. No capítulo 4, propomos um modelo gráfico, denominado DP*Graph^{+}, para ser utilizado na representação de programas paralelos MPI. Além disso, desenvolvemos modelos para estruturas de repetições determinísticas e para aplicações do tipo mestre/escravo. No capítulo 5, destacamos o uso de uma ferramenta chamada Scilab durante as atividades de análise e predição de desempenho e apresentamos a modelagem de duas versões diferentes de um programa de multiplicação de matrizes, a fim de validar os modelos analíticos desenvolvidos. Finalmente, algumas conclusões e contribuições de nosso trabalho são apresentadas no capítulo 6. No Apêndice A, apresentamos os códigos dos programas utilizados no desenvolvimento deste trabalho e no Apêndice B comentamos sobre as funções Scilab desenvolvidas.

Capítulo 2

COMPUTAÇÃO PARALELA

2.1 Introdução

Nos tradicionais sistemas computacionais cada instrução do programa é executada sequencialmente, uma após a outra, pela única Unidade Central de Processamento (CPU) que compõe a máquina. No entanto, desde o desenvolvimento dos primeiros computadores, sempre buscamos uma forma alternativa de executar mais instruções simultaneamente. O objetivo sempre foi aumentar o poder computacional e a velocidade de processamento dos computadores para que aplicações complexas pudessem ser resolvidas com o auxílio da computação. Nesse contexto, surgiu a computação paralela, que é vista como uma evolução da computação seqüencial. Assim como no mundo real, onde eventos complexos e intimamente relacionados ocorrem a todo momento e simultaneamente, a computação paralela busca simular esse paralelismo de ações.

Basicamente, podemos dizer que a computação paralela consiste no uso simultâneo de múltiplos recursos computacionais, como por exemplo processadores. Normalmente, os problemas podem ser divididos em partes menores e independentes e resolvidos separadamente, em paralelo, através de múltiplos processos distribuídos entre os processadores disponíveis.

Um dos motivos que impulsionaram o desenvolvimento da computação paralela foi a possibilidade de resolver aplicações complexas até então intratáveis com o auxílio da computação tradicional. Por volta dos anos 80, acreditavam que isso seria possível com o advento de processadores cada vez mais rápidos e eficientes. No entanto, logo notaram que isso não bastaria e começaram a investir em um novo modelo de computação:

processamento paralelo. Essa nova abordagem de desenvolvimento tounou-se uma opção para aqueles que desejam obter alto desempenho em seus sistemas computacionais. Hoje em dia, não só aplicações científicas, mas também comerciais (exploração de petróleo, banco de dados, dentre outras) estão incentivando o desenvolvimento da computação paralela.

Neste capítulo, descrevemos os principais modelos, paradigmas e ambientes de programação paralela e discutimos as características do padrão MPI. Além disso, apresentamos as principais rotinas definidas por esse padrão e mostramos alguns fatores que influenciam o desempenho de programas MPI.

2.2 Modelos de Programação Paralela

Os modelos de programação paralela existem como uma camada de abstração sobre a arquitetura do *hardware* e da memória do computador [Barney02]. No entanto, esses modelos não são específicos de uma determinada arquitetura nem de um tipo de memória. Na literatura encontramos vários modelos de programação paralela, porém, os mais comuns são [Barney02, Moura99]:

- memória compartilhada;
- threads;
- paralelismo de dados;
- passagem de mensagem;
- híbrido.

Geralmente, a escolha do modelo a ser utilizado depende das habilidades e preferências do programador, do tipo de *hardware* disponível e das características da aplicação. Não podemos dizer que existe um modelo melhor que o outro, o que existe são melhores implementações para cada modelo [Barney02]. Nas próximas seções descrevemos, sucintamente, cada um dos modelos mencionados anteriormente.

2.2.1 Memória Compartilhada

Esse modelo de programação é caracterizado pela existência de um espaço de endereçamento comum (memória), compartilhado por todas as tarefas que estão sendo executadas no sistema computacional. No modelo de memória compartilhada, as tarefas podem ler e escrever na memória sem nenhuma restrição. Por isso, visando manter a consistência e a coerência das informações armazenadas, precisamos utilizar mecanismos de sincronização [Ben-Ari90], como por exemplo semáforos, para controlar o acesso à memória.

2.2.2 Threads

No modelo baseado em *threads*, os processos podem ter vários caminhos possíveis de execução concorrente. Cada caminho de execução recebe o nome de *thread*. Apesar de uma *thread* possuir suas próprias variáveis locais, ela também compartilha recursos do sistema com outras *threads*.

Considerando um programa com várias subrotinas, nesse modelo podemos associar uma *thread* a cada uma delas. Com isso, todas as subrotinas podem ser executadas ao mesmo tempo, concorrentemente. Normalmente, o modelo de programação baseado em *threads* é utilizado em arquiteturas de memória compartilhada. Para que as *threads* se comuniquem, elas utilizam a memória global do sistema, lendo e escrevendo sobre variáveis compartilhadas. Assim como no modelo de memória compartilhada, também precisamos utilizar mecanismos de sincronização para evitar que duas ou mais *threads* escrevam, simultaneamente, sobre uma mesma posição de memória.

No início do desenvolvimento desse modelo existiam várias implementações possíveis para as *threads*, elaboradas por cada vendedor de *hardware*. Assim, ficava difícil para o programador desenvolver aplicações pensando em portabilidade. Na tentativa de estabelecer um padrão para esse modelo de programação surgiram duas diferentes implementações de *threads*: *POSIX Threads* [Barney03] e *OpenMP* [OpenMP03].

O *POSIX Threads* se baseia em bibliotecas e requer uma codificação paralela do programa. Essa implementação está disponível apenas para a linguagem C e é comumente chamada de *Pthreads*. O fato do paralelismo da aplicação ser explorado explicitamente no código do programa requer muita atenção do programador quanto aos detalhes de implementação.

Por outro lado, a programação em *OpenMP* é baseada em diretivas do compilador, inseridas no código do programa. Essa versão de implementação está disponível para as linguagens C e C^{++}, sendo portável para as plataformas Unix e Windows NT.

2.2.3 Paralelismo de Dados

A principal característica desse modelo é permitir que as tarefas sejam executadas em paralelo sobre elementos diferentes de uma estrutura de dados regular, como por exemplo vetor ou matriz [Culler99]. Suponha que desejamos somar 5 unidades a cada elemento de um vetor A, que contem 30 posições (A[30]) e temos 3 processos disponíveis para realizar essa operação. Com isso, podemos dividir igualmente o vetor A entre os processos e determinar quais posições cada um irá somar 5 unidades. Assim, cada processo executa o mesmo programa sobre partes distintas do vetor A. Essa característica recebe o nome de SIMD (*Single Instruction Multiple Data*). A Figura 2.1 ilustra esse exemplo.

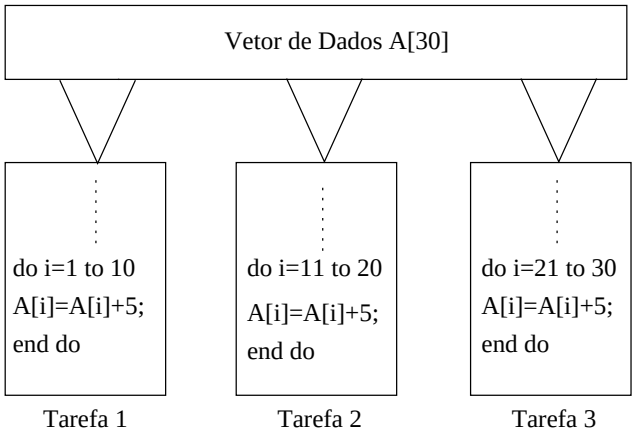

Figura 2.1: Paralelismo de dados (SIMD).

Normalmente, aplicações científicas manipulam grandes estruturas de dados regulares e, por isso, esse modelo é bastante adequado e eficiente para ser utilizado nessa classe de aplicação.

2.2.4 Passagem de Mensagem

As bibliotecas de passagem de mensagem permitem escrever programas paralelos em ambientes de memória distribuída. Nesse modelo de programação, utilizamos primitivas de comunicação explícitas, do tipo *send* e *receive*, para transmitir dados através de uma rede de interconexão. Tipicamente, esse modelo de programação é escolhido quando pretendemos desenvolver programas paralelos para *clusters* de PCs.

Além de oferecer primitivas de comunicação, essas bibliotecas ainda possuem rotinas para inicializar e configurar o ambiente de execução. Nesse ambiente, um conjunto de tarefas é executado simultaneamente por processos distintos, distribuídos entre as máquinas disponíveis. Cada processo utiliza a memória local da máquina em que está rodando. As duas principais "bibliotecas" de passagem de mensagem são: PVM *(Parallel Virtual Machine*) [Oak02] e MPI (*Message Passing Interface*) [Mpi02].

2.2.5 Híbrido

Algumas vezes queremos combinar características de modelos diferentes para desenvolver programas paralelos. Quando isso é desejável, precisamos utilizar um modelo de programação híbrido, que agrupe características de nossos interesses.

Um modelo híbrido, muito utilizado no desenvolvimento de programas paralelos, combina o modelo de passagem de mensagem com os modelos de *threads* ou de memória compartilhada. Um outro modelo híbrido, bastante comum, combina o paralelismo de dados com o de passagem de mensagem. Alguns modelos, como o *High Performance Fortran* (HPF) [Hpf03], utilizam o paralelismo d e dados sobre arquiteturas de memória distribuída. No entanto, esse modelo utiliza passagem de mensagem para transmitir os dados entre as tarefas, de forma transparente ao programador.

2.3 Paradigmas de Programação Paralela

Existem vários paradigmas de programação paralela que podem ser escolhidos pelo programador para estruturar ou organizar o desenvolvimento de programas. A escolha de um ou de outro paradigma depende das características da aplicação, dos recursos

computacionais disponíveis para quem vai desenvolver o programa e do tipo de paralelismo encontrado no problema. Nas próximas seções, explicamos, resumidamente, alguns dos paradigmas mais comumente utilizados na implementação de programas paralelos [Moura99].

2.3.1 Mestre/Escravo

Esse paradigma é caracterizado pela existência de duas entidades fundamentais: uma mestre e outra escravo. A idéia desse paradigma é atribuir ao mestre a característica de coordenador da execução das atividades e aos escravos a função de executar as tarefas e as "ordens" do processo mestre.

Normalmente, o processo mestre divide uma tarefa em subtarefas e as distribui entre os escravos existentes. Em seguida, cada um dos escravos deve resolver suas tarefas e retornar os resultados ao mestre. Depois de receber os dados enviados por todos os escravos, o processo mestre se encarrega de produzir o resultado final da computação. Vale ressaltar que, normalmente, todas as comunicações existentes na aplicação acontecem entre entidades distintas (mestre e escravo) e não entre os próprios escravos ou mestres.

Os processos escravos podem estar espalhados em várias máquinas e ser submetidos a diferentes cargas de trabalho. Por isso, visando distribuir melhor os trabalhos entre os escravos, esse paradigma pode utilizar algum tipo de balanceamento de carga. O balanceamento de carga, estático ou dinâmico, pode influenciar o desempenho das aplicações.

No balanceamento estático, a divisão e a distribuição das tarefas são feitas no início do programa e conta com a participação do mestre no processo de computação. Já o balanceamento dinâmico pode ser aplicado quando existe a presença de nós de processamento saturado, quando não conhecemos o número de tarefas criadas no início da execução do programa a serem distribuídas ou no caso em que o número de tarefas a serem executadas ultrapassa a quantidade de processadores disponíveis no sistema.

As vantagens obtidas no uso do paradigma mestre/escravo são: o alto grau de escalabilidade alcançado e o *speedup* normalmente obtido. Por outro lado, devemos tomar alguns cuidados para que o controle centralizado atribuído ao mestre não se torne um

gargalo para o sistema. Em situações com grande quantidade de processadores disponíveis, a existência de um único mestre pode prejudicar o desempenho do sistema. Para que isso não ocorra, podemos aumentar o número de processos mestres e fazer com que cada um deles cuide de um grupo distinto de processos escravos.

2.3.2 Divisão e Conquista

O paradigma de divisão e conquista é bem conhecido e muito utilizado no desenvolvimento de programas seqüenciais. A idéia é dividir recursivamente um problema em vários subproblemas, resolvê-los individualmente e combinar os resultados obtidos para gerar o resultado final.

Em uma outra abordagem desse paradigma, os subproblemas podem ser resolvidos em paralelo, diminuindo consideravelmente o tempo de execução total do programa. Ao tornar cada subproblema independente dos demais, estaremos elimininando qualquer necessidade de comunicação entre os processos.

2.3.3 Modelos Híbridos

Um modelo híbrido é construído e utilizado quando uma determinada aplicação permite explorar, com eficiência, mais de um tipo de paralelismo presente em suas características. Normalmente, esses paradigmas são aplicados na resolução de problemas de grande porte. Assim, conceitos e estratégias de diferentes paradigmas são empregados para estruturar e propor uma boa solução computacional.

2.4 Ambientes de Programação Paralela

Para que possamos utilizar e explorar todos os recursos disponíveis em um computador paralelo, devemos conhecer as linguagens e os ambientes de programação disponíveis, além dos modelos e paradigmas de programação existentes. Esse conhecimento é importante para escrevermos programas paralelos eficientes. A classificação dos ambientes estão baseadas nos paradigmas de programação que podem ser utilizados em cada um deles. Nas próximas seções apresentamos os dois ambientes de programação paralela mais conhecidos [Pramanick99].

2.4.1 Ambientes de Memória Compartilhada

Os ambientes de memória compartilhada são caracterizados pela presença de vários processadores, compartilhando o acesso a uma única memória. Os processadores podem funcionar de forma independente, mas qualquer mudança no conteúdo das variáveis armazenadas na memória é visível a todos os outros processadores.

Baseado no tempo de acesso à memória, gasto por cada processador pertencente ao sistema, podemos dividir as máquinas de memória compartilhada em 2 classes: UMA (*Uniform Memory Access*) e NUMA (*Non Uniform Memory Access*).

Nas máquinas pertencentes à arquitetura UMA, o tempo gasto no acesso à uma mesma posição de memória é igual para qualquer processador. Infelizmente, com o aumento natural do número de processadores que compõem essas máquinas, o barramento de acesso à memória pode ficar saturado e se tornar um gargalo para o sistema.

A arquitetura NUMA surgiu para solucionar a saturação de barramento mencionada anteriormente. Cada processador possui um módulo de memória associado, utilizado somente por tarefas locais. O conjunto de todos esses módulos de memória formam a memória global do sistema. Ao contrário das máquinas da arquitetura UMA, nessa classe de memória compartilhada os processadores não possuem o mesmo tempo de acesso à memória. A comunicação entre os processadores acontece através da leitura e escrita de dados na memória compartilhada pelo sistema.

O ambiente de memória compartilhada oferece algumas vantagens e desvantagens sobre os demais ambientes de programação paralela [Barney02]. Como principais vantagens, podemos citar:

- a existência de um espaço de endereçamento global torna a programação nesse ambiente bastante amigável para o programador;
- como a memória está próxima às CPUs, o compartilhamento dos dados entre as tarefas é rápido e uniforme.

Por outro lado, citamos como desvantagens:

- a baixa escalabilidade do número de processadores, uma vez que o aumento demasiado de CPUs pode congestionar o barramento de acesso à memória;

- a dificuldade e necessidade em manter a coerência de cache;
- a sincronização entre os acessos à memória global é de responsabilidade do programador;
- o preço para projetar e produzir máquinas de memória compartilhada é ainda muito alto.

2.4.2 Ambientes de Memória Distribuída

Nesse modelo, cada processador possui sua própria memória local e não existe uma memória compartilhada pelo sistema. Com isso, os processadores podem trabalhar independentemente, acessando somente sua memória local sem afetar os dados utilizados pelos demais processadores.

Eventualmente, um processador precisa acessar dados armazenados na memória local de outros processadores. Quando isso for necessário, cabe ao programador definir, explicitamente, como e quando o dado será acessado. Para isso, barreiras de sincronização entre as tarefas que estão sendo executadas em cada processador devem ser definidas, a fim de coordenar as atividades.

Esses ambientes têm impulsionado a utilização dos paradigmas de passagem de mensagem, tais como o PVM e o MPI. Esses paradigmas utilizam, explicitamente, primitivas de comunicação, como *send* e *receive,* para realizar a transferência de dados ou de mensagens entre os processadores distribuídos pelo sistema.

Esse ambiente também possui algumas vantagens e desvantagens em relação aos demais[Barney02]. Como vantagens citamos:

- a quantidade de memória do sistema aumenta com a adição de novas CPUs, podendo melhorar o desempenho das aplicações. Para inserir um novo processador ao sistema computacional também é necessário adicionar uma memória local;
- cada processador pode acessar rapidamente sua memória local, sem nenhuma interferência dos demais processadores;
- ausência da necessidade de manter a coerência de cache.

As maiores desvantagens encontradas nesse ambiente são:

- o tempo de acesso à memória não é uniforme (NUMA);
- muitos detalhes associados à comunicação existente entre os processadores ficam sob a responsabilidade do programador;
- a dificuldade em mapear estruturas de dados já existentes em ambientes de memória global.

Na próxima seção, apresentamos algumas características da interface de passagem de mensagem MPI, utilizada com maior freqüência nos ambientes de memória distribuída, além de suas principais primitivas de comunicação.

2.5 Programação com Passagem de Mensagem (MPI)

Esse paradigma de programação paralela, ao contrário do que muitas pessoas pensam, não está restrito à aplicações acadêmicas e científicas. O interesse comercial pela programação com passagem de mensagem é antigo. Os primeiros computadores paralelos para fins comerciais utilizavam o paradigma de passagem de mensagens no desenvolvimento das aplicações. Assim, visando interesses de cada fabricante, muitas bibliotecas de passagem de mensagem foram implementadas para um *hardware* específico. As principais diferenças encontradas entre as implementações estavam relacionadas à sintaxe das primitivas de cada biblioteca.

Por esses motivos, houve a necessidade de criar uma biblioteca padrão que pudesse ser utilizada sobre o *hardware* de diferentes fabricantes. Com esse intuito surgiu o MPI (*Message Passing Interface*), definido como uma especificação para os programadores e usuários de bibliotecas de passagem de mensagem. O MPI é capaz de prover um ambiente prático, portável, eficiente e flexível para o desenvolvimento de aplicações paralelas de alta performance [Al-Tawil94, Barney02b]. Por esses e outros motivos, ele tem se tornado um padrão emergente para a implementação de programas paralelos sobre os ambientes de memória distribuída [Nupairoj94].

A definição do padrão MPI contou com a participação de mais de 40 organizações, dentre elas: fabricantes de *hardware*, programadores e pesquisadores. O principal objetivo desse padrão é oferecer uma biblioteca de programação eficiente e amplamente portável, sem prejudicar o desempenho do programa [Al-Tawil94]. Além disso, esse padrão é visto como uma camada de comunicação flexível, que dispõe de vários mecanismos para a transferência de mensagems, ponto-a-ponto ou coletivas [Nupairoj94, Al-Tawil94].

Desde a definição do padrão, várias implementações surgiram e se tornaram publicamente disponíveis, tais como: CHIMP [Alasdair94], LAM [Burns94], MPICH [Gropp94]. Apesar dessa variedade de implementações, todas apresentam um desempenho semelhante quando comparadas à uma mesma plataforma e configurações [Nupairoj94]. Neste trabalho, utilizamos a implementação LAM durante os testes experimentais.

Nessa abordagem, os programas são constituídos por um conjunto de processos, que podem estar distribuídos em diferentes máquinas, executando trechos de códigos independente ou cooperando na resolução de um problema. Para que isso seja possível, os processos trocam informações (dados) através de primitivas de comunicações disponibilizadas pelo MPI.

Normalmente, esse tipo de paradigma de programação é utilizado sobre redes de estações de trabalhos (NOWs) ou *clusters*. Nessas plataformas, as trocas de mensagens entre cada um dos processadores/nós são utilizadas tanto para a transferência de dados, quanto para sincronizar tarefas.

No MPI, as duas operações básicas para trocas de mensagens são: *send* e *receive*. Para escrever qualquer programa paralelo, ainda é necessário ter uma função capaz de especificar a quantidade de processos que estão participando da computação e uma outra capaz de retornar um identificador associado a cada processo. Essas funções são chamadas de *numprocs* e *taskid*, respectivamente. As demais operações surgiram para facilitar a implementação de algumas comunicações de comportamentos distintos, que apresentam características específicas das aplicações (um *broadcast*, por exemplo).

As primitivas de comunicação utilizam o processamento disponível da CPU para transmitir os dados entre os diferentes nós de um *cluster* e, conseqüentemente, acabam “roubando” o processamento que poderia estar sendo utilizado para realizar as

computações locais. No entanto, as plataformas recentes oferecem um suporte adicional, através do *hardware*, para enviar e receber mensagens. O *hardware* da *interface* de rede suporta o DMA (*Direct Memory Access*) e transferência de mensagem assíncrona. Essa interface de rede possibilita a transferência das mensagens de um *buffer* de memória para o *buffer* de destino, sem qualquer intervenção da CPU. Com isso, os processos não precisam interromper a sua computação local para enviar ou receber uma mensagem.

Dentre as razões para usarmos o MPI nas implementações de programas paralelos, citamos [Barney02b, Mpi02]:

- padronização: o MPI foi definido como um padrão para implementação de bibliotecas de passagem de mensagem;
- portabilidade: não existe a necessidade de modificar o código fonte de um programa MPI para portar uma aplicação a outras platarformas;
- desempenho: fabricantes de *hardware* podem explorar as características do próprio equipamento para melhorar o desempenho de programas MPI;
- funcionalidade: existem mais de 115 rotinas definidas no padrão;
- disponibilidade: muitas implementações do padrão podem ser encontradas, seja elas de domínio público ou particular.

Originalmente, o MPI foi projetado e desenvolvido para sistemas de memória distribuída. Hoje em dia, ele também está sendo utilizado para implementar alguns modelos de memória compartilhada, como o paralelismo de dados, sobre arquiteturas de memória distribuída [Barney02b].

2.5.1 Definição de Grupos e *Communicators*

A maioria das rotinas do padrão MPI exige a especificação de um parâmetro que determina o conjunto de processos envolvidos na transmissão de uma mensagem. Um grupo pode ser definido como um conjunto ordenado de processos, sendo que cada processo possui um identificador único (*id*) [Barney02b]. O grupo definido pelo programador é

representado, dentro do sistema de memória, como um simples objeto que está sempre associado a um *communicator*. Assim, um *communicator* envolve um grupo de processos que poderão trocar mensagens entre si. Para os programadores, que utilizam o MPI, esses dois conceitos se misturam e se fundem em uma única definição.

Assim, os grupos e *communicators* devem permitir:

- a organização de tarefas baseada em suas funções (grupos de tarefas);
- operações de comunicação coletiva para um subconjunto de processos;
- especificação e implementação de topologias virtuais pelo programador.

Algumas restrições e considerações devem ser observadas durante a definição de grupos na implementação de programas MPI. Tanto os grupos, quanto os *communicators* são objetos dinâmicos e, portanto, podem ser criados ou destruídos durante a execução do programa. Os processos criados podem fazer parte de mais de um grupo definido pelo programador, sendo identificado por *ids* diferentes em cada um deles.

Existe um conjunto de primitivas destinadas ao gerenciamento dos grupos ou *communicators* e as principais funcionalidades associadas são:

- determinar o grupo associado a um dado *communicator*;
- retornar o identificador (*id*) do processo dentro do grupo;
- retornar a quantidade de processos que fazem parte de um grupo;
- combinar dois grupos distintos (união de grupos).

Ainda existem muitas outras primitivas com funcionalidades específicas, disponíveis para os usuários do MPI. As demais primitivas de gerenciamento de grupo e a especificação formal da sintaxe de cada uma delas podem ser encontradas em [Barney02b, Mpi02].

2.5.2 Rotinas de Gerenciamento do Ambiente MPI

O principal propósito das rotinas de gerenciamento é inicializar e finalizar o ambiente MPI. Contudo, também devemos destacar a importância das demais rotinas. Assim, apresentamos nesta seção algumas das principais rotinas de gerenciamento encontradas no padrão:

- **MPI_Init**: inicializa o ambiente. Deve ser chamada antes de qualquer outra rotina MPI;
- **MPI_Comm_size**: determina o número de processos que está sendo utilizado na aplicação;
- **MPI_Comm_rank**: devolve o identificador (id) do processo que executou a rotina. Onde $0 \leq id \leq numprocs - 1$;
- **MPI_Finalize**: finaliza o ambiente de execução. Essa rotina deve ser a última a ser chamada nos programas MPI.

Algumas outras rotinas de gerenciamento do ambiente ainda podem ser encontradas em [Barney02b, Mpi02].

2.5.3 Rotinas de Comunicação

O padrão MPI define a semântica de várias primitivas de comunicação destinadas a trocar mensagens (dados) ou a estabelecer um sincronismo entre os processos criados e que estão empenhados cooperativamente na resolução de um problema. Existem primitivas que se restringem a enviar ou receber dados de um único processo a cada chamada, outras são capazes de considerar grupos de processos e algumas sincronizam a execução das tarefas entre cada um dos processos. No entanto, antes de apresentarmos cada uma das primitivas disponíveis na implementação LAM do padrão MPI, vamos definir alguns conceitos básicos e importantes para compreender a semântica de cada uma delas. Segundo [Burns94, Mpi02, Barney02b], podemos classificar as primitivas de comunicação do padrão MPI de acordo com o número de processos envolvidos na comunicação e do modo de transmissão das mensagens.

Quanto ao número de processos envolvidos na comunicação, as primitivas podem ser classificadas em:

- **Ponto-a-ponto**: envolvem a transmissão de mensagens entre dois processos distintos, enquanto um envia o outro recebe os dados.
- **Coletiva**: envolvem todos os processos de um grupo (*communicator*) previamente definido. Em uma operação coletiva, temos várias mensagens ponto-a-ponto sendo executadas quase que concorrentemente. Cada processo, participante do grupo envolvido na comunicação, deve chamar a mesma primitiva MPI. As principais operações coletivas definidas pelo padrão são: *broadcast*, *gather*, *scatter* e *reduce*. As demais operações coletivas encontradas nas diferentes implementações do padrão são combinações ou variações dessas.

Por outro lado, em relação ao modo de transmissão utilizado durante uma comunicação, as primitivas podem ser:

- **Bloqueante**: quando uma primitiva bloqueante é chamada, ela não permite que o processo continue a execução de suas instruções até que os recursos utilizados (como um *buffer* de dados, por exemplo) estejam liberados para reuso. Isso ocorre para ter certeza de que os dados armazenados no *buffer* não serão afetados por qualquer tipo de modificação. No caso do *receive bloqueante*, o processo receptor só pode continuar sua execução após ter recebido os dados da mensagem e esses estarem prontos para serem utilizados pelo programa.
- **Não-bloqueante**: permite que o processo retorne a sua execução normal imediatamente após a chamada da rotina. Isso significa que nenhum tipo de comunicação pode ter acontecido. Conseqüentemente, não é seguro modificar o conteúdo do *buffer* antes de saber realmente se a mensagem foi transmitida. Nesse tipo de comunicação, o programador não consegue prever o momento em que a mensagem será enviada. Um dos objetivos de utilizar esse modo de comunicação é melhorar o desempenho das aplicações, pois o processo emissor não fica bloqueado em nenhum instante.

2.5.3.1 Rotinas de Comunicação Ponto-a-Ponto

As rotinas de comunicação *send* e *receive* podem assumir diferentes modos de transmissão, de acordo com as necessidades do programador. A escolha de um ou outro modo pode estar relacionado com a sincronização requerida pelos processos ou com a necessidade de desempenho, imposta por determinadas aplicações [Burns94].

O padrão MPI define os seguintes tipos de comunicações ponto-a-ponto: *send* e *receive* bloqueante e não-bloqueante, *send synchronous*, *send ready*, *send standard* e *send buffered*. Vale ressaltar que podemos combinar qualquer tipo de *send* e *receive* em uma comunicação. Assim, os possíveis modos de *send* são:

- ***standard***: o processo que está enviando a mensagem só pode continuar a execução de suas instruções depois que o *buffer* da aplicação existente no processo emissor estiver livre para ser reutilizado.
- ***synchronous***: o processo *send* fica bloqueado até que o *buffer* da aplicação do processo emissor esteja livre para ser reutilizado e que o processo receptor já tenha começado a receber a mensagem. Dependendo da implementação do padrão MPI, esse modo de comunicação é confundido com o modo anterior.
- ***ready***: deve existir um protocolo de confirmação para o processo emissor do estado atual do processo receptor. O processo que envia os dados manda uma mensagem para o receptor perguntando se o mesmo está pronto para receber a mensagem. Se sim, o emissor inicia a transmissão da mensagem; caso contrário, ele fica esperando até que o outro processo esteja apto a receber os dados.
- ***buffered***: permite ao programador definir um *buffer* onde os dados serão copiados e armazenados até serem transmitidos. A execução do processo emissor continua após os dados terem sido copiados do *buffer* da aplicação para o *buffer* alocado pelo programador.

Como podemos observar na tabela 2.1, todas as rotinas de comunicação ponto-a-ponto bloqueantes seguem um mesmo padrão para os parâmetros utilizados durante a comunicação. A seguir, explicamos o significado de cada um deles:

Primitivas	Sintaxe das Primitivas
Send Standard	*MPI_Send (buffer, count, datatype, dest, tag, comm)*
Send Synchronous	*MPI_Ssend (buffer, count, datatype, dest, tag, comm)*
Send Ready	*MPI_Rsend (buffer, count, datatype, dest, tag, comm)*
Send Buffered	*MPI_Bsend (buffer, count, datatype, dest, tag, comm)*
Receive	*MPI_Recv (buffer, count, datatype, source, tag, comm, status)*

Tabela 2.1: Primitivas ponto-a-ponto bloqueantes.

- **buffer**: nas primitivas de envio de mensagem, o *buffer* é o espaço de endereçamento alocado pelo programa para armazenar os dados que serão enviados. Já nas primitivas de recebimento, o *buffer* é o espaço alocado para armazenar os dados recebidos;
- **count**: é a quantidade de elementos do tipo *datatype*, enviado ou recebido;
- **datatype**: é o tipo do dado a ser enviado ou recebido. O padrão MPI possui um conjunto básico de tipos pré-definidos [Barney02b, Mpi02]. No entanto, também é permitido ao programador criar outros tipos distintos;
- **dest**: indica qual processo receberá a mensagem enviada. Isso é feito especificando o *id* do processo receptor;
- **source**: indica qual processo enviou a mensagem. De forma semelhante ao parâmetro *dest*, isso é feito ao especificar o *id* do processo emissor;
- **tag**: é um número inteiro não negativo escolhido pelo programador para identificar uma mensagem. O parâmetro *source* indica o processo emissor da mensagem e o *tag* indica qual mensagem será recebida;
- **comm**: indica quais processos participarão das comunicações;
- **status**: utilizado na primitiva *receive* para verificações de *status* da transmissão da mensagem.

A tabela 2.2 apresenta as principais primitivas ponto-a-ponto não-bloqueantes. Nas operações não-bloqueantes, os processos *send* e *receive* são capazes de continuar a execução de suas instruções antes mesmo da transmissão da mensagem ser completada. Por esse motivo, não podemos garantir que o conteúdo da mensagem esteja

Primitivas	Sintaxe das Primitivas
Send Standard	*MPI_Isend (buffer, count, datatype, dest, tag, comm, request)*
Send Synchronous	*MPI_Issend (buffer, count, datatype, dest, tag, comm,request)*
Send Ready	*MPI_Irsend (buffer, count, datatype, dest, tag, comm, request)*
Send Buffered	*MPI_Ibsend (buffer, count, datatype, dest, tag, comm, request)*
Receive	*MPI_Recv (buffer, count, datatype, source, tag, comm, request)*

Tabela 2.2: Primitivas ponto-a-ponto não-bloqueantes.

totalmente seguro. Assim, cabe ao programador tomar o cuidado de não realizar qualquer alteração sobre os dados que participam da comunicação, antes de verificar se a transmissão foi completada. Por esse motivo, as operações não-bloqueantes são acompanhadas de operações de checagem, que indicam se a semântica da comunicação recém-inicializada foi violada ou não. Essas primitivas são *MPI_Test* e *MPI_Wait* [Barney02b, Mpi02].

O *MPI_Test* é uma primitiva não-bloqueante que indica se a comunicação foi completada. Já o *MPI_Wait* é uma primitiva bloqueante que faz o processo aguardar até que a comunicação seja finalizada.

2.5.3.2 Rotinas de Comunicação Coletiva

As rotinas de comunicação coletiva especificam em seus parâmetros quais processos participarão das trocas de mensagens. Por *default*, o MPI considera que todos os processos existentes estão envolvidos nas comunicações. No entanto, o programador pode definir um conjunto específico de processos (*communicator*), de acordo com as suas necessidades.

O MPI oferece várias primitivas de comunicação coletiva. Algumas são destinadas à sincronização de tarefas (*MPI_Barrier*, por exemplo), outras à transmissão de dados (*MPI_Broadcast*, por exemplo) e as demais realizam algum tipo de operação (redução) coletiva (*MPI_Add*, *MPI_Max*, por exemplo).

A seguir, explicamos o significado de cada uma das primitivas mostradas na tabela 2.3.

Primitivas	Sintaxe das Primitivas
Broadcast	*MPI_Bcast (buffer, count, datatype, root, comm)*
Scatter	*MPI_Scatter (sendbuffer, sendcount, senddatatype, recvbuffer, recvcount,recvdatatype, root, comm)*
Gather	*MPI_Gather (sendbuffer, sendcount, senddatatype, recvbuffer, recvcount, recvdatatype, root, comm)*
Allgather	*MPI_Allgather (sendbuffer, sendcount, senddatatype, recvbuffer, recvcount, recvdatatype, comm)*
Reduce	*MPI_Reduce (sendbuffer, recvbuffer, count, datatype, op, root, comm)*
Allreduce	*MPI_Allreduce (sendbuffer, recvbuffer, count, datatype, op, comm)*
Reduce_scatter	*MPI_Reduce_scatter (sendbuffer, recvbuffer, recvcount, datatype, op, comm)*
Alltoall	*MPI_Alltoall (sendbuffer, sendcount, senddatatype, recvbuffer, recvcount, recvdatatype, comm)*
Barrier	*MPI_Barrier (comm)*

Tabela 2.3: Primitivas de comunicação coletivas.

- **broadcast**: envia os dados armazenados no *buffer* do processo *root* para todos os outros processos do mesmo grupo. A quantidade de dados enviados pelo processo emissor deve ser exatamente a mesma recebida por cada um dos processos receptores;
- **scatter**: envia dados do processo *root* para os demais processos integrantes do grupo. No entanto, a primitiva *scatter* envia partes distintas do dado armazenado no *buffer* para cada processo envolvido na comunicação;
- **gather**: recebe mensagens distintas de cada processo pertencente ao grupo;
- **allgather**: junta os dados de todos os processos do grupo;
- **reduce**: aplica alguma operação de redução sobre os dados de todos os processos do grupo e armazena o resultado no *root*;
- **allreduce**: aplica uma operação de redução e armazena o resultado em todos os processos definidos no grupo;
- **reduce_scatter**: combinação das primitivas *reduce* e *scatter*;

- **alltoall**: essa primitiva faz com que cada processo do grupo realize uma operação de *scatter*. Cada processo envia partes distintas do dado armazenado no *buffer* para todos os outros processos.

2.5.4 Desempenho de Programas MPI

Nesta seção apresentamos alguns fatores que podem contribuir no desempenho de programas paralelos implementados com MPI. Esses fatores podem estar relacionados com as características do próprio ambiente ou da aplicação. Na maioria das vezes, eles são complexos e difíceis de serem estimados. Além disso, o desempenho de aplicações MPI pode sofrer influência de elementos relacionados com a plataforma utilizada, a rede de interconexão, a aplicação e/ou a implementação do próprio MPI [Barney02c].

Em relação aos fatores associados à plataforma, destacamos as configurações da CPU, da memória, o tipo de adaptador de rede utilizado e o próprio sistema operacional. Como sabemos, a velocidade do *clock* e a quantidade de CPUs disponíveis no sistema possuem grande influência sobre o desempenho das aplicações. Da mesma forma, o tipo de memória (SDR/DDR) utilizado também contribui significativamente no desempenho.

Quanto à rede de interconexão, existem fatores associados ao tipo do *hardware* utilizado (*switch* ou *hub*), aos protocolos (TCP/IP, UDP/IP ou outros), à taxa de transmissão e aos possíveis e, imprevisíveis, momentos de saturação da rede. Quando o volume de dados transmitidos pela rede satura a largura de banda total temos um momento de contenção, prejudicando, consideravelmente, o desempenho das comunicações.

Além disso, temos a influência causada por elementos da própria aplicação. A eficiência do algoritmo implementado, a proporção das comunicações em relação à computação local, o padrão de acesso à memória, o tamanho das mensagens transmitidas e a complexidade das operações executadas também afetam o desempenho do programa. Além desses fatores, devemos lembrar que o tipo de comunicação escolhida (bloqueante, não-bloqueante, ponto-a-ponto ou coletiva) também influencia. Quando desejamos desempenho, as primitivas não-bloqueantes são mais adequadas.

Ainda existem fatores relacionados com a implementação do MPI. Pelo fato do padrão MPI não definir qual protocolo de passagem de mensagem (*eager* ou *rendezvous*)

[Barney02c] deve ser utilizado em cada primitiva de comunicação, diferentes implementações (LAM, MPICH, etc) podem utilizar protocolos distintos em uma mesma rotina. Conseqüentemente, variando a implementação do MPI podemos obter diferenças no desempenho do sistema. Além disso, funções internas podem ser implementadas de forma mais eficiente em algumas implementações do MPI do que em outras.

Outro grande responsável pelo desempenho dos programas MPI é o tamanho das mensagens transmitidas entre os processos. Na maioria das vezes, quando aumentamos o tamanho da mensagem, utilizamos melhor a largura de banda disponível na rede e, portanto, melhoramos o desempenho das aplicações.

Como podemos observar, muitas grandezas devem ser consideradas em análise e predição de desempenho de programas MPI. No entanto, é praticamente impossível criar modelos que caracterizem todos esses fatores. Pelo fato da predição de desempenho ser uma estimativa aproximada do valor real que se deseja descobrir, apenas os fatores mais importantes acabam sendo considerados no estudo, dentre eles: a latência da rede de comunicação, o número de processadores utilizados, o tamanho das mensagens e alguns outros *overheads*.

2.6 Considerações Finais

Conforme observamos neste capítulo, existem muitas considerações a serem feitas sobre a computação paralela. O conhecimento dos modelos, paradigmas e ambientes de programação paralela é fundamental antes de iniciarmos qualquer projeto de sistema paralelo. Elaborar programas paralelos eficientes não é uma tarefa trivial diante dos inúmeros fatores que podem contribuir no desempenho das aplicações (seção 2.5.4).

Contudo, o uso crescente dos ambientes de memória distribuída tem favorecido o desenvolvimento de aplicações, que utilizam passagem de mensagem como modelo de programação. Dentre as bibliotecas de passagem de mensagem, o MPI tem se destacado pelas vantagens oferecidas e, principalmente, por ser o único padrão para esse modelo de programação.

Capítulo 3

MODELAGEM E PREDIÇÃO DE DESEMPENHO

3.1 Introdução

Os sistemas paralelos têm sido cada vez mais utilizados na implementação de problemas que exigem um poder computacional maior do que o disponível em máquinas convencionais [Clement94]. Devido à importância desses problemas, em sua maioria científicos, pesquisas têm sido realizadas com o intuito de melhorar ainda mais o desempenho das máquinas atuais. No entanto, a evolução não está restrita somente aos avanços tecnológicos. Embora os processadores e a memória hoje estejam muito mais rápidos, ainda buscamos uma forma alternativa de melhorar o desempenho das aplicações.

A busca por ferramentas e técnicas de avaliação e predição de desempenho de programas paralelos tem sido intensificada, devido a complexidade dos sistemas desenvolvidos para as arquiteturas paralelas. Analisando o programa, tentamos identificar os principais fatores que contribuem no desempenho da aplicação e as possíveis mudanças sobre o código. Para isso, algumas abordagens são utilizadas na modelagem, avaliação e predição de desempenho de aplicações paralelas. Neste capítulo, apresentamos as técnicas de modelagens mais comuns e as diferentes abordagens existentes.

3.2 Modelagem de Programas Paralelos

Para compreendermos os fatores que influenciam o desempenho das aplicações podemos seguir duas abordagens distintas [Meira95]. A primeira abordagem compreende um conjunto sistemático de passos que conduz o estudo da aplicação. Inicialmente, devemos executar o programa sobre o ambiente de teste (testes experimentais) e adquirir medidas de desempenho. Em seguida, baseado nos resultados obtidos identificamos as potenciais alterações sobre o código do programa para melhorar seu desempenho. Depois de modificar o programa, voltamos a executá-lo sobre o ambiente e repetimos o processo descrito. Fazemos isso até que o desempenho alcançado seja satisfatório.

Na segunda abordagem, elaboramos modelos matemáticos para representar o comportamento do programa. As características do sistema paralelo são abstraídas através de um conjunto de funções e parâmetros. Essa abordagem, além de utilizar modelos matemáticos, pode fazer uso de uma representação gráfica (modelos baseados em grafos) para ilustrar a estrutura estática do programa, os relacionamentos e as dependências existentes entre os componentes da aplicação.

Na primeira abordagem, é mais difícil identificarmos as razões ou os fatores que influenciam o desempenho das aplicações. Isso acontece, porque é quase impossível estabelecermos uma relação entre o código do programa e os fatores externos inerentes às características do ambiente. Esses fatores podem estar relacionados com o sistema operacional utilizado, o tipo de memória do sistema, os adaptadores de redes, dentre outros.

Por outro lado, a segunda abordagem permite um estudo mais detalhado sobre os fatores que contribuem no desempenho da aplicação. Esses fatores podem ser representados através dos parâmetros existentes nas funções elaboradas. Os modelos desenvolvidos, não só permitem analisar o desempenho da aplicação, mas também predizê-lo em situações hipotéticas. Uma vez elaborado o modelo do programa, podemos estimar o valor de algumas variáveis, como por exemplo o tempo de execução do programa considerando uma outra quantidade de processadores envolvidos no processamento. Apesar de sabermos que a construção de modelos precisos e ao mesmo tempo simplificados não é uma tarefa fácil, essa abordagem consegue desenvolver modelos de predição mais eficientes. Por esses motivos, ela vem sendo muito utilizada no estudo de desempenho de programas paralelos [Meira95].

Segundo [Meira95], podemos seguir duas estratégias distintas durante a elaboração dos modelos dos programas. A escolha por uma delas depende da intenção da pessoa que desenvolverá o estudo de desempenho. As estratégias citadas em [Meira95] são:

- *Bottom-up*: o sistema é representado através de um conjunto de funções e parâmetros. Utilizamos essa estratégia quando pretendemos investigar a influência de fatores inerentes ao ambiente no desempenho da aplicação. A modelagem analítica utiliza esse tipo de estratégia.
- *Top Down*: considera os aspectos relacionados à implementação da aplicação (código do programa). Esses aspectos incluem o nível de paralelismo empregado durante a implementação das tarefas. Modelos baseados em descrições e análise estatística são exemplos de técnicas que fazem parte dessa estratégia.

Nas próximas seções, descrevemos as técnicas mais comuns para modelagem e avaliação de desempenho de programas paralelos.

3.2.1 Técnicas de Modelagem e Avaliação de Desempenho

Basicamente, utilizamos as técnicas de avaliação de desempenho em duas situações distintas. Na primeira delas, estamos interessados em comparar sistemas diferentes, mas de um mesmo propósito, e descobrir qual deles é mais eficiente. Na segunda situação, desejamos estudar alguns parâmetros do sistema e descobrir qual o melhor valor para cada um deles, de modo que o desempenho da aplicação seja melhorado [Jain91]. Algumas técnicas de avaliação são utilizadas com maior freqüência para esse propósito, dentre elas, destacamos: as medições, as simulações, as modelagens analítica, estrutural e híbrida.

Alguns fatores importantes devem ser previamente considerados para determinarmos a técnica mais adequada aos nossos objetivos. Uma relação desses fatores pode ser encontrada em [Jain91]. Basicamente, eles estão relacionados com o estágio de desenvolvimento da aplicação, a precisão desejada do modelo a ser desenvolvido e o custo de execução da técnica escolhida.

Apesar da relevância individual de cada fator, devemos atribuir um peso maior ao estágio de desenvolvimento da aplicação. Dependendo desse estágio, não podemos

escolher determinada técnica. Por exemplo, não é possível realizar medições de desempenho em um sistema se o mesmo não estiver totalmente implementado.

Além disso, o desempenho das aplicações pode ser caracterizado em termos de suas computações locais e das comunicações existentes [Block95].

3.2.1.1 Medições

Essa técnica pode ser aplicada somente se o programa estiver totalmente implementado. Normalmente, utilizamos as medições em conjunto com outras técnicas de modelagem, avaliação e predição de desempenho, exceto quando desejamos apenas avaliar ou comparar o desempenho de sistemas semelhantes.

A partir dos valores medidos entre execuções consecutivas do programa, podemos perceber a variação do comportamento do desempenho da aplicação. Durante os testes experimentais, medimos os valores de algumas variáveis diretamente relacionadas com o desempenho do sistema, como por exemplo o tempo de execução. Entre um teste e outro, modificamos o código do programa visando melhorá-lo. Esse processo se repete até que o resultado obtido com as alterações seja satisfatório. Um dos objetivos do processo descrito é encontrar informações ou padrões relacionados com a aplicação que poderão ser ajustados para melhorar o desempenho do sistema.

Para alcançarmos o objetivo descrito anteriormente, podemos utilizar monitores de *software* ou de *hardware* [Hu97]. Os monitores são vistos como uma ferramenta de auxílio, utilizada para monitorar determinadas atividades de um sistema. Esses monitores coletam estatísticas de desempenho, analisam os dados obtidos e apresentam os resultados alcançados [Jain91]. Geralmente, os monitores utilizam comandos de linguagem de alto nível para desempenhar suas funções. As variáveis medidas podem variar de um sistema para outro. Por exemplo, em aplicações do tipo cliente/servidor podemos estar interessados em medir o tempo de resposta do servidor, enquanto que em um programa de multiplicação de matrizes desejamos obter o tempo total de execução. Portanto, dependendo das características da aplicação, escolhemos o fator a ser medido a fim de avaliar o desempenho do sistema.

Pelo fato de instrumentarmos o código com os monitores, alguns *overheads* ou perturbações são introduzidos e podem, conseqüentemente, afetar o comportamento

do sistema. Instruções adicionais devem ser executadas, provocando um aumento natural, mesmo que pequeno, do tempo de execução do programa. Por esse motivo, os dados coletados durante os testes experimentais podem não representar precisamente os valores reais.

3.2.1.2 Simulações

A simulação é utilizada tanto em avaliação de desempenho, quanto na validação de modelos analíticos. Ao contrário das medições, as simulações baseiam-se em modelos abstratos do sistema. Logo, ela não exige que o sistema esteja totalmente implementado para ser aplicada. Assim, os modelos utilizados durante a simulação são elaborados através da abstração de características essenciais do projeto do sistema, sendo que a complexidade e o grau de abstração do mesmo pode variar de um sistema para outro.

Durante uma simulação podemos controlar com maior eficiência os valores assumidos por alguns parâmetros do sistema. Com isso, fica mais fácil obter informações relevantes para a avaliação de desempenho. Além disso, as simulações permitem criar modelos mais detalhados, gerando resultados mais precisos.

Através de simulações, podemos verificar e prever o comportamento de programas sobre arquiteturas, ainda não utilizadas em testes. Para isso, existe uma técnica de simulação conhecida como *execution-driven*. Basicamente, essa técnica intercala execuções da aplicação na arquitetura real com simulações sobre a arquitetura desejada [Hu97].

Nessa técnica, uma aplicação paralela é vista como uma composição de eventos globais separados por computações locais. Uma computação local é restrita a um processo específico e não pode ser vista nem influenciada por outros processos. Já um evento global, pode ser visto e influenciado por outros processos. Esse tipo de evento pode ocasionar a mudança do caminho de execução de um programa paralelo. As operações básicas de um evento global são: o acesso à memória compartilhada e as atividades de sincronização [Hu97]. Apesar de ser uma técnica relativamente nova, tem se mostrado bastante eficiente.

3.2.1.3 Modelagem Analítica

A modelagem analítica utiliza um conjunto de equações e funções matemáticas para descrever o comportamento da aplicação. Os fatores que influenciam e interferem no comportamento da aplicação são modelados e representados através dos parâmetros das equações matemáticas. Essas equações são chamadas de modelos analíticos. Apesar desses modelos considerarem parâmetros específicos de uma arquitetura, podem ser facilmente adaptados para outras plataformas. Com isso, é possível realizar a avaliação de desempenho do programa em outros ambientes [Li01a].

Durante a construção dos modelos analíticos, devemos levar em consideração a sua complexidade e praticidade. Esses aspectos e a precisão desses modelos estão diretamente relacionadas com a quantidade de parâmetros existentes nas funções e equações elaboradas. No entanto, modelos de alta complexidade devem ser evitados para facilitar sua aplicação.

Normalmente, para que o modelo analítico não seja muito complexo, muitas simplificações e suposições devem ser feitas, prejudicando a precisão do mesmo [Jain91]. No entanto, os modelos analíticos permitem uma análise mais ampla e aprofundada em relação aos efeitos causados pelos parâmetros definidos nas equações sobre a aplicação. Além disso, também podemos estabelecer possíveis relacionamentos entre cada um dos parâmetros considerados.

A modelagem analítica, quando comparada às demais técnicas de avaliação de desempenho, apresenta menor custo de execução. Para validar os resultados alcançados através dos modelos elaborados, a modelagem analítica pode compará-los aos valores reais medidos em testes experimentais. Esses valores poderão comprovar as predições realizadas através dos modelos analíticos. Por outro lado, esses modelos também podem ser validados através de simulações [Jain91].

Segundo [Meira95], podemos seguir três abordagens distintas ao realizarmos uma modelagem analítica. Essas abordagens são:

1. Modelagem com Parâmetros Escalares: o comportamento do sistema é modelado através de um conjunto de parâmetros escalares, que caracteriza o comportamento do sistema sob algumas condições especificadas durante a sua elaboração. A precisão do modelo depende da quantidade de parâmetros considerados na modelagem. Alguns parâmetros, como *overhead* do sistema operacional,

dificilmente são modelados e, se considerados, podem tornar o modelo muito complexo e difícil de ser aplicado.

2. Modelagem com Funções: expressa a influência de parâmetros relacionados com o *software* e o *hardware* sobre o tempo de execução da aplicação. Apesar dessa abordagem possibilitar o desenvolvimento de modelos mais precisos, quando comparada à modelagem com parâmetros escalares, a complexidade desses modelos pode ser maior. Isso acontece, pois é necessário descobrir o tipo de função matemática (linear, quadrática, logarítmica, etc) que melhor representa o comportamento dinâmico do sistema. Além disso, é preciso reconhecer e modelar os parâmetros que serão utilizados pelas funções, para que o modelo seja consistente.

3. Modelos Estatísticos: utiliza ferramentas de modelagem estatística, como os modelos de Markov, para caracterizar o comportamento do sistema paralelo. Nessa abordagem, os parâmetros definidos são ferramentas estatísticas. Normalmente, esses modelos são utilizados para analisar o comportamento de sistemas que possuem seu *workload* ideal bem definido. Os parâmetros estatísticos são utilizados para representar o comportamento assintótico do sistema.

3.2.1.4 Modelagem Baseada em Descrição

Essa modelagem está baseada em informações ou descrições oferecidas pelo usuário sobre a estrutura e o comportamento do sistema que está sendo modelado. Para a elaboração do modelo do programa, a modelagem baseada em descrição utiliza uma representação gráfica auxiliar (baseada em grafos), denominada *task graph*, que indica os potenciais pontos de paralelização da aplicação. Por outro lado, alguns parâmetros relacionados às características do *hardware* também são considerados durante a elaboração do modelo.

3.2.1.5 Modelagem com Análise Estática

Nessa técnica, o ponto de partida para o processo de modelagem é a própria aplicação. Essa modelagem obtém informações do sistema através de uma análise realizada pelo

próprio compilador. Como a análise do programa acontece em tempo de compilação, dizemos que é uma modelagem estática. Existem ferramentas específicas para o desenvolvimento essa abordagem, no entanto, o alto custo associado inibe a sua aplicação.

3.2.1.6 Modelagem Híbrida

Combina características de vários modelos distintos, utilizando conceitos e estratégias das diferentes técnicas de modelagem existentes. Essa abordagem vem sendo bastante utilizada e os resultados alcançados têm sido bastante satisfatórios.

3.3 Sistemática para Avaliação de Desempenho

Para efetuar uma avaliação de desempenho com qualidade, além de escolher e reconhecer a técnica a ser aplicada, outros aspectos também devem ser considerados. É importante definir e seguir um esquema sistemático, que descreva todos os passos envolvidos em cada etapa do processo de avaliação e predição de desempenho. Essas etapas compreendem atividades que vão desde a definição e compreensão do sistema até a apresentação dos resultados alcançados com os modelos de predição. Apesar de sabermos que as métricas, a carga de trabalho e a técnica de avaliação de desempenho escolhida variam de um sistema para outro, existe um conjunto de passos comum envolvidos (vide Figura 3.1) nessas atividades [Jain91].

Antes de iniciarmos o processo de avaliação, devemos definir os critérios que serão utilizados durante o estudo. Esses critérios são conhecidos como métricas de desempenho [Hu97] e permitem avaliar o desempenho de um sistema específico ou fazer comparações entre sistemas equivalentes. O tempo de resposta de um sistema é uma métrica que pode ser escolhida.

Durante as definições dos critérios, freqüentemente, escolhemos as métricas mais fáceis de serem medidas e calculadas, ao invés das mais relevantes. Isso é um erro comum que deve ser evitado, para que a qualidade do modelo seja priorizada. Não existe um conjunto padrão de métricas que devem ser utilizadas, uma vez que essa escolha depende das particularidades de cada sistema.

Após definidas as métricas de avaliação, devemos identificar as fontes de influência sobre o desempenho do sistema. Os parâmetros que representam essas fontes de

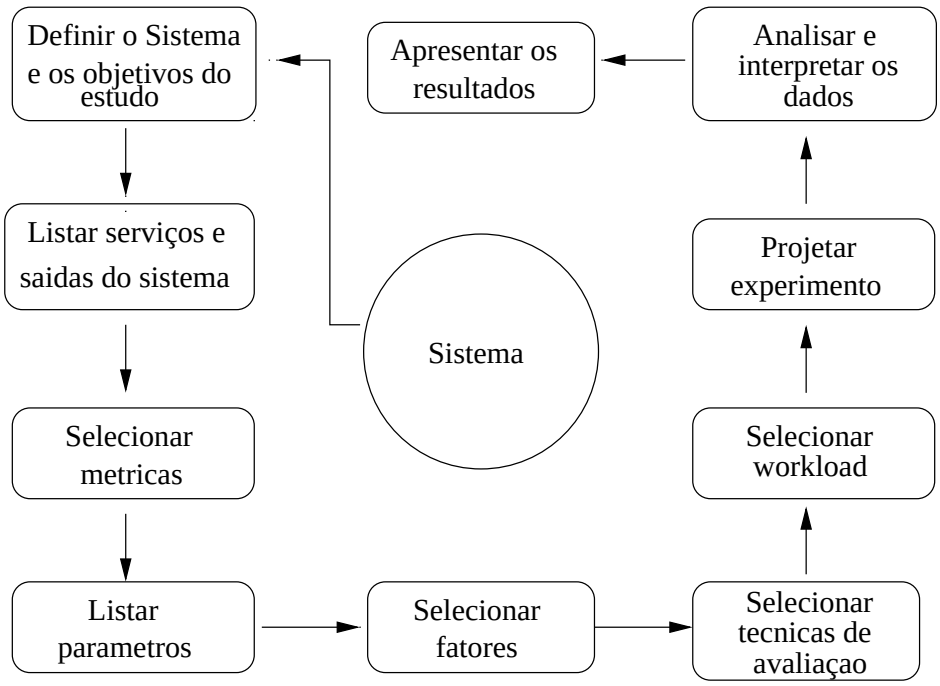

Figura 3.1: Esquema para avaliação e predição de desempenho.

influência identificadas podem ser divididos em duas categorias [Jain91]: de sistema e de *workload*. Os parâmetros de sistema, geralmente, não variam e podem estar relacionados tanto com o *hardware* quanto com o *software*. Já os parâmetros de *workload* estão relacionados com as aplicações.

Segundo [Oed81] apud [Hu97], o *workload* é definido como o conjunto de todas as tarefas individuais, transações, e dados a serem processados em um dado período de tempo. Em outras palavras, podemos dizer que o *workload* é a quantidade de trabalho ou carga computacional submetida ao sistema. É fundamental escolhermos um *workload* adequado para evitar que conclusões precipitadas, e até mesmo erradas, possam ser geradas durante a avaliação de desempenho do sistema.

Após a escolha dos parâmetros descritos anteriormente, devemos escolher a técnica de modelagem e avaliação de desempenho e projetar o experimento, definindo a seqüência do processo. O objetivo principal é alcançar o melhor resultado possível com um esforço reduzido, poupando tempo e trabalho. Durante o projeto do experimento, definimos o número de medições a serem realizadas, a estratégia de seleção dos dados e outras particularidades. Os resultados da avaliação depende muito da qualidade do projeto elaborado.

Por último, devemos analisar e interpretar os resultados alcançados durante os experimentos. Os resultados obtidos nas medições e simulações podem diferir a cada vez que o experimento é repetido. Por esse motivo, saber interpretá-los é fundamental em avaliação de desempenho. Simplesmente apresentar resultados e não compreendê-los restringe muito o trabalho realizado. A apresentação dos resultados e conclusões deve ser feita de forma fácil e prática, de modo que as pessoas possam compreender como o trabalho foi conduzido.

3.4 Predição de Desempenho

Neste trabalho, entendemos por predição de desempenho o ato de estimar valores aproximados de algumas variáveis inerentes à aplicação, como por exemplo o tempo de execução do programa que está sendo avaliado. Dessa forma, a atividade de predição de desempenho consiste em realizar estudos que permitem antecipar e predizer o desempenho de um sistema sob determinadas condições previamente estabelecidas.

Os métodos de predição existentes têm usado, freqüentemente, um modelo de representação baseado em dois níveis. O primeiro nível considera aspectos relacionados com a aplicação, como por exemplo o grau de paralelismo explorado no programa. Já os aspectos relacionados com o ambiente, como o número de processadores utilizados, são considerados no segundo nível de representação [Yan96].

Li [Li01a] também utiliza uma representação de dois níveis para separar os fatores internos e externos, que contribuem no desempenho da aplicação. Os fatores internos podem ser: o tipo de paralelismo utilizado, os métodos de sincronização escolhidos, dentre outros. Como fatores externos, podemos citar: o número de processadores e as características das máquinas utilizadas.

Antes de iniciarmos qualquer predição de desempenho, devemos construir um modelo capaz de representar com precisão as características da aplicação, seguindo alguma das técnicas descritas na seção 3.2.1. Dependendo do grau de abstração que pretendemos explorar, o modelo desenvolvido pode ser simplemesmente uma representação gráfica do programa, elaborado através de grafos orientados. No entanto, um modelo mais detalhado pode ser expressões matemáticas que descrevem o comportamento do programa em função de alguns parâmetros de entrada.

Segundo [Crovella94], os modelos podem ser desenvolvidos através de duas análises distintas da aplicação. A primeira é uma análise estática e se baseia no código do programa. A segunda é uma análise dinâmica e compreende uma inferência baseada em valores conhecidos da função de desempenho da aplicação, obtidos durante os testes experimentais.

A elaboração de modelos gráficos acontece durante a análise estática. O trechos de código do programa, relacionados com as computações locais e as comunicações, são representados através de um conjunto de símbolos. Já o modelo de mais baixo nível é gerado durante a análise dinâmica. Com auxílio de métodos matemáticos, elaboramos funções para representar o comportamento dinâmico do sistema.

3.5 Trabalhos Relacionados

Devido ao grande interesse e importância das atividades de análise e predição de desempenho de programas paralelos, muitos trabalhos podem ser encontrados na literatura. Existem trabalhos que utilizam simplesmente a técnica de medir-modificar, outros baseiam-se em simulações, técnicas estatísticas, modelos analíticos e híbridos para modelar e predizer o desempenho de programas MPI. Nesta seção, apresentamos os principais trabalhos relacionados que foram estudados durante a pesquisa bibliográfica que realizamos.

3.5.1 PEVPM

Em [Grove01] é proposto um sistema capaz de modelar o desempenho de programas paralelos, que utilizam o modelo de programação de passagem de mensagem. Esse sistema, denominado PEVPM (*Performance Evaluating Virtual Parallel Machine*), utiliza máquinas paralelas virtuais para realizar as atividades relacionadas com a avaliação de desempenho.

Inicialmente, o programa é instrumentado com diretivas PEVPM e depois é simulado sobre uma máquina paralela virtual. Ao invés de executar o código do programa, o sistema simula a execução das diretivas que foram inseridas e constrói os modelos de predição. As medidas de desempenho são obtidas através de uma técnica, denominada Monte Carlo [Kalos86] apud [Grove01].

A estrutura do programa é dividida em segmentos de código seqüenciais e de comunicação, que são modelados separadamente. O código seqüencial é modelado através dos métodos tradicionais [Aho83] apud [Grove01], enquanto as comunicações são modeladas com o auxílio de técnicas probabilísticas [Gautama98] apud [Grove01].

Como o estudo é realizado sobre uma máquina paralela virtual, muitos fatores relacionados com os detalhes arquiteturais, que contribuem para a complexidade de uma máquina paralela real, podem ser abstraídos. Com isso, a construção dos modelos torna-se mais fácil.

Segundo o autor do trabalho, o sistema proposto tem condições de elaborar modelos de predição de desempenho precisos a um custo relativamente baixo. A aplicação desse sistema é interessante para as situações em que não temos a máquina alvo do estudo a nossa disposição.

3.5.2 CAPSE

CAPSE [Wabnig93] é um ambiente composto por um conjunto de ferramentas, que dão suporte ao desenvolvimento de *software* paralelo orientado a desempenho. O objetivo é permitir que o desenvolvedor realize atividades de avaliação de desempenho logo no início do ciclo de vida do *software*. Nesse ambiente, existe um gerador automático dos modelos de predição de desempenho e um simulador para obter os resultados das predições.

A ferramenta de predição de desempenho é composta por 3 camadas:

- 1^a camada: denominada camada de especificação. Informações sobre o *workload* do programa, as características do ambiente (*hardware*) e alguns mapeamentos são especificados nessa primeira camada. Esses parâmetros podem ser variados, permitindo analisar o comportamento do programa sobre diferentes condições;

- 2^a camada: chamada de camada de transformação. Obtém informações definidas na camada de especificação e gera o modelo de desempenho do programa paralelo. Redes de Petri temporizadas são utilizadas para analisar as especificações;

- 3^a camada: utiliza o modelo gerado pela camada anterior para realizar a análise e predição de desempenho do programa modelado. Os resultados das predições são obtidos através de um simulador de Redes de Petri.

CAPSE também utiliza um modelo gráfico, baseado em grafos acíclicos direcionados, para representar a estrutura do programa. Nesse grafo, os nós representam as tarefas e os arcos ilustram as comunicações ou alguma relação de dependência entre os nós.

3.5.3 PAMELA

Em [Gemund93a, Gemund96], uma nova metodologia para predição de desempenho de sistemas de computadores paralelos é apresentada. Esses sistemas envolvem o programa paralelo imperativo e a máquina que ele é executado. A metodologia desenvolvida pode ser aplicada em ambientes de memória compartilhada e de memória distribuída.

Nessa metodologia, um modelo simbólico do programa e um da máquina são gerados, através de uma linguagem imperativa de simulação orientada a processo, denominada PAMELA. Após isso, eles são combinados em um único modelo, que une as características do programa e da arquitetura. Por último, é possível realizar estudos de desempenho variando parâmetros de entrada, como o número de processadores e o tamanho do problema.

PAMELA pode ser vista como uma linguagem de simulação de desempenho, permitindo que o modelo seja desenvolvido dinamicamente. Além disso, modelos analíticos podem ser gerados estaticamente em tempo de compilação, através de uma análise sobre a linguagem.

Durante a análise, uma técnica de serialização, baseada em técnicas tradicionais de análise de programas paralelos, é utilizada. Ela inclui um estudo probabilístico sobre os problemas de contenção envolvidos no sistema.

3.5.4 GUBITOSO, M. D.

Em [Gubitoso96], modelos analíticos combinados com simulações são utilizados para predizer o desempenho de aplicações sobre sistemas de memória compartilhada virtual. Nessa abordagem, um sistema de memória distribuída utiliza o paradigma de

passagem de mensagens em um nível mais baixo, mas para o programador é oferecido um sistema de memória compartilhada, facilitando a programação [Nitzberg91] apud [Gubitoso96].

Gubitoso também aprofunda um estudo sobre as estratégias de alocação de processadores para execução paralela de laços com interações independentes (DOALL). Além disso, uma combinação de modelos analíticos com simulações é utilizada para predizer o desempenho de aplicações baseadas em decomposição de domínio sobre máquinas que utilizam comunicação assíncrona.

Um novo método para elaboração de modelos de desempenho baseado na implementação de um sistema, denominado VOTE, sobre uma máquina conhecida como MANNA é proposto. O objetivo principal do método é modelar o custo das sobrecargas. Para isso, todos os pontos do programa que podem gerar uma sobrecarga são identificados. A idéia do método é permitir que os modelos gerados sejam aplicados sobre outras arquiteturas.

3.5.5 FAHRINGER, T.

Uma nova abordagem para predição de desempenho de programas paralelos é apresentada em [Fahringer93]. A atividade de predição de desempenho de programas paralelos é realizada com o auxílio de uma ferramenta denominada P^3T (*Parameter based Performance Prediction Tool*). Essa ferramenta modela três dos mais importantes aspectos de desempenho de um programa paralelo: balanceamento de carga, *overhead* de comunicação e localidade dos dados. Durante a compilação do programa, a P^3T calcula o número de mensagens e a quantidade de dados transferidos, o tempo gasto nessas transferências, o número de falta de cache, a distribuição de trabalho, entre outros parâmetros. Essas medidas auxiliam a realização de análises de desempenho de versões diferentes de um mesmo programa.

Os valores para os parâmetros citados anteriormente, podem ser determinados para conjuntos específicos de instruções, estruturas de repetição (laços), procedimentos específicos ou para todo o programa paralelo. Com base nos resultados obtidos pela P^3T, modificações no código do programa são realizadas através de um compilador denominado VFCS (*Vienna Fortran Compilation System*). O VFCS é um compilador integrado a P^3T, que automaticamente transforma programas Fortran em programas

paralelos. A ferramenta P^3T simplesmente avalia versões diferentes de programas paralelos gerados pelo VFCS e determina qual oferece melhor desempenho.

3.5.6 LI, K. C.

Li propõe uma metodologia de análise e predição de desempenho de programas paralelos implementados com MPI sobre redes de estações de trabalhos (NOWs) [Li01a, Li01b]. Essa metodologia é uma abordagem híbrida, que combina a técnica de modelagem analítica com medições para realizar o estudo de desempenho dos programas MPI.

Uma representação gráfica e outra matemática são utilizadas para avaliar o desempenho dos programas MPI. Para a representação gráfica, foram desenvolvidas duas novas classes de grafos de tempo, uma denominada T-Graph* (alto nível) e a outra DP*Graph (baixo nível). No T-Graph*, apenas a estrutura geral do programa é representada. Já no DP*Graph, os trechos de códigos seqüenciais, as comunicações e o fluxo de execução do programa são ilustrados.

Cada trecho de código do programa paralelo representado no DP*Graph possui características individuais (tempo de execução, quantidade de processamento e de comunicação). Após desenvolver o DP*Graph, o programa é submetido aos testes experimentais sobre um *cluster* de PCs. Durante esses testes, alguns parâmetros, como por exemplo o número de nós e o tamanho do problema, são modificados. Através de medições, diferentes valores para o tempo de execução de cada um dos trechos representados no DP*Graph são coletados. Esses resultados experimentais permitem a elaboração dos modelos analíticos.

Esquema da Metodologia

A aplicação da metodologia deve obdecer uma seqüência de passos sistemáticos, que ajuda a alcançar resultados confiáveis e precisos, conforme descrito a seguir:

1. O primeiro passo consiste na elaboração do DP*Graph, a partir do código fonte do programa paralelo. Para isso, uma análise sobre o código é realizada, identificando e representando através da simbologia do DP*Graph, os trechos de código

correspondentes às computações locais e às comunicações existentes entre os nós do *cluster*. A partir do DP*Graph, podemos desenvolver uma representação do programa paralelo com um nível de abstração maior, denominado T-Graph*.

2. Depois de elaborado o DP*Graph, o código do programa é instrumentado com monitores de tempo que auxiliam a realização das medições. Para cada um dos trechos representados no DP*Graph (seqüenciais ou comunicação), monitores são colocados imediatamente antes e após cada um deles. Assim, através do acionamento e travamento de um *clock* (monitor), é possível medir o tempo gasto em cada um dos trechos identificados.

3. Uma vez instrumentado o programa, testes experimentais sobre o ambiente são realizados.

4. Após os testes, um modelo matemático para cada um dos trechos identificados no DP*Graph é construído, representando a variação do tempo de execução do programa em relação ao número de nós e ao tamanho do problema. Esse modelo está diretamente associado com a natureza e a complexidade do trecho modelado.

5. Depois de elaborados os modelos para cada um dos trechos identificados, o tempo de execução total do programa é dado pela somatória dos tempos gastos em cada trecho. O modelo analítico do programa, utilizado durante as predições de desempenho, é dado pela somatória dos modelos parciais.

6. Por fim, predições de desempenho podem ser realizadas com o auxílio do modelo desenvolvido.

A Figura 3.2 ilustra um esquema gráfico da metodologia proposta por Li.

T-Graph*

O T-Graph* é um modelo gráfico com alto nível de abstração, capaz de representar os trechos de código do programa que está sendo modelado e os locais onde existem

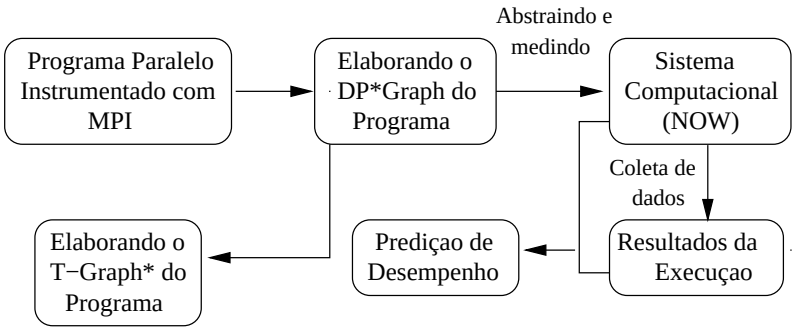

Figura 3.2: Esquema da metodologia proposta por Li

uma divisão ou junção do fluxo de controle. A principal diferença do T-Graph* com o DP*Graph está no nível de detalhes que é representado. Enquanto o DP*Graph ilustra com mais clareza a estrutura e a organização do código da aplicação, o T-Graph* apenas dá uma idéia do programa, mostrando os possíveis caminhos de execução em cada processo modelado.

A Figura 3.3 ilustra os símbolos que compõem o modelo gráfico de alto nível denominado T-Graph*.

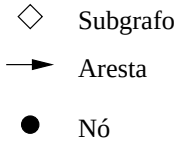

Figura 3.3: Componentes do T-Graph*.

DP*Graph

O DP*Graph é formado por um conjunto de símbolos utilizados para representar detalhes da estrutura do programa paralelo MPI. Com a simbologia oferecida é possível destacar os trechos de código sequenciais e as comunicações existentes. Um dos objetivos desse modelo gráfico é identificar os possíveis gargalos existentes no código do programa e facilitar a sua instrumentação com monitores de tempo para a realização dos testes experimentais.

Uma análise sobre o DP*Graph permite descobrir se o código elaborado explora com eficiência todo o paralelismo da aplicação ou se ainda existem pontos a serem melhorados. A Figura 3.4 ilustra a simbologia utilizada pelo DP*Graph.

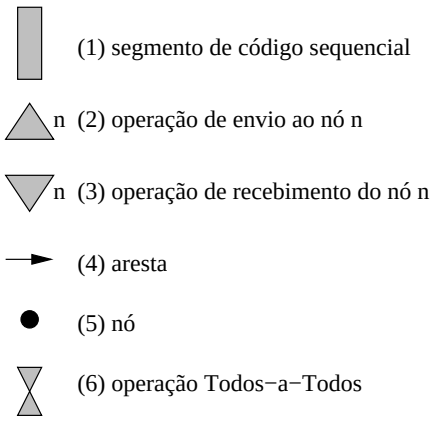

Figura 3.4: Símbolos do DP*Graph.

Para mostrar como é uma representação de um programa paralelo MPI, através do DP*Graph, apresentamos um código de um programa de multiplicação de matrizes, implementado em C com primitivas de comunicação da biblioteca MPI, e em seguida o seu DP*Graph (Figura 3.5).

```
FILE: matrix.c
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include <unistd.h>

#include "mpi.h"
#define MAX_WORKERS 16
double timer[TIMERS][MAX_WORKERS];  /* pega os valores de timer */
double timer_avg[TIMERS];           /* média dos timers */
double tavg;
/* CONSTANTES */
#define NRA 2500       /* números de linhas da matriz A */
#define NCA 2500       /* número de colunas da matriz A */
#define NCB 2500       /* número de colunas da matriz B */
#define MASTER 0       /* taskid do mestre */
#define FROM_MASTER 1
#define FROM_WORKER 2
/* VARIAVEIS GLOBAIS */
MPI_Status status;
double a[NRA][NCA],    /* matriz A */
       b[NCA][NCB],    /* matriz B */

       c[NRA][NCB];    /* matriz resultado C */
/* PROGRAMA PRINCIPAL */
main(int argc, char **argv)
{
int numtasks,   /* número de tarefas */
     taskid,     /* identificador de cada tarefa */
     numworkers, /* número de escravos */
     source,     /* taskid da mensagem de origem */
     dest,       /* taskid da mensagem de destino */
     nbytes,     /* número de bytes na mensagem */
     mtype,      /* tipo da mensagem */
     intsize,    /* tamanho de um int em bytes */
     dbsize,     /* tamanho de um double float em bytes */
```

```
     rows,       /* linhas da matriz A enviada a cada escravo */
     averow, extra, offset,/* variaveis de controle */
     i, j, k,
     count;
FILE *fp;
int tid, wid;
intsize = sizeof(int);
dbsize = sizeof(double);
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
numworkers = numtasks-1;
/************* PROCESSO MESTRE ***************/
if (taskid == MASTER) {
   printf("Number of worker tasks = %d\n", numworkers);
   for (i=0; i<NRA; i++)
     for (j=0; j<NCA; j++)
       a[i][j]= 0.5;
   for (i=0; i<NCA; i++)
     for (j=0; j<NCB; j++)
       b[i][j]= 2.0;
   /* envia a matriz de dados para os escravos */
   averow = NRA/numworkers;
   extra = NRA%numworkers;
   offset = 0;
   mtype = FROM_MASTER;
   for (dest=1; dest<=numworkers; dest++) {
     rows = (dest <= extra) ? averow+1 : averow;
     MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
     MPI_Send(&rows, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
     count = rows*NCA;
     MPI_Send(&a[offset][0], count, MPI_DOUBLE, dest, 3, MPI_COMM_WORLD);
     count = NCA*NCB;
     MPI_Send(&b, count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
     offset = offset + rows;
   }/*end for*/
   /* espera pelos resultados de todos os escravos */
   mtype = FROM_WORKER;
   for (i=1; i<=numworkers; i++) {
     source = i;
     MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
     MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
     count = rows*NCB;
         MPI_Recv(&c[offset][0], count, MPI_DOUBLE, source, mtype,
                  MPI_COMM_WORLD, &status);
   }/*end for*/
}/* fim do mestre */
/************* PROCESSO ESCRAVO *****************/
if (taskid > MASTER) {
   mtype = FROM_MASTER;
   source = MASTER;
   MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
   MPI_Recv(&rows, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
   count = rows*NCA;
   MPI_Recv(&a, count, MPI_DOUBLE, source, 3, MPI_COMM_WORLD, &status);
   count = NCA*NCB;
   MPI_Recv(&b, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
   for (k=0; k<NCB; k++)
     for (i=0; i<rows; i++) {
       c[i][k] = 0.0;
       for (j=0; j<NCA; j++)
         c[i][k] = c[i][k] + a[i][j] * b[j][k];
       }/*end for*/
   mtype = FROM_WORKER;
   MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
   MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
       MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
                TIMER_STOP(15);
} /* fim do escravo */
```

```
    MPI_Finalize();
} /* fim do main */
```

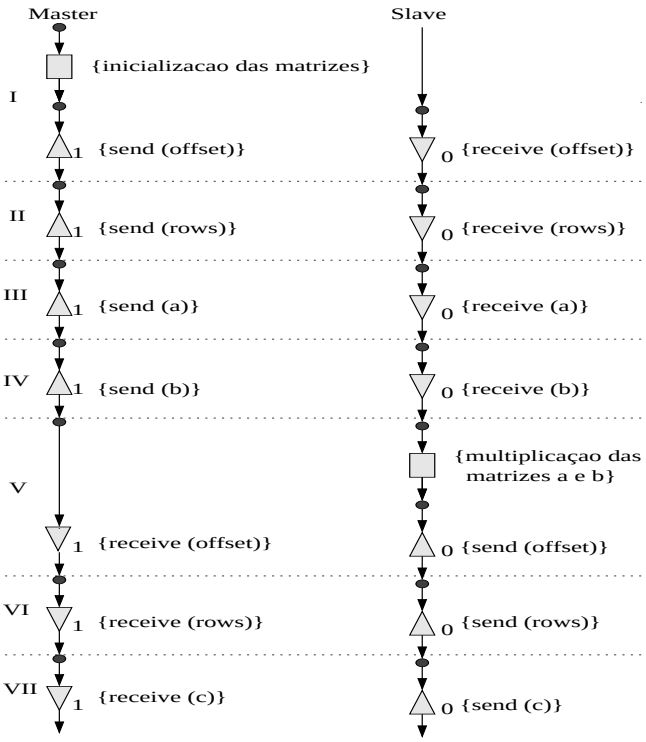

Figura 3.5: DP*Graph do programa matrix.c

Apesar do DP*Graph ter sido desenvolvido para representar detalhadamente os programas paralelos MPI, verificamos que algumas estruturas de código seqüenciais não estão sendo representadas. Dentre elas, destacamos os laços de repetição e as estruturas condicionais. Além disso, através do símbolo utilizado para representar as computações locais não é possível identificar o tipo de código seqüencial que está sendo representado.

Notamos também que não existem símbolos para algumas primitivas de comunicação MPI. Apesar da simbologia adotada pelo DP*Graph torná-lo simples e fácil de ser elaborado, a generalização concedida através dos símbolos utilizados para representar as operações ponto-a-ponto e coletivas tornou o DP*Graph muito abstrato e com pouco poder de representação. Por exemplo, o mesmo símbolo é usado para representar todos os modos de *send* e não existe símbolos individuais para as primitivas de comunicação coletiva.

Uma outra característica que deve ser representada no modelo gráfico diz respeito ao modo de transmissão das mensagens: bloqueantes ou não-bloqueantes. Atualmente, nenhuma distinção é feita entre essas primitivas. Ainda podemos destacar que existem vários modos de transmissão para a primitiva MPI_Send que não estão sendo diferenciados: *standard*, *buffered*, *ready* e *synchronous*.

Para tornar a construção do DP*Graph um processo mais natural e direto, é necessário que as faltas por nós observadas sejam supridas através de novos símbolos. Assim, decidimos realizar uma extensão nos símbolos utilizados pelo DP*Graph e propor uma nova simbologia (vide seção 4.2) para representar os programas paralelos MPI.

3.6 Considerações Finais

Neste capítulo, apresentamos algumas técnicas de modelagem, avaliação e predição de desempenho e discutimos as principais características, vantagens e desvantagens de cada uma delas. Apresentamos também um esquema, que pode ser seguido durante o processo de avaliação de desempenho. No conjunto de passos ilustrados, mostramos as atividades que devem ser seguidas para que o resultado das predições seja confiável.

Apesar de encontrarmos poucos trabalhos que distingüem com clareza os termos análise, avaliação e predição de desempenho, tentamos definir o que é cada uma dessas atividades. Como a predição é uma estimativa de algum valor real, pequenos erros percentuais são aceitáveis e podem ocorrer durante a comparação dos tempos preditos pelo modelo analítico com os medidos nos testes experimentais.

Por fim, relacionamos alguns trabalhos desenvolvidos na área de análise e predição de desempenho e comentamos algumas particularidades encontradas em cada abordagem.

Capítulo 4

DESENVOLVENDO MODELOS DE PREDIÇÃO

4.1 Introdução

Para que uma metodologia de análise e predição de desempenho possa ser aplicada, ela deve oferecer meios para modelarmos todas as situações encontradas nos programas a serem analisados. Tratando-se de uma metodologia para análise e predição de desempenho de programas paralelos MPI, é importante que as primitivas de comunicação ou sincronização existentes no padrão sejam representadas por modelos precisos, devido à grande influência que elas exercem no desempenho desses programas.

Freqüentemente, cada processo que colabora na execução de um programa paralelo apresenta dependências relacionadas com as execuções dos demais. Geralmente, um processo necessita de dados que estão sendo produzidos por outro processo para que ele possa continuar a execução de suas tarefas. No caso do programa de multiplicação de matrizes, apresentado na seção 3.4, o processo mestre deve esperar os resultados das multiplicações, realizadas por cada processo escravo, para continuar a sua execução e determinar a matriz produto C. Desse modo, as dependências de dados entre os processos é um fator importante que deve ser observado e considerado durante a modelagem das aplicações paralelas.

Os pontos de sincronizações existentes entre os processos é outro aspecto a ser observado. Sincronismos podem ser estabelecidos entre os processos através de barreiras

ou de uma comunicação bloqueante. Quando isso ocorre, o tempo gasto nesse sincronismo é dado pelo tempo do processo que demora mais. Além disso, as estruturas de repetição, envolvendo primitivas de comunicação e/ou computação local devem ser previstas e tratadas pela metodologia de predição utilizada.

A metodologia proposta em [Li01a] ainda precisa tratar alguns pontos para tornar-se abrangente. Como exemplo, podemos citar: a modelagem das estruturas de repetição e de programas do tipo mestre/escravo. Assim, pretendemos estender o estudo feito por Li e colaborar com o desenvolvimento da sua metodologia.

Como a ocorrência de laços de repetição no código das aplicações é freqüente, uma metodologia de análise e predição de desempenho deve oferecer meios para modelar essas estruturas. No entanto, algumas complicações acontecem quando tratamos de casos não determinísticos, como estruturas formadas por comandos *while* ou *do-while*. Nesse caso não conseguimos prever, antecipadamente, o número de iterações que serão executadas, dificultando a construção de um modelo analítico que represente o comportamento real da estrutura. Por outro lado, laços do tipo *for* possuem o número de iterações estaticamente definido, facilitando a modelagem.

Em aplicações paralelas, é comum utilizarmos o paradigma mestre/escravo na organização das soluções. Modelar programas com essas características não é uma tarefa trivial, devido aos diversos fatores que influenciam a execução dos processos envolvidos. No entanto, pretendemos desenvolver modelos analíticos simplificados para representar essas aplicações. A partir desses modelos, podemos melhorá-los até obtermos o nível de precisão desejado.

Na metodologia original [Li01a], conforme mostrado na seção 3.5.6, existe um modelo gráfico denominado DP*Graph, que auxilia o desenvolvimento dos modelos analíticos. Estudando as simbologias existentes no DP*Graph, descobrimos que era preciso dar mais poder de representação ao modelo. Por isso, estamos propondo um novo conjunto de símbolos para representar os programas paralelos com passagem de mensagem. Com a nova simbologia, podemos facilmente relacionar a representação gráfica e o código do programa modelado. Entre as expansões realizadas estão: a criação de símbolos individuais para as primitivas de comunicação e a distinção entre as operações bloqueantes e não-bloqueantes.

4.2 DP*Graph$^+$

O DP*Graph$^+$ possui uma representação mais sofisticada do que o DP*Graph. Este novo modelo permite representar o código de um programa paralelo MPI com maior precisão, sem perder muitos detalhes de implementação. Nossa idéia é propor um modelo gráfico que colabore ainda mais na elaboração dos modelos analíticos. Além disso, esperamos que o modelo permita realizar análises mais precisas sobre a estrutura dos programas.

Para suprir a ausência de símbolos destinados à representação de estruturas de repetição determinísticas e comandos condicionais, criamos os novos símbolos apresentados na Figura 4.1. O índice *i*, utilizado na representação dos laços de repetição, indica a quantidade de iterações a ser executada. Já os caminhos A e B, presentes nas estruturas condicionais, podem representar o *then* e o *else* do comando *if*. Esses caminhos de execução podem ser compostos por códigos seqüenciais, comunicação ou ambos.

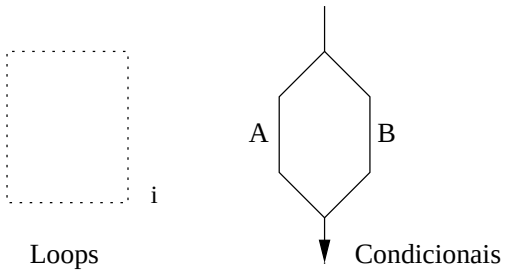

Figura 4.1: Estruturas de repetição e condicionais.

Conforme discutimos na seção 3.5.6, o DP*Graph não utiliza símbolos individuais para representar as comunicações bloqueantes e não-bloqueantes. Para diferenciá-las, elaboramos símbolos distintos capazes de representar os dois modos de comunicação. A estratégia adotada consiste na utilização de linhas ou contornos pontilhados para simbolizar as operações não-bloqueantes e contornos cheios para as operações bloqueantes. A Figura 4.2 exemplifica a nossa estratégia para a representação da primitiva *receive*. A letra y indica o *id* do processo emissor da mensagem, pois como sabemos a primitiva *receive* está sempre associada a uma primitiva *send*.

A distinção das operações bloqueantes e não-bloqueantes também é feita para as primitivas *send*. Essas primitivas, também possuem outros modos de comunicação

Figura 4.2: Símbolos para o *receive* bloqueante e não-bloqueante.

referentes ao protocolo utilizado durante a transmissão da mensagem. Conforme discutimos na seção 2.5.3.1, o *send* pode ser: *synchronous*, *standard*, *buffered* ou *ready*. Para diferenciarmos cada um desses modos, desenvolvemos os símbolos ilustrados na Figura 4.3.

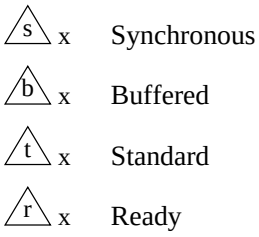

Figura 4.3: Símbolos para os modos de *send* bloqueantes.

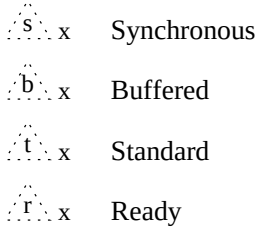

Figura 4.4: Símbolos para os modos de *send* não bloqueantes.

Com a simbologia apresentada nas Figuras 4.2, 4.3 e 4.4, conseguimos representar, individualmente, todas as primitivas de comunicação ponto-a-ponto disponíveis no padrão MPI.

As comunicações coletivas foram representadas atraves de símbolos diferenciados, conforme mostrados na Figura 4.5. Os processos emissor e receptor utilizam símbolos diferentes para representar a mesma primitiva. Essa distinção foi feita pois a mesma primitiva coletiva é utilizada por quem envia e por quem recebe a mensagem. Os trechos de computação local e o sentido de execução do código está representado através dos símbolos da Figura 4.6.

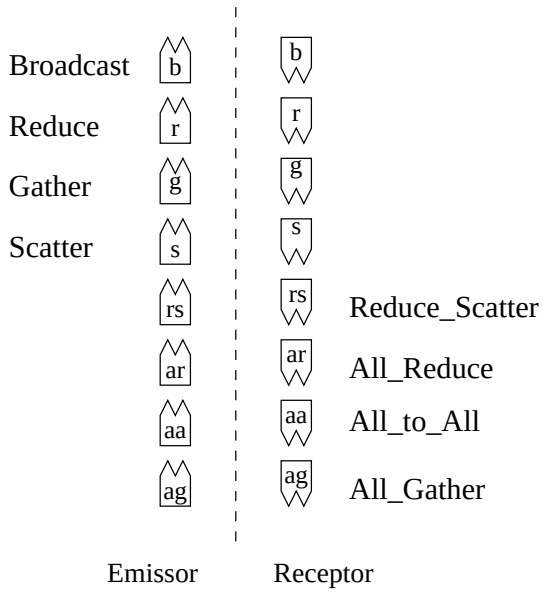

Figura 4.5: Símbolos para as primitivas coletivas.

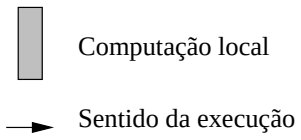

Figura 4.6: Símbolos para computação local e sentido de execução.

Com essa nova simbologia, o programa de multiplicação de matrizes, apresentado na seção 3.5.6, passa a ser representado conforme ilustra a Figura 4.7.

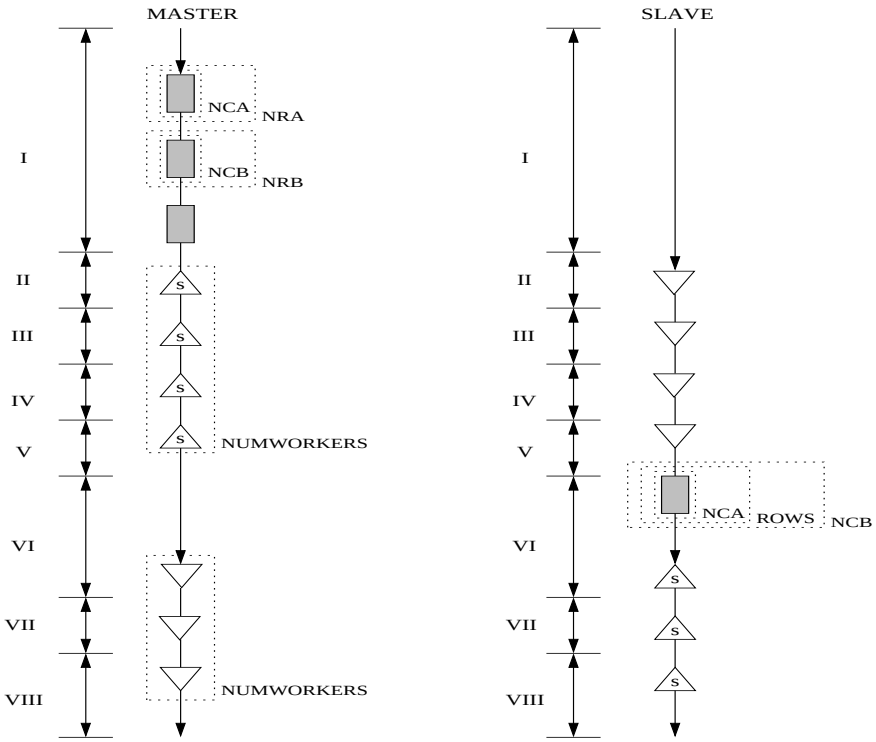

Figura 4.7: DP*Graph^{+} do programa de multiplicação de matrizes.

Como podemos observar na Figura 4.7, os laços de repetição referentes à inicialização das matrizes A e B (trecho I do *Master*) e a multiplicação das matrizes A e B (trecho VI do *Slave*) estão representados no DP*Graph^{+}. Analisando apenas esse modelo gráfico é possível saber que o primeiro trecho seqüencial do processo *master* é melhor representado através de um polinômio do segundo grau, devido a existência de dois laços aninhados. Como sabemos, a complexidade do trecho de inicialização das matrizes A e B é O (n^2). Da mesma forma, sabemos que o trecho VI do processo *slave* tem complexidade O (n^3), devido à existência de três laços aninhados destinados à multiplicar as matrizes A e B.

Conforme podemos visualizar na representação do DP*Graph^{+}, a execução das primitivas *send* e *receive* acontecem dentro de um laço. Na antiga simbologia isso ficava omitido. Percebemos também que todas as primitivas de comunicação ponto-a-ponto utilizadas são bloqueantes, devido ao contorno sólido dos símbolos. Já as

primitivas *send* utilizam o modo de comunicação *standard* durante a transmissão das mensagens. Nas próximas seções, apresentamos o método utilizado no desenvolvimento dos métodos analíticos, os estudos e as modelagens realizadas para as estruturas de repetição e para as aplicações que utilizam o modelo mestre/escravo.

4.3 Metodologia Para o Desenvolvimento dos Modelos Analíticos

A metodologia utilizada durante o desenvolvimento dos modelos analíticos, que caracterizam o comportamento das aplicações estudadas, pode ser resumida através dos seguintes passos:

1. instrumentação do código do programa MPI com monitores de tempo;

2. realização de testes experimentais com o programa instrumentado sobre o ambiente de teste. Durante esses testes, variamos o tamanho da mensagem transmitida (*n*) e a quantidade de processos (*p*);

3. coleta dos tempos envolvidos em cada trecho do programa instrumentado;

4. aplicação de técnicas de ajuste de curva sobre os dados coletados para o desenvolvimento dos modelos analíticos.

O método de ajuste de curvas adotado neste trabalho é o dos mínimos quadrados [Press92]. Esse método gera polinômios de grau *g* e pode ser descrito pelos seguintes passos:

1. Dado o vetor das abscissas $X_{1\times N}$, devemos encontrar a matriz $A_{N\times k}$, onde $k = g + 1$ e $a_{i,j} = x_i^{j-1}$;

2. Calcular a matriz $A^{+}_{k\times N}$ como a pseudo-inversa de A;

3. Encontrar o vetor de coeficientes $C_{k\times 1}$, tal que $C = A^{+} * Y$;

4. O polinômio gerado, denominado modelo analítico, é dado por $y = c_1 + c_2 * x + c_3 * x^2 + ... + c_k * x^{k-1}$.

O método descrito procura encontrar os coeficientes $a_0,\ a_1,\ a_2,\ ...,\ a_g$, minimizando o quadrado da diferença $Y - A * C$:

$$m\acute{i}nimo_{\{a_0,a_1,...,a_g\}}(Y - A * C)^2$$

$$\Sigma_{i=1}^{N}[y_i - (a_0 + a_1 * x + a_2 * x^2 + ... + a_g * x_i^g)]^2 \tag{4.1}$$

4.3.1 Modelagem de Estruturas de Repetição

No trabalho desenvolvido em [Li01a], nenhum modelo analítico específico para as estruturas de repetição foi desenvolvido. Pelo fato dessas estruturas aparecerem com freqüência nos códigos de programas, achamos imprescindível o desenvolvimento de modelos de predição para tal. Assim, nesta seção analisamos como uma estrutura de repetição formada por primitivas de comunicação e/ou computação local pode ter seu tempo de execução representado através de um modelo analítico.

Em [Fahringer93] é apresentado um estudo detalhado sobre estruturas de repetição. Modelos matemáticos para representar a distribuição de trabalho, o número de mensagens transmitidas, a quantidade de dados transferidos e o tempo de transferência das primitivas de comunicação localizadas no interior das estruturas de repetição foram desenvolvidos. Apesar da precisão alcançada, os modelos elaborados apresentam um certo grau de complexidade que acaba inibindo a sua aplicação. Portanto, nossa intenção é trabalhar com estruturas de repetição menos complexas e elaborar modelos analíticos simplificados, mas também precisos e capazes de orientar as atividades de predição de desempenho.

Dentre os possíveis laços existentes em uma linguagem de programação, estamos interessados naqueles em que o número de repetições é conhecido já em tempo de compilação, como por exemplo o *for*. Portanto, nossa análise está restrita apenas às estruturas de repetição determinísticas. Laços de repetição não determinísticos, como

por exemplo o *while,* oferece uma dificuldade maior para ser modelado, pois através de uma análise estática do programa não conseguimos descobrir o número de iterações do mesmo.

Uma estrutura de repetição pode ser formada por simples instruções locais ou primitivas de comunicação. Quando existe apenas instruções sendo executadas localmente, a modelagem do laço de repetição é mais fácil. O problema maior é modelar laços formados por primitivas de comunicação. Nesse último caso, precisamos conhecer o modelo analítico que representa a primitiva de comunicação existente na estrutura de repetição, exigindo um estudo anterior.

Além do que já comentamos, uma estrutura de repetição pode conter outros laços de repetição aninhados. Em uma primeira situação, que chamamos de estuturas de repetição simples, existe somente um laço composto por instruções locais e/ou primitivas de comunicação MPI. Uma segunda possibilidade, é a existência de estruturas de repetição aninhadas que aumentam a dificuldade da análise e do desenvolvimento dos modelos de predição. Nas próximas seções, apresentamos um estudo sobre cada uma dessas situações.

4.3.1.1 Estruturas de Repetição Simples

Uma estrutura de repetição simples, conforme mencionamos, é aquela formada por um único laço composto por instruções locais e/ou primitivas de comunicação. A ausência de laços aninhados facilita o desenvolvimento dos modelos analíticos que serão usados durante as predições de desempenho. Para exemplificar, a seguir apresentamos um trecho de código que ilustra uma estrutura de repetição simples e o seu DP*Graph^{+}correspondente.

```
for (i=1; i < 10; i++){
   a = b * c;
   b = b + 2;
   MPI_Send(&a, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
   count = NCA*NCB;
   NCA = NCA * 4.5;
   MPI_Send(&b, count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
}/*end for*/
```

Para determinarmos o tempo de execução do laço de repetição ilustrado anteriormente, precisamos conhecer o tempo gasto em cada uma de suas instruções internas e

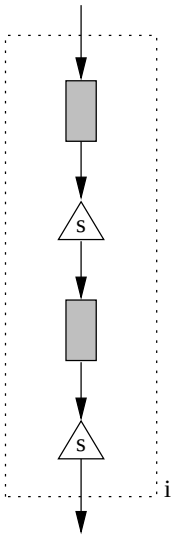

Figura 4.8: DP*Graph^{+} do código que representa a estrutura de repetição simples.

o número de vezes que cada uma será executada (número de iterações do laço). Assim, em uma análise prévia podemos dizer que o tempo de execução total da estrutura é dado pela somatória dos tempos de execução de suas instruções internas multiplicado pelo número de repetições do laço. O tempo de execução das instruções internas pode ser dado por um único modelo analítico ou por uma composição de vários modelos, desenvolvidos para instruções distintas. Além do tempo de execução das instruções que compõem o laço, podem existir outros fatores capazes de contribuir no tempo de execução total, como por exemplo *overheads* adicionais ocasionados nas estruturas de controle do laço de repetição.

Seja R_s uma estrutura de repetição simples, i o número de iterações de R_s e I_j ($1 \leq j \leq n$) o conjunto de todas as instruções internas ao laço. Considerando que essas instruções não dependem de outros processos para serem executadas, o tempo de execução total da estrutura de repetição simples pode ser dado pela seguinte fórmula:

$$t_{exec}(R_s) = i * (\sum_{j=1}^{n} t_{exec}(I_j)) + \alpha \tag{4.2}$$

onde α simboliza *overheads* adicionais, como o tempo gasto durante as comparações dos índices do laço.

4.3.1.2 Estruturas de Repetição Aninhadas

As estruturas de repetição aninhadas são caracterizadas pela presença de dois ou mais laços aninhados, aumentando a complexidade da análise. Assim como nas estruturas de repetição simples, esses laços também podem ser formados por instruções locais e/ou primitivas de comunicação MPI. Um simples exemplo é mostrado através do trecho de código ilustrado a seguir e o seu DP*Graph^{+} pode ser visualizado na Figura 4.9:

```
for (i=1; i < 10; i++){
   a = b * c;
   b = b + 2;
   MPI_Send(&a, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
   for (j=1; j < 5; j++){
       count = NCA*NCB;
       NCA = NCA * 4.5;
       MPI_Send(&b, count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
   }/*end for j*/
}/*end for i*/
```

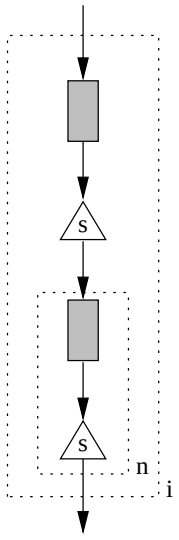

Figura 4.9: DP*Graph^{+} do código que representa a estrutura de repetição aninhada.

A análise do tempo de execução dessas estruturas é conduzida de forma similar à anterior, com algumas diferenças em relação à forma como enxergamos o conteúdo do laço. Basicamente, podemos dizer que o tempo de execução total de uma estrutura de repetição formada por laços aninhados é dado pela soma dos tempos de execução

das instruções que compõem o laço mais externo, mais o tempo gasto na execução dos laços aninhados, vezes o número de iterações do laço mais externo.

Seja Lp um laço de repetição no nível de aninhamento j e I o conjunto de todas as instruções (comandos) que aparecem no corpo de Lp (excluindo as instruções pertencentes aos laços aninhados). Além disso, seja ϑ_L o conjunto de todos os laços no nível de aninhamento $j+1$ que existem no corpo de Lp. Dessa forma, podemos dizer que o tempo de execução de Lp é dado pela seguinte expressão:

$$t_{exec}Lp(Lp) = \sum_{t \epsilon I} t_{exec}(t) + \sum_{l \epsilon \vartheta_L p} t_{exec}Lp(l) \tag{4.3}$$

Para exemplificar como este modelo é aplicado, vamos considerar o exemplo de estrutura de repetição aninhada ilustrada na Figura 4.9. Supondo que temos os seguintes modelos para representar cada trecho do DP*Graph$^+$: $a+b$ é o modelo analítico elaborado para o corpo do laço i (não considerando o laço n interno) e $c+d$ para o corpo do laço n. Assim, aplicando os modelos definidos nas equações 4.2 e 4.3 temos:

$$t_{exec}Lp(i) = (a+b) * i + \sum_{l \epsilon \vartheta_L p} t_{exec}Lp(n) \tag{4.4}$$

$$= (a+b) * i + ((c+d) * n) * i \tag{4.5}$$

$$= (a+b+c*n+d*n) * i \tag{4.6}$$

Se a estrutura de repetição for composta por laços retangulares, portanto $i = n$, o modelo final pode ser dado por:

$$t_{exec}Lp(i) = (a+b) * i + (c+d) * i^2 \tag{4.7}$$

4.3.2 Modelagem de Aplicações Mestre/Escravo

O paradigma de programação mestre/escravo é utilizado com freqüência na implementação de aplicações paralelas envolvendo trocas de mensagens. Na literatura, encontramos uma variedade de aplicações distribuídas que utilizam esse paradigma durante a organização da implementação, dentre elas podemos citar: multiplicação de matri-

zes, métodos de ordenação (*quicksort* e *bubble sort*, por exemplo), cálculo do pi, entre outras.

Como essas aplicações não foram modeladas no trabalho desenvolvido por [Li01a], resolvemos desenvolver modelos capazes de representar o comportamento dessas entidades. Nosso objetivo principal é modelar o tempo de comunicação gasto pelo processo mestre durante a distribuição de tarefas aos processos escravos. O intuito da modelagem é construir modelos analíticos em função de p (número de escravos) e n (tamanho da mensagem), e não apenas em função de n como feito por Li. Com isso, podemos predizer o tempo de transmissão do processo mestre, variando tanto o tamanho das mensagens transmitidas quanto o número de processos envolvidos na comunicação.

A implementação de um programa mestre/escravo com primitivas de comunicação MPI pode ser feita de várias formas. O mestre pode enviar mensagens aos escravos através das seguintes primitivas de comunicação: *send*, *broadcast* e *scatter*.

A opção pelo primitiva *send* merece algumas considerações iniciais. O processo mestre pode enviar as mensagens aos escravos através de uma seqüência de primitivas MPI_Send ou utilizar uma estrutura de repetição (laço) composta pela primitiva de envio citada. No entanto, para o último caso, o desempenho dos programas pode ser prejudicado devido às comparações efetuadas durante as iterações do laço de repetição. Por isso, a primeira estratégia de implementação é mais eficiente. Contudo, escolhemos a segunda opção porque também estamos interessados em analisar o comportamento das estruturas de repetição.

Já as primitivas de comunicação coletivas (*broadcast* ou *scatter*) não necessitam de qualquer estrutura de repetição para serem utilizadas. Uma simples chamada a qualquer uma dessas primitivas é capaz de distribuir os dados aos vários processos receptores, simultaneamente. A principal diferença entre essas primitivas é que o *scatter* tem a capacidade de enviar partes distintas do dado aos diferentes escravos, enquanto o *broadcast* envia sempre o mesmo conjunto de dados a todos os escravos.

Além disso, um programa mestre/escravo pode apresentar algumas configurações relacionadas com o sincronismo estabelecido entre a execução das atividades pertencentes ao mestre e aos escravos. A essas possibilidades denominamos *situações de sincronismo* entre mestre e escravos.

Na primeira situação de sincronismo, o processo mestre alcança a primitiva de envio de mensagem (*send, broadcast* ou *scatter*) antes dos escravos estarem prontos para receberem os dados (*situação 1*). Quando isso acontece e a primitiva de envio de mensagem utilizada é bloqueante, dependendo do tamanho da mensagem enviada, o processo mestre fica bloqueado esperando os escravos iniciarem a recepção das mensagens, ocasionando um atraso (*overhead*) na transmissão das mensagens. Na segunda situação, os escravos ficam prontos para a recepção antes do mestre iniciar o envio das mensagens (*situação 2*). Na última situação, as duas entidades envolvidas na comunicação ficam prontas ao mesmo tempo (*situação 3*). A ilustração dessas situações podem ser visualizadas na Figura 4.10.

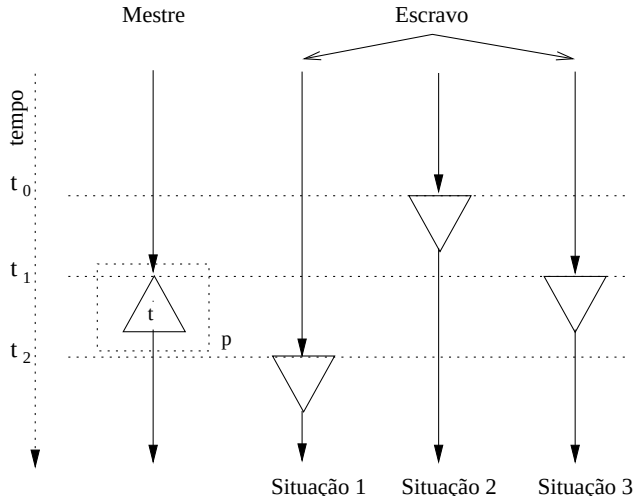

Figura 4.10: Exemplificando as situações de sincronismo.

Em nossos estudos, consideramos somente o tempo de envio da mensagem pelo processo mestre, pois o tempo de recepção gasto pelos processos escravos pode ser facilmente representado através de um modelo analítico para a primitiva MPI_Recv. Considerando apenas o comportamento do mestre, notamos que nas situações 2 e 3 o estado dos escravos não influenciam no tempo de envio do mestre. Em ambas, o mestre inicializa a transmissão das mensagens sem qualquer dependência dos processos escravos, pois os mesmos já estão prontos para iniciar a recepção dos dados. Por esse motivo, apresentamos uma única modelagem para essas duas situações de sincronismo. Por outro lado, na situação 1 o mestre é influenciado pelos escravos que ainda não estão prontos para receber os dados. Por este motivo, devemos diferenciar a modelagem deste último caso.

Com base nessas particularidades, a seguir mostramos os modelos analíticos desenvolvidos para cada um das situações mencionadas anteriormente. Para validar os modelos elaborados, comparamos os valores preditos com os resultados obtidos durante os testes experimentais.

Desenvolvemos um programa simplificado do tipo mestre/escravo para modelar as situações comentadas. Nesse programa, o processo mestre inicializa duas matrizes e envia uma delas aos escravos, através de uma estrutura de repetição formada pela primitiva MPI_Send. Do outro lado, os escravos simplesmente recebem a matriz com a primitiva MPI_Recv. O objetivo desse programa é demonstrar o comportamento do processo mestre durante a transmissão de dados variando n e p. O DP*Graph$^+$ do programa desenvolvido pode ser visualizado na Figura 4.11. Conforme já comentamos, nosso interesse principal é elaborar uma equação matemática em função do número de escravos (p) e do tamanho da mensagem (n) para representar o tempo gasto pelo processo mestre no envio das mensagens. Nas próximas seções, apresentamos os modelos elaborados para representar cada uma das *situações de sincronismo* comentadas.

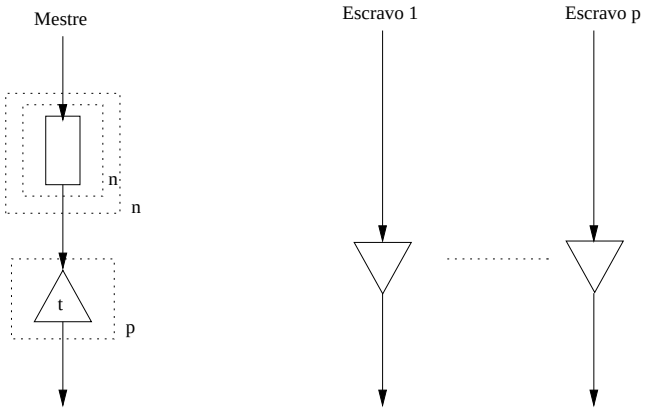

Figura 4.11: DP*Graph$^+$ do programa mestre/escravo utilizado.

4.3.2.1 Situação de Sincronismo 1 (S1).

Nessa primeira situação de sincronismo, o processo mestre alcança a primitiva de envio das mensagens antes que os escravos estejam prontos para iniciar a recepção dos dados. Como primitivas bloqueantes são utilizadas e as mensagens transmitidas são grandes, o mestre fica bloqueado esperando que os escravos inicializem a recepção.

Assim, além do tempo gasto no envio dos dados, um Δt adicional, decorrente da espera ocasionada pelo sincronismo, influencia o tempo de transmissão das mensagens. Conforme podemos observar na Figura 4.11, o processo emissor utiliza uma estrutura de repetição e uma primitiva do tipo *send* para transmitir as mensagens. Assim, podemos dizer que o modelo teórico para predição do tempo de envio dos dados pelo processo mestre pode ser dado por:

$$t^{s1}_{envio}(p,n) = \alpha + p * t_{send}(n) + \eta(p,n) \tag{4.8}$$

onde α é o *overhead* adicional provocado pelo tempo de espera no sincronismo, p é o número de processos receptores (quantidade de iterações do laço de repetição), n é o tamanho da mensagem em bytes, $t_{send}(n)$ é o modelo analítico da primitiva MPI_Send e $\eta(p,n)$ algum *overhead* adicional decorrente de fatores como comparações durante o incremento do contador do laço de repetição, por exemplo.

Nos testes experimentais, variamos o tamanho das mensagens transmitidas e a quantidade de escravos envolvidos na comunicação para verificar o comportamento do laço de repetição com a primitiva *send*. Em ambos os casos, aumentando o tamanho da mensagem transmitida ou o número de escravos envolvidos, o tempo gasto na transmissão aumentou linearmente. Baseado nesse comportamento, elaboramos um modelo analítico (polinômio de grau 1) para representar o tempo de envio das mensagens dentro do laço de repetição.

Para cada configuração (*n*, *p*) realizamos 30 execuções e utilizamos a média dos tempos medidos para comparar com os valores preditos e determinar o erro percentual obtido. Esse cuidado também foi adotado durante os testes com os demais casos de sincronismos.

Os resultados dos testes experimentais, referente a primeira situação de sincronismo, podem ser visualizados no gráfico ilustrado na Figura 4.12. Nesse gráfico também podemos verificar o comportamento do modelo de predição elaborado e alguns erros percentuais.

Durante os testes, verificamos que o tempo de *send* gasto com a primitiva MPI_Send para o primeiro escravo era um pouco menor que o tempo de envio para os demais. Como cada iteração do laço de repetição é executada uma após a outra, o *buffer* de rede do mestre pode estar ocupado por informações da primeira mensagem quando

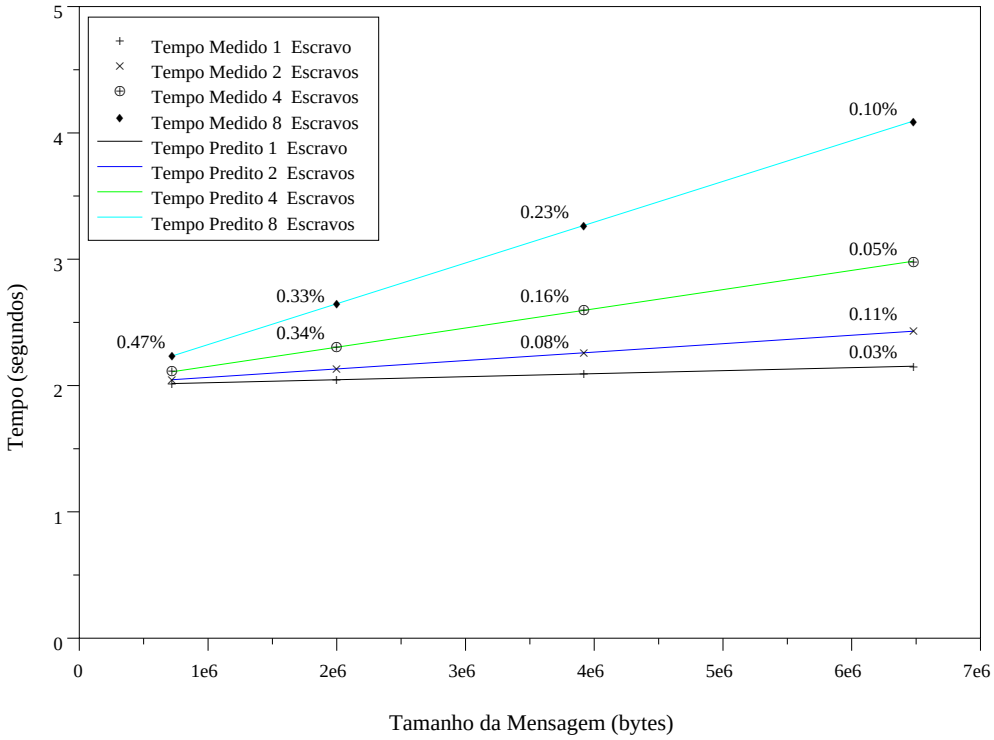

Figura 4.12: Tempos medidos e preditos para 1, 2, 4 e 8 escravos (S1).

ele enviar os dados para o segundo escravo. Se isso acontece, o mestre deverá esperar até que o *buffer* seja liberado para que uma nova mensagem possa ser transmitida, ocasionando um atraso na transmissão. Esse pode ser um dos motivos que implica no aumento do tempo gasto no envio de uma mensagem aos demais escravos. A partir do segundo escravo, o tempo de transmissão não sofreu variações significativas. Baseado nesse comportamento, decidimos elaborar o modelo analítico considerando as particularidades citadas para o envio das mensagens ao primeiro e aos demais escravos.

Para esta primeira situação de sincronismo, o tempo de espera do mestre é dado pelo tempo que o escravo demora para alcançar a primitiva MPI_Recv menos o tempo gasto pelo mestre na inicialização das matrizes A e B. Dessa forma, o modelo gerado está representado através de um polinômio de grau 2, influência do trecho de inicialização das matrizes. Assim, o modelo de predição elaborado para representar a primeira situação de sincronismo é dado pela fórmula:

$$t^{s1}_{envio}(p,n) = 1,9968968 - 1,876 * 10^{-08} * n - 6 * 10^{-18} * n^2 \\ +5,312 * 10^{-04} * p + 4,271 * 10^{-08} * n * p \quad (4.9)$$

A partir do modelo elaborado, podemos predizer o tempo de envio, gasto pelo processo mestre, variando tanto a quantidade de escravos quanto o tamanho da mensagem enviada.

Realizamos algumas predições de desempenho para 3, 4, 6 e 10 escravos com mensagens variando de 720 Kbytes a 18 MBytes. Os erros percentuais do tempos preditos ficaram em média 0,22 % acima dos valores medidos, conforme pode ser visto nas Tabelas 4.1 e 4.2. Os resultados obtidos comprovam a precisão do modelo desenvolvido. Como os erros percentuais são calculados com base nos tempos preditos, os valores negativos indicam que o tempo predito ficou abaixo do tempo medido. Por outro lado, os erros positivos indicam o oposto.

O gráfico ilustrado na Figura 4.13 apresenta os tempos medidos e preditos e alguns erros percentuais para a primeira situação de sincronismo.

	3 Escravos			4 Escravos		
n (bytes)	Predito	Medido	Erro %	Predito	Medido	Erro %
720.000	2,0797	2,0757	0,19	2,1195	2,1089	0,5
2.000.000	2,2206	2,2117	0,4	2,13116	2,3036	0,34
3.920.000	2,424	2,4149	0,37	2,5998	2,5955	0,16
6.480.000	2,692	2,6852	0,25	2,9862	2,9847	0,05
9.680.000	3,0244	3,02	0,11	3,4712	3,4697	0,04
13.520.000	3,4249	3,4257	- 0,02	4,0572	4,0536	0,08
18.000.000	3,8972	3,9008	- 0,09	4,7344	4,7349	- 0,01

Tabela 4.1: Tempos medidos e preditos para 3 e 4 escravos (em segundos).

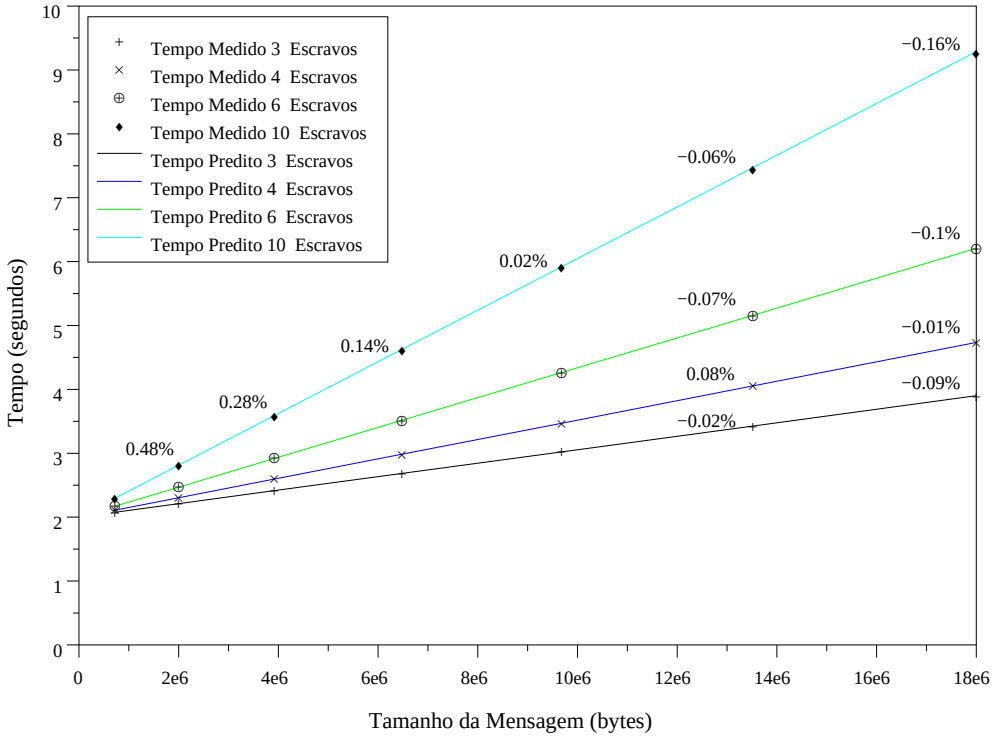

Figura 4.13: Tempos medidos e preditos para 3, 4, 6 e 10 escravos (S1).

	6 Escravos			10 Escravos		
n (bytes)	Predito	Medido	Erro %	Predito	Medido	Erro %
720.000	2,1839	2,1695	0,66	2,3094	2,2946	0,63
2.000.000	2,4831	2,4694	0,54	2,8269	2,8132	0,48
3.920.000	2,9306	2,9187	0,4	3,6004	3,5903	0,28
6.480.000	3,5204	3,5169	0,09	4,6326	4,6258	0,14
9.680.000	4,2625	4,2628	- 0,01	5,9199	5,9182	0,02
13.520.000	5,1559	5,1597	- 0,07	7,4667	7,4712	- 0,06
18.000.000	6,2022	6,2089	- 0,1	9,2709	9,2858	- 0,16

Tabela 4.2: Tempos medidos e preditos para 6 e 10 escravos (em segundos).

4.3.2.2 Situações de Sincronismo 2 e 3 (S2e3).

Nessas situações de sincronismo, o processo mestre não precisa esperar os escravos para iniciar a transmissão das mensagens. Assim, o tempo de envio dos dados não sofre nenhum Δt adicional decorrente de esperas de sincronismo entre o mestre e os escravos, como acontecia na situação apresentada na seção 4.3.2.1. Assim, podemos dizer que o tempo gasto pelo processo mestre para enviar as mensagens aos escravos é dado, basicamente, pelo número de iterações do laço de repetição vezes o tempo gasto pela primitiva MPI_Send. Comparações e incrementos efetuados em cada iteração do laço de repetição também podem aumentar o tempo de envio das mensagens, expressado através da seguinte fórmula:

$$t_{envio}^{s2e3}(p,n) = p * t_{send}(n) + \eta(p,n) \tag{4.10}$$

onde $\eta_i(p,n)$ caracteriza algum *overhead* adicional, p é o número de receptores (que também é a quantidade de iterações do laço de repetição), n é o tamanho da mensagem em bytes e $t_{send}(n)$ é o modelo analítico da primitiva MPI_Send.

Conforme apresentamos em [**?**], as primitivas *send* e *receive* podem ser modeladas através de um polinômio do primeiro grau. Assim, a partir dos resultados obtidos durante os testes experimentais e baseado no modelo teórico proposto anteriormente, utilizamos o método de ajuste de curvas descrito na seção 4.3 e desenvolvemos o seguinte modelo analítico para representar o comportamento do processo mestre:

$$\begin{aligned} t_{envio}^{s2e3}(p,n) = -0,0031032 - 0,018 * 10^{-08} * n + 0,0005312 * p \\ +4,271 * 10^{-08} * n * p \end{aligned} \tag{4.11}$$

onde *n* é o tamanho da mensagem em bytes e *p* a quantidade de escravos. Podemos verificar agora que, sem influência do tempo de inicialização das matrizes, o polinômio gerado é do primeiro grau, conforme esperado.

No gráfico ilustrado na Figura 4.14 expressamos os valores medidos e preditos e algumas comparações ilustradas através de erros percentuais.

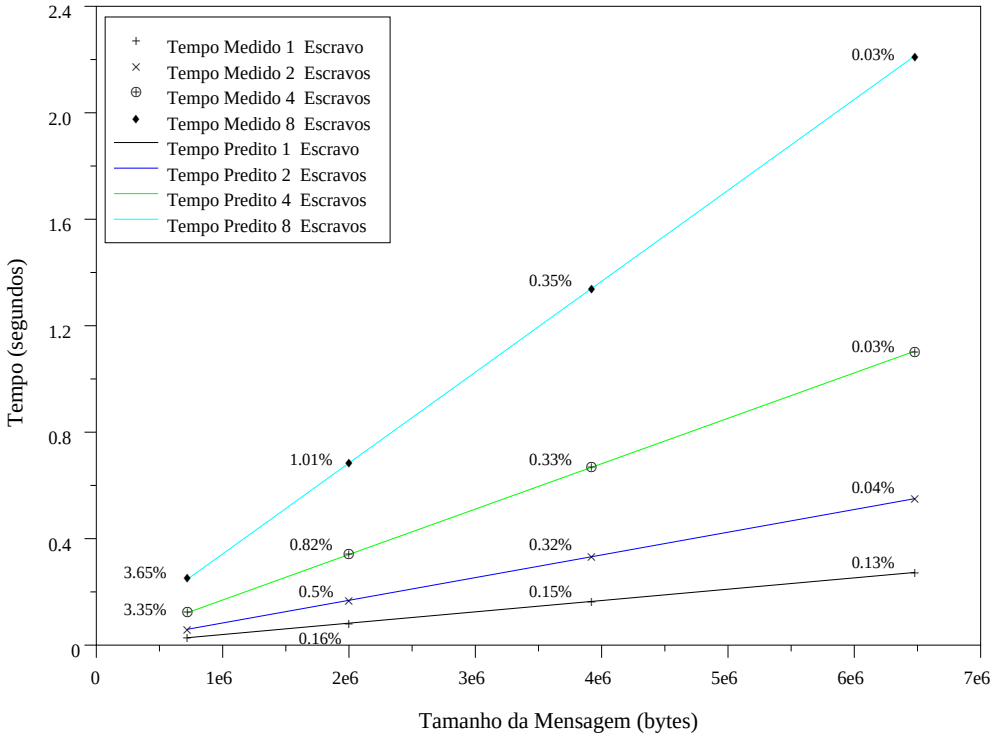

Figura 4.14: Tempos medidos e preditos para 1, 2, 4 e 8 escravos (S2e3).

As predições de desempenho para outras quantidades de escravos e tamanhos de mensagens podem ser vistas no gráfico apresentado na Figura 4.15. Conforme podemos verificar, os erros percentuais ficaram em média 0,74 % acima dos tempos medidos. Assim, podemos afirmar que o modelo desenvolvido para as situações 2 e 3 também mostrou-se preciso. Os resultados completos dos testes experimentais e da atividade de predição podem ser vistos nas Tabelas 4.3 e 4.4.

4.4 Considerações Finais

Neste capítulo, apresentamos algumas extensões sobre a metodologia inicial desenvolvida por [Li01a]. Dentre essas extensões, podemos citar: a proposta de um novo modelo gráfico denominado DP*Graph$^+$, a modelagem de estruturas de repetição compostas por primitivas de comunicação MPI e o desenvolvimento de modelos para aplicações do tipo mestre/escravo.

Conforme verificamos na seção 4.2, o DP*Graph$^+$ permite uma análise mais de-

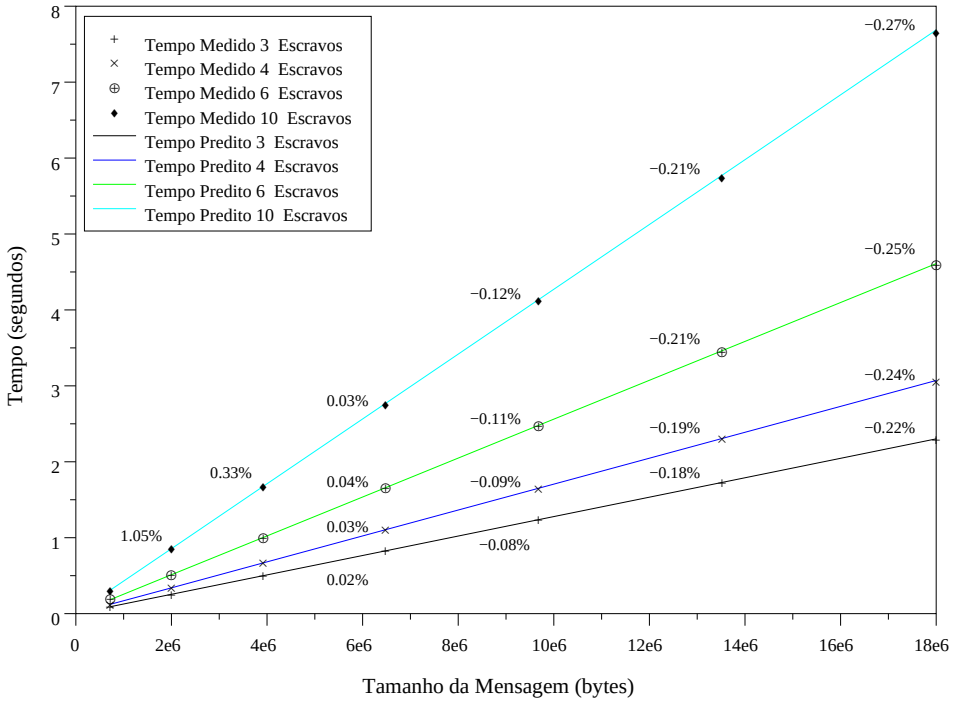

Figura 4.15: Tempos medidos e preditos para 3, 4, 6 e 10 escravos (S2e3).

	3 Escravos			4 Escravos		
n (bytes)	Predito	Medido	Erro %	Predito	Medido	Erro %
720.000	0,0935	0,0906	3,18	0,1261	0,1218	3,35
2.000.000	0,2561	0,2543	0,69	0,3431	0,3403	0,82
3.920.000	0,5016	0,5	0,32	0,6702	0,668	0,33
6.480.000	0,8278	0,8276	0,02	1,1052	1,1048	0,03
9.680.000	1,2359	1,2370	- 0,08	1,6493	1,651	- 0,09
13.520.000	1,7252	1,7283	- 0,18	2,3019	2,3063	- 0,19
18.000.000	2,2964	2,3015	- 0,22	3,0634	3,07	- 0,24

Tabela 4.3: Tempos medidos e preditos para 3 e 4 escravos (em segundos).

	6 Escravos			10 Escravos		
n (bytes)	Predito	Medido	Erro %	Predito	Medido	Erro %
720.000	0,1911	0,1844	3,51	0,3214	0,3095	3,69
2.000.000	0,5172	0,5122	0,96	0,8652	0,8560	1,05
3.920.000	1,007	1,003	0,31	1,6814	1,6757	0,33
6.480.000	1,6601	1,6594	0,04	2,7696	2,7686	0,03
9.680.000	2,4761	2,4789	- 0,11	4,1297	4,1347	- 0,12
13.520.000	3,4547	3,4622	- 0,21	5,7616	5,7741	- 0,21
18.000.000	4,5977	4,6095	- 0,25	7,6658	7,6867	- 0,27

Tabela 4.4: Tempos medidos e preditos para 6 e 10 escravos (em segundos).

talhada sobre o código da aplicação. Nesse novo modelo, inserimos representações gráficas para laços de repetição, diferenciamos os modos de transmissão bloqueante e não-bloqueante e criamos representações para as primitivas de comunicação coletiva. Além do DP*Graph^{+}, desenvolvemos uma modelagem sobre estruturas de repetições formadas por primitivas de comunicação. Nosso objetivo foi propor um modelo simplificado, porém capaz de representar com boa precisão o comportamento dessas estruturas.

Uma outra extensão sobre a metodologia original [Li01a] diz respeito à modelagem de aplicações mestre/escravo. Na metodologia proposta por Li, apenas o tamanho dos dados de entrada podem ser variados durante as atividades de predição de desempenho. Como isso restringe a classe de programas que podem ser modelados, resolvemos desenvolver modelos analíticos capazes de representar o comportamento de aplicações variando não somente o tamanho dos dados de entrada mas também a quantidade de escravos envolvidos na computação do problema. Conforme verificamos na seção 4.3.2, em programas do tipo mestre/escravo existem algumas configurações possíveis relacionadas com as situações de sincronismo entre o mestre e os escravos. Dependendo da situação, verificamos que o modelo analítico sofre modificações para atender as questões relacionadas com o tempo despendido no sincronismo estabelecido. Os modelos desenvolvidos foram testados e, conforme mostramos, os erros percentuais ficaram muito pequenos. No próximo capítulo apresentamos a modelagem e a predição de desempenho de duas versões diferentes de um programa de multiplicação de matrizes, aplicando os modelos elaborados e testados neste capítulo.

Capítulo 5

MODELAGEM DE PROGRAMAS MPI

5.1 Introdução

Os programas paralelos MPI são difíceis de serem modelados. A existência de primitivas de comunicação complica o desenvolvimento dos modelos analíticos. A preocupação com a precisão pode ocasionar o desenvolvimento de modelos complexos e de pouca praticidade. Quando utilizamos a modelagem analítica, é comum desenvolvermos modelos repletos de grandezas (variáveis e constantes) que as vezes são difíceis de serem obtidas. Portanto, é importante perceber quais fatores devem ser considerados durante o desenvolvimento dos modelos para que os mesmos sejam precisos e, ao mesmo tempo, de fácil compreensão e aplicação.

A modelagem de programas paralelos MPI oferece uma dificuldade a mais, relacionada com a caracterização das primitivas de comunicação. Em programas paralelos que utilizam passagem de mensagem podem existir alguns tipos de sincronismos entre os processos envolvidos na computação. O tempo gasto nesse sincronismo também deve ser mensurado durante o desenvolvimento dos modelos de predição. Portanto, é fundamental conhecer os detalhes do programa que está sendo modelado, pois um simples tempo de sincronização pode comprometer toda a precisão do modelo desenvolvido. Algumas vezes, antes de chegarmos ao modelo final, utilizado nas predições, podemos desenvolver e testar vários modelos iniciais.

Durante o desenvolvimento de uma atividade de análise e predição de desempenho, a grande quantidade de dados gerados nos testes experimentais dificulta o trabalho de análise e seleção dos mesmos. Essa dificuldade está relacionada com a complexidade em selecionar os valores que realmente nos interessam. Por esse motivo, procuramos o auxílio de uma ferramenta capaz de facilitar todo o processo de análise e predição de desempenho, automatizando etapas de nossas atividades. A ferramenta utilizada em nosso trabalho foi o Scilab [Scilab].

Neste capítulo, além de apresentar o Scilab e caracterizar o nosso ambiente de teste, também mostramos a modelagem de duas versões de um programa de multiplicação de matrizes. A primeira delas utiliza somente um mestre e um escravo na organização do programa, e a única variável existente no modelo de predição é a dimensão (n) das matrizes A e B. Já a segunda versão é caracterizada por utilizar diferentes quantidades de escravos para multiplicar as matrizes. Portanto, além da variável relacionada com as dimensões das matrizes temos também a quantidade de escravos (p) utilizada na solução do problema. Conforme apresentamos, em ambos os casos, apesar de simplificado, os modelos analíticos apresentam baixos erros percentuais, satisfazendo nossas expectativas iniciais.

5.2 Scilab

As atividades de análise e predição de desempenho apresentam várias dificuldades relacionadas com a complexidade das tarefas envolvidas no desenvolvimento dos modelos de predição. A manipulação de grandes quantidades de dados, oriundos de testes experimentais, é um problema a ser considerado. Realizar uma seleção desses dados e descobrir o comportamento da aplicação que está sendo analisada, demanda tempo e trabalho. Portanto, é fundamental dispormos de ferramentas de *software* capaz de auxiliar todo o processo de análise e predição de desempenho. Com o uso dessas ferramentas, várias etapas do processo podem ser automatizadas, minimizando o nosso trabalho.

O Scilab [Scilab] é uma ferramenta capaz de manipular grandes quantidades de dados e dar suporte as atividades de análise e predição de desempenho. Através de uma variedade de funções disponibilizadas, o Scilab oferece facilidades para trabalharmos

com matrizes e listas de dados, funções matemáticas, sistemas lineares e desenvolvimento de gráficos para apresentação de resultados. Além disso, um ambiente de programação é disponibilizado, de modo que podemos implementar funções e criar novas bibliotecas de acordo com nossas necessidades. Isso é fundamental para alcançarmos um dos nossos objetivos que consiste em automatizar o maior número de tarefas possível.

Além do que já comentamos, o Scilab disponibiliza:

- tipos de dados variados com uma sintaxe natural e fácil de ser usada;
- um conjunto de primitivas que podem ser empregadas em uma variedade de cálculos matemáticos;
- um ambiente de programação onde novas primitivas e funções podem ser construídas;
- bibliotecas que podem ser desenvolvidas através de *toolboxes*, contendo funções específicas de algumas aplicações, como por exemplo: controle linear, processamento de sinal, análise de redes de interconexão, entre outras.

Assim, com o auxílio do Scilab, implementamos um conjunto de funções para automatizar várias etapas do processo de modelagem e predição de desempenho. Essas tarefas são apresentadas na próxima seção.

5.3 Tarefas Automatizadas

Na modelagem analítica, antes de realizarmos qualquer predição de desempenho, precisamos cumprir várias etapas iniciais que dizem respeito à coleta e seleção de dados experimentais. Normalmente, a quantidade de dados gerado nos testes experimentais é grande, o que torna a seleção e análise dos mesmos um trabalho árduo e demorado. Portanto, desenvolver rotinas para automatizar essa seleção é importante para facilitar o trabalho e diminuir o tempo gasto nessas tarefas iniciais. Existem algumas estratégias que podem ser adotadas durante a seleção dos dados [Jain91]. No entanto, a escolha de quais dados devem ser selecionados pode depender da aplicação que está sendo modelada e do comportamento apresentado durante os testes. Assim, nem sempre podemos utilizar uma mesma estratégia de seleção para todos os tipos de aplicações.

Na modelagem analítica, depois de selecionar os dados, desenvolvemos um modelo matemático capaz de representar o comportamento da aplicação. Em nosso trabalho, utilizamos o método dos mínimos quadrados, descrito na seção 4.3, para gerar uma equação matemática, denominada modelo analítico. A partir do modelo elaborado, podemos estimar o tempo de execução do programa para diferentes situações, variando os parâmetros que influenciam no tempo de execução, como por exemplo a quantidade dos dados de entrada do programa.

Conforme discutimos, existem muitas etapas que compõe uma atividade de análise e predição de desempenho. Assim, com o auxílio do Scilab, automatizamos as seguintes tarefas:

- seleção dos dados gerados nos testes experimentais;
- cálculo da média dos tempos de execução;
- geração dos modelos de predição a partir dos dados selecionados;
- cálculo dos tempos de predição e erros percentuais em relação aos tempos medidos;
- geração de gráficos com os tempos preditos e medidos.

No Apêndice B, uma explicação mais detalhada sobre cada uma das funções implementadas pode ser encontrada.

5.4 Instrumentação do Código

Além das tarefas descritas anteriormente, ainda existe uma outra atividade, relacionada com a instrumentação do código do programa com monitores de tempo, que deve ser automatizada. Nessa instrumentação, devemos inserir monitores para medir o tempo de execução associado a cada um dos trechos ilustrados no DP*Graph^{+}. Isso é preciso, pois cada parte do programa pode apresentar um comportamento diferente, exigindo assim um modelo análitico distinto para representar cada um dos trechos identificados.

Nesta seção, apresentamos um possível algoritmo para instrumentar o código dos programas MPI. Nossa intenção é apenas mostrar um caminho para que, em trabalhos

futuros, alguma ferramenta possa ser implementada para automatizar essa atividade. Os passos que compõem esse algoritmo são:

1. Analisar o fluxo de execução do programa, reconhecendo os trechos de código associados a cada processo. Através dessa análise inicial, esperamos identificar os caminhos de execução seguidos pelos processos;
2. Encontrar as primitivas de comunicação ou sincronização MPI;
3. Inserir monitores de tempo antes e após as primitivas encontradas;
4. Inserir monitores de tempo entre os principais trechos de código seqüenciais (estrutura de repetição e instruções com custo computacional significativo). As intruções mais simples, como atribuições, influenciam pouco no desempenho das aplicações e podem ser computadas juntamente com os trechos de maior custo computacional, localizados imediatamente antes ou após.

Desenvolver uma ferramenta para fazer a instrumentação do código pode não ser um trabalho simples. No entanto, seria interessante que essa atividade também fosse automatizada. O conjunto das funções desenvolvidas poderia constituir uma ferramenta para análise e predição de desempenho de programas paralelos MPI.

5.5 Caracterizando o Ambiente de Teste

Para a realização dos testes experimentais utilizamos um *cluster* de PCs do Laboratório de Arquitetura e Software Básico (LASB), do Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo (USP). Esse *cluster* possui as seguintes características:

- 16 nós homogêneos de processamento. Cada nó possui um processador INTEL Celeron de 433 MHz, com 128 MB de memória SDRAM (66 MHz), 32 KB de cache L1, 128 KB de cache L2. Além disso, eles são equipados com uma placa de rede INTEL Ether-Express Pro 100 Mbits;
- sistema operacional Linux Red Hat 6.2;

- rede de interconexão Fast-Ethernet;
- switch 3COM SuperStack3300;
- implementação do padrão MPI: LAM-MPI 6.4.

5.6 Multiplicação de Matrizes

A multiplicação de matrizes é uma aplicação clássica e muito utilizada em atividades de análise e predição de desempenho. Nosso objetivo, além de exemplificar como utilizar a metodologia de modelagem e predição de desempenho de programas paralelos MPI proposta em [Li01a], é aplicar e validar todos os modelos desenvolvidos no capítulo anterior.

Existem diversas maneiras de elaborar um programa para multiplicar duas matrizes de forma distribuída. Quando utilizamos um paradigma de passagem de mensagem, como por exemplo o MPI, as primeiras distinções podem ser feitas através das primitivas de comunicações utilizadas para distribuir as matrizes que serão multiplicadas. Conforme veremos nas seções 5.6.1 e 5.6.2, podemos realizar as comunicações com as seguintes primitivas: MPI_Send, MPI_Recv, MPI_Bcast, entre outras. Dependendo das primitivas utilizadas teremos um comportamento distinto para a aplicação. Uma outra diferença está na forma com que as matrizes podem ser distribuídas para os escravos. Nessa distribuição, podemos enviar linhas ou colunas das matrizes, mudando a forma de realizar a multiplicação. Além disso, podemos permutar a ordem dos laços de repetição envolvidos no trecho específico da multiplicação. Mudando a seqüência dos laços de repetição, podemos alterar a maneira de acesso às matrizes armazenadas na memória. Como sabemos, a linguagem C armazena as matrizes por linha na memória do computador. Assim, o acesso por linha durante a execução do programa pode melhorar o desempenho do mesmo.

Em nosso trabalho não estamos muito interessados com o desempenho do programa e sim com a modelagem do seu comportamento para futuras predições. Para validar os modelos desenvolvidos no capítulo anterior, apresentamos nas próximas seções a modelagem de duas versões de um programa de multiplicação de matrizes.

5.6.1 Versão 1

Nessa primeira versão do programa de multiplicação de matrizes (ver código no Apêndice A), estamos interessados em analisar e modelar o comportamento das primitivas de comunicação ponto-a-ponto dentro de uma aplicação. Por este motivo, utilizamos somentes as primitivas de comunicação MPI_Send e MPI_Recv para realizar as trocas de mensagens.

Segundo a metodologia proposta por [Li01a], inicialmente devemos elaborar o modelo gráfico do programa que será modelado. Este modelo permite ao analista abstrair as características do programa e visualizar como o mesmo está organizado. Para tanto, iremos usar a nova simbologia proposta na seção 4.2. Assim, o DP*Graph^{+} da Figura 5.1, representa a primeira versão do programa de multiplicação de matrizes.

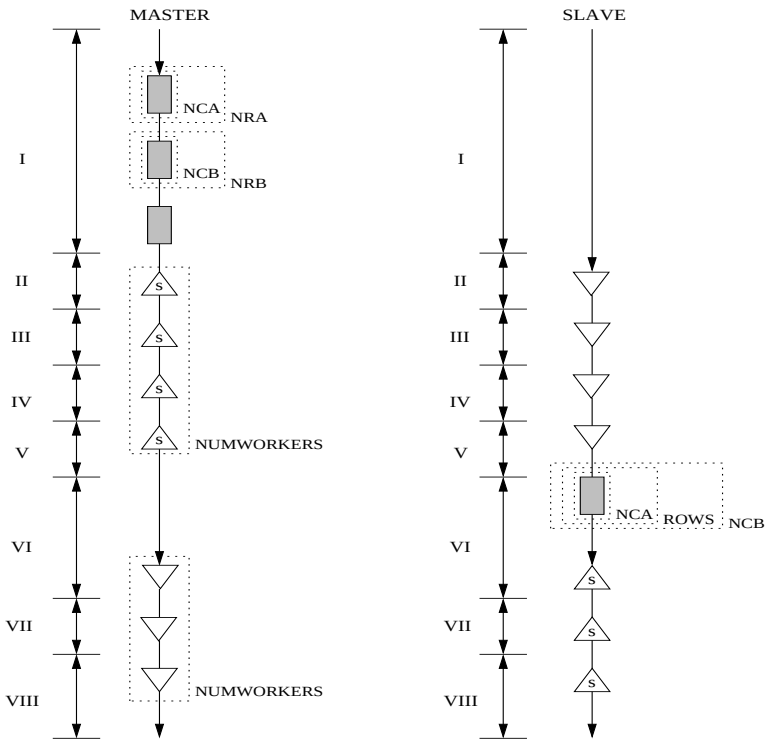

Figura 5.1: DP*Graph^{+} da versão 1 do programa de multiplicação de matrizes.

Através do DP*Graph^{+}, podemos verificar que todas as primitivas de comunicação utilizadas na implementação do programa são bloqueantes e que o modo de transmissão adotado para a primitiva MPI_Send é o *standard*. Além disso, existem duas estru-

turas de repetição no processo mestre (trechos representados pelos retângulos pontilhados) formadas por um conjunto de primitivas MPI_Send e MPI_Recv. Na primeira delas, o mestre envia as matrizes A e B, enquanto que na segunda estrutura o mestre recebe a matriz produto C calculada pelo escravo.

Para cada um dos trechos identificados no DP*Graph^{+}, podemos elaborar um modelo analítico específico. Com isso, é possível realizar predições de desempenho para cada um dos trechos individuais. No entanto, a predição geralmente é feita sobre o tempo total de execução da aplicação que está sendo modelada e não apenas sobre partes dela. Portanto, devemos encontrar um modelo final, composto pelos modelos parciais, capaz de estimar o tempo total de execução.

Depois de elaborado o modelo gráfico, devemos instrumentar o código do programa, inserindo instruções para medir os tempos de execução de cada um dos trechos identificados no DP*Graph^{+}. Feito isso, submetemos o programa aos testes experimentais. No caso dessa primeira versão do programa de multiplicação de matrizes, o único parâmetro variável é a dimensão das matrizes (n), pois realizamos testes com apenas um escravo e um mestre.

A fim de selecionar os dados obtidos durante os testes experimentais, desconsideramos todos os resultados que ficaram 30 % acima do menor valor obtido. A média dos tempos de execuções selecionados foram utilizadas na construção dos modelos analíticos, através do método de ajuste de curvas dos mínimos quadrados.

O tempo total de execução do programa de multiplicação de matrizes é dado pelo tempo do processo mestre, pois é ele quem finaliza a execução do programa. Essa análise é feita conforme mostrado a seguir:

$$t_{exec} = max\,(\overbrace{t_I + t_{II} + ... + t_{VIII}}^{mestre}, \overbrace{t_I + t_{II} + ... + t_{VIII}}^{escravo}) \tag{5.1}$$

Assim, a princípio podemos dizer que o modelo matemático que representa o tempo total de execução é dado pela somatória dos modelos parciais que compõe o processo mestre. No entanto, modelos de outros trechos associados ao escravo, que também influenciam no tempo de execução do mestre, devem ser considerados no desenvolvimento do modelo final. Por exemplo, depois de enviar as matrizes A e B ao processo escravo, o mestre fica bloqueado esperando o escravo realizar a multiplicação

e devolver a matriz produto C. Portanto, essa espera, referente ao tempo da multiplicação, deve ser computada no tempo de execução total do programa.

Os trechos de código que representam as computações locais devem ser modeladas com as tradicionais técnicas de modelagem de programas seqüenciais, baseadas na complexidade do código. Por exemplo, o trecho I do mestre, representa a inicialização das matrizes A e B. Conforme sabemos, sua complexidade é O (n^2). Portanto, um polinômio de grau 2 deve ser adequado para representar seu comportamento. Em [Laine02], realizamos um estudo desse programa e construímos várias equações buscando encontrar uma que melhor representasse o comportamento do trecho de inicialização das matrizes. O modelo que se mostrou mais adequado para representar o trecho de inicialização das matrizes é este apresentado a seguir:

$$t_{inic}(n) = 2,12 * 10^{-7} * n^2 - 1,27 * 10^{-6} * n - 0,00043 \quad (5.2)$$

onde *n* representa a dimensão das matrizes utilizadas.

A validade deste modelo foi provada através de testes experimentais que comprovaram a precisão dos tempos preditos. Os resultados obtidos em t_{inic} apresentaram erros percentuais inferior a 1%, conforme ilustrado na Tabela 5.1.

Dimensão (n)	100	200	300	400	500
Tempo Medido	0,001558	0,007773	0,018397	0,032864	0,052005
Tempo Predito	0,001565	0,0078	0,018277	0,032997	0,051958
Erro %	0,42	0,34	- 0,65	0,40	- 0,09

Tabela 5.1: Tempos medidos e preditos para a inicialização das matrizes A e B (em segundos).

Em seguida, modelamos o comportamento do trechos de comunicações, calculando o tempo despendido nas primitivas *send* e *receive*. Segundo a metodologia proposta em [Li01a], uma comunicação ponto-a-ponto pode ser decomposta nos seguintes componentes:

- $t_e(n)$: tempo gasto para transferir *n* elementos de dados da memória para a interface de rede (1);

- $t_t(n)$: tempo gasto para transferir *n* elementos de dados pela rede de interconexão (2);
- $t_r(n)$: tempo gasto para transferir *n* elementos de dados da interface de rede para a memória (3).

Os componentes relacionados anteriormente são ilustrados na Figura 5.2.

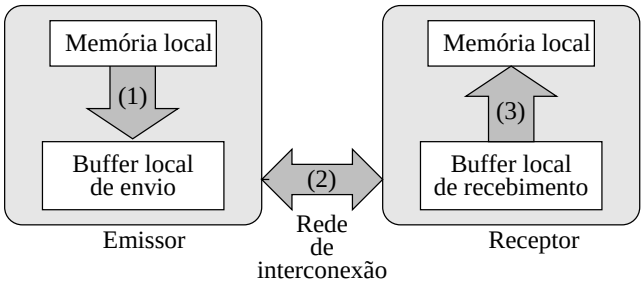

Figura 5.2: Componentes do tempo de comunicação.

Depois de um estudo inicial sobre a comunicação entre processos e baseado nas particularidades de cada componente, podemos dizer que os tempos de comunicações gastos em $t_e(n)$, $t_t(n)$ e $t_r(n)$ são representados através das seguintes equações:

$$t_e(n) = n * c_1 \tag{5.3}$$

$$t_t(n) = k_b * n + k_l \tag{5.4}$$

$$t_r(n) = n * c_2 \tag{5.5}$$

onde:

- *n* é a quantidade de elementos transmitidos na mensagem;
- c_1 e c_2 são constantes;
- k_b representa a largura de banda;
- k_l representa a latência da rede.

Contudo, a fim de simplificar o modelo analítico, consideramos apenas dois dos principais fatores que influenciam o desempenho, sendo eles: a largura de banda e a latência da rede. Com esses componentes, podemos dizer que o tempo de comunicação é expressado através da seguinte fórmula:

$$t_c(n) = t_e(n) + t_t(n) + t_r(n)$$

$$t_c(n) = n * c_1 + k_b * n + k_l + n * c_2$$

$$t_c(n) = (c_1 + k_b + c_2) * n + k_l$$

$$t_c(n) = c * n + k_l \tag{5.6}$$

onde *c* é uma constante.

Para a modelagem dos tempos de comunicação, utilizamos, durante os testes experimentais, o *benchmark* MPBench [Mucci98]. Com os resultados obtidos através da execução desse *benchmark*, construímos os modelos analíticos para as primitivas *send* e *receive* [Laine02]. Através dos dados gerados pelo MPBench, concluímos que o modelo final seria mais preciso se elaborássemos funções distintas para diferentes intervalos de *n*. Assim, desenvolvemos uma modelagem por intervalo e caracterizamos as comunicações da seguinte forma:

$$t_{send}(n) = 0,0879182 * n - 82,0523 \tag{5.7}$$

$$t_{recv}(n) = 0,0852 * n + 9,706 \tag{5.8}$$

para $6824 \leq n \leq 65536$.

$$t_{send}(n) = 0,0849931 * n - 2065,49 \tag{5.9}$$

$$t_{recv}(n) = 0,0849968 * n + 2146,19 \quad (5.10)$$

para $65536 \leq n \leq 5592404$.

Baseado nos modelos parciais elaborados para os trechos identificados no modelo gráfico, desenvolvemos o seguinte modelo final para representar o tempo de execução total do programa de multiplicação de matrizes:

$$t_{exec}(n) = 3,14 * 10^{-7} * n^3 + 2,65 * 10^{-5} * n^2 - 9,39 * 10^{-3} * n + 0,64 \quad (5.11)$$

A partir do DP*Graph$^+$, ilustrado na Figura 4.7, podemos observar que no trecho VI do mestre existe uma dependência em relação à execução do escravo. Para executar o *receive* do trecho VI, o mestre deve esperar o escravo multiplicar as matrizes A e B e executar o próximo *send*. Dessa forma, o tempo gasto no *receive* do trecho VI do mestre deve incluir esse tempo de espera. Por este motivo, o polinômio encontrado para t_{exec} é de grau três, pois inclui o modelo desenvolvido para representar o trecho das multiplicação de matrizes (trecho VI do escravo).

Os resultados das atividades de predição podem ser vistos na Tabela 5.2. Nessa tabela, temos os valores preditos por t_{exec} e os medidos durante os testes experimentais, além dos erros percentuais.

Conforme podemos observar na Tabela 5.2, os resultados das predições foram bons. Na maioria dos casos testados, os erros percentuais ficaram abaixo dos 3 %. Portanto, embora o modelo desenvolvido seja simples, ele se mostrou preciso durante a predição do tempo de execução para as dimensões analisadas. O gráfico da Figura 5.3 mostra a curva da predição, elaborada a partir do modelo, juntamente com os valores medidos e alguns erros percentuais que já foram expressados na Tabela 5.2.

Dimensão (n)	100	200	250	300
Tempo Medido	0,316625	2,785438	5,326959	9,47386
Tempo Predito	0,307599	2,551355	5,279105	9,418468
Erro %	- 2,85	- 8,40	- 0,90	- 0,58

Dimensão (n)	400	450	500
Tempo Medido	23,52449	32,80233	45,87755
Tempo Predito	22,95936	32,86457	45,20871
Erro %	- 2,42	0,19	- 1,46

Tabela 5.2: Comparação entre os tempos medidos e preditos (em segundos).

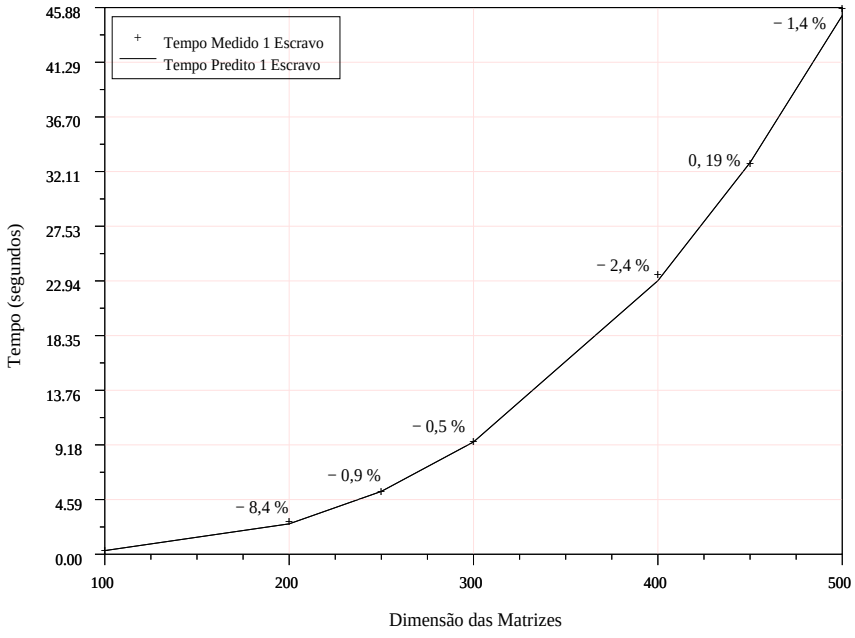

Figura 5.3: Tempos medidos e preditos para a versão 1 da multiplicação de matrizes.

5.6.2 Versão 2

Na última seção, modelamos e estimamos o desempenho do programa de multiplicação de matrizes ilustrado na Figura 4.7. Na primeira versão, utilizamos as seguintes primitivas de comunicação MPI para distribuir as matrizes A e B: *MPI_Send* e *MPI_Recv*. Conforme comentamos, durante os testes experimentais consideramos a presença de apenas um mestre e um escravo. Por esse motivo, todos os modelos analíticos foram gerados em função da dimensão das matrizes (*n*), e não permitiam efetuar predições de desempenho variando a quantidade de escravos (*p*). Assim, resolvemos criar uma nova versão para o programa de multiplicação de matrizes, permitindo que a matriz resultante C fosse calculada por vários escravos. Assim, foi possível aplicar os modelos desenvolvidos na seção 4.3.2 para testar a precisão dos mesmos durante a modelagem de uma aplicação real um pouco mais complexa.

O código dessa versão pode ser visto no Apêndice A. Nessa nova versão, representada no DP*Graph$^+$ da Figura 5.4, incluímos também a primitiva de comunicação coletiva *MPI_Bcast*, utilizada para distribuir a matriz B a todos os escravos participantes da multiplicação. Cada escravo recebe uma parte distinta da matriz A, enviada pelo mestre através de uma primitiva MPI_Send. Para isso, a dimensão da matriz A é dividida pelo número de escravos, se a divisão for exata, o quociente indica quantas linhas da matriz devem ser enviadas a cada um dos escravos. Caso a divisão não seja exata, as linhas que sobram devem ser redistribuídas entre os escravos. Com isso, alguns escravos recebem mais elementos para multiplicar do que outros. Quando utilizamos dimensões de matrizes não divisíveis pelo número de escravos, notamos que o tempo gasto no trecho de multiplicação é diferente para alguns escravos, justamente porque alguns recebem uma maior quantidade de dados para multiplicar. Isso dificulta o processo de modelagem e predição de desempenho e deve ser observado durante as atividades.

Embora essa nova versão apresente algumas diferenças em relação à anterior, a complexidade do programa de multiplicação de matrizes continua sendo $O(n^3)$. No entanto, a presença do *broadcast* acaba dificultando a análise do programa à medida que novos *overheads*, resultantes do tempo de espera entre a recepção do *broadcast* do primeiro ao último escravo, são adicionados ao tempo de execução do processo mestre. Esse *overhead* depende da característica do algoritmo utilizado para implementar essa primitiva coletiva. O LAM-MPI utiliza dois algoritmos diferentes para imple-

mentar o MPI_Bcast. Até 3 escravos, a implementação utiliza um algoritmo linear; a partir do quarto escravo é utilizado um algoritmo baseado em hipercubo, de complexidade $O(logn)$. Dependendo da quantidade de processos utilizados na operação de *broadcast*, temos uma dimensão diferente para o hipercubo. Essas particularidades podem ser observadas no código fonte da implementação LAM-MPI. Um estudo detalhado das primitivas de comunicação ponto-a-ponto e coletivas pode ser visto em [Oliveira02b].

Uma outra particularidade desse programa está relacionada com o tempo gasto no trecho específico da multiplicação das matrizes A e B (trecho VI do escravo). Através dos testes experimentais notamos que o tempo de multiplicação é inversamente proporcional a quantidade de escravos utilizados, atingindo quase o *speedup* ideal. Assim, ao dobrarmos a quantidade de escravos, o tempo de multiplicação é reduzido pela metade, ou bem próximo a isso.

Através do DP*Graph$^+$ da Figura 5.4, podemos observar cada um dos trechos identificados no código fonte do programa de multiplicação de matrizes. Analisando a linha de execução do processo mestre, podemos verificar que existe uma parte inicial constituída por um código seqüencial, responsável pela inicialização das matrizes A e B (trecho I). Em seguida, há um laço formado por primitivas MPI_Send responsável por enviar a matriz A, uma operação coletiva MPI_Bcast (trecho V) para enviar a matriz B e, finalmente, um laço composto por primitivas MPI_Recv para receber a matriz C. Por outro lado, todos os escravos podem ser representados conforme mostra o DP*Graph$^+$. Depois de receber as matrizes A e B (trechos IV e V), através das primitivas MPI_Recv e MPI-Bcast, os escravos realizam a multiplicação das matrizes (trecho VI). Terminado a multiplicação das partes que cabem à cada escravo, a matriz parcial C é enviada ao mestre através da primitiva MPI_Send (trecho IX).

O modelo analítico que caracteriza o comportamento do programa de multiplicação de matrizes, foi desenvolvido a partir da modelagem de cada um dos trechos identificados e destacados no DP*Graph$^+$. Assim, o modelo analítico de predição do programa é dado por:

$$t_{exec}(n,p) = \Sigma_{i=I}^{VIII} f_i^M(n,p) + \alpha(n,p) \tag{5.12}$$

onde:

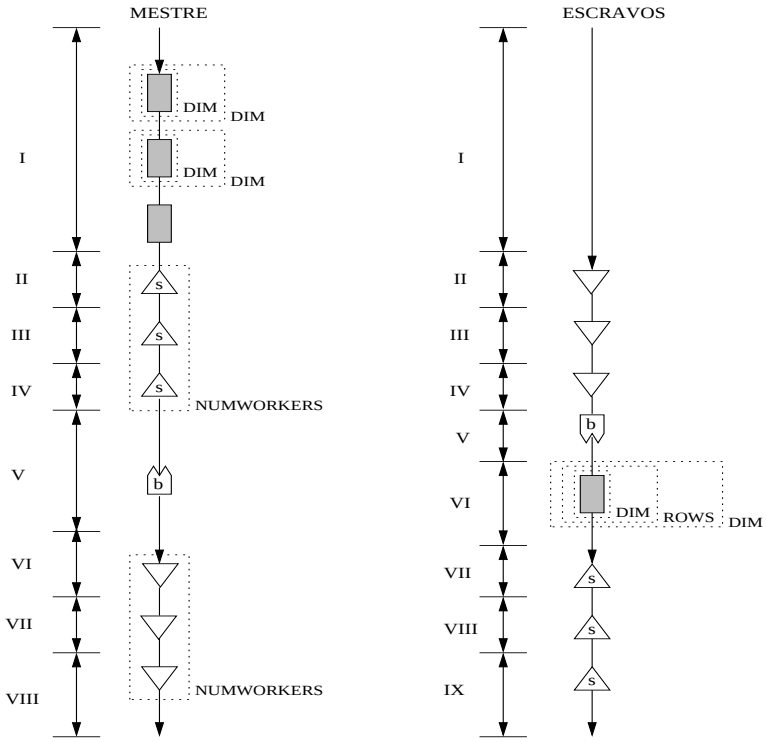

Figura 5.4: DP*Graph^{+} da versão 2 do programa de multiplicação de matrizes.

- $f_i^M(n)$ representa os modelos análiticos gerados para cada trecho identificado;
- α caracteriza algum *overhead* adicional.

Para gerar os modelos parciais, utilizamos o método matemático descrito na seção 4.3. Dessa forma, realizamos testes experimentais, considerando algumas configurações para a quantidade de escravos e a dimensão das matrizes e elaboramos os seguintes modelos de predição:

- Tempo de inicialização das matrizes A e B (trecho I do mestre):

Esse modelo descreve o comportamento do trecho onde o processo mestre inicializa as matrizes A e B usando dois laços aninhados. Como a complexidade inerente a esse trecho de código é $O\left(n^2\right)$, sabemos que um polinômio de grau dois é mais adequado para representá-lo. Assim, o seguinte modelo foi elaborado:

$$t_{inic}(n) = -0,0017538171 + 21,757429 * 10^{-09} * n - 3,109 * 10^{-17} * n^2 \qquad (5.13)$$

- Tempo da estrutura de repetição dos trechos II, III e IV (processo mestre):

Nessa estrutura de repetição, o mestre utiliza uma seqüência de MPI_Send para enviar parâmetros de controle e partes distintas da matriz A a cada um dos escravos. O modelo analítico foi elaborado baseado na equação 4.2 e o comportamento linear em n e p é determinado pelo tempo gasto com as operações de comunicação. Portanto, desenvolvemos o modelo apresentado a seguir:

$$t_{for1}(n,p) = p * (-0,001359 + 1,061 * 10^{-07} * \frac{n}{p}) \tag{5.14}$$

- Tempo da operação coletiva *broadcast* (trecho V):

Como já explicamos, o LAM utiliza um algoritmo $log_2 p$ para implementar a operação coletiva *broadcast*. Um hipercubo composto por um grupo de processos é usado para realizar a distribuição dos dados, de modo que a primitiva de comunicação MPI_Bcast apresenta um comportamento logarítmico em p. Em [Oliveira02b] algumas primitivas de comunicação coletiva são analisadas e modeladas, sendo o *broadcast* caracterizado através do seguinte modelo:

$$t_{bcast}(n,p) = (a_0 + a_1 * n) + (a_2 + a_3 * n) + \lfloor (log_2(p)) \rfloor \tag{5.15}$$

Baseado nessa equação, linear no tamanho da mensagem e logarítmica na quantidade de escravos, elaboramos o seguinte modelo para representar o tempo gasto no trecho V dos processos escravos:

$$\begin{aligned} t_{bcast}(n,p) &= -0,0280938 + 1,103 * 10^{-07} * n \\ &+(0,0255270 + 1,019 * 10^{-07} * n) * \lfloor (log_2(p)) \rfloor \end{aligned} \tag{5.16}$$

- Tempo da multiplicação das matrizes A e B (trecho VI do escravo):

Para realizar a multiplicação das matrizes o programa utiliza o clássico algoritmo de complexidade $O(n^3)$, formado por três laços aninhados. Considerando que podemos

variar o número de escravos e a dimensão das matrizes, elaboramos o modelo apresentado a seguir baseado na equação ilustrada em 4.3:

$$t_{mult}(dim,p) = 2,0003729 - \frac{147,94586}{p} + (-0,012814 + \frac{0,901976}{p}) * dim$$
$$+(0,0000227 - \frac{0,0016335}{p}) * dim^2 + (-1,259 * 10^{-08} + \frac{9,716 * 10^{-07}}{p}) * dim^3 \quad (5.17)$$

- Tempo da estrutura de repetição dos trechos VI, VII e VIII (processo mestre):

Através do DP*Graph$^+$, ilustrado na Figura 4.7, podemos verificar que antes do executar a última estrutura de repetição, o mestre deve esperar que os escravos recebam a matriz B através do MPI_Bcast e façam a multiplicação de A por B. Por isso, o tempo gasto pelo mestre nessa estrutura deve incluir esse tempo de espera, decorrência do sincronismo estabelecido entre ele e os escravos. Assim, ao modelarmos esse trecho de código consideramos o tempo gasto com as operações MPI_Recv (trechos VI, VII e VIII) mais o tempo decorrido entre o instante em que o mestre finaliza o *broadcast* até o momento em que o último escravo recebe os dados (tempo de espera). Por causa dessa relação de sincronismo, o modelo apresentado a seguir para essa estrutura de repetição também está em função de $log_2 p$, influência da primitiva MPI_Bcast.

$$t_{for2}(n,p) = p * (-0,0209271 + 1,244 * 10^{-07} * \frac{n}{p}) - 0,0279167$$
$$+1,103 * 10^{-07} * n + (0,0255270 + 1,019 * 10^{-07} * n) * \lfloor (log_2(p)) \rfloor \quad (5.18)$$

Para cada um dos modelos desenvolvidos, n representa a quantidade total de bytes, p indica a quantidade de escravos utilizados e dim simboliza a dimensão das matrizes A e B. O modelo final de predição do tempo de execução da multiplicação de matrizes é dado pela somatória dos modelos apresentados anteriormente.

$$t_{exec}(n,p) = t_{inic} + t_{for1} + t_{bcast} + t_{mult} + t_{for2} \quad (5.19)$$

A partir do modelo elaborado, fizemos predições de desempenho para algumas quantidades de escravos, variando também a dimensão das matrizes utilizadas. O trecho responsável pela multiplicação das matrizes (trecho VI do escravo) é o que mais contribui no tempo total de execução. Portanto, se o modelo de predição para esse trecho não for preciso, erros percentuais significativos podem ser gerados. O gráfico ilustrado na Figura 5.5, apresenta os tempos medidos e preditos para a multiplicação de matrizes, considerando 1, 2, 4, 6, 10 e 16 escravos e as seguintes dimensões para as matrizes A e B: 300, 600, 900 e 1200. Cada elemento dessas matrizes é do tipo *double*, portanto 8 *bytes* cada. O modelo de predição desenvolvido mostrou-se preciso

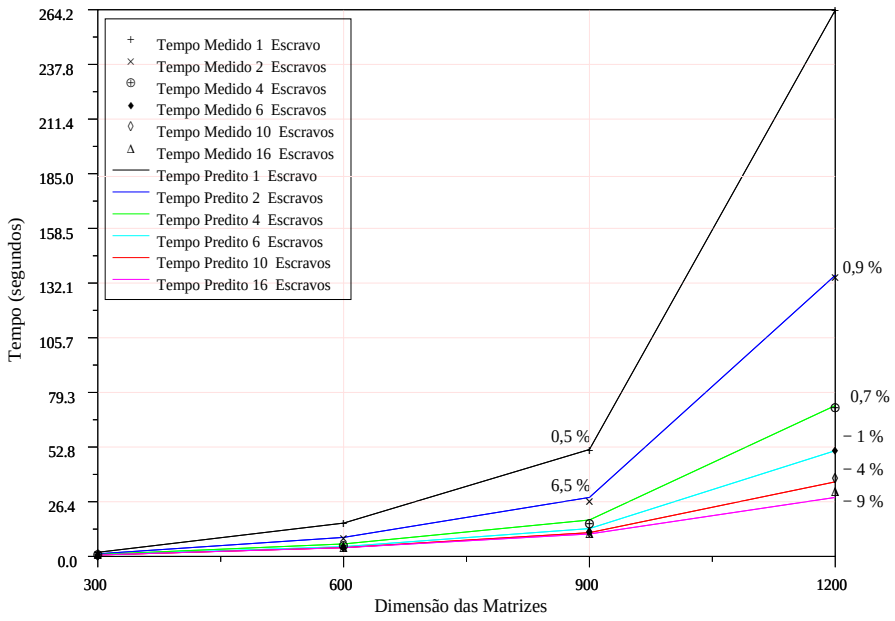

Figura 5.5: Comparação entre os tempos medidos e preditos.

para quase todos os casos considerados. Os valores percentuais, expressados no gráfico da Figura 5.5, indicam o erro percentual do tempo predito em relação ao medido. Conforme já comentamos, os valores negativos indicam que o tempo predito ficou abaixo do medido e valores positivos indicam o contrário. Através das curvas representadas no gráfico, podemos visualizar o crescimento do tempo de execução em função do aumento da dimensão das matrizes A e B para cada quantidade de escravo utilizada. Além disso, verificamos que dobrando a quantidade de escravos para matrizes com

mesma dimensão, o tempo de execução total é reduzido quase pela metade. Assim, podemos dizer que o tempo de execução é inversamente proporcional ao número de escravos existentes.

As tabelas 5.3, 5.4, 5.5, 5.6, 5.7 e 5.8 apresentam os resultados preditos e medidos para o tempo de execução do programa de multiplicação de matrizes e os respectivos erros percentuais encontrados. Conforme pode ser observado nas tabelas, durante as predições de desempenho, utilizamos matrizes quadradas com dimensões variando entre 300 e 1200 para as seguintes quantidades de escravos: 1, 2, 4, 6, 10 e 16. O erro percentual médio ficou em torno de 5 %. Este erro é bastante pequeno e muito aceitável em atividades de predições de desempenho, mostrando a validade e a precisão do modelo analítico elaborado. Os maiores erros percentuais acontecem quando realizamos predições para matrizes pequenas. Uma possível causa para esses erros pode ser o método de interpolação (mínimos quadrados) utilizado. Por outro lado, algum *overhead* adicional não considerado durante a geração dos modelos também pode influenciar no resultado. O método dos mínimos quadrados minimiza os erros percentuais dos valores maiores e, conseqüentemente, aumenta os erros percentuais para os pequenos valores. Contudo, os erros absolutos para essas dimensões são pequenos, conforme pode ser visto nas próximas tabelas.

Dimensão	Bytes	Tempo medido	Tempo predito	Erro %
300	720.000	2,091	1,984	-5,09
600	2.880.000	16,090	16,091	0,01
900	6.480.000	51,365	51,653	0,56
1200	11.520.000	264,217	264,029	-0,07

Tabela 5.3: Tempos de execução medidos e preditos para 1 Escravo (em segundos).

Dimensão	Bytes	Tempo medido	Tempo predito	Erro %
300	720.000	1,234	1,227	-0,55
600	2.880.000	8,777	9,184	4,63
900	6.480.000	26,747	28,505	6,57
1200	11.520.000	134,930	135,850	0,68

Tabela 5.4: Tempos de execução medidos e preditos para 2 Escravos (em segundos).

Dimensão	Bytes	Tempo medido	Tempo predito	Erro %
300	720.000	0,889	0,915	2,92
600	2.880.000	5,639	6,017	6,68
900	6.480.000	16,143	17,584	8,92
1200	11.520.000	72,289	72,928	0,88

Tabela 5.5: Tempos de execução medidos e preditos para 4 Escravos (em segundos).

Dimensão	Bytes	Tempo medido	Tempo predito	Erro %
300	720.000	0,807	0,714	-11,54
600	2.880.000	4,701	4,718	0,36
900	6.480.000	12,822	13,457	4,95
1200	11.520.000	51,818	51,123	-1,34

Tabela 5.6: Tempos de execução medidos e preditos para 6 Escravos (em segundos).

Dimensão	Bytes	Tempo medido	Tempo predito	Erro %
300	720.000	0,834	0,698	-16,27
600	2.880.000	4,445	4,264	-4,08
900	6.480.000	11,654	11,473	-1,55
1200	11.520.000	37,698	36,026	-4,43

Tabela 5.7: Tempos de execução medidos e preditos para 10 Escravos (em segundos).

Dimensão	Bytes	Tempo medido	Tempo predito	Erro %
300	720.000	0,892	0,695	-22,04
600	2.880.000	4,609	4,206	-8,73
900	6.480.000	11,444	10,876	-4,96
1200	11.520.000	31,380	28,502	-9,17

Tabela 5.8: Tempos de execução medidos e preditos para 16 Escravos (em segundos).

Um dos problemas que tivemos durante a análise do comportamento da aplicação esteve relacionado com a grande variação apresentada pelo tempo de execução do programa para algumas configurações de n e p (quantidade de *bytes* e número de escravos, respectivamente). Algumas vezes, a diferença encontrada entre os tempos mínimo e máximo chegou a 20 %. Para tempos na casa das centenas, isso representa algo em torno de 20 segundos. Esse fato, além de dificultar a política de seleção dos dados, obtidos nos testes experimentais, prejudica o desenvolvimento do modelo de predição e, conseqüentemente, os valores preditos. Por outro lado, se o tempo de execução medido apresentar um comportamento mais regular, os erros percentuais podem ser menores. Durante todos os testes experimentais tomamos o cuidado de garantir o uso exclusivo do *cluster* para rodar nossa aplicação, descartando interferências relacionadas com o escalonamento de processos provocado por qualquer outra aplicação de usuários. Uma possível causa dessa variação está relacionada com o estado do sistema computacional no momento da execução do programa, uma vez que o processador pode estar ocupado executando tarefas do próprio sistema operacional.

Uma outra dificuldade foi modelar a operação de *broadcast*. Sabemos que o LAM-MPI utiliza um algoritmo baseado em hipercubo ou árvores para implementar essa primitiva e, portanto, existe um atraso entre o instante em que o primeiro e o último escravo recebem a mensagem através do MPI_Bcast. Esse atraso ocasiona um *overhead* no tempo de execução do processo mestre, pois ele fica esperando o resultado da multiplicação de todos os escravos envolvidos na computação. Logo, se alguns escravos demoram mais para receber a mensagem, o sincronismo estabelecido entre o mestre e os escravos é prejudicado. Uma possível solução para o problema seria considerar a diferença entre o tempo despendido pelo mestre no *broadcast* e o tempo que leva para o último escravo receber a mensagem. Baseado na variação desse tempo, em função do número de escravo e do tamanho da mensagem transmitida, podemos estimar este *overhead* e incluí-lo no modelo de predição, melhorando a sua qualidade. Em nosso modelo final estimamos e incluímos esse atraso.

Finalmente, dependendo das particularidades da aplicação que está sendo modelada, muitos problemas inerentes as suas características podem influenciar no seu desempenho.

5.7 Considerações Finais

Neste capítulo, destacamos algumas atividades relacionadas com o processo de análise e predição de desempenho, bem como as principais dificuldades encontradas. Na seção 5.6, apresentamos a modelagem de duas versões diferentes de um programa de multiplicação de matrizes. Na primeira versão, utilizando apenas um mestre e um escravo, a predição de desempenho apresentou erros percentuais menores, quando comparada a predição da segunda versão. Isso pode ser explicado através de dois motivos principais. O primeiro se deve ao fato de que na segunda modelagem mais um parâmetro é variado durante as predições (quantidade de escravos), dificultando a modelagem. O segundo fator pode ser atribuído às condições de sincronismos, estabelecidos entre os escravos e o mestre durante a execução do programa. Apesar das dificuldades encontradas no decorrer do trabalho, conseguimos desenvolver modelos com boa precisão, apresentando erros em torno de 5 %.

Capítulo 6

CONCLUSÕES E TRABALHOS FUTUROS

Nos últimos anos, os *clusters* de PCs têm sido uma plataforma muito utilizada na implementação de programas paralelos. Até pouco tempo atrás, o desenvolvimento de aplicações paralelas estava focado, quase que totalmente, nas máquinas multiprocessadas com alto poder de processamento. Essas máquinas são capazes de oferecer as condições necessárias para alcançarmos o desempenho desejado com a paralelização da aplicação. A opção pelos *clusters* pode ser explicada devido ao baixo custo associado a essa plataforma, quando comparada com as máquinas paralelas. Com isso, o mesmo poder de processamento das máquinas paralelas pode ser alcançado por um *cluster* de PCs a um preço muito mais baixo. Por isso, a utilização dos *clusters* como ambiente de programação paralela é cada vez mais freqüente, favorecendo o desenvolvimento do paradigma de programação baseado em passagem de mensagens.

Enquanto as máquinas multiprocessadas possuem uma memória compartilhada, os *clusters* utilizam uma memória distribuída entre os nós que o compõem. Por isso, a programação sobre essas plataformas são diferenciadas. No primeiro caso, a programação é mais fácil e muito semelhante à forma tradicional. Já sobre os *clusters*, o programador precisa se preocupar com a sincronização e a comunicação entre os processos que estão distribuídos pelos nós. Nesse contexto, a programação paralela através de passagem de mensagem vem sendo muito utilizada e explorada pelos programadores que desenvolvem aplicações para *clusters* de PCs. Uma “biblioteca” que vem sendo muito utilizada com esse objetivo é o MPI.

Em nosso trabalho, desenvolvemos modelos analíticos capazes de representar o comportamento de aplicações paralelas, implementadas com primitivas de trocas de mensagens MPI sobre *cluster* de PCs. Além de criação de modelos matemáticos, realizamos predições de desempenho sobre as aplicações modeladas para estimar o valor de algumas variáveis em situações hipotéticas.

O nosso trabalho baseou-se em uma metodologia de análise e predição de desempenho de programas paralelos MPI proposta em [Li01a]. Nessa metodologia, além de criar uma nova sistemática para modelagem e predição de desempenho, o autor desenvolveu uma série de modelos matemáticos para caracterizar as primitivas de comunicações MPI. No entanto, muitos aspectos relacionados com as características das aplicações, como modelagem de estruturas de repetição, não foram tratados. Por esse motivo, realizamos a expansão dessa metodologia, com o propósito de torná-la mais completa, precisa e interessante.

Desenvolvemos também um novo modelo gráfico, denominado DP*Graph^{+}. Nesse modelo, criamos símbolos para diferenciar cada uma das primitivas de comunicação, os modos de transmissão (bloqueantes e não-bloqueantes), as estruturas de repetição, dentre outras particularidades.

Na metodologia original, estruturas de repetição compostas por primitivas de comunicação não foram modeladas. Por isso, desenvolvemos modelos matemáticos capazes de representar o comportamento dessas estruturas em função do número de iterações executadas. Assim, apresentamos alguns modelos que podem ser aplicados na modelagem de laços de repetição. A idéia dos modelos, apresentados na seção 4.3.1, foi validada através do estudo de caso realizado sobre o programa de multiplicação de matrizes mostrado na seção 5.6. Os modelos foram desenvolvidos para duas classes distintas de estruturas de repetição. Na primeira delas, existe apenas um laço de repetição, enquanto que na segunda existem estruturas de repetição formadas por laços aninhados. Em ambas as situações, consideramos a possibilidade de existir não só primitivas de comunicação, mas também códigos de computação local. Todos os modelos elaborados foram testados durante a modelagem e predição de desempenho do programa de multiplicação de matrizes apresentado.

Depois de modelado os laços de repetição, desenvolvemos modelos de predição para as aplicações do tipo mestre/escravo. A idéia foi desenvolver modelos analíticos em função da quantidade de dados transmitidos (n) e do número de nós envolvidos na

comunicação (p). Os resultados apresentados na seção 4.3.2, mostraram que os modelos desenvolvidos, apesar de simplificados, tiveram boa precisão. Assim como na modelagem dos laços de repetição, aplicamos os modelos desenvolvidos para as estruturas de repetição no programa de multiplicação de matrizes. Os resultados obtidos foram satisfatórios. Nas duas versões do programa de multiplicação de matrizes obtivemos erros percentuais muito baixos. No estudo realizado sobre a primeira versão, o erro percentual médio foi inferior a 3 %. Já na segunda versão, os erros percentuais médios ficaram em torno dos 5 %.

Como sabemos, as atividades de análise e predição de desempenho não são triviais. Uma das maiores dificuldades, dependendo do tipo de modelagem utilizada, está relacionada com a grande quantidade de dados que devem ser analisados e selecionados durante os testes experimentais. Diante disso, com o auxílio da ferramenta chamada Scilab, implementamos um conjunto de funções para automatizar algumas etapas do processo de análise e predição de desempenho.

Como principais contribuições deste trabalho, destacamos: a proposta de um novo modelo gráfico para representar programas paralelos MPI, o desenvolvimento de modelos para predizer o tempo gasto em estruturas de repetição formadas por primitivas de comunicação, a elaboração de modelos analíticos para representar o comportamento de aplicações do tipo mestre/escravo e as funções implementadas com o Scilab.

6.1 Trabalhos Futuros

Como sugestões de trabalhos futuros, podemos citar:

- Modelagem de aplicações para estudos de predição sobre *clusters* com redes de interconexão diferentes da que usamos neste trabalho, como por exemplo *Myrinet, Gigabit Ethernet*, dentre outras. Com isso, é possível repetir os estudos realizados e verificar a influência da rede nos modelos desenvolvidos neste trabalho, bem como no desempenho das aplicações;

- Agrupar as funções Scilab desenvolvidas neste trabalho e acrescentar novas funcionalidades, como a instrumentação automática do código do programa, em

uma ferramenta capaz de facilitar as atividades de análise e predição de desempenho;

- Modelagem e predição de desempenho de programas mais complexos para testar a validade dos modelos desenvolvidos neste trabalho;

- Modelagem de estruturas condicionais através de uma abordagem mista entre as modelagens analítica e estatística. Os modelos a serem desenvolvidos ampliarão o poder de representação da metodologia;

- Testar os modelos desenvolvidos em versões mais novas do LAM-MPI ou sobre outras implementações do padrão MPI, como por exemplo o MPICH. Através dos testes será possível comparar os resultados obtidos e verificar se os modelos desenvolvidos neste trabalho podem ser aplicados em outros ambientes;

- Repetir os estudos que realizamos neste trabalho em outros tipos de *clusters*, como por exemplo os *clusters* Windows.

Referências Bibliográficas

[Adve93] ADVE, V. S. **Analysing the Behavior and Performance of Parallel Programs**. Tese (Doutorado). 1993. 127p. Department of Computer Science, University of Wisconsin-Madison. Madison.

[Aho83] AHO, A.; HOPCROFT, J.; ULMAN, J. **Data Structures and Algorithms**. Addison-Wesley Publishers, 1983.

[Alasdair94] ALASDAIR, R. et al. CHIMP Version 2.0 User Guide. University of Edinburg, Mar. 1994.

[Al-Tawil94] AL-TAWIL, K. E.; Moritz C. A. Performance Modeling and Evaluation of MPI. **Journal of Parallel and Distributed Computing**, v.61, p. 202-223, 2001.

[Andrews93] ANDREWS, G. R.; OLSSON, R.A. The SR Programming Language: Concurrency in Practice. Benjamin/Cummings, Redwood City, California, 1993.

[Barney02] BARNEY, B. M. Introduction to Parallel Computing. 2002. Disponível em: <http://www.llnl.gov/computing/tutorials/workshops/workshop/home/MAIN.html>. Acesso: 03 de jun. 2002.

[Barney02b] BARNEY, B. M. Message Passing Interface (MPI). 2002. Disponível em: <http://www.llnl.gov/computing/tutorials/workshops/workshop/home/MAIN.html>. Acesso: 03 de jun. 2002.

[Barney02c] BARNEY, B. M. MPI Performance Topics. 2002. Disponível em: <http://www.llnl.gov/computing/tutorials/workshops/workshop/home/MAIN.html>. Acesso: 03 de jun. 2002.

[Barney03] BARNEY, B. M. Posix Threads. 2003. Disponível em: <http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html>. Acesso: 20 de fev. 2003.

[Ben-Ari90] BEN-ARI, M. **Principles of Concurrent and Distributed Programming**. Prentice Hall, 1990. 225p.

[Block95] BLOCK R. J. et al. Automated Performance Prediction of Message-Passing Parallel Programs. In: Supercomputingt'95, San Diego, CA, 1995. **Proceedings**.

[Burns94] BURNS, G. et al. LAM: An Open Cluster Environment for MPI. Ohio Supercomputer Center, May, 1994.

[Buyya99] BUYYA, R. **High Performance Cluster Computing: Programming and Applications**. New Jersey: Prentice Hall PTR, 1999. 664p.

[Clement94] CLEMENT, M. J.; QUINN, M. J. Analytical Performance Prediction on Multicomputers. Department of Computer Science, Oregon State University, Oregon, 1994.

[Crovella94] CROVELLA, M. E. **Performance Prediction and Tuning of Parallel Programs**. 1994. 105p. Tese (Doutorado). Computer Science Department, University of Rochester. Rochester, New York.

[Culler99] CULLER, D. E.; SINGH, J. P. **Parallel Computer Architecture: A Hardware/Software Approach**. San Francisco, California: Morgan Kaufmann Publishers, Inc, 1999. 1025p.

[Fahringer93] FAHRINGER, T. **Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers.** 1993. 194p. Tese (Doutorado).

[Gautama98] GAUTAMA, H. **A Probabilistic Approach to the Analysis of Program Execution Time**. Dissertação (Mestrado). 1998. Technische Universiteit Delft. Delft.

[Geist96] GEIST, G. A. et al. PVM and MPI: a Comparison of Features. **Calculateurs Paralleles**, v.8, n.2, 1996.

[Gemund93a] GEMUND, A. J. C. van. Performance Prediction of Parallel Processing Systems: The PAMELA Methodology. In: 7th ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING (ICS'93), Tokyo, 1993. **Proceedings**. p.318-327.

[Gemund96] GEMUND, A. J. C. van. **Performance Modeling of Parallel Systems**. Delft, 1996. Tese (Doutorado). Delft University of Technology, Delft University Press, ISBN 90-407-1326-X.

[Gropp94] GROPP, B. et al. Portable MPI Model Implementation. Argonne National Laboratory, July, 1994.

[Grove01] GROVE, D. A.; CODDINGTON, P. D. **A Performance Modeling System for Message-Passing Parallel Programs**. Adelaide, Australia: Department of Computer Science, Adelaide University, 2001. (DHCP Technical Report 105).

[Gubitoso96] GUBITOSO, M. D. **Modelos Analíticos de Desempenho para Sistemas de Memória Compartilhada Virtual**. Tese (Doutorado). 1996. 95p. Departamento de Ciência da Computação, Instituto de Matemática e Estatística, Universidade de São Paulo. São Paulo.

[Hpf03] HPF - High Performance Fortran: conceitos e definições. Disponível em: <http://www.crpc.rice.edu/HPFF>. Acesso: 20 de fev. 2003.

[Hu97] HU, L.; GORTON I. **Performance Evaluation for Parallel Systems: A Survey**. Austria: Department of Computer Systems, University of NSW, 1997. (Technical Report UNSW-CSE-TR-9797).

[Jain91] JAIN, R. **The Art of Computer Systems Performance Analysis: thechiniques for experimental desing, measurement, simulation, and modeling**. New York: John Wiley & Sons, INC, 1991, 685p.

[Kalos86] KALOS, M. H.; WHITLOCK, P. A. **Monte Carlo Methods: Fundamental Algorithms**. Addison-Wesley Publishers, 1986.

[Laine02] LAINE, J. M. et al. Análise e Predição de Desempenho de Programas MPI em Redes de Estações de Trabalho. In: WORKSHOP EM

DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMPUTAÇÃO, 1, Florianópolis-SC, 2002. **Anais**. SBC, 2002. p.25-36.

[Li01a] LI, K. C. **Análise e Predição de Desempenho de Programas Paralelos em Redes de Estações de Trabalho**. 2001. 113p. Tese (Doutorado) - Departamento de Engenharia de Computação e Sistemas Digitais. Escola Politécnica, Universidade de São Paulo. São Paulo.

[Li01b] LI, K. C.; SATO, L. M. Representing parallel programs with MPI by exploiting a graph-based approach. In: VII ARGENTINE CONGRESS ON COMPUTER SCIENCE, Argentine, 2001. **Proceedings**.

[Meira95] MEIRA, W. Jr. **Modeling Performance of Parallel Programs**. Rochester, New York: Computer Science Department, The University of Rochester, 1995. (Technical Report 589).

[Moura99] MOURA e SILVA, L.; BUYYA, R. Parallel Programming Models and Paradigms. In: BUYYA, R. **High Performance Cluster Computing: Programming and Applications**. New Jersey: Prentice Hall, PTR, 1999. p. 4-27.

[Mpi02] MPIFORUM. **MPI: A Message Passing Interface Standard**. Knoxville: University of Tennessee, 1995. (Technical Report). Disponível em: <http://www.mpi-forum.org>. Acesso: 03 de jun. 2002.

[Mucci98] MUCCI, P. J. et al. **The MPBench Report**. Tennesse: Department of Computer Science, University of Tennessee, 1998. (Technical Report).

[Nitzberg91] NITZBERG, B.; LO, V. Distributed Shared Memory: A Survey of Issues and Algorithms. **IEEE Computer**, v. 24, n.8, p.52-60, 1991.

[Nupairoj94] NUPAIROJ, N.; NI, L. M. Performance Evaluation of Some MPI Implementations on Workstation Clusters. In: SCALABLE PARALLEL LIBRARIES CONFERENCE, 1994. **Proceedings**. IEEE Computer Society Press, 1994. p.98-105.

[Oak02] OAK RIDGE NATIONAL LABORATORY. PVM - Parallel Virtual Machine. Disponível em: <http://www.scm.ornl.gov/pvm>. Acesso: 25 de jul. 2002.

[Oed81] OED, W.; MERTENS, B. Characterization of Computer System Workload. **Computer Performance**, v. 2, n. 2, p. 77-83, 1981.

[Oliveira02a] OLIVEIRA, H. M. et al. Performance Analysis and Prediction of some MPI Communication Primitives. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS (PDPTA'02). Las Vegas, Nevada, USA, 2002. **Proceedings**.

[Oliveira02b] OLIVEIRA, H. M. **Modelagem e Predição de Desempenho de Primitivas de Comunicação MPI**. 2003. 98p. Dissertação (Mestrado). Departamento de Engenharia de Computação e Sistemas Digitais. Escola Politécnica, Universidade de São Paulo. São Paulo.

[OpenMP03] OPENMP - Conceitos e Definições. Disponível em: <http://www.openmp.org>. Acesso: 20 de fev. 2003.

[Pramanick99] PRAMANICK, I. Parallel Programming Languages and Environments. In: BUYYA, R. **High Performance Cluster Computing: Programming and Applications**. New Jersey: Prentice Hall, PTR, 1999. p. 28-47.

[Press92] PRESS, W. et al. **Numerical Recipes in C: The Art of Scientific Computing**. 2nd Edition. Cambridge University Press, 1992. 994p.

[Scilab] SCILAB. Introduction to Scilab. INRIA - Unité de recherche de Rocquencourt - Projet Meta2, França. Disponível em: <http://www-rocq.inria.fr/scilab>

[Yan96] YAN, Y.; ZHANG, X. An Effective and Practical Model for Parallel Computing on Non-Dedicated Heterogeneous NOW. **Journal of Parallel and Distributed Computing**, v.38, n.1, p.63-80, 1996.

[Wabnig93] WABNIG, H.; KOTSIS, G.; HARING, G. Performance Prediction of Parallel Programs. In: MMB'93, Vienna, Austria, 1993. p.64-76.

Apêndice A

CÓDIGO DOS PROGRAMAS MODELADOS

Neste Apêndice estão os códigos dos programas que foram utilizados durante os testes experimentais para o desenvolvimento e validação dos modelos de predição. Inicialmente, apresentamos os programas utilizados para o desenvolvimento dos modelos de predição das aplicações mestre/escravo (vide seção 4.3.2). Conforme discutido, existem 3 possíveis situações de sincronismo entre os processos escravos e o processo mestre. Para cada situação modelada reaproveitamos o mesmo programa, fazendo pequenos ajustes, como inserção de barreiras ou *delays*, para estabeler o sincronismo desejado. Por fim, exibimos as duas versões do programa de multiplicação de matrizes que foram modeladas e tiveram seu tempo de execução predito através dos modelos desenvolvidos.

Escravos prontos antes do mestre

```
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include <unistd.h>
#include "mpi.h"
#define MAX_WORKERS 16

/* SOME CONSTANTS */

#define NRA 2500                /* number of rows in matrix A */
#define NCA 2500                /* number of columns in matrix A*/
#define NCB 2500                /* number of columns in matrix B */
```

```
#define MASTER 0               /* taskid of first task */

/* GLOBAL VARIABLE */

MPI_Status status;

double a[NRA][NCA],            /* matrix A to be multiplied */
       b[NCA][NCB],            /* matrix B to be multiplied */
       c[NRA][NCB];            /* result matrix C */

/* MAIN PROGRAM */

main(int argc, char **argv)
{
int numtasks,                  /* number of tasks in partition */
    taskid,                    /* a task identifier */
    numworkers,                /* number of worker tasks */
    source,                    /* task id of message source */
    dest,                      /* task id of message destination */
    i, j,                      /* general counter */
    constante,                 /* constant of send */
    elements,                  /* number of elements to send */
    timercount;                /* counter to timer */

char * arq,                   /* file name to write the results */
     * str;                   /*  */

char nome[5];

FILE *fp;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  numworkers = numtasks-1;
  elements = NRA*NCA;
  timercount = 0;

  if (taskid == MASTER) {
    printf("Number of worker tasks = %d\n", numworkers);
    printf("Tamanho das MATRIZES = %d\n", NRA);
    // inicializacao das matrizes
    printf("Inicializando as matrizes!\n");
    for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
        a[i][j]= 0.5;
    for (i=0; i<NCA; i++)
      for (j=0; j<NCB; j++)
        b[i][j]= 2.0;
  }//if
```

```
  // ENVIO DA MATRIZ A SEND(n)

  if (taskid==MASTER){
    for (dest=1; dest<=numworkers; dest++) {
      MPI_Send(&a, elements, MPI_INT, dest, 1, MPI_COMM_WORLD);
    }
  }
  else {
    MPI_Recv(&a, elements, MPI_INT, MASTER, 1, MPI_COMM_WORLD, &status);
  } //if

  MPI_Barrier(MPI_COMM_WORLD);

  // ENVIO DE UMA CONSTANTE SEND(k)

  if (taskid==MASTER){
    constante = 7;
    for (dest=1; dest<=numworkers; dest++) {
      MPI_Send(&constante, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
    } //for
  } else{
    MPI_Recv(&constante, 1, MPI_INT, MASTER, 2, MPI_COMM_WORLD, &status);
  } //if
// print results on file

  if (taskid == MASTER){
    fp = fopen ("master", "a+");
    for (i=0; i<timercount; i++)
      fprintf (fp,"     %f", timer[i]);
    fprintf (fp, "\n");
    fclose(fp);
  }
  else {
    itoa(taskid,nome,16);
    printf(nome);
    fp = fopen (nome, "a+");
    for (i=0; i<timercount; i++)
      fprintf (fp,"     %f", timer[i]);
    fprintf (fp, "\n");
    fclose(fp);
  }

  MPI_Finalize();

} /* of main */
```

Escravos prontos depois do mestre

```
#include <stdio.h>
#include <sys/time.h>
```

```
#include <math.h>
#include <unistd.h>
#include "mpi.h"

#define MAX_WORKERS 16

/* SOME CONSTANTS */

#define NRA 2500                /* number of rows in matrix A */
#define NCA 2500                /* number of columns in matrix A*/
#define NCB 2500                /* number of columns in matrix B */
#define MASTER 0                /* taskid of first task */

/* GLOBAL VARIABLE */

MPI_Status status;

double a[NRA][NCA],             /* matrix A to be multiplied */
       b[NCA][NCB],             /* matrix B to be multiplied */
       c[NRA][NCB];             /* result matrix C */

/* MAIN PROGRAM */

main(int argc, char **argv)
{
int numtasks,                   /* number of tasks in partition */
    taskid,                     /* a task identifier */
    numworkers,                 /* number of worker tasks */
    source,                     /* task id of message source */
    dest,                       /* task id of message destination */
    i, j,                       /* general counter */
    constante,                  /* constant of send */
    elements,                   /* number of elements to send */
    timercount;                 /* counter to timer */

char * arq,                    /* file name to write the results */
     * str;                   /*  */

char nome[5];

FILE *fp;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  numworkers = numtasks-1;
  elements = NRA*NCA;
  if (taskid == MASTER) {
    printf("Number of worker tasks = %d\n", numworkers);
    printf("Tamanho das MATRIZES = %d\n", NRA);
    // inicializacao das matrizes
```

```
  printf("Inicializando as matrizes!\n");
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= 0.5;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= 2.0;
}//if
else{
 printf("Sou escravo e estou esperando ....");
 sleep(2);
}
// ENVIO DA MATRIZ A SEND(n)
if (taskid==MASTER){
  for (dest=1; dest<=numworkers; dest++) {
    MPI_Send(&a, elements, MPI_INT, dest, 1, MPI_COMM_WORLD);
  }
}
else {
  MPI_Recv(&a, elements, MPI_INT, MASTER, 1, MPI_COMM_WORLD, &status);
} //if

// ENVIO DE UMA CONSTANTE SEND(k)

if (taskid==MASTER){
  constante = 7;
  for (dest=1; dest<=numworkers; dest++) {
    MPI_Send(&constante, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
  } //for
} else{
  MPI_Recv(&constante, 1, MPI_INT, MASTER, 2, MPI_COMM_WORLD, &status);
} //if

if (taskid == MASTER){
  fp = fopen ("master", "a+");
  for (i=0; i<timercount; i++)
    fprintf (fp,"     %f", timer[i]);
  fprintf (fp, "\n");
  fclose(fp);
}
else {
  itoa(taskid,nome,16);
  printf(nome);
  fp = fopen (nome, "a+");
  for (i=0; i<timercount; i++)
    fprintf (fp,"     %f", timer[i]);
  fprintf (fp, "\n");
  fclose(fp);
}

MPI_Finalize();
```

```
} /* of main */
```

Escravos e mestre sincronizados

```
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include <unistd.h>
#include "mpi.h"

#define MAX_WORKERS 16

/* SOME CONSTANTS */

#define NRA 900         /* number of rows in matrix A */
#define NCA 900         /* number of columns in matrix A*/
#define NCB 900                 /* number of columns in matrix B */
#define MASTER 0                /* taskid of first task */

/* GLOBAL VARIABLE */

MPI_Status status;

double a[NRA][NCA],             /* matrix A to be multiplied */
       b[NCA][NCB],             /* matrix B to be multiplied */
       c[NRA][NCB];             /* result matrix C */

/* MAIN PROGRAM */

main(int argc, char **argv)
{
int numtasks,                   /* number of tasks in partition */
    taskid,                     /* a task identifier */
    numworkers,                 /* number of worker tasks */
    source,                     /* task id of message source */
    dest,                       /* task id of message destination */
    i, j,                       /* general counter */
    constante,                  /* constant of send */
    elements,                   /* number of elements to send */
    timercount;                 /* counter to timer */


char * arq,                    /* file name to write the results */
     * str;                   /*  */

char nome[5];

FILE *fp;
```

```
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  numworkers = numtasks-1;
  elements = NRA*NCA;
  if (taskid == MASTER) {
    printf("Number of worker tasks = %d\n", numworkers);
    printf("Dimensao das matrizes: %d\n", NCA);
    // inicializacao das matrizes
    printf("Inicializando as matrizes!\n");
    for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
        a[i][j]= 0.5;
    for (i=0; i<NCA; i++)
      for (j=0; j<NCB; j++)
        b[i][j]= 2.0;
  }//if

  MPI_Barrier(MPI_COMM_WORLD);

  // ENVIO DA MATRIZ A SEND(n)

  if (taskid==MASTER){
    for (dest=1; dest<=numworkers; dest++) {
      MPI_Send(&a, elements, MPI_INT, dest, 1, MPI_COMM_WORLD);
    }
  }
  else {
    MPI_Recv(&a, elements, MPI_INT, MASTER, 1, MPI_COMM_WORLD, &status);
    TIMER_STOP(timercount);
  } //if

  MPI_Barrier(MPI_COMM_WORLD);

  // ENVIO DE UMA CONSTANTE SEND(k)

  if (taskid==MASTER){
    constante = 7;
    for (dest=1; dest<=numworkers; dest++) {
      MPI_Send(&constante, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
    } //for
  } else{
    MPI_Recv(&constante, 1, MPI_INT, MASTER, 2, MPI_COMM_WORLD, &status);
  } //if

// print results on file

  if (taskid == MASTER){
    fp = fopen ("master", "a+");
    for (i=0; i<timercount; i++)
      fprintf (fp,"     %f", timer[i]);
```

```
    fprintf (fp, "\n");
    fclose(fp);
  }
  else {
    itoa(taskid,nome,16);
    printf(nome);
    fp = fopen (nome, "a+");
    for (i=0; i<timercount; i++)
      fprintf (fp,"     %f", timer[i]);
    fprintf (fp, "\n");
    fclose(fp);
  }

  MPI_Finalize();

} /* of main */
```

Versão 1 do programa de multiplicação de matrizes

O código apresentado a seguir é referente à versão 1 do programa de multiplicação de matrizes, cujo modelo e resultados foram apresentados na seção 5.6.1. Nesse programa, o processo mestre utiliza apenas primitivas de comunicação ponto-a-ponto dentro de uma estrutura de repetição para distribuir as matrizes A e B entre os escravos.

```
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include <unistd.h>
#include "mpi.h"

#define NRA 50                  /* number of rows in matrix A */
#define NCA 50                  /* number of columns in matrix A*/
#define NCB 50                  /* number of columns in matrix B */
#define MASTER 0                /* taskid of first task */
#define FROM_MASTER 1           /* setting a message type */
#define FROM_WORKER 2           /* setting a message type */

MPI_Status status;
main(int argc, char **argv)
{
int numtasks,                   /* number of tasks in partition */
    taskid,                     /* a task identifier */
    numworkers,                 /* number of worker tasks */
    source,                     /* task id of message source */
    dest,                       /* task id of message destination */
    nbytes,                     /* number of bytes in message */
    mtype,                      /* message type */
    intsize,                    /* size of an integer in bytes */
    dbsize,                     /* size of a double float in bytes */
```

```
    rows,                       /* rows of matrix A sent to each wor-
ker */
    averow, extra, offset,      /* used to deter-
mine rows sent to each worker */
    i, j, k,                    /* misc */
    count;

double a[NRA][NCA],             /* matrix A to be multiplied */
       b[NCA][NCB],             /* matrix B to be multiplied */
       c[NRA][NCB];             /* result matrix C */

intsize = sizeof(int);
dbsize = sizeof(double);

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
numworkers = numtasks-1;
/******************** master task ****************************/
if (taskid == MASTER) {
  printf("Number of worker tasks = %d\n",numworkers);
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= j;

  /* send matrix data to the worker tasks */
  averow = NRA/numworkers;
  extra = NRA%numworkers;
  offset = 0;
  mtype = FROM_MASTER;
  for (dest=1; dest<=numworkers; dest++) {
    rows = (dest <= extra) ? averow+1 : averow;
    printf("   sending %d rows to task %d\n",rows,dest);
    MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
    count = NRA*NCA;
    MPI_Send(&a, count, MPI_DOUBLE, dest, 3, MPI_COMM_WORLD);
    count = NCA*NCB;
    MPI_Send(&b, count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    offset = offset + rows;
    }

  /* wait for results from all worker tasks */
  mtype = FROM_WORKER;
  for (i=1; i<=numworkers; i++) {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &sta-
tus);
```

```
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    count = rows*NCB;
    MPI_Recv(&c[offset][0], count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD,
    &status);
    }
/* end of master section */

/********************** worker task *************************/
if (taskid > MASTER) {
  mtype = FROM_MASTER;
  source = MASTER;
  MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
  count = NRA*NCA;
  MPI_Recv(&a, count, MPI_DOUBLE, source, 3, MPI_COMM_WORLD, &status);
  count = NCA*NCB;
  MPI_Recv(&b, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &sta-
tus);
  for (k=0; k<NRA; k++)
      for (j=0; j<NCA; j++)
        c[k][j] = a[k][j] + b[k][j];
      }
  mtype = FROM_WORKER;
  MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
  }  /* end of worker */
  MPI_Finalize();
} /* of main */
```

Versão 2 do programa de multiplicação de matrizes

No próximo código, referente à versão 2 do programa de multiplicação de matrizes, o processo mestre realiza a distribuição das matrizes A e B de uma forma diferente. Agora, a matriz A é enviada através de uma primitiva ponto-a-ponto (MPI_Send), enquanto a matriz B é transmitida aos escravos através de uma primitiva de comunicação coletiva (MPI_Bcast).

```
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include <unistd.h>
#include "mpi.h"
#define DIM 1600 /* DIM is the dimension of matrix */
#define numworkers 14 /* number of slaves*/
#define MASTER 0               /* taskid of first task */
#define FROM_MASTER 1          /* setting a message type */
#define FROM_WORKER 2          /* setting a message type */
```

```
/* GLOBAL VARIABLE */

MPI_Status status;

double a[DIM][DIM],             /* matrix A to be multiplied */
       b[DIM][DIM],             /* matrix B to be multiplied */
       c[DIM][DIM];             /* result matrix C */

/* MAIN PROGRAM */

main(int argc, char **argv)
{
int numtasks,                   /* number of tasks in partition */
    taskid,                     /* a task identifier */
    source,                     /* task id of message source */
    dest,                       /* task id of message destination */
    nbytes,                     /* number of bytes in message */
    mtype,                      /* message type */
    intsize,                    /* size of an integer in bytes */
    dbsize,                     /* size of a double float in bytes */
    rows,                       /* rows of matrix A sent to each wor-
ker */
    averow, extra, offset,      /* used to deter-
mine rows sent to each worker */
    timercount,                 /* used to measure the times */
    i, j, k,                    /* misc */
    count;

char arq[50], /*output file name*/
     *str, auxstr[5];
FILE *fp;
int tid, wid;

intsize = sizeof(int);
dbsize = sizeof(double);
timercount = 0;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

/************************ MASTER TASK ******************************/

if (taskid == MASTER) {

  printf("Number of worker tasks = %d\n", numworkers);
  for (i=0; i<DIM; i++)
    for (j=0; j<DIM; j++)
      a[i][j]= 0.5;
  for (i=0; i<DIM; i++)
```

```
      for (j=0; j<DIM; j++)
        b[i][j]= 2.0;

    /* send matrix data to the worker tasks */
    averow = DIM/numworkers;
    extra = DIM%numworkers;
    offset = 0;
    mtype = FROM_MASTER;
    timercount++;

    for (dest=1; dest<=numworkers; dest++) {
      rows = (dest <= extra) ? averow+1 : averow;
      MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
      timercount++;
      MPI_Send(&rows, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
      count = rows*DIM;
      MPI_Send(&a[offset][0], count, MPI_DOUBLE, dest, 3, MPI_COMM_WORLD);
      offset = offset + rows;
    }//end for

    // Send Matrix b to all slaves
    MPI_Bcast(&b,DIM*DIM,MPI_DOUBLE,0,MPI_COMM_WORLD);
    /* wait for results from all worker tasks */
    mtype = FROM_WORKER;

    for (i=1; i<=numworkers; i++) {
      source = i;
      MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &sta-
tus);
      MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
      count = rows*DIM;
      MPI_Recv(&c[offset][0], count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD,
      &status);
    }//and for
    // master file
    str = "Resultados/master14s";
    for(i=0;i<strlen(str);i++) arq[i] = *(str+i);
    arq[i] = '\0';
    itoa(DIM,auxstr,16);
    strcat(arq,auxstr); /*concatena MASTER+DIM*/
    }
  /* end of master section */


  /************************ WORKER TASK ********************************/

  if (taskid > MASTER) {

    mtype = FROM_MASTER;
    source = MASTER;
    MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
```

```
  MPI_Recv(&rows, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
  count = rows*DIM;
  MPI_Recv(&a, count, MPI_DOUBLE, source, 3, MPI_COMM_WORLD, &status);
  count = DIM*DIM;
  MPI_Bcast(&b, DIM*DIM, MPI_DOUBLE,0, MPI_COMM_WORLD);
  for (k=0; k<DIM; k++)
    for (i=0; i<rows; i++) {
      c[i][k] = 0.0;
      for (j=0; j<DIM; j++)
        c[i][k] = c[i][k] + a[i][j] * b[j][k];
      }
  mtype = FROM_WORKER;
  MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  TIMER_STOP(timercount);
  MPI_Send(&c, rows*DIM, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
  // slave file
  str = "Resultados/Slave14s";
  for(i=0;i<strlen(str);i++) arq[i] = *(str+i);
  arq[i] = '\0';
  itoa(DIM,auxstr,16);
  strcat(arq,auxstr); /*concatena Slave+DIM*/
  i = strlen(arq);
  arq[i] = '-';
  arq[i+1] = '\0';
  itoa(taskid,auxstr,16);
  strcat(arq,auxstr); /*concatena Slave+DIM+taskid*/

}  /* end of worker */

MPI_Finalize();

} /* of main */
```

Apêndice B

FUNÇÕES SCILAB IMPLEMENTADAS

Neste Apêndice apresentamos as principais funções implementadas com o auxílio do Scilab. Estas funções foram desenvolvidas para automatizar algumas etapas do processo de modelagem e predição de desempenho.

1. ***MediaTempoTotal.sci***: lê os arquivos que contêm o tempo total de execução dos programas testados, seleciona os valores segundo uma política definida e calcula a média dos tempos totais selecionados. Essas médias calculadas são comparadas com os valores preditos para checar a validade dos modelos desenvolvidos. Como entrada para essa função, passamos o diretório onde estão os arquivos com os tempos medidos, as dimensões das matrizes utilizadas durante os testes experimentais e a quantidade de execuções realizadas.

2. ***MediaTempoParcial.sci***: semelhante à função anterior, só que as operações são realizadas sobre arquivos que contém os tempos parciais medidos para cada trecho identificado no DP*Graph^{+}. As médias calculadas são utilizadas durante o desenvolvimento dos modelos de predição. Essa função recebe como parâmetros de entrada as mesmas informações fornecidas à função anterior.

3. ***lscf.sci***: implementa o método dos mínimos quadrados, descrito na seção 4.3. Utilizamos essa função para gerar os modelos de predição para os trechos identificados no programa modelado. Essa função utiliza como parâmetros de entrada um vetor com os valores do eixo X, um vetor com os valores do eixo Y e a

dimensão do polinômio gerado.

4. ***PreditoMedido.sci***: realiza predições para aplicações mestre/escravo, considerando cada um dos três casos de sincronismo descritos na seção 4.3.2. É uma função interativa que permite ao usuário escolher qualquer um dos possíveis casos de sincronismos preditos. Além de fazer as predições, essa função também gera os gráficos com a curva dos valores preditos e os pontos medidos durante os testes experimentais. Para essa função, passamos o diretório com os tempos medidos e a quantidade de bytes transmitidos.

5. ***PredicaoMatriz.sci***: realiza as predições da segunda versão do programa de multiplicação de matrizes. Depois de predizer o tempo de execução do programa sobre uma situação definida pelo usuário, essa função gera os gráficos comparativos entre esses tempos e os tempos medidos. Como parâmetros de entrada, essa função recebe a dimensão das matrizes, o número de escravos utilizados e o diretório com os tempos medidos.

Resumidamente, essas são as características das funções que implementamos para auxiliar o processo de modelagem e predição de desempenho.