**HÉLIO MARCI DE OLIVEIRA**

# Modelagem e Predição de Desempenho de Primitivas de Comunicação MPI

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

São Paulo
2003

**HÉLIO MARCI DE OLIVEIRA**

# Modelagem e Predição de Desempenho de Primitivas de Comunicação 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

Agradeço à Deus pela capacidade e proteção, oportunidades e desafios, e tudo mais que me foi concedido... embora muitas vezes eu ainda não estivesse apto a compreender.

Aos meus pais Antonio e Flordinice, também aos meus irmãos Marcos e Edson, pelo apoio fundamental e invariavelmente presente.

Ao meu orientador, prof. Edson Midorikawa, pela confiança em mim depositada e pelos auxílios em muitos casos imprescindíveis nestes quase dois anos de mestrado.

Ao amigo e companheiro Jean, votos sinceros de sucesso e realizações. Como já dizia o Raul: "nunca se vence uma guerra lutando sozinho"!

Aos colegas de laboratório (às vezes parceiros de festas também!), em especial Gisele, Fabio, Augusto (este de mais longa data!), Mário e Li. Ainda a todos os demais cuja convivência no LASB contribuiu para minha formação profissional e pessoal.

À profa. Liria, sempre dedicada e prestativa, o meu respeito e agradecimento.

Ao amigo Marcio, por todo o incentivo e pelas palavras de conforto. Ao Rodrigo, pela cooperação e companheirismo em tantos momentos.

Finalmente, ao CNPq pelo suporte financeiro disponibilizado para a realização deste trabalho.

# RESUMO

O desenvolvimento de programas paralelos e distribuídos encontra na programação baseada em passagem de mensagens uma abordagem eficaz para explorar adequadamente as características das máquinas de memória distribuída. Com o uso de *clusters* e de bibliotecas de suporte às trocas de mensagens, como o padrão MPI (*Message Passing Interface*), aplicações eficientes e economicamente viáveis podem ser construídas. Em tais sistemas, o tempo despendido nas comunicações constitui um importante fator de desempenho a ser considerado e requer a utilização de procedimentos e cuidados para a sua correta caracterização. Neste trabalho, modelos analíticos de primitivas de comunicação bloqueante MPI são desenvolvidos segundo uma metodologia de análise e predição apropriada. São tratadas algumas das principais operações ponto-a-ponto e coletivas e, utilizando técnicas de ajuste de curvas e tempos experimentais, o comportamento das primitivas de comunicação é representado em equações, possibilitando ainda a realização de análises e predições de desempenho em função do tamanho das mensagens e do número de processos envolvidos. Através de testes em um *cluster* de estações de trabalho, a precisão dos modelos elaborados é comprovada. Sendo a maioria dos erros percentuais inferiores a 8%, os resultados obtidos confirmam a validade do processo de modelagem. Além disso, o trabalho apresenta um conjunto de funções construídas com o objetivo de oferecer suporte a atividades de análise e predição, procurando facilitar e automatizar sua execução.

# ABSTRACT

The development of parallel and distributed programs finds at message-passing programming a powerful approach to explore properly the distributed memory machines issues. Using clusters and message-passing libraries, as MPI standard (Message Passing Interface), efficient and cost effective applications can be constructed. In these systems, the time spent with communications means a important performance factor to be considered and its correct characterization requires procedures and cautions. In this work, analytic models for MPI blocking communication primitives are developed according one appropriate methodology for analysis and prediction. Some of the main peer-to-peer and collective operations are treated, and through curve fitting techniques and experimental times the behavior of the communication primitives is represented in equations, allowing also the accomplishment of performance analysis and prediction in function of the message length and the number of processes. Tests realized in a cluster of workstations prove the accuracy of the elaborated models. With most of errors within 8%, the obtained results show the validity of the modeling process. Also, the work presents a set of functions constructed with the purpose of support analysis and prediction activities, in order to facilitate and automate them.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Capítulo 1

# INTRODUÇÃO

O processamento paralelo é geralmente destinado à resolução de problemas que demandem grande poder computacional. As máquinas de memória distribuída têm sido bastante utilizadas nestes casos, oferecendo alto desempenho e alta disponibilidade muitas vezes a um preço bastante inferior aos computadores paralelos de memória compartilhada.

Um *cluster* pode ser visto como um sistema de processamento paralelo ou distribuído trabalhando como um recurso computacional único e integrado, embora seja composto por um conjunto de computadores distintos [Moura99]. Devido a características como escalabilidade, alto desempenho e custo financeiro relativamente baixo, estes sistemas têm sido objeto de estudo e merecido destaque tanto no meio acadêmico como no empresarial. Podendo apresentar diversas configurações possíveis, através de variações no número e nas características dos nós de processamento, por exemplo, os *clusters* são construídos basicamente pela interconexão de um conjunto de máquinas em uma rede de alta velocidade.

As aplicações projetadas para *clusters* costumam utilizar bibliotecas de suporte às operações de comunicação, procurando desta forma explorar melhor a arquitetura de memória distribuída com a programação baseada em passagem de mensagens. O PVM (*Parallel Virtual Machine*) e o MPI (*Message Passing Interface*) oferecem as mais conhecidas bibliotecas com tal finalidade. Sendo uma solução mais recente, o MPI constitui um padrão para o desenvolvimento de programas paralelos baseados em trocas de mensagens, disponibilizando um conjunto

de primitivas capazes de oferecer suporte a várias operações de comunicação e definindo semânticas e sintaxes próprias para estas operações.

Contudo, somente a utilização do padrão MPI, ou mesmo do PVM, não é suficiente para a elaboração de aplicações eficientes baseadas em passagem de mensagens. É necessária a distribuição apropriada de tarefas entre os nós de processamento e o uso adequado das primitivas de comunicação disponibilizadas. Neste contexto, as técnicas de medição, análise e predição de desempenho podem oferecer mecanismos para avaliação destas aplicações, procurando indicar justificativas para desempenhos ruins ou auxiliar a obtenção de programas com melhores tempos de execução.

No presente trabalho, descrevemos a aplicação de algumas técnicas na elaboração de modelos analíticos para análise e predição de desempenho de operações de comunicação MPI. São caracterizadas algumas primitivas MPI e, através dos modelos gerados, predições de tempos de execução podem ser realizadas variando o número de nós de processamento envolvidos e o tamanho das mensagens transmitidas. Além disso, apresentamos um conjunto de funções desenvolvidas com o objetivo de oferecer suporte ao processo de medição e modelagem utilizado, facilitando e automatizando algumas tarefas envolvidas.

## 1.1 Objetivo do trabalho

O objetivo principal deste trabalho é a caracterização de primitivas de comunicação MPI através da elaboração de modelos analíticos, com vistas a análise e predição de desempenho destas primitivas. Neste intuito, os seguintes propósitos foram concretizados:

**1.** Desenvolvimento de extensões à metodologia de análise e predição de desempenho apresentada por Li [Li01], com a caracterização das seguintes operações de comunicação não tratadas:

- MPI_Bsend (modo *buffered* do *send* padrão);
- MPI_Ssend (modo síncrono do *send* padrão);
- MPI_Rsend (modo *ready* do *send* padrão);

- MPI_Barrier (operação de barreira);

**2.** Realização de novas análises e modelagens das seguintes operações já consideradas no trabalho de Li, procurando, contudo, aprimorar o trabalho desenvolvido e apresentar modelos mais robustos:

- MPI_Send (*send* padrão);
- MPI_Bcast (primitiva de comunicação coletiva *broadcast*);
- MPI_Reduce (operação coletiva de redução).

**3.** Identificação de técnicas e ferramentas capazes de oferecer suporte ao processo de análise e predição estabelecido pela metodologia, facilitando a sua aplicação e procurando automatizar este processo.

## 1.2 Motivação

A construção de aplicações paralelas e distribuídas para serem executadas sobre *clusters* de estações de trabalho tem sido estimulada pela necessidade de sistemas computacionais de alto desempenho e menores custos financeiros. Os *clusters* podem representar soluções viáveis para estas aplicações devido também a sua escalabilidade, que facilita a inserção e remoção de nós de processamento.

Porém, a exploração adequada das potencialidades destas máquinas de memória distribuída por vezes trata-se de uma tarefa nada trivial. A distribuição de tarefas entre os nós de processamento e a escolha da estratégia de paralelização a ser utilizada, por exemplo, podem determinar a eficiência de um certo programa. Neste contexto, as técnicas de modelagem analítica constituem um poderoso mecanismo para análise de desempenho de programas paralelos. Com a elaboração de modelos compostos por um conjunto de equações, podemos analisar o comportamento da aplicação e, além disso, realizar predições de tempos de execução para algoritmos ainda não implementados ou programas executando sobre condições variadas - alterando o número de nós de processamento envolvidos na computação ou o tamanho do problema tratado.

## 1.3 Justificativas

Recentemente, diversos pesquisadores têm utilizado técnicas de modelagem analítica em trabalhos de análise e predição de desempenho [Abandah96, Li01, Xu96]. No trabalho desenvolvido por Li, é proposta uma metodologia de análise e predição aplicável a programas paralelos que utilizem primitivas de comunicação MPI. Ainda, são apresentados modelos analíticos para algumas destas primitivas e testes realizados em um *cluster* de estações de trabalho.

Embora Li tenha tratado de algumas das operações de comunicação mais utilizadas em programas MPI, outras não foram envolvidas em sua pesquisa. Desta forma, neste trabalho realizamos uma extensão à metodologia proposta com o desenvolvimento de modelos analíticos para outras primitivas e repetimos a modelagem de algumas já tratadas, porém, buscando a execução de novas análises.

## 1.4 Metodologia

O desenvolvimento do presente trabalho envolveu a execução das seguintes atividades principais:

1. Pesquisa bibliográfica incluindo a revisão de assuntos e trabalhos relacionados;
2. Estudo do código da implementação LAM-MPI;
3. Testes com *benchmarks* e programas especialmente elaborados para medição de tempos de execução de primitivas MPI;
4. Elaboração de um conjunto de funções para auxiliar e automatizar atividades envolvidas no processo de análise e predição de desempenho;
5. Geração de modelos analíticos;
6. Realização de testes comprovando a validade dos modelos gerados.

## 1.5 Organização do trabalho

Este trabalho está organizado da seguinte maneira. O capítulo 2 aborda temas relacionados à programação paralela através de passagem de mensagens e o padrão MPI, discutindo características importantes dos *clusters* e questões envolvidas com a comunicação nestes sistemas, entre outros temas. São ainda comentadas algumas características e primitivas MPI. As principais técnicas e atividades envolvidas em um processo de análise e predição de desempenho são descritas no capítulo 3, incluindo a metodologia proposta por Li e, sucintamente, alguns trabalhos relacionados.

A análise e a modelagem das primitivas de comunicação MPI podem ser conferidas nos dois capítulos seguintes. Iniciamos o capítulo 4 comentando, entre outros assuntos, os procedimentos adotados nos processos de medição de desempenho e construção dos modelos analíticos, apresentando em seguida os resultados obtidos com as primitivas ponto-a-ponto. As comunicações coletivas são tratadas no capítulo 5, que aborda, além das primitivas previstas nos objetivos, uma análise da operação coletiva *all-to-all*. No último capítulo descrevemos as conclusões e sugerimos alguns temas a serem estudados em trabalhos futuros. Por fim, o Apêndice A apresenta um conjunto de funções desenvolvidas no intuito de oferecer suporte à realização de atividades de análise e predição.

# Capítulo 2

# PROGRAMAÇÃO COM TROCA DE MENSAGENS E MPI

## 2.1 Introdução

A necessidade de aplicações capazes de oferecer melhor desempenho tem impulsionado o desenvolvimento de tecnologias na área de computação paralela. Uma forma de melhorar este desempenho consiste na elaboração de técnicas e ferramentas adequadas às arquiteturas das máquinas paralelas, sejam elas de memória compartilhada ou de memória distribuída.

Através da correta exploração das características das máquinas paralelas e do paralelismo implícito nas aplicações, os pesquisadores procuram possibilitar a obtenção de menores tempos de execução. O aperfeiçoamento das bibliotecas de passagem de mensagem, por exemplo, pode ajudar projetistas de sistemas a explorarem melhor este modelo de programação e obterem programas mais eficientes.

Além disso, devemos considerar também a crescente necessidade de sistemas computacionais que combinem alto desempenho com menor custo financeiro. Neste caso, o emprego de *clusters* pode representar uma solução alternativa para o desenvolvimento de aplicações com requisito de alto desempenho ou alta disponibilidade. Tais máquinas podem ser construídas a partir de um conjunto de estações de trabalho interligadas por uma rede de alta velocidade. Sendo caracterizado como um sistema de memória distribuída, um *cluster* pode ter seu potencial

explorado adequadamente com a utilização de tecnologias baseadas em trocas de mensagens.

O modelo de programação baseado em passagem de mensagens requer o uso de uma biblioteca de funções oferecendo suporte à comunicação entre diferentes processos ou nós de processamento. Atualmente, o MPI (*Message Passing Interface*) [MPI Forum] tem se consolidado como um padrão para o desenvolvimento de aplicações baseadas em trocas de mensagens. O padrão MPI propõe uma biblioteca de funções específicas, permitindo a realização de diversas operações de comunicação, e possui diversas implementações disponíveis.

Neste capítulo discutimos diversos aspectos relacionados à programação paralela com passagem de mensagens - incluindo as principais características encontradas em *clusters*, os possíveis modos de comunicação e questões envolvidas com o desempenho do sistema durante as trocas de mensagens. São também apresentadas algumas características e primitivas de comunicação do padrão MPI.

## 2.2 Programação paralela através de passagem de mensagens

O projeto e a implementação de uma aplicação paralela costumam ser realizados segundo um modelo de programação, que por sua vez deve refletir as características da máquina a ser utilizada para executar a aplicação. Conforme [Culler99], nós podemos descrever os principais modelos de programação paralela da seguinte forma:

- **Endereços compartilhados**: as comunicações são executadas através de espaços de endereçamento compartilhados, cujo uso é análogo à utilização de um quadro de avisos onde as pessoas podem escrever e ler informações. Este modelo de programação é muitas vezes também denominado de memória compartilhada;

- **Passagem de mensagens**: na programação através de passagem de mensagens, um conjunto de eventos possibilita as trocas de informações entre

emissores e receptores específicos, não existindo uma memória compartilhada. Tais eventos constituem a base para harmonizar atividades individuais;

- **Processamento paralelo de dados**: neste modelo, mais conhecido como processamento vetorial, vários agentes executam uma ação simultânea sobre elementos distintos de um conjunto de dados. Após a execução da ação, os dados podem ser reorganizados através de acessos a endereços compartilhados ou trocas de mensagens.

Neste trabalho, estamos interessados em análise e predição de desempenho de programas paralelos MPI. Por isso concentramos as discussões sobre o modelo de programação baseado em trocas de mensagens, procurando caracterizar melhor este modelo e esclarecer aspectos mais relevantes. Um estudo com maior nível de detalhamento deste e dos demais modelos pode ser encontrado em [Culler99] ou em outras bibliografias relacionadas à programação paralela.

As operações de comunicação mais comuns em sistemas de passagem de mensagens são o *send* e o *receive*, incluindo suas diversas variantes possíveis. Como sugerem os nomes, o aparecimento de um *send* em um programa indica a realização de uma troca de mensagem entre dois processos, sob a condição de existir uma respectiva operação *receive* para a concretização da comunicação. A combinação destas operações determina um evento de sincronização entre os processos participantes e permite a cópia de dados do espaço de endereçamento do processo emissor para o espaço de endereçamento do receptor.

Existem diversas variantes possíveis para as operações *send/receive*. Dependendo de quando o *send* é finalizado em relação à execução do *receive*, ou do momento em que o *buffer* do emissor é disponibilizado para reuso ou, ainda, de quando a requisição é aceita, implementações distintas destas operações podem determinar diferentes eventos de sincronização [Culler99]. Assim, diferentes semânticas e requisitos de implementação são necessários de acordo com as possíveis variantes do *send/receive*.

### 2.2.1 Principais características dos *clusters*

Nos últimos anos os sistemas *clusters* têm sido vistos como soluções viáveis para a construção de aplicações paralelas e distribuídas, tanto na comunidade acadêmica como no meio empresarial. O uso de *clusters* na solução de problemas que exigem características como grande poder computacional e alta disponibilidade tem merecido destaque e ganho a atenção de diversos grupos de pesquisa. Isso se deve às vantagens apresentadas por esta arquitetura, dentre as quais podemos destacar:

- *Hardware* de baixo custo e fácil aquisição: *clusters* podem ser construídos a partir de estações de trabalho ou de computadores pessoais ligados através de uma rede de alta velocidade;
- Componentes de *software* padronizados: as bibliotecas de suporte à programação baseada em trocas de mensagens, como o MPI (*Message Passing Interface*) e o PVM (*Parallel Virtual Machine*), e o uso de sistemas operacionais não proprietários, como o Linux, facilitam e barateiam o desenvolvimento de aplicações para serem executadas em *clusters*;
- Escalabilidade: dependendo da aplicação, podemos facilmente inserir ou retirar máquinas do sistema computacional e obtermos significativos ganhos de desempenho. A inserção e remoção de nós de processamento muitas vezes não acarreta em grandes alterações nos programas ou na maneira como estes são desenvolvidos. Além disso, com um custo relativamente pequeno, podemos obter significativas melhorias no desempenho de programas paralelos adicionando novos nós de processamento ao *cluster*.

Segundo [Moura99], um *cluster* é caracterizado como um sistema de processamento paralelo ou distribuído, composto por um conjunto de computadores distintos trabalhando como um recurso computacional único e integrado. Dependendo da configuração das máquinas constituintes do sistema, este sistema de processamento pode ser dito homogêneo ou heterogêneo, sendo a primeira classificação atribuída àqueles sistemas compostos por máquinas idênticas, enquanto um sistema heterogêneo é formado por máquinas de configurações distintas. Desta

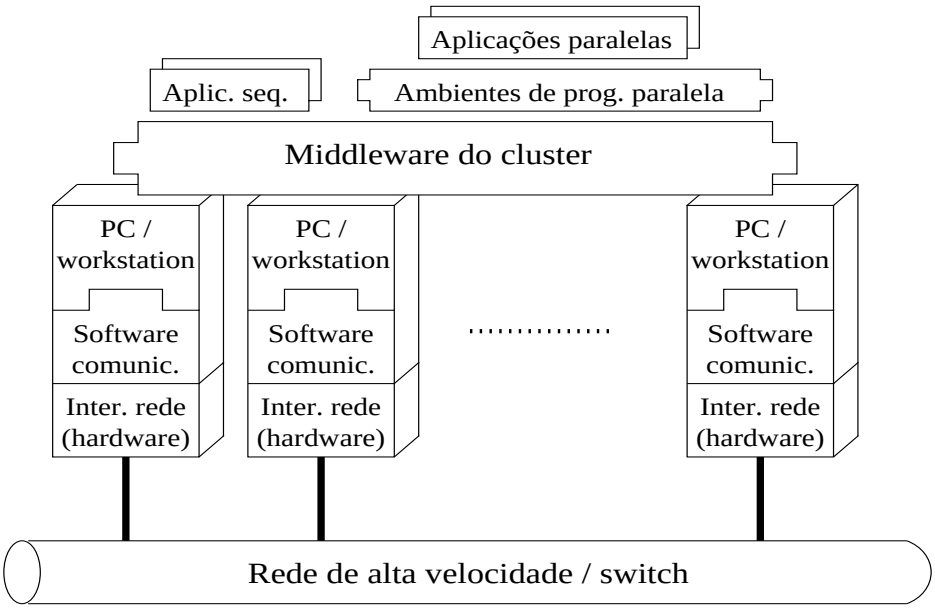


Figura 2.1: Arquitetura de um *cluster*.

forma, cada nó de processamento pode ser uma máquina mono ou multiprocessada com sua própria memória local, dispositivos de entrada e saída e um sistema operacional.

A Figura 2.1 [Moura99] ilustra as principais características da arquitetura de um sistema *cluster*. Como podemos observar, um conjunto de máquinas conectadas através de uma rede local de alta velocidade e um *switch* são utilizadas para executar aplicações seqüenciais e paralelas. Cada uma destas máquinas, por exemplo, um PC ou uma estação de trabalho, possui sua própria interface de rede, responsável por receber e transmitir pacotes de dados através da rede de interconexão. Garantir a fidelidade dos dados transmitidos é uma das funções do *software* de comunicação, que está diretamente ligado aos processos de empacotamento e desempacotamento das mensagens transmitidas.

A criação da imagem de um sistema de processamento integrado e único é uma característica comumente observada em *clusters*. Tal imagem é gerada com o suporte de um *middleware*, que não apenas possibilita o processamento paralelo

e distribuído mas também permite a execução de aplicações seqüenciais. Desta forma, as máquinas podem trabalhar como um sistema computacional único ou como computadores individuais.

Para possibilitar a execução de programas paralelos, é necessário ainda um ambiente de programação adequado. Estes ambientes podem envolver, por exemplo, bibliotecas de passagens de mensagens.

### 2.2.2 Implementação do modelo de passagem de mensagens

Na implementação das trocas de mensagens, os detalhes de movimentação de dados costumam ser escondidos do programador em uma biblioteca de passagem de mensagens. Uma camada de *software*, inserida entre as camadas das primitivas de comunicação e o *hardware* do sistema, possibilita ao programador construir aplicações paralelas sem precisar preocupar-se com pequenos detalhes envolvidos nas transferências de dados.

Em [Culler99], são discutidos alguns aspectos de baixo nível referentes à implementação das operações de trocas de mensagens, que apresentamos a seguir. Embora estas operações poderiam ser implementadas diretamente por *hardware*, suas características (como, por exemplo, bufferização) são melhor tratadas por implementação de *software*. Assim, em todas as máquinas paralelas, o modelo de programação através de passagem de mensagens é realizado com uma camada de *software* construída sobre uma abstração de comunicação mais simples. Primitivas mais básicas de transferência de dados podem ser utilizadas para implementar tal abstração, em uma solução suportada diretamente por *hardware*. Uma outra forma de implementação desta abstração seria considerarmos a adoção de um espaço de endereçamento virtual compartilhado, permitindo que as comunicações sejam realizadas através de operações de escrita e leitura em *buffers* compartilhados e envolvam eventos de sincronização apropriados [Culler99].

### 2.2.3 Sincronização e endereçamento

O modelo de programação através de passagem de mensagens tem sido bastante utilizado no desenvolvimento de aplicações paralelas, principalmente em máquinas de memória distribuída. Geralmente, todos os processos executam cópias

idênticas de um único programa [Culler99] e as trocas de mensagens constituem um poderoso mecanismo de sincronização entre processos cooperantes.

Um exemplo de sincronização implementada com a utilização de mecanismos de trocas de mensagens trata-se das operações *send/receive*. Cada par *send/receive* pode estabelecer um evento de sincronização em um determinado ponto do programa envolvendo os processos emissor e receptor. Operações de comunicação coletivas também podem permitir a sincronização de processos, dependendo da implementação realizada.

Conforme é discutido em [Culler99], em uma máquina de passagem de mensagens, um processador pode referenciar apenas endereços em sua memória local e cada um dos demais processadores. Assim, é permitido a um processo usuário acessar seus endereços privados e transferir dados usando primitivas de comunicação. Cada processo possui seu espaço de endereçamento privado, onde são realizadas as operações locais segundo a ordem de execução do programa.

### 2.2.4 Modos de comunicação

Conforme discuido anteriormente, as operações mais utilizadas em sistemas baseados em trocas de mensagens são o *send* e o *receive*. Devido ao fato de envolverem apenas dois processos, um enviando e outro recebendo dados, estas operações são chamadas **ponto-a-ponto**.

Em diversas situações surge a necessidade de um processo enviar ou receber dados de vários processos, como por exemplo no caso em que um mestre distribui informações ou recebe resultados de seus escravos. Operações de **comunicação coletiva** podem ser utilizadas com este intuito, permitindo transferências de dados nos sentidos um para vários, vários para um e vários para vários. A seguir listamos alguns exemplos de comunicações coletivas:

- **um-para-vários**: *broadcast* e *scatter*;
- **vários-para-um**: *gatter* e *reduce*;
- **vários-para-vários**: *all-to-all*.

As respectivas semânticas destes exemplos segundo o padrão MPI serão descritas na seção 2.4.

Uma comunicação *send*/*receive* pode ainda ser bloqueante ou não-bloqueante. Considerando uma operação de envio, por exemplo, costumamos caracterizá-la como bloqueante se o processo emissor for impedido de proceguir sua execução até que o *buffer* de dados utilizado para transmitir a mensagem possa ser reutilizado. O uso de operações bloqueantes pode levar a ocorrência de *deadlocks*, sendo responsabilidade do programador evitar tais situações, contudo as não-bloqueantes são menos seguras pois permitem o reuso de *buffers* que ainda estão sendo utilizados em uma comunicação. Uma diferenciação mais clara destes modos de comunicação é apresentada a seguir [MPI Forum]:

- **bloqueante**: o retorno da primitiva indica que o usuário pode reutilizar com segurança os recursos especificados na chamada (como *buffers*);
- **não bloqueante**: a primitiva pode retornar antes que a operação de comunicação complete e antes que o usuário possa reutilizar os recursos especificados na chamada.

### 2.2.5 Bibliotecas de suporte às trocas de mensagens

No modelo de programação paralela através de passagem de mensagens, costuma-se utilizar bibliotecas apropriadas como suporte à comunicação entre os processos. Bibliotecas ou interfaces de passagem de mensagens, como muitas vezes também são denominadas, possibilitam a construção de programas paralelos eficientes para sistemas de memória distribuída [Moura99].

Tais bibliotecas oferecem diversas rotinas para suportar a inicialização e finalização do ambiente e o envio e recebimento de pacotes de dados. Além das tradicionais comunicações *send* e *receive*, algumas variações destas operações costumam ser implementadas, disponibilizando formas de comunicação ponto-a-ponto com diferentes eventos de sincronização e, em alguns casos, introduzindo o uso de *buffers* para a execução das trocas de mensagens. As operações de comunicação coletiva e as barreiras constituem outros importantes mecanismos de trocas de informações oferecidos. Em síntese, podemos dizer que geralmente as bibliotecas de passagem de mensagem implementam operações ponto-a-ponto e

coletivas, bloqueantes e não-bloqueantes, conforme os modos de comunicação descritos na seção 2.2.4.

As interfaces mais utilizados na programação através de trocas de mensagens são o PVM (*Parallel Virtual Machine*) [Maui, Oak] e o padrão MPI (*Message Passing Interface*) [MPI Forum, Tennessee], definido pelo Fórum MPI. Ambas procuram oferecer ao programador o suporte necessário para o desenvolvimento de aplicações paralelas eficientes baseadas em trocas de mensagens e possuem implementações para as linguagens C, C++ e Fortran, além de Java cujo suporte têm sido desenvolvido [Pramanick99b]. Contudo, tarefas de paralelização como comunicação e sincronização entre processos, particionamento e distribuição de dados e mapeamento dos processos entre os processadores disponíveis, ainda ficam encarregadas ao programador [Moura99].

A maior desvantagem do PVM em relação ao MPI trata-se do desempenho, fato que por vezes estimula projetistas e pesquisadores a escolherem a segunda interface. Para conseguir maior flexibilidade, o PVM acaba sacrificando seu desempenho [Pramanick99a]. O melhor desempenho do MPI tem impulsionado o crescimento de sua popularidade e atualmente este é o padrão adotado pela maioria dos centros de pesquisa e fabricantes. Estando o presente trabalho interessado em análise e predição de desempenho de programas paralelos MPI, nas seções 2.3 e 2.4 nós descreveremos mais detalhes sobre o padrão e as primitivas de comunicação por ele definidas.

### 2.2.6 Desempenho das comunicações

Um dos fatores que influenciam fortemente o desempenho das aplicações paralelas baseadas em trocas de mensagens refere-se ao tempo despendido com as comunicações. O desempenho das comunicações, por sua vez, depende de diversos elementos constituintes do sistema computacional (como as características da rede de interconexão e das máquinas envolvidas). Nesta seção analisamos este desempenho, procurando identificar os principais componentes do tempo de comunicação.

A Figura 2.2 [Hennessy96] ilustra os parâmetros de desempenho de uma rede de interconexão. O *overhead* de envio e o *overhead* de recepção de uma mensa-

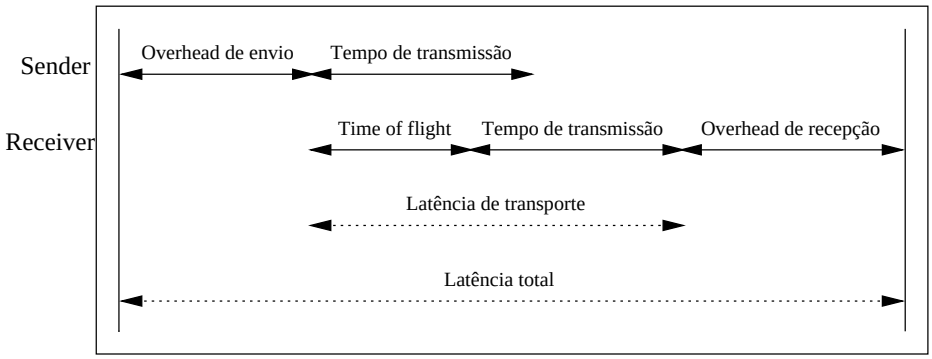


Figura 2.2: Parâmetros de desempenho de uma rede de interconexão.

gem indicam as frações de tempo em que os processadores envolvidos na comunicação, emissor e receptor respectivamente, ficam dedicados às tarefas de disponibilizar e retirar a mensagem da rede de interconexão. Durante este tempo os processadores não realizam outros processamentos.

O tempo necessário para que o primeiro bit alcance a interface de rede da máquina de destino é indicado pelo componente na figura denominado como *time of flight*. A soma deste tempo com o tempo de transmissão, durante o qual o restante da mensagem passa através da rede, constitui a latência de transporte. Na Figura 2.2 podemos observar melhor este comportamente, além de visualizar a latência total da comunicação.

Tratando-se do desempenho de redes de interconexão, um dos termos mais utilizados é a largura de banda (*bandwidth*). Geralmente medida em megabits por segundos (Mbits/s), a largura de banda refere-se à taxa máxima segundo a qual a rede pode propagar informação e está diretamente relacionada com o tempo de transmissão de uma mensagem. Assumindo que não existam outras mensagens concorrendo pelo uso da rede, o tempo de transmissão equivale ao tamanho da mensagem dividido pela largura de banda.

Definidos estes termos e componentes, podemos indicar a latência total de uma mensagem pela seguinte expressão [Hennessy96]:

$$Latência\ total = Overhead\ de\ envio + Time\ of\ flight + \frac{Tamanho\ da\ mensagem}{largura\ de\ banda} + Overhead\ de\ recepção$$

Um erro por vezes observado em projetos de avaliação de desempenho trata-se da adoção da largura de banda como a única medida de desempenho da rede.

Porém, conforme é comentado em [Hennessy96], para muitas aplicações e redes, a maior influência na latência de comunicação de uma mensagem é exercida sobretudo pelos *overheads* de envio e recepção.

## 2.3 Características básicas do padrão MPI

Até alguns anos atrás o desenvolvimento de aplicações paralelas e distribuídas baseadas em trocas de mensagens não se mostrava uma solução atraente aos fabricantes de *software*. Um dos motivos disto relaciona-se à diversidade de bibliotecas disponíveis que implementavam o modelo de programação baseado em passagem de mensagens, sem contudo existir um mesmo padrão aceito e utilizado pelos projetistas.

Neste contexto, o PVM e o padrão MPI propiciaram novos impulsos à disseminação do uso de passagem de mensagens na construção de aplicações para máquinas de memória distribuída. O padrão MPI, criado mais recentemente, possibilita a diferentes fabricantes de *software* utilizar um mesmo conjunto de primitivas de comunicação que, embora possam apresentar diferentes implementações, são definidas segundo uma mesma sintaxe e semântica. As implementações do padrão obviamente possuem características internas distintas, porém disponibilizam a mesma interface de passagem de mensagens.

O MPICH [Gropp] e o LAM [Ohio] são as mais conhecidas implementações do padrão MPI. Em [Nupairoj94], são apresentados os resultados de alguns testes realizados em um *cluster* de estações de trabalho, utilizando um conjunto de *benchmarks* desenvolvido para avaliar operações de comunicação ponto-a-ponto e coletivas. Foram analisadas quatro implementações distintas (CHIMP, LAM, MPICH e Unify) e, dentre as conclusões discutidas, podemos destacar a escolha do LAM como a melhor implementação disponível considerando o ambiente de testes utilizado.

No entanto, variações significativas de desempenho podem ou não ser verificadas entre implementações diferentes, dependendo de fatores como as métricas adotadas ou as características do ambiente de teste. As próprias versões analisadas constituem um outro importante fator a ser considerado na comparação entre implementações do padrão MPI. Para o desenvolvimento do presente trabalho nós

Tabela 2.1: Tipos de dados MPI e seus correspondentes na linguagem C.

| MPI | C |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short |
| MPI_INT | signed int |
| MPI_LONG | signed long |
| MPI_UNSIGNED _CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_BYTE | - |
| MPI_PACKED | - |

enfatizamos os estudos na implementação LAM, procurando com isso elaborar modelos de predição de desempenho mais precisos. Desta forma, na seção 2.4, descrevemos algumas primitivas de comunicação MPI incluindo detalhes que possivelmente apresentem pecularied ades distintas em outras implementações, devido a características internas diferentes.

### 2.3.1 Tipos de dados

A comunicação de uma mensagem MPI envolve a transmissão de uma determinada quantidade de dados de um tipo pré-definido. Desta foma, em uma operação de comunicação o usuário necessita indicar o número de elementos transmitidos e não a respectiva quantidade de bytes da mensagem, o que torna esta tarefa independente das caracteríticas da máquina utilizada, pois não é requerido o conhecimento do tamanho em bytes dos elementos envolvidos na comunicação. A Tabela 2.1 apresenta os principais tipos de dados MPI e seus correspondentes na linguagem C [MPI Forum].

Os dois últimos tipos apresentados na Tabela 2.1 não possuem correspondentes na linguagem C. Enquanto cada elemento de dado MPI_BYTE é composto exatamente por um 1 byte, o MPI_PACKED na realidade constitui um tipo de

dados derivado e permite ao usuário empacotar elementos homogêneos não contínuos em um único *buffer* que será enviado. Para o empacotamento e posterior desempacotamento dos dados, realizado após a recepção da mensagem, o padrão MPI apresenta as respectivas operações MPI_pack() e MPI_unpack(). Maiores informações sobre outros tipos de dados derivados podem ser encontrados na definição do padrão MPI [MPI Forum].

### 2.3.2 *Communicators*, contextos de comunicação e grupos de processos

Segundo a definição do padrão MPI [MPI Forum], os seguintes requisitos são esperados em uma biblioteca paralela robusta:

- criação de um espaço de comunicação seguro, garantindo a execução de operações de comunicação sem que ocorram conflitos com comunicações não relacionadas à biblioteca;
- permitir o uso de escopo de grupo para facilitar a realização de comunicações coletivas;
- identificação dos processos usuários através de nomes abstratos;
- permitir ao usuário ou construtor da biblioteca estender notações do modelo de passagem de mensagens.

Com vistas à satisfação destes requisitos, o padrão MPI inclui entre suas definições os conceitos de *communicators*, contextos de comunicação e grupos de processos. Em um programa MPI, uma operação de comunicação é executada dentro de um determinado contexto, que é especificado por um certo *communicator* e indica os possíveis processos receptores da mensagem. A seguir apresentamos uma breve descrição de cada um destes conceitos [MPI Forum].

#### Grupos de processos

Um grupo de processos constitui uma coleção ordenada de processos, onde cada um destes possiu uma identificação única (um número inteiro, denominado *rank*,

iniciando em 0). Um processo pode pertencer a mais de um grupo e, neste caso, possuir diferentes *ranks*.

#### Contextos de comunicação

O conceito de contexto permite particionar o espaço de comunicação, de forma que a operação de envio de uma mensagem possa ser direcionada a um grupo específico de processos. Mensagens enviadas dentro de um contexto não podem ser recebidas em outro. Além disso, convém ressaltar que contextos não constituem objetos MPI, mas precisam ser indicados em todas as primitivas de comunicação através de um parâmetro denominado *communicator*.

#### *Communicators*

Um *communicator* trata-se de um objeto MPI que provê o escopo apropriado para a execução de uma operação de comunicação MPI. Tal operação pode ser executada dentro de um único grupo de processos, utilizando um *intra-communicator*, ou pode constituir uma comunicação ponto-a-ponto envolvendo dois grupos distintos (indicada pelo uso de um *inter-communicator*). Podemos citar como exemplos de *intra-communicators* pré-definidos pelo padrão o MPI_COMM_WORLD e o MPI_COMM_SELF, sendo a abrangência do primeiro todos os processos envolvidos na execução do programa MPI e a do segundo apenas o próprio processo.

## 2.4 Primitivas de comunicação MPI

### 2.4.1 Comunicação ponto-a-ponto

O padrão MPI disponibiliza um conjunto de primitivas para realizar operações de comunicação ponto-a-ponto bloqueantes e não-bloqueantes (Tabela 2.2). Tais primitivas diferenciam-se principalmente por implementarem diferentes mecanismos de sincronismo entre os processos envolvidos (o receptor e o emissor), ou pelo uso de *buffers* no processo de transmissão da mensagem.

Como podemos observar na Tabela 2.2, os parâmetros utilizados nas primitivas bloqueantes (MPI_Send, MPI_Bsend, MPI_Rsend e MPI_Ssend) são exa-

Tabela 2.2: Operações de comunicação ponto-a-ponto MPI.

| Operação | Primitiva MPI |
|---|---|
| Send padrão | MPI_Send(&buf, count, dtype, dest, tag, comm); |
| Send buffered | MPI_Bsend(&buf, count, dtype, dest, tag, comm); |
| Send ready | MPI_Rsend(&buf, count, dtype, dest, tag, comm); |
| Send síncrono | MPI_Ssend(&buf, count, dtype, dest, tag, comm); |
| Send não bloqueante | MPI_Isend(&buf, count, dtype, dest, tag, comm, &req); |
| Receive bloqueante | MPI_Recv(&buf, count, dtype, source, tag, comm, &status); |
| Receive não bloqueante | MPI_Irecv(&buf, count, dtype, source, tag, comm, &req); |

tamente os mesmos. Em uma operação de comunicação ponto-a-ponto MPI, o processo emissor envia um certo número de elementos (count) de um determinado tipo (dtype) para o processo receptor (dest). Os dados a serem enviados devem estar no *buffer* de envio buf e a variável dest identifica o receptor dentro do grupo de processos indicado pelo *communicator* comm. Existe ainda um outro importante parâmetro, chamado tag, que pode ser usado como uma identificação da mensagem transmitida. A seguir podemos visulizar melhor estes parâmetros em uma chamada da primitiva MPI_Send:

```
MPI_Send(void * buf, int count, MPI_Datatype dtype,
         int dest, int tag, MPI_Comm comm)
```

Sendo o nosso enfoque neste trabalho as formas de comunicação bloqueante, a seguir apresentamos uma descrição dos quatro modos de *send* bloqueantes (*standard*, *buffered*, *synchronous* e *ready*). Informações sobre as primitivas MPI não-bloqueantes e suas respectivas semânticas podem ser encontradas nas referências do padrão MPI .

***Standard send***

O *send* padrão trata-se do modo de comunicação mais utilizado para o envio de uma mensagem entre dois processos MPI. A operação é dita bloqueante pois, ao ser executada, ela apenas poderá retornar após a mensagem ter sido armazenada com segurança, sendo então permitido ao processo emissor reutilizar o *buffer* de envio [MPI Forum]. Ainda segundo as definições do Fórum MPI, a mensagem pode ser transmitida diretamente para o *buffer* do receptor ou simplesmente ser copiada em um *buffer* de sistema, sendo realmente transferida ao seu destino em um segundo momento. É de responsabilidade do MPI decidir quando os dados envolvidos em uma comunicação são ou não armazenados localmente; por isso, podemos dizer que a primitiva de comunicação pode ou não bloquear dependendo de sua implementação.

Conforme comentado em [Al-Tawil01], o comportamento do *send* padrão não é definido precisamente pelo MPI. Assim, se pretendemos compreender o que realmente ocorre durante a transmissão de uma mensagem neste modo de comunicação, precisamos analizar as particulariedades da implementação MPI utilizada. Após estudarmos o código da implementação LAM, verificamos que a primitiva MPI_Send pode ou não bloquear dependendo, dentre outros fatores, do tamanho da mensagem transmitida e do número de mensagens pendentes a serem recebidas pelo processo de destino. Assim, é possível que ao executarmos uma primitiva MPI_Send em um programa LAM o emissor encerre a operação antes da transferência dos dados pela rede.

***Buffered send***

A principal diferença do *buffered send* em relação aos demais modos de *send* está no uso de um *buffer* de dados para onde a mensagem é copiada antes de ser transmitida. O usuário deve criar este *buffer* explicitamente, através de funções específicas disponibilizadas pelo MPI, antes de chamar a primitiva MPI_Bsend. É responsabilidade do usuário definir um *buffer* suficientemente grande para alojar a mensagem transmitida e cuidar para que este não seja reutilizado indevidamente, evitando a ocorrência de erros ou a sobreposição de dados.

O tempo despendido pelo processo emissor em uma comunicação no modo

*buffered* tende a ser menor em relação aos demais *sends* bloqueantes. Isso acontece pois, ao contrário das operações MPI_Send, MPI_Ssend e MPI_Rsend, a ocorrência de uma primitiva MPI_Bsend em um programa não exige a existência de uma operação de recepção (MPI_Recv) para a sua correta finalização. A mensagem é simplesmente copiada para o *buffer*, de onde será transmitida ao respectivo receptor, e o processo emissor pode continuar sua execução normalmente. Por outro lado, a necessidade da alocação deste *buffer* gera um *overhead* que pode ser significativo dependendo do tamanho da mensagem. Assim, caso o objetivo do usuário ao utilizar o MPI_Bsend seja obter melhor desempelho, talvez uma alternativa melhor possa ser oferecida pelo *send* não-bloqueante.

#### *Synchronous send*

A operação MPI_Ssend pode ser inicializada com ou sem a ocorrência do respectivo MPI_Recv, mas somente poderá ser finalizada quando o receptor começar a receber a mensagem enviada [Tennessee]. Desta forma, o MPI garante que ao final da execução da operação de envio o processo receptor atingiu um certo ponto de sua execução e o *buffer* do emissor pode ser reutilizado [MPI Forum].

#### *Ready send*

No modo *ready* do *send* bloqueante a operação de comunicação somente terá início quando houver um receptor esperando a mensagem. Quando ocorre uma chamada da primitiva MPI_Rsend, o MPI verifica a existência da respectiva operação MPI_Recv e, caso a resposta seja negativa, a comunicação fica aguardando. Caso contrário, a transmissão da mensagem é estabelecida e uma vez encerrada a primitiva será finalizada.

Convém notarmos a diferença entre os eventos de sincronismo estabelecidos pelos modo síncrono e *ready*. Enquanto no primeiro temos a transferência dos dados e em seguida a sincronização dos processos, no *ready* os processos devem estar sincronizados antes do início da transmissão da mensagem em si.

Tabela 2.3: Principais operações de comunicação coletiva MPI.

| Operação | Primitiva MPI |
|---|---|
| *Broadcast* | MPI_Bcast(&buf, count, dtype, root, comm); |
| *Scatter* | MPI_Scatter(&sendbuf, sendcount, sendtype, &recv-buf, recvcount, recvtype, root, comm); |
| *Gather* | MPI_Gather(&sendbuf, sendcount, sendtype, &recv-buf, recvcount, recvtype, root, comm); |
| *Reduce* | MPI_Reduce(&sendbuf, &recvbuf, count, dtype, op, root, comm); |
| Barreira | MPI_Barrier(comm); |
| *All-to-all* | MPI_Alltoall(&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm); |

***Receive bloqueante***

Os quatro modos de comunicação ponto-a-ponto bloqueante discutidos nesta seção requerem a mesma primitiva para a recepção dos dados, apresentada abaixo:

```
MPI_Recv(void * buf, int count, MPI_Datatype dtype, int
   source, int tag, MPI_Comm comm, MPI_Status * status)
```

Conforme é comentado na definição do padrão MPI [MPI Forum], a primitiva MPI_Recv não exige a finalização do respectivo *send* para completar sua execução. Porém, é claro que ela somente terá início após a inicialização da operação *send*.

### 2.4.2 Comunicação coletiva

As primitivas descritas a seguir oferecem suporte às comunicações coletivas em um programa MPI. São classificadas como operações coletivas aquelas envolvendo um grupo de processos [MPI Forum]. Na Tabela 2.3 podemos conferir algumas destas operações e as primitivas MPI correspondentes.

Enquanto nas operações ponto-a-ponto o emissor e o receptor precisam realizar chamadas de primitivas distintas (por exemplo MPI_Ssend e MPI_Recv, respectivamente), nas operações coletivas os processos envolvidos utilizam a mesma primitiva. A distinção entre qual é o processo emissor (ou emissores) e quais são

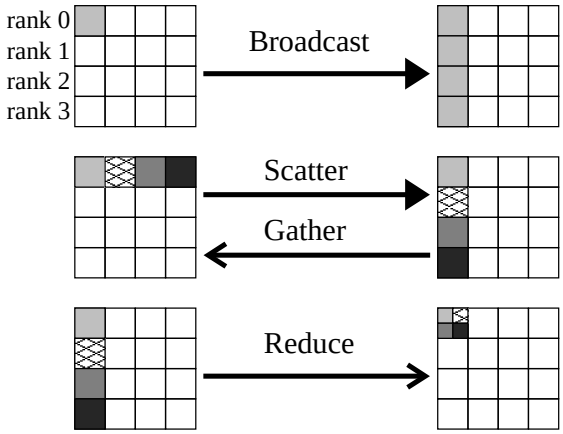


Figura 2.3: Operações de comunicação coletiva.

os receptores (ou receptor) é dada pelo parâmetro root, segundo a semântica da operação. O grupo de processos para o qual a operação coletiva é direcionada, por sua vez, é indicado através do parâmetro comm, especificando o *communicator* utilizado.

Conforme é discudo na definição do padrão [MPI Forum], a finalização de uma primitiva coletiva indica que o *buffer* utilizado na comunicação pode ser acessado com segurança, embora não necessariamente todos os processos envolvidos tenham completado a operação (ou até mesmo a iniciado). Esta informação não é válida para a operação MPI_Barrier, onde ocorre explicitamente a sincronização dos processos. Porém, para os demais casos, sugere que podemos ter ou não ter um evento de sincronismo associado à ocorrência da comunicação coletiva.

Nas subsessões seguintes nós descrevemos as principais operações coletivas MPI, listadas na Tabela 2.3. As respectivas semânticas do *broadcast*, do *scatter*, do *gather* e do *reduce* podem ainda ser observadas na Figura 2.3 [Ohio].

### *Broadcast*

A operação *broadcast*, cuja primitiva MPI é apresentada abaixo, caracteriza-se como uma comunicação coletiva do tipo um-para-vários, onde o processo root envia uma mesma mensagem a todos os processos do grupo, inclusive ele mesmo. Assim como ocorre com as demais operações coletivas, é comum o uso de operações ponto-a-ponto em sua implementação e a forma de distribuição dos dados

pode variar dependendo do algoritmo utilizado.

```
MPI_Bcast(void * buf, int count, MPI_Datatype dtype,
          int root, MPI_Comm comm)
```

### *Scatter* e *gather*

Nas operações *scatter* e *gather* as mensagens transmitidas são diferentes, ao contrário do que ocorre com o *broadcast*. As comunicações são classificadas, respectivamente, como um-para-vários e vários-para-um. Quando o MPI_Scatter é executado, o root divide a quantidade de dados sendcount pelo número de processos pertencentes ao grupo e então envia mensagens diferentes a todos os participantes do grupo, inclusive a ele mesmo [Ohio]. Para os demais processos, os parâmetros do emissor (sendbuf, sendcount e sendtype) não são significantes.

A semântica da operação *gather* é o oposto do *scatter*. Assim, cada processo envia sendcount elementos do tipo sendtype ao root, que recebe os dados e os armazena segundo a ordem dos *ranks*. Convém ressaltar que tanto no MPI_Scatter como no MPI_Gather os tipos denotados pelos parâmetros sendtype e recvtype devem ser compatíveis e o número de elementos enviados é sempre igual ao de recebidos.

Ambas as primitivas requerem exatamente os mesmos parâmetros. Abaixo apresentamos suas respectivas definições:

```
MPI_Scatter(void * sendbuf, int sendcount, MPI_Datatype
        sendtype, void * recvbuf, int recvcount,
        MPI_Datatype recvtype, int root, MPI_Comm comm)
MPI_Gather(void * sendbuf, int sendcount, MPI_Datatype
        sendtype, void * recvbuf, int recvcount,
        MPI_Datatype recvtype, int root, MPI_Comm comm)
```

### *Reduce*

A primitiva MPI_Reduce estabelece uma comunicação coletiva do tipo vários-para-um, onde o processo root recebe dados enviados pelos demais integrantes do grupo e realiza uma operação de redução sobre estes dados. Todos os processos devem especificar a mesma operação de redução através do parâmetro op. Alguns

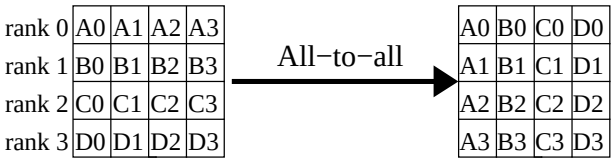


Figura 2.4: Operação coletiva *all-to-all*.

exemplos são: calcular a soma ou o produto dos elementos, encontrar o mínimo ou o máximo e as operações lógicas *AND* e *OR*.

```
MPI_Reduce(void * sendbuf, void * recvbuf, int count,
    MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)
```

**Barreira**

Uma barreira pode ser criada explicitamente em um programa MPI através da primitiva MPI_Barrier, permitindo a sincronização de todos os processos do grupo. Ao executar a primitiva, o processo fica bloqueado até que todos os processos do grupo especificado pelo *communicator* comm tenham realizado uma chamada a esta primitiva.

```
MPI_Barrier(MPI_Comm comm)
```

***All-to-all***

O *all-to-all* constitui uma operação de comunicação vários-para-vários e, em sua execução, cada processo envia partes distintas de um conjunto de dados para os demais integrantes do grupo (Figura 2.4 [MPI Forum]). A quantidade de elementos que compõe a mensagem é dividida pelo número de participantes da comunicação ($p$) e, em seguida, cada uma destas partes é destinada ao *buffer* de um certo receptor. Desta forma, todo processo envia e recebe $p - 1$ diferentes mensagens fazendo uso da seguinte primitiva:

```
MPI_Alltoall(void * sendbuf, int sendcount,
      MPI_Datatype sendtype, void * recvbuf, int
      recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

## 2.5 Considerações finais

Neste capítulo, discutimos alguns pontos importantes do modelo de programação baseado em passagem de mensagens. As caracteríticas dos *clusters* e detalhes da implementação do modelo foram descritos, assim como questões relacionadas ao desempenho das operações de comunicação.

Em uma segunda parte do texto, abordamos o padrão MPI e seus modos de comunicação. Dentre as primitivas disponibilizadas pelo padrão, descrevemos a sintaxe e a semântica das principais operações ponto-a-ponto e coletivas.

# Capítulo 3

# ANÁLISE E PREDIÇÃO DE DESEMPENHO

## 3.1 Introdução

A busca por melhores desempenhos para aplicações paralelas tem impulsionado o desenvolvimento de técnicas e metodologias visando reduzir os tempos de execução destas aplicações. Tratando-se especificamente de sistemas baseados em passagem de mensagens, tanto o tempo gasto com computações locais como também a influência das comunicações no desempenho da aplicação devem ser considerados.

Para obtermos programas paralelos eficientes, é importante explorar corretamente as características das arquiteturas paralelas e o paralelismo implícito nas aplicações. Neste intuito, as pesquisas nas áreas de análise e predição de desempenho podem auxiliar os projetistas a identificar possíveis razões de tempos de execução ruins ou fornecer parâmetros para a elaboração de códigos mais eficientes.

Contudo, o desenvolvimento de atividades relacionadas à análise e predição de desempenho deve atender a certos cuidados e utilizar técnicas apropriadas. Aspectos como, por exemplo, a escolha das técnicas a serem utilizadas e a correta aplicação destas constituem pontos essenciais para o sucesso do processo de análise e predição.

Neste contexto, o presente capítulo aborda diversos pontos importantes em estudos de análise e predição de desempenho. Discutimos, entre outros tópicos, os principais erros cometidos na avaliação de desempenho, as atividades envolvidas nestes estudos e alguns conceitos envolvidos. São ainda descritos alguns trabalhos relacionados.

## 3.2 Fatores a serem considerados

Os processos de análise e avaliação de desempenho costumam ser realizados com objetivos de obtenção de explicações para desempenhos ruins ou a melhoria destes. Contudo, por causa de erros cometidos neste processo ou devido a não utilização correta de técnicas apropriadas, muitas vezes os resultados alcançados mostram-se incapazes de atingir tais objetivos.

Por isso é importante procurarmos conhecer quais são os erros mais cometidos nestes processos, tentando aprender como evitá-los. Com este intuito, nas subseções a seguir comentamos alguns destes erros e discutimos técnicas que podem auxiliar a realização de atividades envolvidas não somente com análise e avaliação, mas também com predição de desempenho.

### 3.2.1 Erros cometidos na avaliação e predição de desempenho

Em [Jain91] são identificados diversos erros freqüentemente observados em projetos de avaliação de desempenho. Dentre tais erros, podemos destacar os seguintes:

#### Falta de objetivos

É fundamental para um bom projeto identificar corretamente seus objetivos. Atividades como definição de métricas, caracterização do *workload* e escolha da metodologia dependem destes objetivos, o que salienta a importância de se defini-los nas fases iniciais do projeto. Além disso, os objetivos traçados devem ser claros e não tendenciosos, evitando-se nortear as atividades da avaliação de desempenho com o intuito de se alcançar conclusões pré-determinadas (por exemplo, o objetivo de se provar que um sistema é melhor que outro pode induzir a adoção de certas métricas em detrimento de outras, conduzindo a conclusões errôneas).

**Falhas na definição de parâmetros, métricas e *workloads***

As métricas, os parâmetros e o *workload* constituem importantes elementos em um projeto de avaliação de desempenho. Muitas vezes os analistas falham na adoção de critérios adequados para a identificação destes elementos, definido-os arbitrariamente.

As métricas adotadas (tempo de resposta, por exemplo) indicam o critério usado para caracterizar o desempenho do sistema, enquanto os parâmetros podem ser vistos como características do sistema e do *workload* que afetam o desempenho (por exemplo, o número de usuários) [Jain91]. Um erro comum na seleção de métricas e parâmetros é escolher aqueles mais fáceis de serem medidos, embora não sejam os mais relevantes.

Segundo [Oed81], o *workload* de um sistema constitui as tarefas, transações e dados a serem processados em um determinado período de tempo, ou simplesmente a demanda de trabalho requerida pelos usuários. Desta forma, podemos dizer que ele representa o volume de trabalho ao qual o sistema é submetido, como, por exemplo, o fluxo de pacotes enviados ou recebidos através de uma rede. Considerando este exemplo, o *workload* usado na comparação entre redes distintas deve consistir de pacotes de tamanhos semelhantes aos trocados habitualmente em aplicações reais.

**Escolha equivocada da técnica de avaliação**

A escolha da técnica a ser utilizada para analisar o desempenho do sistema precisa considerar, entre outros aspectos, o estágio de desenvolvimento em que se encontra o sistema, o tempo disponível para realização das atividades de análise, a precisão desejada e o custo envolvido [Jain91]. Contudo, é comum um analista preferir uma determinada técnica por desconhecer ou saber pouco sobre outras técnicas possíveis, ignorando estes aspectos. Algumas das principais técnicas utilizadas para avaliação e predição de desempenho são discutidas na seção 3.4.

Um outro fator importante a ser considerado na adoção de uma técnica refere-se a sua complexidade. Quando duas análises distintas apresentam os mesmos resultados, é mais conveniente optar por aquela mais simples [Jain91]. A simplicidade de um modelo tende a facilitar sua implementação e, em outro extremo,

objetivos difíceis de serem alcançados ou muito ambiciosos podem inviabilizar projetos de medição e de simulação.

**Falhas na análise dos resultados**

Não basta coletar grandes quantidades de dados de desempenho, é preciso saber interpretar estas informações para que o processo de avaliação produza resultados concretos. Além disso, não devemos ignorar a influência de parâmetros do sistema e do *workload* em questão, evitando-se desta forma conclusões precipitadas sobre a validade da aplicação dos resultados obtidos em ambientes com características diferentes.

### 3.2.2 Considerações sobre a construção de modelos

Modelos podem ser definidos como coleções de atributos e regras que regem as interações sobre estes atributos [Mullender93]. A criação de modelos permite abstrair as características fundamentais de um certo sistema, representando tal sistema de forma organizada e facilitando estudos de análise e predição de desempenho.

Conforme é discutido em [Meira95], modelos provêem uma maneira estruturada para compreender problemas de desempenho, além de possuir habilidade de predição, permitindo estimar os efeitos sobre o desempenho causados por diferentes paralelizações ou variações no ambiente de execução. Aumentando ou reduzindo o número de atributos e regras tratadas é possível construir diferentes modelos para uma mesma aplicação e, desta forma, variar o nível de abstração adotado. O nível de abstração, que indica o grau de detalhamento da representação, em muitos casos é ainda um fator determinante para a complexidade do modelo. Geralmente, um número maior de parâmetros tratados indica a elaboração de um modelo mais preciso e de mais baixo nível, ao passo que representações de mais alto nível de abstração costumam apresentar-se mais simples porém menos precisas.

Contudo, construir um modelo capaz de representar adequadamente um sistema complexo não é uma tarefa trivial. Abstrações de sistemas paralelos devem ser simples o suficiente para serem tratáveis e suficientemente detalhadas para

serem precisas [Meira95]. A elaboração do modelo exige compreensão do sistema analisado e pode priorizar algumas características em detrimento de outras, dependendo dos objetivos que o processo de modelagem se propõe a alcançar.

Segundo [Hu97], dois princípios precisam ser satisfeitos por um bom modelo: ele deve descrever corretamente o comportamento do sistema, incluindo todos os detalhes necessários que definem este comportamento, e deve ainda ser tão simples quanto possível. Assim, na construção de um modelo precisamos procurar minimizar o número de parâmetros envolvidos, facilitando a sua aplicação sem comprometer a sua representatividade. A inclusão de novos parâmetros tende a melhorar a acurácia do modelo, porém, um número elevado deles pode tornar o modelo intratável.

### 3.2.3 Seleção de métricas e caracterização do *workload*

A seleção de métricas e a caracterização do *workload* são duas etapas importantes em trabalhos de análise e predição de desempenho. Como foi mencionado anteriormente, a execução destas etapas depende da prévia identificação dos objetivos do projeto. Falhas na definição dos objetivos, como aquelas discutidas anteriormente, podem prejudicar a seleção de métricas mais convenientes ou comprometer a correta caracterização do *workload*.

Em estudos de desempenho devemos primeiramente selecionar as métricas a serem adotadas, considerando a situação ou problema em particular a ser resolvido [Hu97]. Exemplos de métricas geralmente utilizadas incluem o tempo de resposta, o número de requisições de serviços executadas por unidade de tempo (*throughput*) e a utilização de um recurso (fração de tempo em que o recurso está ocupado tratando as requisições) [Jain91].

A definição do *workload* também é uma tarefa bastante dependente das particulariedades do sistema analisado. Diferentes sistemas costumam exigir *workloads* variados, como, por exemplo, um determinado número de instruções de adição a serem executadas ou um certo conjunto de funções representando operações características de uma aplicação particular (uso de *benchmarks*) [Jain91].

Em muitos casos o *workload* real do sistema é bastante complexo e difícil de ser caracterizado, criando a necessidade de se utilizar modelos de *workload*. Tais

modelos devem satisfazer os seguintes requisitos [Hu97]:

- o modelo deve executar as mesmas funções do *workload* real, de forma proporcional e utilizando os mesmos recursos do sistema;
- ele deve suportar modificações e reproduções, facilitando sua utilização em diferentes estudos;
- para que o *workload* possa ser portável a outros sistemas, o modelo precisa ainda ser compacto.

## 3.3 Medição, análise e predição de desempenho

Não é comum encontrarmos na literatura muitos trabalhos preocupados em estabelecer distinções claras entre medição, análise e predição de desempenho. Por vezes os pesquisadores desenvolvem trabalhos agregando atividades destas três áreas (como, por exemplo, é comum a utilização de medições como apoio à análise e predição de desempenho).

Em [Crovella94] é apresentada a seguinte classificação para os trabalhos relacionados a estas áreas:

- Ferramentas de medição de desempenho e técnicas para programas paralelos;
- Técnicas de análise de desempenho para programas paralelos;
- Técnicas de predição de desempenho para programas paralelos;

Enquanto os trabalhos classificados nas duas primeiras áreas geralmente não oferecem subsídios suficientes para predizer o desempenho dos programas, as técnicas de predição procuram prever tempos de execução considerando programas e máquinas específicas. Segundo [Crovella94], uma técnica de predição de desempenho deve incluir um componente de análise estática, que considera a estrutura do programa, e um componente de análise dinâmica, capaz de obter os custos de

execução deste programa para um determinado ambiente e uma máquina específica. Para obter estes valores, a modelagem dinâmica pode fazer uso de técnicas de medição.

As ferramentas de medição provêm informações referentes ao desempenho da aplicação, como tempos de execução, procurando assim indicar as possíveis razões de desempenhos ruins. Por outro lado, a análise de desempenho costuma apresentar estudos assintóticos, avaliando as variações do desempenho frente a alterações no tamanho do problema e no número de processadores envolvidos na execução de uma aplicação.

## 3.4 Atividades e abordagens empregadas na análise e predição de desempenho

As atividades envolvidas na análise e predição de desempenho de sistemas paralelos podem variar dependendo dos objetivos traçados inicialmente. Tais atividades costumam utilizar técnicas para construção de modelos, simulações e medições. Conforme discutimos nesta seção, a escolha de uma ou outra técnica pode ainda depender do estágio de desenvolvimento em que se encontra a aplicação.

Medições são obtidas a partir de execuções da aplicação e, portanto, sua utilização exige que o sistema esteja ao menos parcialmente implementado. Instrumentando o código fonte de uma aplicação, por exemplo, podemos extrair tempos de execução e realizar avaliações de desempenho a partir dos valores medidos. Os cuidados a serem observados em atividades de medição de desempenho incluem a correta seleção dos parâmetros de entrada e saída e a forma como os parâmetros de saída deveriam ser medidos [Kant92].

Por vezes é comum o uso de técnicas de simulação para a análise de desempenho de sistemas ainda não implementados. Nestes casos, a simulação pode ser utilizada para avaliar a viabilidade computacional de uma aplicação ou comparar possíveis implementações. Também é comum o uso de simulações quando é necessário estudar parâmetros de desempenho difíceis de serem obtidos através de instrumentação, por exemplo, devido a limitações de *cache* [Meira95]. Contudo, uma das desvantagens que restringe o uso de tais técnicas refere-se ao custo

da simulação. Simulações de sistemas paralelos complexos costumam demandar muito tempo em máquinas seqüenciais [Hu97].

Para a simulação de um sistema, costuma-se construir um modelo para o comportamento do sistema e direcioná-lo com uma abstração apropriada do *workload* [Kant92]. Ainda segundo [Kant92], os seguintes fatores devem ser considerados: (i) o que simular e o nível de detalhe (isso terá forte influência no custo da simulação); (ii) pode ser necessário utilizar técnicas estatísticas para analisar as grandes quantidades de dados gerados; e (iii) um projeto experimental cuidadoso é essencial para manter o custo da simulação baixo.

A construção de modelos muitas vezes também é realizada antes da implementação da aplicação. Contudo, as técnicas de modelagem são bastante úteis na avaliação de desempenho de programas total ou parcialmente implementados e têm sido objeto de estudo de diversos pesquisadores. A metodologia utilizada em nosso trabalho para realizar predições de desempenho de programas, por exemplo, é baseada no uso de técnicas de medição para obtenção de tempos de execução e na construção de modelos analíticos a partir dos tempos medidos.

Tanto a análise como a predição de desempenho costumam utilizar técnicas de modelagem como suporte ao seu desenvolvimento. De forma geral, podemos identificar duas abordagens principais para a modelagem de aplicações paralelas [Meira95]:

- *Bottom-up*: a modelagem do sistema é realizada através de um conjunto de parâmetros ou funções representando o *software*, o *hardware* e as interações envolvidas (alguns importantes exemplos desta abordagem são as técnicas de modelagem analítica);

- *Top-down*: também denominada modelagem estrutural, esta abordagem considera como ponto inicial a própria aplicação a ser modelada e obtém as informações necessárias através de descrições de usuários ou análises de compilação (um bom exemplo trata-se da modelagem por análise estática).

As seções seguintes descrevem um pouco mais detalhadamente duas das principais técnicas utilizadas na modelagem de sistemas paralelos: modelagem analítica e estrutural.

### 3.4.1 Modelagem analítica

As características de um sistema paralelo são abstraídas na modelagem analítica como um conjunto de parâmetros ou funções parametrizadas [Meira95]. O sistema é descrito em termos de funções e equações, ou inequações, que podem ser utilizadas tanto para a análise como para a predição de desempenho.

O conjunto de equações e fórmulas que constituem um modelo analítico descrevem o funcionamento da máquina e da aplicação de um modo abstrato [Li01]. O nível de abstração é determinado pelos parâmetros utilizados na modelagem, sendo que o uso de mais parâmetros intensifica o detalhamento do modelo e, conseqüentemente, reduz seu nível de abstração.

Os parâmetros adotados também determinam a precisão da modelagem do sistema. Simplificações podem ser realizadas no modelo eliminando-se alguns parâmetros, o que tende a facilitar a sua aplicação embora possa reduzir a sua acurácia. Em outro extremo, um modelo corre o risco de se tornar intratável quando um grande número de parâmetros está em uso. Além disso, muitas vezes alguns destes parâmetros são difíceis de serem obtidos, aumentando substancialmente o custo da modelagem. Segundo [Meira95], a modelagem analítica pode ainda apresentar as seguintes variações, considerando a possibilidade de se substituir parâmetros escalares por funções ou ferramentas estatísticas:

- Modelagem com parâmetros escalares: os parâmetros são valores escalares capazes de expressar o comportamento do sistema sob certas condições;
- Modelagem com funções: a substituição de valores escalares por funções matemáticas melhora a flexibilidade e expressividade do modelo, mas pode aumentar sua complexidade devido principalmente à necessidade de se determinar a forma e os coeficientes das funções;
- Modelos estatísticos: neste caso, ferramentas estatísticas (como cadeias de Markov ou redes de Petri) são utilizadas e a modelagem caracteriza o comportamento assintótico do sistema.

Apesar de oferecer flexibilidade, as ferramentas estatísticas apresentam maior custo de utilização e exigem conhecimentos específicos, dos quais a maioria dos programadores não dispõem.

A simplicidade requerida em modelos analíticos não constitui necessariamente um obstáculo, mas pode oferecer vantagens. Conforme é discutido em [Kant92], modelos mais simples tendem a ser mais robustos que aqueles mais complexos, são mais fáceis de se entender, ajustar e modificar e em muitas situações os parâmetros de entada ou os resultados desejados não são bem definidos, dificultando o tratamento de parâmetros mais detalhados.

### 3.4.2 Modelagem estrutural

Como mencionamos anteriormente, na modelagem estrutural o ponto de partida é a aplicação em si. O modelo gerado é capaz de abstrair o desempenho de um sistema como um conjunto de componentes e um modelo *top-level*, que descreve as relações entre estas representações [Schopf98]. Podemos então dizer que a modelagem estrutural permite decompor uma aplicação em componentes, representando estes componentes em modelos distintos e as relações entre eles em um modelo de alto nível.

Para a construção do modelo, as informações podem ser obtidas através de descrições de usuários ou análises de compilação. Assim, em [Meira95] a modelagem estrutural é dividida em:

- Modelagem estrutural baseada em descrição: o modelo é baseado em uma descrição do programa e consiste de um grafo de tarefas, ou equivalente, descrevendo o paralelismo da aplicação e alguns parâmetros expressando as características do *hardware*;

- Modelagem estrutural através de análise estática: requer apenas o código fonte do programa, podendo utilizar ferramentas de análise estática que geralmente são partes de ambientes de compilação. Contudo, o desenvolvimento destas ferramentas é custoso e a acurácia do modelo é restrita por elas.

As técnicas de modelagem analítica e estrutural não são mutuamente exclusivas, pelo contrário, elas podem ser combinadas buscando-se melhores resultados. A estrutura da aplicação e a análise estática podem prover informações difíceis de

se obter e que são requeridas do usuário nos modelos analíticos [Meira95]. Essa combinação permite a obtenção de representações melhor elaboradas e mais precisas.

## 3.5 Trabalhos relacionados

### 3.5.1 Culler et al

O modelo LogP [Culler93] foi proposto no intuito de caracterizar adequadamente os principais fatores de desempenho em máquinas paralelas de memória distribuída. Estimulado pela tendência de crescimento no uso destas máquinas, o trabalho desenvolvido por Culler et al pode ser utilizado na modelagem e no estudo de aplicações baseadas em trocas de mensagens. Para isso, o modelo enfatiza os aspectos de desempenho relacionados à rede de interconexão e procura sintetizá-los nos seguintes parâmetros:

- *L*: latência envolvida na transmissão da mensagem através da rede;
- *o*: *overhead* relacionado às tarefas de envio e recepção da mensagem (durante este intervalo de tempo o processador encontra-se dedicado, não podendo executar outras operações);
- *g*: denominado de *gap*, este parâmetro trata-se do tempo mínimo que deve ser respeitado entre transmissões ou recepções consecutivas;
- *P*: número de nós de processamento.

Através dos parâmetros citados, o modelo LogP procura auxiliar a construção de algoritmos paralelos eficientes e portáveis. O trabalho ainda descreve algumas experiências no desenvolvimento de algoritmos compatíveis com o modelo, além de suas respectivas implementações em uma máquina baseada em passagem de mensagens (*Thinking Machines CM-5*).

Assim como discutimos na subseção 2.2, dentre os fatores que influenciam o desempenho da rede de interconexão surgem a latência (antes denominada latência de transporte) e os *overheads* de envio e recepção. Contudo, o trabalho de

Culler destingue um novo parâmetro, o *gap*, que na análise apresentada na subseção 2.2 estaria possivelmente incluso nos *overheads*.

### 3.5.2 Xu e Hwang

Xu e Hwang [Xu96] desenvolveram um modelo analítico para estimar o desempenho de primitivas de comunicação executadas sobre uma máquina IBM SP2. O modelo apresentado, que podemos visualizar abaixo, considera a latência da rede $t_o(n)$, a largura de banda $r_\infty(n)$ e o tamanho da mensagem $m$ como parâmetros para o cálculo do tempo de comunicação. A latência e a largura de banda são estimadas em termos do número de nós envolvidos na operação ($n$).

$$t = t_o(n) + \frac{m}{r_\infty(n)}$$

Desta forma, são obtidas equações lineares para caracterizar primitivas ponto-a-ponto e modelos logarítmicos para comunicações coletivas. A modelagem de primitivas é realizada com o apoio de diversas medições de desempenho executas no IBM SP2, variando o tamanho das mensagens e o número de nós envolvidos. Através de um ajuste de curvas sobre os tempos medidos, equações matemáticas são então construídas e adequadas ao modelo apresentado acima.

Os resultados descritos no artigo demonstram a acurácia dos modelos analíticos, gerados com a aplicação de técnicas de medição e ajustes de curva. O trabalho utilizou nos testes o MPICH, do Argonne National Laboratory, e a biblioteca MPL da IBM.

### 3.5.3 Abandah e Davidson

Um outro trabalho analisando o desempenho das comunicações no IBM SP2 foi desenvolvido por Abandah e Davidson [Abandah96]. Os seguintes fatores de influência no tempo despendido em comunicações são considerados: *overhead* de escalonamento (tempo para alocação e liberação de processos), passagem de mensagem (tempo gasto com as transferências de dados através da rede de interconexão) e *overhead* de sincronização (tempo envolvido com a sincronização de processos).

Dentre estes fatores, os autores ressaltam que a maior influência é exercida pelo tempo gasto com as transferências de dados via rede. A modelagem deste tempo é realizada com a elaboração de equações para as primitivas ponto-a-ponto, em função do tamanho da mensagem, e para as coletivas em termos do tamanho da mensagem e do número de processadores. Assim como no trabalho comentado na subseção anterior, Abandah e Davidson utilizam medições e técnicas de ajuste de curvas no processo de confecção dos modelos analíticos.

Contudo, este trabalho apresenta um importante diferencial nos modelos construídos. O domínio dos tamanhos de mensagens transmitidas é dividido em regiões e são obtidas equações que variam conforme a região. Para mensagens de uma determinada faixa de bytes existem respectivas equações para estimar os tempos das primitivas de comunicação. Variando a quantidade de bytes transmitidos poderemos incidir em um outro intervalo e, neste caso, os tempos das primitivas podem ser calculados com o uso de outras equações. O trabalho apresenta ainda um algoritmo em alto nível de abstração propondo que as regiões sejam delimitadas segundo um método puramente experimental: iniciada uma região com dois pontos, novos pontos são adicionados incrementalmente e a acurácia do modelo é testada. Uma nova região será definida quando a margem de erro observada ultrapassar um nível pré-determinado.

## 3.6 Metodologia proposta por Li

O trabalho desenvolvido por Li [Li01] apresenta uma metodologia para análise e predição de desempenho de programas MPI. O objetivo principal da metodologia está na predição de tempos de execução, através do desenvolvimento de grafos e modelos analíticos capazes de representar as comunicações e as computações locais. São propostos alguns elementos gráficos para expressar adequadamente o comportamento de programas baseados em passagem de mensagens, a partir das quais é possível a elaboração de equações de predição de desempenho.

Assim, são mescladas técnicas das modelagens analítica e estrutural. O restante desta seção descreve as características principais da metodologia, bem como os procedimentos envolvidos no processo estabelecido pela sua aplicação.

### 3.6.1 Modelagem através de classes de grafos

A metolologia proposta por Li apresenta extensões sobre os grafos de tempos definindo duas novas classes de grafos, denominadas DP*Graph e T-graph*. A partir do código fonte de um programa paralelo são construidos grafos capazes de representar sua estrutura, facilitando a realização de análises estruturais e estudos com objetivos de avaliação e predição de desempenho.

O processo de modelagem através destas classes de grafos permite caracterizar um programa em dois níveis de abstração. Enquanto a criação de grafos T-graph* possibilita um estudo da aplicação em alto nível, a representação da mesma aplicação em DP*Graph permite uma análise mais detalhada em que cada passo da seqüência de execução pode ser identificado.

Com o intuito de isolar os efeitos não-determinísticos da contenção na rede, as comunicações são analisadas isoladamente. Os pontos dentro do código do programa onde ocorrem sincronizações ou comunicações explícitas são identificados, destacados e os seus tempos de execução são computados em separado das computações locais. Estas execuções locais, realizadas entre dois pontos sucessivos de comunicação, são abstraídas como um conjunto de segmentos de código e, desta forma, os tempos de cada um destes conjuntos também podem ser computados separadamente.

A criação das classes de grafos T-graph* e DP*Graph, bem como a adoção da proposta de se efetuar predição de desempenho utilizando a concepção de dois níveis, procuram permitir que programas paralelos implementados com o padrão MPI possam ser representados de forma completa e precisa. As duas próximas seções descrevem sucintamente as classes de grafos T-graph* e DP*Graph, incluindo as simbologias utilizadas e alguns detalhes sobre a confecção dos modelos.

#### 3.6.1.1 Representação de baixo nível

Conforme apresentado na seção 2.4, o padrão MPI define um conjunto de primitivas de comunicação para realizar as trocas de mensagens em sistemas paralelos e distribuídos. Com o emprego do conjunto de estruturas da classe de grafos DP*Graph, apresentado na Figura 3.1, podemos representar as comunicações e

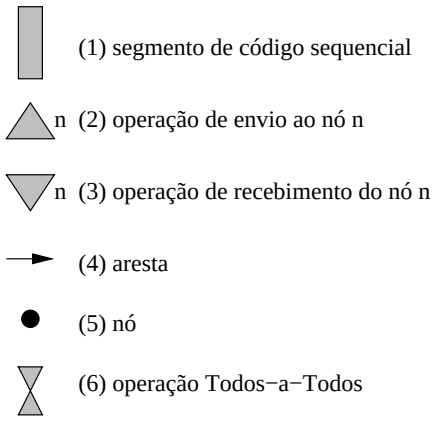


Figura 3.1: Estruturas da classe de grafos DP*Graph.

execuções locais de um programa MPI e detalhar a sua seqüência de execução.

O emprego destas estruturas possibilita representar um programa paralelo baseado em trocas de mensagens, incluindo as suas comunicações explícitas. Os trechos seqüenciais são representados pelo componente (1), ao passo que os componentes (4) e (5) são fundamentais para expressar possíveis fluxos de execução do programa e indicar determinados pontos entre execuções distintas. As estruturas (2) e (3) indicam, respectivamente, o envio de dados para o nó de processamento *n* e a recepção de dados provinientes de um certo nó *n* (por exemplo as operações *send* e *receive*). As operações de comunicação envolvendo todos os nós de processamento do sistema são expressadas pelo componente (6).

#### 3.6.1.2 Representação de alto nível

A representação de um programa MPI em baixo nível torna a elaboração de seu modelo T-graph* quase imediada, visto que as ocorrências importantes (como comunicações e trechos de código seqüenciais) já foram identificadas e a seqüência de execução do programa encontra-se esclarecida e explicitada. O conjunto de estruturas da classe de grafos T-graph* é apresentado a seguir na Figura 3.2.

As estruturas (1), (3) e (4) são análogas àquelas apresentadas na Figura 3.1. Assim, a simbologia T-graph* apresenta apenas um novo componente em relação à classe de grafos DP*Graph; o símbolo (2) indica um subgrafo, podendo ser utilizado para representar um conjunto qualquer de instruções, e permite a abstração

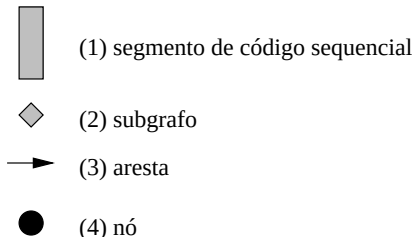


Figura 3.2: Estruturas da classe de grafos T-graph*.

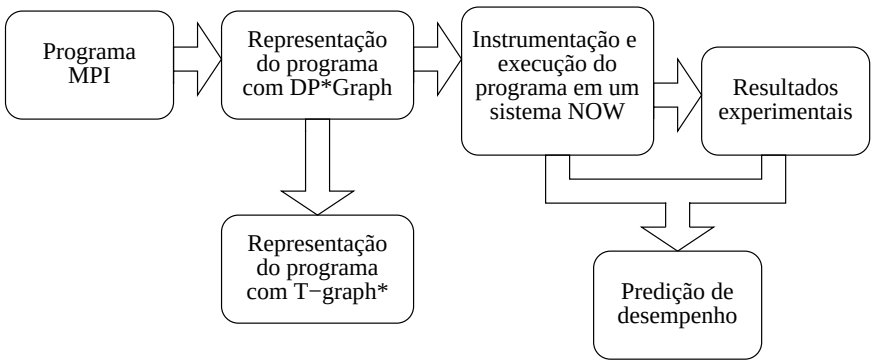


Figura 3.3: Metodologia de análise e predição de desempenho [Li01].

de detalhes de um determinado trecho de código em um modelo de mais alto nível.

### 3.6.2 Aplicação da metodologia

A metodologia de análise e predição de desempenho, descrita nestas seções, deve ser aplicada segundo alguns passos estabelecidos e utilizar um conjunto de notações oferecidas. As estruturas apresentadas pelas classes de grafos DP*Graph e T-graph*, anteriormente descritas, constituem tais notações. O esquema apresentado na Figura 3.3 ilustra a aplicação da metodologia.

A partir do código fonte de um programa paralelo baseado em trocas de mensagens é construída uma representação DP*Graph. Tal representação possibilita a obtenção do modelo em alto nível T-graph* e facilita a instrumentação do código fonte com primitivas para medir os tempos de execução dos diversos trechos que compõem o programa estudado. O processo de medição de desempenho prossegue executando-se a nova versão do programa, agora instrumentado, em um sistema NOW. Os resultados experimentais obtidos, bem como a versão do có-

digo instrumentado, são fundamentais para a geração de modelos analíticos que serão utilizados na realização de análises e predições de desempenho.

Os passos a seguir detalham melhor o processo envolvido pela metodologia, conforme é descrito em [Li01]:

1. Após uma análise inicial do programa paralelo, é construída a sua representação em baixo nível utilizando as estruturas da classe de grafos DP*Graph.

2. Observando-se a sua representação em baixo nível, o programa paralelo é instrumentado, permitindo a realização de medições dos tempos gastos nas comunicações e nas execuções locais. Tempos de execução podem então ser obtidos, executando-se a versão instrumentada do programa no ambiente de estações de trabalho com variações no número de nós de processamento e no tamanho do problema. Desta forma, são computados tempos de execução para cada um dos trechos do programa com execuções locais ou com comunicação, em diversas configurações possíveis.

3. Com os tempos experimentais obtidos na passo anterior, são construídas equações que caracterizam os tempos de execução de cada um dos trechos do programa instrumentado. Os tempos gastos em comunicações podem ser caracterizados por equações matemáticas em termos do tamanho da mensagem e do número de processos envolvidos, através da aplicação de técnicas de ajuste de curvas. Para os trechos delimitados por comunicações sucessivas, também são geradas equações capazes de caracterizar adequadamente seus respectivos tempos de execução. O tempo total do programa é então dado por um modelo analítico, composto pela somatória dos tempos de cada trecho.

4. O modelo analítico construído no passo anterior permite a realização de predições de tempos de execução, considerando o programa inteiro ou apenas alguns de seus trechos. Para isso, deve-se fixar o número de nós de processamento e variar o tamanho do problema ou, analisando uma outra possibilidade, manter o tamanho do problema fixo e alterar o número de nós.

5. Uma outra representação do programa paralelo, utilizando a classe de grafos T-graph*, pode ser elaborada partindo-se da representação de baixo nível gerada anteriormente. Este modelo em alto nível possibilita visualizar o possível fluxo de execução do programa, porém sem explicitar maiores detalhes como as comunicações realizadas.

### 3.6.3 Cálculo do tempo de execução de um programa paralelo

Conforme discutido na subseção anterior, o tempo total de um programa é fornecido por um modelo analítico resultante da somatória das equações que representam os tempos de cada trecho do programa. É necessário, portanto, analisar cada um deste trechos ou segmentos e construir equações capazes de expressar adequadamente seus comportamentos. Para isso, os tempos de execução do programa, obtidos através de testes experimentais no ambiente de estações de trabalho, são utilizados como parâmetros e comunicações e computações locais devem ser estudadas isoladamente.

Os tempos despendidos em computações locais podem ser calculados através de análises de complexidade dos algoritmos utilizados. Sendo o enfoque da nossa pesquisa justamente o tempo gasto com comunicações, não nos deteremos em maiores detalhes sobre a construção de equações para os trechos seqüenciais; porém, convém ressaltar que o cálculo do tempo de um determinado trecho deve considerar todas as operações nele realizadas.

Um mesmo segmento pode ser executado paralelamente por diversos processos ou os processos de um programa podem estar realizando diferentes operações em paralelo. Nestes casos, precisamos analisar o fluxo de execução de cada processo envolvido, calculando seus respectivos tempos de execução como a somatória de seus tempos parciais, e o tempo total do programa será determinado pela maior das somatória encontradas. Considerando o programa cujas representações T-graph* e DP*Graph são ilustradas pela Figura 3.4, o tempo total de execução equivale ao máximo entre os tempos dos nós 0 e 1:

$$T_{total} = máximo(x_1 + x_2 + x_3,\ y_1 + y_2 + y_3)$$

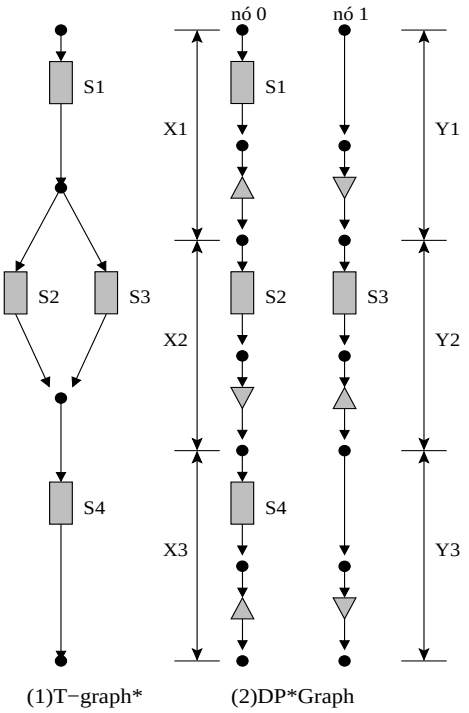


Figura 3.4: Exemplo de cálculo do tempo de execução de um programa.

### 3.6.4 Componentes do tempo de comunicação

O tempo gasto com comunicações constitui um importante fator a ser considerado no estudo e na análise de aplicações paralelas projetadas para *clusters* de estações de trabalho. Conforme discutimos na subseção 2.2.6, este tempo é influenciado por diversos fatores como a largura de banda da rede de interconexão, o tamanho da mensagem transmitida e, principalmente, os *overheads* envolvidos no envio e no recebimento da mensagem. Em [Li01] são definidos três componentes para o tempo de comunicação de uma mensagem de *n* elementos:

- $t_e(n)$ : tempo para transportar *n* dados da memória local para a interface de rede (tempo de envio);
- $t_t(n)$ : tempo para transferir *n* dados através da rede de interconexão entre dois nós de processamento (tempo de transferência);
- $t_r(n)$ : tempo para transportar *n* dados da interface de rede para a memória local (tempo de recebimento).

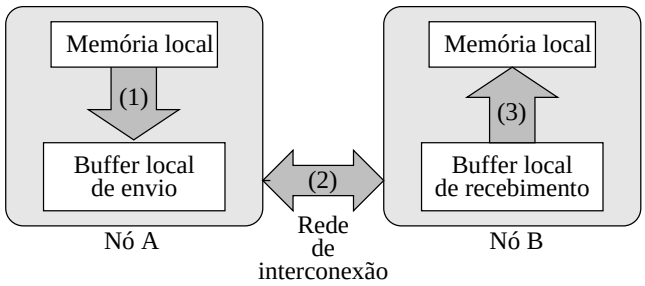


Figura 3.5: Representação dos componentes do tempo de comunicação.

A Figura 3.5 ilustra o processo envolvido na transmissão de uma mensagem entre dois nós de processamento, indicando as respectivas fases referentes aos tempos de envio (1), transferência (2) e recebimento (3). Desta forma, o tempo de comunicação de uma mensagem de tamanho *n* é dado pela expressão:

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

## 3.7 Considerações finais

As duas principais abordagens utilizadas na modelagem de sistemas paralelos (analítica e estrutural) foram comentadas neste capítulo. Alguns fatores pertinentes à correta realização das atividades de análise e predição de desempenho também foram disticudos, incluindo erros cometidos e outros detalhes a serem destacados.

Além disso, apresentamos de forma sucinta alguns trabalhos relacionados, procurando destacar contribuições e observações mais relevantes ao nosso estudo. Dentre os trabalhos relacionados, o maior destaque foi dado a uma metodologia de análise e predição de desempenho proposta por Li [Li01]. Tal metodologia, baseada no uso de medições de tempos experimentais e modelagem analítica com fins de análise e predição, foi utilizada para a obtenção dos modelos apresentados nos capítulos 4 e 5.

# Capítulo 4

# MODELAGEM DE PRIMITIVAS PONTO-A-PONTO

## 4.1 Introdução

A análise e predição de desempenho de primitivas de comunicação requer a elaboração de modelos capazes de caracterizar diversos fatores envolvidos na transmissão da mensagem, muitas vezes retratando a influência exercida pelos algoritmos utilizados na implementação das primitivas. Assim, faz-se necessário um estudo da implementação MPI, procurando identificar detalhes relacionados à maneira como as operações de comunicação são concretizadas.

Através da caracterização das primitivas MPI consideradas neste trabalho, procuramos obter modelos de análise e predição precisos e simples, possibilitando a correta descrição do comportamento destas primitivas e a realização de predições de tempos de execução com razoável acurácia. Para isso, são utilizadas algumas técnicas de ajuste de curvas sobre tempos experimentais coletados e, desta forma, elaboramos modelos analíticos consistindo de um conjunto de equações em função do tamanho da mensagem comunicada e, nos casos de comunicação coletiva, também do número de processos envolvidos.

O processo de análise e predição estabelecido pela metodologia proposta por Li [Li01], comentada no capítulo anterior, combina técnicas de modelagem analítica e medições de tempos de execução. Com o uso de análises estruturais dos có-

digos dos programas estudados, torna-se possível a correta instrumentação destes programas utilizando medidores de tempo e, após isso, medições de desempenho são realizadas em um *cluster*. Em nosso trabalho, executamos um procedimento semelhante e obtivemos as equações aplicando ainda um método de ajuste de curvas sobre os tempos coletados.

Além da aplicação destas técnicas, foi necessário o desenvolvimento de programas específicos com o objetivo de realizar medições dos tempos despendidos nas comunicações. A elaboração destes programas e a posterior coletagem de tempos para realizar os ajustes de curvas foram realizadas no Laboratório de Arquitetura e Software Básico, fazendo uso dos recursos computacionais descritos na seção 4.4. Neste capítulo, iniciamos com uma descrição dos cuidados e procedimentos adotados em nosso trabalho, visando o cumprimento correto das atividades de medição, análise e predição e, conseqüentemente, a elaboração de modelos robustos. Em seguida apresentamos a caracterização das primitivas ponto-a-ponto, incluindo as análises realizadas, os modelos elaborados e os resultados obtidos.

## 4.2 Procedimentos e ferramentas adotadas

Em um trabalho de medição de desempenho, esperamos obter dados que reflitam corretamente o comportamento do sistema, podendo ser utilizados posteriormente, por exemplo, em um processo de avaliação ou modelagem. Para alcançar este objetivo, é aconselhável a adoção de certos cuidados durante a obtenção e o tratamento dos dados, procurando eliminar anormalidades e selecionar as informações realmente significativas.

Algumas anormalidades costumam ocorrer durante a execução de programas baseados em passagem de mensagens devido a fatores provenientes da rede de interconexão ou, com menos freqüência, influências estranhas à aplicação observadas nas computações locais. Considerando o primeiro caso, anomalias no desempenho das comunicações podem ser provocadas, por exemplo, quando a execução de outros programas leva ao congestionamento da rede. Para o segundo caso, o tempo de uma computação local medido em um momento de instabilidade do sistema operacional, por exemplo, pode apresentar um valor incoerente com o

comportamento normal do programa. Contudo, devemos salientar que é preciso estudar as causas destas anormalidades antes de descartarmos dados obtidos no processo de medição, garantindo com isso a correta caracterização da aplicação estudada. Se um dos objetivos da medição é justamente caracterizar o desempenho das comunicações quando o tráfego na rede é intenso, devemos simular situações com um comportamento semelhante ao desejado.

Nas subseções seguintes discutimos os procedimentos adotados em nosso trabalho para a realização das medições de desempenho. Além disso, apresentamos na seção 4.2.4 uma ferramenta utilizada como suporte ao processo de tratamento dos dados e construção dos modelos analíticos.

### 4.2.1 Tratamento de *outliers*

Muitas vezes, em um processo de medição de desempenho, obtemos um conjunto variado de dados sendo necessário identificar aqueles que não traduzem o comportamento normal do sistema e descartá-los. Por exemplo, ao realizarmos diversas execuções de uma mesma primitiva de comunicação podemos nos deparar com tempos distintos, alguns deles refletindo a influência de fatores estranhos (como as anormalidades comentadas anteriormente). Segundo [Jain91], as observações incompatíveis com a maioria dos resultados obtidos podem ser consideradas *outliers*.

Ao encontrarmos *outliers* durante as medições, antes de decidirmos descartar ou considerar tais observações precisamos procurar compreender os fatores que possivelmente causaram variações nos resultados. Se por um lado a inclusão de um determinado *outlier* pode acarretar em alterações significativas nas conclusões de um processo de análise, sua exclusão talvez conduza à obtenção de conclusões equivocadas caso os valores descartados representem observações corretas [Jain91].

Como as medidas de desempenho coletadas em nossos testes tratam-se de tempos despendidos em operações de comunicação, os *outliers* são ocasionados principalmente por oscilações no desempenho da rede de interconexão. Tendo em vista a correta representação das primitivas MPI, durante o processo de medição procuramos utilizar técnicas capazes de selecionar adequadamente os tempos e

minimizar as variações observadas, conforme discutimos a seguir.

### 4.2.2 Medição dos tempos de comunicação

Quando uma mesma mensagem é transmitida por diversas vezes consecutivas, possivelmente o sistema operacional mantém os dados constituintes da mensagem na memória *cache* e os reutilize nas transmissões conseguintes. Este procedimento, embora seja comumente praticado pelo sistema com vistas à melhoria de desempenho, pode mascarar o *overhead* envolvido na comunicação pois tende a eliminar o tempo normalmente gasto para acessar os dados na memória principal e copiá-los para a *cache*. Por isso, de forma semelhante ao trabalho desenvolvido por Luecke, Raffin e Coyle sobre desempenho de primitivas de comunicação coletiva MPI [Luecke99], procuramos evitar este procedimento forçando a atualização da *cache* sempre que uma mensagem fosse retransmitida. Com a intercalação de execuções de outros programas entre as retransmissões, esperamos melhorar a caracterização do tempo despendido na fase inicial de uma comunicação.

Outros cuidados ainda foram tomados para evitar ou pelo menos minimizar as influências de outras aplicações e do sistema operacional, como as anormalidades discutidas anteriormente, que podem favorecer o surgimento de *outliers*. Tanto a rede de interconexão como as máquinas envolvidas nas execuções foram utilizadas com exclusividade durante a realização das medições. Para cada configuração testada (por exemplo, uma operação *broadcast* de 500 mil inteiros envolvendo 8 processos), uma quantidade considerável de execuções foi realizada, variando entre 30 e, na maioria dos casos, 50 testes. Cada nó de processamento envolvido alojava apenas um processo, sendo, portanto o número de máquinas equivalente ao de processos.

Dentre os tempos medidos em cada conjunto de testes, os valores maiores eram descartados e a média dos tempos era calculada considerando apenas os menores valores. Conforme [Xu96], os menores tempos geralmente são os mais precisos pois sofreram menor influência do sistema operacional e de outras aplicações. Assim, dependendo da distribuição observada entre os valores medidos, selecionamos aqueles que estivessem abaixo de um determinado limite (equivalente a 1,8 vezes o menor tempo), calculando em seguida a média sobre a seleção

realizada.

Uma outra política de seleção possível seria considerarmos os tempos situados no intervalo de $t_m - S$ a $t_m + S$, sendo $t_m$ e $S$ o valor médio e o desvio padrão respectivamente. Contudo, o nosso maior objetivo no processo de medição era simplesmente a obtenção de valores capazes de representar apropriadamente o desempenho das primitivas MPI, descartando os tempos muito elevados, por isso não nos detivemos demasiadamente na escolha da melhor política, apenas selecionamos os valores abaixo de 1,8 vezes o menor tempo.

O processo de medição executado seguiu estes cuidados, utilizando ainda o ambiente computacional descrito na seção 4.4 e alguns programas instrumentados com medidores de tempo, que serão comentados na seção 4.3. Desta forma, esperamos obter tempos médios capazes de refletir adequadamente o comportamento das comunicações, propiciando a construção de modelos mais precisos.

### 4.2.3 Construção dos modelos analíticos

A construção dos modelos analíticos foi possibilitada com o uso de técnicas de ajuste de curvas [Press92]. Mais especificamente, o método adotado para o ajuste de curvas gera polinômios de grau $g$ e pode ser sintetizado nos seguintes passos:

1. Dado o vetor das abcissas $X_{1\times N}$ , encontrar a matriz $A_{N\times k}$ onde $k = g + 1$ e $a_{i,j} = x_i^{j-1}$;
2. Calcular a matrix $A^+_{k\times N}$ como a pseudo-inversa de $A$;
3. Sendo $Y_{N\times 1}$ o vetor das ordenadas, encontrar o vetor de coeficientes $C_{k\times 1}$ tal que $C = A^+ * Y$;
4. O polinômio é dado por $y = c_1 + c_2 * x + c_3 * x^2 + c_4 * x^3 + ... + c_k * x^g$.

Com a aplicação do método descrito, conseguimos elaborar equações para modelar o comportamento das primitivas de comunicação MPI estudadas. Na realidade, o método procura encontrar os coeficientes $a_0$, $a_1$, $a_2$, ..., $a_g$ :

$$minimo_{\{a_0,a_1,...a_g\}}(Y - A * C)^2$$
$$= \Sigma_{i=1}^{N}[yi - (a_0 + a_1 * x_i + a_2 * x_i^2 + ... + a_g * x_i^g)]^2$$

### 4.2.4 Scilab 2.6

Em trabalhos de medição e análise de desempenho é comum nos depararmos com grandes quantidades de dados a serem tratados. Os tempos coletados em nossos testes, por exemplo, acumulam muitos valores provenientes das diversas configurações executadas, variando os números de processos envolvidos e os tamanhos de mensagens transmitidas. Por isso, é conveniente o uso de ferramentas de *software* adequadas para facilitar e automatizar tarefas relacionadas ao processo de análise e predição (seleção de dados, cálculos de médias, construção de gráficos, etc).

O Scilab [INRIA] é um *software* livre desenvolvido pelo INRIA (*Institut National de Recherche en Informatique et en Automatiqueu, France*) destinado à construção de aplicações para processamento de sinais e controle de sistemas. Através de sua vasta biblioteca de funções, oferece suporte à realização de operações envolvendo matrizes, geração de modelos matemáticos e gráficos multidimensionais, entre muitas outras funcionalidades. Além disso, o Scilab disponibiliza ao usuário facilidades para escrita e leitura de dados em arquivos, possibilitando a elaboração de programas capazes de automatizar tarefas relacionadas a atividades de análise e predição de desempenho.

Desta forma, em nosso trabalho utilizamos o Scilab 2.6 como uma ferramenta de suporte no tratamento dos dados e na obtenção dos modelos analíticos. Através do conjunto de funções descrito no Apêndice A, desenvolvidas fazendo uso do *software,* possibilitamos a automatização de algumas tarefas como:

- seleção dos tempos coletados e cálculo dos tempos médios;
- construção de gráficos para análise do comportamento das primitivas de comunicação;
- geração de equações de predição de desempenho;
- elaboração de dados comparativos com tempos medidos, valores preditos e erros percentuais.

## 4.3 Programas elaborados para as medições

Considerando os problemas encontrados em um processo de medição de desempenho e os cuidados a serem respeitados, conforme comentado anteriormente, elaboramos alguns programas específicos para coletagem dos tempos de comunicação. Basicamente, utilizamos medidores para contar o tempo despendido durante a ocorrência de uma comunicação e, ao final da execução, os valores medidos eram gravados em um arquivo. As execuções eram repetidas um número determinado de vezes e, com o objetivo de forçar a atualização da memória *cache*, intercaladas com um outro programa que realizava uma multiplicação de matrizes.

O trecho de código a seguir pertence ao programa construído para medir o tempo do *send* padrão e ilustra o uso dos medidores de tempo:

```
if (0 == rank) {
  TIMER_CLEAR;
  TIMER_START;
  MPI_Send(buf,BUFSIZE,MPI_INT,1,11,MPI_COMM_WORLD);
  TIMER_STOP;
  printf("%g",TIMER_ELAPSED); //tempo do sender
}
else {
  TIMER_CLEAR;
  TIMER_START;
  MPI_Recv(buf,BUFSIZE,MPI_INT,0,11,MPI_COMM_WORLD,&status);
  TIMER_STOP;
  printf("    %g\n",TIMER_ELAPSED); //tempo do receiver
}
```

As contagem do tempo é inicializada fazendo uso das funções TIMER_CLEAR e TIMER_START e finalizada com o TIMER_STOP. Os programas usados nas medições dos demais modos de *send* e das primitivas coletivas apresentam o mesmo procedimento, com medidores antes e após a comunicação. Desta forma, em nossos testes a coletagem de tempos era realizada estando os processos envolvidos previamente sincronizados. Convém ressaltar que em aplicação podem ocorrer eventuais atrasos na execução de uma primitiva, se os processos não estiverem em sincronismo. Contudo, como nosso enfoque é justamente o tempo despendido nas

comunicações, não tratamos os casos de dependências entre as operações executadas.

## 4.4 Ambiente computacional utilizado nos testes

Os testes realizados para a construção e avaliação dos modelos de predição de desempenho utilizaram um ambiente computacional do LASB - Laboratório de Arquitetura e Software Básico, do Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da USP. Trata-se de um *cluster* com as características apresentadas abaixo:

- nós de processamento: 16 máquinas contendo cada uma um processador INTEL Celeron de 433 MHz, 128 Mbytes de SDRAM (66 MHz), 32 Kbytes de cache L1, 128 Kbytes de cache L2 e uma placa de rede INTEL EtherExpress Pro; uma máquina multi-processada com 4 processadores INTEL Pentium 2 Xeon de 400 MHz, 256 Mbytes de SDRAM, 32 Kbytes de cache L1, 512 Kbytes de cache L2 e a mesma placa de rede das demais máquinas;
- rede de interconexão: Fast-Ethernet;
- *switch*: 3COM SuperStack II 3300 de 24 portas;
- sistema operacional: Linux Red Hat versão 6.2 (nos nós homogêneos) e versão 7.0 (na máquina multi-processada);
- implementações do padrão MPI disponíveis: LAM-MPI versões 6.4 e 6.5.6.

## 4.5 Primitivas de comunicação ponto-a-ponto

Dentre as primitivas de comunicação ponto-a-ponto bloqueante, analisamos e modelamos os quatro modos de *send* descritos na subseção 2.4.1: padrão, *buffered*, síncrono e *ready*. A modelagem destas primitivas foi realizada segundo a metodologia de análise e predição de desempenho proposta por Li [Li01], descrita na seção 3.6, considerando os componentes do tempo de comunicação identificados em seu trabalho ($t_c(n) = t_e(n) + t_t(n) + t_r(n)$).

Contudo, a única comunicação ponto-a-ponto que Li modelou em seu trabalho trata-se do par *send*/*receive* no modo padrão. Assim, apresentamos nesta seção uma extensão à sua modelagem, incluindo os demais modos de *send*, além de descrevermos alguns testes realizados.

### 4.5.1 Considerações sobre a implementação dos modos de *send*

Após estudarmos o código fonte da implementação LAM-MPI, identificamos algumas semelhanças e diferenças entre os quatro modos de *send*, conforme discutimos a seguir:

- as primitivas MPI_Rsend e MPI_Ssend são bastante semelhantes ao *send* padrão, distinguindo-se pelo uso de diferentes mecanismos de sincronismo, segundo as suas respectivas semânticas. Conforme veremos a seguir, o desempenho destas primitivas é bastante parecido;

- a implementação do MPI_Bsend é semelhante ao MPI_Isend (*send* não-bloqueante), sendo a maior diferença o *overhead* gerado pelo alocação do *buffer*. Assim, no modo *buffered* o tempo despendido para transmitir uma mensagem tende a ser bem menor em relação aos demais modos bloqueantes.

### 4.5.2 Modelagem analítica dos modos de *send* bloqueantes

Os testes descritos nesta seção foram realizados no ambiente computacional apresentado na seção 4.4, utilizando os nós de processamento homogêneos e o LAM-MPI 6.4. As análises e modelos descritos podem também ser verificados no artigo intitulado "*Performance and Prediction of some MPI Communication Primitives*" [Oliveira02].

Foram realizados testes com mensagens de 100000, 250000, 400000, 550000, 700000, 850000 e 1000000 elementos do tipo inteiro (MPI_INT). Os tempos medidos são apresentados na Tabela 4.1. Tanto os tempos médios do processo emissor como do receptor são dados em segundos. Podemos verificar através de uma

Tabela 4.1: Tempos medidos em segundos para os quatro modos de *send*.

| Modo de send | | Tamanho da mensagem ($10^3$ inteiros) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 100 | 250 | 400 | 550 | 700 | 850 | 1000 |
| Padrão | Emissor | 0,0385 | 0,0990 | 0,1611 | 0,2223 | 0,2839 | 0,3463 | 0,4076 |
| | Receptor | 0,0484 | 0,1178 | 0,1881 | 0,2588 | 0,3294 | 0,4007 | 0,4723 |
| Buffered | Emissor | 0,0067 | 0,0168 | 0,0266 | 0,0375 | 0,0485 | 0,0578 | 0,0690 |
| | Receptor | 0.0484 | 0,1178 | 0,1876 | 0,2592 | 0,3308 | 0,3988 | 0,4707 |
| Ready | Emissor | 0,0401 | 0,1025 | 0,1660 | 0,2288 | 0,2918 | 0,3551 | 0,4174 |
| | Receptor | 0,0499 | 0,1213 | 0,1926 | 0,2639 | 0,3351 | 0,4062 | 0,4777 |
| Síncrono | Emissor | 0,0397 | 0,1017 | 0,1649 | 0,2272 | 0,2898 | 0,3529 | 0,4148 |
| | Receptor | 0,0503 | 0,1234 | 0,1964 | 0,2689 | 0,3415 | 0,4148 | 0,4890 |

análise dos valores que em todos os casos os processos demonstraram um crescimento quase linear em seus tempos de comunicação - tal comportamento poderá ser melhor visualizado nas Figuras 4.1 e 4.2.

Considerando particularmente o modo *buffered*, é fácil verificar que o tempo do emissor foi muito menor em comparação com os demais *sends*. Este comportamento era previsível, como comentamos anteriormente, pois os valores observados para o emissor no modo *buffered* correspondem, basicamente, a um tempo de leitura de um conjunto de dados na memória e sua transferência para o *buffer* da interface de rede. Portanto, percebemos que o tempo despendido pela primitiva MPI_Bsend pode ser resumido ao primeiro componente do tempo de comunicação ($t_e(n)$).

**Modelos analíticos dos modos de *send***

A partir dos tempos coletados e resumidos na Tabela 4.1 e aplicando-se técnicas de ajuste de curvas, obtivemos os modelos analíticos a seguir:

- *Send* padrão:

$$t_{send}(n) = -0,00315782 + (4,10604 * 10^{-7}) * n \tag{4.1}$$

$$t_{receive}(n) = 0,000242238 + (4,7111 * 10^{-7}) * n \tag{4.2}$$

- *Buffered send*:

$$t_{send}(n) = -0,000512821 + (6,92293 * 10^{-8}) * n \tag{4.3}$$

$$t_{receive}(n) = 0,00080281 + (4,69539 * 10^{-7}) * n \tag{4.4}$$

- *Ready send*:

$$t_{send}(n) = -0,00205469 + (4,1978 * 10^{-7}) * n \tag{4.5}$$

$$t_{receive}(n) = 0,00249505 + (4,75167 * 10^{-7}) * n \tag{4.6}$$

- *Send* síncrono:

$$t_{send}(n) = -0,00222082 + (4,17289 * 10^{-7}) * n \tag{4.7}$$

$$t_{receive}(n) = 0,00152971 + (4,86643 * 10^{-7}) * n \tag{4.8}$$

As Figuras 4.1 e 4.2, exibidas ao final desta seção, ilustram a semelhança nos comportamentos das curvas representando os tempos medidos e os calculados com auxílio das equações. Nos gráficos, as curvas rotuladas por sender_predict e receiver_predict indicam, respectivamente, os valores obtidos com a aplicação dos modelos para o processo emissor ($t_{send}(n)$) e para o receptor ($t_{receive}(n)$). Os tempos medidos são representados pelas curvas sender_measure e receiver_measure.

### 4.5.3 Predição de desempenho

As equações obtidas nos permitem realizar predições de tempos de comunicação, variando o tamanho da mensagem transmitida. Na Tabela 4.2, expomos um quadro comparativo entre tempos preditos e medidos para os quatro modos de *send* com mensagens de 40000, 60000, 1600000 e 1700000 inteiros.

Os resultados mostram-se satisfatórios, envolvendo erros na maioria dos casos abaixo de 6% e comprovando a validade dos modelos. Os maiores erros foram verificados com o modo *buffered* e, principalmente, envolvendo as menores mensagens. Buscando compreender e justificar este fato, devemos considerar os seguintes aspectos:

Tabela 4.2: Predições realizadas para os quatro modos de *send* *.

| | | Send | | | | Receive | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 40 | 60 | 1600 | 1700 | 40 | 60 | 1600 | 1700 |
| P | medição | 0,0124 | 0,0224 | 0,6521 | 0,6927 | 0,0208 | 0,0301 | 0,7526 | 0,7995 |
| | predição | 0,0133 | 0,0215 | 0,6538 | 0,6949 | 0,0191 | 0,0285 | 0,7540 | 0,8011 |
| | erro (%) | 6,8460 | 4,0179 | 0,2684 | 0,3129 | 8,1300 | 5,3156 | 0,1840 | 0,1979 |
| B | medição | 0,0027 | 0,0042 | 0,1090 | 0,1162 | 0,0207 | 0,0300 | 0,7521 | 0,7985 |
| | predição | 0,0023 | 0,0036 | 0,1103 | 0,1172 | 0,0196 | 0,0290 | 0,7521 | 0,7990 |
| | erro (%) | 16,009 | 14,286 | 1,1494 | 0,8741 | 5,2743 | 3,3333 | 0,0002 | 0,0680 |
| S | medição | 0,0130 | 0,0224 | 0,6659 | 0,7074 | 0,0216 | 0,0301 | 0,7779 | 0,8267 |
| | predição | 0,0145 | 0,0215 | 0,6654 | 0,7072 | 0,0210 | 0,0285 | 0,7802 | 0,8288 |
| | erro (%) | 11,704 | 4,0179 | 0,0631 | 0,0341 | 2,6412 | 5,3156 | 0,2872 | 0,2599 |
| R | medição | 0,0131 | 0,0235 | 0,6693 | 0,7110 | 0,0214 | 0,0312 | 0,7662 | 0,8135 |
| | predição | 0,0147 | 0,0231 | 0,6696 | 0,7116 | 0,0215 | 0,0310 | 0,7628 | 0,8103 |
| | erro (%) | 12,135 | 1,7021 | 0,0496 | 0,0831 | 0,4006 | 0,6410 | 0,4427 | 0,3950 |

* P = padrão, B = buffered, S = síncrono, R = ready, tempo em segundos

- pequenas oscilações no comportamento do sistema, devido a influências de outros fatores relacionados à rede de interconexão ou à máquina, costumam acarretar alterações de desempenho mais significativas em mensagem menores. Sendo o tempo de transmissão destas mensagens pequeno, qualquer oscilação no sistema computacional pode representar alterações proporcionalmente significativas;

- considerando que o MPI_Bsend tende a apresentar menores tempos de comunicação, conforme discutido anteriormente, a primitiva está mais sujeita às oscilações comentadas no primeiro item;

- o método de ajuste de curvas utilizado, devido às suas caracteríticas, minimiza os erros dos valores maiores e com isso acaba aumentando os erros de valores menores.

### 4.5.4 Uso de intervalos na construção de modelos

Uma alternativa para melhorarmos a precisão dos modelos analíticos das primitivas de comunicação trata-se do uso de intervalos no processo de modelagem.

O tempo despendido por um *send*, por exempo, pode variar conforme os tamanhos de mensagens (*n*) enviadas e apresentar curvas de desempenho que, embora demonstrem comportamentos lineares, possuam inclinações diferentes. Assim, elaborando equações para intervalos específicos de *n* podemos reduzir os erros da modelagem.

No trabalho de Abandah e Davidson [Abandah96], descrito sucintamente na seção 3.5, são geradas mais de uma equação para modelar o comportamento de uma mesma operação de comunicação. De forma semelhante, no artigo intitulado "*Análise e Predição de Desempenho de Programas MPI em Redes de Estações de Trabalho*" [Laine02], caracterizamos o desempenho das primitivas MPI_Send e MPI_Recv dividindo o domínio de *n* em duas regiões. Através de um conjunto de testes realizados com o *benchmark* MPBench [Mucci98], coletamos vários tempos de comunicação com mensagens de 6.824 a 5.592.404 bytes. Em seguida, aplicando técnicas de ajuste de curvas, obtivemos os seguintes modelos analíticos em função do número de bytes (*n*) para predição de tempos em $\mu s$ [1]:

$$6.824 \leq n \leq 65.536 \begin{cases} t_{send}(n) = -82,0523 + 0,0879182 * n \\ t_{receive}(n) = 9,706 + 0,0852 * n \end{cases} \tag{4.9}$$

$$65.536 < n \leq 5.592.404 \begin{cases} t_{send}(n) = 2.065,49 + 0,0849931 * n \\ t_{receive}(n) = 2.146,19 + 0,0849968 * n \end{cases} \tag{4.10}$$

Os resultados apresentados no artigo demonstram a validade do uso de intervalos no desenvolvimento de modelos analíticos. Comparando os tempos medidos e os valores calculados com as equações acima, o maior erro encontrado foi de apenas 4,142%, observado no *send* com *n* equivalente a 6.824. A Tabela 4.3 mostra uma comparação entre a medições e a predições, do emissor e do receptor, ilustrando a acurácia da modelagem realizada. Os modelos também mostraram-se precisos ao serem utilizados na predição de desempenho de um programa de

[1] Note que os modelos dos quatro modos de *send*, apresentados anteriormente, consideram o tamanho da mensage (*n*) em inteiros e o tempo em segundos. Por isso, existe uma grande disparidade entre as constantes das equações 4.9 e 4.10 e dos modelos anteriores.

Tabela 4.3: Resultados dos testes com a modelagem por intervalos *.

| n | $T_{send}$ | | | $T_{receive}$ | | |
|---|---|---|---|---|---|---|
| (bytes) | Medição | Predição | erro(%) | Medição | Predição | erro(%) |
| 6.824 | 540,280 | 517,901 | -4,1420 | 589,330 | 591,111 | 0,3022 |
| 13.652 | 1.130,440 | 1.118,207 | -1,0821 | 1.173,100 | 1.172,856 | -0,0208 |
| 21.844 | 1.830,230 | 1.838,433 | 0,4482 | 1.873,660 | 1.870,815 | -0,1519 |
| 43.688 | 3.686,130 | 3.758,918 | 1,9746 | 3.733,970 | 3.731,924 | -0,0548 |
| 65.536 | 5.726,130 | 5.679,755 | -0,8099 | 5.593,890 | 5.593,373 | -0,0092 |
| 131.072 | 13.198,730 | 13.205,706 | 0,0528 | 13.246,380 | 13.286,891 | 0,3058 |
| 218.452 | 20.629,029 | 20.632,403 | 0,0164 | 20.704,529 | 20.713,911 | 0,0453 |
| 349.524 | 31.766,000 | 31.772,618 | 0,0208 | 31.851,359 | 31.854,612 | 0,0102 |
| 524.288 | 46.636,770 | 46.626,352 | -0,0223 | 46.744,352 | 46.708,992 | -0,0756 |
| 1.048.576 | 91.211,102 | 91.187,215 | -0,0262 | 91.303,109 | 91.271,795 | -0,0343 |
| 1.747.624 | 150.591,703 | 150.601,471 | 0,0065 | 150.686,453 | 150.688,638 | 0,0014 |
| 3.495.252 | 299.122,563 | 299.137,793 | 0,0051 | 299.219,250 | 299.231,425 | 0,0041 |
| 5.592.404 | 477.389,344 | 477.381,242 | -0,0017 | 477.482,906 | 477.482,634 | -0,0001 |

* tempos em $\mu s$

multiplicação de matrizes, que envolvia entre outros cálculos operações de comunicação. Outras informações sobre este estudo podem ser conferidas no artigo publicado [Laine02].

## 4.6 Considerações finais

Os procedimentos descritos neste capítulo e utilizados em nosso trabalho refletem cuidados importantes a serem observados em um processo de medição e modelagem envolvendo operações de comunicação. Tais cuidados podem auxiliar na obtenção de modelos mais precisos e na realização de análises consistentes.

O capítulo ainda apresentou comentários sobre os programas construídos para as medições e uma ferramenta utilizada em nosso estudo, através da qual um conjunto de funções foi desenvolvido para automatizar algumas tarefas inerentes à execução das análises e elaboração dos modelos. Por fim, as análises e predições envolvendo as primitivas ponto-a-ponto foram descritas.

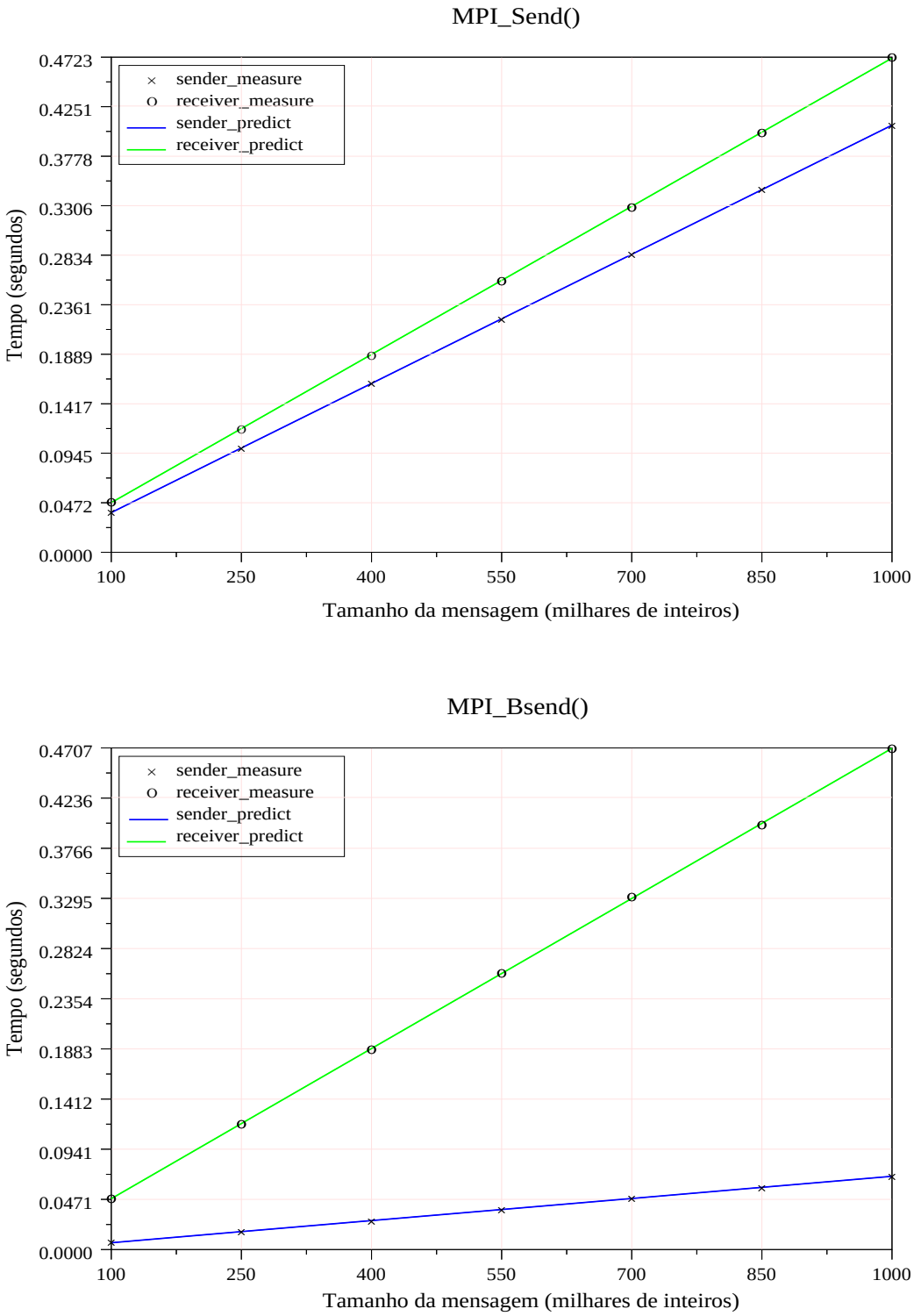


Figura 4.1: Desempenho do MPI_Send e do MPI_Bsend.

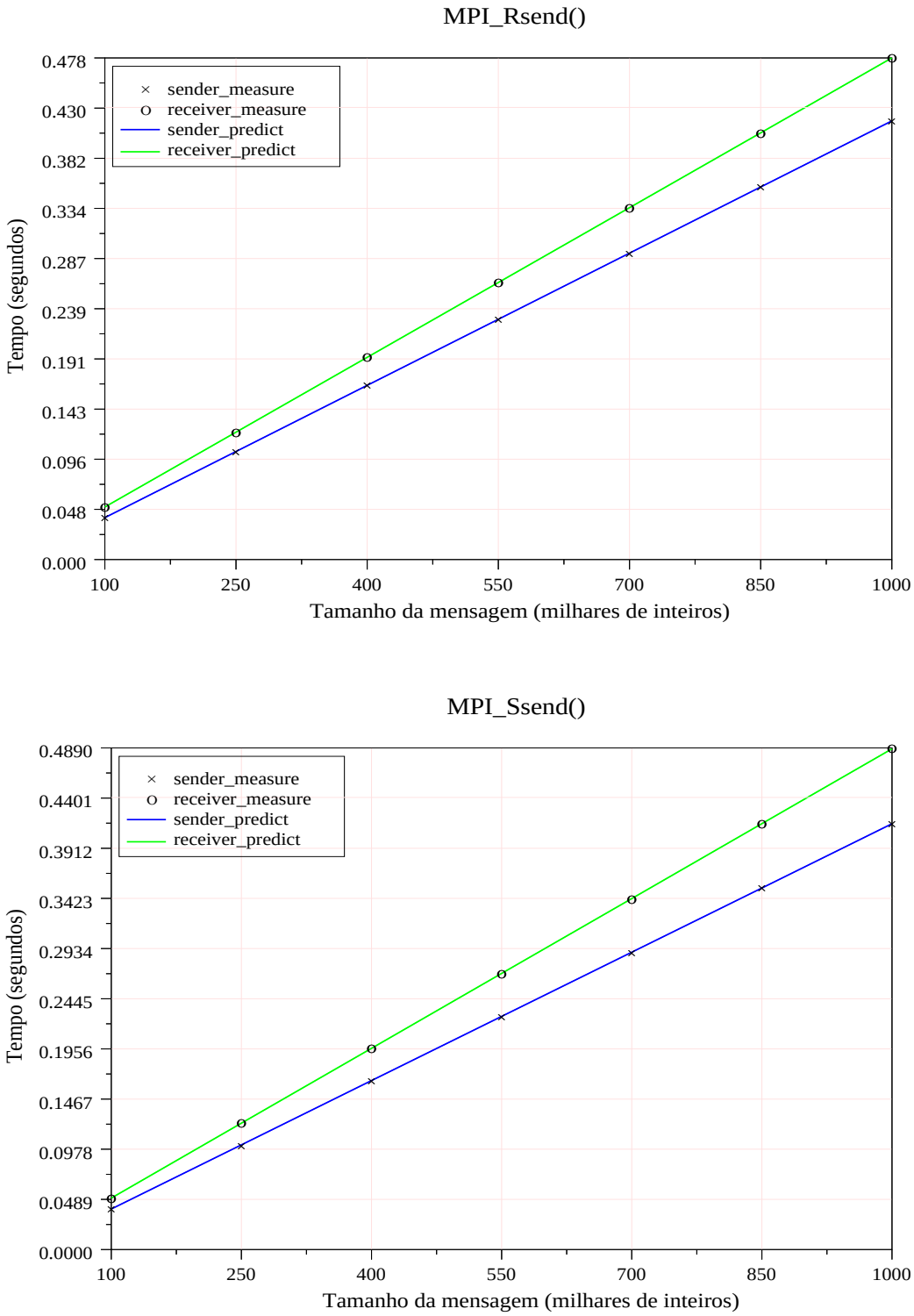


Figura 4.2: Desempenho do MPI_Rsend e do MPI_Ssend.

# Capítulo 5

# PRIMITIVAS DE COMUNICAÇÃO COLETIVA

## 5.1 Introdução

A modelagem de primitivas ponto-a-ponto descrita no capítulo anterior foi concretizada com o uso de equações lineares. Na maioria dos casos, os tempos de comunicação expressos pelos modelos refletiam, basicamente, o tempo despendido na transmissão em si da mensagem ($t_c(n) = t_e(n) + t_t(n) + t_r(n)$), sendo influenciados por fatores como a quantidade de elementos e as características da rede de interconexão. Porém, em comunicações coletivas outros fatores precisam ser considerados.

Durante a execução de uma primitiva de comunicação coletiva, vários processos enviam ou recebem mensagens simultaneamente. Dependendo da semântica da operação, podemos utilizar diferentes implementações para realizar a distribuição dos dados. Assim, o algoritmo adotado constitui um importante aspecto a ser estudado no processo de análise e modelagem.

Por vezes, nos deparamos com o uso de primitivas ponto-a-ponto na implementação de operações coletivas. Devido ao fato das comunicações ponto-a-ponto apresentarem crescimento linear no tamanho da mensagem, conforme descrevemos neste capítulo tal comportamento pode ser também observado no desempenho de operações coletivas.

Sendo o número de processos envolvidos na comunicação variável, é interessante a inclusão deste novo parâmetro nas equações elaboradas. Desta forma, procuramos caracterizar as primitivas coletivas MPI estudadas com o desenvolvimento de modelos analíticos em função de duas variáveis: o tamanho da mensagem e o número de processos.

As seções seguintes deste capítulo apresentam análises e modelos de predição de desempenho das operações *broadcast*, *reduce* e barreira. Além destas primitivas, previstas nos objetivos do nosso trabalho, descrevemos um estudo do MPI_Alltoall, expondo análises e testes realizados.

## 5.2 *Broadcast*

### 5.2.1 Considerações sobre a implementacão do *broadcast*

Uma forma possível de implementação da primitiva *broadcast* é através do uso de algoritmos baseados em árvores ou hipercubos. Para implementar o MPI_Bcast, o LAM-MPI utiliza um algoritmo linear para operações com até 3 escravos e outro de complexidade $O(log\,p)$ para 4 ou mais escravos, criando um hipercubo com os processos envolvidos na comunicação. Tal constatação pôde ser feita ao analisarmos o código fonte do LAM-MPI e a implementação da função MPI_Bcast.

Assim, quando o número de escravos for igual ou superior a 4 o algoritmo prossegue da seguinte maneira. Em um primeiro momento a mensagem é enviada pelo processo mestre ao primeiro escravo. Após isso, o mestre faz a transmissão para outro escravo e, simultaneamente, aquele escravo que havia recebido a mensagem a retransmite para um terceiro escravo. Assim, os dados vão sendo enviados paralelamente até que todos os processos tenham recebido a mensagem. A dimensão do hipercubo formado depende do número de processos envolvidos na operação de comunicação.

### 5.2.2 Testes executados com o *broadcast*

Assim como no estudo das primitivas ponto-a-ponto bloqueantes, os testes com o *broadcast* também foram realizados no ambiente computacional apresentado na

seção 4.4. Foram utilizadas ambas as versões LAM-MPI disponíveis (6.4 e 6.5.6).

Os testes envolveram diversas configurações, variando o número de nós de processamento envolvidos e o tamanho das mensagens, sendo cada execução repetida 30 ou, na maioria dos casos, 50 vezes. As mensagens utilizaram as seguintes quantidades de elementos do tipo MPI_INT: 100000, 200000, 300000, 400000, 500000, 600000, 700000, 800000, 900000 e 1000000, ao passo que o número de processos receptores variou em 1, 2, 3, 4, 6, 7, 8, 12, 14, 15 e 16. Conforme mencionado na subseção 4.2.2, os processos eram alocados a máquinas distintas. Podemos conferir os resultados obtidos na Tabela 5.3, exposta ao final desta seção.

### 5.2.3 Análise em função do tamanho da mensagem

Os tempos medidos com a operação *broadcast* (Tabela 5.3) nos permitem analisar e modelar o desempenho da primitiva. Podemos notar um comportamento linear do MPI_Bcast em função do tamanho da mensagem (*n*) para todas as quantidades de escravos envolvidos. Este comportamento pode ser melhor visualizado na Figura 5.1, que exibe o desempenho da primitiva variando *n*.

Aplicando o método de ajuste de curvas descrito na subseção 4.2.3, elaboramos os seguintes modelos analíticos, em função do tamanho da mensagem, para o processo emissor da operação *broadcast*:

- 1 escravo: $t_{bcast}(n) = -0,0020199 + (3,398 * 10^{-7}) * n$
- 2 escravos: $t_{bcast}(n) = -0,0030569 + (6,799 * 10^{-7}) * n$
- 3 escravos: $t_{bcast}(n) = -0,0049747 + (10 * 10^{-7}) * n$
- 4 escravos: $t_{bcast}(n) = -0,0051941 + (10 * 10^{-7}) * n$
- 8 escravos: $t_{bcast}(n) = 0,0028577 + (14 * 10^{-7}) * n$
- 12 escravos: $t_{bcast}(n) = -0,0003569 + (14 * 10^{-7}) * n$
- 14 escravos: $t_{bcast}(n) = -0,0036268 + (14 * 10^{-7}) * n$
- 15 escravos: $t_{bcast}(n) = 0,0002058 + (14 * 10^{-7}) * n$

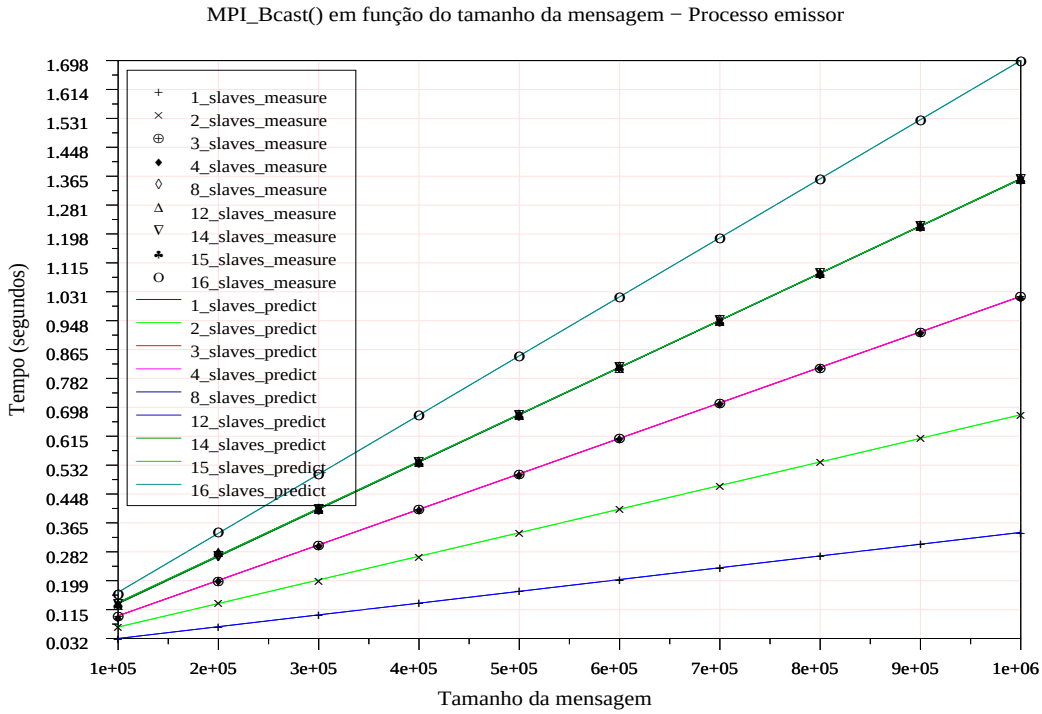


Figura 5.1: Análise do MPI_Bcast em função do tamanho da mensagem.

- 16 escravos: $t_{bcast}(n) = -0,0050184 + (17 * 10^{-7}) * n$

O comportamento destes modelos também é representado no gráfico da Figura 5.1. Em uma análise do gráfico exposto, podemos perceber que as curvas referentes a 8, 12, 14 e 15 escravos são sobrepostas, assim como as curvas de 3 e 4 escravos. O segundo caso é provavelmente ocasionado pelo fato de ambos os algoritmos utilizados ($p = 3$ e $p = 4$) estabelecerem 3 etapas na comunicação, ocorrendo uma transmissão da mensagem em cada uma delas. Contudo, o primeiro caso é explicado pelo uso do algoritmo baseado em hipercubo. A proximidade dos tempos é claramente expressa se compararmos os respectivos valores na Tabela 5.3.

### 5.2.4 Análise em função do número de escravos

Conforme mencionamos no início desta seção, o algoritmo utilizado pelo LAM-MPI na implementação da operação *broadcast* com mais de 3 escravos apresenta

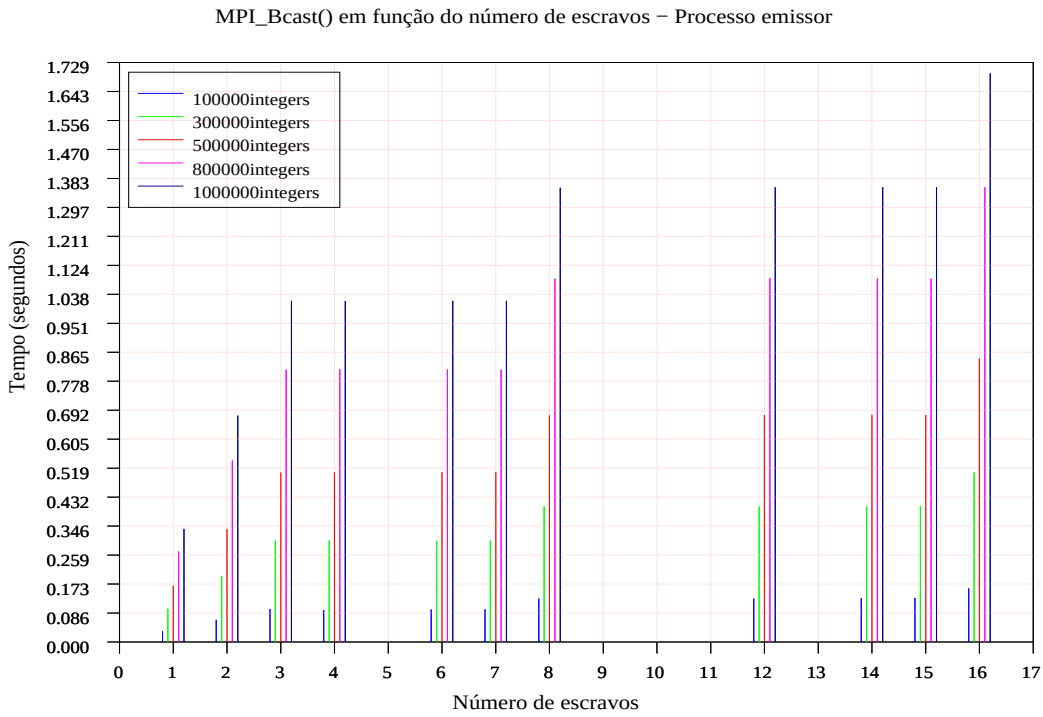


Figura 5.2: Análise do MPI_Bcast em função do número de escravos.

complexidade $O(log\ p)$. A Figura 5.2 exibe o gráfico da primitiva MPI_Bcast em função do número de escravos envolvidos na comunicação, para cada um dos tamanhos de mensagem utilizados em nossos testes.

Podemos observar dois saltos, ou degraus, nas curvas de desempenho: o primeiro entre 7 e 8 escravos e o segundo entre 15 e 16. Os valores destes degraus quase não apresentam variação e correspondem, aproximadamente, aos tempos de transmissão da mensagem entre dois processos. Considerando as curvas das mensagens de 400000, 800000 e 1000000 inteiros, por exemplo, os valores aproximados de seus degraus equivalem a 0,136, 0,27 e 0,34. Tais valores podem ser calculados com auxílio da Tabela 5.3 e, observando a mesma tabela, podemos verificar que eles se assemelham bastante aos tempos despendidos pelo mestre para efetuar o *broadcast* quando apenas um escravo era envolvido.

### 5.2.5 Modelo analítico do *broadcast* e predição de desempenho

Como observamos anteriormente, o MPI _Bcast apresenta um comportamento linear no tamanho da mensagem (*n*) e logarítmico no número de escravos envolvidos (*p*). Considerando tais comportamentos e a partir dos tempos medidos, foi possível a elaboração de um modelo analítico em função de *n* e *p* para o tempo do processo emissor quando 4 ou mais escravos participam da comunicação.

Em um primeiro momento, aplicamos as técnicas de ajuste de curvas sobre os tempos medidos com 4, 8 e 16 escravos e obtivemos o conjunto de equações para o tempo do mestre apresentado na Tabela 5.1.

Tabela 5.1: Modelos do MPI_Bcast em função do número de escravos (*p*).

| Número de inteiros | $t_{bcast}(p) = c_0 + c_1 * \lfloor log_2 p \rfloor$ |
|---|---|
| 100000 | $0,0309941 + 0,0324253 * \lfloor log_2 p \rfloor$ |
| 200000 | $0,0600513 + 0,0701442 * \lfloor log_2 p \rfloor$ |
| 300000 | $0.0999419 + 0,1016122 * \lfloor log_2 p \rfloor$ |
| 400000 | $0,1329724 + 0,1360371 * \lfloor log_2 p \rfloor$ |
| 500000 | $0,1666606 + 0,1699918 * \lfloor log_2 p \rfloor$ |
| 600000 | $0,2021226 + 0,203741 * \lfloor log_2 p \rfloor$ |
| 700000 | $0,2366444 + 0,2375567 * \lfloor log_2 p \rfloor$ |
| 800000 | $0,2694978 + 0,2718298 * \lfloor log_2 p \rfloor$ |
| 900000 | $0,3049658 + 0,3052317 * \lfloor log_2 p \rfloor$ |
| 1000000 | $0,3367411 + 0,3399822 * \lfloor log_2 p \rfloor$ |

Para cada tamanho de mensagem, modelamos o tempo do *broadcast* através da equação $t_{bcast}(p) = c_0 + c_1 * \lfloor log_2 p \rfloor$, sendo a função piso utilizada para ajustar os tempos conforme a quantidade de escravos considerada. Os valores das constantes $c_0$ e $c_1$ são bastante semelhantes aos tempos despendidos pela primitiva nas configurações com apenas um escravo (Tabela 5.3). Esta semalhança deve-se à influência do algoritmo baseado em hipercubo usado na implementação do MPI_Bcast, pois tais constantes correspondem, aproximadamente, ao tempo necessário para que o mestre transmita a mensagem a um outro processo. Dependendo do número de escravos envolvidos e, conseqüentemente, da dimensão do hipercubo formado, para levar os dados aos demais escravos o sistema deverá gastar o tempo equivalente a $\lfloor log_2 p \rfloor$ novas transmissões.

Após a construção dos modelos expostos na Tabela 5.1, aplicamos novamente a técnica de ajuste de curvas e modelamos as constantes $c_0$ e $c_1$ em função de *n*. Assim, expressões lineares foram elaboradas para representar estas constantes e o seguinte modelo analítico do *broadcast* em *n* e *p* pode ser obtido:

$$\begin{aligned} t_{bcast}(n,p) &= (a_0 + a_1 * n) + (a_2 + a_3 * n) * \lfloor log_2 p \rfloor \\ t_{bcast}(n,p) &= (-0,0046202 + 3,431 * 10^{-7} * n) + \\ &\quad (0,0000878 + 3,396 * 10^{-7} * n) * \lfloor log_2 p \rfloor \end{aligned} \tag{5.1}$$

**Predição de desempenho**

Através do modelo 5.1 em função de *n* e *p*, podemos realizar predições de tempos de comunicação da primitiva MPI_Bcast para o processo mestre. A Tabela 5.2 ilustra a acurácia do modelo exibindo uma comparação entre tempos preditos e medidos com 7 e 12 escravos:

Tabela 5.2: Predição de desempenho em segundos da primitiva MPI _Bcast.

| $10^3$ | 7 escravos | | | 12 escravos | | |
|---|---|---|---|---|---|---|
| inteiros | Predição | Medição | erro(%) | Predição | Medição | erro(%) |
| 100 | 0,0977854 | 0,0977026 | 0,0847481 | 0,1318332 | 0,1297542 | 1,6022612 |
| 400 | 0,4044754 | 0,4046213 | -0,0360580 | 0,5404032 | 0,5407695 | -0,0677387 |
| 700 | 0,7111654 | 0,7114338 | -0,0377294 | 0,9489732 | 0,9487701 | 0,0214074 |
| 1000 | 1,0178554 | 1,0178646 | -0,0009012 | 1,3575432 | 1,3575507 | -0,0005541 |

Como podemos conferir, os erros percentuais foram pequenos em todos os tamanhos de mensagens utilizados. Infelizmente, devido ao número de máquinas do nosso *cluster*, o número de escravos envolvidos na comunicação teve de ser limitado em 16, pois nos testes executados apenas um processo era alocado por nó de processamento. Assim, com os nós homegêneos e a máquina paralela disponíveis no ambiente de teste (conforme descrito na seção 4.4) limitamos a quantidade total de processos em 17.

Tabela 5.3: Tempos medidos em segundos com a primitiva MPI_Bcast.

| Número de escravos | Tamanho da mensagem ($10^3$ inteiros) | | | | |
|---|---|---|---|---|---|
| | 100 | 200 | 300 | 400 | 500 |
| 1 | 0,0321932 | 0,0657033 | 0,0997402 | 0,1341129 | 0,1680622 |
| 2 | 0,0662090 | 0,1337528 | 0,1973767 | 0,2684709 | 0,3376312 |
| 3 | 0,0982296 | 0,1979974 | 0,3025457 | 0,4045943 | 0,5056477 |
| 4 | 0,0951482 | 0,1998954 | 0,3029085 | 0,4048234 | 0,5064152 |
| 6 | 0,0967899 | 0,2005406 | 0,3021767 | 0,4046339 | 0,5065627 |
| 7 | 0,0977026 | 0,2001609 | 0,3027913 | 0,4046213 | 0,5068202 |
| 8 | 0,1296627 | 0,2713725 | 0,4052939 | 0,5415298 | 0,6770941 |
| 12 | 0,1297542 | 0,2812076 | 0,4049071 | 0,5407695 | 0,6774760 |
| 14 | 0,1304026 | 0,2705197 | 0,4051909 | 0,5405236 | 0,6775840 |
| 15 | 0,1311611 | 0,2807639 | 0,4054793 | 0,5404646 | 0,6773785 |
| 16 | 0,1599987 | 0,3401838 | 0,5061329 | 0,6768975 | 0,8463988 |

| Número de escravos | Tamanho da mensagem ($10^3$ inteiros) | | | | |
|---|---|---|---|---|---|
| | 600 | 700 | 800 | 900 | 1000 |
| 1 | 0,2015452 | 0,2358408 | 0,2701916 | 0,3039462 | 0,3376195 |
| 2 | 0,4053886 | 0,4733735 | 0,5416239 | 0,6093149 | 0,6757295 |
| 3 | 0,6091327 | 0,7110549 | 0,8128486 | 0,9159813 | 1,0178558 |
| 4 | 0,6096469 | 0,7118614 | 0,8136966 | 0,9157627 | 1,0172122 |
| 6 | 0,6094756 | 0,7108951 | 0,8134946 | 0,9159775 | 1,0180799 |
| 7 | 0,6093632 | 0,7114338 | 0,8126001 | 0,9159853 | 1,0178646 |
| 8 | 0,8132610 | 0,9491074 | 1,0839087 | 1,2199937 | 1,355674 |
| 12 | 0,8123119 | 0,9487701 | 1,0856632 | 1,220401 | 1,3575507 |
| 14 | 0,8137130 | 0,9496697 | 1,0848826 | 1,2219815 | 1,3572618 |
| 15 | 0,8135913 | 0,9484307 | 1,0843244 | 1,2218329 | 1,3571228 |
| 16 | 1,0171289 | 1,1869748 | 1,3573562 | 1,526226 | 1,6971765 |

## 5.3 *Reduce*

### 5.3.1 Considerações sobre a implementação do *reduce*

A implementação da primitiva MPI_Reduce, considerando o LAM-MPI, é parecida com o caso do *broadcast*. Novamente, através de análises do código da operação, observamos o uso de um algoritmo de complexidade linear quando são envolvidos até 3 escravos na comunicação e um algoritmo $O(log\ p)$ nos demais casos. O envio e o recebimento dos dados são baseados nas operações ponto-a-ponto *send* e *receive* e a dimensão do hipercubo, formado quando existem 4 ou mais escravos participando da comunicação, depende justamente do número de escravos envolvidos.

Quando é adotado o algoritmo linear, o mestre entra em um laço no qual a cada iteração dados são recebidos e a operação de redução é executada. Com o algoritmo de complexidade $O(log\ p)$, paralelamente alguns processos enviam dados a outros que recebem e realizam a redução destes dados. A comunicação e a subseqüente redução são repetidas até a conclusão da primitiva MPI_Reduce, quando o resultado final pode ser encontrado no *buffer* do mestre.

Porém, o processamento envolvido na redução dos dados adiciona um custo computacional extra à primitiva. A operação de redução executada pode ser uma das pré-definidas pelo padrão MPI ou uma outra definida pelo usuário. Nos testes realizados, utilizamos a operação pré-definida de soma, MPI_SUM, que apresenta complexidade linear no tamanho da mensagem e possui um custo computacional relativamente pequeno, pois constitui apenas uma soma de vetores. Desta forma, em nossos testes o tempo despendido na execução da primitiva MPI_Reduce foi determinado principalmente pelo tempo de comunicação em si.

### 5.3.2 Testes executados com o *reduce*

Os testes com a primitiva MPI_Reduce, realizados no sistema computacional apresentado na seção 4.4 com o LAM 6.5.6, utilizaram as seguintes quantidades de escravos: 2, 3, 4, 6, 8, 12 e 16. Foram executados testes com mensagens de 100000, 200000, 300000, 400000, 500000, 600000, 700000, 800000, 900000 e 1000000 inteiros. O número de repetições com cada configuração variou entre

Tabela 5.4: Tempos medidos em segundos com a primitiva MPI_Reduce.

| $10^3$ inteiros | Número de escravos | | | | | | |
|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 6 | 8 | 12 | 16 |
| 100 | 0,110004 | 0,158755 | 0,170498 | 0,173004 | 0,228302 | 0,230216 | 0,282141 |
| 200 | 0,212065 | 0,308099 | 0,330471 | 0,334748 | 0,442248 | 0,446571 | 0,548434 |
| 300 | 0,313124 | 0,456439 | 0,493591 | 0,500616 | 0,66121 | 0,66453 | 0,818264 |
| 400 | 0,415041 | 0,606389 | 0,652259 | 0,660781 | 0,876636 | 0,881875 | 1,078851 |
| 500 | 0,517274 | 0,760023 | 0,821215 | 0,831634 | 1,103612 | 1,108921 | 1,355867 |
| 600 | 0,620051 | 0,909924 | 0,975855 | 1,02075 | 1,308685 | 1,317672 | 1,610075 |
| 700 | 0,72161 | 1,058296 | 1,138025 | 1,145589 | 1,521236 | 1,531353 | 1,872243 |
| 800 | 0,827991 | 1,20759 | 1,300196 | 1,311219 | 1,737046 | 1,748567 | 2,137439 |
| 900 | 0,923458 | 1,34575 | 1,464809 | 1,477634 | 1,955013 | 1,967034 | 2,40843 |
| 1000 | 0,997512 | 1,491655 | 1,63699 | 1,66741 | 2,19278 | 2,210384 | 2,703436 |

30 e 50 e os resultados podem ser conferidos na Tabela 5.4.

### 5.3.3 Análise em função do tamanho da mensagem

A Figura 5.3 reflete o comportamento do MPI_Reduce em decorrência de alterações no tamanho da mensagem, para diferentes quantidades de escravos envolvidos. Podemos constatar o crescimento linear no tempo de comunicação através do gráfico ou analisando os valores apresentados na Tabela 5.4.

Além dos valores medidos com o *reduce*, a Figura 5.3 ilustra curvas construídas com o auxílio das equações lineares listadas a seguir:

- 2 escravos: $t_{reduce}(n) = 0,0136753 + (10 * 10^{-7}) * n$
- 3 escravos: $t_{reduce}(n) = 0,0129248 + (15 * 10^{-7}) * n$
- 4 escravos: $t_{reduce}(n) = 0,0055991 + (16 * 10^{-7}) * n$
- 6 escravos: $t_{reduce}(n) = 0,0074579 + (16 * 10^{-7}) * n$
- 8 escravos: $t_{reduce}(n) = 0,0097495 + (22 * 10^{-7}) * n$
- 12 escravos: $t_{reduce}(n) = 0,0093086 + (22 * 10^{-7}) * n$

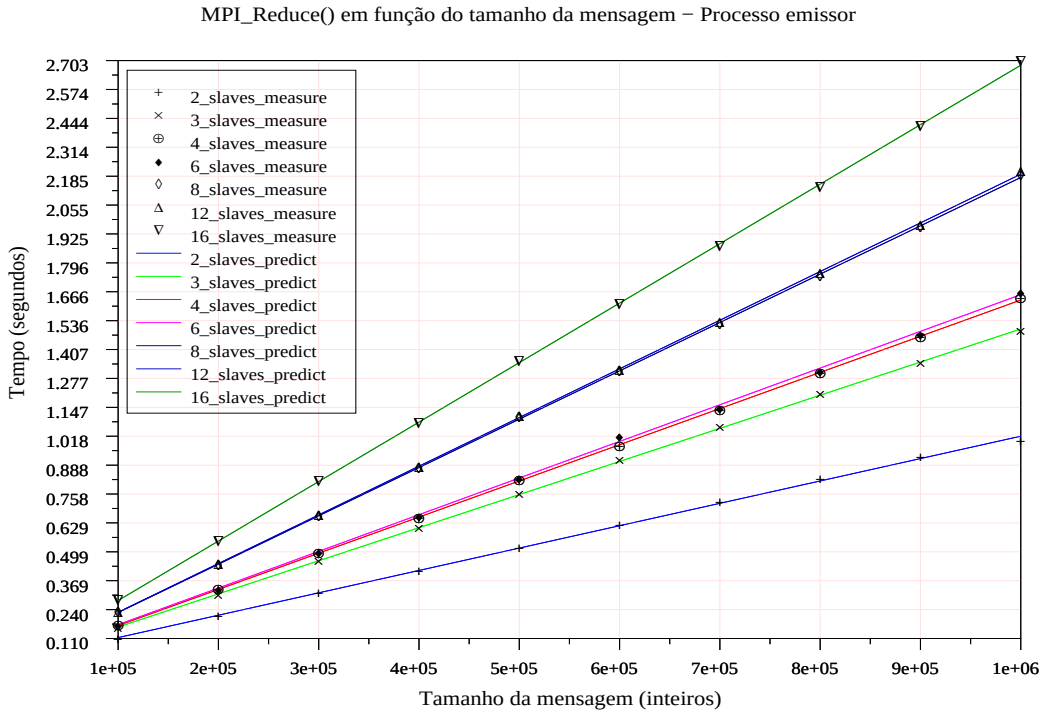


Figura 5.3: Análise do MPI_Reduce em função do tamanho da mensagem.

### 5.3.4 Análise em função do número de escravos

Ao fixarmos o tamanho da mensagem transmitida, variando apenas o número de escravos envolvidos, observamos a influência determinante do algoritmo no tempo de comunicação do *reduce*. Como discutimos no início da seção, o algoritmo adotado pela implementação LAM da primitiva MPI_Reduce apresenta complexidade linear até 3 escravos e logarítmica acima deste número.

O gráfico ilustrado na Figura 5.4 deixa evidente este comportamento. Analisando somente as barras verticais relativas a 4, 8 e 16 escravos podemos perceber um crescimento logarítmico para todos os tamanhos de mensagem. Contudo, nos casos de 6 e 12 escravos os valores apresentados foram bastante semelhantes aos de 4 e 8, respectivamente, indicando um comportamento constante do MPI_Reduce de 4 a 7 e de 8 a 15 escravos. Tal constância pode ser atribuída ao algoritmo adotado em sua implementação, baseado em um hipercubo, segundo o qual o mestre recebe mensagens de somente $\lfloor log_2 p \rfloor$ escravos e realiza a redução sobre estes dados.

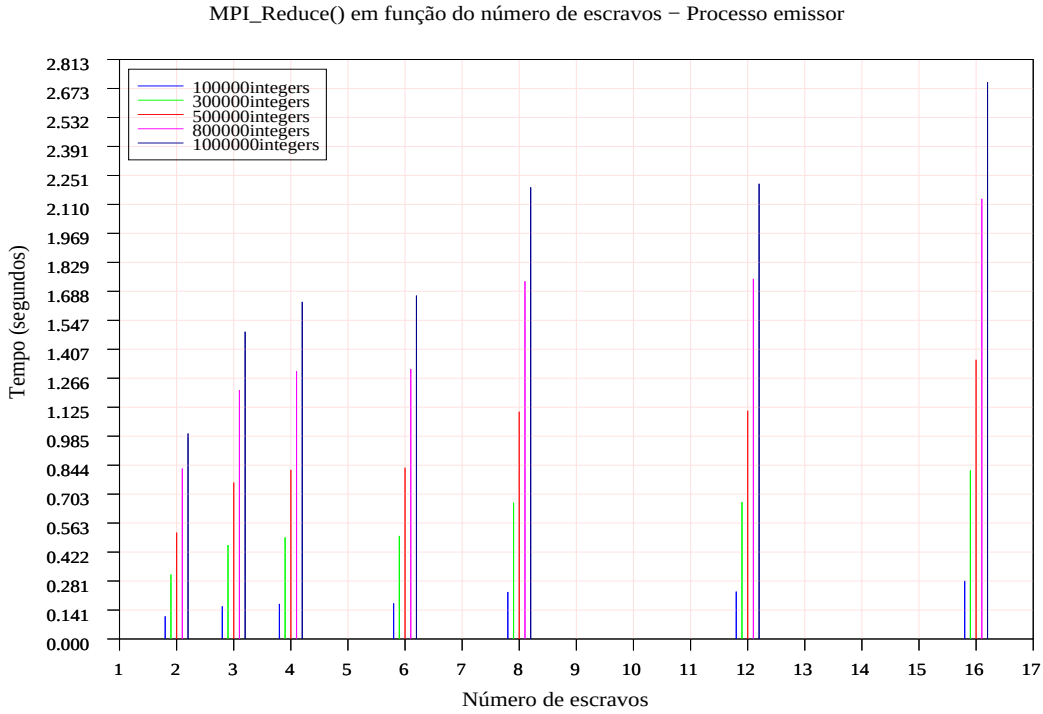


Figura 5.4: Análise do MPI_Reduce em função do número de escravos.

### 5.3.5 Modelo analítico do *reduce* e predição de desempenho

O MPI_Reduce e o MPI_Bcast apresentam comportamentos parecidos devido às semelhanças em seus algoritmos. De forma análoga à modelagem da primitiva *broadcast*, o *reduce* pode ser caracterizado com uma equação linear no tamanho da mensagem e logarítmica no número de escravos. Assim, na obtenção de seu modelo executamos o mesmo procedimento adotado na modelagem do *broadcast*, primeiramente construindo equações logarítmicas em $p$ e, a partir destas equações, gerando o modelo em função de $n$ e $p$.

A Tabela 5.5 exibe as equações logarítmicas em $p$, geradas com os tempos medidos nos testes utilizando 4, 8 e 16 escravos.

O gráfico do MPI_Reduce exibido na Figura 5.4, bem como a análise em função do número de escravos apresentada anteriormente, indicam comportamentos constantes do *reduce* em determinados intervalos de $p$. Por exemplo, de 4 a 7 e de 8 a 15 escravos a variação no tempo de *reduce* foi bastante pequena, podendo ser atribuída a oscilações no desempenho da rede de interconexão ou dos nós de pro-

Tabela 5.5: Modelos do MPI_Reduce em função do número de escravos (*p*).

| Número de inteiros | $t_{reduce}(p) = c_0 + c_1 * \lfloor log_2 p \rfloor$ |
|---|---|
| 100000 | $t_{reduce}(p) = 0,0595166 + 0,0558214 * \lfloor log_2 p \rfloor$ |
| 200000 | $t_{reduce}(p) = 0,1134395 + 0,1089816 * \lfloor log_2 p \rfloor$ |
| 300000 | $t_{reduce}(p) = 0,170679 + 0,1623366 * \lfloor log_2 p \rfloor$ |
| 400000 | $t_{reduce}(p) = 0,2293606 + 0,213296 * \lfloor log_2 p \rfloor$ |
| 500000 | $t_{reduce}(p) = 0,2915878 + 0,2673256 * \lfloor log_2 p \rfloor$ |
| 600000 | $t_{reduce}(p) = 0,3468758 + 0,3171097 * \lfloor log_2 p \rfloor$ |
| 700000 | $t_{reduce}(p) = 0,4091738 + 0,3671091 * \lfloor log_2 p \rfloor$ |
| 800000 | $t_{reduce}(p) = 0,4690298 + 0,4186214 * \lfloor log_2 p \rfloor$ |
| 900000 | $t_{reduce}(p) = 0,5273167 + 0,4718116 * \lfloor log_2 p \rfloor$ |
| 1000000 | $t_{reduce}(p) = 0,5780735 + 0,5332229 * \lfloor log_2 p \rfloor$ |

cessamento. Devido à existência destes intervalos, os modelos do MPI_Reduce apresentados na Tabela 5.5 utilizam a função piso para ajustar os tempos preditos conforme o número de escravos envolvidos na comunicação.

Como podemos observar analisando a Tabela 5.5, as constantes $c_0$ e $c_1$ apresentam crescimento linear no tamanho da mensagem. Assim, utilizando equações lineares em *n* para estimar os valores aproximados destas constantes, obtivemos o seguinte modelo analítico para o tempo do processo mestre:

$$\begin{aligned} t_{reduce}(n,p) &= (a_0 + a_1 * n) + (a_2 + a_3 * n) * \lfloor log_2 p \rfloor \\ t_{reduce}(n,p) &= (-0,0021825 + 5,849 * 10^{-7} * n) + \\ &\quad (0,0039279 + 5,23 * 10^{-7} * n) * \lfloor log_2 p \rfloor \end{aligned} \tag{5.2}$$

**Predição de desempenho**

Com a aplicação do modelo analítico do MPI_Reduce é possível predizermos tempos de execução da primitiva para outros tamanhos de mensagens ou números de escravos. Confirmando a validade da modelagem realizada, a Tabela 5.6 apresenta tempos medidos e preditos, bem como os respectivos erros percentuais, para testes variando *n* e *p*.

Em todos os testes realizados objetivando a obtenção de modelos para as pri-

Tabela 5.6: Predição de desempenho da primitiva MPI _Reduce *.

| $10^3$ inteiros | 18 escravos | | | 19 escravos | | |
|---|---|---|---|---|---|---|
| | Predição | Medição | erro(%) | Predição | Medição | erro(%) |
| 50 | 0,1473741 | 0,1468832 | 0,3342064 | 0,1473741 | 0,1479854 | -0,4130820 |
| 70 | 0,2009121 | 0,2012079 | -0,1470167 | 0,2009121 | 0,2015464 | -0,3147186 |
| 1700 | 4,564259 | 4,5628476 | 0,0309326 | 4,564259 | 4,55656 | 0,1689648 |
| 2000 | 5,367329 | 5,3658528 | 0,0275105 | 5,367329 | 5,3570719 | 0,1914690 |

* tempos em segundos.

mitivas de comunicação MPI, conforme mensionado na seção 4.2, apenas um processo era alojado por máquina. Contudo, as predições de desempenho do *reduce* exibidas na Tabela 5.6 incluem até 19 escravos, excedendo o número de nós de processamento disponíveis em nosso *cluster*. Para isso, foi necessário alojarmos mais de um escravo na máquina paralela (os demais nós receberam somente um processo) [1].

Embora isso possa, teoricamente, aumentar os *overheads* de envio e recepção nas operações de comunicação envolvendo a máquina paralela, os resultados foram bastante satisfatórios. Somente processos escravos eram direcionados a esta máquina e os erros entre os valores preditos e os tempos medidos ficaram abaixo de 1%, inclusive nas predições com mensagens menores.

## 5.4 *Barrier*

### 5.4.1 Considerações sobre a implementação da barreira

Na execução de uma barreira, a sincronização pode ser realizada através de trocas de pequenas mensagens entre os processos participantes da comunicação. Analisando o código do LAM-MPI, observamos que estas mensagens constituem apenas sinais de reconhecimento, transmitindo 0 elementos do tipo MPI_BYTE, e são trocadas fazendo uso das primitivas MPI_Send e MPI_Recv (modo padrão do *send* bloqueante). Ao executar a primitiva MPI_Barrier, o processo mestre (*rank*

[1]Nas execuções com 19 escravos, por exemplo, o mestre e os 15 primeiros escravos eram alojados nos nós monoprocessados e os 4 restantes na máquina paralela.

igual a 0) primeiramente recebe e depois envia um sinal aos demais.

O LAM-MPI utiliza um algoritmo de complexidade linear e outro $O(log\ p)$ na implentação do MPI_Barrier. Quando a comunicação envolve um número de até 4 processos, o mestre executa dois laços: no primeiro ele recebe e no segundo envia sinais de reconhecimento aos demais processos. Se o número for superior a 4, um hipercubo igual ao criado no caso do *broadcast* é utilizado e o mestre recebe e envia sinais a $\lfloor log_2 p \rfloor$ outros processos, que ficam encarregados de se comunicarem com os demais participantes da operação de barreira.

### 5.4.2 Testes executados com barreiras

A Tabela 5.7 apresenta os resultados obtidos nos testes com o MPI_Barrier, variando o número de processos entre 2 e 17:

Tabela 5.7: Tempos medidos nos testes com a primitiva MPI_Barrier *.

| p | Tempo | p | Tempo |
|---|---|---|---|
| 2 | 0,0001609 | 10 | 0,0004684 |
| 3 | 0,000229 | 11 | 0,000462 |
| 4 | 0,0002432 | 12 | 0,000486 |
| 5 | 0,00031 | 13 | 0,0004912 |
| 6 | 0,0003350 | 14 | 0,0004986 |
| 7 | 0,0003453 | 15 | 0,0005170 |
| 8 | 0,0004602 | 16 | 0,0006218 |
| 9 | 0,0004674 | 17 | 0,0006244 |

* p = número de processos, tempo em segundos.

Como as mensagens trocadas durante a execução do MPI_Barrier constituem apenas sinais de reconhecimento, os tempos de comunicação tendem a ser bastante pequenos. Conseqüentemente, as oscilações apresentadas costumam ser maiores que nos testes executados com o *broadcast* e o *reduce*, por exemplo, nos quais as mensagens e os tempos são muito maiores.

Procurando minimizar tais oscilações, nos testes com a operação de barreira adotamos um número maior de repetições (100 execuções) e forçamos uma sincronização explícita dos processos antes de medirmos o tempo da operação. No

programa elaborado para a medição da barreira, a primitiva MPI_Barrier é chamada duas vezes, sendo a primeira utilizada apenas para garantir que os processos realmente estejam sincronizados antes da medição. Desta forma, podemos evitar a influência de eventuais atrasos provenientes da criação dos processos. Nos testes com as demais primitivas de comunicação não foi necessário executar a sincronização prévia pois, considerando que os tempos eram muito maiores, estes atrasos não são capazes de provocar grandes alterações percentuais nos tempos medidos.

### 5.4.3 Análise em função do número de processos

O gráfico da Figura 5.5 ilustra o desempenho do MPI_Barrier em função do número de processos (*p*). Podemos notar um comportamento semelhante ao apresentado pelas primitivas *broadcast* e *reduce*, principalmente a partir de 5 processos, quando visualizamos três degraus distintos ($5 \leq p \leq 7$, $8 \leq p \leq 15$ e $p \geq 16$).

Em virtude dos valores serem pequenos, observamos na curva do MPI_Barrier alguns pontos nos quais o comportamento da primitiva não se mostrou uniforme. Por exemplo, devido aos algoritmos utilizados na implementação da operação de barreira, espera-se um desempenho linear entre 2 e 4 processos e constante nos intervalos de 5 a 7 e de 8 a 15, contudo, a curva exibe oscilações no tempo da primitiva. Embora tais variações tenham ocorrido no comportamento do MPI_Barrier, seu desempenho pode ser modelado através de uma equação logarítmica para $p \geq 5$, como é possível notar analisando os pontos referentes a 5, 8 e 16 processos.

### 5.4.4 Modelo analítico

O gráfico do MPI_Barrier, elaborado a partir dos tempos medidos (Figura 5.5), exibe um comportamento logarítmico em função do número de processos. Desta forma, através de uma equação logarítmica podemos modelar adequadamente o desempenho da primitiva e demonstrar a influência do algoritmo usado em sua implementação.

Conforme discutimos na subseção 5.4.1, quando mais de 4 processos são envolvidos na comunicação o mestre primeiramente recebe e depois envia mensagens a $\lfloor log_2 p \rfloor$ escravos. Neste caso os tempos de *send* e *receive* são considerados

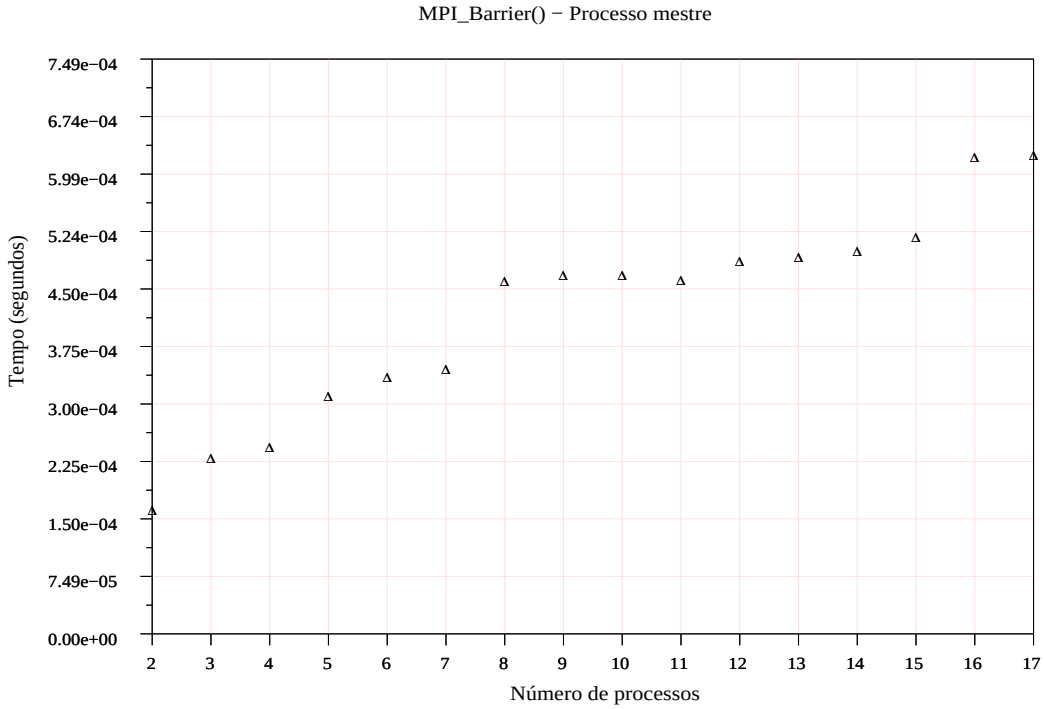


Figura 5.5: Análise do MPI_Barrier em função do número de processos.

constantes, pois todas as mensagens transmitem 0 elementos, e o desempenho do MPI_Barrier pode ser caracterizado através da equação:

$$\begin{aligned} t_{barrier}(p) &= \lfloor log_2 p \rfloor * t_{receive}(k) + \lfloor log_2 p \rfloor * t_{send}(k) + \alpha \\ &= \lfloor log_2 p \rfloor * (t_{receive}(k) + t_{send}(k)) + a_0 = a_0 + \lfloor log_2 p \rfloor * a_1 \end{aligned} \tag{5.3}$$

A constante indicada por $\alpha$ representa algum *overhead* ocasionado pela execução da primitiva, podendo ser decorrente, por exemplo, de pequenas operações executadas. Considerando esta equação e a partir dos tempos coletados nos testes com o MPI_Barrier, elaboramos o seguinte modelo analítico para o tempo do processo mestre:

$$t_{barrier}(p) = 0,0000463 + 0,0001441 * \lfloor log_2 p \rfloor \tag{5.4}$$

Tabela 5.8: Acurácia do modelo analítico da primitiva MPI_Barrier *.

| Processos | Predição | Medição | erro (%) |
|---|---|---|---|
| 5 | 0,0003345 | 0,00031 | 7,903 |
| 6 | 0,0003345 | 0,0003350 | -0,149 |
| 7 | 0,0003345 | 0,0003453 | -3,128 |
| 8 | 0,0004786 | 0,0004602 | 3,998 |
| 9 | 0,0004786 | 0,0004674 | 2,396 |
| 10 | 0,0004786 | 0,0004684 | 2,178 |
| 11 | 0,0004786 | 0,000462 | 3,593 |
| 12 | 0,0004786 | 0,000486 | -1,523 |
| 13 | 0,0004786 | 0,0004912 | -2,565 |
| 14 | 0,0004786 | 0,0004986 | -4,011 |
| 15 | 0,0004786 | 0,0005170 | -7,427 |
| 16 | 0,0006227 | 0,0006218 | 0,145 |
| 17 | 0,0006227 | 0,0006244 | -0,272 |

* tempos em segundos

Na construção do modelo, calculamos as médias dos tempos para cada intervalo ($5 \leq p \leq 7$, $8 \leq p \leq 15$ e $p \geq 16$) e aplicamos o método de ajuste de curvas descrito na seção 4.2.3 sobre estas médias, obtendo uma equação logarítmica. Assim como nos casos do *broadcast* e do *reduce*, utilizamos a função piso no intuito de ajustar a predição conforme o número de processos envolvidos. Embora o comportamento da primitiva durante os testes tenha apresentado algumas oscilações, o modelo elaborado mostrou uma precisão razoável. A Tabela 5.8 exibe os tempos preditos, os medidos e os respectivos erros percentuais, que em todos os casos foram inferiores a 8%.

## 5.5 *All-to-all*

A modelagem das primitivas coletivas, descrita nas seções anteriores, possibilitou a conclusão do estudo das operações previstas em nossos objetivos iniciais. Em vista dos bons resultados alcançados, decidimos incluir em nossa pesquisa uma outra primitiva com um padrão de comunicação diferente, podendo apresentar novas complicações ao processo de modelagem. Desta forma, nesta se-

ção efetuamos uma extensão ao nosso trabalho detalhando algumas análises do MPI_Alltoall. Tratando-se de uma comunicação do tipo vários-para-vários, a quantidade de mensagens trocadas entre os processos é relativamente alta e outros fatores relacionados ao desempenho da rede de interconexão, como o número de colisões no *switch*, tendem a exercer maior influência no tempo de execução da primitiva.

### 5.5.1 Considerações sobre a implementação do *all-to-all*

Na operação de comunicação *all-to-all*, cada processo envia e recebe uma mensagem de cada um dos demais integrantes do grupo. O fato das mensagens trocadas serem todas distintas dificulta o uso em sua implementação de algoritmos baseados em árvores ou hipercubos, que são mais eficientes e apresentam complexidade logarítmica. No caso destes algoritmos, um melhor desempenho é obtido através da paralelização das trocas de mensagens: considerando o MPI_Bcast, por exemplo, o mestre pode enviar os dados a alguns escravos e encarregá-los de repassar aos demais simultaneamente.

Porém, como os dados transmitidos e recebidos durante a execução do *all-to-all* são diferentes para cada processo, a solução mais direta é realizar a comunicação utilizando primitivas ponto-a-ponto. O LAM-MPI implementa o MPI_Alltoall desta maneira, fazendo uso ainda de estruturas de repetição para enviar e receber as mensagens. Assim, ignorando a influência de outros fatores decorrentes do grande tráfego de dados criado na rede de interconexão e no *switch*, podemos dizer que o desempenho esperado do MPI_Alltoall é linear tanto no tamanho da mensagem como no número de processos envolvidos.

### 5.5.2 Testes realizados com o *all-to-all*

Os testes realizados com o MPI_Alltoall utilizaram tamanhos de mensagens ($n$) de 120000, 360000, 600000 e 1080000 inteiros e as seguintes quantidades de processos ($p$): 2, 4, 5, 8, 10, 12 e 16. A Tabela 5.9 expõe as médias dos tempos, calculadas com os valores inferiores a 1,8 vezes o menor tempo (conforme descrito na subseção 4.2.2), e o respectivo desvio padrão $S$ de cada configuração.

Tabela 5.9: Tempos medidos (segundos) e desvio padrão S do MPI_Alltoall.

| | | Tamanho da mensagem (inteiros) | | | | |
|---|---|---|---|---|---|---|
| | | 120000 | 360000 | 600000 | 840000 | 1080000 |
| 2 processos | tempo | 0,0362835 | 0,1029136 | 0,1738478 | 0,2440657 | 0,3162579 |
| | S | 0,0000973 | 0,0058279 | 0,0180261 | 0,0215186 | 0,0203007 |
| 4 processos | tempo | 0,0495185 | 0,1417801 | 0,234671 | 0,3292119 | 0,4225442 |
| | S | 0,0016014 | 0,0024166 | 0,0013210 | 0,0029709 | 0,0311542 |
| 5 processos | tempo | 0,0540149 | 0,1569911 | 0,2567173 | 0,3635781 | 0,4676668 |
| | S | 0,0035756 | 0,0293664 | 0,0811419 | 0,0455725 | 0,1175860 |
| 8 processos | tempo | 0,0542019 | 0,1756027 | 0,2910413 | 0,4662323 | 0,5457241 |
| | S | 0,0035309 | 0,0910173 | 0,0387500 | 0,0627439 | 0,0950959 |
| 10 processos | tempo | 0,0564610 | 0,2722008 | 0,4064324 | 0,4663500 | 0,5909906 |
| | S | 0,0008809 | 0,0609094 | 0,0439259 | 0,0274218 | 0,0661314 |
| 12 processos | tempo | 0,0561209 | 0,2367363 | 0,4199213 | 0,4738201 | 0,5742007 |
| | S | 0,0004053 | 0,1050488 | 0,0504195 | 0,0728449 | 0,0408647 |
| 16 processos | tempo | 0,0574607 | 0,3251201 | 0,4093525 | 0,4910278 | 0,5895586 |
| | S | 0,1087531 | 0,0717731 | 0,0985304 | 0,064100 | 0,0433598 |

Analisando os valores de *S*, podemos constatar que em alguns casos os tempos medidos apresentaram uma variação muito grande. Por exemplo, nas execuções com mensagens de tamanho 360000, verificamos um desvio superior a 40% da média dos valores para o caso de 12 processos e de 50% para o caso de 8. Nos testes com $p$=16 e $n$=120000 encontramos a maior variação entre valores medidos, por isso a Tabela 5.9 indica um desvio padrão de 0,1087531 para um tempo de apenas 0,0574607 segundos.

Em várias outras configurações podemos observar ainda desvios superiores a 10% de seus tempos médios. Provavelmente, estes números refletem influências do grande tráfego de dados na rede de interconexão e, principalmente, no *switch*. Enquanto os valores de *S* observados para $p$=2 e $p$=4 foram, em sua maioria, inferiores a 10% de seus tempos médios, nos testes envolvendo quantidades maiores de processos tal percentagem mostrou-se em geral elevada. Assim, os resultados indicam que o valor percentual do desvio padrão foi maior quando mais processos estavam envolvidos na comunicação.

Obviamente, quanto maiores os tamanhos de mensagens e o número de pro-

cessos, o congestionamento da rede e do *switch* tendem a aumentar. Embora o *switch* possa, teoricamente, prover a largura de banda integral da rede mesmo quando várias mensagens são transmitidas simultaneamente, é possível que ocorra congestionamento em uma de suas portas e, conseqüentemente, algum *overhead* extra se muitas mensagens forem endereçadas a ela.

### 5.5.3 Análise em função do tamanho da mensagem

Os gráficos expostos na Figura 5.6 ilustram o comportamento do MPI_Alltoall em função do tamanho da mensagem. Além do desempenho observado nos tempos medidos, são exibidas algumas curvas geradas com auxílio das equações lineares listadas abaixo:

- 2 processos: $t_{alltoall}(n) = -0,0006015 + (2,921 * 10^{-7}) * n$
- 4 processos: $t_{alltoall}(n) = 0,0021743 + (3,89 * 10^{-7}) * n$
- 5 processos: $t_{alltoall}(n) = 0,0013209 + (4,308 * 10^{-7}) * n$
- 8 processos: $t_{alltoall}(n) = -0,011858 + (5,307 * 10^{-7}) * n$
- 10 processos: $t_{alltoall}(n) = 0,0426849 + (5,263 * 10^{-7}) * n$
- 12 processos: $t_{alltoall}(n) = 0,033849 + (5,305 * 10^{-7}) * n$
- 16 processos: $t_{alltoall}(n) = 0,0669781 + (5,125 * 10^{-7}) * n$

Considerando as curvas de 2, 4 e 5 processos, a Figura 5.6 mostra comportamentos de acordo com o esperado (linear em *n*, conforme mencionamos no início da seção). Porém, nos demais valores de *p* observamos desempenhos variados e não compatíveis com as curvas obtidas a partir das equações lineares. No intervalo de *n* variando entre 120000 e 600000, as curvas de medição referentes a 8 e 12 processos demonstraram comportamento linear, ao passo que a curva de 10 processos mostrou-se bastante irregular. Analisando a configuração com o maior valor de *p*, 16, entre mensagens de 360000 a 1080000 inteiros podemos perceber um crescimento próximo ao linear.

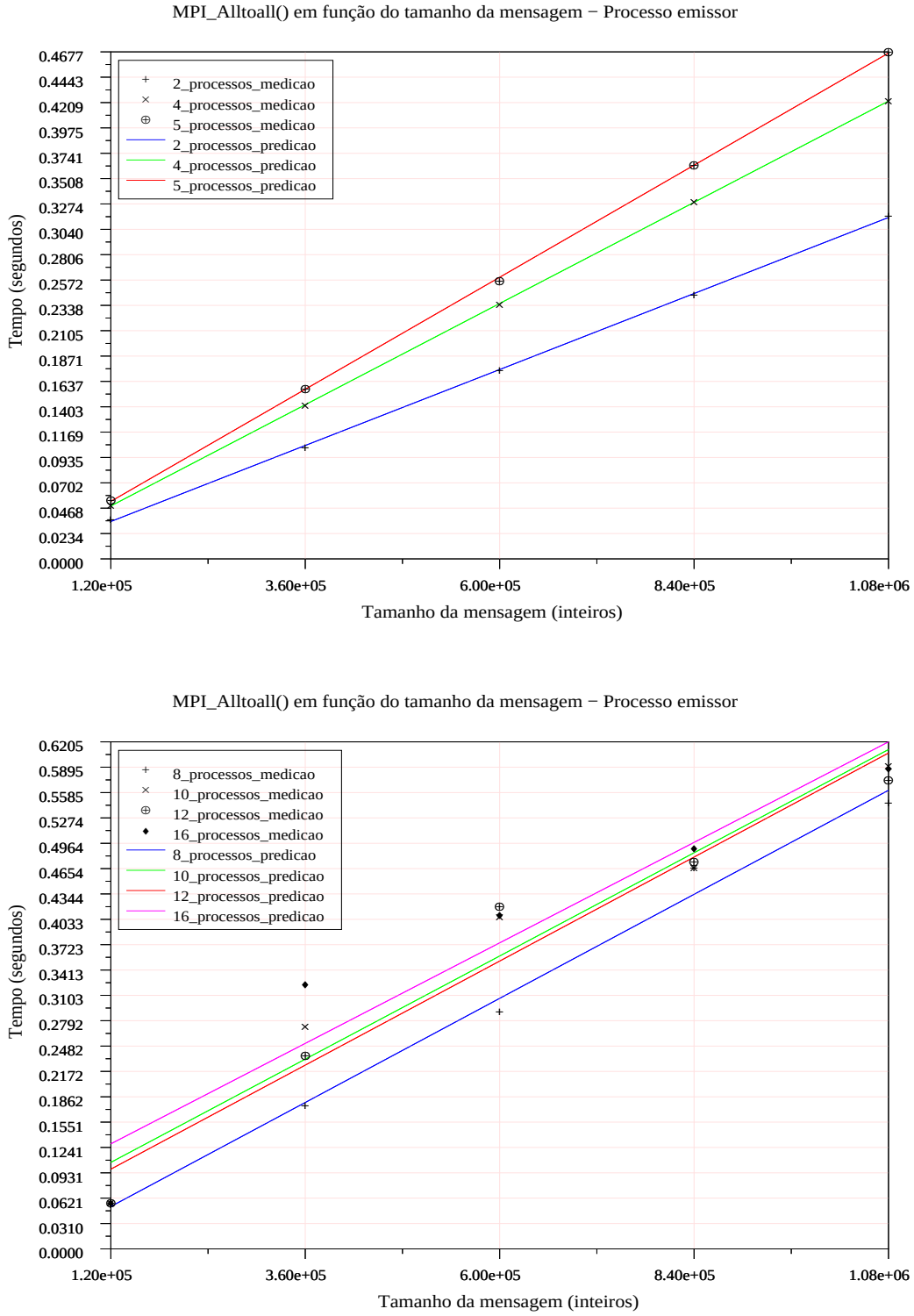


Figura 5.6: MPI_Alltoall em função do tamanho da mensagem.

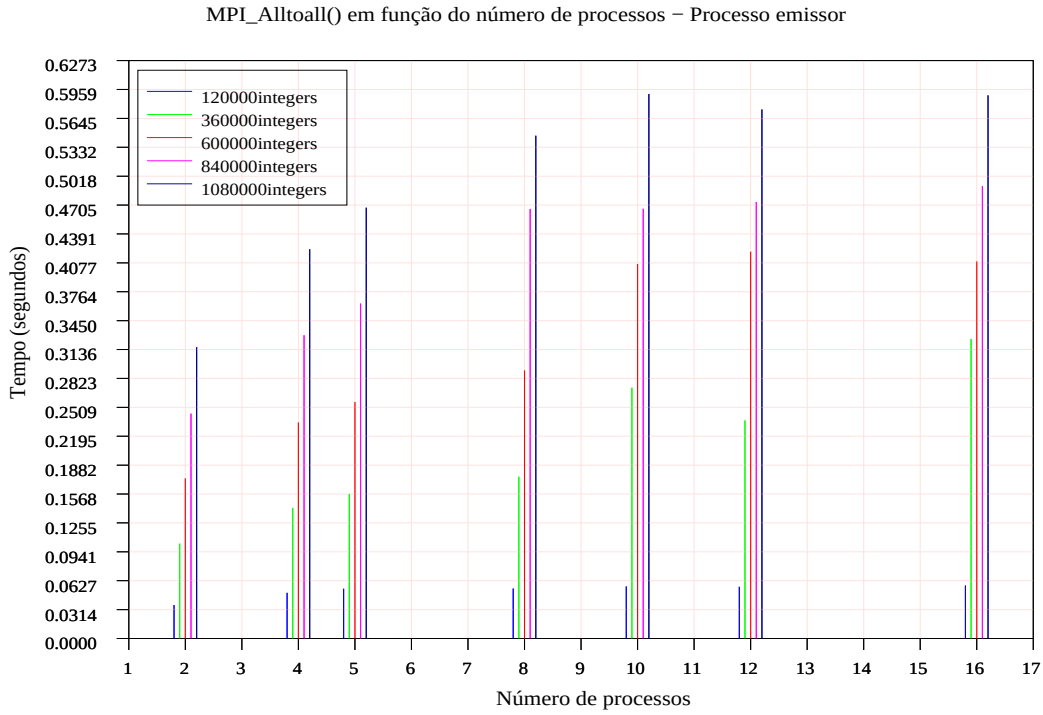


Figura 5.7: MPI_Alltoall em função do número de processos.

### 5.5.4 Análise em função do número de processos

Observando o gráfico do MPI_Alltoall apresentado na Figura 5.7, percebemos um comportamento linear em *p*, no intervalo de 2 a 5 processos, para todos os tamanhos de mensagens utilizados. De forma semelhante à análise em função de *n* apresentada na subseção anterior, nas configurações com $p \geq 8$ encontramos desempenhos variados. Aumentando o número de processos entre 8 e 16, como podemos verificar no gráfico, o comportamento da primitiva mostra-se irregular com mensagens de 360000 e 1080000 e constante com mensagens de 120000 elementos.

Desempenhos distintos também podem ser notados ao assumirmos para *n* os valores de 600000 e 840000. Ao passo que com a primeira quantidade de elementos o tempo permanece praticamente constante em 10, 12 e 16 processos, no segundo caso o período de constância é observado com *p* entre 8 e 12.

### 5.5.5 Influência de outros fatores

As oscilações observadas no comportamento do MPI_Alltoall dificultam o estudo e a modelagem do seu comportamento. Tratando-se de um trabalho com fins de predizer tempos de execução, precisamos elaborar modelos capazes de abstrair os principais fatores que determinam este tempo. Assim, faz-se necessário compreender os motivos das variações de comportamento da primitiva e identificar mecanismos para expressá-los por meio de modelos.

As análises apresentadas do *all-to-all* indicam desempenhos irregulares, principalmente quando um número maior de processos estava envolvido. Considerando o alto volume de mensagens enviadas e recebidas por cada processo nestes casos, é possível que ocorra uma sobrecarga em algumas das portas do *switch*. Cada uma destas portas é conectada a um certo nó de processamento do *cluster*, sendo utilizada tanto para envio como recepção de dados, e está sujeita a determinada largura de banda. Além disso, o número de colisões ocorridas e o padrão de comunicação implementado também podem provocar alterações no desempenho da primitiva.

Portanto, o comportamento do MPI_Alltoall pode ser influenciado por fatores relacionados ao grande volume de comunicações estabelecidas. Uma forma de estudar estas influências, talvez identificando novos parâmetros para modelar o tempo despendido na execução da primitiva, seria através do monitoramento das portas do *switch*. Analisando o comportamento de uma de suas portas em situações envolvendo um volume significativo de trocas de dados, acreditamos ser possível compreender melhor seu desempenho e, conseqüentemente, sua influência no tempo de comunicação.

## 5.6 Predições com grandes quantidades de processos

Em virtude do número limitado de máquinas disponíveis em nosso laboratório, não foi possível a realização de testes com maiores quantidades de processos ($p$). Contudo, nesta seção apresentamos um estudo sobre o provável comportamento das primitivas MPI_Bcast, MPI_Reduce e MPI_Barrier quando números grandes

são atribuídos a *p*. Fazendo uso dos modelos analíticos elaborados e apresentados nas seções anteriores, variamos *p* e obtivemos novos tempos de comunicação.

A Figura 5.8 mostra os gráficos construídos com os resultados obtidos. No caso do *broadcast* e do *reduce*, utilizamos mensagens de 1500000 e 2000000 inteiros e variamos o número de escravos de 4 a 256. As quantidades de processos assumidas nas predições com a operação de barreira também pertenciam ao intervalo de 4 a 256. Devido ao fato dos modelos terem sido elaborados a partir dos testes executados no ambiente computacional descrito na seção 4.4, as curvas indicam o possível comportamento das primitivas em um *cluster* específico.

## 5.7 Considerações finais

Através das análises e modelos analíticos apresentados neste capítulo, executamos uma segunda fase de nossos estudos e finalizamos os trabalhos desenvolvidos. Foram caracterizadas algumas primitivas de comunicação coletiva MPI, procurando esclarecer os principais fatores que influenciam os seus tempos de execução. Além das primitivas previstas em nossos objetivos, descrevemos um estudo com o MPI_Alltoall. Ao final do capítulo, alguns gráficos expostos ilustram o desempenho esperado do MPI_Bcast, do MPI_Reduce e do MPI_Barrier segundo os modelos elaborados, simulando a existência de grandes quantidades de processos.

Um último comentário, comparando os tempos obtidos com o MPI_Send (Tabela 4.1) e o MPI_Bcast nos testes com apenas um escravo (Tabela 5.3), ainda se faz necessário. Analisando os valores, podemos perceber que os tempos do *broadcast* ficaram abaixo daqueles observados com a operação *send*. Uma possível justificatica para esse fato está relacionada com a disposição das máquinas em nosso *cluster*: durante os testes experimentais, as máquinas envolvidas nas medições com as operações ponto-a-ponto não foram as mesmas utilizadas nas execuções do MPI_Bcast com um escravo. Embora os nós de processamento dispusessem da mesma configuração, estavam conectados a diferentes portas do *switch*. Contudo, maiores conclusões sobre estas variações observadas nos tempos requerem a realização de novos testes e de um estudo mais aprofundado.

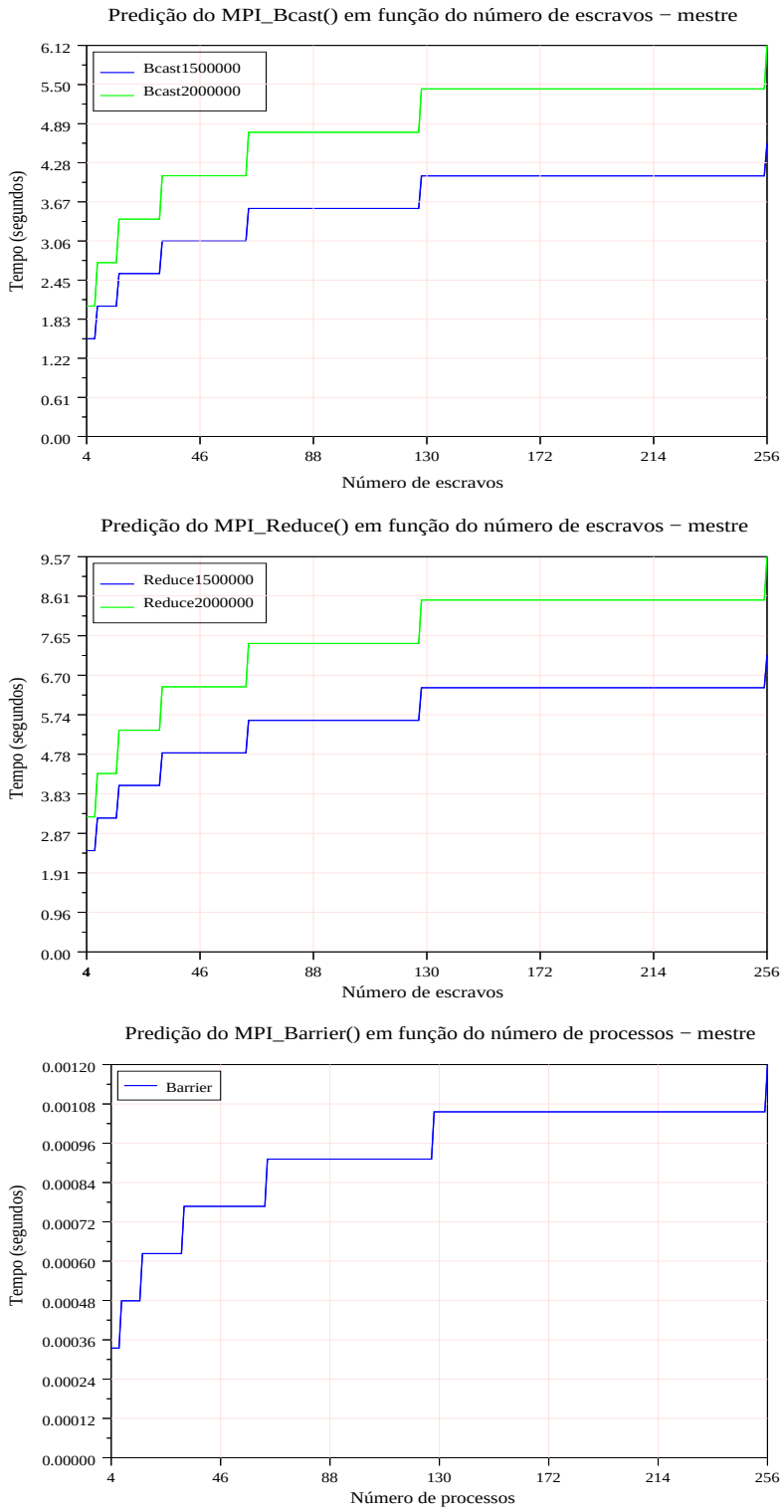


Figura 5.8: Predições em função de $p$ (*broadcast*, *reduce* e barreira).

# Capítulo 6

# CONCLUSÕES

O uso de *clusters* de estações de trabalho na solução de problemas com requisitos de alto desempenho e alta disponibilidade, principalmente nos últimos anos, tem demonstrado crescimento e impulsionado o desenvolvimento de tecnologias na área. Devido ao fato destas máquinas constituirem sistemas computacionais de memória distribuída, o potencial para o processamento paralelo e distribuído provisto por suas arquiteturas pode ser melhor explorado em aplicações baseadas em trocas de mensagens.

Neste intuito, o modelo de programação através de passagem de mensagens pode oferecer condições para a obtenção de melhores desempenhos. Fazendo uso de bibliotecas apropriadas, como o conjunto de funções disponibilizadas pelo padrão MPI (*Message Passing Interface*), aplicações são construídas nas quais a troca de informações entre os processos é realizada com primitivas de comunicação. E está justamente no tempo despendido na execução de tais primitivas o enfoque do trabalho apresentado.

Apoiados em medições de tempos experimentais e técnicas de ajuste de curvas, realizamos análises e elaboramos modelos analíticos de predição de desempenho de operações de comunicação bloqueante MPI. Desta forma, foram caracterizadas operações ponto-a-ponto e coletivas e o trabalho ainda apresentou um conjunto de funções desenvolvidas com o objetivo de oferecer suporte a atividades de análise e predição.

## 6.1 Realização dos objetivos

Conforme os objetivos propostos inicialmente, neste trabalho desenvolvemos modelos analíticos e análises de desempenho das seguintes primitivas de comunição MPI:

- MPI_Send (*send* padrão);
- MPI_Bsend (modo *buffered* do *send* padrão);
- MPI_Ssend (modo síncrono do *send* padrão);
- MPI_Rsend (modo *ready* do *send* padrão);
- MPI_Bcast (primitiva de comunicação coletiva *broadcast*);
- MPI_Barrier (operação de barreira);
- MPI_Reduce (operação coletiva de redução).

Através dos modelos gerados, foram realizadas predições de tempos de execução variando o tamanho das mensagens e, nas comunicações coletivas, também o número de processos envolvidos. Os resultados obtidos demonstraram a precisão dos modelos, sendo os maiores erros percentuais observados nos casos em que os tempos eram pequenos. Na maioria dos testes e predições executadas, encontramos diferenças entre valores preditos e medidos inferiores a 8%. Além das primitivas MPI listadas acima, apresentamos algumas análises da operação coletiva *all-to-all*.

Tratando-se de um estudo sobre o tempo despendido em comunicações, muitas vezes encontramos comportamentos estranhos nestes tempos, principalmente em face às oscilações no desempenho da rede de interconexão e do *switch*. Para caracterizar corretamente os tempos das primitivas MPI, precisamos adotar procedimentos apropriados e isolar as influências de fatores não-determinísticos. Através de alguns cuidados obedecidos durante o processo de medição e tratamento dos dados, procuramos obter valores capazes de expressar de forma coerente os tempos de comunicação.

Os testes foram executados com o LAM-MPI e, com o intuito de compreender melhor o comportamento das primitivas, estudamos o código referente à implementação das operações de comunicação analisadas. A influência dos algoritmos utilizados ficou fortemente caracterizada nas equações das operações coletivas. Por outro lado, em um estudo com primitivas ponto-a-ponto conseguimos melhorar a precisão dos modelos realizando a modelagem por intervalos e, desta forma, obtivemos equações distintas para diferentes tamanhos de mensagens.

O método utilizado nos ajustes de curva, denominado de mínimos quadrados, tende a reduzir os erros com valores maiores em detrimento dos menores valores. Uma maneira de aperfeiçoarmos a técnica, possibilitando reduzir os erros percentuais, seria usarmos uma matriz de pesos na obtenção dos coeficientes das equações e atribuirmos pesos distintos para diferentes pontos da curva de desempenho.

Dentre os objetivos iniciais do trabalho, pretendíamos ainda identificar técnicas e ferramentas capazes de oferecer suporte ao processo estabelecido pela metodologia de análise e predição adotada. Neste intuito, desenvolvemos um conjunto de funções para facilitar e automatizar algumas tarefas inerentes à aplicação da metodologia. Convém destacar as seguintes facilidades oferecidas pelas funções:

- seleção de tempos medidos e cálculo de médias e do desvio padrão;
- elaboração de modelos analíticos;
- geração de gráficos de desempenho e dados comparativos (medição, predição e erros percentuais).

## 6.2 Trabalhos futuros

Com o cumprimento dos objetivos propostos, podemos vislumbrar outros trabalhos a serem realizados, complementando e estendendo o estudo descrito. A seguir listamos algumas sugestões de temas a serem tratados em pesquisas futuras:

- Elaboração de uma biblioteca de funções Scilab para análise e predição de desempenho. Com o aprimoramento e extensão das funções já disponibili-

zadas, é possível o desenvolvimento de uma biblioteca de suporte a atividades de análise e predição com suporte do Scilab.

- Testes com outras implementações do padrão MPI. Uma extensão do trabalho apresentado seria realizada através de testes dos modelos teóricos em outras implementações MPI, como por exemplo o MPICH. Possivelmente, algumas variações de comportamento sejam constatadas e, neste caso, novos modelos devem ser elaborados. Porém, com o suporte das funções disponibilizadas e das análises apresentadas, o trabalho seria facilitado.

- Testes em *clusters* heterogêneos. Em nosso trabalho, não tratamos de outras influências no desempenho das comunicações que poderiam ser observadas em *clusters* heterogêneos. O uso de diferentes máquinas, dispondo de configurações de *hardware* e *software* distintas, talvez provoque alterações no comportamento das primitivas.

- Estudo aprofundado do *all-to-all*. Considerando as análises apresentadas do MPI_Alltoall, faz-se necessário um estudo mais aprofundado da influência do *switch* e de outros fatores no desempenho das comunicações. Uma alternativa para a realização deste estudo seria o monitoramento das portas do *switch*.

- Modelagem de outras primitivas de comunicação MPI. Operações como o *gather*, o *scatter* e o *sendreceive* não foram tratadas em nossos estudos e constituem possibilidades para pesquisas futuras.

- Análises de desempenho em outras redes e *clusters*. Nossos testes foram realizados em um *cluster* Linux utilizando uma rede de interconexão Fast-Ethernet, conforme descrito na seção 4.4. Com o uso de um *cluster* Windows e uma rede Gigabit, por exemplo, outros resultados poderiam ser gerados confirmando a validade das análises realizadas ou permitindo a obtenção de novas conclusões.

# Referências Bibliográficas


[Abandah96] ABANDAH, G. A.; DAVIDSON, E. S. Modeling the Communication Performance of the IBM SP2. In: INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM, 10., Honolulu, Hawaii, April 1996. **Proceedings**. p.246-257.

[Al-Tawil01] AL-TAWIL, K.; MORITZ, C. A. Performance Modeling and Evaluation of MPI. **Journal of Parallel and Distribuited Computing**, 61, p.202-223, 2001.

[Crovella94] CROVELLA, M. E. **Performance Prediction and Tuning of Parallel Programs**. August 1994. 105p. Phd thesis - The University of Rochester, Computer Science Department. Rochester, New York.

[Culler93] CULLER, D. et al. LogP: Towards a Realistic Model of Parallel Computation. In: SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (ACM SIGPLAN), 4., San Diego, CA, May 1993. **Proceedings**. p.1-12.

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

[Gropp] GROPP, W. et al. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. **Parallel Computing**, v.22, n.6, p.789-828, September 1996.

[Hennessy96] HENNESSY, J. L.; PATTERSON, D. **Computer Architecture: A Quantitative Approach**. Second Edition. San Francisco, California: Morgan Kaufmann, 1996. 760p.

[Hu97] HU, L.; GORTON, I. **Performance Evaluation for Parallel Systems: A Survey**. Sydney, Australia: University of NSW, School of Computer Science and Engeneering, Department of Computer Systems, October 1997. (UNSW-CSE-TR-9707).

[INRIA] INRIA (INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE), SCILAB HOME PAGE. France. Disponível em: <http://www-rocq.inria.fr/scilab>. Acesso em: 14/10/02.

[Jain91] JAIN, R. **The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling**. New York: John Wiley & Sons, 1991. 685p.

[Kant92] KANT, K. **Introduction to Computer System Performance Evaluation**. New York: McGraw-Hill, 1992. 612p.

[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, Julho de 2002. **Anais**. SBC, 2002. p.25-36.

[Li01] 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. São Paulo.

[Luecke99] LUECKE, G. R.; RAFFIN, B.; COYLE, J. J. The Performance of the MPI Colletive Communication Routines for Large Messages on the Cray T3E600, the Cray Origin 2000, and

the IBM SP. **The Journal of Performance Evaluation and Modeling for Computer Systems**, July 1999. Disponível em: <http://hpc-journals.ecs.soton.ac.uk/PEMCS/Papers/Paper10>. Acesso em: 10/10/02.

[Maui] MAUI HIGH PERFORMANCE COMPUTING CENTER, PARALLEL VIRTUAL MACHINE. Disponível em: <http://www.uni-karlsruhe.de/Uni/RZ/Hardware/SP2/Workshop.mhpcc/pvm/PvmIntro.html>. Acesso em: 25/07/2002.

[Meira95] MEIRA JR, W. **Modeling Performance of Parallel Programs**. Rochester, New York: The University of Rochester, Computer Science Departament, June 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, 1999. v.2. p.4-27.

[MPI Forum] MPI FORUM. **MPI: A Message-passing Interface Standard**. Knoxville, Tennessee: University of Tennessee, June 1995. (Technical Report, version 1.1).

[Mucci98] MUCCI, P. J.; LONDON, K.; THURMAN, J. **The MPBench Report**. Department of Computer Science, Univerity of Tenessee, November 1998. (Technical report).

[Mullender93] MULLENDER, S. (Ed). Distributed Systems. Second edition. New York: ACM Press, 1993. In: HU, L.; GORTON, I. **Performance Evaluation for Parallel Systems: A Survey**. Sidney, Australia: University of NSW, School of Computer Science and Engeneering, Department of Computer Systems, October 1997. (UNSW-CSE-TR-9707).

[Nupairoj94] NUPAIROJ, N; NI, L. Performance Evaluation of Some MPI Implementations on Workstation Clusters. In: SCALABLE PA-

RALLEL LIBRARIES CONFERENCE, 1994. **Proceedings**. IEEE Computer Society Press, October 1994. p.98-105.

[Oak] OAK RIDGE NATIONAL LABORATORY COMPUTER SCIENCE AND MATHEMATICS DIVISION, PVM: PARALLEL VIRTUAL MACHINE. Disponível em: <http://www.scm.ornl.gov/pvm>. Acesso em: 25/07/2002.

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

[Oliveira02] 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**.

[Pramanick99a] PRAMANICK, I. Parallel Programming: Languages and Environments. In: BUYYA, R. **High Performance Cluster Computing: Programming and Applications**. New Jersey: Prentice-Hall, 1999. v.2. p.28-47.

[Pramanick99b] PRAMANICK, I. MPI and PVM Programming. In: BUYYA, R. **High Performance Cluster Computing: Programming and Applications**. New Jersey: Prentice-Hall, 1999. v.2. p.48-86.

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

[Schopf98] SCHOPF, J. M. **Performance Prediction and Scheduling for Parallel Applications on Multi-User Clusters**. 1998. 247p. Phd thesis - University of California. San Diego.

[Ohio] THE OHIO STATE UNIVERSITY, MPI PRIMER: DEVELOPING WITH LAM. Columbus, Ohio. 1996. Disponível em:

<http://www.lam-mpi.org/download/files/lam61.nol.doc.pdf>. Acesso em: 10/07/2002.

[Tennessee] UNIVERSITY OF TENNESSEE, MPI: A MESSAGE-PASSING INTERFACE STANDARD. Knoxville, Tennessee. 1995. Disponível em: <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html #Node0>. Acesso em: 10/07/2002.

[Xu96] XU, Z.; HWANG, K. Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2. **IEEE Parallel & Distributed Technology**, v.4, n.1, p.9-23, 1996.

# Apêndice A

# FUNÇÕES IMPLEMENTADAS COM O SCILAB

Muitas atividades de análise e predição de desempenho apresentam um custo operacional bastante significativo. Considerando em particular a metodologia adotada em nosso trabalho, algumas tarefas como a seleção dos tempos de comunicação e o desenvolvimento de modelos analíticos, por exemplo, podem dificultar a aplicação da metodologia e até mesmo desestimular o seu emprego.

Procurando facilitar e automatizar algumas atividades implícitas no processo de análise e predição adotado, construímos o conjunto de funções apresentado a seguir. Tais funções foram elaboradas com o suporte do Scilab-2.6, utilizando algumas facilidades e funções pré-definidas pela ferramenta. Para que possam ser úteis em atividades de análise e predição de desempenho, sua execução somente pode ser realizada no ambiente oferecido pelo Scilab e, portanto, faz-se necessária a prévia instalação do mesmo.

**1. MPI_Bcast**

**timesbcast( ) :** gera o arquivo com os tempos médios do MPI_Bcast. Lê arquivos com tempos medidos e calcula os tempos médios aplicando uma política de seleção de dados.

**plotbcast( ) :** gera equações e gráficos "Tamanho da mensagem X Tempo" para

o MPI_Bcast, exportando os gráficos para um arquivo XFig[1].

**plotbcast2( ) :** gera o gráfico de barras verticais "Numero de escravos X Tempo" para o MPI_Bcast, exportando o gráfico para um arquivo XFig.

**coeffbcastnp( ) :** lê valores de arquivos de tempos médios, gera e escreve os coeficientes do seguinte modelo em um arquivo de saída: $t_{bcast}(n,p) = (a_0 + a_1 * n) + (a_2 + a_3 * n) * \lfloor log_2 p \rfloor$

**predictbcast( ) :** realiza predições de desempenho em função do tamanho da mensagem através do modelo $t_{bcast}(n) = (a_0 + a_1 * n)$ . Lê os coeficientes de um arquivo, faz a predição e gera um arquivo com dados comparativos (predição, medição e erro).

**predictbcastnp( ) :** predição de desempenho em função do tamanho da mensagem e do número de escravos, através do modelo $t_{bcast}(n,p) = (a_0 + a_1 * n) + (a_2 + a_3 * n) * \lfloor log_2 p \rfloor$ . Lê os coeficientes de um arquivo, faz a predição e gera um arquivo com dados comparativos (predição, medição e erro).

### 2. MPI_Reduce

**timesreduce( ) :** idem a **timesbcast( )**. Lê arquivos com tempos medidos e calcula os tempos médios do MPI_Reduce aplicando uma política de seleção de dados.

### 3. MPI_Barrier

**timesbarrier( ) :** idem a **timesbcast( )**. Lê arquivos com tempos medidos e calcula os tempos médios do MPI_Barrier aplicando uma política de seleção de dados.

**plotbarrier( ) :** gera o gráfico "Número de processos X tempo" com a curva de medição do MPI_Barrier. Lê o respectivo arquivo de tempos médios e exporta o gráfico gerado para um arquivo XFig.

[1] XFig: aplicativo Linux para construção de gráficos e diagramas.

**coeffbarrier( ) :** gera os coeficientes da equação $t_{barrier}(p) = a_0 + a_1 * \lfloor log_2 p \rfloor$ . Lê tempos do arquivo de médias do MPI_Barrier, calcula e escreve $a_0$ e $a_1$ em um arquivo de saída.

**predictbarrier( ) :** predição em função do número de processos, através do modelo $t_{barrier}(p) = a_0 + a_1 * \lfloor log_2 p \rfloor$ . Lê os coeficientes do MPI_Barrier de seu respectivo arquivo, faz a predição e gera um arquivo com dados comparativos (predição, medição e erro).

### 4. MPI_Alltoall

**timesalltoall( ) :** idem a **timesbcast( )**. Lê arquivos com tempos medidos e calcula os tempos médios do MPI_Alltoall aplicando uma política de seleção de dados.

### 5. Outras funções

**lscf(x,y,g) :** calcula os coeficientes do polinômio de ordem *g*, aplicando o método dos mínimos quadrados (*least squares curve fitting*) sobre o conjunto de pares ordenados dado pelos vetores *x* e *y*. Esta função é utilizada por todas as demais que necessitam realizar um ajuste de curvas.

**predcurve( ) :** gera o gráfico "Número de processos X Tempo" somente com curvas de predição para primitivas coletivas (MPI_Bcast, MPI_Reduce e MPI_Barrier).

**plotcollective( ) :** generalização do **plotbcast( )** para o MPI_Bcast, o MPI_Reduce e o MPI_Alltoall.

**plotcollective2( ) :** generalização do **plotbcast2( )** para as primitivas coletivas MPI_Bcast, MPI_Reduce e MPI_Alltoall.

**coeffcollectivenp( ) :** generalização do **coeffbcastnp( )** para as primitivas coletivas MPI_Bcast e MPI_Reduce.

**predictnp( ) :** generalização do **predictbcastnp( )** para as primitivas coletivas MPI_Bcast e MPI_Reduce.

**plotsends( ) :** desenvolvido apenas para gerar gráficos com curvas de medição e predição dos quatro modos de *send* bloqueantes - MPI_Send, MPI_Bsend, MPI_Rsend e MPI_Ssend.

**stdev( ) :** lê os tempos medidos de um arquivo e calcula o desvio padrão.