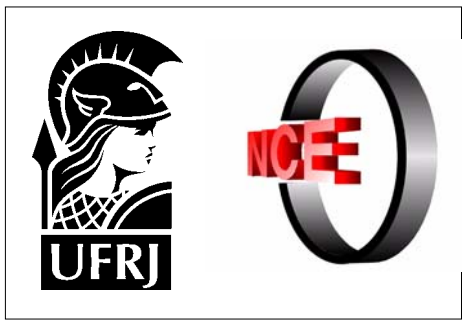

PMMPI: Uma Implementação do MPI para Estações de Trabalho com Suporte a Multiprocessamento Simétrico

SIDNEY BATISTA FILHO

INSTITUTO DE MATEMÁTICA
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Mestrado em Informática

Orientador: Oswaldo Vernet de Souza Pires, D.Sc.

Rio de Janeiro - Brasil

2002

PMMPI: Uma Implementação do MPI para Estações de Trabalho com Suporte a Multiprocessamento Simétrico

SIDNEY BATISTA FILHO

INSTITUTO DE MATEMÁTICA
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Mestrado em Informática

Orientador: Oswaldo Vernet de Souza Pires, D.Sc.

Rio de Janeiro - Brasil

2002

PMMPI: Uma Implementação do MPI para Estações de Trabalho com Suporte a Multiprocessamento Simétrico

Autor: Sidney Batista Filho

Dissertação submetida ao corpo docente do Instituto de Matemática / Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro – UFRJ, como parte dos requisitos necessários à obtenção do grau de Mestre em Informática.

Aprovada por:

Prof. _____ - Orientador

(Oswaldo Vernet de Souza Pires – D.Sc.)

Prof. _____

(Adriano Joaquim de Oliveira Cruz – Ph.D.)

Prof. _____

(Carlos Augusto Paiva da Silva Martins – D.Sc.)

Rio de Janeiro - Brasil

2002

Filho, Sidney Batista.

PMMPI: Uma Implementação do MPI para Estações de Trabalho com Suporte a Multiprocessamento Simétrico / Sidney Batista Filho. Rio de Janeiro, 2002.

xv, 105 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro – UFRJ, Instituto de Matemática / Núcleo de Computação Eletrônica – IM/NCE, 2002.

Orientador: Oswaldo Vernet de Souza Pires, D.Sc.

1. Técnicas de programação: programação paralela. 2. Arquitetura de computadores: multiprocessamento simétrico. 3. Sistemas operacionais: gerenciamento de processos/*threads* (*multithreading*). 4. Ambientes de programação paralela: modelos de memória compartilhada/passagem de mensagem. 5. Estruturas de dados: listas e filas.

DEDICATÓRIA

Este trabalho é uma homenagem póstuma ao Professor Júlio Salek Aude, que foi o idealizador do projeto Multiplus, o qual continua sendo desenvolvido no NCE/UFRJ.

AGRADECIMENTO

Gostaria de agradecer aos professores Oswaldo Vernet, Adriano Cruz, Carlos Martins, Eliana Aude, Cláudio Santos, Mário Júnior, Ageu Pacheco, Gabriel Silva, Fabrício Silva, Lucila e Oswaldo Carvalho, pelo incentivo, pelos preciosos comentários e pela ajuda.

Também sou muito grato a todos os meus amigos e, em especial, ao Dario, Iuri, César, Roberta-e-Magnos, Leonardo-e-Cláudia, Marluce, Cláudio Libanio, Leila, Denise Candal, Raquel, Midori, Ana Paula Paes, Rodrigo Lemos e Carlos França, pelo companheirismo e incentivo.

Agradeço a todos os meus familiares e, em especial, Tia Cida, Tio Carlinhos, Tia Jô, Tio Pedrinho, Ísis, Stela, Tio Balto, Adir, Luciana, Lineu, Leandro, Regininha, Raum, Janaina e, principalmente, meus pais por todo carinho e amor.

Finalmente, gostaria de fazer uma outra homenagem póstuma ao meu avô, Onofre Matos.

AGRADECIMENTO

Gostaria de agradecer ao FINEP/CNPq pelo fornecimento da bolsa de estudos e ao NCE/UFRJ, pelos recursos oferecidos como laboratórios, salas de aula e biblioteca e, também, pela bolsa de estudos fornecida.

Agradeço aos colaboradores do NCE/UFRJ pelos excelentes serviços prestados.

RESUMO

Filho, Sidney Batista. **PMMPI**: Uma Implementação do MPI para Estações de Trabalho com Suporte a Multiprocessamento Simétrico. Rio de Janeiro: UFRJ/IM/NCE, 2002. Dissertação (Mestrado em Informática).

O MPI é uma especificação de uma biblioteca para programação paralela usando passagem de mensagem. As implementações convencionais deste padrão mapeiam cada tarefa MPI em um processo do sistema operacional. Para a realização desta dissertação de mestrado, foi projetada e desenvolvida, desde o início, a biblioteca PMMPI (*Portable Multithreaded Message Passing Interface*), que é uma implementação do MPI para estações de trabalho com suporte a multiprocessamento simétrico onde cada tarefa MPI é mapeada em uma *thread* de um mesmo processo. Deste modo pretende-se beneficiar das vantagens inerentes às aplicações *multithreaded*. O PMMPI foi construído sobre uma camada abstrata de primitivas para gerenciamento/sincronização de *threads*, o que evita que sejam feitas chamadas diretamente ao sistema operacional. Para avaliar esta implementação foram feitos testes de desempenho comparativos entre o PMMPI e uma implementação do MPI baseada em processos feita pela Sun Microsystems, Inc.

ABSTRACT

Filho, Sidney Batista. **PMMPI**: Uma Implementação do MPI para Estações de Trabalho com Suporte a Multiprocessamento Simétrico. Rio de Janeiro: UFRJ/IM/NCE, 2002. Dissertação (Mestrado em Informática).

MPI is a specification of a library for parallel programming using message-passing. The traditional implementations usually map each task onto a process of the operating system. In this dissertation, we designed and developed from scratch the library named PMMPI (Portable Multithreaded Message Passing Interface), that is an implementation of MPI for workstations with symmetric multiprocessing suport, in which the tasks of an application are mapped onto sibling threads under a single process, benefiting from the inherent advantages of multithreaded applications. PMMPI was built on an abstract layer of thread-management/synchronization primitives, avoiding to make operating system calls directly. To evaluate this implementation, we made tests to compare the performance of PMMPI and another implementation based on process developed by Sun Microsystems, Inc.

LISTA DE SIGLAS

SMP	- Symmetric Multiprocessor
MPI	- Message Passing Interface
MIMD	- Multiple Instruction, Multiple Data
PVM	- Parallel Virtual Machine
PMMPI	- Portable Multithreaded Message Passing Interface
TMPI	- Threaded-based MPI

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE GRÁFICOS

1 Introdução

1.1 Motivação

Existem várias aplicações computacionais que podem ser divididas em sub-tarefas independentes, que podem ser executadas simultaneamente em máquinas paralelas (AUDE, 1998, p.73). A programação paralela permite que sejam utilizados os diversos módulos de processamentos de máquinas paralelas para se executar tais aplicações.

Estas aplicações, quando executadas em máquinas paralelas, podem ter o seu tempo de execução muito reduzido se comparado ao tempo de execução das mesmas em uma máquina com um único módulo de processamento, por mais rápido que seja o processamento deste módulo. Segundo Messina (1998, p.37), "Para apreciar o avanço no poder da computação científica recentemente, considere que o desempenho dos computadores mais rápidos usados na ciência e engenharia foram aumentados por um fator de 500 durante a década de 90, enquanto o desempenho de um único processador aumentou em um fator de apenas 15".

As máquinas paralelas podem ser divididas em dois grupos: máquinas com memória primária fisicamente compartilhada, conhecidas como multiprocessadores e máquinas com memória distribuída, conhecidas como multicomputadores. Por outro lado, existem dois modelos de programação, que determinam se a memória é logicamente compartilhada ou não. No primeiro modelo, a comunicação entre as tarefas pode ser feita utilizando-se variáveis compartilhadas entre as mesmas. No segundo modelo de programação, a comunicação é feita através de troca de mensagens. Estas duas classificações – memória fisicamente compartilhada ou não e memória logicamente compartilhada ou não – são ortogonais. (ANDREW S. TANENBAUM, 1999, p.526)

Existem diversos ambientes de programação paralela que visam auxiliar o desenvolvimento de aplicações que exploram os recursos de máquinas paralelas. Cada ambiente procura explorar da melhor forma os recursos de determinados tipos de máquinas. A biblioteca PVM (GEIST, 1994), por exemplo, permite que vários computadores interligados por uma rede local sejam vistos pelo programador como um

computador paralelo virtual, com memória distribuída, onde a comunicação é feita através de troca de mensagem. Outro exemplo é Linda (CARRIERO, 1989), que permite ao programador ter uma visão de uma máquina com memória logicamente compartilhada, enquanto, na realidade, a máquina possui a memória fisicamente distribuída. E, como último exemplo, existe a biblioteca Pthreads (NICHOLS, 1996), onde o programador tem uma máquina com memória lógica e fisicamente compartilhada.

Dentre os diversos ambientes de programação paralela, destaca-se o MPI (SNIR, 1996; MPI FORUM, 1995), por ter se tornado um padrão *de facto* para qualquer tipo de máquina paralela. O MPI (*Message Passing Interface*) é uma especificação de uma biblioteca para programação paralela usando troca de mensagens. Esta especificação permite que sejam desenvolvidas implementações do MPI para qualquer tipo de máquina paralela e, ao mesmo tempo, que estas implementações possam aproveitar da melhor forma os recursos de cada tipo de máquina.

1.2 Objetivos deste trabalho

No ambiente MPI, uma aplicação é dividida em uma coleção de tarefas idênticas, entretanto cada uma delas executa um fluxo de instruções independente das demais. As implementações tradicionais – como MPICH (LUSK, 1996) e LAM-MPI (LUMSDAINE, 2002) – costumam mapear cada tarefa em um processo do sistema operacional, utilizando primitivas de troca de mensagens para implementar as facilidades de comunicação do MPI.

O objetivo principal desta dissertação é descrever uma implementação de um subconjunto de primitivas do padrão MPI, na qual as tarefas de uma aplicação são mapeadas em *threads* irmãs, ou seja *threads* criadas por um mesmo processo, beneficiando-se das vantagens inerentes às aplicações *multithreaded*, a saber: o compartilhamento espontâneo da área de dados e a maior rapidez na criação e troca de contexto entre tarefas em máquinas multiprocessadas.

Para tanto, foi projetada e desenvolvida – desde o início – a biblioteca PMMPI (*Portable Multithreaded Message Passing Interface*), escrita na linguagem de programação C ANSI (KERNIGHAN, 1988). O PMMPI não faz chamadas diretamente ao sistema operacional, pois são utilizadas as primitivas de programação paralela do Mulplix (AZEVEDO, 1993), que é um sistema operacional desenvolvido para o multiprocessador Multiplus (AUDE, 1997). Conseqüentemente, o PMMPI é independente tanto em relação ao sistema operacional, quanto ao modelo de *threads* usado, pois para portar o PMMPI para outro ambiente, basta implementar estas poucas primitivas de programação do Mulplix para o sistema operacional / modelo de *threads* desejado.

1.3 Estrutura do trabalho

Este trabalho está estruturado como se segue. O capítulo 2 descreve a biblioteca MPI destacando seus principais objetivos, descrevendo os seus conceitos básicos – utilizando um pequeno exemplo para ilustrá-los – e descrevendo as principais rotinas de comunicação ponto-a-ponto e coletivas do MPI. A seguir, no capítulo 3, é apresentado o projeto Multiplus, em desenvolvimento no NCE/UFRJ, bem como o sistema operacional Mulplix, destacando as principais características das suas primitivas de programação paralela. A biblioteca PMMPI é descrita no capítulo 4, onde são apresentadas justificativas para se implementar o padrão MPI em máquinas de memória compartilhada, bem como para se mapear as tarefas MPI em *threads* ao invés de mapeá-las em processos. Neste capítulo, também são apresentadas as estruturas de dados usadas pelo PMMPI e as rotinas implementadas. No capítulo 5, é especificado o ambiente usado para o desenvolvimento do PMMPI e realização dos testes, bem como são relacionados os resultados comparativos entre o PMMPI e a biblioteca Sun MPI 4.0 da Sun Microsystems (SUN, 1999). Finalmente, as conclusões desta dissertação de mestrado, bem como as propostas de trabalhos futuros integram o capítulo 6.

2 O Ambiente MPI

Este capítulo apresenta a especificação da interface padrão para troca de mensagens – o MPI –, enumerando seus principais objetivos e comentando sobre suas versões. Em seguida, são descritos os conceitos básicos do MPI, bem como um pequeno exemplo para que se tenha uma familiarização rápida com o ambiente MPI. Finalmente, são descritas as suas principais operações de comunicação ponto-a-ponto e coletiva.

2.1 Apresentação do MPI

MPI (*Message Passing Interface*) é uma biblioteca de funções padrão e portátil, que define a sintaxe e a semântica de um núcleo de rotinas útil para se escrever programas que exploram a existência de múltiplos processadores com troca de mensagem (PETER PACHECO, 1998, p.3).

Segundo (SNIR, 1996, p.xi): "O padrão MPI é o resultado de um esforço que envolveu mais de 80 pessoas de 40 organizações, principalmente dos Estados Unidos e Europa. A maioria dos maiores vendedores de computadores concorrentes da época estava envolvida com o MPI, bem como pesquisadores de universidades, de laboratórios do governo e da indústria".

Este grupo, que definiu o MPI, passou a se chamar MPI Forum (MPI-FORUM) e é responsável pela manutenção deste padrão.

2.1.1 Objetivos do MPI

Os principais objetivos do MPI são:

- Permitir que sejam desenvolvidas implementações da especificação MPI em máquinas com arquiteturas diferentes. A partir do mesmo código fonte, que utiliza o MPI, pode-se gerar um programa que pode ser executado em multicomputadores com memória distribuída, em multiprocessadores com memória compartilhada, em *clusters* de estações de trabalho – COWs (*Clusters of Workstations*) – e em uma

combinação destes tipos de máquinas – ambientes heterogêneos – desde que haja uma implementação da biblioteca MPI para a plataforma específica.

- Permitir uma implementação eficiente de comunicação:
  - ♦ evitando, quando possível, cópia de memória para memória,
  - ♦ permitindo sobreposição de comunicação e computação.
- Prover uma semântica de interface que seja independente de linguagem de programação. Existem implementações de MPI que podem ser usadas por programas escritos nas linguagens C, C++ e Fortran.
- Prover uma interface de comunicação confiável. Falhas de comunicação devem ser tratadas pelo subsistema de comunicação da plataforma.
- Definir uma interface familiar para os usuários dos sistemas de troca de mensagens mais comuns como por exemplo: PVM (GEIST, 1994) e P4 (BUTLER, 1992).
- Projetar uma interface que permita *thread-safety*, ou seja, que permita que as funções possam ser invocadas de forma segura por múltiplas *threads* concorrentemente (ROBBINS, 1996, p.23).

2.1.2 Versões do MPI

A primeira versão do MPI, chamada MPI-1.0, foi lançada em 1994 pelo MPI Forum. Em 1995, foi disponibilizada outra versão, a MPI-1.1, sobre a qual este trabalho se baseia (MPI FORUM, 1995; GROPP, 1996).

Posteriormente, foi lançada a versão MPI-2, que consiste de extensões à versão anterior. Nestas extensões estão incluídas operações não existentes nas anteriores como, por exemplo: operações para criação e gerenciamento de processos, entrada e saída paralela e operações para acesso remoto à memória (GROPP, 1999, p.4).

2.2 Conceitos Básicos em MPI

2.2.1 Tarefas e *Communicators*

Um programa MPI consiste de várias tarefas autônomas, que executam seu próprio fluxo de instruções, podendo ser classificado, portanto, como sendo um programa

MIMD (*multiple instruction multiple data*), segundo a taxonomia de Flynn (FLYNN, 1972). As tarefas se comunicam através de chamadas às primitivas de comunicação do MPI.

Todas as rotinas do MPI relacionadas à comunicação entre tarefas possuem como parâmetro um *communicator*, que é uma estrutura de dados que representa um conjunto ordenado de tarefas que podem trocar mensagens entre si. Portanto, esta estrutura de dados representa o domínio da comunicação.

Toda tarefa MPI pertence a pelo menos um *communicator* e possui, para cada *communicator*, um *rank*, que é a sua identificação unívoca no mesmo. Os valores dos *ranks* são números inteiros consecutivos que variam de 0 até o número de tarefas do *communicator* menos 1.

Todo par (*communicator*, *rank*) identifica apenas uma tarefa em um programa MPI.

2.2.2 Comunicação por Passagem de Mensagem

Existem dois tipos de comunicação por passagem de mensagem: a comunicação ponto-a-ponto, que consiste na transmissão de dados entre um par de tarefas, uma delas enviando e a outra recebendo e a coletiva, que é usada para se transmitir dados entre todas as tarefas de um mesmo grupo especificado por um *communicator*.

A figura que se segue ilustra, de forma simplificada, um exemplo de comunicação ponto-a-ponto em duas etapas. Na primeira etapa, a tarefa emissora coloca a mensagem no seu *buffer* de envio e a tarefa receptora possui um *buffer* de recepção cujo conteúdo é inválido neste momento. Na segunda etapa, a mensagem é copiada para o *buffer* de recepção. Esta cópia pode ser feita de duas formas: diretamente do *buffer* de envio para o *buffer* de recepção (representado pela seta contínua), ou indiretamente, sendo copiada para um ou mais *buffers* temporários antes de ser copiada para o *buffer* de recepção (representado pelas setas pontilhadas).

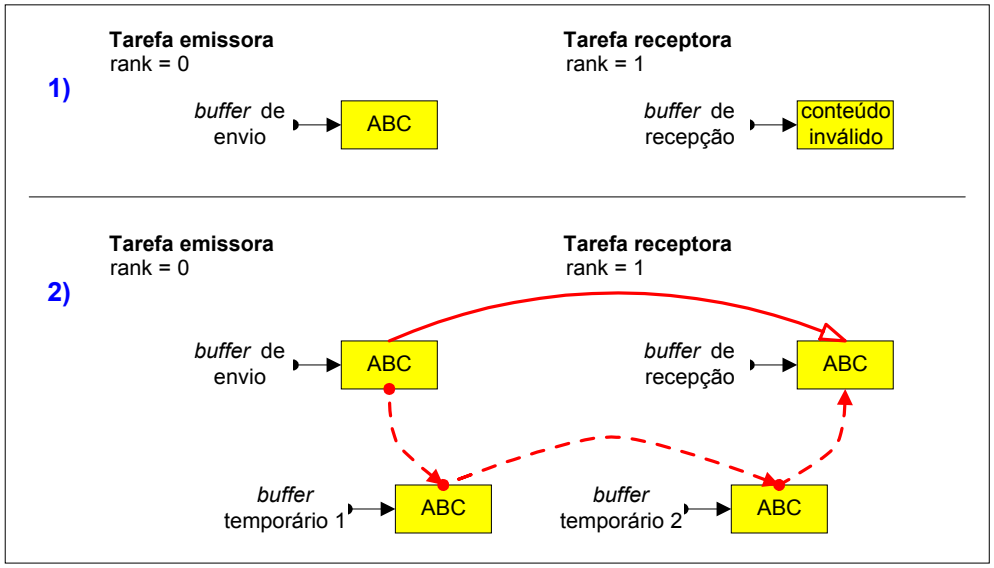

Figura 1: Comunicação por passagem de mensagem

2.2.3 Tipos de chamadas MPI

Os seguintes termos são usados para se discutir as rotinas do MPI:

- local: se a conclusão de uma rotina depender somente da tarefa local que está em execução. Tal operação não requer uma comunicação explícita com outra tarefa do usuário.
- não-local: se a conclusão da rotina necessitar da execução de alguma rotina MPI em outra tarefa.
- bloqueante: se a rotina não retorna enquanto não for permitido ao usuário reutilizar os recursos especificados na chamada da mesma. Por exemplo, uma rotina de envio bloqueante não retorna até que o *buffer* de envio tenha sido copiado para outro lugar, de modo que o emissor fique livre para alterá-lo.
- não-bloqueante: se a rotina puder retornar antes que a operação iniciada pela mesma conclua. Neste caso, não será permitido ao usuário reutilizar os recursos (como *buffers*) especificados na chamada. Uma chamada a uma rotina não-bloqueante pode iniciar alterações no estado da tarefa chamadora que ocorrem depois que a chamada retorna. Por exemplo, uma chamada não-bloqueante pode iniciar uma operação de recepção, mas a mensagem é realmente recebida após o retorno da rotina.
- coletiva: se todas as tarefas em um grupo de tarefas precisarem invocar a rotina.

2.2.4 Primeiro exemplo

Como primeiro exemplo, é apresentado um programa simples que usa a biblioteca MPI. O objetivo deste programa é fazer com que cada tarefa cujo *rank* é diferente de 0 envie uma mensagem para a tarefa cujo *rank* é 0. A tarefa 0, por sua vez, exibe as mensagens recebidas. Todas as tarefas pertencem a um único *communicator*: MPI_COMM_WORLD.

Quadro 1: Todas as tarefas enviam mensagem para a tarefa 0

```
#include <stdio.h>
#include <mpi.h>

#define MAXLEN 100

main(int argc, char** argv) {
      int myRank;             /* rank da tarefa */
      int n;                  /* numero de tarefas */
      int source;             /* rank da origem */
      int dest;               /* rank do destino */
      int tag = 1;            /* tag para as mensagens */
      char message[MAXLEN];   /* buffer para a mensagem */
      MPI_Status status;      /* status de retorno para o receptor */

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
      MPI_Comm_size(MPI_COMM_WORLD, &n);

      if(myRank != 0) {
            dest = 0;
            sprintf(message, "Alo! Da tarefa %d!", myRank);
            MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
                  tag, MPI_COMM_WORLD);
      } else {
            for(source = 1; source < n; source++) {
                  MPI_Recv(message, MAXLEN, MPI_CHAR, source, tag,
                        MPI_COMM_WORLD, &status);
                  printf("%s\n", message);
            }
      }
      MPI_Finalize();
}
```

2.2.4.1 Inicialização e execução

A inicialização de programas MPI está fora do escopo da especificação MPI (MPI-FORUM, 1995), que diz como escrever programas MPI, não como executá-los. Portanto, embora os programas MPI sejam portáteis, os *scripts* que os invocam não o são. Entretanto, alguns passos comuns devem ocorrer em qualquer sistema, desde que seja executada uma tarefa em cada processador. São eles:

i) o usuário executa um *script* que coloca uma cópia do programa executável em cada processador.
ii) cada processador inicia a execução da sua cópia do executável.
iii) diferentes tarefas podem executar diferentes sentenças usando os desvios do programa. Tipicamente, os desvios são feitos baseando-se nos *ranks* das tarefas.

Um programa MPI usa o paradigma SPMD (*single program multiple data*). O efeito obtido é o seguinte: o mesmo código é executado por diferentes programas em diferentes processadores, porém usa-se o *rank* da tarefa para se tomar desvios diferentes. No exemplo mostrado no Quadro 1, as sentenças executadas pela tarefa 0 são diferentes das executadas pelas outras tarefas, apesar de todas as tarefas executarem o mesmo programa. Este é o método mais usado para se escrever programas MIMD (PACHECO, 1998, p.6).

Quando o programa for executado especificando-se 2 tarefas para o ambiente de inicialização, a saída será:

```
Alo! Da tarefa 1!
```

Se, por outro lado, forem especificadas 4 tarefas, a saída será:

```
Alo! Da tarefa 1!
Alo! Da tarefa 2!
Alo! Da tarefa 3!
```

2.2.4.2 Características básicas

Os programas MPI devem incluir o arquivo de cabeçalho `mpi.h`, que contém definições das estruturas de dados, macros e protótipos de funções necessários para se compilar os programas MPI.

```
#include <mpi.h>
```

Todo programa MPI deve chamar a rotina `MPI_Init` antes de qualquer outra rotina do MPI e apenas uma vez. Esta rotina inicializa o ambiente de execução do programa MPI. Seus argumentos são os endereços dos argumentos da função `main`. Por outro lado, a rotina `MPI_Finalize` deve ser chamada para se finalizar o ambiente de execução do MPI. Nenhuma outra rotina MPI pode ser chamada depois desta e o usuário é obrigado a garantir que toda a comunicação pendente envolvendo esta tarefa tenha se completado antes de chamar esta rotina de finalização. Portanto, todo programa MPI possui o seguinte *layout*:

Quadro 2: Inicialização e finalização de um programa MPI

```
...
#include <mpi.h>
...
main(int argc, char** argv) {
      ...
      /* nenhuma rotina MPI pode ser chamada antes desta */
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
      MPI_Comm_size(MPI_COMM_WORLD, &n);
      ...
      MPI_Finalize();
      /* nenhuma rotina MPI pode ser chamada após esta */
}
```

As rotinas `MPI_Comm_rank` e `MPI_Comm_size` são chamadas para se obter informações sobre o domínio de comunicação. Ambas recebem como primeiro argumento o *communicator* sobre o qual se deseja obter informações. No fragmento de código do Quadro 2, é passado como parâmetro `MPI_COMM_WORLD`, que é pré-definido e que representa o conjunto de todas as tarefas do programa MPI que está sendo executado.

A rotina `MPI_Comm_rank` retorna, através do seu segundo argumento, o *rank* da tarefa que a chamou, no *communicator* passado como primeiro argumento.

A rotina `MPI_Comm_size` retorna, através do seu segundo argumento, o número de tarefas existentes no *communicator* passado como primeiro argumento.

2.2.4.3 Mensagem

As tarefas MPI precisam trocar mensagens entre si para se comunicarem. No fragmento do Quadro 3, são destacadas as rotinas básicas do MPI para troca de mensagem.

Quadro 3: Envio e recepção de mensagem

```
...
main(int argc, char** argv) {
      ...
      int tag = 1;            /* tag para as mensagens */
      ...
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
      MPI_Comm_size(MPI_COMM_WORLD, &n);

      if(myRank != 0) {
            dest = 0;
            sprintf(message, "Alo! Da tarefa %d!", myRank);
            MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
                  tag, MPI_COMM_WORLD);
      } else {
            for(source = 1; source < n; source++) {
                  MPI_Recv(message, MAXLEN, MPI_CHAR, source, tag,
                        MPI_COMM_WORLD, &status);
                  printf("%s\n", message);
            }
      }
      MPI_Finalize();
}
```

A rotina `MPI_Send` envia uma mensagem para uma tarefa destino, enquanto `MPI_Recv` recebe uma mensagem da tarefa origem. Os parâmetros destas rotinas são:

- o endereço da região de memória (*buffer*) que contém o dado a ser enviado ou recebido;
- um número inteiro que indica quantos elementos serão enviados ou quantos elementos podem ser recebidos. Deve-se notar que, neste exemplo, o número de elementos a ser enviado é o tamanho da mensagem mais 1, para se incluir o caracter final de cadeia (\0);
- o tipo de dado de cada elemento do *buffer*;
- o *rank* do destino ou da origem da mensagem;

- um inteiro, definido pelo usuário da biblioteca MPI, que é usado para auxiliar a identificação de uma mensagem;
- o *communicator* ao qual as tarefas emissora e receptora pertencem;
- Na rotina `MPI_Recv` existe ainda um sétimo parâmetro, que é usado para se obter informações sobre a mensagem recebida.

Maiores detalhes destas rotinas são apresentados no item 2.3.2.Operações bloqueantes.

Quase todas as rotinas do MPI retornam um código de erro cujos valores simbólicos estão definidos em `mpi.h`. Se a rotina for realizada com sucesso, será retornado `MPI_SUCCESS`. Por outro lado, se ocorrer algum erro na execução da rotina, o código retornado é dependente da implementação do MPI.

2.3 Comunicação Ponto a Ponto

2.3.1 Modos de comunicação

O modo de comunicação permite que se escolha a semântica da operação de envio e, conseqüentemente, permite que se influencie no protocolo da transmissão dos dados.

Os tipos de modos de comunicação são:

- *standard*: quando a operação de envio se completa, não significa, necessariamente, que a operação de recepção correspondente tenha começado e nenhuma suposição pode ser feita pelo programa de aplicação sobre se a mensagem enviada foi copiada, pelo MPI, do *buffer* de envio para um *buffer* temporário.
- *buffered*: o usuário precisa garantir que uma certa quantidade de espaço está disponível para o *buffer* temporário, ou seja, o espaço deste *buffer* precisa ser explicitamente alocado pelo programa de aplicação.
- *synchronous*: uma semântica de "ponto de encontro" é usada entre o emissor e o receptor. Uma operação de envio neste modo pode ser iniciada independentemente do início de uma operação de recepção compatível. Entretanto, um envio será completado com sucesso apenas se uma recepção compatível tiver sido iniciada e se

esta operação tiver iniciado a recepção da mensagem. Portanto, a conclusão de um envio *synchronous* indica que o receptor chegou a um certo ponto da sua execução.

- *ready*: permite ao usuário explorar conhecimentos extras para simplificar o protocolo e potencialmente atingir alto desempenho. Em um envio neste modo, o usuário assegura que o receptor correspondente já foi inciciado. Caso contrário, ocorre um erro.

Os conceitos de chamadas bloqueantes e não-bloqueantes – apresentados na secção 2.2.3.Tipos de chamadas MPI – e estes conceitos de modos de comunicação são ortogonais. Logo, para cada modo de comunicação, existe uma rotina de envio bloqueante e outra não-bloqueante.

2.3.2 Operações bloqueantes básicas

Esta secção descreve apenas as seguintes operações bloqueantes: de envio – no modo *standard* – e de recepção.

2.3.2.1 Envio bloqueante

Sinopse

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int receiverRank, int tag, MPI_Comm comm)
```

`MPI_Send` realiza um envio de mensagem bloqueante e no modo *standard*.

Os parâmetros desta rotina são: `buf`, o endereço inicial do *buffer* de envio; `count`, o número de elementos existentes no *buffer*; `datatype`, o tipo de dado de cada elemento do *buffer*; `receiverRank`, o *rank* da tarefa que recebe a mensagem; `tag`, um número, definido pelo usuário, usado para auxiliar a identificação de uma mensagem e `comm`, o *communicator* ao qual as tarefas emissora e receptora pertencem, ou seja, representa o domínio da comunicação.

Antes de se chamar esta rotina, deve-se alocar espaço para o *buffer* de envio (`buf`) de modo a suportar `count` elementos consecutivos do tipo `datatype`. Após o retorno da

mesma, o *buffer* de envio pode ser alterado, pois a mensagem já foi copiada para outro lugar.

O *buffer* de envio consiste de `count` entradas sucessivas cujo tipo é indicado por `datatype` e inicia-se no endereço indicado por `buf`. Deve-se notar que o tamanho da mensagem é especificado pelo número de entradas e não pelo número de *bytes*. O `count` pode ser zero, neste caso a parte da mensagem que representa o dado está vazia. Os tipos de dados básicos correspondem aos tipos de dados básicos da linguagem de programação usada. Os valores possíveis para este argumento para a linguagem C e os tipos de dados em C correspondentes estão listados na tabela abaixo:

Tabela 1: Tipos de dados em MPI e os correspondentes em linguagem C

MPI	C ANSI
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	unsigned char
MPI_PACKED	

Não há, em C ANSI, o tipo de dado correspondente à MPI_PACKED, que representa um tipo de dados criado pelo usuário em tempo de execução. Este tipo de dado não é tratado neste trabalho.

Para que uma mensagem seja enviada com sucesso, a mesma deve possuir, além do dado a ser transmitido, um envelope, que consiste de: um *communicator*, o *rank* do emissor, o *rank* do receptor e um rótulo. Estes dados são usados para se distinguir e seletivamente receber as mensagens. Em uma operação de envio, a origem da mensagem é implicitamente determinada pela identidade do emissor da mensagem, enquanto os outros campos são especificados através dos argumentos.

2.3.2.2 Recepção bloqueante

Sinopse

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

`MPI_Recv` realiza uma recepção de mensagem bloqueante.

Os parâmetros desta rotina são: `buf`, o endereço inicial do *buffer* de recepção; `count`, o número máximo de elementos que o *buffer* pode comportar; `datatype`, o tipo de dado de cada elemento do *buffer*; `senderRank`, o *rank* da tarefa da qual se deseja receber a mensagem; `tag`, um número, definido pelo usuário, usado para auxiliar a identificação de uma mensagem; `comm`, o *communicator* ao qual as tarefas emissora e receptora pertencem, ou seja, representa o domínio da comunicação e `status`, que é o endereço de uma variável do tipo `MPI_Status`, onde são retornadas informações sobre a mensagem recebida.

Antes de se chamar esta rotina, deve-se alocar espaço para o *buffer* de recepção (`buf`) de modo a comportar no máximo `count` elementos consecutivos do tipo `datatype`. Após o retorno da mesma, o *buffer* de recepção contém a mensagem recebida. O tamanho da mensagem recebida deve ser menor ou igual ao tamanho do *buffer* de recepção. Se uma mensagem menor do que o tamanho do *buffer* de recepção chegar, a mesma é armazenada nas posições iniciais do *buffer* de recepção e as posições restantes não são modificadas.

Uma mensagem pode ser recebida se seu envelope for compatível com os valores da origem, do `tag` e do `comm` especificados pelos argumentos da operação de recepção. Pode-se especificar um valor coringa para o `senderRank` (`MPI_ANY_SOURCE`) e/ou um valor coringa para o `tag` (`MPI_ANY_TAG`), indicando-se que qualquer origem e/ou `tag` é (são) aceito(s). Não se pode especificar um valor coringa para o `comm`. O parâmetro `tag` pode ser usado para se distinguir mensagens vindas de um mesmo emissor.

Esta rotina não especifica o tamanho da mensagem a ser recebida, mas apenas um valor máximo. A origem ou o `tag` da mensagem recebida podem ser desconhecidos se forem usados valores coringas na operação de recepção. Estas informações podem ser obtidas através do argumento `status`, cujo tipo é definido pelo MPI. Em C, `status` é uma estrutura que contém os campos `MPI_SOURCE`, `MPI_TAG` e `MPI_ERROR`.

O argumento `status` também possui a informação sobre o tamanho da mensagem recebida. Entretanto, esta informação não é disponibilizada diretamente através de um campo da variável `status`. É necessário se fazer uma chamada à rotina `MPI_Get_count` para se decodificar esta informação.

Sinopse

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
                  int *count)
```

`MPI_Get_count` computa o número de entradas no *buffer* recebidas.

Os parâmetros desta rotina são: o `status` e o `datatype` usados na recepção da mensagem, bem como um ponteiro para uma variável inteira, onde é retornado o número de entradas recebidas.

2.3.3 Comunicação não-bloqueante

Pode-se melhorar o desempenho em vários sistemas sobrepondo-se comunicação e computação.

As operações de comunicação não-bloqueantes são feitas em duas etapas: primeiro, usa-se uma operação de início de envio (ou de recepção) da mensagem e, após se fazer toda a computação que não dependa desta comunicação, usa-se outra operação para se concluir a comunicação.

Para se fazer a ligação entre estas duas etapas, usa-se uma variável do tipo `MPI_Request`, chamada de objeto *request*. Os objetos *requests* são alocados pelo MPI.

O usuário pode acessar objetos *request* através de várias rotinas MPI para investigar sobre o status das operações de comunicação pendentes ou para esperar pela sua conclusão.

Envios não-bloqueantes podem ser compatíveis com recepções bloqueantes e vice-versa.

Segundo Marc Snir et al. (1996, p.50), "A comunicação geralmente terá menos *overhead* se um *buffer* de recepção já tiver sido alocado previamente quando um emissor iniciar a comunicação. O uso de recepção não-bloqueante permite que se inicie uma recepção "cedo" e, assim, que se alcance um *overhead* de comunicação mais baixo sem que o receptor seja bloqueado enquanto espera pelo emissor".

2.3.3.1 Operações de início (*posting operations*)

O prefixo I (de *immediate*) indica que a chamada é não-bloqueante.

Sinopse

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int receiverRank, int tag, MPI_Comm comm,
              MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int senderRank, int tag, MPI_Comm comm,
              MPI_Request *request)
```

`MPI_Isend` inicia um envio não-bloqueante e no modo *standard*. Os parâmetros desta rotina são os mesmos da rotina `MPI_Send`, descritas no item 2.3.2.1.Envio bloqueante, acrescidos da variável `request`, descrita no item 2.3.3.Comunicação não-bloqueante.

Uma operação de início de envio não-bloqueante indica que o sistema pode iniciar a cópia do dado do *buffer* de envio para outro lugar. O emissor não pode fazer acesso a nenhuma parte do *buffer* de envio depois que tal operação for iniciada, até que uma outra operação de conclusão de envio retorne.

MPI_Irecv inicia uma recepção não-bloqueante. Os parâmetros desta rotina são os mesmos da rotina `MPI_Recv`, descrita no item 2.3.2.2.Recepção bloqueante – porém sem o argumento `status` –, acrescidos da variável `request`, descrita no item 2.3.3.Comunicação não-bloqueante.

Estas rotinas alocam um objeto request e retornam um descritor para o mesmo através do parâmetro `request`, que é usado para se verificar o status da comunicação ou para se esperar pela sua conclusão.

Um início de recepção não-bloqueante indica que o sistema pode começar a escrever o dado no *buffer* de recepção. O receptor não pode fazer acesso a nenhuma parte do *buffer* de recepção depois que esta operação for iniciada, até que uma outra operação de conclusão de recepção retorne.

2.3.3.2 Operações de Conclusão (*completion operations*)

Para se concluir as operações de envio e recepção não-bloqueantes, são usadas as rotinas `MPI_Wait` e `MPI_Test`. Após a conclusão de uma operação de envio, o emissor pode alterar o conteúdo do *buffer* de envio. Após a conclusão de uma operação de recepção, o *buffer* de recepção contém a mensagem, o receptor pode fazer acesso ao mesmo e o argumento `status` está atualizado.

Sinopse

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

MPI_Wait espera a conclusão da operação identificada por `request`. Após o retorno desta função, o objeto apontado por `request` foi desalocado e a ele é atribuído o valor `MPI_REQUEST_NULL`. O objeto `status` contém informações sobre a operação concluída.

MPI_Test verifica se a operação identificada por `request` foi concluída. Neste caso, o parâmetro `flag` possui o valor 1 (`true`), o objeto `status` contém informações sobre a operação concluída e o objeto apontado por `request` foi desalocado por esta operação e

a ele é atribuído o valor `MPI_REQUEST_NULL`. Caso contrário, o `flag` retorna o valor 0 (`false`) e o valor de `status` é indefinido.

Para ambas as rotinas, o valor da variável `status` é indefinido para uma operação de envio.

2.3.4 Operações nos outros modos

Conforme apresentado no item 2.3.1.Modos de comunicação, existem três outros modos de envio de mensagem além do modo *standard*. As rotinas que realizam estes envios iniciam-se com uma letra indicando o modo: B, para *buffered*, S, para *synchronous* e R, para *ready*. Existe apenas um modo de recepção, que é compatível com todos os modos de envio. Para cada modo de envio existe uma rotina bloqueante e outra não-bloqueante. Os nomes das rotinas não-bloqueantes iniciam-se com a letra I.

2.3.4.1 Bloqueantes

Sinopse

```
int MPI_Bsend(void *buf, int count, MPI_Datatype datatype,
              int receiverRank, int tag, MPI_Comm comm)

int MPI_Ssend(void *buf, int count, MPI_Datatype datatype,
              int receiverRank, int tag, MPI_Comm comm)

int MPI_Rsend(void *buf, int count, MPI_Datatype datatype,
              int receiverRank, int tag, MPI_Comm comm)
```

`MPI_Bsend`, **`MPI_Ssend`** e **`MPI_Rsend`** realizam um envio bloqueante, nos modos *buffered*, *synchronous* e *ready*, respectivamente. Os argumentos destas funções são os mesmos da rotina `MPI_Send`, descrita no item 2.3.2.1.Envio bloqueante.

2.3.4.2 Não-bloqueantes

Sinopse

```
int MPI_Ibsend(void *buf, int count, MPI_Datatype datatype,
               int receiverRank, int tag, MPI_Comm comm,
               MPI_Request *request)

int MPI_Issend(void *buf, int count, MPI_Datatype datatype,
               int receiverRank, int tag, MPI_Comm comm,
               MPI_Request *request)

int MPI_Irsend(void *buf, int count, MPI_Datatype datatype,
               int receiverRank, int tag, MPI_Comm comm,
               MPI_Request *request)
```

`MPI_Ibsend`, **`MPI_Issend`** e **`MPI_Irsend`** realizam um envio não-bloqueante, nos modos *buffered*, *synchronous* e *ready*, respectivamente. Os argumentos destas funções são os mesmos da rotina `MPI_Isend`, descrita no item 2.3.3.1.**Operações de início (posting *operations)***.

Estas operações são completadas com as mesmas rotinas Wait e Test usadas no envio em modo *standard*.

2.3.4.3 Alocação e uso de *buffer* para envio no modo *buffered*

Para enviar mensagens no modo *buffered*, a aplicação precisa especificar um *buffer*, que será usado como *buffer* temporário.

Sinopse

```
int MPI_Buffer_attach(void *buffer, int size)
```

`MPI_Buffer_attach` fornece ao MPI um *buffer* para ser usado no armazenamento das mensagens que são enviadas no modo *buffered*. O argumento *buffer* é o endereço do início da região de memória que será usada como *buffer* temporário e que precisa ser alocada previamente pelo usuário. O argumento `size` indica qual é o tamanho do *buffer* especificado, em bytes. Este tamanho deve ser grande o suficiente para comportar todas

as mensagens que serão enviadas – no modo *buffered* –, antes que ocorram as recepções compatíveis. Apenas um *buffer* pode ser anexado de cada vez (por tarefa).

Sinopse

```
int MPI_Buffer_detach(void *buffer, int size)
```

`MPI_Buffer_detach` desanexa o *buffer* associado ao MPI. A chamada retorna o endereço e o tamanho do *buffer* desacoplado. Esta operação será bloqueada até que todas as mensagens presentes no *buffer* sejam transmitidas. Após o retorno desta função, o usuário pode desalocar ou reutilizar o espaço armazenado para o *buffer*.

No quadro que se segue, é apresentado um exemplo com chamadas para anexar e desanexar *buffers* temporários para envio no modo *buffered*.

Quadro 4: Exemplo de chamadas para anexar e desanexar *buffers* temporários.

```
#define BUFFSIZE 10000
int size;
char *buff;
buff = (char *)malloc(BUFFSIZE);
MPI_Buffer_attach(buff, BUFFSIZE);
/* Um buffer com 10000 bytes pode ser usado por MPI_Bsend */
...
MPI_Buffer_detach(&buff, &size);
/* O tamanho do buffer foi reduzido a 0 */
MPI_Buffer_attach(buff, size);
/* Um buffer de 10000 bytes disponível de novo */
...
```

2.3.5 Resumo das operações ponto-a-ponto

Tabela 2: Operações de envio

Nome	Características	Observações
`MPI_Send`	bloqueante, modo *standard*	`----------`
`MPI_Isend`	não-bloqueante, modo *standard*	`----------`
`MPI_Ssend`	bloqueante, modo *synchronous*	`----------`
`MPI_Issend`	não-bloqueante, modo *synchronous*	`----------`
`MPI_Rsend`	bloqueante, modo *ready*	`----------`
`MPI_Irsend`	não-bloqueante, modo *ready*	`----------`
`MPI_Bsend`	bloqueante, modo *buffered*	`MPI_Buffer_atach` e `MPI_Buffer_detach` são usadas no envio no modo buffered.
`MPI_Ibsend`	não-bloqueante, modo *buffered*	

Tabela 3: Operações de recepção

Operações de Recepção	
Nome	**Características**
`MPI_Recv`	bloqueante
`MPI_Irecv`	não-bloqueante

Tabela 4: Operações de conclusão

Operações de Conclusão
`MPI_Wait e MPI_Test`

2.4 Comunicação Coletiva

2.4.1 Introdução

As operações de comunicação coletiva são usadas para se transmitir dados entre todas as tarefas de um mesmo grupo especificado por um *communicator*, com exceção da operação barreira, que é usada para sincronizar tarefas sem troca de dados.

Em relação às operações de comunicação ponto-a-ponto, as operações coletivas são mais restritivas e simplificadas. Uma restrição é que, em contraste com aquelas, estas exigem que a quantidade de dados enviada seja igual à quantidade de dados especificada pelo receptor.

A principal simplificação é que todas as operações coletivas são bloqueantes e no modo *standard*, ou seja, uma função coletiva pode retornar assim que a sua participação em todo o processo estiver completada, independentemente das outras tarefas. Além disso, sua conclusão indica que o chamador da função pode fazer acesso e modificar os *buffers* de comunicação.

Algumas destas operações possuem uma única origem ou uma única tarefa receptora, que é denominada *root*. Nas chamadas onde se especifica uma tarefa *root*, alguns dos argumentos especificados são significativos somente para a tarefa *root*, enquanto as demais tarefas ignoram tais argumentos.

2.4.2 Principais operações coletivas

Na Figura 2 são representadas as principais operações coletivas. Cada linha das matrizes representa os dados pertencentes a uma tarefa. Portanto, no caso do *broadcast*, inicialmente apenas a primeira tarefa contém o dado A0, mas após a operação de *broadcast*, todas as tarefas contêm uma cópia deste dado.

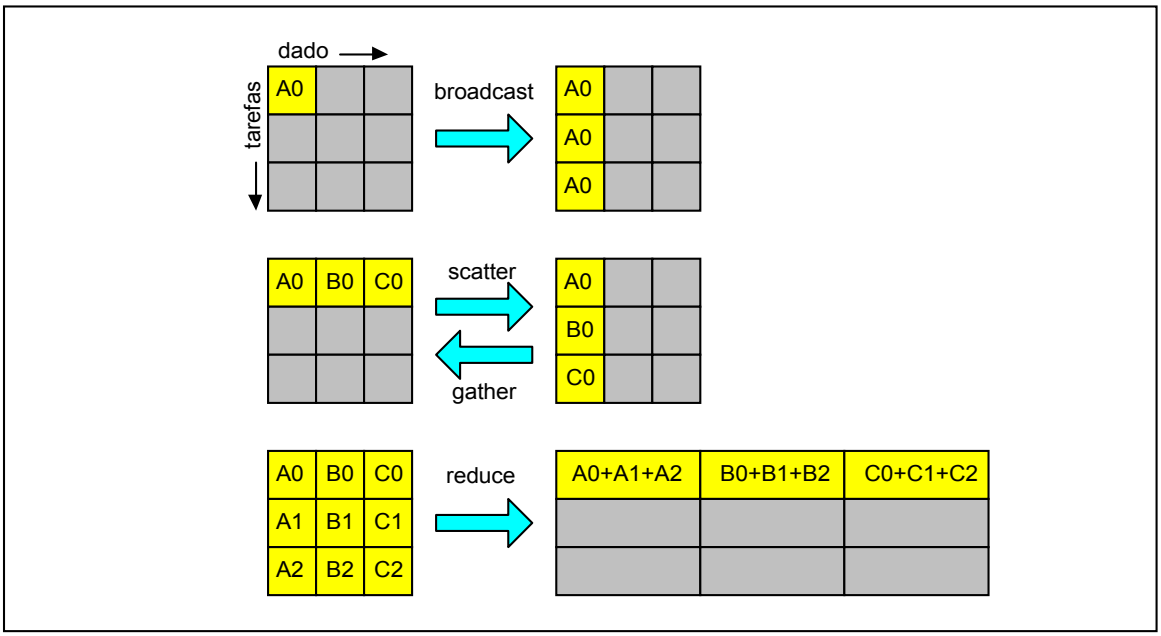

Figura 2: Principais operações coletivas

2.4.2.1 Operação barreira

Sinopse

```
int MPI_Barrier(MPI_Comm comm)
```

`MPI_Barrier` bloqueia a tarefa chamadora até que todos os membros do grupo tenham executado esta operação. Como parâmetro, é passado o *communicator* que define o grupo de tarefas.

2.4.2.2 Operação *Broadcast*

Sinopse

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

`MPI_Bcast` transmite a mensagem da tarefa *root* para todas as tarefas do grupo, que é especificado pelo argumento `comm`.

Os argumentos `count`, `datatype`, `root` e `comm` devem ter seus respectivos valores idênticos entre todas as tarefas do grupo. O argumento `buffer` é o endereço de um *buffer* alocado pelo usuário e, no caso da tarefa *root*, contém o dado a ser transmitido, enquanto para as demais tarefas, conterá o dado recebido após o retorno da rotina.

O fragmento de código do Quadro 5, exemplifica o uso da operação *broadcast* para a transmissão de 100 inteiros da tarefa 0 para as demais tarefas do grupo.

Quadro 5: Transmissão de 100 inteiros da tarefa 0 para as demais tarefas do grupo

```
MPI_Comm comm;
int      array[100];
int      root = 0;
...
MPI_Bcast(array, 100, MPI_INT, root, comm);
...
```

2.4.2.3 Operação *Gather*

Sinopse

```
int MPI_Gather(void* sendBuffer, int sendCount, MPI_Datatype sendType,
               void* recvBuffer, int recvCount, MPI_Datatype recvType,
               int root, MPI_Comm comm)
```

`MPI_Gather` coleta as mensagens enviadas por todas as tarefas do grupo. Cada tarefa (inclusive a *root*) transmite o conteúdo do seu *buffer* de envio para a tarefa *root*. A tarefa *root* recebe as mensagens e as armazena no *buffer* de recepção ordenadas pelo *rank* do emissor.

Os argumentos desta rotina são: `sendBuffer`, `sendCount` e `sendType`, especificam o *buffer* de envio; `recvBuffer`, `recvCount` e `recvType`, especificam o *buffer* de recepção para a tarefa *root* e são ignorados por todas as outras tarefas; `root`, identifica a tarefa *root* e `comm`, identifica o *communicator*.

A assinatura do tipo (`sendCount` e `sendType`) em todos os processos deve ser igual a assinatura do tipo (`recvCount` e `recvType`) da tarefa *root*.

Deve-se notar que o argumento `recvCount` indica o número de elementos recebidos de **cada** tarefa pela tarefa *root* e não o número total de elementos recebidos.

No fragmento de código do Quadro 6, são agrupados na tarefa *root* 100 inteiros de cada tarefa do grupo. A Figura 3 ilustra esta operação.

Quadro 6: Agrupamento de 100 inteiros de cada tarefa na tarefa *root*

```
MPI_Comm comm;
int      gsize, sendBuf[100];
int      root, myRank, *recvBuf;
...
MPI_Comm_rank(comm, &myRank);
if (myRank == root) {
    MPI_Comm_size(comm, &gsize);
    recvBuff = (int *) malloc(gsize * 100 * sizeof(int));
}
MPI_Gather(sendBuf, 100, MPI_INT, recvBuf, 100, MPI_INT, root, comm);
...
```

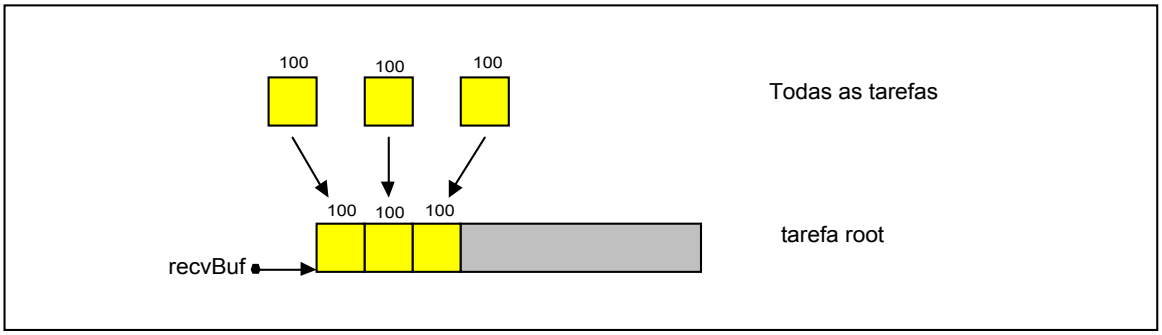

Figura 3: Operação Gather

2.4.2.4 Operação *Scatter*

Sinopse

```
int MPI_Scatter(void* sendBuffer, int sendCount,MPI_Datatype sendType,
                void* recvBuffer, int recvCount,MPI_Datatype recvType,
                int root, MPI_Comm comm)
```

`MPI_Scatter` é a operação inversa à `MPI_Gather`. A tarefa *root* distribui o conteúdo do seu *buffer* de envio para as demais tarefas.

Os argumentos desta rotina são: `sendBuffer`, `sendCount` e `sendType`, especificam o *buffer* de envio para a tarefa *root* e são ignorados pelas demais tarefas; `recvBuffer`, `recvCount` e `recvType`, especificam o *buffer* de recepção; `root`, identifica a tarefa *root* e `comm`, identifica o *communicator*.

A assinatura do tipo (`recvCount` e `recvType`) em todos os processos deve ser igual a assinatura do tipo (`sendCount` e `sendType`) da tarefa *root*.

No fragmento de código do Quadro 7, são distribuídos conjuntos de 100 inteiros da tarefa *root* para as demais tarefas. A Figura 4 ilustra esta operação.

Quadro 7: Distribuição de conjuntos de 100 inteiros da tarefa *root* para as demais

```
MPI_Comm comm;
int      gsize, *sendBuf;
int      root, myRank, recvBuf[100];
...
MPI_Comm_rank(comm, &myRank);
if (myRank == root) {
    MPI_Comm_size(comm, &gsize);
    sendBuff = (int *) malloc(gsize * 100 * sizeof(int));
}
MPI_Scatter(sendBuf, 100, MPI_INT, recvBuf, 100, MPI_INT, root, comm);
...
```

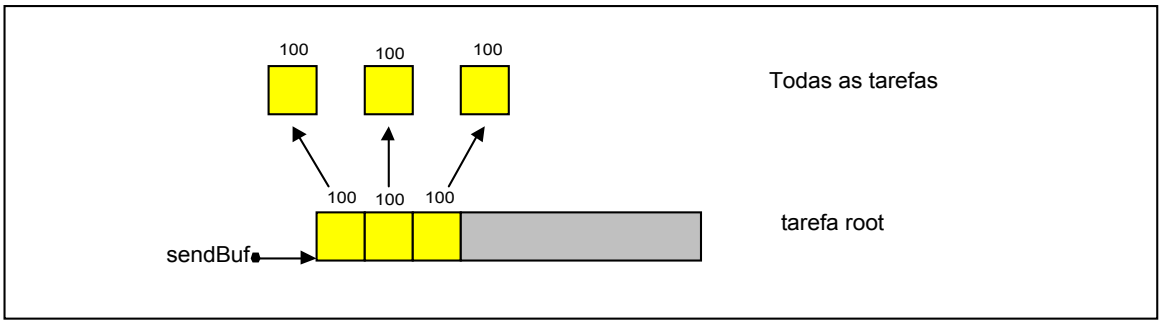

Figura 4: Operação Scatter

2.4.2.5 Operação de redução

Sinopse

```
int MPI_Reduce(void* sendBuf, void* recvBuf, int count,
               MPI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm)
```

`MPI_Reduce` combina os elementos do *buffer* de entrada de cada tarefa do grupo, usando a operação `op`, e retorna o valor combinado no *buffer* de saída da tarefa *root*.

O *buffer* de entrada é definido pelos argumentos `sendBuf`, `count` e `datatype`; o *buffer* de saída é definido pelos argumentos `recvBuf`, `count` e `datatype`. Ambos possuem o mesmo número de elementos com o mesmo tipo. Os argumentos `count`, `datatype`, `op`, `root` e `comm` devem possuir o mesmo valor em todas as tarefas.

O argumento `op` pode ser uma das constantes de operações pré-definidas listadas na Tabela 5.

Tabela 5: Tipos de operações de redução

Nome	Significado
MPI_MAX	máximo
MPI_MIN	mínimo
MPI_SUM	soma
MPI_PROD	produto
MPI_LAND	e lógico
MPI_BAND	e aritmético
MPI_LOR	ou lógico
MPI_BOR	ou aritmético
MPI_LXOR	ou exclusivo lógico
MPI_BXOR	ou exclusivo aritmético
MPI_MAXLOC	valor máximo e localização
MPI_MINLOC	valor mínimo e localização

Maiores informações sobre estas operações e as combinações possíveis entre `op` e `datatype` podem ser obtidas em (SNIR, 1996, p.178).

O exemplo mostrado no Quadro 8 implementa o cálculo do produto interno entre dois vetores com a participação de 3 tarefas MPI. Este exemplo é ilustrado na Figura 5

Quadro 8: Cálculo do produto interno de dois vetores

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
      int v1[] = {1, 2, 3};
      int v2[] = {4, 5, 6};
      int rank, n, prod, sum;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &n);
      if (n != 3) {
            printf("Executar este programa com 3 tarefas.\n");
            MPI_Abort(1);
      }
      prod = v1[rank] * v2[rank];
      MPI_Reduce(&prod, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
      if (rank == 0) {
            printf("sum = %d\n", sum);
      }
      MPI_Finalize();
      return 0;
}
```

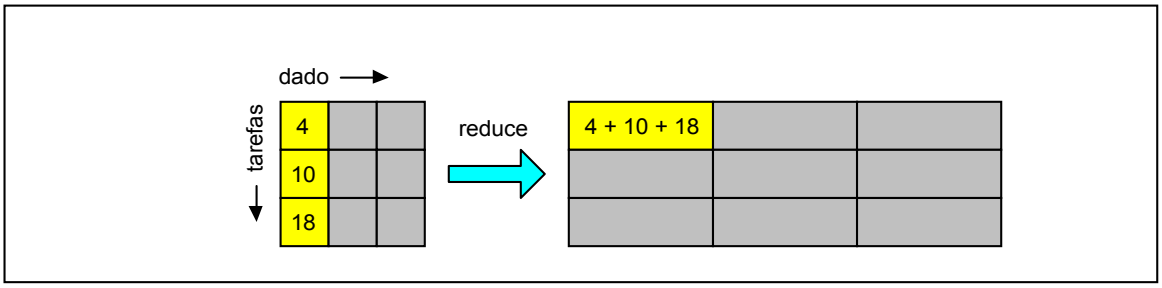

Figura 5: Cálculo do produto interno de dois vetores

2.4.3 Outras operações coletivas

Existem outras operações coletivas no MPI, que são combinações das operações descritas na secção anterior. Maiores informações sobre estas operações podem ser obtidas em (SNIR, 1996, p.147).

3 O Projeto Multiplus e o Sistema Operacional Mulplix

Este capítulo apresenta o projeto Múltiplus, bem como o sistema operacional Mulplix. São feitas algumas considerações sobre processos e *threads* e, finalmente, são apresentadas as primitivas de programação paralela do Mulplix.

3.1 Projeto Multiplus

O projeto Multiplus (AUDE, 1996; AUDE, 1997) tem como objetivo contribuir para o aumento da capacitação tecnológica nacional nas tecnologias e aplicações de computação paralela de alto desempenho. O projeto, que está em desenvolvimento no Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro (NCE/UFRJ), é dividido em 5 linhas de pesquisas fundamentais: arquiteturas paralelas, sistemas operacionais, microarquiteturas de alto desempenho, ambientes de programação paralela e algoritmos paralelos (AUDE, 1998, p. 89).

Neste projeto, está sendo desenvolvido um multiprocessador, também chamado Multiplus, que possui arquitetura modular de memória compartilhada fisicamente distribuída, que suporta até 1024 processadores do tipo SPARC e 32 Gbytes de memória.

3.2 Sistema Operacional Mulplix

Mulplix (AZEVEDO, 1993) é um sistema operacional do tipo Unix capaz de suportar de modo eficiente a exploração de paralelismo na arquitetura do multiprocessador Multiplus. Este sistema operacional é baseado no Plurix (FALLER, 1989), desenvolvido no NCE/UFRJ para dar suporte à concorrência de processos em arquiteturas de memória compartilhada centralizada.

A grande evolução do Mulplix, em relação ao Plurix, se deu na mudança do conceito de processo e na definição do conceito de *thread*. Além disso, o Mulplix disponibiliza

primitivas de programação paralela para o usuário e adequa as políticas de gerência de memória e de escalonamento de *threads* às características da arquitetura do Multiplus.

3.2.1 Processos e *Threads*

Segundo Mauro (2001, p.14), "Um processo é uma abstração que contém o ambiente para um programa do usuário. Ele consiste de um ambiente de memória virtual, recursos para o programa tais como uma lista de arquivos abertos e pelo menos uma linha de execução (*thread*)".

Existem dois modelos no qual um programa pode ser executado: o modelo de processos e o modelo de *threads*.

No modelo de processo, um processo é composto, basicamente, por duas partes:

- Uma parte, que contém os recursos alocados por um programa (dados globais, instruções de programa e arquivos, por exemplo);
- Outra parte, que contém informações sobre o fluxo de execução de um programa (o registrador PC – *Program Counter* – e os dados locais, por exemplo).

Estas partes podem ser tratadas de forma independente pelo sistema operacional.

No modelo de *threads*, um processo pode conter várias *threads*, ou linhas de execução. Neste modelo, um processo é representado pela primeira parte de um processo descrita anteriormente, enquanto cada *thread* é representada pela segunda. Portanto, cada *thread* possui o seu próprio fluxo de execução.

Sun Microsystems (2001, p.18) define *thread* como sendo "uma seqüência de instruções executadas dentro do contexto de um processo".

Um programa pode ser dividido na memória virtual em pelo menos quatro regiões principais: *text*, *data*, *heap* e *stack*. Na região *text*, ficam as instruções do programa; na região *data*, ficam as variáveis de classe estáticas (globais ou locais); na *heap*, ficam as

variáveis dinâmicas globais (as criadas com a função `malloc`) e, na *stack*, as variáveis dinâmicas locais.

No modelo de *threads*, Um processo é, basicamente, representado pelos seus identificadores (PID, UID, GID, etc), pelos recursos alocados (arquivos abertos, *sockets* etc) e pelas regiões de um programa: *text*, *data* e *heap*. Enquanto cada *thread* deste processo é representada por registradores (SP, PC etc) e pela região *stack*.

Neste modelo, cada *thread* compartilha os recursos alocados pelo processo que a criou, bem como as regiões *text*, *data* e *heap* deste mesmo processo.

As motivações para o uso do modelo de *threads* em detrimento do modelo de processos são: (NICHOLS, 1996; STALLINGS, 1998)

- A troca de contexto é mais rápida entre *threads* irmãs – que são criadas pelo mesmo processo – do que entre processos;
- A criação de uma *thread* é mais rápida;
- Possibilidade de compartilhamento de dados entre *threads* irmãs.

3.2.2 Primitivas para programação paralela do Mulplix

O Mulplix disponibiliza um conjunto de primitivas implementadas como chamadas ao sistema operacional, para o desenvolvimento de aplicações de programação paralela na arquitetura Multiplus. Essas primitivas permitem a criação e extinção de *threads*, a alocação e liberação de memória privada e compartilhada e operações de sincronização por exclusão mútua ou ordem parcial (AZEVEDO, 1993a).

Nesta secção, são apresentadas as principais características do ambiente de programação paralela do Mulplix através de exemplos. Cláudio M. P. Santos (1998) descreve todas as primitivas de programação paralela do Mulplix em detalhes.

3.2.2.1 Criação e controle de *threads*

No Mulplix, uma aplicação paralela consiste de um único processo, que possui uma ou mais *threads* executando concorrentemente. Cada processo possui pelo menos uma *thread*, que é a *thread* principal cujo número de identificação é igual a 0 (zero). As *threads* criadas pela *thread* principal também possuem um número de identificação maior do que zero. Este identificador é unívoco entre as *threads* de um mesmo processo.

No programa a seguir, são criadas 3 *threads* que exibem, na saída padrão, uma mensagem com: a sua identificação, a sua ordem dentro do grupo de *threads* criadas e o valor de uma variável global, compartilhada entre todas as *threads*.

Quadro 9 : Exemplo 1 - Criação de *threads*

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

int varGlobal = 8;

void funcaoInicial(int arg, int ord) {
      int tid = thr_id();

      printf("tid = %d - ord = %d - "
            " varGlobal = %d\n", tid, ord, varGlobal);

}

void main(void) {
      int tid = thr_id();

      printf("tid = %d - Eu sou a thread principal.\n", tid);
      thr_spawns(3, funcaoInicial, 0, NULL);
      printf("tid = %d - Eu criei 3 threads filhas.\n");
}
```

Uma das saídas possíveis deste programa é:

```
tid = 0 - Eu sou a thread principal.
tid = 1 - ord = 0 -  varGlobal = 8
tid = 2 - ord = 1 -  varGlobal = 8
tid = 3 - ord = 2 -  varGlobal = 8
tid = 0 - Eu criei 3 threads filhas.
```

Quando o programa acima é executado, a primeira primitiva do Mulplix usada é a `thr_id`, que retorna a identificação da *thread* no processo. Como, neste momento, só existe a *thread* principal, esta rotina retorna 0.

A próxima primitiva do Mulplix, `thr_spawns`, é chamada para a criação de 3 *threads* de forma síncrona, ou seja, a *thread* que chamou esta rotina fica bloqueada até que todas as *threads* criadas terminem a sua execução. Como parâmetro, esta rotina recebe o número de *threads* a serem criadas, o endereço da função que será executada pelas *threads* criadas, o parâmetro passado para as novas *threads* (neste caso, 0 indica nenhum) e um vetor que indica a qual processador cada *thread* deve ser associada (neste caso, como foi passado nulo, o sistema operacional encarrega-se de alocar as *threads* da melhor forma possível).

Cada *thread* criada (*thread* filha), inicia a sua execução na função `funcaoInicial`. Esta função armazena na variável local `tid` o número de identificação desta *thread* e exibe na saída padrão: a sua identificação (`tid`), a sua ordem dentro do grupo de *threads* criadas (`ord`) e o valor da variável global `varGlobal`, que é compartilhada por todas as *threads* e, portanto possui o valor 8 em todas elas.

Após as 3 *threads* criadas pela *thread* principal terem terminado a sua execução, a *thread* principal é desbloqueada e exibe a mensagem: "`tid = 0 - Eu criei 3 threads filhas.`".

Cada *thread* pode terminar a sua execução de 3 formas: executa a última instrução do seu código, chama uma primitiva de término ou é sinalizada por outra *thread* indicando que deve terminar. Quando uma *thread* termina a sua execução, as suas *threads* filhas são automaticamente finalizadas. Portanto, se a *thread* prinicpal terminar a sua execução, todas as *threads* do processo também serão finalizadas.

No Mulplix há duas formas de se criar *threads*: criação síncrona (`thr_spawns`, usada neste exemplo) e criação assíncrona (`thr_spawn`). Os parâmetros recebidos por estas duas primitivas são os mesmos, porém enquanto a primeira cria *threads* e fica

bloqueada até que todas as *threads* criadas terminem a execução, a última cria *threads* e continua a sua execução normalmente, sem ficar bloqueada.

No Mulplix existem dois mecanismos para se fazer a sincronização entre várias *threads*, que são: sincronização por exclusão mútua e por evento.

3.2.2.2 Sincronização por exclusão mútua

No Mulplix, existe o tipo `MUTEX`, que representa um semáforo de exclusão mútua funcionalmente idêntico ao semáforo definido por Dijkstra (Dijkstra, 1965), onde as rotinas `mx_lock` e `mx_free` são correspondentes às operações `P` e `V`, respectivamente. Através da alocação e liberação dos recursos associados a um mutex, é possível criar uma região de exclusão mútua. Para entrar na região crítica, deve-se alocar um dos recursos associados ao mutex. Ao sair, deve-se liberar o recurso para que outras *threads* possam entrar na região crítica.

No exemplo anterior, as *threads* fizeram um acesso concorrente à variável `varGlobal`. Como foi realizada apenas a leitura do valor desta variável, não houve necessidade de sincronizar este acesso.

No próximo exemplo, foram feitas algumas modificações no exemplo anterior onde todas as *threads* filhas alteram o valor da variável compartilhada `varGlobal`, incrementando o valor da mesma antes de exibi-lo. O código que altera esta variável é, portanto, a região crítica que deve ser protegida para que ocorra o acesso concorrente sem problemas. As modificações estão em destaque no quadro abaixo.

Quadro 10: Exemplo 2 - usando exclusão mútua para a sincronização das *threads*

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

MUTEX mutex;
int varGlobal = 8;

void funcaoInicial(int arg, int ord) {
      int tid = thr_id();

      mx_lock(mutex);
      printf("tid = %d - ord = %d - "
            " varGlobal = %d\n", tid, ord, ++varGlobal);
      mx_free(mutex);

}

void main(void) {
      int tid = thr_id();

      printf("tid = %d - Eu sou a thread principal.\n", tid);
      mutex = mx_create(1);
      thr_spawns(3, funcaoInicial, 0, NULL);
      mx_delete(mutex);
      printf("tid = %d - Eu criei 3 threads filhas.\n", tid);
}
```

Uma das saídas possíveis deste programa é:

```
tid = 0 - Eu sou a thread principal.
tid = 1 - ord = 0 -  varGlobal = 9
tid = 2 - ord = 1 -  varGlobal = 10
tid = 3 - ord = 2 -  varGlobal = 11
tid = 0 - Eu criei 3 threads filhas.
```

Neste exemplo foi declarada a variável global `mutex`, que representa o semáforo de exclusão mútua. A *thread* principal atribui a esta variável o semáforo criado com a primitiva `mx_create`. Esta primitiva cria um semáforo de exclusão mútua com o número de recursos associados que foi passado como parâmetro (neste caso, este semáforo possui apenas 1 recurso associado).

Cada *thread* criada, para entrar na região crítica, ou seja, antes de incrementar o valor de `varGlobal`, chama a função `mx_lock`, que aloca um recurso associado ao semáforo passado como parâmetro. Se o número de recursos disponíveis for maior do que zero, decrementa esse valor e permite a *thread* ter acesso à região crítica. Caso contrário, a *thread* será suspensa, sendo colocada em uma lista de espera associada ao semáforo, até

que seja liberada sua entrada na região crítica. Isso só será possível quando outra *thread* sair da região crítica e liberar um dos recursos.

Após incrementar a variável compartilhada entre as *threads*, cada *thread* deve liberar o recurso chamando a função `mx_free`. Esta rotina libera um recurso associado ao semáforo passado como parâmetro, ou seja, incrementa o número de recursos disponíveis. Com isso, permite que uma *thread*, que esteja aguardando na fila de espera associada ao semáforo, possa entrar na região crítica.

A função `mx_delete` destrói o semáforo que é passado como parâmetro.

3.2.2.3 Sincronização por evento

O Mulplix possui o tipo `EVENT`, que é a implementação de um mecanismo de sincronização usado para avisar uma ou mais *threads* sobre a ocorrência de um evento. Esses semáforos permitem que várias *threads* participem de um evento de duas formas: sinalizando a sua participação no evento e/ou esperando a ocorrência do mesmo. Quando uma *thread* sinaliza a sua participação em um evento, ela indica que alcançou um determinado ponto do seu processamento. Após a sinalização, a *thread* continua a sua execução normalmente. Por outro lado, se uma *thread* espera por um evento, a mesma ficará bloqueada até que o evento ocorra. Um evento ocorre quando todas as *threads* envolvidas tiverem participado do mesmo (sinalizando e/ou esperando).

O próximo exemplo é uma modificação do exemplo 1, mostrado no Quadro 9. Neste exemplo, usa-se a função `thr_spawn` (criação assíncrona), ao invés da `thr_spawns` (criação síncrona). Para impedir que a *thread* principal termine antes que as *threads* filhas tenham completado a sua execução, foi usada a sincronização por evento.

O Quadro 11, apresenta o código deste terceiro exemplo, destacando-se as modificações em relação ao primeiro exemplo.

Quadro 11: Exemplo 3 - usando sincronização por evento

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

EVENT sema;
int varGlobal = 8;

void funcaoInicial(int arg, int ord) {
      int tid = thr_id();

      printf("tid = %d - ord = %d - "
            " varGlobal = %d\n", tid, ord, varGlobal);
      ev_signal(sema);
}

void main(void) {
      int tid = thr_id();

      printf("tid = %d - Eu sou a thread principal.\n", tid);
      sema = ev_create(3, 1);
      thr_spawn(3, funcaoInicial, 0, NULL);
      ev_wait(sema);
      ev_delete(sema);
      printf("tid = %d - Eu criei 3 threads filhas.\n", tid);
}
```

Os resultados obitidos com a execução deste exemplo são os mesmos da execução do primeiro exemplo.

Quando este programa é executado, a *thread* principal cria o semáforo de sincronização por evento chamando a função `ev_create`. Esta função recebe como parâmetro, respectivamente, o número de *threads* que devem sinalizar a sua participação no evento e o número de *threads* que devem esperar a ocorrência do evento. A variável global `sema` recebe o semáforo criado pela função `ev_create`.

A função `thr_spawn` é, então, chamada para se criar 3 *threads* filhas, que executarão a função `funcaoInicial`. A `thr_spawn`, por ser assíncrona, não bloqueia a *thread* principal. Para impedir que a *thread* principal termine a sua execução e, conseqüentemente, interrompa a execução das *threads* filhas e termine o programa, é chamada a função `ev_wait`. Esta função bloqueia a *thread* principal até que o evento representado por `sema`, que foi passado como parâmetro, ocorra.

Cada *thread* filha exibe uma mensagem na saída padrão e chama a função `ev_signal` para sinalizar o semáforo `sema` indicando que terminou a sua execução. Quando as 3 *threads* filhas terminarem a sua execução, terão ocorrido 3 sinalizações no semáforo `sema`, a condição para a ocorrência do evento é satisfeita, pois a *thread* principal já executou a espera. A *thread* principal é, então, desbloqueada, exibe uma mensagem e termina a execução do programa.

4 PMMPI: Uma Implementação do MPI Baseada em *Thread*

Neste capítulo, descrevemos o PMMPI (*Portable Multithreaded Message Passing Interface*), que é a implementação que realizamos da especificação MPI para estações de trabalho com suporte a multiprocessamento simétrico, ou seja, estações SMPs (*Symmetric Multiprocessors*).

4.1 Introdução

4.1.1 Implementação do MPI em máquinas SMPs

MPI usa troca de mensagens, que é um modelo de programação (ANDREWS, 1983, p.25) normalmente usado em máquinas paralelas de memória distribuída. Entretanto, existem alguns motivos para se usar este modelo de programação em máquinas SMPs:

i) Como os processadores modernos têm se tornado cada vez mais rápidos, o gerenciamento de seus caches e da hirarquia de memória como um todo tem se tornado a chave para se explorar todo o potencial dos mesmos. O ajuste de desempenho para código MPI em SMPs é normalmente mais fácil, uma vez que o código e o dado particionados apresentam boa localidade na hierarquia de memória (GROPP, 1996, p. 8);

ii) Um programa que usa MPI pode ser transportado facilmente para qualquer máquina paralela sem restrições de plataforma. Isso é especialmente importante para as futuras infraestruturas computacionais heterogêneas tais como "Nasa Information Power Grids" (NASA, 2002);

iii) Pode ser necessária a integração de novas aplicações – desenvolvidas em máquinas SMPs – com programas MPI existentes.

4.1.2 Mapeamento das tarefas MPI em *threads*

As implementações convencionais do padrão MPI mapeiam cada tarefa MPI em um processo do sistema operacional. No intuito de aproveitar de forma mais eficiente os recursos oferecidos pelas estações de trabalho SMPs, a idéia central da nossa

implementação é mapear cada tarefa MPI em uma *thread*. Deste modo, espera-se obter um melhor desempenho pelos seguintes motivos:

i) A troca de contexto entre as *threads* irmãs, ou seja, criadas por um mesmo processo, é mais rápida do que a troca de contexto entre processos;

ii) A criação de *threads* é mais rápida;

iii) Pode-se aproveitar o espaço de endereçamento compartilhado entre as *threads* para a troca de mensagens, usando-se variáveis globais. Por outro lado, como os processos não compartilham memória, a troca de mensagens entre duas tarefas MPI mapeadas em processos precisa ser feita utilizando-se *buffer* do sistema, o que degrada a eficiência da comunicação.

Portanto, uma aplicação PMMPI em execução consiste de um processo que possui várias *threads*, que implementam as tarefas MPI. A comunicação entre as tarefas é feita, pela biblioteca, através de memória compartilhada.

4.1.3 Arquitetura do PMMPI

A Figura 6 representa a arquitetura do PMMPI.

O **Programa do Usuário** usa o **PMMPI** através da **Interface MPI**. A **Implementação Interna** do PMMPI é o objetivo principal deste trabalho. Nela estão as estruturas de dados e os algoritmos necessários para se fazer o mapeamento das tarefas MPI em *threads*, bem como a comunicação entre as mesmas. O PMMPI utiliza as **Primitivas de Programação Paralela do Mulplix**, que são constituídas por aproximadamente 20 protótipos de funções, que podem ser implementadas em diversos **Sistemas Operacionais** e podem usar diversos **Modelos de *Threads***.

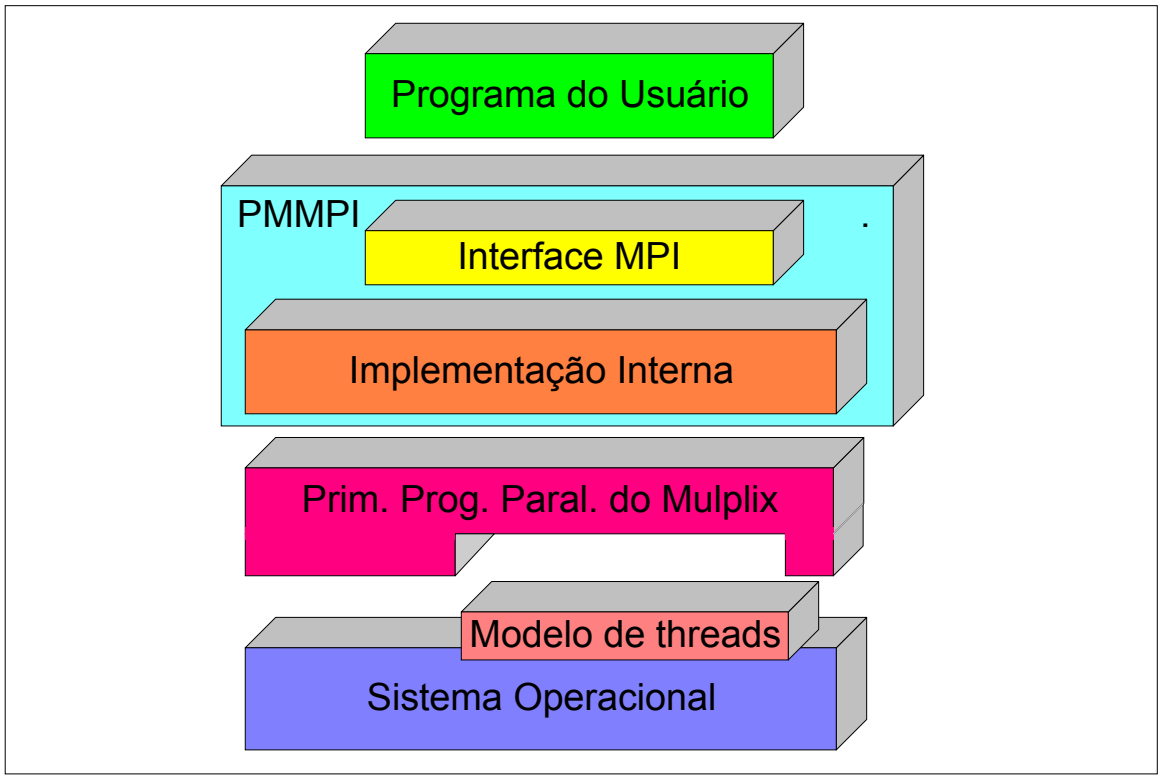

Figura 6: Arquitetura do PMMPI

4.1.4 Portabilidade do PMMPI

O PMMPI não faz nenhuma chamada ao sistema operacional diretamente, pois são utilizadas as primitivas de programação paralela do Mulplix. Conseqüentemente, esta implementação do MPI é independente tanto em relação ao sistema operacional, quanto ao modelo de *threads* usado.

Esta independência do PMMPI não compromete o desempenho, pois as primitivas de programação paralela do Mulplix são apenas protótipos de funções, que são implementados fazendo chamadas ao sistema operacional e usando um modelo de threads específico.

A versão atual do PMMPI pode ser usada por programas escritos em linguagem C ANSI, que podem ser executados no sistema operacional Solaris (para arquitetura SPARC ou INTEL) e pode usar os seguintes modelos de *threads*: *Solaris* LWP (*lightweight process*), *Solaris Threads* e *Pthreads*.

Para se transportar os programas feitos com a biblioteca PMMPI para outros sistemas, basta implementar as primitivas de programação paralela do Mulplix para o sistema desejado.

4.1.5 Trabalhos correlatos

O PMMPI faz parte do Projeto Multiplus. Como parte deste projeto, foi desenvolvida a biblioteca MPVM (SANTOS, 1998) – que é uma implementação do PVM (GEIST, 1994) – também projetada para trabalhar eficientemente em arquiteturas paralelas que aceitam o conceito de *multithreading* e o modelo de memória compartilhada. O MPVM também usa as primitivas de programação paralela do Mulplix.

O PVM, assim como o MPI, é uma biblioteca de troca de mensagens. Embora essas bibliotecas sejam parecidas, há algumas diferenças entre elas, conforme apontado em (GROPP, 1998). Uma característica importante da biblioteca MPI é que suas rotinas são *thread safe*, enquanto as rotinas da biblioteca PVM, não são.

Há outros esforços para desenvolver ambientes PVM baseados em threads, que são: TPVM (FERRARI, 1995), PLWP (CHUANG, 1994) e LPVM (ZHOU, 1997). Dentre estes ambientes, apenas no MPVM e no LPVM a troca de mensagens é realmente *multithreading*.

A maioria das primeiras implementações do padrão MPI é focada em máquinas de memória distribuída ou em *cluster* de estações de trabalho (BRUCK, 1997). O projeto MPI-SIM (PRAKASH, 1998) usou *multithreading* para simular a execução de MPI em máquinas de memória distribuída para predição de desempenho. Existem vários trabalhos que se preocupam em como múltiplas *threads* podem ser invocadas em uma única tarefa MPI, os quais podem ser encontrados em: (MPI-FORUM, 2002; PROTOPOPOV, 1998; SKJELLUM, 1996), além dos produtos comerciais de MPI desenvolvidos pela Sun, IBM e SGI. Entretanto, estes trabalhos não se preocupam em como executar cada tarefa MPI como uma thread.

Foi encontrado apenas um trabalho que, assim como o PMMPI, executa cada tarefa MPI como uma thread: o TMPI (TANG, 2000), que foi desenvolvido em UCSB (*University of California Santa Barbara*). Esta implementação do MPI propõe técnicas a serem aplicadas em tempo de compilação e em tempo de execução, que permitem que uma grande classe de códigos escritos em linguagem C ANSI e que usam MPI, seja executada como *threads* em máquinas SMPs.

4.2 Diretrizes básicas para se criar programas PMMPI

O programador usa a biblioteca PMMPI praticamente da mesma forma em que usaria a biblioteca MPI, ou seja, cada tarefa possui seus próprios dados e a comunicação entre as mesmas é feita através das operações de troca de mensagem fornecidas pela biblioteca. Entretanto, como um programa PMMPI é essencialmente *multithreaded*, deve-se seguir as seguintes diretrizes:

- Sincronizar o acesso às variáveis globais do programa;
- Usar apenas rotinas *thread-safe*, o que implica em eliminar as variáveis locais estáticas. Para mais informações, vide 5.2.3.Custo da alocação de memória.

Nas implementações convencionais do MPI, as tarefas podem usar variáveis globais e locais estáticas sem problemas pois, normalmente, elas mapeiam cada tarefa em processo. Como cada processo possui seu próprio espaço de endereçamento, é garantido que cada tarefa possui suas variáveis globais e estáticas independentes das mesmas variáveis das outras tarefas. Entretanto, como o PMMPI mapeia cada tarefa em uma *thread* de um mesmo processo, as variáveis globais e estáticas do processo são compartilhadas por todas as *threads*, logo todas as tarefas podem fazer acessos concorrentemente às mesmas variáveis globais e estáticas, o que pode deixar estas variáveis em um estado inconsistente se não for feita uma sincronização nestes acessos.

4.3 Ambiente de execução

O MPI Forum criou - com o MPI-2 - uma definição padrão para um comando de execução chamado `mpiexec`. Tal comando não é obrigatório em uma implementação

MPI para que a mesma esteja compatível com o padrão, mas se esse comando existir, então os argumentos precisam ter o significado especificado no padrão.

Neste trabalho é implementada uma versão compacta do comando `mpiexec`, o `pmmpiexec`. Através dos argumentos deste, são indicados quantas tarefas serão criadas, o nome do programa executável e os seus argumentos.

O comando `pmmpiexec` executa o programa que foi recebido como parâmetro e passa para esse programa, através de uma variável de ambiente, o número de tarefas que serão criadas.

4.4 Estruturas de dados do PMMPI

Algumas estruturas de dados foram implementadas usando as idéias sobre *objeto estático* (ROBBINS, 1996, p.33) – que é uma implementação em C de uma estrutura de dados como se fosse um objeto, ou seja, a estrutura de dados e todas as funções que podem fazer acesso à mesma estão em um mesmo arquivo – e sobre *monitor* (HOARE, 1974) – que é a associação do objeto estático com a exclusão mútua implícita no uso das suas funções.

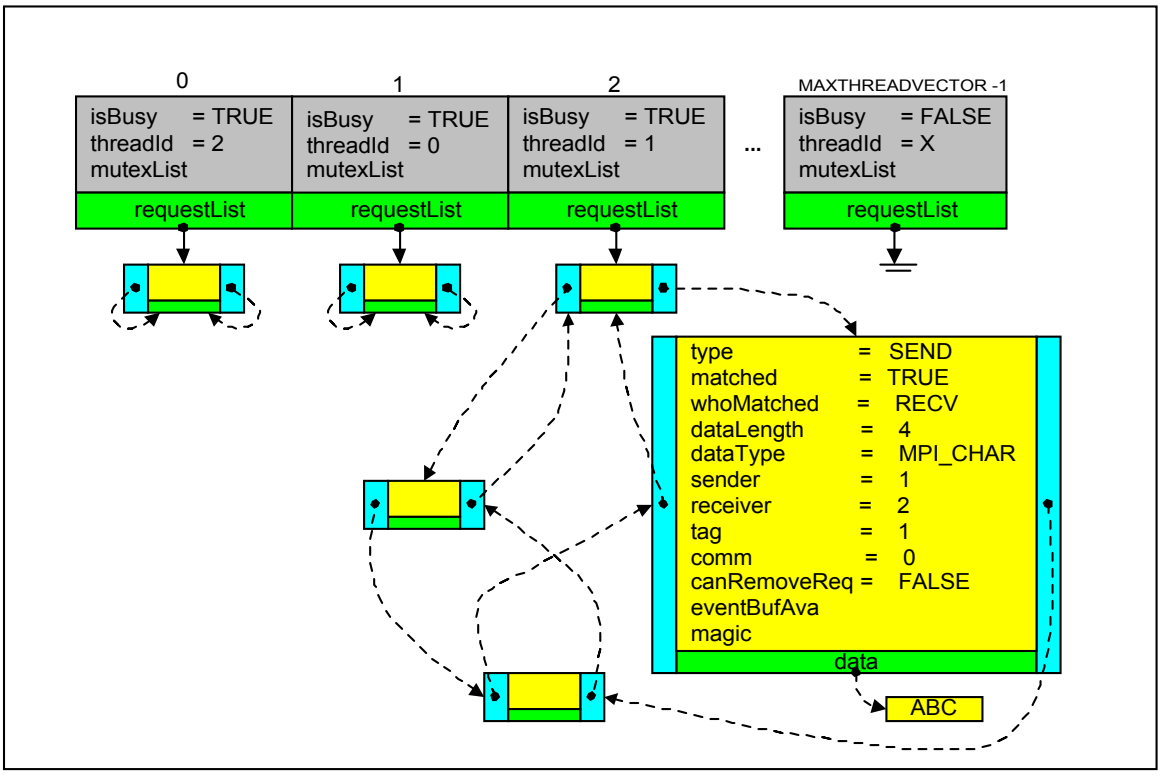

Figura 7: Estrutura de dados ThreadVector

`ThreadVector` é o vetor de *threads*, onde estão armazenadas informações sobre todas as tarefas da aplicação PMMPI, ou seja, todas as *threads* da aplicação. Cada entrada do vetor de *threads* possui os seguintes campos:

- `isBusy`: variável booleana que indica se esta entrada está sendo usada;
- `threadId`: identificador desta *thread*;
- `requestList`: ponteiro para a lista de requisições de mensagens para esta *thread*;
- `mutexList`: variável de exclusão mútua, usada para sincronizar o acesso à `requestList`.

`mutexThreadVector` é uma variável de exclusão mútua usada para sincronizar o acesso ao `ThreadVector`.

`RequestList` é uma lista que contém as requisições de mensagens para a *thread*. Esta é uma lista com cabeça, duplamente encadeada; deste modo, a remoção de qualquer nó da lista é facilitada. É garantido que as requisições de mensagens vindas de um mesmo emissor são armazenadas na lista na ordem em que foram enviadas, assim como também é garantido que as requisições de mensagens geradas pelo receptor são

colocadas na lista na ordem em que foram geradas. Por outro lado, não é garantida a ordem total da comunicação de todas as tarefas PMMPI, conforme especificado no padrão MPI (SNIR, 1996, p.35). Cada nó da lista possui uma entrada com os seguintes campos:

- `type`: indica qual foi o tipo de operação que criou esta requisição. Os tipos podem ser: `SEND`, `ISEND`, `SSEND`, `ISSEND`, `RECV` e `IRECV` indicando envio bloqueante no modo *standard*, envio não-bloqueante no modo *standard*, envio bloqueante no modo *synchronous*, envio não-bloqueante no modo *synchronous*, recepção bloqueante e recepção não-bloqueante, respectivamente.
- `eventBufAva`: variável de sincronização usada para indicar a ocorrência do evento *buffer disponível*, ou seja, a mensagem já foi transmitida entre operações de comunicação ponto-a-ponto compatíveis e o *buffer* do receptor já contém a mensagem.
- `data`: é um ponteiro para a região de memória que contém o dado a ser transmitido – no caso da requisição ter sido criada por uma operação de envio - , ou, para a região de memória onde o dado será copiado – no caso da requisição ter sido criada por uma operação de recepção.
- `dataLength`: o tamanho do dado, em número de entradas em `data` e não em número de bytes.
- `datatype`: o tipo de cada entrada em `data`.
- `sender`: o *rank* da tarefa que envia a mensagem, também chamado de origem.
- `receiver`: o *rank* da tarefa que recebe a mensagem, também chamado de destino.
- `tag`: um rótulo para a mensagem, usado para auxiliar a identificação de uma mensagem.
- `comm`: representa o domínio da comunicação, o *communicator*.
- `matched`: variável booleana que indica se já foi realizada uma operação compatível com esta requisição.
- `whoMatched`: indica qual é o tipo da operação que foi realizada para completar a comunicação representada por esta requisição.
- `canRemoveReq`: variável booleana que indica se a requisição já pode ser removida. É usada, quando as duas operações da comunicação são não-bloqueantes, para decidir qual operação irá remover a requisição.
- `magic`: usado para verificar a consistência da requisição.

`MPI_Request` é um ponteiro para um nó da lista de requisições.

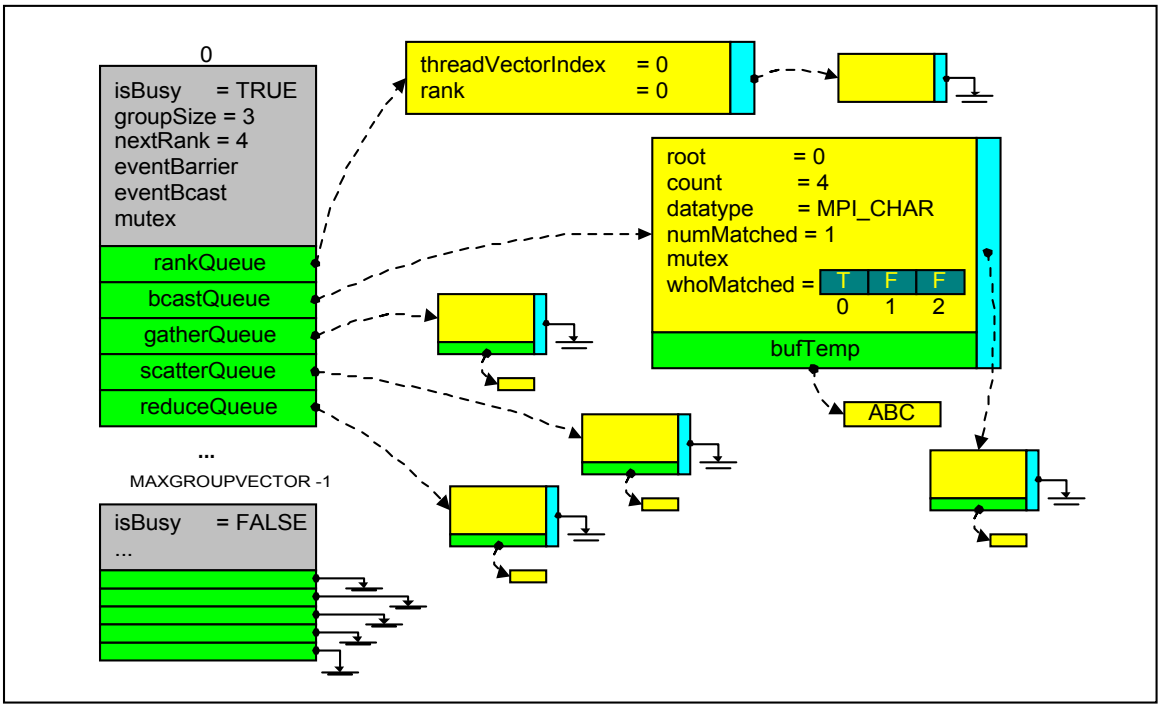

Figura 8: Estrutura de dados GroupVector

`GroupVector` é o vetor de grupos de tarefas. Neste vetor são armazenados todos os grupos de tarefas da aplicação. Atualmente, este vetor possui apenas um grupo com todas as tarefas, pois as rotinas para se criar novos grupos não foram implementadas. Cada entrada do vetor de grupos possui os seguintes campos:

- `isBusy`: variável booleana que indica se esta entrada está sendo usada.
- `rankList`: é a lista de *ranks*, onde são armazenados os *ranks* de todas as tarefas do grupo.
- `mutex`: variável de exclusão mútua. Usada para sincronizar o acesso à `rankQueue`.
- `groupSize`: indica quantas tarefas pertencem ao grupo.
- `eventBarrier`: é uma variável de sincronização, usada na operação `MPI_Barrier` para bloquear as tarefas do grupo até que todas elas executem esta operação.
- `nextRank`: é o *rank* da próxima *thread* que for incluída no grupo.
- `bcastQueue`: é a fila de mensagens criadas pela operação *broadcast*.

- `eventBcast`: é uma variável de sincronização, usada para indicar que foi criada uma nova entrada na `bcastQueue`.
- `scatterQueue`: é a fila de mensagens criadas pela operação *scatter*.
- `eventScatter`: é uma variável de sincronização, usada para sinalizar a criação de uma nova entrada na `scatterQueue`;
- `gatherQueue`: é a fila de mensagens criadas pela operação *gather*;
- `eventGather`: é uma variável de sincronizaçao, usada para sinalizar a criação de uma nova entrada na `gatherQueue`;
- `eventGatherComp`: é uma variável de sincronização, usada para sinalizar a conclusão de uma operação *gather*;
- `reduceQueue`: é a fila de mensagens criadas pela operação *reduce*;
- `eventReduce`: é uma variável de sincronização, usada para sinalizar a conclusão de uma operação *reduce*.
- `eventReduceComp`: é uma variável de sincronização, usada para sinalizar a conclusão de uma operação *reduce*.

`mutexGroupVector` é uma variável de exclusão mútua usada para sincronizar o acesso ao `GroupVector`.

`rankList` é uma lista contendo os *ranks* de todas as tarefas do grupo. Cada nó da lista possui uma entrada com os seguintes campos:

- `threadVectorIndex`: é o descritor da *thread* no `ThreadVector`.
- `rank`: é o *rank* da *thread* no grupo. É um número inteiro que varia de 0 até `groupSize` –1.

`bcastQueue`, `gatherQueue` e `scatterQueue` são filas de mensagens das operações *broadcast*, *gather* e *scatter*, respectivamente. Cada nó destas filas possui uma entrada com os seguintes campos:

- `root`: *rank* da tarefa que é a origem do *broadcast*.
- `count`: é o número de entradas em `bufTemp`.
- `datatype`: é o tipo de cada entrada em `bufTemp`.
- `numMatched`: é o número de tarefas que já "casaram" esta mensagem, ou seja, já copiaram o dado para os seus próprios *buffers*.

- `whoMatched`: é um vetor de valores booleanos. Cada entrada deste vetor indica se a tarefa cujo *rank* é igual a posição desta entrada no vetor já copiou o dado de `bufTemp` ou não.
- `bufTemp`: é um *buffer* temporário onde fica armazenada a mensagem que é transmitida na operação *broadcast*.
- `mutex`: variável de exclusão mútua. Usada para sincronizar o acesso à esta entrada da `bcastQueue`.

`reduceQueue`: é uma fila de mensagens da operação *reduce*. Cada nó desta fila possui os mesmos campos de `bcastQueue`, acrescidos do campo:

- `op`: é uma constante que indica qual operação será realizada na operação de redução.

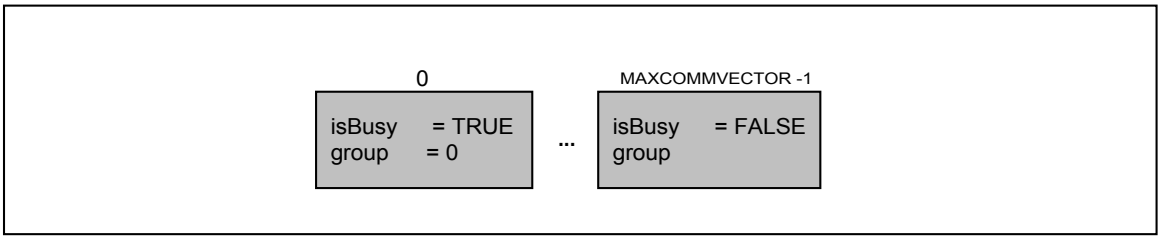

Figura 9: Estrutura de dados CommVector

`CommVector` é o vetor de *communicators*. Como não foram implementadas as operações para a criação de *communicators*, este vetor possui apenas uma entrada que representa o *communicator* com todas as tarefas da aplicação. Tal *communicator* pode ser identificado através do descritor `MPI_COMM_WORLD`. Cada entrada do vetor de *communicators* possui os seguintes campos:

- `isBusy`: variável booleana que indica se esta entrada está sendo usada.
- `group`: é um descritor do grupo de tarefas no `GroupVector`.

`mutexCommVector` é uma variável de exclusão mútua usada para sincronizar o acesso ao `CommVector`.

4.5 Descrição das rotinas implementadas

Como era esperado, a interface das rotinas não precisou ser alterada, pois o padrão MPI foi projetado para ser portátil e, confirmando os testes feitos por (TREUMANN, 1998), as rotinas do MPI são *thread-safe*.

4.5.1 Rotinas implementadas no PMMPI

As rotinas do MPI implementadas neste trabalho são:

Tabela 6: Rotinas implementadas no PMMPI

`MPI_Init()`	`MPI_Wait()`
`MPI_Finalize()`	`MPI_Test()`
`MPI_Comm_rank()`	`MPI_Barrier()`
`MPI_Comm_size()`	`MPI_Bcast()`
`MPI_Send()`	`MPI_Gather()`
`MPI_Recv()`	`MPI_Scatter()`
`MPI_Isend()`	`MPI_Reduce()`
`MPI_Irecv()`	`MPI_Abort()`

4.5.2 Criação e destruição do ambiente de execução do PMMPI

A figura que se segue representa a inicialização do ambiente de execução do PMMPI.

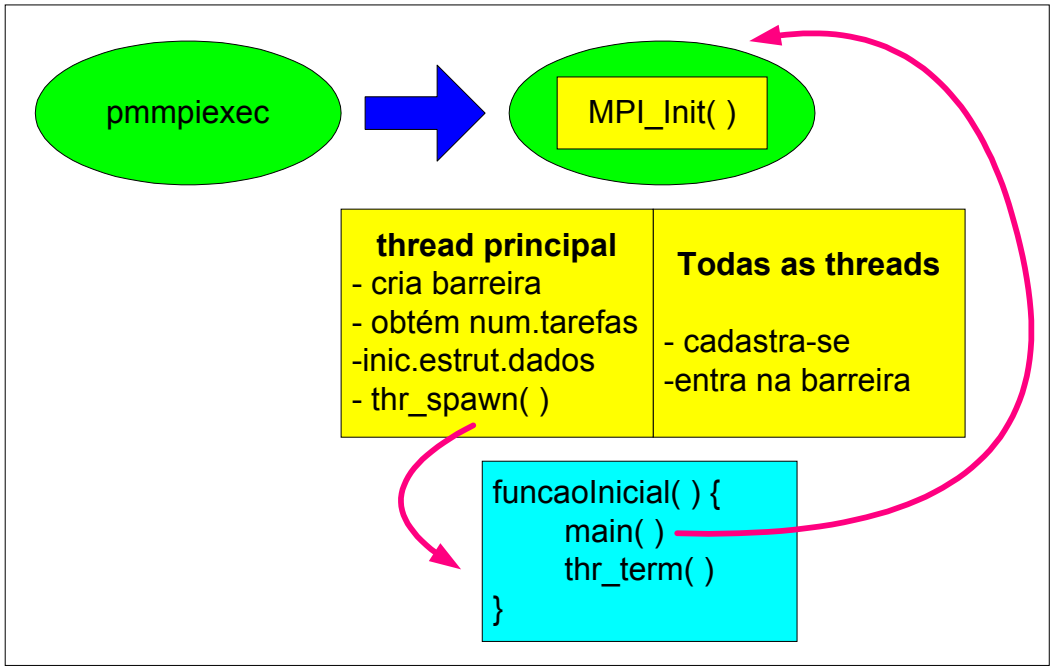

Figura 10: inicialização do ambiente de execução do PMMPI

O programa `pmmpiexec`, descrito no item 4.3.Ambiente de execução, cria um novo processo e troca a imagem do mesmo pelo arquivo executável que recebeu como parâmetro. Portanto, este novo processo representará a aplicação PMMPI. Este comando também passa, através de variável de ambiente, o número de tarefas que deverão ser criadas.

A aplicação, então, inicia a sua execução com apenas uma *thread*, que é a *thread* principal do processo, cujo número de identificação (tid) é igual a 0 (zero). A primeira rotina da biblioteca PMMPI chamada é a `MPI_Init()`, que recebe como parâmetro os endereços dos argumentos `argc` e `argv` da função `main()`.

Sinopse

```
int MPI_Init(int *argc, char ***argv);
```

`MPI_Init()` inicializa o ambiente de execução da biblioteca PMMPI. Esta rotina cria as estruturas de dados básicas da biblioteca e as inicializa, bem como cria as *threads* que implementam as tarefas e as cadastra nas estruturas de dados.

No Quadro 12, é apresentado o algoritmo desta rotina.

Quadro 12: Algoritmo da rotina MPI_Init( )

```
//variaveis globais
int *argcGlobal;
char ***argvGlobal;

int MPI_Init(int *argc, char ***argv) {
      int numDeThreads;

      numDeThreads = obterNumeroDeThreads();  /* do ambiente */
      tid = thr_id();
      se(tid == 0) {
            //código executado apenas pela thread principal
            barreiraInicial  =  ev_create(numDeThreads,  numDeThreads);
            inicializarEstruturasDeDados();

            //criar as demais tarefas
            argcGlobal = argc;
            argvGlobal = argv;
            numDeFilhas = numDeThreads – 1;
            thr_spawn(numDeFilhas, funcaoInicial, 0, NULL);
            //alterar o nivel de concorrencia
            novoNivel = (numOfThreads < MAXPROCESSOR)
                        ? numOfThreads : MAXPROCESSOR;
            thr_setConcurrency(novoNivel);
      }

      //código executado por todas as threads
      cadastrarNoThreadVector(tid);
      cadastrarNoGroupVector(tid);

      //entrar na barreira inicial
      ev_swait(barreiraInicial);
}

funcaoInicial(int arg, int ord) {
      main(*globalArgc, *globalArgv);
      thr_term(0);
}
```

No `MPI_Init()`, existe um trecho de código que é executado apenas pela *thread* principal. Nele são criadas as estruturas de dados do PMMPI, bem como as *threads*, que implementam as outras tarefas MPI e, finalmente, é dada uma sugestão para o sistema operacional de quantas *threads* serão executadas em paralelo, através da rotina `thr_setConcurrency()`. Para se criar estas *threads*, é chamada uma primitiva do

Mulplix, a rotina `thr_spawn()`. Esta rotina recebe, através dos seus parâmetros, o número de *threads* que serão criadas (que é o número de tarefas da aplicação menos 1) e a função inicial que será executada pelas *threads* após a criação. Antes de se chamar estas rotinas, atribui-se às variáveis globais `globalArgc` e `globalArgv` as variávies recebidas como parâmetro pela rotina `MPI_Init()`. Cada *thread* criada inicia a sua execução em `funcaoInicial()`. Nesta função, é chamada a função `main()`, passando-se como parâmetro os conteúdos das variáveis globais `globalArgc` e `globalArgv`. Desta forma, cada *thread* criada executa a função `main()` e, conseqüentemente, executa a função `MPI_Init()`, executando apenas as sentenças desta rotina comuns a todas as *threads*. A última sentença da `funcaoInicial()` chama a primitiva do Mulplix para finalização da thread: `thr_term()`.

Nas sentenças da rotina `MPI_Init()` que são executadas por todas as *threads*, cada *thread* se cadastra nas estruturas de dados do PMMPI e, em seguida, fica bloqueada em uma barreira até que todas as outras *threads* chegem nesta barreira, para garantir que o ambiente de execução do PMMPI está completamente inicializado.

Sinopse

```
int MPI_Finalize(void);
```

`MPI_Finalize()` libera a memória usada pelas estruturas de dados. Nenhuma outra rotina MPI pode ser chamada depois desta e o usuário é obrigado a garantir que toda a comunicação pendente envolvendo esta tarefa tenha se completado antes de chamar esta rotina de finalização.

4.5.3 Informações sobre o ambiente de execução

A rotina **`MPI_Comm_rank`** retorna, através do seu segundo argumento, o *rank* da tarefa que a chamou, no *communicator* passado como primeiro argumento.

No Quadro 13, é apresentado o algoritmo desta rotina.

Quadro 13: Algoritmo do MPI_Comm_rank( )

```
int MPI_Comm_rank(MPI_Comm comm, int *rank) {
      int grupo;

      grupo = obterGrupoNoCommVector(com);
      *rank = obterRankNoGroupVector(grupo);
}
```

A rotina **`MPI_Comm_size`** retorna, através do seu segundo argumento, o número de tarefas existentes no *communicator* passado como primeiro argumento.

No Quadro 14, é apresentado o algoritmo desta rotina.

Quadro 14: Algoritmo da rotina MPI_Comm_size( )

```
int MPI_Comm_size(MPI_Comm comm, int *tamanho) {
      int grupo;

      grupo = obterGrupoNoCommVector(com);
      *tamanho = obterTamanhoNoGroupVector(grupo);
}
```

4.5.4 Comunicação ponto-a-ponto

4.5.4.1 Comunicação bloqueante

Sinopse

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
               int receiverRank, int tag, MPI_Comm comm);
```

`MPI_Send` realiza um envio de mensagem bloqueante e no modo *standard*.

Os parâmetros desta rotina são: `buf`, o endereço inicial do *buffer de envio*; `count`, o número de elementos existentes no *buffer*; `datatype`, o tipo de dado de cada elemento do *buffer*; `receiverRank`, o *rank* da tarefa que recebe a mensagem; `tag`, um número, definido pelo usuário, usado para auxiliar a identificação de uma mensagem e `comm`, o *communicator* ao qual as tarefas emissora e receptora pertencem, ou seja, representa o domínio da comunicação.

No Quadro 15, é apresentado o algoritmo desta rotina.

Quadro 15: Algoritmo da rotina MPI_Send( )

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
                  int receiverRank, int tag, MPI_Comm comm){

      int descReceptor;       //descritor do receptor
      EntradaDaLista entrada; //entrada do RequestList
      MPI_Request nóEncontrado;
      Booleana encontrou;

      descReceptor = obterDescritorEmThreadVector(comm, receiverRank);

      lockRequestList(descReceptor);

      encontrou = procurarNaRequestListDoReceptor(&nóEncontrado);
      se (encontrou) {
            //copiar o dado para o buffer do receptor
            //sinalizar o receptor
      } senão {
            preencherNovoRequest(entrada);
            alocarBufferTemporario(entrada.data);
            copiarDadoParaBufferTemporario(entrada.data, buf);
            incluirNaRequestList(entrada);
      }

      unlockRequestList(descReceptor);
}
```

Sinopse

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
               int senderRank, int tag, MPI_Comm comm,
               MPI_Status *status);
```

`MPI_Recv` realiza uma recepção de mensagem bloqueante.

Os parâmetros desta rotina são: `buf`, o endereço inicial do *buffer de recepção*; `count`, o número máximo de elementos que o *buffer* pode comportar; `datatype`, o tipo de dado de cada elemento do *buffer*; `senderRank`, o *rank* da tarefa que envia a mensagem; `tag`, um número, definido pelo usuário, usado para auxiliar a identificação de uma mensagem; `comm`, o *communicator* ao qual as tarefas emissora e receptora pertencem, ou seja, representa o domínio da comunicação e `status`, que é um ponteiro para uma variável do tipo `MPI_Status`, onde são retornadas informações sobre a mensagem recebida.

No Quadro 16, é apresentado o algoritmo desta rotina.

Quadro 16: Algoritmo da rotina MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
            int senderRank, int tag, MPI_Comm comm,
            MPI_Status *status) {

      int descReceptor;       //descritor do receptor no ThreadVector
      EntradaDaLista entrada; //entrada do RequestList
      MPI_Request nóTemp;     //nó temporário
      MPI_Request nóEncontrado;
      Booleana encontrou;
      int tid = thr_id();

      descReceptor = obterDescritorEmThreadVector(tid);
      lockRequestList(descReceptor);
      encontrou = procurarNaRequestListDoReceptor(&nóEncontrado);
      se (encontrou) {
            //copiar o dado para o buffer do receptor
            preencherStatus(status);
            se (tipo do nó encontrado == SEND) {
                  removerBufferTemporario(nóEncontrado);
                  removerRequestDaLista(nóEncontrado);
            } senão se (tipo do nó encontrado == ISEND) {
                  //sinalizar o emissor
            }
      } senão {
            preencherNovoRequest(entrada); //entrada.data = buf;
            nóTemp = incluirNaRequestList(entrada);

            //esperar o buffer ficar disponível

            preencherStatus(status);
            removerRequestDaLista(nóEncontrado);
      }
      unlockRequestList(descReceptor);
}
```

A Figura 11 representa, em quatro etapas, a execução de um envio de uma mensagem usando a operação `MPI_Send()`, seguida da execução de uma recepção da mesma com a operação `MPI_Recv()`. Na etapa 1, na operação de envio, foi procurada uma requisição compatível na lista de requisições da tarefa receptora, mas não foi encontrada, pois a lista estava vazia. Na etapa 2, ainda na operação de envio, foi preenchida uma nova requisição; foi criado um *buffer* temporário; o dado da mensagem foi copiado para este *buffer* e a nova requisição foi inserida na lista de requisições da tarefa receptora. Na etapa 3, na operação de recepção, foi encontrada uma recepção compatível; o dado foi copiado do *buffer* temporário para o *buffer* de recepção e foi preenchida a variável `status`. Na etapa 4, foi liberado o *buffer* temporário e a requisição foi retirada da lista.

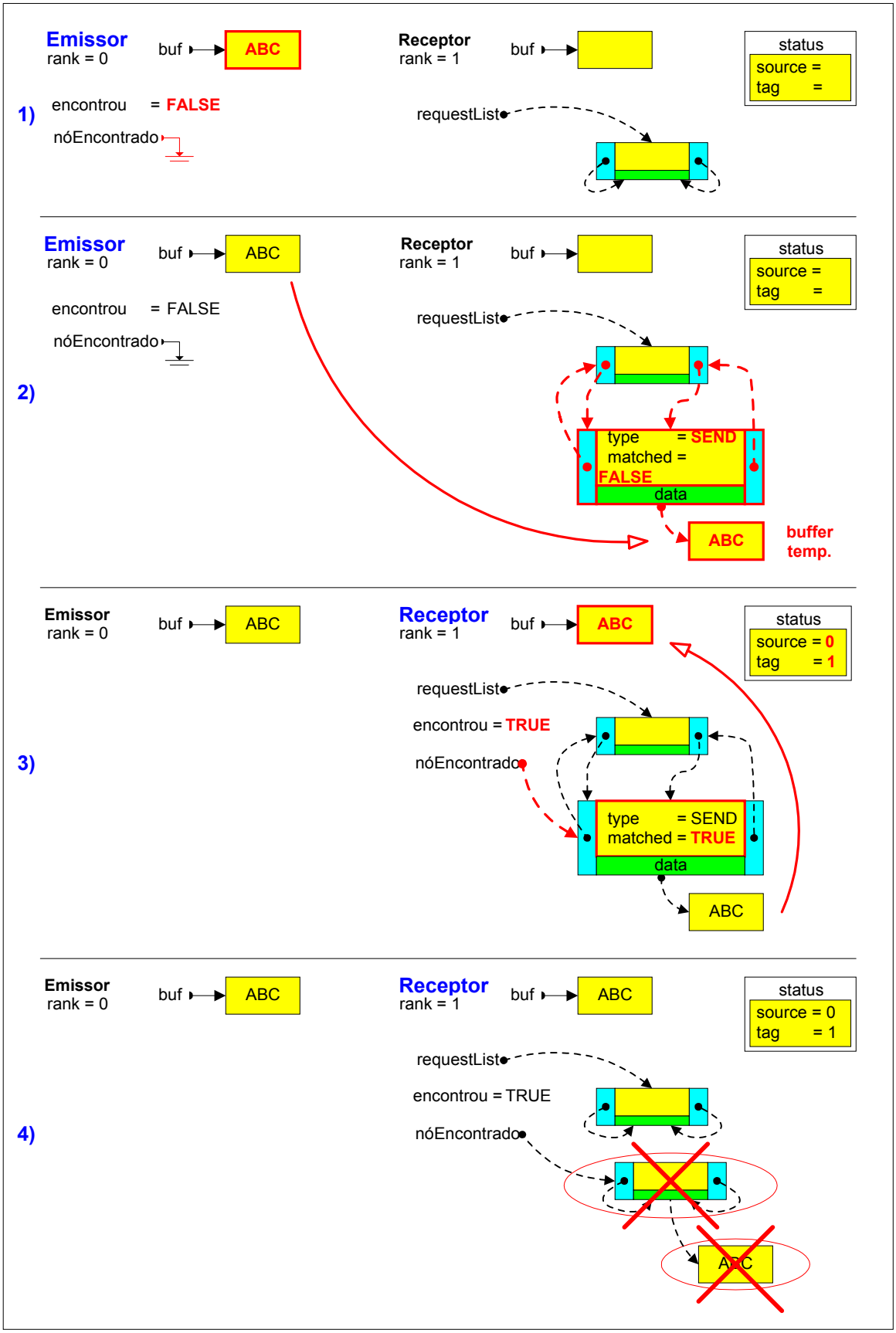

Figura 11: Exemplo da execução de um Send( ) seguido de um Recv( )

4.5.4.2 Comunicação não-bloqueante

Sinopse

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
            int receiverRank, int tag, MPI_Comm comm,
            MPI_Request *request);
```

`MPI_Isend` inicia um envio não-bloqueante e no modo *standard*. Os parâmetros desta rotina são os mesmos da rotina `MPI_Send`, acrescidos da variável `request`.

No Quadro 17, é apresentado o algoritmo desta rotina.

Quadro 17: Algoritmo da rotina Isend

```
int MPI_Isend(void *buf, const int count, MPI_Datatype datatype,
            int receiverRank, int tag, MPI_Comm comm,
            MPI_Request *request) {

      int descReceptor;       //descritor do receptor no ThreadVector
      EntradaDaLista entrada; //entrada do RequestList
      MPI_Request nóEncontrado;
      Booleana encontrou;

      descReceptor = obterDescritorEmThreadVector(comm, receiverRank);

      lockRequestList(descReceptor);

      encontrou = procurarNaRequestListDoReceptor(&nóEncontrado);
      se (encontrou) {
            //copiar o dado para o buffer do receptor
            //sinalizar o receptor
            se (tipo do nó encontrado == IRECV) {
                  *request = nóEncontrado;
            } senão {
                  *request = MPI_REQUEST_NULL;
            }
      } senão {
            preencherNovoRequest(entrada); //entrada->info.data = buf
            *request = incluirNaRequestList(entrada);
      }

      unlockRequestList(descReceptor);
}
```

Sinopse

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
            int senderRank,   int tag, MPI_Comm comm,
            MPI_Request *request);
```

`MPI_Irecv` inicia uma recepção não-bloqueante. Os parâmetros desta rotina são os mesmos da rotina `MPI_Recv` – porém sem o argumento `status` –, acrescidos da variável `request`.

No próximo quadro, é apresentado o algoritmo desta rotina.

Quadro 18: Algoritmo da rotina Irecv

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
            int senderRank, int tag, MPI_Comm comm,
            MPI_Request *request) {

      int descReceptor;       //descritor do receptor no ThreadVector
      EntradaDaLista entrada; //entrada do RequestList
      MPI_Request nóEncontrado;
      Booleana encontrou;
      int tid = thr_id();

      descReceptor = obterDescritorEmThreadVector(tid);

      lockRequestList(descReceptor);

      encontrou = procurarNaRequestListDoReceptor(&nóEncontrado);
      se (encontrou) {
            //copiar o dado para o buffer do receptor
            *request = nóEncontrado;

            se (tipo do nó encontrado == SEND) {
                  removerBufferTemporario(nóEncontrado);
            } senão se (tipo do nó encontrado == ISEND) {
                  //sinalizar o emissor
            }
      } senão {
            preencherNovoRequest(entrada); //entrada.data = buf;
            *request = incluirNaRequestList(entrada);
      }

      unlockRequestList(descReceptor);
}
```

Sinopse

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

`MPI_Wait` espera a conclusão da operação identificada por `request`. Após o retorno desta função, o objeto apontado por `request` foi desalocado e a ele é atribuído o valor `MPI_REQUEST_NULL`. O objeto `status` contém informações sobre a operação concluída.

No próximo quadro, é apresentado o algoritmo desta rotina.

Quadro 19: Algoritmo da rotina MPI_Wait

```
int MPI_Wait(MPI_Request *request, MPI_Status *status) {
      int descReceptor;       //descritor do receptor no ThreadVector

      se ( (*request) == MPI_REQUEST_NULL ) {
            return 0;
      }
      descReceptor = obterDescritorEmThreadVector(request);
      se ( tipo da requisição == SEND) {
            lockRequestList(descReceptor);
            preencherStatus(status);
            removerRequestDaLista(*request);
            unlockRequestList(descReceptor);
      } senão se ( (tipo da requisição == ISEND)
                        OU (tipo da requisição == IRECV) ) {

            lockRequestList(descReceptor);
            se ( a requisição ainda não foi "casada" ) {
                  unlockRequestList(descReceptor);
                  //esperar o buffer ficar disponível
                  lockRequestList(descReceptor);
            }
            preencherStatus(status);
            se ( (quem casou == ISEND)
                        OU (quem casou == IRECV) ) {

                  //se quem completou a comunicação foi uma operação
                  //bloqueante (ISEND ou IRECV) ...
                  se ( podeRemoverARequisição ) {
                        removerRequestDaLista(*request);
                  } senão {
                        podeRemoverARequisição = TRUE;
                  }
            } senão {
                  removerRequestDaLista((*request));
            }
            unlockRequestList(descReceptor);
      }
      (*request) = MPI_REQUEST_NULL;
}
```

Sinopse

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status);
```

`MPI_Test` verifica se a operação identificada por `request` foi concluída. Neste caso, o parâmetro `flag` possui o valor 1 (`true`), o objeto `status` contém informações sobre a operação concluída e o objeto apontado por `request` foi desalocado por esta operação e a ele é atribuído o valor `MPI_REQUEST_NULL`. Caso contrário, o `flag` retorna o valor 0 (`false`) e o valor de `status` é indefinido.

A implementação desta rotina é semelhante à da rotina `MPI_Wait`, porém se a requisição ainda não foi "casada" – ou seja, se a comunicação representada pela requisição ainda não se completou – a tarefa não é bloqueada e o parâmetro `flag` é retornado com o valor 0.

A figura Figura 12 representa algumas etapas de uma comunicação não-bloqueante, que foi dividida em 5 etapas: 1) uma operação `Isend()`; 2) uma operação `Wait()` feita pelo emissor, que é bloqueado; 3) uma operação `Irecv()`; 4) uma operação `Wait()` feita pelo receptor; 5) uma operação `Wait()` feita pelo emissor, após ter sido sinalizado.

Na etapa 1, o emissor procurou na lista de requisições da tarefa receptora uma requisição compatível, mas não a encontrou, pois a lista estava vazia. Em seguida, criou uma requisição, colocou-a na lista e retornou um ponteiro para a requisição criada. Na etapa 2, o emissor executou a operação `Wait()`, onde foi bloqueado na variável de sincronização `eventBufAva`, pois a requisição ainda não tinha sido "casada". Na etapa 3, o receptor encontrou uma requisição compatível, copiou a mensagem para o *buffer* de recepção, sinalizou o emissor – através da variável `eventBufAva` – e retornou um ponteiro para a requisição. Na etapa 4, o receptor executou a operação `Wait()`, que retornou a variável `status` preenchida. Na etapa 5, o emissor foi desbloqueado e removeu a requisição da lista.

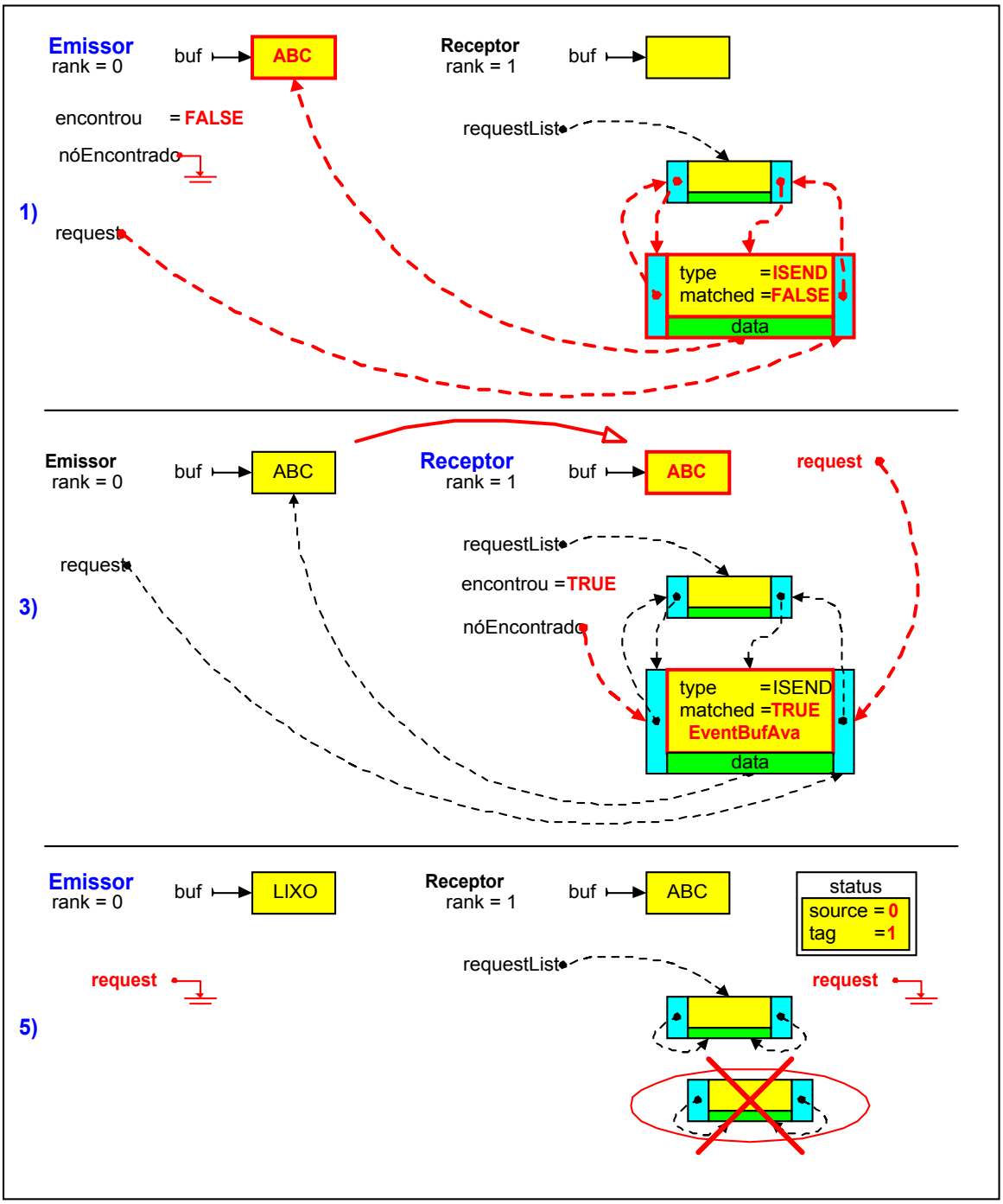

Figura 12: Exemplo da execução de um Isend seguida de um Irecv

4.5.5 Comunicação coletiva

4.5.5.1 Operação barreira

Sinopse

```
int MPI_Barrier(MPI_Comm comm)
```

`MPI_Barrier` bloqueia a tarefa chamadora até que todos os membros do grupo tenham executado esta operação. A implementação desta rotina obtém – através do *communicator* passado como parâmetro (`comm`) – o grupo de tarefas. A variável de sincronização `eventBarrier` deste grupo de tarefas é usada para sinalizar a chegada desta tarefa na barreira e aguardar a chegada das demais tarefas do grupo.

4.5.5.2 Operação *broadcast*

Sinopse

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

`MPI_Bcast` transmite a mensagem da tarefa *root* para todas as tarefas do grupo, que é especificado pelo argumento `comm`.

Os argumentos `count`, `datatype`, `root` e `comm` devem ter seus respectivos valores idênticos entre todas as tarefas do grupo. O argumento `buffer` é o endereço de um *buffer* alocado pelo usuário e, no caso da tarefa *root*, contém o dado a ser transmitido, enquanto para as demais tarefas, conterá o dado recebido após o retorno da rotina.

No próximo quadro, é apresentado o algoritmo desta rotina.

Quadro 20: Algoritmo da rotina MPI_Bcast

```
int MPI_Bcast(void* buf, int count, MPI_Datatype datatype,
                int root, MPI_Comm comm) {
      int myRank;
      int groupSize;
      BcastQueue *bcastQueue;

      obterGrupo(comm, &group);
      obterRank(group, &myRank);
      groupSize = group.groupSize;

      se (myRank == root) {
            //criar entrada da BcastQueue
            //criar buffer temporário
            //copiar dado para buffer temporário
            mx_lock( group.mutex );
            bcastQueue = &(group.bcastQueue);
            //incluir entrada na BcastQueue
            mx_free( group.mutex );

            //Esperar até que todas as threads atinjam a barreira
            barreira();
      } senão {
            //Esperar até que todas as threads atinjam a barreira
            barreira();
            mx_lock( group.mutex );
            bcastQueue = &(group.bcastQueue);
            encontrarBcastCompativel(bcastQueue);
            mx_free( group.mutex );

            //copiar o dado para o seu próprio buffer

            mx_lock( group.mutex );
            entradaEncontrada.numMatched++;
            entradaEncontrada.whoMatched[myRank] = TRUE;
            se ( entradaEncontrada.numMatched == groupSize ) {
                  //remover buffer temporário
                  //remover entrada
            }
            mx_free( group.mutex );
      }
}
```

4.5.5.3 Operação *gather*

Sinopse

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

`MPI_Gather` coleta as mensagens enviadas por todas as tarefas do grupo. Cada tarefa (inclusive a *root*) transmite o conteúdo do seu *buffer* de envio para a tarefa *root*. A

tarefa *root* recebe as mensagens e as armazena no *buffer* de recepção ordenadas pelo *rank* do emissor.

Os argumentos desta rotina estão descritos em 2.4.2.3. Operação *Gather*.
No próximo quadro, é apresentado o algoritmo desta rotina.

Quadro 21: Algoritmo da rotina MPI_Gather

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

      int myRank;
      int groupSize;
      GatherQueue *gatherQueue;

      obterGrupo(comm, &group);
      obterRank(group, &myRank);
      groupSize = group.groupSize;

      se (myRank == root) {
            //criar entrada da GatherQueue
            mx_lock( group.mutex );
            gatherQueue = &(group.gatherQueue);
            //incluir entrada na GatherQueue
            //copiar o dado para o recvBuf do root
            mx_free( group.mutex );

            //Esperar até que todas as threads atinjam a barreira
            barreira();
            //Aguardar a recepcao de todas as mensagens
            ev_wait();
      } senão {
            //Esperar até que todas as threads atinjam a barreira
            barreira();
            mx_lock( group.mutex );
            gatherQueue = &(group.gatherQueue);
            encontrarGatherCompatível(gatherQueue);
            //copiar o dado para o recvBuf do root

            entradaEncontrada.numMatched++;
            entradaEncontrada.whoMatched[myRank] = TRUE;
            se ( entradaEncontrada.numMatched == groupSize ) {
                  //remover entrada
                  //sinalizar o root
            }
            mx_free( group.mutex );
      }
}
```

4.5.5.4 Operação *scatter*

Sinopse

```
int MPI_Scatter(void* sendBuf, int sendCount,
      MPI_Datatype sendType,
      void* recvBuf, int recvCount, MPI_Datatype recvType,
      int root, MPI_Comm comm);
```

`MPI_Scatter` é a operação inversa à `MPI_Gather`. A tarefa *root* distribui o conteúdo do seu *buffer* de envio para as demais tarefas.

Os argumentos desta rotina estão descritos em 2.4.2.4. Operação *Scatter*.

No próximo quadro, é apresentado o algoritmo desta rotina.

Quadro 22: Algoritmo da rotina Scatter

```
int MPI_Scatter(void* sendBuf, int sendCount,
      MPI_Datatype sendType,
      void* recvBuf, int recvCount, MPI_Datatype recvType,
      int root, MPI_Comm comm) {

      int myRank;
      int groupSize;
      ScatterQueue *scatterQueue;
      obterGrupo(comm, &group);
      obterRank(group, &myRank);
      groupSize = group.groupSize;
      se (myRank == root) {
            //criar entrada da scatterQueue
            //criar buffer temporário
            //copiar dado para buffer temporário
            mx_lock( group.mutex );
            scatterQueue = &(group.scatterQueue);
            //incluir entrada na scatterQueue
            //copiar o dado para o recvBuf do root
            mx_free( group.mutex );
            //Esperar até que todas as threads atinjam a barreira
            barreira();
      } senão {
            //Esperar até que todas as threads atinjam a barreira
            barreira();
            mx_lock( group.mutex );
            scatterQueue = &(group.scatterQueue);
            encontrarScatterCompatível(scatterQueue);
            //copiar o dado para o seu próprio recvBuffer
            entradaEncontrada.numMatched++;
            entradaEncontrada.whoMatched[myRank] = TRUE;
```

```
            se ( entradaEncontrada.numMatched == groupSize ) {
                  //remover entrada
            }
            mx_free( group.mutex );
      }
}
```

4.5.5.5 Operação *reduce*

Sinopse

```
int MPI_Reduce(void* sendBuf, void* recvBuf, int count,
      MPI_Datatype datatype,
      MPI_Op op, int root, MPI_Comm comm);
```

`MPI_Reduce` combina os elementos do *buffer* de entrada de cada tarefa do grupo, usando a operação `op`, e retorna o valor combinado no *buffer* de saída da tarefa *root*.

Os argumentos desta rotina estão descritos em 2.4.2.5. Operação de redução.

No próximo quadro, é apresentado o algoritmo desta rotina.

Quadro 23: Algoritmo da rotina MPI_Reduce

```
int MPI_Reduce(void* sendBuf, void* recvBuf, int count,
            MPI_Datatype datatype,
            MPI_Op op, int root, MPI_Comm comm) {
      int myRank;
      int groupSize;
      ReduceQueue *reduceQueue;

      obterGrupo(comm, &group);
      obterRank(group, &myRank);
      groupSize = group.groupSize;

      se (myRank == root) {
            //criar entrada da ReduceQueue
            mx_lock( group.mutex );
            reduceQueue = &(group.reduceQueue);
            //incluir entrada na ReduceQueue
            //copiar o sendBuf para o recvBuf
            mx_free( group.mutex );

            //Esperar até que todas as threads atinjam a barreira
            barreira();
            //Aguardar a recepção de todas as mensagens
            ev_wait();
      } else {
            //Esperar até que todas as threads atinjam a barreira
            barreira();
            mx_lock( group.mutex );
            reduceQueue = &(group.reduceQueue);
            encontrarReduceCompativel(reduceQueue);
            //executar a operação
            entradaEncontrada.numMatched++;
            entradaEncontrada.whoMatched[myRank] = TRUE;
            se ( entradaEncontrada.numMatched == groupSize ) {
                  //remover entrada
                  //sinalizar o root
            }
            mx_free( group.mutex );
      }
}
```

5 Testes e avaliação de desempenho

Este capítulo especifica o ambiente operacional onde foi desenvolvido o PMMPI e onde foram realizados os testes. Em seguida, é feita a avaliação dos resultados dos testes com as operações básicas da biblioteca PMMPI. Para tanto, foram realizados testes com as primitivas do Mulplix – comparando-se as implementações do Mulplix com *lightweigth process* e *Solaris Threads* – e da biblioteca PMMPI – comparando-a com a implementação do MPI desenvolvida pela Sun Microsystems, Inc. Finalmente, são apresentados os resultados de testes de aplicações que utilizam o MPI para se comparar o desempenho de ambas as implementações.

5.1 Ambiente Operacional

O ambiente onde foi desenvolvido o PMMPI e onde foram realizados os testes consiste de:

- Uma estação de trabalho SMP: Sun Enterprise 4500, com 6 processadores (UltraSPARC 400 MHz) e 1.5 Gbytes de memória (SUN ENTERPRISE);
- O sistema operacional Solaris 8 (MAURO, 2001);
- O compilador GNU C (STALLMAN, 1996), com todas as opções de otimização de código habilitadas e
- O Sun MPI 4.0 (SUN MPI, 1999), que faz parte do Sun HPC ClusterTools 3.0 e que utiliza o Sun Cluster Runtime Environment (CRE) 1.0 para o gerenciamento da execução dos programas (SUN HPC, 1999).

A unidade dos valores apresentados é o segundo e os mesmos referem-se aos menores e maiores tempos de execução obtidos em 10 execuções dos programas. Este número de execuções é satisfatório, pois a diferença entre o menor e o maior tempo de execução não aumentou significativamente para mais de 10 execuções.

5.2 Operações básicas do Mulplix

Estes testes têm como objetivo comparar os desempenhos de duas implementações da biblioteca Mulplix: uma baseada em *lightweight process* (LWP) e outra em *Solaris Threads* (THR). Os resultados destes testes são úteis para a avaliação do desempenho do PMMPI, uma vez que esta implementação do MPI utiliza estes dois modelos de *threads*.

Dentre as primitivas de programação paralela do Mulplix, foram escolhidas as que mais foram usadas na implementação do PMMPI.

As tabelas desta secção apresentam os tempos de execução dos testes. A primeira coluna indica o número de iterações da execução. As próximas quatro colunas indicam os menores e maiores tempos das duas implementações do Mulplix. A penúltima coluna indica a percentagem da diferença entre os menores tempos das duas implementações em relação ao maior entre os menores tempos, ou seja, é aplicada a seguinte fórmula: |<LWP - < THR | / máx(<LWP, < THR) * 100. Na última coluna é apresentado o resultado da seguinte fórmula: | > LWP - > THR | / máx(> LWP, > THR) * 100.

5.2.1 Custo da sincronização com exclusão mútua

Este teste mede o tempo de execução de operações de *lock* (`mx_lock`) e *unlock* (`mx_free`) realizadas em uma variável de exclusão mútua do tipo *mutex*. Estas operações foram realizadas por uma única *thread* sem atrasos devido à concorrência. O principal trecho do programa testado é apresentado no próximo quadro.

Quadro 24: Trecho do teste do mutex

```
...
mutex = mx_create(1);
for (i = 0; i < NumIteracoes;  i++) {
      mx_lock(mutex);
      mx_free(mutex);
}
mx_delete(mutex);
...
```

Na Tabela 7, são apresentados os tempos de execução deste teste. O Gráfico 1 também representa o resultado deste teste.

Tabela 7: Teste do custo da sincronização com exclusão mútua

No. de Itera.	Tempo (menor, maior) por biblioteca (em segundos)				% dif. entre LWP e THR	
	< LWP	< THR	> LWP	> THR	<	>
100K	0.022362	0.022386	0.025571	0.025575	0.11	0.02
300K	0.067966	0.066514	0.074988	0.073367	2.14	2.16
600K	0.131317	0.130997	0.152036	0.157007	0.24	3.17
900K	0.201021	0.197897	0.229986	0.225947	1.55	1.76

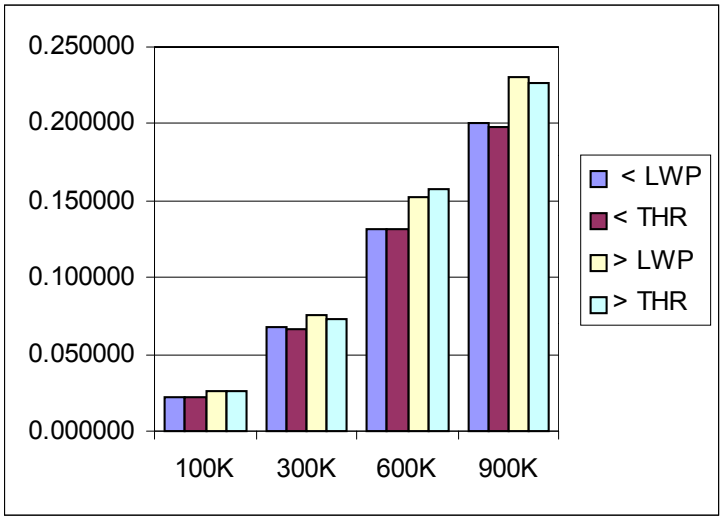

Gráfico 1: Custo da sincronização com exclusão mútua

Para cada número de iterações apresentado, a diferença entre os menores tempos das implementações do Mulplix não é significativa, representando menos de 2.15% do maior valor. O mesmo ocorre com a diferença entre os maiores tempos de execução, que representam menos de 3.18% do maior valor. Portanto, o custo da sincronização por exclusão mútua nas duas implementações é praticamente o mesmo.

5.2.2 Custo da identificação da *thread*

Este teste mede o tempo gasto para se obter a identificação da *thread* utilizando a rotina `thr_id()`. Para tanto, foi criada apenas uma *thread* sem atrasos devido à concorrência. O principal trecho do programa testado é apresentado no próximo quadro.

Quadro 25: Trecho do teste da identificação da *thread*

```
...
for (i = 0; i < numIteracoes;  i++) {
      id = thr_id();
}
...
```

A Tabela 8 e o Gráfico 2 apresentam o resultado deste teste.

Tabela 8: Custo da identificação da *thread*

No. De Itera.	Tempo (menor, maior) por biblioteca				% dif. entre LWP E THR	
	< LWP	< THR	> LWP	> THR	<	>
30K	0.038506	0.001958	0.039730	0.002559	94.92	93.56
60K	0.077890	0.003909	0.079000	0.004921	94.98	93.77
90K	0.116028	0.005409	0.119088	0.007330	95.34	93.84

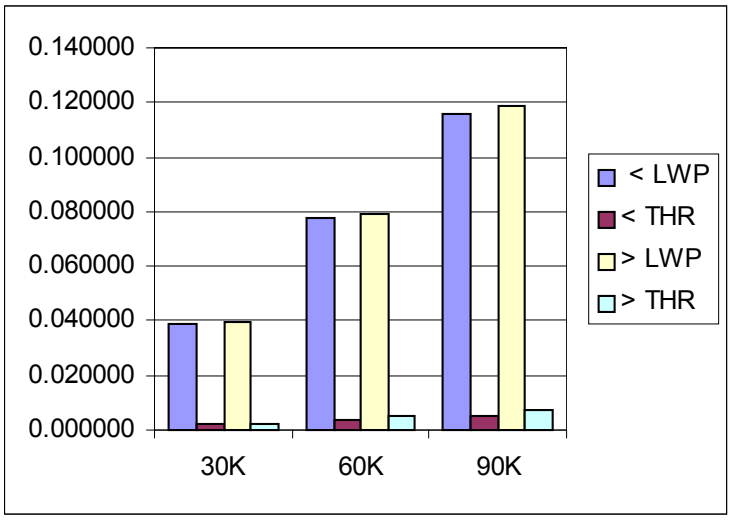

Gráfico 2: Custo da identificação da *thread*

Em todos os testes, os tempos da rotina `thr_id` implementada com *Solaris Threads* (THR) foram muito menores do que com a mesma rotina implementada com

lightweight process (LWP), sendo que a diferença entre estes tempos é acima de 93% do tempo de execução da LWP.

O alto custo do Mulplix com LWP deve-se ao fato de que a rotina `thr_id` é implementada fazendo a chamada ao sistema `_lwp_self()`, enquanto que, com o *Solaris Threads*, esta chamada é substituída pela `_thr_self()`, que faz parte da biblioteca de *threads* e executa no modo usuário.

Quando uma *thread* faz uma chamada ao sistema, a mesma faz uma transição do modo usuário – ou modo não privilegiado – para o modo de *kernel* – ou modo privilegiado. A chamada ao sistema `_lwp_self()` executa instruções de máquina especiais para mudar o processador para o modo privilegiado para que se tenha acesso às estruturas de dados do *kernel* e se obtenha a identificação da *thread*. Antes de retornar, entretanto, esta chamada ao sistema faz a transição de volta ao modo não privilegiado (MAURO, 2001, p.27). Este processo de transição não é necessário para a rotina `_thr_self()`, pois a identificação da *thread* está em uma estrutura de dados no modo usuário. Como, neste caso, o *ovehead* da troca de modo não é necessário, esta rotina é executada em menos tempo do que aquela.

5.2.3 Custo da alocação de memória

As rotinas de alocação de memória das primitivas de programação paralela do Mulplix sincronizam as respectivas chamadas às rotinas da biblioteca `malloc.h`, pois estas estão classificadas no manual de referência como *Safe* e, se uma entrada não for classificada como *MT-Safe*, a mesma deve ser considerada *unsafe*, conforme observado em (MULTITHREADED, 2001, p.232).

Quadro 26: Trecho do teste do custo da alocação de memória

```
...
for (i = 0; i < ite;  i++) {
      if ( ( p = (int *) me_palloc(sizeof(int)) ) == NULL ) {
            printf("sem espaco!\n");
            exit(1);
      }
}
...
```

Cada execução da `me_palloc` aloca um bloco de 4 bytes suficiente para armazenar um número inteiro.

A Tabela 9 apresenta os resultados comparativos entre as versões da me_palloc.

Tabela 9: Custo da alocação de memória

No. de Itera.	Tempo (menor, maior) por biblioteca				% dif. entre LWP e THR	
	< LWP	< THR	> LWP	> THR	<	>
100K	0.092362	0.098821	0.093806	0.099611	6.54	5.83
400K	0.370540	0.396381	0.374492	0.397584	6.52	5.81
800K	0.744337	0.794835	0.749209	0.796532	6.35	5.94
1.2M	1.120466	1.194524	1.124721	1.212579	6.20	7.25

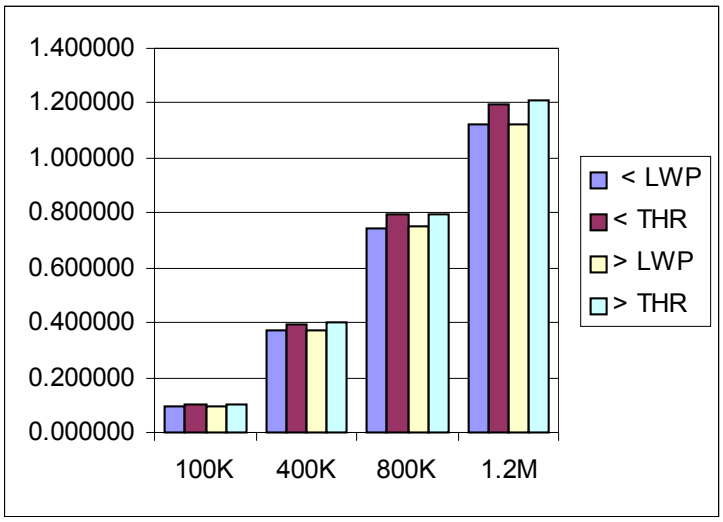

Gráfico 3 : Custo da alocação de memória

As diferenças entre os maiores e menores valores de ambas as implementações foram pequenas (menores do que 1.55% do maior valor). Entretanto, a alocação de memória implementada com LWP é um pouco mais rápida do que com THR, apresentando uma diferença em relação ao tempo da THR inferior a 6.55%. Com exceção da execução com 1.2M de iterações, onde esta percentagem aumentou um pouco para 7.25%.

5.3 Operações básicas do PMMPI em relação ao SUN MPI

O objetivo principal dos próximos testes é comparar e avaliar o desempenho da execução da nossa implementação do MPI baseada em *threads* com outra implementação baseada em processos.

Não foram feitos testes com outra implementação do MPI baseada em *threads*, pois a única encontrada, o TMPI (TANG, 2000), foi desenvolvida para ser executada em outro sistema operacional.

Para avaliar o desempenho da biblioteca PMMPI, são comparados os tempos de execução de operações desta implementação do MPI – usando LWP (MPI-LWP) e *Solaris Threads* (MPI-THR) – e a implementação que faz parte do Sun HPC *(High Performance Computing*) ClusterTools 3.0 (SUN MPI).

O ambiente de execução CRE (*Cluster Runtime Environment*) – usado pelo software SUN MPI – foi configurado de modo a maximizar o desempenho dos programas que utilizam a biblioteca SUN MPI quando apenas um programa é executado de cada vez (SUN HPC, 1999, p.32). Esta configuração foi feita alterando-se o arquivo `hpc.conf` de modo a utilizar o modelo *Performance Template*.

Além desta configuração, foram realizados testes com algumas variáveis de ambiente devidamente alteradas conforme é orientado em (SUN MPI, 1999, p.54), com o intuito de tentar melhorar ainda mais o desempenho em um sistema dedicado, ou seja, quando o número de tarefas MPI é menor ou igual ao número de processadores. Neste caso, o nome da biblioteca virá seguido de um asterisco: SUN MPI*. As variáveis de ambiente alteradas são:

- `MPI_SPIN`: foi atribuído o valor 1, para se obter o melhor desempenho quando o número de tarefas for menor ou igual ao número de CPUs;
- `MPI_PROCBIND`: foi alterado com o valor 1, para ligar cada tarefa MPI ao seu próprio processador, permitindo que cada processador fique dedicado a apenas uma tarefa MPI;

- `MPI_POLLALL`: foi atribuído o valor 0, suprimindo o mecanismo de *polling*, que acarreta um *overhead* desnecessário para programas confiáveis. O *polling* deve ser usado apenas para códigos MPI não confiáveis de modo a tentar reduzir a chance de ocorrer *deadlock*.

Estes testes têm como objetivo avaliar e comparar o desempenho de diferentes implementações do MPI na execução das principais operações deste padrão.

5.3.1 Criação do ambiente de execução

O objetivo deste teste é medir o custo da inicialização do ambiente MPI. Para tanto, foi medido o tempo de execução da rotina `MPI_Init()` de cada tarefa e obtido o maior tempo gasto entre elas através da rotina `MPI_Reduce()`. O quadro que se segue mostra o código do programa testado.

Quadro 27: Custo da rotina MPI_Init

```
...
time = getTime();
MPI_Init(&argc, &argv);
time = getTime() - time;

MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
*sendBuf = time;
MPI_Reduce(sendBuf, recvBuf, 1, MPI_DOUBLE, MPI_MAX,
                  1, MPI_COMM_WORLD);
if (myRank == 1) {
      printf("time = %f\n", *recvBuf);
}
...
```

A Tabela 10 apresenta, na primeira coluna, o número de tarefas criadas, e nas demais, os tempos de execução do teste nas diferentes implementações do MPI.

Tabela 10: Custo da inicialização do ambiente MPI

No. de threads	Tempo de execução (em segundos)							
	< MPI-LWP	< MPI-THR	< SUN MPI	< SUN MPI*	> MPI-LWP	> MPI-THR	> SUN MPI	> SUN MPI*
2	0.002632	0.002576	0.055397	0.058021	0.002874	0.002809	0.124555	0.169773
4	0.003117	0.003304	0.091760	0.093494	0.003405	0.004098	0.224155	0.107497
6	0.003768	0.003987	0.194373	0.143675	0.004042	0.005073	0.306544	0.183545

O tempo de inicialização do ambiente usando a biblioteca PMMPI é muito inferior pois a criação de *threads* é mais rápida do que a de processos. Além disso, o PMMPI não cria nenhum *pool* de recursos antecipadamente, ao passo que o SUN MPI cria *pools* de recursos para serem usados tanto na comunicação ponto-a-ponto quanto para comunicação coletiva.

5.3.2 Envio e recepção de mensagem

Este é um teste clássico de envio e recepção de mensagens conhecido como Ping Pong. Para a sua execução, são criadas duas tarefas: uma envia uma mensagem e a outra a recebe e a envia de volta. Este envio e recepção de mensagens é repetido 1000 vezes.

Quadro 28: Trecho do teste Ping Pong

```
...
task0 = 0;
task1 = 1;
for(count = 0; count < itera; count++) {
      if(myRank == 0) {
            MPI_Send(message, length, MPI_INT, task1, count,
                  MPI_COMM_WORLD);
            MPI_Recv(message, length, MPI_INT, task1, count,
                  MPI_COMM_WORLD, &status);
      } else {
            MPI_Recv(message, length, MPI_INT, task0, count,
                  MPI_COMM_WORLD, &status);
            MPI_Send(message, length, MPI_INT, task0, count,
                  MPI_COMM_WORLD);
      }
}
...
```

A próxima tabela apresenta na primeira coluna o tamanho das mensagens, em bytes, e nas demais os tempos de execução do teste nas diferentes implementações do MPI.

Tabela 11: Teste Ping Pong

Tam.da Mensa.	Tempo de execução (em segundos)							
	< MPI-LWP	< MPI-THR	< SUN MPI	< SUN MPI*	> MPI-LWP	> MPI-THR	> SUN MPI	> SUN MPI*
100	0.15604	0.18630	0.02259	0.01555	0.22224	0.20912	0.02294	0.01580
300	0.15564	0.18527	0.03552	0.02695	0.22307	0.22920	0.03688	0.02775
600	0.15746	0.19794	0.04163	0.03626	0.24344	0.20553	0.04517	0.03669
900	0.15975	0.20742	0.06088	0.05436	0.25072	0.21748	0.06396	0.05505
10K	0.39235	0.46518	0.36791	0.35641	0.49191	0.48620	0.39211	0.36669
30K	**0.99010**	1.05150	**0.99590**	0.98124	1.10837	1.07198	1.08225	0.99597
60K	**1.78997**	1.94310	1.92466	**1.90282**	2.03422	1.98219	2.11155	1.96139
90K	2.70831	**2.81581**	**2.98668**	**2.83606**	2.88415	2.84165	3.12937	2.90277
100K	2.84097	3.12142	3.18396	3.14083	3.20567	3.16773	3.49391	3.22999
300K	8.35521	8.90866	9.86462	9.44014	8.80745	8.97629	10.42842	9.54227
600k	16.95375	18.57692	19.65956	18.87867	18.97681	18.72880	21.16912	19.12101
900K	27.59565	26.38335	29.17920	28.30986	28.63115	26.58835	31.40610	28.64205

O próximo gráfico ilustra a parte hachurada da tabela anterior.

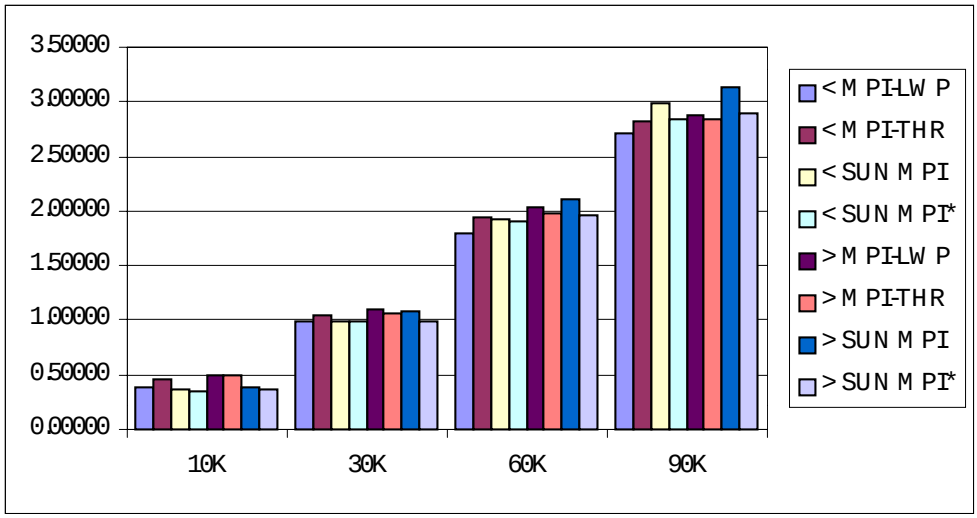

Gráfico 4: Teste Ping Pong

Deve-se ressaltar que, para todos os tamanhos de mensagens, o SUN MPI* é sempre mais rápido do que o SUN MPI. Isso acontece, pois estes testes foram feitos em um sistema dedicado – apenas duas tarefas MPI foram criadas – e o SUN MPI* é configurado para aproveitar ao máximo a execução neste ambiente.

Nos testes cujas mensagens são menores do que 30 Kbytes, o SUN MPI e o SUN MPI* apresentaram os menores tempos de execução. Para mensagens de tamanho 30 Kbytes, o MPI-LWP passa a ser mais rápido do que o SUN MPI, mas ainda não é mais rápido

do que o SUN MPI*. Entretanto, para mensagens acima de 60 Kbytes, o MPI-LWP torna-se mais rápido do que o SUN MPI*.

O MPI-THR passa a ser mais rápido do que o SUN MPI e do que o SUN MPI* para mensagens acima de 90 Kbytes. À partir deste tamanho de mensagens, o MPI-LWP e o MPI-THR apresentam os menores tempos de execução.

A implementação nativa do MPI é mais rápida para mensagens menores do que 30 Kbytes pois, faz uso de algum mecanismo para que os processos compartilhem *buffer* para troca de mensagens ponto-a-ponto cujo tamanho é fixado em 1 Kbyte. Além disso, possui um *pool* de *buffers* pré-alocados (SUM MPI, 1999, p.55). Tais mecanismos não foram explorados pela versão atual do PMMPI. Entretanto, para mensagens maiores do que 60 Kbytes, o MPI nativo torna-se mais lento pois o *buffer* utilizado não comporta toda a mensagem, sendo necessário se fazer mais de uma cópia para o mesmo. Por outro lado, as tarefas do PMMPI compartilham o espaço de endereçamento do processo e a mensagem é copiada apenas uma vez do *buffer* do emissor direto para o do receptor ou, no pior caso, é copiada inteiramente para um *buffer* temporário antes de ser copiada para o *buffer* do receptor.

5.3.3 Operação de *Broadcast*

Quadro 29: Trecho do teste da rotina MPI_Bcast( )

```
...
time = getTime();
for (i=0; i < itera; i++) {
      MPI_Bcast(buf1, length, MPI_INT, ROOTRANK, MPI_COMM_WORLD);
}
MPI_Barrier(MPI_COMM_WORLD);
time = getTime() - time;
if (myRank == 0) {
      printf("time = %f\n", time);
}
...
```

Para este teste, são criadas 6 tarefas MPI e cada uma executa 3.000 vezes a operação `MPI_Bcast()` cujo *root* é a tarefa 0. Esta execução é repetida para diversos tamanhos da mensagem transmitida. Na próxima tabela, é apresentado o resultado deste teste. Na

primeira coluna estão os diferentes tamanhos da mensagem e nas colunas seguintes, os tempos de execução, em segundos, das diferentes implementações do MPI.

Tabela 12: Teste da rotina MPI_Bcast( )

Tam.da Mens.	Tempo de execução (em segundos)							
	< MPI-LWP	< MPI-THR	< SUN MPI	< SUN MPI*	> MPI-LWP	> MPI-THR	> SUN MPI	> SUN MPI*
1K	0.8934	1.6441	0.1592	0.1561	0.9142	1.6674	0.1729	0.1661
3K	1.1608	1.7397	0.3615	0.3496	1.2029	1.7557	0.4077	0.3770
6K	1.3975	1.9756	0.6384	0.6192	1.4935	2.1291	0.6622	0.6589
9K	1.6172	2.3802	0.9668	0.9196	1.8021	2.4379	1.0878	1.0964

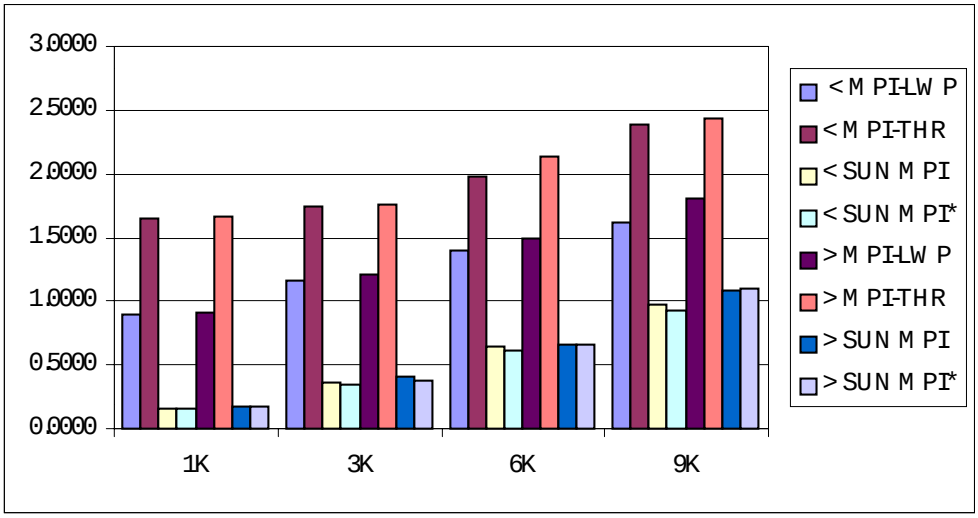

Gráfico 5: Teste da rotina MPI_Bcast( )

A implementação nativa da rotina `MPI_Bcast()` é sempre mais rápida, pois suas operações coletivas são muito otimizadas e fazem uso de um *pool* de *buffers* genérico dentro da memória compartilhada (SUN MPI, 1999, p.57). Por outro lado, a versão atual do PMMPI não utiliza tais recursos.

5.3.4 Operação de redução

Quadro 30: Trecho do teste da rotina MPI_Reduce( )

```
...
time = getTime();
for (i = 0; i < itera; i++) {
      MPI_Reduce(sendBuf, recvBuf, length, MPI_DOUBLE_INT, MPI_MAXLOC,
                  ROOTRANK, MPI_COMM_WORLD);
}
MPI_Barrier(MPI_COMM_WORLD);
time = getTime() - time;
if (myRank == ROOTRANK) {
      sum1 = 0.0;
      sum2 = 0;
      for (i = 0; i < length; i++) {
            sum1 += recvBuf[i].val;
            sum2 += recvBuf[i].rank;
      }
      printf("sum1 = %f - sum2 = %d \n", sum1, sum2);
      printf("time = %f\n", time);
}
...
```

Para testar o desempenho da operação de redução, são criadas 6 tarefas MPI e cada uma executa 3000 vezes a operação `MPI_Reduce()` cujo *root* é a tarefa 0. Esta execução é repetida para diversos tamanhos da mensagem transmitida, que estão representados na primeira coluna da próxima tabela. Nas colunas seguintes, estão os tempos de execução, em segundos, das diferentes implementações do MPI.

Tabela 13: Teste da rotina MPI_Reduce( )

Tam.da	Tempo (menor, maior) (em segundos)							
Mens.	< MPI-LWP	< MPI-THR	< MPI-HPC	< MPI-HPC*	> MPI-LWP	> MPI-THR	> MPI-HPC	> MPI-HPC*
100	1.1311	1.6856	0.4145	0.3792	1.1519	1.6883	0.4157	0.3793
300	1.7050	2.1328	1.0897	1.0725	1.7335	2.1661	1.1686	1.0787
600	2.4024	2.9555	2.1646	2.1574	2.4815	3.0511	2.5091	2.1914
900	**3.1367**	3.8343	**3.2269**	**3.2396**	3.1820	3.9202	3.3116	3.2840
1K	3.3197	4.1000	3.5376	3.5430	3.3254	4.1410	3.5565	3.6087
3K	7.7835	**8.7644**	**10.4947**	**10.2686**	7.8738	8.8141	10.9442	10.3373
6K	14.5881	15.7716	21.2015	21.3512	14.8253	15.8037	21.3699	21.6565
9K	21.7159	22.6203	31.9622	31.8451	21.7241	23.4497	32.2977	32.1059
10K	23.9095	24.8298	35.4425	35.5167	23.9272	26.2600	36.1833	36.0195
30K	68.8666	70.6714	108.2377	108.2506	69.0507	70.8578	109.4619	109.7856
60K	136.9121	140.0847	2989.9580	218.5055	137.0863	151.6118	3110.5068	220.0558
90K	217.6148	219.9742	**Abortado**	327.3259	218.8453	237.3018	**Abortado**	327.8704
100K	255.2519	244.8130	**Abortado**	**Abortado**	262.6078	247.8475	**Abortado**	**Abortado**
300K	830.8090	851.9374	**Abortado**	**Abortado**	847.8316	856.9071	**Abortado**	**Abortado**
600K	1752.6354	1778.1835	**Abortado**	**Abortado**	1785.6730	1778.1835	**Abortado**	**Abortado**
900K	2556.3487	2380.6921	**Abortado**	**Abortado**	2561.2099	2699.6803	**Abortado**	**Abortado**

O próximo gráfico representa a área hachurada da tabela anterior.

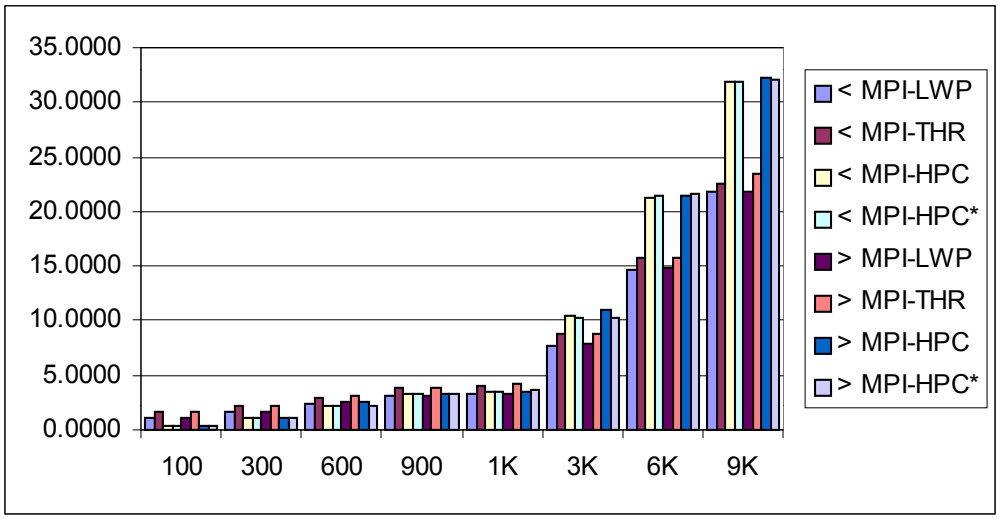

Gráfico 6: Teste da rotina MPI_Reduce( )

Nos testes cujas mensagens são menores do que 900 bytes, o SUN MPI e o SUN MPI* apresentaram os menores tempos de execução. Para mensagens de tamanho maior ou igual a 900 bytes, o MPI-LWP passa a ser mais rápido do que a implementação nativa. O mesmo acontece com o MPI-THR para mensagens maiores do que 3 Kbytes.

Conforme observado na secção anterior, as operações coletivas da implementação nativa são muito otimizadas, o que explica porque, para mensagens até 900 bytes, esta biblioteca é mais rápida. Entretanto, como a operação de redução não envolve somente troca de mensagens, mas, também envolve a execução de uma operação, para mensagens grandes pode haver preempção de uma tarefa e, neste caso, a troca de contexto é mais rápida para a implementação do MPI baseada em *threads*, ou seja, para o PMMPI.

Para mensagens de tamanho maior ou igual a 90 Kbytes, o SUN MPI abortou a execução por falta de recursos. O mesmo aconteceu com o SUN MPI* para mensagens cujo tamanho é maior ou igual a 100 Kbytes. Por outro lado, o PMMPI executou sem problemas com as mensagens testadas. Não foram feitos testes com mensagens acima de 900 Kbytes.

5.4 Testes com Aplicações

Os próximos testes são feitos com aplicações computacionais que utilizam o MPI. Eles representam duas classes bem distintas de aplicações paralelas. O primeiro teste – 5.4.1.Cálculo de π utilizando integração numérica – representa a classe de aplicações paralelas que podem ser divididas em subtarefas que são pouco acopladas, pois realizam pouca comunicação entre si. Por outro lado, o segundo teste – 5.4.2.Eliminação Gaussiana – representa as aplicações nas quais as suas sub-tarefas realizam muita comunicação entre si. Estas aplicações são simples, mas reveladoras.

O objetivo destes testes é comparar e avaliar o desempenho das implementações do MPI na execução de aplicações reais.

5.4.1 Cálculo de π utilizando integração numérica

Conforme salientado por (GROPP, 1996, p.21), há uma classe de aplicações cuja paralelização pode ser obtida de forma ideal, ou seja, as tarefas que irão executar coordenadamente esta aplicação possuem um mínimo de comunicação, o balanceamento de carga é automático e é possível se verificar o resultado. Uma destas aplicações calcula o valor de π por integração numérica. Uma vez que

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x)\,|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4}$$

para se obter o valor de π, basta integrar a função $f(x) = 4/(1+x^2)$. Para se integrar esta função numericamente, divide-se o intervalo de 0 até 1 em um número n de sub-intervalos e somam-se as áreas dos retângulos. Quanto maior o valor de n, mais precisa será a aproximação de π.

No quadro que se segue, está listada a parte principal da aplicação.

Quadro 31: Cálculo de pi usando integração numérica

```
...
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numTarefas);
MPI_Comm_rank(MPI_COMM_WORLD, &meuRank);
h = 1.0 / (double) n;
soma = 0.0;
for (i=meuRank + 1; i <= n; i += numTarefas) {
      x = h * ((double) i - 0.5);
      soma += (4.0 / (1.0 + x*x));
}
meuPi = h * soma;
MPI_Reduce(&meuPi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
...
```

Neste algoritmo, cada tarefa calcula a soma de um subconjunto de retângulos diferente. Ao final da computação, todas as somas locais são combinadas em uma soma global – através da operação `MPI_Reduce` – representando o valor de π. O número de iterações foi de 2.100.000.000, de modo a se obter uma aproximação muito precisa.

Tabela 14: Cálculo de pi usando integração numérica

No. de Tarefas	Tempo de execução (em segundos)							
	< MPI-LWP	< MPI-THR	< SUN MPI	< SUN MPI*	> MPI-LWP	> MPI-THR	> SUN MPI	> SUN MPI*
2	124.6799	124.7287	175.2210	175.1701	124.8459	124.8019	175.6407	175.4332
4	62.4133	62.3829	87.7425	87.6563	62.5543	62.4439	87.9259	88.0453
6	41.8039	41.8445	58.8784	58.9263	41.9316	42.1414	59.5354	59.5297
12	42.1676	41.8287	60.9141	61.7396	43.3575	45.8913	65.9481	65.9463
24	41.8910	41.8091	67.0689	66.3945	42.4455	46.3716	70.9953	70.7477
36	41.8817	41.8098	67.1195	65.9729	42.2182	46.4112	69.6331	71.5885
48	41.8210	41.8223	68.5370	68.6387	42.2415	42.2603	74.5196	72.1078
60	41.8298	41.8495	68.9116	68.4646	41.9931	43.2859	75.0751	73.1106
96	41.8040	41.8590	75.0030	74.1087	41.9323	42.6085	79.5981	79.7565
120	41.8029	41.8592	79.3692	79.3583	41.9976	42.7177	88.1087	88.2079

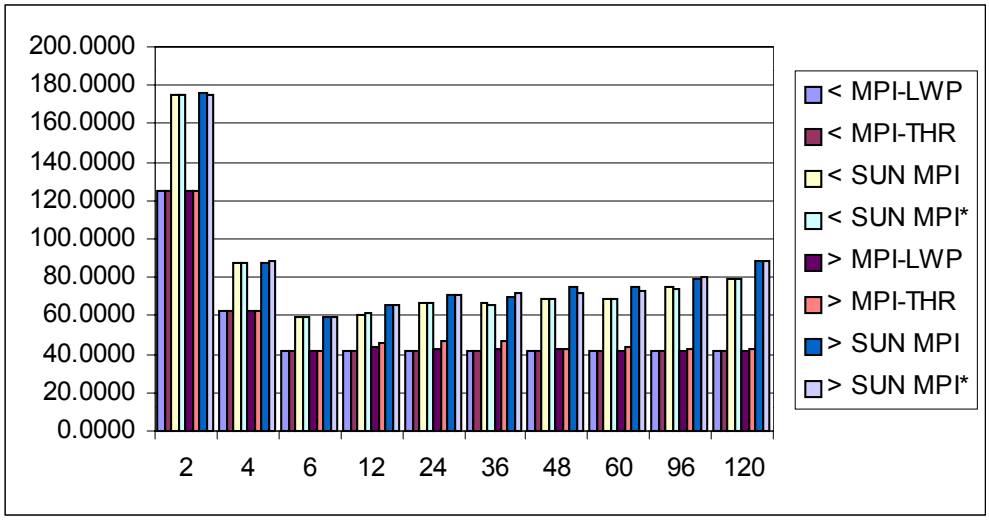

Gráfico 7: Cálculo de pi usando integração numérica

Neste teste, o PMMPI foi mais rápido do que o SUN MPI em todos os casos, ou seja, para qualquer número de tarefas.

É interessante notar que, a partir de 6 tarefas, o tempo de execução com o PMMPI se estabiliza, ou seja, não se consegue melhorar o desempenho aumentando o número de tarefas, pois existem apenas 6 processadores. Por outro lado, para mais de 6 tarefas o tempo de execução do SUN MPI aumenta à medida em que o número de tarefas aumenta. Isso ocorre, pois com o aumento do número de tarefas há mais troca de contexto entre as mesmas, o que penaliza o tempo de execução da implementação baseada em processos, enquanto que a troca de contexto entre *threads* irmãs é rápida pois elas compartilham o espaço de endereçamento.

Entretanto, para a execução do teste com até 6 tarefas, como se explica o PMMPI ser mais rápido já que a troca de contexto não é crítica e o SUN MPI está configurado para obter o máximo de desempenho em um ambiente dedicado?

Isso ocorre porque o PMMPI é baseado em *threads* e, conseqüentemente, usa menos recursos do sistema do que o SUN MPI, que é baseado em processos. Cada processo possui o seu próprio espaço de endereçamento e estado do ambiente operacional. O custo de se manter essa grande quantidade de informações de estado faz com que cada

processo seja muito mais caro do que uma *thread*, tanto em espaço quanto em tempo de execução (MULTITHREADED, 2001, p. 20).

Este resultado demonstra que a implementação do PMMPI conseguiu atingir o seu objetivo, ou seja, a implementação com *threads* utiliza de forma mais eficiente os recursos de máquinas SMPs.

5.4.2 Eliminação Gaussiana

Eliminação gaussiana é um método usado para se resolver um sistema de equações lineares Ax = b. Esta aplicação utiliza um teorema que é a base para a Eliminação Gaussiana: Decomposição LU (FORSYTHE, 1967).

Teorema LU: dada uma matriz quadrada não singular A, de ordem n, existe uma única matriz triangular inferior $L = (m_{i,j})$, com $m_{1,1} = m_{2,2} = \ldots = m_{n,n} = 1$, e uma única matriz triangular superior $U = (u_{i,j})$ tal que LU = A. Além disso, $\det(A) = u_{1,1}\, u_{2,2} \ldots u_{n,n.}$

Uma matriz triangular inferior é uma matriz quadrada $C = (c_{i,j})$ tal que $c_{i,j} = 0$ para $i < j$. Analogamente, se $c_{i,j} = 0$ para $i > j$, C é uma matriz triangular superior.

O algoritmo deste teste divide a matriz A entre as tarefas MPI, ficando cada tarefa responsável por atualizar os valores de um determinado número de linhas da matriz A, bem como o elemento da matriz B que está na respectiva linha. Cada tarefa atribui valores iniciais aos elementos das matrizes A e B, que são obtidos de um vetor de números primos, para se evitar dependência linear entre as linhas, obtendo, deste modo, uma matriz singular.

Quadro 32: Fragmento do algoritmo da eliminação gaussiana (parte 1) - inicialização

```
...
/*determinar quantas linhas para cada task */
quant = (int *) malloc ((numTasks) * sizeof(int));
for(p=0 ; p < numTasks ; p++)
      quant[p] = length / numTasks;
sobra = length % numTasks;
p = 0;
if (sobra > 0) {
      while ( sobra > 0) {
            quant[p]++;
            sobra--;
            p++;
            if (p > numTasks)
            p = 0;
      }
}
matSol = (double *) malloc(length * sizeof(double));
matA   = (double **) malloc(quant[rank] * sizeof(double *));
for(p=0; p < quant[rank]; p++)
      matA[p] = (double *) malloc((length)*sizeof(double));
matB = (double *) malloc(quant[rank] * sizeof(double));
posPrimo = posInicial(rank, length);
if(rank == ROOT) {
      for(p = 0; p < length; p++) {
            matSol[p] = proxPrimo(&posPrimo);
      }
}
MPI_Bcast(matSol, length, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);
/* copiar para matB o elemento de matSol correspondente ao rank */
indexMatSol = 0;
for(p=0; p < rank; p++) {
      indexMatSol += quant[p];
}
for(p=indexMatSol, q=0; p < (indexMatSol + quant[rank]); p++, q++) {
      matB[q] = matSol[p];
}
for(i = 0; i < quant[rank]; i++) {
      for(j = 0; j < length; j++) {
            matA[i][j] = proxPrimo(&posPrimo);
      }
}
...
```

Quadro 33: Fragmento do algoritmo da eliminação gaussiana (parte 2 – cálculo)

```
...
for(a=0; a < length; a++) {
      greatest= pivoteamento(matA, a, quant[rank], fora, &linhaPivot);
      in.val = greatest;
      in.rank = rank;
      MPI_Reduce(&in, &out, 1, MPI_DOUBLE_INT, MPI_MAXLOC, ROOT,
                  MPI_COMM_WORLD);
      MPI_Bcast(&out, 1, MPI_DOUBLE_INT, ROOT, MPI_COMM_WORLD);
      if(rank == out.rank) {
            lp = linhaPivot;
      }
      MPI_Bcast(&lp, 1, MPI_INT, out.rank, MPI_COMM_WORLD);
      vet[a].dono  = out.rank;
      vet[a].linha = lp;
      if(rank == out.rank) {
            fora[lp] = TRUE;
            timeBcast = getTime();
            MPI_Bcast(matA[lp], length, MPI_DOUBLE, out.rank,
                  MPI_COMM_WORLD);
            MPI_Bcast(&matB[lp], 1, MPI_DOUBLE, out.rank,
                  MPI_COMM_WORLD);
            for (p=0; p < length; p++)
                  LIN[p] = matA[lp][p];
                  p_b = matB[lp];
      } else {
            MPI_Bcast(&LIN, length, MPI_DOUBLE, out.rank,
                  MPI_COMM_WORLD);
            MPI_Bcast(&p_b, 1, MPI_DOUBLE, out.rank, MPI_COMM_WORLD);
      }
      triangulariza(matA,matB,quant[rank], a, fora, LIN, p_b, length);
} /* for */
/* Retrosubstituicao */
for(a = length-1; a >= 0; a--) {
      if (rank == vet[a].dono) {
            matSol[a]=( matB[(vet[a].linha)]
                        - cumulated[(vet[a].linha)] )
                        / matA[(vet[a].linha)][a];
      }
      MPI_Bcast(&matSol[a], 1, MPI_DOUBLE, vet[a].dono,
            MPI_COMM_WORLD);
      if (rank == 0) {
            printf("Solution[ %d ] = %10.5f\n", a, matSol[a]);
      }
      for(p = 0; p < quant[rank]; p++)
            if( (rank != vet[a].dono) || (p != vet[a].linha) )
                  cumulated[p] = cumulated[p] + (matA[p][a]
                                    * matSol[a]);
}
...
```

Em seguida, é executado um laço que é responsável pela maior parte do tempo de execução do algoritmo. Este laço percorre todas as colunas da matriz A para decompô-la nas matrizes LU. Durante a decomposição, os valores das matrizes L e U são armazenados na própria estrutura de dados da matriz A, sendo que os valores da matriz L – com exceção da diagonal principal – são armazenados nas posições da matriz U

cujos valores são sempre iguais a zero (onde o índice da linha é maior que o índice da coluna), para se aproveitar espaço na memória.

Após a transformação, é executado um laço que percorre todas as colunas da matriz A – desde a última coluna até a primeira – fazendo a retro-substituição para se descobrir a solução do sistema de equações lineares.

Tabela 15: Eliminação gaussiana

No. de tarefas	Tempo de execução (em segundos)							
	< MPI-LWP	< MPI-THR	< SUN MPI	< SUN MPI*	> MPI-LWP	> MPI-THR	> SUN MPI	> SUN MPI*
2	865.3683	899.5711	1852.5073	1852.4769	961.0698	943.2749	1861.7824	1856.1357
4	579.7315	593.0342	1225.2449	1218.4189	607.8733	614.2148	1228.6231	1218.8794
6	369.6538	380.4067	755.0321	760.0280	376.2051	403.3275	762.9100	765.7064
12	448.8069	459.3814	8163.7297	6196.5596	457.4961	464.9510	8345.1995	7356.5995
24	387.6337	434.8240	20094.4163	17392.8287	391.1376	438.6724	20672.9589	17981.9589

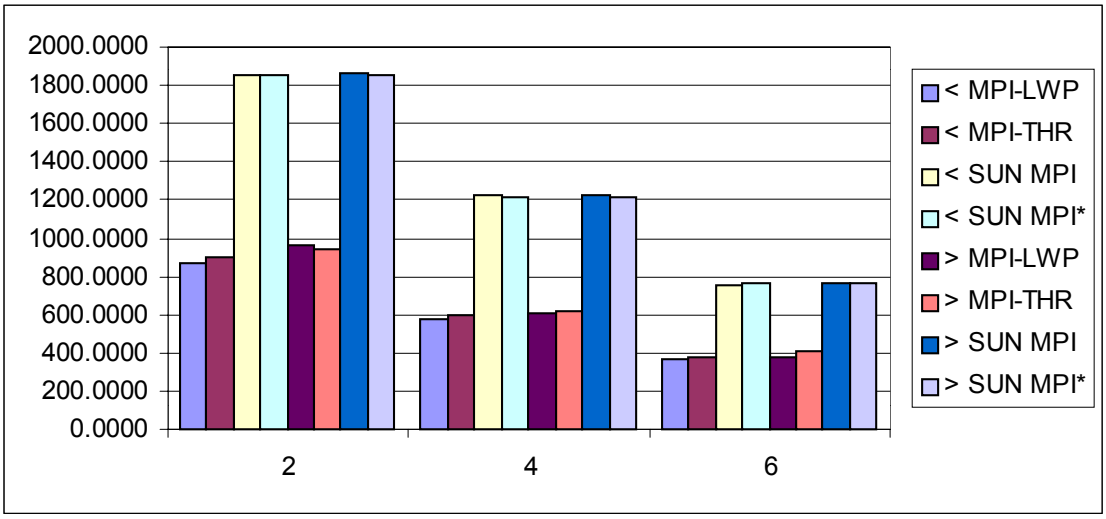

Gráfico 8: Eliminação gaussiana até 6 tarefas

Neste teste, também, o PMMPI foi mais rápido do que o SUN MPI em todos os casos, ou seja, para qualquer número de tarefas.

O maior tempo da computação desta aplicação está no laço que decompõe a matriz A nas matrizes L e U. Neste trecho da aplicação, são executadas muitas operações de difusão (`MPI_Bcast`), entretanto o tempo de execução destas operações é muito pequeno se comparado ao tempo de execução do laço como um todo. A maior parte do tempo deste laço ocorre em duas regiões: na operação de redução (`MPI_Reduce`) e na

triangularização, onde são percorridas as colunas das linhas da matriz que cada tarefa é responsável e não há operações do MPI. Nesta região (`MPI_Reduce` + triangularização), o tempo gasto pelo PMMPI é quase a metade do tempo do SUN MPI, pois a implementação baseada em *thread* exige menos recursos do sistema do que a baseada em processos.

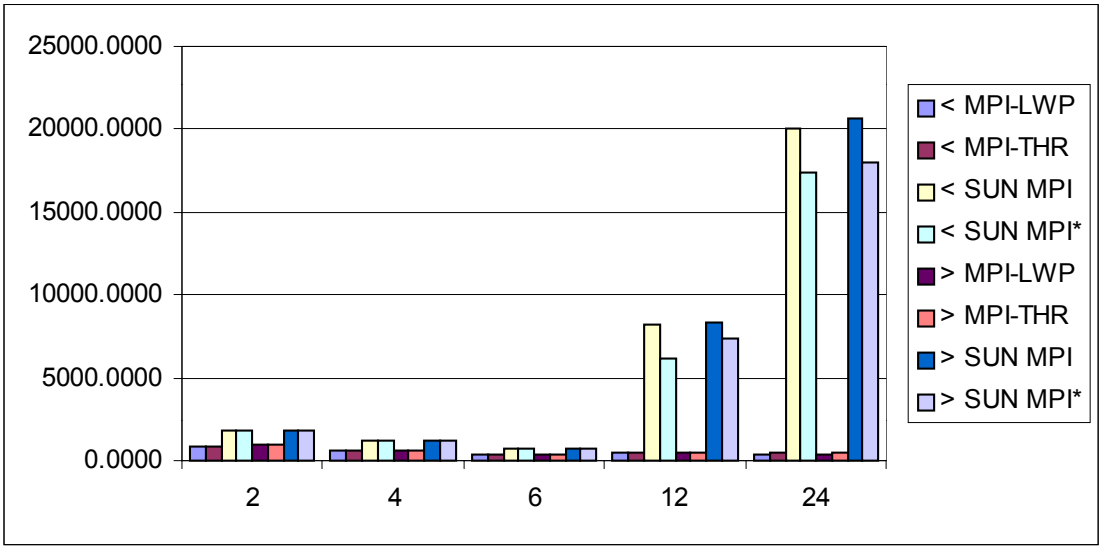

Gráfico 9: Eliminação gaussiana até 24 tarefas

Deve-se ressaltar que, na execução com mais de 6 tarefas o SUN MPI fica muito ineficiente enquanto o PMMPI permanece estável conforme é apresentado no gráfico Gráfico 9. Este resultado mostra que na implementação do MPI baseada em processos não se deve criar mais tarefas do que o número de processadores disponíveis. Por outro lado, na implementação baseada em *threads* esta preocupação é desnecessária.

5.5 Comparação do PMMPI com LWP e com *Solaris Threads*

A biblioteca PMMPI quando *linkeditada* com LWP (MPI-LWP) mostrou-se um pouco mais rápida do que com *Solaris Threads* (MPI-THR), pois ao chamar a primitiva `thr_setConcurrency()` – para indicar ao sistema operacional quantos processadores serão necessários para a execução do programa – o MPI-THR cria *threads de kernel*, além das *threads* de usuário e ambas sofrem troca de contexto. Por outro lado, o MPI-LWP cria apenas *threads de kernel*.

6 Conclusão

6.1 Conclusões sobre o trabalho realizado

Esta dissertação de mestrado descreveu a nossa implementação de um subconjunto de primitivas do padrão MPI, na qual as tarefas de uma aplicação são mapeadas em *threads* irmãs, beneficiando-se das vantagens inerentes às aplicações *multithreaded*, que são: o compartilhamento espontâneo da área de dados e a maior rapidez na criação e troca de contexto entre tarefas em máquinas multiprocessadas.

Esta implementação chama-se PMMPI (*Portable Multithreaded Message Passing Interface*), escrita na linguagem de programação C. O PMMPI não faz chamadas diretamente ao sistema operacional, pois são utilizadas as primitivas de programação paralela do Mulplix. Conseqüentemente, o PMMPI é independente tanto em relação ao sistema operacional, quanto ao modelo de *threads* usado, pois para portar o PMMPI para outro ambiente, basta implementar estas poucas primitivas de programação do Mulplix para o sistema operacional / modelo de *threads* desejado.

Esta independência do PMMPI não compromete o desempenho, pois as primitivas de programação paralela do Mulplix são apenas protótipos de funções, que são implementados fazendo chamadas ao sistema operacional e usando um modelo de *threads* específico.

O capítulo 2 apresentou a especificação MPI destacando seus principais objetivos, descrevendo os seus conceitos básicos e as principais rotinas de comunicação ponto-a-ponto e coletivas. Esta especificação demonstrou-se muito bem projetada, pois a sua interface atendeu perfeitamente a implementação realizada e não precisou ser alterada.

O projeto Multiplus e o sistema operacional Mulplix foram introduzidos no capítulo 3. Neste capítulo também foram apresentados alguns exemplos do uso das primitivas de programação paralela do Mulplix.

A implementação PMMPI foi detalhadamente descrita no capítulo 4, onde foram apresentadas justificativas para se implementar o padrão MPI em máquinas de memória compartilhada, que são:

- Desempenho: o ajuste de desempenho para código MPI em SMPs é normalmente mais fácil, uma vez que o código e o dado particionados apresentam boa localidade na hierarquia de memória;
- Portabilidade: Um programa que usa MPI pode ser transportado facilmente para qualquer máquina paralela sem restrições de plataforma e
- Integração: Pode ser necessária a integração de novas aplicações – desenvolvidas em máquinas SMPs – com programas MPI existentes.

Neste capítulo, também foram destacadas as vantagens de se mapear as tarefas MPI em *threads* ao invés de mapeá-las em processos, as quais são:

- a troca de contexto entre as *threads* irmãs é mais rápida;
- a criação de *threads* é mais rápida e
- a comunicação é mais rápida, pois pode-se utilizar o espaço de endereçamento compartilhado pelas *threads*.

Ainda no capítulo 4, foi citada uma implementação que mapeia as tarefas MPI em *threads*, o TMPI (TANG, 2000). Em relação a esta implementação, o PMMPI apresenta uma vantagem: possui uma camada bem definida – a implementação das primitivas de programação paralela do Mulplix – que facilita o transporte do PMMPI para diversos sistemas operacionais, bem como a utilização de diversos modelos de *threads*.

Por outro lado, o TMPI apresenta algumas vantagens em relação ao PMMPI, que são:

- a existência de um pré-compilador que converte, automaticamente, programas MPI convencionais em programas MPI que podem ser executados com segurança pelo TMPI;
- a implementação de todos os modos de envio do MPI, enquanto a versão atual do PMMPI apenas possui o modo *standard*.

O TMPI possui uma arquitetura de comunicação ponto-a-ponto, que usa uma adaptação de técnicas *lock-free* (ARORA, 1998; HERLIHY, 1991; LUMETTA, 1998;

MASSALIN, 1991) para o modelo de comunicação do MPI, que, segundo os desenvolvedores do TMPI, minimiza o uso de operações atômicas do tipo comparar-e-trocar ou ler-modificar-escrever. Por outro lado, a arquitetura de comunicação ponto-a-ponto que nós projetamos para o PMMPI é mais simples e, no entanto, permitiu o desenvolvimento de uma implementação eficiente do MPI. Para saber qual é a melhor arquitetura de comunicação ponto-a-ponto, é necessário se fazer um estudo comparativo entre estas duas implementações do MPI.

No capítulo 5, é especificado o ambiente usado para o desenvolvimento do PMMPI e realização dos testes, bem como são relacionados os resultados comparativos entre o PMMPI e o SUN MPI.

Os testes das operações básicas do Mulplix, visando comparar o desempenho da sua implementação baseada em LWP com a baseada em *Solaris Threads* (THR), apresentaram o seguinte resultado:

- o tempo de execução da rotina `thr_id` – usada para identificar a *thread* – implementada com *Solaris Threads* é muito menor do que o tempo desta rotina implementada com *lightweight process*, pois, neste caso, é feita uma chamada ao sistema, o que faz com que a *thread* mude do modo usuário para o modo de *kernel*;
- o custo da sincronização com exclusão mútua é praticamente o mesmo;
- a alocação de memória implementada com LWP é um pouco mais rápida do que com THR, apresentando uma diferença em relação ao tempo da THR inferior a 6.55% nos testes realizados até 800.000 iterações;

Os testes comparativos das operações básicas do PMMPI em relação ao SUN MPI tiveram como objetivo comparar e avaliar o desempenho da execução da nossa implementação do MPI baseada em *threads* com outra baseada em processos.

Não foram feitos testes com outra implementação do MPI baseada em *threads*, pois a única encontrada, o TMPI, foi desenvolvida para ser executada em outro sistema operacional.

Os resultados dos testes realizados confirmam que a implementação do PMMPI conseguiu atingir o seu objetivo, ou seja, mostraram que uma implementação baseada em *thread* utiliza de forma mais eficiente os recursos de máquinas SMPs.

Em apenas um dos testes, a implementação SUN MPI obteve o melhor resultado para todos os tamanhos de mensagens enviadas: na operação *broadcast*. Isto ocorreu porque suas operações coletivas são muito otimizadas e fazem uso de um *pool* de *buffers* genérico dentro da memória compartilhada (SUN MPI, 1999, p.57). Por outro lado, a versão atual do PMMPI não utiliza tais recursos.

Entretanto, para o teste com a operação de redução, o PMMPI apresentou melhor resultado – para mensagens acima de 900 bytes – pois esta operação não envolve somente troca de mensagens, mas, também envolve a execução de uma operação. Para mensagens grandes pode haver preempção de uma tarefa e, neste caso, a troca de contexto é mais rápida para a implementação do MPI baseada em *threads*, ou seja, para o PMMPI. Vale relatar que para mensagens de tamanho maior ou igual a 90 Kbytes, o SUN MPI abortou a execução acusando falta de recursos, enquanto o PMMPI não apresentou problemas nestes casos.

A biblioteca PMMPI quando *linkeditada* com LWP (MPI-LWP) mostrou-se um pouco mais rápida do que com *Solaris Threads* (MPI-THR), pois ao chamar a primitiva `thr_setConcurrency()` – para indicar ao sistema operacional quantos processadores serão necessários para a execução do programa – o MPI-THR cria *threads de kernel*, além das *threads* de usuário. Por outro lado, o MPI-LWP cria apenas *threads de kernel*.

O PMMPI comparado com o SUN MPI obteve um melhor desempenho do que o TMPI comparado com o SGI MPI na execução da aplicação Eliminação Gaussiana (TANG, 2000, p. 1017). Supõe-se que um fator importante para este resultado é o fato do *kernel* do Solaris ser *multithreaded*, ou seja, no Solaris a *thread* do *kernel* é a unidade fundamental que é escalonada e executada nos processadores (MAURO, 2001, p.14).

6.2 Contribuições

A principal contribuição do PMMPI foi com o projeto e desenvolvimento de uma arquitetura para comunicação ponto-a-ponto e coletiva para as tarefas MPI mapeadas em threads, que é simples e eficiente.

Nossa implementação do MPI também contribuiu para demonstrar que a especificação MPI conseguiu atingir o seu principal objetivo: "… demonstrar que (usando o MPI) os usuários não precisam comprometer-se entre eficiência, portabilidade e funcionalidade. Isto significa que pode-se escrever programas portáteis os quais podem tomar vantagem do hardware e software especializados oferecidos por vendedores individuais" (GROPP, 1996, p.11).

Durante o desenvolvimento do PMMPI, houve a necessidade de se criar outras primitivas – `ev_test()` e `thr_setConcurrency()` –, contribuindo, assim, para a melhoria do Mulplix.

6.3 Propostas de trabalhos futuros

Esta dissertação de mestrado serve como base para vários trabalhos futuros, a saber:

- Implementar as primitivas do MPI-1.1 que, na presente versão, não foram incluídas;
- Implementar as primitivas do MPI-2;
- Expandir a implementação do PMMPI para um *cluster* de estações de trabalho SMPs, onde a troca de mensagens entre tarefas de um mesmo nó é feita por memória compartilhada entre as *threads* e entre tarefas de nós distintos, por troca de mensagens. Pode-se partir de um estudo sobre este tipo de implementação do MPI que já foram realizadas como, por exemplo: (TANG, 2002; BURNS, 2002; SISTARE, 1999);
- Implementar as primitivas de programação paralela do Mulplix para outro sistema operacional e transportar a implementação extendida do PMMPI para um cluster de PCs;

- Comparar e avaliar o desempenho do PMMPI com as principais implementações do MPI: MPICH, LAM MPI, SGI MPI (implementando a camada do Mulplix para o Irix), TMPI, etc.;
- Otimizar a implementação do PMMPI: criando *pools* de recursos, analisando o código do SUN MPI e incorporando as principais características do mesmo;
- Criar *Benchmarks* que possam ser usados por implementações de MPI baseadas em *threads* sem alterações nos códigos ou transformações dos mesmos;
- Criar uma implementação em Java cujos métodos nativos fazem chamadas às funções do PMMPI e comparar o seu desempenho com implementações do MPI completamente em Java. Pode-se partir de trabalhos como: MPJ (CARPENTER, 1998), jmpi (DINCER, 1998), (GETOV, 2001) e (KIELMANN, 2001).

7 Referências

ANDREWS, Gregory R.; SCHNEIDER, Fred B. Concepts and notations for concurrent programming. ACM Computing Surveys, 1983, 15:3-43

ARORA, N. S.; BLUMOFE, , R. D.; PLAXTON, C. G. Thread Scheduling for Multiprogrammed Multiprocessors. In: PROCEEDINGS OF THE 10th ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, Puerto Vallarta, Mexico, 1998, 119-29.

AUDE, Júlio Salek et al. The Multiplus/Mulplix Parallel Processing Environment. In: INTERNATIONAL SYMPOSIUM ON PARALLEL ARCHITECTURES, ALGORTHMS AND NETWORKS (I-SPAN 96), 1996, Pequim, China. Disponível em: http://www.nce.ufrj.br/multiplus/. Acesso em: 22 mai. 2002.

_____. The Multiplus/Mulplix Project. In: IX SBAC-PAD, 1997, Campos do Jordão, p. 581-586.

_____. Processamento Paralelo de Alto Desempenho. In: II ESCOLA REGIONAL DE INFORMÁTICA, 1998, Belo Horizonte-MG, Goiânia-GO e Viçosa-MG. Belo Horizonte: DCC-UFMG, 1998, p. 71-95.

AZEVEDO, R.P. Mulplix: Um Sistema Operacional Tipo Unix para Programação Paralela. 1993. Dissertação (Mestrado em Ciências) – Coordenação dos Programas de Pós-graduação de Engenharia, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

AZEVEDO, R.P.; AZEVEDO, G.P.; SILVEIRA, J.T.C.; AUDE, J.S. Primitivas para Programação Paralela no MULTIPLUS. In: Anais do V SBAC-PAD, Florianópolis, pp. 761-775, Setembro 1993.

BRUCK, J.; DOLEV, D.; HO, C. T.; ROSU, M. C.; STRONG, R. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. Journal of Parallel and Distributed Computing, 40, 1, (10 Jan.), 19-34, 1997.

BURNS, G.; DAOUD, R.; VAIGL, J. LAM: An Open Cluster Environment for MPI. Ohio Supercomputing Center, 2002.

BUTLER, Ralph; LUSK, Ewing. User's Guide to the P4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, 1992.

CARPENTER, B. et al. MPI for Java: Position Document and Draft API Specification, Technical Report JGF-TR-03, Java Grande Forum, Nov. 1998. Disponível em: http://www.javagrande.org/reports.htm. Acesso em: 22 mai. 2002.

CARRIERO, Nicholas; GELERTNER, David. Linda in Context. Communications of the ACM, vol. 32, no. 4, pp. 444-458, 1989.

CHUANG, W. PVM Light Weight Process Package, Laboratory of Computer Science, Massachusetts Institute of Tecnology, Computation Structures Group, Memo 372, Dezembro, 1994.

DIJKSTRA, E.W. Solution of a problem in concurrent programming control. Communications of the ACM, 8(9):569, 1965.

DINCER, K. jmpi and a Performance Instrum. Analysis and Vis. Tool for jmpi. In: 1st UK WORKSHOP on Java for HPCN, 1998.

FALLER, N.; SALENBAUCH, P. Plurix: A Multiprocessing Unix-like Operating System. In: PROCEEDINGS OF THE 2nd WORKSHOP ON WORKSATION OPERATING SYSTEMS, 1989. Washington: IEEE Computer Society Press, 1989, p. 29-36.

FERRARI, A.; SUNDERAM, V. TPVM: distributed concurrent computing with lightweight processes. In: PROCEEDINGS OF IEEE HIGH PERFORMANCE DISTRIBUTED COMPUTING. IEEE, Washington, D.C., 211-218.

FLYNN, M.J. Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers, vol.C-21, p. 948-960, 1972.

FORSYTHE, G. E., MOLER, C. B. Computer Solution of Linear Algegraic Systems. Prentice-Hall Inc., 1967.

GEIST, A. et al. PVM – A Users Guide and Tutorial for Network Parallel Computing. Cambridge: The MIT Press, 1994. Disponível em: ftp://www.netlib.org/pvm3/book/pvm-book.ps. Acesso em: 22 mai. 2002.

GETOV, Vladmir *et al*. Multiparadigm Communication in Java for Grid Computing. Communications of the ACM, vol.44, no.10, p. 118-125, 2001.

GROPP, W. et al. Using MPI: portable parallel with the message-passing interface. Cambridge: The MIT Press, 1996. 307 p.

_____ et al. Using MPI-2: advanced features of the message-passing interface. Cambridge: The MIT Press, 1999. 382 p.

_____; LUSK, E. PVM and MPI Are Completely Different. Mathematics and Computer Science Division. Argonne National Laboratory, 1998. 15 p. Disponível em: http://www-unix.mcs.anl.gov/mpi/papers/archive/index.html. Acesso em: 25 fev. 2002.

HERLIHY, M. Wait-free synchronization. ACM Transactions on Programming Languages Syst., 1991, 11, 1 (jan.), 124-149.

HOARE, C.A.R. Monitors, an operating system structuring concept. Communications of the ACM, 17(10):549-557.

KERNIGHAN, Brian.W.; RITCHIE, Dennis.M. The C Programming Language. New Jersey: Prentice-Hall, 1988. 272 p.

KIELMANN, Thilo *et al*. Enabling Java for High-Performance Computing. Communications of the ACM, vol.44, no.10, p. 110-117, 2001.

LUMETTA, S. S.; CULLER, D. E. Managing concurrent access for shared memory active messages. In: PROCEEDINGS OF THE INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM. Orlando, Florida, 1998, 272-8.

LUMSDAINE, A. et al. LAM MPI Home page. Disponível em: http://www.lam-mpi.org/. Bloomington: Indiana University. Acesso em: 22 mai. 2002.

LUSK, Ewing; GROPP, William; SKJELLUM. User's Guide for mpich, a Portable Implementation of MPI. Argonne National Laboratory, 1996. Disponível em: http://www-unix.mcs.anl.gov/mpi/mpich/. Acesso em: 22 mai. 2002.

MASSALIN, H.; PU, C. A lock-free multiprocessor OS kernel. Tech. Rep. CUCS-005-91, Columbia University, 1991.

MAURO, Jim; MCDOUGALL, Richard. Solaris Internals: Core kernel Architecture. Palo Alto: Prentice-Hall, 2001. 657 p. Disponível em: http://www.solarisinternals.com. Acesso em: 25 fev. 2002.

MESSINA, Paul et al. Architecture. Communications of the ACM, 1998, vol.41, n. 11, p. 37.

MPI FORUM. Disponível em: http://www.mpi-forum.org. Acesso em: 25 fev. 2002.

_____. MPI: A Message-Passing Interface Standard version 1.1. Knoxville: University of Tennessee, 1995. Disponível em: http://www.mpi-forum.org. Acesso em: 25 fev. 2002.

_____. MPI-2: Extensions to the Message-Passing Interface. Knoxville: University of Tennessee, 1997. Disponível em: http://www.mpi-forum.org. Acesso em: 25 fev. 2002.

MULTITHREADED Programming Guide. Palo Alto: Sun Microsystems, 2001. 387p. Disponível em: http://docs.sun.com . Acesso em: 25 fev. 2002.

NASA. Nasa Information Power Grids. Disponível em: http://www.ipg.nasa.gov. Acesso em: 25 fev. 2002.

NICHOLS, B.; BUTTLAR, D.; FARREL, J. Pthreads Programming. O'Reilly & Associates, Inc., 1996, 267p.

PACHECO, Peter S. A User's Guide to MPI. San Francisco: Department of Mathematics, University of San Francisco, 1998. 51 p. Disponível em: http://www-unix.mcs.anl.gov/mpi. Acesso em: 25 fev. 2002.

PRAKASH, S.; BAGRODIA, R. MPI-SIM: using parallel simulation to evaluate MPI programs. In: PROCEEDINGS OF WINTER SIMULATION. Washington, DC., 467-474, 1998.

PROTOPOPOV, B.; SKJELLUM, A. A multi-threaded message pasing interface (MPI) architcture: performance and program issues. Tech. rep., Computer Science Department, Mississippi State University, 1998.

ROBBINS, Kay A.; ROBBINS, Steven. Practical Unix Programming: a guide to concurrency, communication and multithreading. London: Prentice Hall, 1996. 658p.

SANTOS, Cláudio M. P. MPVM: Uma Implementação de PVM com Uso de Threads para Ambientes com Memória Compartilhada. Dissertação (Mestrado em Ciências) – Coordenação dos Programas de Pós-graduação de Engenharia, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 1998.

SISTARE, S.; VAART, R.; LOH, E. Optimization of MPI Collectives on Clusters of Large-Scale SMP's. In: PROCEEDINGS OF ACM/IEEE SUPERCOMPUTING '99, New York, November 1999.

SKJELLUM, A.; PROTOPOPOV, B.; HEBERT, S. A thread taxonomy for MPI. MPIDC, 1996.

SNIR, Marc. et al. MPI: The Complete Reference. Cambridge: The MIT Press, 1996. 336 p.

STALLINGS, W. Operating Systems: internals and design principles. Prentice Hall, 1998. Disponível em: http://www1.shore.net/~ws/OS3e.html. Acesso em: 25 fev. 2002.

STALLMAN, RICHARD M. Using and Porting GNU Compiler Collection. Boston: Free Software Foundation, 2001. 606 p. Disponível em: http://www.gnu.org/software/gcc/gcc.html. Acesso em: 25 fev. 2002.

SUN ENTERPRISE 4500. Disponível em: http://www.sun.com/servers/midrange/e4500. Acesso em: 25 fev. 2002.

SUN HPC ClusterTools 3.0 Administrator's Guide: With CRE. Palo Alto: Sun Microsystems, 1999. 144p. Disponível em: http://docs.sun.com . Acesso em: 25 fev. 2002.

SUN MPI 4.0 User's Guide: With CRE. Palo Alto: Sun Microsystems, 1999. 74p. Disponível em: http://docs.sun.com . Acesso em: 25 fev. 2002.

TANENBAUM, Andrew S. Structured Computer Organization. New Jersey: Prentice Hall, 1999. 669p.

TANG, Hong; SHEN, Kai; YANG, Tao. Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines. ACM Transactions on Programming Languages and Systems, v.0, n.0, p.999-1025, jan. de 2000. Disponível em: http://www.cs.ucsb.edu/research/tmpi. Acesso em: 25 fev. 2002.

TANG, Hong; YANG, Tao. Optimizing Threaded MPI Execution on SMP Clusters. Disponível em: http://www.cs.ucsb.edu/research/tmpi. Acesso em: 22 mai. 2002.

TREUMANN, R. Experiences in the implementation of a thread safe, threads based MPI for IBM RS/6000 SP. New York: IBM, 1997. Disponível em: http://www.research.ibm.com/actc/Tools/MPI_Threads.htm. Acesso em: 25 fev. 2002.

ZHOU, H.; GEIST, A. LPVM: a step towards multithread PVM. Concurrency – Practice and Experiency, 1997.