UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

André Luís Gobbi Sanches

Sistema de Comunicação de Alto Desempenho Baseado em Programação Genérica

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Antônio Augusto Medeiros Fröhlich

Florianópolis, Dezembro de 2003

Sistema de Comunicação de Alto Desempenho Baseado em Programação Genérica

André Luís Gobbi Sanches

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Computação Paralela e Distribuída e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Dr. Raul Sidnei Wazlawick

Banca Examinadora

Prof. Dr. Antônio Augusto Medeiros Fröhlich

Prof. Dr.-Ing habil. Wolfgang Schröder-Preikschat

Prof. Dr. Mario Dantas

Prof. Dr. José Mazzucco Júnior

Sumário

Lista de Figuras

Lista de Tabelas

”More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.”
W.A. Wulf

”On the other hand, we cannot ignore efficiency.”
Jon Bentley

À lembrança de João Sanches e à Teresinha Gobbi.

Agradecimentos

Agradeço em primeiro lugar a Deus e à família, pelo suporte durante o período. Agradeço também a todos os colegas do LISHA pelo apoio recebido, em especial os participantes do projeto SNOW com trabalhos relacionados (Secco, Charles e Robert) e aqueles com quem tive longas discussões técnicas (Vô e Trier).

Agradeço ao Prof. Fröhlich por me aceitar como aluno, e também aos Profs. Dantas, Mazzucco e Schröder-Preikschat por aceitarem participar da banca. Também sou grato a Universidade Federal de Santa Catarina pela excelente infra-estrutura de ensino e pesquisa, ao Instituto Fraunhoffer-First por ter me aceito como aluno visitante por dois meses e ao Centro GeNESS pelo apoio.

Vários amigos me auxiliaram direta ou indiretamente durante a elaboração deste trabalho, mas não posso citar todos. Cito apenas alguns que colaboraram de forma muito expressiva: Boi, Ale, Caetano, Prof. DeLucca e Djali.

Resumo

Com o surgimento de redes de baixa latência os sistemas tradicionais de comunicação, em uso por anos, mostraram-se inadequados. A falta de flexibilidade e a estrutura rígida em camadas acarreta em maior latência e *overhead* de processamento nessas redes. Com efeito, vários sistemas de comunicação eficientes (GM, PM, BIP) surgiram. Estes sistemas oferecem interfaces distintas, e para que seja possível desenvolver aplicações portáveis interfaces padronizadas também foram desenvolvidas. Dentre estas interfaces, destaca-se a MPI (*The Message Passing Interface Standard*), o qual tornou-se o padrão *de facto* para troca de mensagens. Em virtude disso, *implementações* de MPI foram desenvolvidas sobre as principais bibliotecas de baixo nível.

Apesar de a MPI ser apenas uma *interface* para sistemas de baixo nível, estas implementações costumam ser bibliotecas *middleware* grandes e complexas. A análise destas bibliotecas levou à conclusão de que o tamanho das implementações é causado pela inflexibilidade dos sistemas de comunicação. Para serem genéricos, eles desconsideram o cabeçalhos dos protocolos de camadas superiores a eles, e portanto o sistema de filas de mensagens (que depende dos cabeçalhos) é gerenciado pelas implementações. Grande parte da funcionalidade da MPI, como comunicação imediata e *rendezvous*, depende da manipulação direta das filas, o que obriga as implementações a fornecerem tais serviços, ainda que as bibliotecas de baixo nível os forneçam.

Este texto apresenta um sistema de comunicação baseado em programação genérica, que manipula o cabeçalho da camada superior e gerencia as filas. Como conseqüência, funcionalidades como *rendezvous*, comunicação imediata e cancelamento de mensagens são fornecidas pelo sistema de comunicação, através de

uma interface simples e orientada a objetos. O sistema é baseado em *templates* e metaprogramação estática, o que lhe possibilita alto grau de abstração com desempenho similar às implementações convencionais. O texto também descreve a implementação de MPI sobre o sistema de comunicação proposto, que consiste apenas em uma fina camada que traduz uma *API* para outra. Assim, a implementação resume-se a essência da MPI: uma interface padrão.

O sistema de comunicação não fornece operações coletivas, e estas são implementadas com base nas operações ponto a ponto. Neste aspecto, a implementação também inova em relação às outras implementações: as operações são decompostas em três componentes: topologia, direção e funções. Esta decomposição promove a reutilização de código, e além de reduzir o código necessário para implementar as operações coletivas, permite que elas sejam portadas para outros sistemas de troca de mensagens.

Palavras-Chave: sistemas operacionais orientados à aplicação, programação genérica, programação orientada à aspectos, metaprogramação estática, MPI, comunicação coletiva.

Keywords: application-oriented operating systems, generic programming, aspect oriented programming, static metaprogramming, MPI, collective communication.

Abstract

The upcoming of low-latency networks has proven that traditional communication systems are not suited for high-performance computing. Even being used for many years, the lack of flexibility and the hard layer structure of those systems imply in higher latency and processor overhead. In deed, may efficient communication systems (GM, PM, BIP) have been developed. Each of those systems has its own interface, and since portability is an issue some interface standards have emerged. One of those interfaces is MPI (The Message Passing Interface Standard), which became the *de facto* standard for message passing systems. As a result, MPI *implementations* over most low-level communication libraries are now available.

MPI is just an interface for low-level communication, but even so its implementations are usually large and complex. The analysis of the code of some implementations revealed that the size of the implementations is due to the lack of flexibility of the communication systems. Since being generic is a goal, they don't manipulate the header of the higher layers' protocols, and thus the message queue system (which must access the headers) is handled by the implementations. A large portion of the MPI implementations, such as immediate *rendezvous* communication, requires access to the queue system, and thus the implementations must provide those services, even when the lower lever libraries provide them also.

This paper presents a communication system based on *generic programming*, which access the header of the higher level layer and handle the queues. As a result, functionalities such as *rendezvous*, immediate communication and message cancel are supplied by the communication system, through a simple object-oriented interface.

The system is based on *class templates* and static metaprogramming, that provide performance comparable to the traditional implementation even on high level programs. This paper also describes a MPI implementation over the communication system based on generic programming, which is just a thin layer that translates a API to another. Thus, the implementation is basically the essence of MPI: a standard interface.

The communication system does not provide collective operation, but they are based on the point-to-point operations. In this matter, the proposed implementation also has an innovation over the traditional ones. The operations have been decomposed into three components: *topology*, *direction* and *functions*. This decomposition promotes code reuse and also allow that the collection operations to be ported to another message passing systems.

Keywords: application-oriented operating systems, generic programming, aspect oriented programming, static metaprogramming, MPI, collective communication.

Capítulo 1

Introdução

Os sistemas de comunicação tradicionais costumam ser organizados conforme a pilha de protocolos OSI: cada camada presta serviços às superiores. As implementações geralmente são organizadas em grupos de funções que fornecem os serviços de cada camada [45]. Cada camada enxerga as características das camadas inferiores, mas não das superiores.

Com o surgimento de redes de baixa latência [3] e o seu emprego em computação de alto desempenho os sistemas de comunicação tradicionais mostraram-se inadequados. Sua estrutura rígida resulta em desperdício da largura de banda, aumento da latência e sobrecarga de processamento. Em virtude disto, sistemas de comunicação de alto desempenho freqüentemente acessam o *hardware* de comunicação diretamente e fornecem os serviços de comunicação através de bibliotecas *middleware* [36], o que é conhecido como *OS Bypass*.

Nos últimos anos, surgiu uma miríade de bibliotecas de comunicação de baixo nível e alto desempenho [36, 41, 48]. Estas bibliotecas oferecem serviços similares, embora todas tenham peculiaridades próprias. Mas as interfaces são diferentes, o que dificulta o desenvolvimento de aplicações portáveis.

Devido a isso, há a necessidade de interfaces padronizadas. Da união entre academia e indústria surgiu o padrão MPI (*The Message Passing Interface Standard*) [10], que é atualmente o padrão *de facto* para aplicações de alto desempenho. Pra-

ticamente todas as bibliotecas suportam o padrão MPI, e portanto os programas baseados nesta interface são altamente portáveis.

Entretanto, *implementações* de MPI geralmente são grandes e complexas [24], e portanto adaptar uma biblioteca ao padrão requer esforço considerável. Por exemplo, a implementação de MPI sobre a biblioteca GM para redes Myrinet possui mais de 30.000 linhas de código não portável, bem como mais de 90.000 linhas de código portável. MPI geralmente é fornecida como uma biblioteca *middleware* dividida em camadas, algumas dependentes de arquitetura e outras não. Esta modelagem assemelha-se a dos sistemas de comunicação tradicionais, que por serem ineficientes e inflexíveis motivaram o surgimento de *OS Bypass*. Portanto, uma modelagem mais flexível para sistemas de comunicação, baseada em técnicas modernas de engenharia de software, deve ser mais adequada trazendo assim vantagens.

Considerando o número de linhas de código necessárias para adequar uma biblioteca ao padrão, poderia-se concluir que as bibliotecas possuam diferenças gritantes entre si. Mas ao analisar o código não-portável da implementação, percebe-se que ele lida principalmente com as filas de mensagens e as operações sobre elas (registro, recebimento e cancelamento) e funcionalidades do sistema operacional (gerenciamento de memória e *DMA*).

Mas gerenciamento de fila não é um recurso requerido apenas por programas MPI, e portanto não deve ser responsabilidade da implementação. Este recurso é requerido pela maioria das aplicações e deveria ser fornecido pelas bibliotecas de baixo nível. Porém este recurso é geralmente deixado a cargo de *middleware*, pois as filas devem ser organizadas pelo cabeçalho de protocolo da MPI, de forma que nenhuma mensagem seja confundida. Como as bibliotecas devem ser genéricas e suportar outros protocolos, elas não manipulam os cabeçalhos dos protocolos das camadas superiores. Como as filas dependem dos cabeçalhos, as bibliotecas de baixo nível não as gerenciam.

Entretanto, esta decisão de projeto obriga as bibliotecas de alto nível, ou mesmo as aplicações, a inflarem seus códigos com funcionalidades que não lhes cabem. Cada biblioteca deve implementar suas próprias filas, o que resulta em código duplicado entre as bibliotecas. Neste trabalho, é demonstrado que é possível desenvolver um sis-

tema de comunicação que enxerga os cabeçalhos dos protocolos das camadas superiores mas que mantém-se genérico, baseado em técnicas modernas de engenharia de software. Também se demonstra que esta com mudança reduz-se significativamente o tamanho de uma implementação de MPI e de outras bibliotecas, sem desrespeitar a interface e o comportamento definidos pelo padrão e obtendo-se desempenho similar. Desta forma, obtém-se uma implementação que não é uma biblioteca *middleware*, mas apenas a essência da MPI: A **Interface** Padrão de Troca de Mensagens.

O texto também faz uma profunda análise das operações coletivas da MPI, e elas são decompostas em 3 componentes: topologia, direção e funções. Esta decomposição permite implementar as operações de forma elegante e com alta reutilização de código, sendo possível portar as operações para outros sistemas de trocas de mensagens.

O *próximo capítulo* apresenta os conceitos que serviram de base para este trabalho. Inicia-se descrevendo as técnicas de engenharia de software empregadas e as topologias de rede mais comuns em ambientes paralelos. Em seguida, apresenta-se o sistema operacional EPOS e o seu sistema de comunicação, que serviu de base para o objeto deste trabalho.

O *capítulo 3* descreve o sistema de comunicação baseado em programação genérica. Neste capítulo é demonstrado como o sistema consegue gerenciar o sistema de filas e as vantagens desta abordagem. O funcionamento do sistema é descrito em detalhes e no final do capítulo há um exemplo de código que utiliza o sistema.

O *capítulo 4* descreve a implementação de MPI para o sistema proposto. A seção 4.1 faz uma breve análise do padrão MPI e a seção 4.2 demonstra como a comunicação ponto a ponto é facilmente implementada graças à flexibilidade e aos recursos do sistema. A seção 4.3 descreve uma forma altamente configurável de gerenciar os tipos de dados da MPI. A seção 4.4 descreve os mecanismos usados para decompor as operações coletivas e as vantagens da decomposição. Por último, apresenta-se uma comparação de desempenho entre um protótipo do sistema e uma implementação convencional de MPI.

O *capítulo 5* conclui o texto. Nele listam-se as vantagens do sistema

baseado em programação genérica sobre os sistemas convencionais, bem como suas limitações. Por último, citam-se trabalhos futuros.

Capítulo 2

Conceitos

2.1 Técnicas de Engenharia de Software

As técnicas descritas a seguir compõem um conjunto de paradigmas de programação complementares a orientação a objetos. É utilizada uma abordagem multi-paradigma conforme proposto por Stroustrup [46], ao invés de orientação a objetos pura. Como será demonstrado, esta abordagem oferece benefícios em organização, flexibilidade e desempenho.

2.1.1 Programação Genérica e Metaprogramação Estática

O paradigma de programação genérica consiste em desenvolver algoritmos capazes de operar sobre diferentes tipos de dados. Os algoritmos devem impôr um reduzido número de *restrições* aos tipos sobre os quais podem operar, tornando-se independentes da representação dos dados. Desta forma, a representação das estruturas de dados pode ser *encapsulada* e o comportamento adequado é obtido através de polimorfismo.

Entretanto, geralmente se implementa polimorfismo através de ligação dinâmica, o que acarreta em grande perda de performance. Em algoritmos onde desempenho é importante, o polimorfismo costuma ser evitado sacrificando assim a capacidade dos algoritmos de serem genéricos. Portanto, é necessário encontrar formas de expressar

algoritmos genéricos que não causem impacto na performance do código gerado.

Em vista disso, linguagens que suportam programação genérica oferecem diretivas que permitem especificar algoritmos independentes de *tipos*. Em C++, esta diretiva denomina-se *template*. O código a seguir exemplifica o uso desta diretiva:

```
template <class T> T soma(T v1, T v2) {
return (v1 + v2);
}

...

int i;
double d;
i = soma(1, 2);
d = soma(1.3, 2.5);
```

A função `soma` recebe dois argumentos e retorna um valor do tipo `T`. `T` pode ser mapeado para qualquer tipo, seja uma classe ou um tipo pré-definido. A única *restrição* feita ao tipo `T` é que implemente o *operador* `+`. O tipo que `T` representa é reconhecido pelos tipos de seus argumentos, que devem ser iguais. O compilador gera uma versão da função `soma` para cada tipo chamado no programa, neste caso para `T=int` e `T=char`. Como o tipo de dado é definido em tempo de compilação para todas as funções *parametrizadas* (genéricas), não há necessidade de ligação dinâmica e da manutenção de informação sobre tipos em tempo de execução, eliminando as perdas de desempenho ocasionadas pelo uso de *polimorfismo*.

Algoritmos genéricos bem projetados devem impor o mínimo de restrições possível aos tipos sobre os quais pode operar. As restrições impostas constituem os operadores e métodos requisitados, consistência na semântica e complexidade dos métodos requisitados. Esses algoritmos podem ser adaptativos, e com base na com-

plexidade dos métodos de seus parâmetros, alterarem seu comportamento. Algoritmos podem ter comportamento diferenciado ao inserirem elementos em listas ligadas e simples vetores, por exemplo. Mas devido ao encapsulamento, as adaptações são opcionais e transparentes ao usuário.

Uma classe definida pelo usuário pode implementar este operador da seguinte forma:

```
class Int {
int value;
public:
inline Int(int _value=0) value(_value);
inline ~Int();
inline int operator+(int _other) { return (value+=other); }
inline operator int() { return value; }
};
```

No exemplo, é definida a classe `Int` que representa um valor inteiro. A classe `Int` define o operador `+`, e portanto pode ser utilizado na função `soma`. A diretiva `inline` indica que quando este operador for referenciado, a sua implementação deve substituir a referência, evitando a necessidade de uma chamada de função. Desta forma, o código a seguir:

```
...

Int I(3);

std::cout << I+4 << std::endl;
```

```
...
```

é equivalente a:

```
...

Int I(3);

std::cout << I.value+4 << std::endl;
```

de forma que, eliminando a chamada de função, não há perda de desempenho no uso de *encapsulamento*. Portanto, é possível definir qualquer nível de abstração sem prejudicar o desempenho do código gerado.

As diretivas `template` e `inline` definem como o compilador deve gerar o código do programa, e portanto a sua utilização é conhecida como *Metaprogramação Estática*. Conforme foi demonstrado, através de Metaprogramação Estática é possível utilizar *encapsulamento* e *polimorfismo* sem acarretar em perda de desempenho. Considerando que a herança não acarreta em perda de desempenho, pode-se assim obter completa funcionalidade da orientação a objetos. Pelo contrário, com o ganho em flexibilidade é possível otimizar os algoritmos.

Entretanto, o uso de metaprogramação estática tem duas desvantagens:

1. Todas as informações de tipos devem estar disponíveis em tempo de compilação;
2. Para cada variação dos parâmetros de um `template`, uma nova classe será gerada, aumentando o código gerado.

Neste projeto, conforme será demonstrado nos próximos capítulos, ambas as desvantagens são contornadas.

2.1.2 Padrões de Projeto

Gamma e co-autores [18] definem padrões de projeto como “um novo mecanismo para expressar experiência de projeto orientado a objetos”. Padrões de projeto representam estruturas que ocorrem freqüentemente na análise de sistemas. O seu uso, além de permitir o reuso de análise, padroniza o sistema, facilitando o seu entendimento por terceiros. Em [18] é proposto um catálogo de padrões de projeto cuja ocorrência é freqüênte. O catálogo não é de forma alguma completo, e a identificação de novos padrões é encourajada.

Para ilustrar o uso de padrões de projeto, o padrão *Observer* será descrito a seguir. Este padrão representa a relação entre dois objetos, denominados *Alvo* (*subject*) e *Observador* (*Observer*). O Observador é um objeto que monitora e reage a mudanças no estado do Alvo. Um mesmo Alvo pode ser monitorado por diversos objetos Observadores simultaneamente.

Este padrão geralmente é implementado através de uma lista de funções *callback*, que representam as reações dos Observadores às mudanças de estado do Alvo. Assim, sempre que o Alvo alterar seu estado, ele *notifica* todos os observadores chamando todas as funções da lista.

O padrão *Observer* é geralmente utilizado em interfaces GUI (*Graphical User Interface*), onde os componentes visuais são os Observadores e os modelos que eles representam são os Alvos. O próprio modelo MVC (*Model-View-Controller*), base de diversas interfaces gráficas, constitui um padrão de projeto.

Diversos padrões de projeto foram identificados por Gamma e co-autores [19] ou podem ser encontrados em outros catálogos.

O Projeto de um sistema orientado a objetos pode ser baseado nas interações entre classes, ou seja, nos *papéis* que cada classe assume em cada padrão de projeto. Esta técnica de projeto é freqüentemente conhecida como *Projeto Baseado em Papéis* (*Role Based Design*) [51].

2.1.3 Implementação Através de Programação Genérica

VanHilst e co-autores demonstram que projetos baseados em papéis podem ser eficientemente implementados através de classes parametrizadas (programação genérica)[51]. Cada papel é representado por uma classe cujo parâmetro é a classe que assume o papel. Além disso, padrões de projeto podem ser classes parametrizadas cujos parâmetros sáo os papéis que a compõem. Esta abordagem permite que padrões de projeto sejam utilizados sem incorrer em perda de performance, e torna o código organizado e padronizado.

2.1.4 Programação Orientada a Aspectos

Kiczales e co-autores [32] identificaram que determinadas funcionalidades são difíceis de isolar em classes utilizando apenas orientação a objetos. Estas funcionalidades tem por características *atravessar* a hierarquia de classes, sendo divididas em várias classes. Como exemplo destas funcionalidades pode-se citar tratamento de exceções e sincronização, entre outras.

A política de tratamento de exceções (ex: quais exceções tratar, e como notificar o usuário) afeta todas as classes do sistema que podem disparar exceções, tornando difícil o isolamento desta funcionalidade em uma entidade específica. O mesmo ocorre com o código responsável pela sincronização: ele afeta todas as classes que alteram seu comportamento quando há concorrência.

Kiczales e co-autores propõem uma técnica de programação conhecida como ”Programação Orientada a Aspectos”que é complementar a Programação Orientada a Objetos. Por *aspectos* entende-se toda funcionalidade que *atravessa* (*cross-cut*) o sistema, como o tratamento de exceções e a sincronização supra-citados. Os aspectos são isolados em entidados e inseridos no programa por meio de um *weaver*. Aspectos podem ser definidos por uma linguagem própria que define um padrão que, se reconhecido, ativa a inserção do código de tratamento do aspecto. Coady e co-autores [6] obteram bons resultados ao aplicarem programação orientada a aspectos no desenvolvimento de sistemas operacionais.

É possível, assim, marcar na implementação de um sistema operacional todas as classes que necessitam de sincronização na presença de concorrência de forma que um *weaver* possa inserir o código apropriado.

O Sistema Operacional *EPOS* trata o *aspecto* de sincronização através de adaptadores (extensões) para as classes que necessitam de sincronização. O código de sincronização é chamado antes e após quaisquer métodos destas classes.

Os benefícios da aplicação de Programação Orientada a Aspectos em sistemas operacionais foram demonstrados por Coady e co-autores [6].

2.1.5 Programação Multiparadigma

Stroustrup [46] propõe que o desenvolvimento de software não seja baseado em apenas um paradigma, mas em uma coleção. Esta abordagem opõe-se a metodologia de orientação a objetos pura, mas é consistente. Todas as técnicas descritas nas seções anteriores são complementares ou ortogonais ao paradigma de orientação a objetos, e possibilitam melhor organizar o código e contornar alguns dos problemas de sistemas orientados a objetos, em especial perda de desempenho. Em aplicações de alto desempenho, este quesito é determinante quanto à escolha do paradigma.

2.1.6 Contêineres e Iteradores

> "*Os iteradores são a cola que mantém unidos contêineres e algoritmos. Eles fornecem uma visão abstrata dos dados de modo que quem escreve um algoritmo não precisa se preocupar com detalhes concretos de uma miríade de estruturas de dados.*"
>
> Bjarni Stroustrup

A biblioteca padrão da linguagem C++, STL (*Standard Template Library*), oferece uma série de contêineres metaprogramados e formas práticas de acesso aos seus conteúdos. Os contêineres oferecidos vão desde um simples vetor (`std::vector`) até estruturas mais complexas como dicionários que podem ser indexados por qualquer

objeto. Stroustrup recomenda a utilização dos contêineres padrão aos invés dos simples vetores de C [46].

Os contêineres padrão oferecem de forma unificada as operações básicas de acesso, resumidas na tabela 2.1. Uma listagem completa pode ser obtida em [46].

Deve-se prestar atenção especial na última linha, que descreve o atributo `iterator` como `value_type*`. O *iterador* de um contêiner consiste em um objeto que permita *iterar* por seu conjunto de dados. Não há restrições quanto à classe do iterador, mas ele deve ser capaz de referenciar seu alvo de forma semelhante a um ponteiro e respeitar uma interface padrão, a da classe `std::iterator`.

Portanto, o seguinte trecho de código:

```
#define SIZE 10
char s[SIZE], *p;

...

for (p=s; p<(s+SIZE); p++)
printf("%c", *p);
```

é equivalente a:

```
const int size=10;
std::string s;
std::string::iterator p;

...

for (p=s.begin(); p<s.end(); p++)
std::cout << *p;
```

`[]`	Subscrito, acesso não-verificado
`at()`	Subscrito, acesso verificado
`begin()`	*Aponta* para o primeiro elemento
`end()`	*Aponta* para um elemento após o último
`rbegin()`	*Aponta* para o primeiro elemento na ordem inversa
`rend()`	*Aponta* para um elemento após o último na ordem inversa
`size()`	Retorna o número de elementos
`value_type`	Tipo de elemento armazenado
`iterator`	Comporta-se como `value_type*`

Tabela 2.1: Operações básicas de acesso a elementos de contêineres

O segundo trecho de código demonstra uma forma de acesso a dados independente de representação. `p` é um iterador, podendo ou não ser um ponteiro. O tipo do iterador é definido pela implementação da STL. Através de metaprogramação estática é possível utilizar um ponteiro sem perda de desempenho e implementar o operador `*` como uma função. Como o acesso independe da implementação, é possível definir funções parametrizadas que iteram sobre qualquer tipo de iterador, como pode ser observado no trecho de código a seguir:

```
template<class container> void inc(container vector) {
// increments each element of the container

container::iterator iter;
for (iter=vector.begin(); iter<vector.end; iter++)
++(*iter);
}

std::string s("Iterating...");
```

```
std::vector<int> v(10);

...

inc(s);
inc(v);
```

Como pode ser observado, contêineres e iteradores oferecem alto grau de abstração, sem implicar em perda de performance ou aumento de complexidade. O seu uso permite que algoritmos, ao interagirem apenas com iteradores, sejam implementados de forma genérica. Tais algoritmos podem ser executados sobre qualquer estrutura de dados, contínua ou não, uma vez que os iteradores abstraem o método de acesso. Iteradores são uma alternativa a ponteiros na programação orientada a objetos.

2.1.7 Objetos de Função

A linguagem C++ oferece o recurso de encapsular funções em objetos. Este encapsulamento permite substituir ponteiros para funções por referências a objetos, de forma eficiente. Em parâmetros de *templates*, o uso de objetos de função é recomendado, pois facilita a eliminação da chamada de função.

2.2 Topologias de Rede

Para a compreensão da análise de operações coletivas a ser apresentada adiante é necessário entendimento de topologias. Portanto, nesta seção será feita uma revisão das topologias regulares mais comumente encontradas.

A topologia de uma rede constitui-se na configuração das conexões entre seus nodos. Através da análise da topologia é possível definir os algoritmos e as rotas adequadas para as comunicações, evitando gargalos.

2.2.1 Características de Topologias

Foster [13] classifica a comunicação entre entidades de processamento de um programa paralelo com base em quatro características:

- Caso cada nodo se comunique apenas com um conjunto restrito de outros nodos, a comunicação é dita *local*. Caso contrário é dita *global*;
- Caso um nodo e o conjunto com o qual se comunica formem uma estrutura (topologia) regular, como uma árvore ou um *mesh*, a comunicação é dita *estruturada*. Caso contrário, é dita *não estruturada*;
- Caso os parceiros nas comunicações sejam fixos, a comunicação é dita *estática*. Caso contrário é dita *dinâmica*;
- Caso as comunicações sejam executadas em ordem pré-determinada e esperada por todos os participantes, elas são ditas *síncronas*. Caso contrário, são ditas *assíncronas*.

Ao executar aplicações paralelas, é precisar levar em conta as características da topologia da aplicação, tanto nas rotinas de troca de mensagens quanto no mapeamento das tarefas. Caso a comunicação entre as tarefas tenha tendência *estática* e *local*, é importante mapear as tarefas que mais comunicam-se entre si a nodos vizinhos, diminuindo o número de *hops* necessários. Caso seja também *estruturada*, os subgrupo formarão subtopologias, e algoritmos adaptativos poderão valer-se deste fato para otimizar a utilização da rede. Caso a comunicação tenha forte tendência *global*, o centro da rede provavelmente será sobrecarregado, e uma rede com grande *largura de bisseção* é mais apropriada. Caso haja forte dependência entre as entidades de processamento, uma rede de baixa latência pode ser adequada. Caso hajam poucas interdependências, como em uma aplicação SPMD, direcionar os recursos financeiros para a aquisição de entidades de processamento pode ser mais vantajoso.

Portanto, analisar as características da aplicação a ser executada possibilita identificar a rede física mais apropriada para a execução da aplicação, levando

em conta eficiência e custo. O grau de dependência entre as entidades de processamento determina a tecnologia de interconexão necessária e a relação entre multiprocessamento e multicomputação. A característica da comunicação, ou o grafo de dependências entre entidades, determina a topologia ideal da rede para a execução da aplicação. A escolha adequada da rede possibilita melhor desempenho e menor desperdício de conexões disponíveis.

Entretanto, em alguns casos diversas aplicações são executadas no mesmo computador ou agregado. Neste caso, a rede de conexão deve ser adequada a várias aplicações, possivelmente com topologias naturais diferentes, e portanto não é possível implementar a rede física ideal da aplicação. É possível, também, que a topologia da aplicação seja demasiado irregular, e os administradores podem optar por uma topologia mais regular.

Em ambos os casos, é necessário que seja feito um mapeamento entre a topologia requerida pela aplicação (sua topologia *natural*) e a topologia física da rede, evitando gargalos e garantindo o bom aproveitamento da rede disponível. Uma implementação de MPI deve lidar com variações de topologias.

A escolha precisa do roteamento de mensagens garante baixa latência na transmissão de mensagens e evita gargalos. Geralmente, a melhor escolha costuma ser a que passa por menos nodos (*hops*).

Na seção a seguir são descritas as topologias básicas. A grande maioria das topologias encontradas em sistemas paralelos são formadas por estas topologias ou variações e combinações delas.

2.2.2 Topologias Básicas

Para avaliar as diferentes topologias, serão considerados os parâmetros de comparação apresentados a seguir. A *distância* entre dois nodos é o número mínimo de *hops* de uma mensagem entre eles. N é o número de processos e k é um número tal que $N = k^2$.

- *Diâmetro:* a distância máxima entre dois nodos quaisquer.

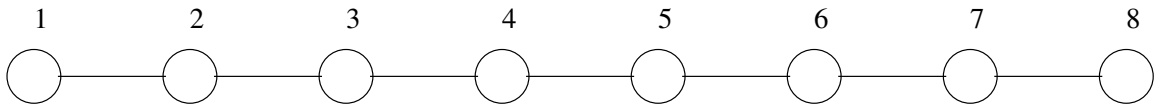

Figura 2.1: Topologia Linear

- *Conectividade:* Número de rotas possíveis entre qualquer par de nodos.
- *Largura da Bisseção:* Mínimo número de linhas de conexão que precisam ser removidas para que a rede seja dividida pela metade.
- *Custo:* Número de linhas de conexão.

O diâmetro da rede afeta a latência das mensagens entre nodos distantes. A conectividade e a largura da bisseção determinam quantas comunicações simultâneas a rede pode suportar. O custo significa a quantidade de recursos financeiros dispendidos para implantar a rede.

Com base nestes parâmetros, as topologias básicas serão descritas a seguir.

2.2.3 Linear

A topologia *linear*, ilustrada na figura 2.1, consiste em organizar os nodos em seqüência, ligando cada nodo ao seus vizinhos (e apenas a eles). O primeiro nodo da seqüência é conectado apenas ao segundo, assim como o último é conectado apenas ao penúltimo. Todos os demais são conectados aos seus antecessores e sucessores.

Diâmetro:	$N/2$
Conectividade:	1
Largura da Bisseção:	1
Custo:	$N-1$

As vantagens da topologia linear são o baixo custo e a simplicidade. Entretanto, como há sempre apenas uma rota para cada mensagem, o centro da rede tende

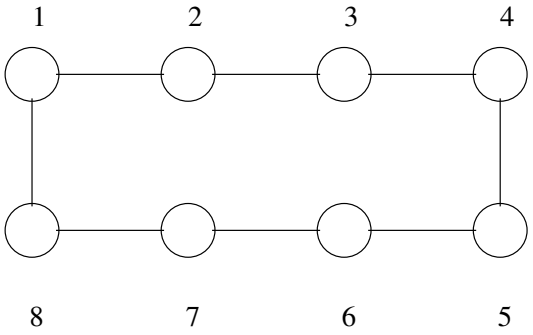

Figura 2.2: Topologia Circular ou em Anel

a ser um gargalo e comunicações entre nodos distantes tem grande latência. A topologia circular, descrita a seguir, diminui este problema.

2.2.4 Circular ou em Anel

A topologia circular é semelhante a topologia linear, mas tem uma linha de conexão entre o primeiro e o último processo, gerando um anel. Desta forma, há sempre duas rotas para cada nodo, diminuindo os problemas da topologia linear. O custo da rede é ligeiramente superior. A figura 2.2 ilustra a topologia circular.

Diâmetro:	$N/2$
Conectividade:	2
Largura da Bisseção:	2
Custo:	N

A rede circular é adequada para pequenas redes, mas em redes maiores a baixa largura da bisseção torna-se um problema.

2.2.5 Mesh

A topologia *Mesh* consiste em dividir uma rede linear de tamanho N em diversas sub-redes de tamanho k, tal que $N = k^2$ e conectar os nodos correspondentes, gerando uma grade. A figura 2.3 ilustra esta topologia.

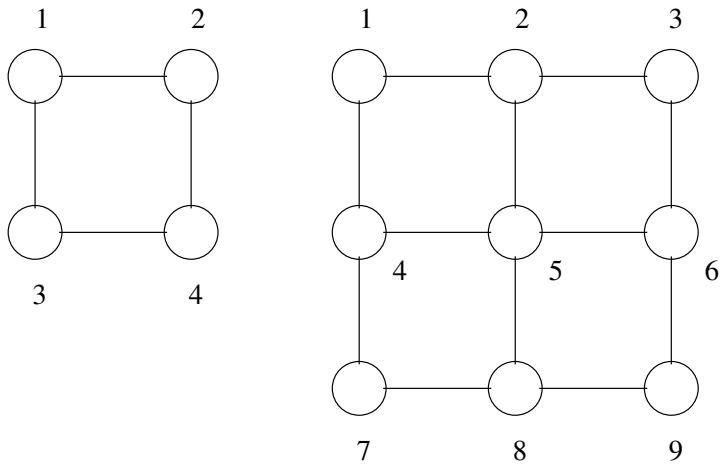

Figura 2.3: Topologia Mesh

A topologia *Mesh* tem o diâmetro inferior e a largura da bisseção superior às topologias linear e circular, e portanto menos gargalos. Entretanto, a topologia é assimétrica e a conectividade da rede é baixa.

Diâmetro:	$2(k-1)$
Conectividade:	2
Largura da Bisseção:	k
Custo:	$2k(k-1)$

2.2.6 Torus

Assim como a topologia *Mesh* consiste em dividir uma rede em sub-redes lineares, a topologia *Torus* consiste em dividir uma rede em sub-redes circulares e conectar os nodos correspondentes. Um Torus bidimensional pode ser observado na figura 2.4.

No caso de redes com muitos nodos, pode-se utilizar uma topologia Torus multidimensional. Neste caso a rede é complexa, mas a conectividade e a largura da bisseção são maiores, evitando gargalos. O diâmetro diminui com o número de dimensões, reduzindo a latência.

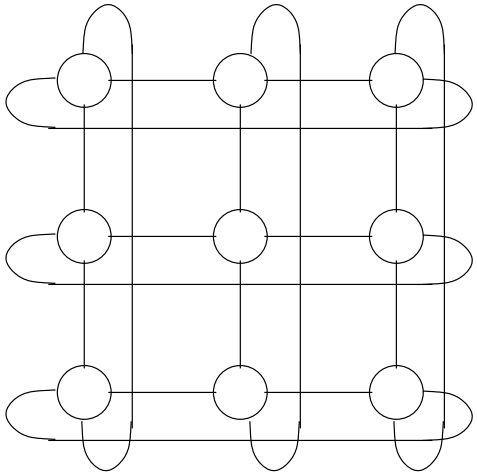

Figura 2.4: Topologia Torus bidimensional

Diâmetro:	k
Conectividade:	4
Largura da Bisseção:	$2k$
Custo:	$2k^2$

2.2.7 Hipercubo

A topologia hipercubo consiste em organizar os nodos em forma de um cubo de dimensão d tal que $N = 2^d$, o qual é formado por dois cubos de dimensão $d - 1$ com linhas de conexão entre os nodos correspondentes. Todos os nodos são conectados a d outros nodos. A figura 2.5 exemplifica esta topologia.

Diâmetro:	d
Conectividade:	d
Largura da Bisseção:	$N/2$
Custo:	d

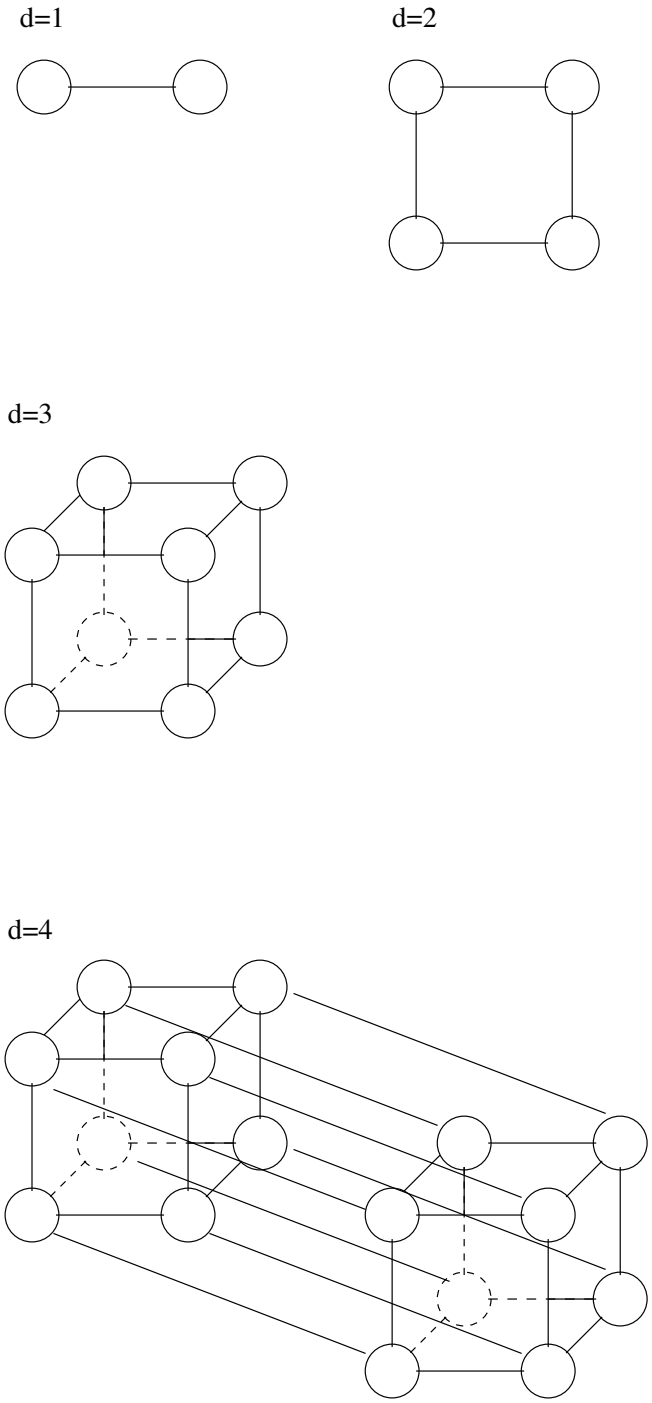

Figura 2.5: Topologia Hipercubo

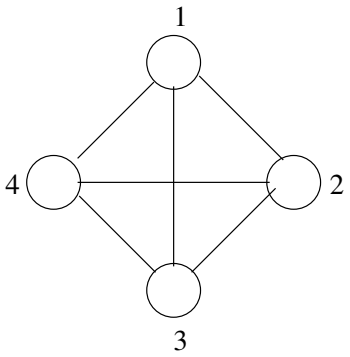

Figura 2.6: Topologia Completa

2.2.8 Completa

Em uma rede com topologia completa, todos os nodos estão conectados a todos os outros nodos. Esta topologia é eficiente, porém o custo aumenta de forma quadrática em relação a N.

Diâmetro:	1
Conectividade:	$N-1$
Largura da Biseção:	$N^2/4$
Custo:	$p(p-1)/2$

2.2.9 Estrela

Na topologia em estrela um nodo central está conectado a todos os outros. Não há conexões com os demais. O nodo central é o gargalo da rede, como pode ser observado na figura 2.7. Esta topologia é natural para aplicações mestre-escravo, nas quais o nodo mestre sempre será um gargalo, mas é pouco adequada a outras classes de aplicações.

2.2.10 Barramento

Nesta topologia, todos os nodos estão conectados a um dispositivo de interconexão que age como um barramento comum. Apenas uma mensagem pode ser

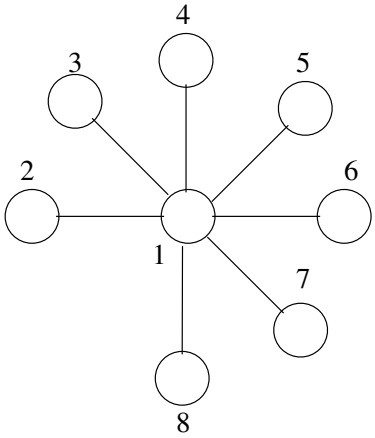

Figura 2.7: Topologia em Estrela

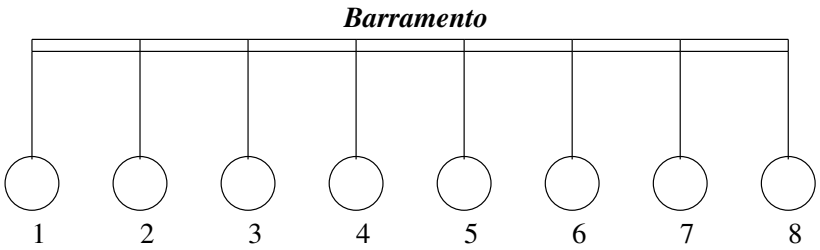

Figura 2.8: Topologia em Barramento

transmitida na rede ao mesmo tempo. A topologia barramento é bastante econômica, tem *broadcast* nativo em *hardware* e é bastante comum em agregados *Beowulf*. Entretanto, o dispositivo de interligação tende a ser um gargalo, e portanto a topologia em barramento é pouco escalável. Por este motivo, é pouco comum em grandes redes. A figura 2.8 ilustra a topologia em barramento.

2.2.11 Árvore

Na topologia em árvore, um nodo raiz divide a rede em duas subárvores, que por sua vez também tem uma raiz que a subdivide em duas, e assim sucessivamente, conforme pode ser observado na figura 2.9. Os nodos mais abaixo na árvore são conhecidos como nodos folha.

A topologia em árvore é adequada a aplicações regulares com

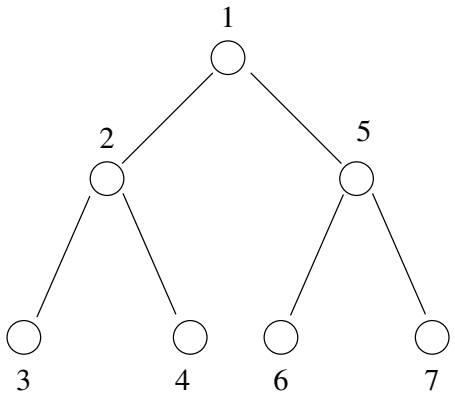

Figura 2.9: Topologia em Árvore

comunicação local, e aplicações com esta topologia são comuns.

2.3 O Sistema Operacional EPOS

O sistema operacional *EPOS* (Embedded Parallel Operating System) foi desenvolvido no instituto de pesquisa alemão GMD-First e é baseado nos princípios de *AOSD* (Application Oriented System Design), descrito adiante. *EPOS* é um sistema configurável, que permite ao usuário determinar o nível de funcionalidade requerido para cada um de seus subsistemas, como gerência de processos, gerência de memória ou sistema de comunicação. É possível, por exemplo, desligar o suporte a multiprocessamento ou desabilitar a segurança entre processos. Para facilitar a configuração, um *script* analisa o código fonte das aplicações a serem executadas e determina quais recursos do *EPOS* serão necessários. O usuário também pode utilizar uma ferramenta gráfica para habilitar otimizações extras.

A configuração do sistema *EPOS* é baseada em meta-programação estática [46] e orientação a aspectos [32], para oferecer configurabilidade similar a de sistemas reflexivos sem incorrer em perda de desempenho. Em testes [17], o sistema *EPOS* obteve desempenho superior ao do sistema *Linux* em redes de comunicação *Myrinet* [3].

A configuração do sistema *EPOS* ocorre durante compilação do sistema, e portanto é classificada como *configuração estática*. Esta característica limita sua aplicação a sistemas dedicados. Este nicho é bastante amplo, pois muitos agregados e computadores de grande porte executam apenas um conjunto pré-definido de aplicações. E ao desabilitar funções não necessárias, *EPOS* obtém excelente desempenho nestes sistemas.

2.3.1 Application Oriented System Design

AOSD [16] é a metodologia com qual o sistema *AOSD* baseia-se em 4 conceitos: *famílias de abstrações*, *aspectos de cenário*, *mediadores do sistema* e *interfaces infladas*. Estes conceitos serão descritos a seguir.

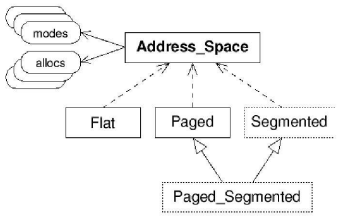

Figura 2.10: Família de Gerência de Memória do *EPOS* [16]

2.3.2 Famílias de Abstrações

As abstrações do sistema operacional (Gerência de Processos, Gerência de Memória, Sistema de Comunicação) são organizadas em *famílias* [39], onde cada *membro* representa um *perfil* ou um nível de funcionalidade. Configurar a família significa escolher o membro mais apropriado para representá-la. A família de Gerenciamento de Memória de *EPOS*, por exemplo, possui quatro membros: Flat, Paged, Segmented e Paged_Segmented. Flat define que o gerenciamento de memória é simples (o endereço virtual corresponde ao físico), indicado para a maioria dos sistemas embutidos. Segmentação e paginação serão utilizadas apenas se forem necessárias. Assim, *EPOS* pode ser aplicado tanto em sistemas muito simples, quanto em sistemas mais complexos.

Para permitir que o membro mais indicado de cada família seja escolhido, é importante que hajam poucas dependências de membros específicos entre as famílias. Por exemplo, o sistema de comunicação de *EPOS* depende do sistema de memória, mas as funções que utiliza (*malloc* e *free*) são implementadas por todos os membros. Assim, há dependência entre famílias, mas não há restrições quanto à escolha dos membros.

2.3.3 Aspectos de Cenário

As famílias de abstrações de sistemas *AOSD* devem ser desenvolvidas de forma a serem independentes de características específicas do *ambiente de execução*. Estas características são representadas por *aspectos de cenário* [15], que estendem a funcionalidade do *EPOS*.

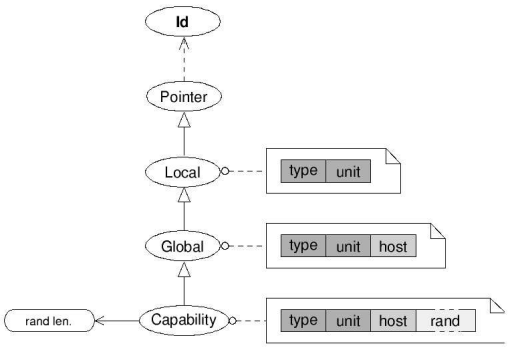

Figura 2.11: Família de Identificação de Objetos do *EPOS* [16]

Por exemplo, o aspecto *Identification* de *EPOS* define como os objetos em *EPOS* são identificados. A família *ID* possui quatro membros:

- *Pointer:* os objetos são referenciados pelo seu endereço no sistema. O objeto é referenciado apenas em um processo. Este é o modo mais simples de identificação, e que implica em menor sobrecarga.

- *Local:* os objetos são referenciados pela dupla <classe, unidade>. O objeto é refenciado em apenas um *host*, mas por qualquer processo.

- *Global:* os objetos são referenciados pela tripla <host, classe, unidade>. O objeto é referenciado em mais de um *host*.

- *Capability:* O sistema associa um número aleatório a cada identificador para evitar que referências a objetos já destruídos acessem outros objetos do sistema. *EPOS* possui um contador de referências para impedir a destruição de objetos ainda referenciados, mas este membro provê maior segurança contra possíveis erros de programação.

O membro adequado para a família *ID* pode ser determinado com base nas funções de *ROI* (Remote Object Invocation) que são chamadas. O uso de *Capability* é facultativo, e portanto o usuário deve informar o sistema.

O conceito de *Aspectos de Cenários* baseia-se em *Programação Orientada a Aspectos* [32]), mas é implementado através de metaprogramação estática em C++,

dispensando o uso de um *weaver*.

2.3.4 Mediadores do Sistema

Para que o sistema seja portável, é fundamental que todo o código dependente de dispositivos seja separado do restante. A abstração de gerência de memória é genérica, sendo válida para qualquer arquitetura. Entretanto, a *MMU* (*Memory Management Unit*) é representada por uma família de mediadores onde cada membro representa uma arquitetura diferente. Esta separação clara permite que *EPOS* seja executado tanto na arquitetura *IA32* quanto em processadores de 8 *bits* para sistemas embutidos sem qualquer alteração no código, exceto em mediadores.

2.3.5 Interfaces Infladas

Para evitar dependências de membros específicos entre famílias (quando possível), as famílias devem ser referenciadas através de sua interface comum. As interfaces das famílias são organizadas em 4 tipos:

- *Uniforme:* Todos os membros apresentam a mesma interface. Não há qualquer restrição quanto a escolha do membro.

- *Incremental:* Cada membro é um aperfeiçoamento de outro. Através dos métodos chamados, é possível determinar o nível de funcionalidade mínima necessária, e quais membros a fornecem.

- *Combinada:* Cada membro oferece um conjunto de funções, sem repetições. A combinação de membros necessária é determinada pelos métodos referenciados, e obtida através de herança múltipla.

- *Dissociada:* Famílias que não se enquadram em nenhum dos outros casos.

As interfaces infladas de cada família são descritas em um arquivo *XML*, onde o *custo* de cada membro é relacionado. Quando mais de um membro atende às

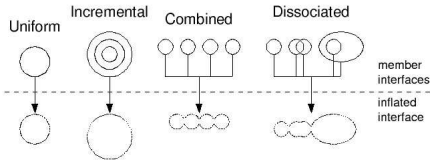

Figura 2.12: Interfaces infladas do *EPOS* [16]

exigências da aplicação, o de menor custo é selecionado. Assim, *EPOS* é configurável que sistema baseados em módulos, como por exemplo o *Linux*.

Através do *script* que analisa a aplicação, é possível determinar grande parte da configuração do sistema automaticamente. Poucas opções precisam ser informadas pelo usuário (sendo a maioria otimizações opcionais). Este mecanismo diminui a complexidade de sistemas desenvolvidos com base em *AOOS*·

2.4 O Sistema de Comunicação do EPOS

O sistema de comunicação do *EPOS* é composto por quatro entidades: *Communicator*, *Channel*, *Network* e *Envelope*. Os processos comunicam-se entre si através do *Communicator*, que aje como uma interface para um canal (*Channel*) de comunicação implementado sobre uma rede (*Network*). As mensagens enviadas por um *Communicator* podem ser especificadas como seqüências de *bytes* de tamanho conhecido ou podem ser cobertas por um *envelope*.

Channel gerencia detalhes de transporte independentes do tipo de rede (camada OSI 4). Detalhes dependentes do tipo de rede são gerenciados por *Network*, que provê uma visão uniforme de redes para *Channel*. A abstração *Envelope* representa as mensagens, lida com tipagens de dados e também é responsável por transferências com cópia zero.

As aplicações interagem apenas com as abstrações *Communicator* e *Envelope*. Nas operações de envio, *Envelopes* são inicializados com o conteúdo da mensagem a ser transferida e são enviados a um *Communicator*.

2.5 A Interface Padrão de Troca de Mensagens

2.5.1 O Padrão

A *MPI* [10] é uma interface padrão de troca de mensagens estabelecida pelo *Fórum da MPI* , composto pela maioria dos fabricantes de computadores paralelos e pesquisadores de universidades, laboratórios mantidos por governos e da indústria.

O *Fórum da MPI* [10] descreve seus objetivos como:

- Projetar uma interface para programação de aplicações (API);
- Permitir comunicação eficiente: evitar cópia de memória para memória e permitir concorrência entre processamento e comunicação e transmissão para o co-processador de comunicação, quando disponível;
- Permitir que implementações sejam usadas em sistemas heterogêneos;
- Permitir ligações C e Fortran 77 convenientes para a interface;
- A interface de comunicação deve ser confiável: o programador não precisa lidar com falhas de transmissão. Tais falhas são gerenciadas pelo subsistema de comunicação;
- Definir uma interface que não seja tão diferente da prática atual, como *PVM*, *NX*, *Express*, *p4*, etc., e forneça extensões que ofereçam maior flexibilidade;
- Definir uma interface que possa ser implementada em plataformas de muitos fornecedores, sem mudanças significativas no sistema comunicação e no *software* do sistema;
- As semânticas da linguagem devem ser independentes de linguagem;
- A interface deve ser projetada de modo a permitir segurança entre *threads*.

É importante citar que a *MPI* foi definida para ser a base de outras bibliotecas de programação paralela, estas podendo ser de maior nível, empregando outro

paradigma de comunicação ou serem voltadas a domínios específicos. Sua maior vantagem é o fato de ser o padrão de fato da indústria, e praticamente qualquer ambiente de execução paralela possui uma implementação. Portanto, programas e bibliotecas baseados em *MPI* são altamente portáveis. MPI freqüentemente é a interface de mais alto nível oferecida pelo desenvolvedor: um contrato entre desenvolvedores de sistemas de comunicação e desenvolvedores de bibliotecas ou aplicações paralelas.

Um bom exemplo destas bibliotecas é *Janus* [21], que é composta por um conjunto de estruturas de dados e algoritmos úteis em um a ampla gama de aplicações de dados paraelos. Através de sua estrutura de mediadores, *Janus* pode ser implementada sobre *MPI* ou interfaces nativas do sistema. Mapear *Janus* sobre uma implementação convencional de *MPI* (*MPICH*) implica em 10% de sobrecarga adicional. Em troca a *MPI* oferece grande portabilidade. Desta forma, *Janus* pode ser executado com máxima eficiência em suas arquiteturas nativas, mas também pode ser executado com eficiência aceitável em praticamente qualquer outra arquitetura.

A interface *MPI* foi padronizada em 1994 (versão 1.0), sendo revisada em 1995 (versão 1.1) e novamente em 1997 (versão 2.0). Juntamente com a versão 2.0 foi lançada a 1.2, que trazia apenas pequenas correções em relação a 1.1, e deveria ser seguida durante a migração para a versão 2.0. Tal migração foi longa, e até a data de escrita deste documento as implementações livres da *MPI* ainda não estão em total conformidade com o padrão 2.0. A implementação proposta neste texto não visa adequar-se à versão 2.0, e portanto quando o termo *MPI* for usado adiante, a versão 1.2 estará sendo referenciada.

A seguir, os recursos que a *MPI* oferece serão descritos. Esta descrição é bastante sucinta, e concentra-se em demonstrar as *semelhanças* entre funções similares e os *aspectos* que as diferenciam. O padrão é completamente descrito em [10].

2.5.1.1 Comunicação Ponto-a-Ponto

Em *MPI* existem quatro modos básicos de comunicação:

1. *Buffered(B):* Caso a função de recebimento ainda não houver sido chamada, os dados devem ser armazenados em um buffer do sistema e a função de envio é com-

pletada. O fato da função de envio completar não significa que os dados já tenham sido recebidos. Este modo implica em menor latência, sendo recomendado para pequenas mensagens, mas necessita de cópia de memória, não sendo indicado para grandes mensagens;

2. *Synchronous(S):* A função de envio será completada apenas quando a função de recebimento já houver sido chamada. Assim, o fato da função de envio ser completada é uma evidência de que o recebimento já começou. Esta semântica é geralmente conhecida como *Rendezvous*. Este modo jamais implica em cópia de memória, mas as comunicações apresentam maior latência, sendo pouco indicado para pequenas mensagens;

3. *Ready(R):* Neste modo o programador garante à implementação que a função de recebimento já foi chamada antes da função de envio. Este modo permite otimizações em certas arquiteturas, porém se a função de recebimento não houver sido chamada, o comportamento do programa é indefinido;

4. *Standard:* Este modo pode ser mapeado para *Buffered* ou *Synchronous*, dependendo da implementação. Quando a função for completada, a função de envio pode ter sido chamada ou não.

Para cada modo, existe uma função que faz comunicação *Bloqueante* e *Imediata(I)*. As funções *Bloqueantes* apenas retornam quando a função for completada e as *Imediatas* retornam imediatamente. O padrão *MPI* fornece funções para testar se uma comunicação imediata já foi completada ou aguardar até que isto ocorra.

As funções *Imediatas* são mais complexas, mas permitem que computações ocorram durante a comunicação.

Para o recebimento de dados, existem apenas uma função bloqueante e uma imediata. Qualquer uma das duas pode receber de qualquer modo de operação, e funções bloqueantes podem receber dados de funções imediatas, e vice-versa.

As funções básicas da *MPI* para comunicação ponto-a-ponto estão listadas na tabela 2.2.

Função	Características
MPI_SEND	Envio Standard Bloqueante
MPI_BSEND	Envio Buffered Bloqueante
MPI_SSEND	Envio Synchronous Bloqueante
MPI_RSEND	Envio Ready Bloqueante
MPI_ISEND	Envio Standard Imediato
MPI_IBSEND	Envio Buffered Imediato
MPI_ISSEND	Envio Synchronous Imediato
MPI_IRSEND	Envio Ready Imediato
MPI_RECV	Recebimento Bloqueante
MPI_IRECV	Recebimento Imediato

Tabela 2.2: Funções para comunicação ponto-a-ponto da *MPI*

2.5.1.2 Tipos de Dados Definidos pelo Usuário

O padrão *MPI* permite que o programador especifique o tipo de dado que está sendo enviado em função de comunicação, caso queira. Existem duas vantagens em especificar o tipo do dado:

1. A implementação encarrega-se de calcular o tamanho do vetor de dados com base no número de registros;

2. Em ambientes heterogêneos, a implementação converte os dados para a representação adequada.

O padrão especifica os tipos básicos suportados, que correspondem aos tipos básicos das linguagens de programação. Entretanto, existem funções que permitem ao usuário definir novos tipos como seqüências de tipos pré-definidos ou definidos anteriormente. Estas seqüencias podem ser homogêneas (todos os elementos são do mesmo tipo) ou heterogêneas. Podem também ser contínuas (sem espaçamentos entre os elementos) ou descontínuas. As descontínuas são organizadas em blocos, e o tamanho destes

Função	Homogênea	Contínua	Fixos	Elementos
MPI_TYPE_CONTIGUOUS	sim	sim	-	-
MPI_TYPE_VECTOR	sim	não	sim	sim
MPI_TYPE_HVECTOR	sim	não	sim	não
MPI_TYPE_INDEXED	sim	não	não	sim
MPI_TYPE_HINDEXED	sim	não	não	não
MPI_TYPE_STRUCT	não	não	não	não

Tabela 2.3: Funções para definições de novos tipos da *MPI*

blocos e o espaçamentos entre eles podem ser fixos para todos os blocos ou não. O espaçamento pode ser definido por número de elementos (ser for homogênea) ou por número de *bytes*. As funções para declaração de tipos da *MPI* são combinações destas características. Estas funções estão listadas na tabela 2.3. Note que a função MPI_TYPE_STRUCT é a mais geral e todas as outras são casos particulares dela que permitem otimizações.

Os tipos de dados descontínuos são úteis para que apenas partes de uma estrutura complexa de dados sejam enviados pela rede, como apenas as colunas de uma matriz organizada em linhas. A *MPI* também oferece funções de agrupamento para compactar estruturas de dados esparsas.

2.5.1.3 Comunicação Coletiva

Um recurso muito interessante da *MPI* é o seu conjunto de *operações coletivas*. Operações coletivas são operações que envolvem um *grupo* de processos, ao invés de apenas dois.

Basicamente, as funções coletivas da *MPI* enquadram-se em três categorias:

1. Funções que espalham dados;

2. Funções que recolhem dados;

3. Combinações das anteriores.

Como exemplos básicos das duas primeiras categorias temos as funções *GATHER* que recolhe dados de todos os processos e os reúne em um *buffer* em um processo, e *SCATTER*, que divide um *buffer* pelo número de processos e envia cada pedaço para o processo correspondente. Como um caso particular de *SCATTER*, temos *BROADCAST*, que seria o caso em que os dados a serem enviados para todos os processos são iguais (embora seja implementado de forma diferente para alcançar maior desempenho). A função *ALLGATHER* representa um *GATHER* seguido de um *BROADCAST* e a função *ALLTOALL* representa todos os processos enviando e recebendo dados de cada outro processo (e portanto pode ser visto como uma combinação de *SCATTER* e *GATHER*).

Todas as operações são representadas por duas funções: Uma que utiliza *buffers* de mesmo tamanho para todos os processos e outra que utiliza diferentes.

A *MPI* também possui funções de *Redução Global*. Tais funções são baseadas nas operações coletivas básicas, mas ao invés de concatenarem os dados no *buffer* de destino, combinam os valores conforme a *operação de redução* especificada, e retornam apenas um valor. O padrão oferece uma função de redução global que retorna para um processo o resultado da combinação de dados de todos os processos, outra que envia o resultado para todos os processos e outra que executa um número de operações simultâneas equivalente ao número de processos, e envia o resultado de uma operação para cada processo. Há ainda uma função que envia para cada processo *N* os resultados das operações entre os processos *0..N*.

As operações de redução global podem ser vistas como versões extendidas das operações coletivas básicas. Ao invés do dado ser apenas repassado, ele é tratado. Este *aspecto* é a única diferença entre as duas classes de operações.

Por último, o padrão oferece uma função de sincronização. Tal função bloqueia a execução do processo até que todos os processos tenham chegado àquele ponto. Como consiste em reunir uma informação e espalhá-la para todos os processos, pode ser vista como um caso específico de *ALLGATHER*. Mas otimizações em hardware podem ser feitas.

2.5.1.4 Grupos de Processos e Topologias

As operações coletivas da *MPI* sempre envolvem todos os processos de um *grupo*. Na inicialização, as implementações da *MPI* geram um grupo contendo todos os processos. Para evitar desperdício de comunicação, o padrão oferece funções para que o programador defina novos grupos formados que sejam um subconjunto de um grupo já existente ou a união de dois.

Este recurso é especialmente útil em redes geograficamente distribuídas, por haver grandes disparidades nas velocidades de transmissão entre os processos.

2.5.1.5 Recursos Auxiliares

A funcionalidade da *MPI* resume-se em comunicação ponto-a-ponto e operações coletivas. Os tipos de dados definidos pelo usuário e a gerências de grupos permitem otimizações em diversos casos. O padrão *MPI* é basicamente formado pelo que foi exposto neste capítulo.

O padrão oferece também funções para gerenciar a alocação e gerência dos *buffers* usados no modo de comunicação *Buffered* e um sistema de gerência de erros baseado em tratadores definidos pelo usuário.

2.5.2 Protocolos de Comunicação

As implementações convencionais da MPI costumam utilizar três diferentes protocolos para comunicação ponto a ponto: *Short*, *Eager* e *Rendezvous*. No protocolo *Short* os dados são enviados juntamente com o cabeçalho da mensagem, no protocolo *Eager* os dados são enviados logo após o cabeçalho e no protocolo *Rendezvous* os dados são retidos no remetente até que o destinatário esteja pronto para o recebimento.

O protocolo *Short* tem a menor latência, e portanto é adequado a pequenas mensagens. O protocolo Rendezvous evita uma cópia de memória no destinatário, e é adequado a longas mensagens, pois a maior taxa de transferência e a menor utilização

da memória compensa o aumento da latência. Estes protocolos são descritos em detalhes em [25].

O protocolo a ser utilizado em cada transferência depende do tamanho da mensagem. Os limites de tamanho podem ser definidos de forma estática baseada em estatísticas de execuções anteriores ou dinamicamente baseado em testes de inicialização, como descrito em [50].

2.5.3 MPI com Cópia Zero

O'Carrol demonstrou em [37] que as operações ponto a ponto da MPI podem ser implementadas de forma a evitar cópias de memória. Esta abordagem melhora o desempenho na transferência de mensagens longas, reduzindo também a utilização da memória. O princípio da cópia zero é utilizar o protocolo *Rendezvous*, mas para evitar *deadlock* a utilização da memória da interface de rede deve ser cuidadosa, de forma a não impedir que alguma transferência aconteça.

Duas filas são necessárias para o recebimento de mensagens: *expected* e *unexpected*. A fila *expected* contém a lista de mensagens que são esperadas, ou seja, que o usuário requisitou através de um método de recebimento. As mensagens contém o cabeçalho que as identifica e o *buffer* aonde devem ser armazenadas. A fila *unexpected* contém as mensagens que chegaram pela rede, mas que o usuário ainda não requisitou. Quando uma mensagem chega pela rede, o sistema verifica se há um cabeçalho equivalente em *expected*. Caso haja, o conteúdo é copiado e a comunicação é completada. Caso contrário, o conteúdo é armazenado em um *buffer* temporário e a mensagem é armazenada em *unexpected*.

Se *rendezvous* for usado, duas filas também são necessárias para o envio: *requested* e *unrequested*. A fila *requested* armazena as mensagens que devem ser enviadas imediatamente, pois uma função de recebimento já foi chamada no processo destinatário. A fila *unrequested* armazena as mensagens que estão esperando um requerimento do destinatário para serem enviadas. Quando uma mensagem *rendezvous* vai ser enviada, o sistema procura por um requerimento em *requested*. Caso encontre, en-

via a mensagem imediatamente. Caso contrário, armazena a mensagem em *unrequested*. Quando um requerimento chega pela rede, o sistema verifica se a mensagem já está em *unrequested*. Em caso positivo, envia imediatamente. Caso contrário, armazena o requerimento em *requested*.

2.5.4 Implementações Adaptativas

Vadhiyar e co-autores [50] obteram ganhos de 30%-650% no desempenho de operações coletivas ao implementar operações coletivas adaptativas. O algoritmo adequado para cada operação é definido através de testes de inicialização que identificam o algoritmo mais eficiente para a topologia física.

Ao analisar a topologia da aplicação, as operações coletivas devem ser analisadas separadamente. Cada operação tem topologia própria, que independe do restante da aplicação [1]. Assim, o algoritmo apropriado para cada operação depende apenas da infra-estrutura de rede e da topologia física. O algoritmo está relacionado ao sistema, e não à aplicação. Assim sendo, o ganho obtido por Vadhiyar advém do fato de que a maioria das implementações genéricas de *MPI* ignoram a topologia da rede. Por exemplo a implementação *MPICH*, descrita na seção a seguir, utiliza algoritmos hipercubo. É possível alterar estes algoritmos reimplementando as operações coletivas, porém devido a complexidade envolvida a maioria dos agregados utilizam os algoritmos padrão.

2.5.5 MPICH

A seguir, será apresentada a estrutura de uma popular implementação de *MPI*, a *MPICH* [24], que serve de referência para a implementação proposta.

A implementação *MPICH* foi escolhida como referência por três motivos:

- é a mais usada em projetos de pesquisa;

[1] Em operações que envolvem apenas parte da rede, levar em conta as operações envolvendo outros nodos pode envolver grande complexidade.

- tem por objetivo: portabilidade e flexibilidade sem comprometimento significativo de desempenho, e portanto também é organizada de forma modular e adaptável;

- possui suporte às redes Myrinet [3], que equipam o agregado *SNOW* do Laboratório de Bioinformática da UFSC, no qual são feitas as comparações de desempenho.

2.5.5.1 Arquitetura da MPICH

A *MPICH* é organizada em duas camadas. A camada superior é *independente de arquitetura*, enquanto a inferior, conhecida como ADI é dependente do sistema operacional e do dispositivo de comunicação. A *ADI* é responsável por toda a comunicação ponto a ponto, em todos os modos que a *MPI* prevê, de forma bloqueante ou imediata. Operações coletivas, gerência de tipos, erros e sessão, entre outros recursos, são independentes de arquitetura.

Esta abordagem visa oferecer maior portabilidade. A camada superior seria suportada pelos desenvolvedores da *MPICH*, e a camada dependente de dispositivo seria fornecida pelo fabricante do dispositivo de comunicação. A figura 2.13, retirada de [24] demonstra esta relação usando a função MPI_Reduce como exemplo.

Os desenvolvedores da *MPICH* também estabeleceram uma subcamada da *ADI* conhecida como *interface de canal*. Esta interface exige que apenas as funções básicas de envio e recebimento ponto a ponto sejam implementadas, e toda a funcionalidade restante da *ADI* é genérica. Assim, fornecedores que desejem adicionar suporte apenas precisam implementar a *interface de canal*, e em versões posteriores modificar a *ADI* visando otimizações.

A *MPICH* é a base de muitas implementações. A Myricom, fabricante das redes Myrinet [3] usa uma adaptação da *MPICH* baseada em seu *driver GM* para fornecer *MPI* a seus usuários.

Funções da camada independente de arquitetura (como as operações coletivas) podem ser modificadas pelos fornecedores para obter melhor desempenho, pois são implementadas como símbolos fracos (`#pragma`). Isto freqüentemente ocorre, e portanto os fornecedores publicam uma distribuição completa da *MPICH* a cada nova versão.

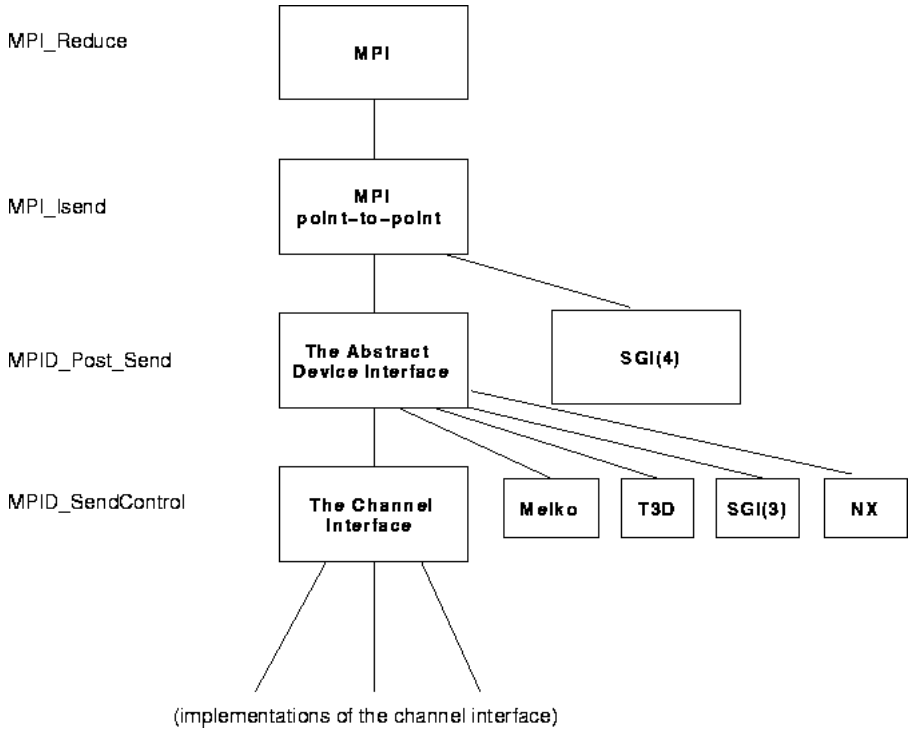

Figura 2.13: Arquitetura da MPICH (MPI_Reduce)

Este sistema traz uma desvantagem: os fornecedores precisam continuamente manter suas distribuições, mesmo que não façam qualquer alteração, para que seus usuários possam utilizar os recursos das novas versões da *MPICH*.

Como as seções mantidas pelos fornecedores e as originais não são claramente separadas (como em um modelo orientado a objetos) e a a *interface de canal*, a *ADI* e a camada superior compartilham as mesmas estruturas de dados, modificações significativas na distribuição original exigem grande esforço de adaptação por parte dos fornecedores, e aumentam a possibilidade de ocorrência de erros de programação.

Recursos opcionais e relacionados a cenário que afetam significativamente o desempenho (como heterogeneidade do ambiente) são definidos como diretivas de compilação, que habilitam ou não o código relacionado.

A *MPICH* é organizada como uma biblioteca de rotinas, e portanto apenas as rotinas chamadas (e suas dependências) são ligadas ao código. Entretanto, funcionalidades necessárias pelas rotina chamadas (como gerenciamento de tipos de dados

definidos pelo usuário) são sempre incluídos completamente, mesmo que o usuário não tenha definido nenhum tipo, e use apenas os da *MPI*. A implementação proposta permite maior grau de configuração, conforme será demonstrado adiante.

Capítulo 3

Sistema de Comunicação de Alto Desempenho Baseado em Programação Genérica

A seção 2.4 descreveu o sistema de comunicação do EPOS, e seus quatro componentes (`Communicator`, `Channel`, `Network` e `Envelope`), conforme proposto por Fröhlich [16]. Entretanto, este sistema não possui as características necessárias para aliviar as camadas superiores da gerência de filas e obter cópia zero (seção 2.5.3). Por isso, o sistema foi modificado, de forma que o sistema de comunicação possa manipular o cabeçalho da camada superior. De fato, ele passa a ser um atributo da abstração `Envelope` de EPOS, conforme descrito nas seções a seguir.

3.1 A Abstração Cabeçalho

O sistema de comunicação exige que uma classe que represente o cabeçalho da camada superior seja definida, e que ela realize a interface (classe abstrata) `Header`. Esta interface requer que os operadores `==`, `<` e `=` sejam implementados. O operador `==` define se dois cabeçalhos são iguais ou diferentes, e é usado para definir se as mensagens que chegam pela rede são esperadas ou não. O operador `<` define como as

filas do sistema de comunicação devem ser ordenadas, de forma a obter maior eficiência. Por último, o operador = define como copiar cabeçalhos.

Também é exigido que a classe seja contínua. Classes contínuas podem ser duplicadas com simples cópias de memória, e portanto são mais eficientes. Esta restrição apenas implica que nenhuma classe de cabeçalho pode conter atributos ponteiros ou referências. Geralmente, cabeçalhos são contínuos, e portanto esta exigência dificilmente causa alguma restrição.

3.2 O Envelope Parametrizado

Um cabeçalho identifica uma mensagem, e portanto é um atributo dela. A abstração que representa mensagem em EPOS é a `Envelope`. Portanto, envelope tem um atributo chamado `header` que representa seu cabeçalho. A classe de `header` deve ser genérica, de forma que possa representar o cabeçalho de qualquer protocolo que possa ser usado.

Para obter este comportamento, poder-se-ia declarar `header` como uma instância da classe abstrata `Header` e definir a classe do cabeçalho na instanciação. Mas este procedimento implicaria no uso de métodos virtuais, o que prejudicaria o desempenho do sistema. Por isso, ao invés de polimorfismo, será usada programação genérica (seção 2.1.1) através de *templates*. A classe de `header` é um parâmetro da classe `Envelope`. Desta forma, o sistema de comunicação é genérico e pode suportar vários protocolos, mas o código gerado é idêntico ao obtido se a classe fosse definida normalmente. Portanto, o desempenho do sistema não é alterado.

Para definir um envelope que suporte determinado tipo de cabeçalho, basta instanciar o *template*, como no exemplo abaixo, no qual a classe `mpi_envelope` representa as mensagens da MPI:

```
typedef Envelope<mpi_header> mpi_envelope;
```

A classe base do envelope parametrizado depende do fato da rede ser heterogênea ou não. Caso a rede seja heterogênea, o aspecto *heterogeneous* de *EPOS* será

habilitado. Neste caso, o cabeçalho parametrizado deriva do envelope tipado (*Typed*). Caso contrário, ele deriva do envelope não tipado (*Untyped*). Desta forma, a heterogeneidade da rede é gerenciada pelo sistema de comunicação.

A classe `Envelope` possui os seguintes atributos:

- `header`: o cabeçalho da mensagem;
- `buf`: endereço com o conteúdo da mensagem;
- `len`: tamanho da mensagem, sem considerar o cabeçalho;
- `node`: destino (se envio) ou origem (se recebimento) da mensagem;
- `rendezvous`: indica se *rendezvous* deve ser usado ou não;
- `complete`: indica se a operação já foi completada ou não, útil em operações imediatas.

As operações de comunicação consistem em instanciar um `Envelope`, definir seus atributos e repassá-lo ao `Communicator` através dos operadores `<<` (envio) e `>>` (recebimento). As operações de envio são imediatas, e retornam imediatamente. A camada superior pode verificar se a operação já foi completada através do atributo `complete`.

O uso ou não de comunicação *rendezvous* deve ser definido pela camada superior, pois cada protocolo tem uma política diferente. Mas cabe ao `Communicator`, que gerencia as filas, realizar a comunicação. Por isso, `Envelope` possui um atributo que define se este tipo de comunicação é desejada ou não. Em caso de omissão, o atributo é definido como `false`, para que sistema que nunca utilizem *rendezvous* possam simplesmente ignorar o atributo.

3.3 O Comunicador Parametrizado

A gerência das filas de mensagens é responsabilidade do `Communicator`. As filas são conjuntos de envelopes parametrizados identifica-

dos e ordenados por seus cabeçalhos. Quatro filas são necessárias, como descrito na seção 2.5.3: *expected*, *unexpected*, *requested* e *unrequested*. Estas filas são da classe `Envelope_queue`, que é parametrizada pela classe do cabeçalho a ser utilizado.

As filas são atributos do `Communicator`, e portanto ele também precisa ser parametrizado pela classe do cabeçalho. Esta característica impõe uma restrição: o sistema suporta apenas um protocolo. Caso mais de um protocolo seja necessário, o usuário pode desabilitar o suporte a filas e gerenciá-las na camada superior, como nos sistemas de comunicação tradicionais. Esta restrição raramente afeta sistemas de alto desempenho, aonde apenas um protocolo (geralmente MPI) é usado a cada vez.

Parametrizar o `Communicator` traz uma grande vantagem: como o cabeçalho é conhecido, o sistema de comunicação pode gerenciar as filas, aliviando as camadas superiores. Várias funcionalidades eram suportadas nas camadas superiores por dependerem das filas, e também podem ser repassadas para o sistema. Uma destas funcionalidades é a comunicação imediata. O recebimento imediato consiste em registrar o cabeçalho da mensagem em *expected*, e o envio imediato consiste registrar a mensagem na fila *unrequested* caso ela não possa ser enviada. Cancelar uma mensagem consiste apenas em removê-la das filas do sistema. Estas operações tornam-se responsabilidade do `Communicator`.

A classe `Communicator` possui três métodos:

- `<<`: envia uma mensagem;
- `>>`: recebe uma mensagem;
- `check_messages`: verifica se alguma mensagem chegou no dispositivo de rede.

As operações de recebimento e envio têm apenas um parâmetro: um `Envelope`. No caso do recebimento, o cabeçalho do `Envelope` é comparado aos das mensagens nas filas, e quando uma mensagem *igual* (`==`) é encontrada, seu conteúdo é armazenado no *buffer* indicado no `Envelope`. A função `check_messages` é utilizada para completar operações imediatas: ela verifica se alguma mensagem chegou pelo dis-

positivo de rede. Em caso positivo, procura por um cabeçalho igual na fila *expected*. Caso encontre completa a mensagem, e caso contrário registra a mensagem na fila *unexpected*.

3.4 Suporte ao Protocolo Rendezvous

O protocolo *Rendezvous* é suportado no sistema *EPOS* através do aspecto de cenário *Synchronous*. Caso este aspecto seja habilitado, o envio de mensagens utiliza um sistema de filas semelhante ao utilizado para o recebimento de mensagens. Duas filas são criadas: *requested* e *unrequested*. *Requested* armazena as requisições de mensagem cuja função de recebimento já foi executada, e portanto devem ser enviados imediatamente. Caso a requisição ainda não tenha sido recebida a mensagem a ser enviada é armazenada na fila *unrequested*. Caso o aspecto *Rendezvous* seja habilitado, a classe *Envelope* conterá um atributo extra que define se a mensagem deve ou não utilizar *Rendezvous*.

3.5 Envio de Mensagens

Quando uma mensagem deve ser enviada, a camada superior instancia e inicializa um envelope e seu cabeçalho e o envia ao `Communicator` do *EPOS* através do operador `<<`. O envelope é a interface entre o sistema de comunicação e as camadas superiores,

Caso o atributo `rendezvous` do envelope tenha valor `true`, este tipo de comunicação será usada. Neste caso, o *Comunicador* procura por uma requisição pela mensagem na fila *requested*. Caso encontre, envia a mensagem imediatamente. Caso não encontre, a mensagem é armazenada na fila *unrequested*. Quando uma requisição chega, o *Comunicador* procura pela mensagem em questão na fila *unrequested*. Caso encontre, envia a mensagem e completa a operação. Caso contrário, a requisição é armazenada na fila *requested*.

Caso o protocolo *rendezvous* não seja utilizado, a mensagem é enviada imediatamente ou armazenada em um *buffer* do sistema para envio posterior. Ao término

da comunicação, o atributo `complete` do envelope é definido como `true`, sinalizando que a mensagem foi completada.

3.6 Recebimento de Mensagens

O recebimento de mensagens é análogo ao envio de mensagens com *rendezvous*. A camada superior instancia e inicializa um `Envelope` com o cabeçalho da mensagem esperada, o *buffer* no qual ela deve ser armazenada. Em seguida, passa o `Envelope` ao `Communicator` através do operador `>>`. O `Communicator` verifica se há alguma mensagem na fila *unexpected* igual a esperada. Caso haja completa a mensagem, e caso contrário a registra na fila *expected*. O operador retorna imediatamente. A camada superior pode verificar se a operação foi completada através do atributo `complete` do envelope. Ao final da operação, o atributo `node` indica a origem da mensagem.

3.7 A Abstração Network

O recebimento de mensagens que chegam pela rede precisa ser feito em duas etapas: extração do cabeçalho e transferência do conteúdo. A primeira etapa é necessária para determinar o destino da transferência. Caso haja um cabeçalho igual na fila *expected*, o destino dos dados será o *buffer* indicado pela mensagem na fila. Caso contrário, o conteúdo será transferido para um *buffer* temporário. O cabeçalho define o destino da transferência, e portanto ele deve ser extraído antes que ela comece.

Entretanto, a abstração `Network` do EPOS conforme proposta por Fröhlich [16] não suporta o recebimento parcial de mensagens. Assim, é necessária uma modificação nesta abstração. O método `receive` passa a ter um quarto parâmetro, que indica a partir de qual *byte* deseja-se transferir a mensagem. Para que a classe `Network` seja compatível com código escrito antes da alteração, o parâmetro será opcional e terá valor 0 em caso de omissão, de forma que possa ser ignorado. Com esta mudança, a assinatura do método `receive` passa a ser:

```
int receive(Node_Id * s, void * b, unsigned int * l,
unsigned int const &offset);
```

Com esta mudança, o sistema de comunicação pode suportar o envelope parametrizado com cópia zero. Ao receber uma mensagem, o método `receive` é chamado com `offset=0` e `l=sizeof(header_t)` para extrair o cabeçalho, e após determinar o destino do conteúdo o método é chamado com `offset=sizeof(header_t)`, para ignorar o cabeçalho na transferência. Sem esta alteração, o conteúdo precisaria ser extraído junto com o cabeçalho para um *buffer* temporário, e uma cópia extra seria necessária.

3.8 Exemplo

Para ilustrar o funcionamento do sistema de comunicação é apresentado um exemplo de uso. No código abaixo, um processo envia uma mensagem para outro. O cabeçalho utilizado é o da MPI (`mpi_header`). No exemplo, apenas uma mensagem é enviada, mas caso várias mensagens fossem enviadas fora de ordem, a comparação entre os cabeçalhos garantiria que a ordem de recebimento seja a esperada.

```
typedef mpi_header header_t;
typedef Envelope<header_t> message_t;
typedef Communicator<header_t> communicator_t;

int main(int argc, char **argv) {
  int rank(atoi(argv[2]));
  communicator_t comm(rank);
  message_t message;
  char s[10];

  message.header.context = 1;
```

```
  message.header.tag = 2;
  message.header.src = 0;
  message.header.dest = 1;
  message.buf = s;
  message.len = 10;

  if (rank==0) {
    message.node = 1;
    strcpy(s,"mensagem\n");
    comm << message;
  } else {
    message.node = 0;
    comm >> message;
    while (!message.complete)
      comm.check_messages();
    std::cout << s;
  }

  return 0;
}
```

Capítulo 4

Implementação de MPI para EPOS

O capítulo anterior descreveu um sistema de comunicação baseado em programação genérica, e como é possível aliviar as camadas superiores da gerência das filas. Para validar o sistema, uma implementação de MPI foi feita sobre o sistema, tirando vantagem de suas características. Este capítulo descreve os detalhes da MPI, e demonstra que praticamente todas a funcionalidade de comunicação ponto a ponto da MPI pode ser repassada para o sistema. De fato, a implementação foi tão menor e mais simples que as convencionais [24], que desenvolvê-la por completo exigiu menos esforço do que adaptar outra implementação.

Este capítulo está organizado em 5 seções. Na seção 4.1 é feita uma breve análise do padrão MPI. Na seção 4.2 a implementação das operações ponto a ponto sobre o sistema de comunicação proposto é descrita. Na seção 4.3 os mecanismos para fornecer suporte aos tipos de dados específicos da MPI. A seção 4.4 descreve os mecanismos utilizados para implementar as operações coletivas, que não fazem parte do sistema de comunicação. Por último, a seção 4.5 faz uma comparação de desempenho entre um protótipo da implementação proposta e uma implementação tradicional.

4.1 Análise da MPI

As funcionalidades da *MPI* podem ser organizadas em 3 unidades:

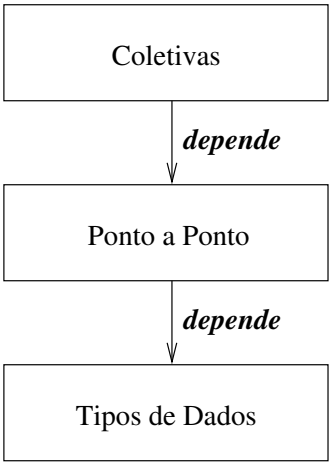

Figura 4.1: Dependências entre as unidades funcionais

1. Comunicação Ponto a Ponto
2. Gerência de Tipos de Dados
3. Comunicação Coletiva

Há relações de dependência entre as 3 unidades:

- a comunicação coletiva necessita da comunicação ponto a ponto e;
- a comunicação ponto a ponto exige suporte a todos os tipos de dados utilizados pelo usuário, pré-definidos ou não.

As seções a seguir descrevem as unidades funcionais e sua integração com o sistema de comunicação baseado em programação genérica.

4.2 Comunicação Ponto a Ponto

A comunicação ponto a ponto é baseada no sistema descrito no capítulo 3. As operações coletivas são baseadas na comunicação ponto a ponto da MPI, e portanto com as camadas inferiores. A gerência de tipos de dados pode alterar o comportamento do sistema EPOS, pois determina algumas de suas configurações. Mas apenas esta unidade

interage de fato com o sistema. Esta seção descreve a implementação e demonstra como ela obtém proveito das características do sistema.

4.2.1 Modos de Envio de Mensagens

Conforme descrito na seção 2.5.1.1, a MPI oferece quatro funções para envio:

- *Ready:* é presumido que o destinatário da mensagem já tenha chamado a função de recebimento;
- *Buffered:* caso não seja possível enviar a mensagem imediatamente, ela deve ser armazenada pelo sistema e a mensagem deve retornar;
- *Synchronous:* retorna apenas quando a função de recebimento já houver sido chamada;
- *Standard:* pode comportar-se como *Buffered* ou como *Synchronous*.

Como foi visto na seção 2.5.3, quando uma das funções de envio é chamada os dados podem ser enviados imediatamente ou o sistema de comunicação pode aguardar até que a função de recebimento tenha sido chamada e os dados sejam requisitados pelo destinatário (*Rendezvous*). As diferenças entre as funções de comunicação resumem-se, em grande parte, ao uso ou não de *Rendezvous*. Caso a função *Ready* seja usada, presume-se que a função de recebimento já tenha sido chamada. Logo *Rendezvous* não é usado, mesmo que a mensagem seja longa. Caso a função *Synchronous* seja chamada, ela somente deverá retornar quando o recebimento houver sido iniciado. Logo, aguardar por uma mensagem do destinatário. Neste caso, *Rendezvous* é usado mesmo para pequenas mensagens. Nos modos *Buffered* e *Standard*, o uso de *Rendezvous* é decidido apenas pelo tamanho da mensagem. Mas no modo *Buffered*, caso *Rendezvous* seja usado a mensagem é armazena em um *buffer* do sistema e a função de envio retorna. Caso seja decidido que *rendezvous* deve ser usado, o atributo correspondente de `Envelope` é definido como `true`.

O *Comunicador* de *EPOS* oferece a opção de configuração *Buffering*. Esta opção será habilitada para dar suporte ao modo de comunicação *Buffered*. O protocolo *Rendezvous* também é suportado pelo sistema de comunicação, conforme descrito em detalhes na seção 3.4.

4.2.2 Identificação de Mensagens

O padrão *MPI* estabelece que quatro dados identificam uma mensagem:

1. contexto;
2. origem;
3. destino;
4. *tag*.

O contexto é definido pelo Comunicador da *MPI* utilizado, e a *tag* é um parâmetro da função de envio. Estas quatro informações compõem o *cabeçalho* de uma mensagem *MPI*, que é representado pela classe `!mpi_header!`. Esta classe realiza a interface `Header`, e portanto pode ser usado como parâmetro de classe para `Envelope`. O padrão especifica dois valores *curingas* para os atributos (`MPI_ANY_SOURCE` e `MPI_ANY_TAG`), que são levados em conta pelo operador de comparação (`==`). Através da classe `mpi_header`, o protocolo da MPI é suportado pelo sistema de comunicação baseado em programação genérica.

4.2.3 Envio de Mensagens

O envio de mensagens consiste em inicializar um `Envelope` e passá-lo ao `Communicator`, conforme descrito na seção 3.5. O operador `<<` pode ser usado para implementar operações imediatas, pois nunca bloqueia. O código abaixo demonstra a implementação da função `MPI_Send`, para mostrar a simplicidade da implementação:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
      int dest, int tag, MPI_Comm comm) {
  Envelope<mpi_header> message(
      mpi_header(comm, MPI_rank, dest, tag),
      buf, count, rank2node_id(dest));

  return ((*epos_comm) << message);
}
```

4.2.4 Recebimento de Mensagens

Conforme descrito na seção 3.6, a operação de recebimento consiste em inicializar um `Envelope` com um cabeçalho que identifique a mensagem esperada e o *buffer* de destino e passá-lo para o `Communicator` através do operador `>>`. Caso a operação seja imediata, o envelope será armazenado na fila *expected*, e será completada quando os dados chegarem pela rede e o método `check_messages` for chamado. As funções `MPI_Wait`, `MPI_Test` e similares testam o atributo `complete` do envelope para verificar se a operação já foi completada. Caso a operação seja bloqueante, a função `MPI_Wait` é chamada após o operador `>>` ser chamado. O código abaixo demonstra a implementação das funções `MPI_Recv` e `MPI_Wait`.

```
typedef henvelope<mpi_header>* MPI_Request;
const MPI_Request MPI_REQUEST_NULL = 0;

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

  message_t message(
```

```
      mpi_header(comm, source, MPI_rank, tag),
      buf, count, rank2node_id(source));

  MPI_Request request(&message);

  (*epos_comm) >> message;

  MPI_Wait(&request, status);

  return 0;
}

inline int MPI_Wait(MPI_Request *request,
    MPI_Status *status) {

  if (*request==MPI_REQUEST_NULL)
    return 0;
  // wait for the message
  while (!(*request)->complete) {
    epos_comm->check_messages();
  }
  set_status(status, *request, 0);
  free_request(*request);

  return 0;
}
```

Nas operações de recebimento pode-se perceber as vantagens do sistema de comunicação. Nas implementações tradicionais, a rotina de recebimento precisa

gerenciar as filas na rotina de recebimento. Nesta implementação ela define a identidade da mensagem, mas não deixa toda a comunicação para o sistema.

4.3 Gerência de Tipos de Dados

A gerência de tipos pode ser vista como uma funcionalidade auxiliar das comunicações ponto a ponto. Em cada comunicação, os dados informados são convertidos na forma conveniente para transmissão.

A gerência consiste em oferecer funções que permitam definir novos tipos, e oferecer tradução dos tipos definidos pelo usuário para os pré-definidos, e dos pré-definidos para o tipo mais básico, MPI_BYTE. As funções de definição de dados utilizadas definem qual a complexidade da tradução de tipos. Quatro níveis de conversão de tipos são considerados:

1. *Nenhuma:* apenas o tipo `MPI\_Byte` foi utilizado. A função `get_extent()` retorna sempre a constante 1 e descontinuidade não é suportada;

2. *Básica:* apenas os tipos pré-definidos são utilizados. O tamanho de cada tipo pré-definido é armazenado em um vetor de constantes indexado pelo identificador do tipo. A função `get_extent()` retorna o elemento correspondente do vetor. Descontinuidade também não é suportada neste nível;

3. *Contínua:* o usuário define tipos, mas apenas tipos contínuos através da função `MPI_Type_contiguous`. Naturalmente, descontinuidade não é suportada. A função `get_extent` retorna o produto do número de elementos do tipo pelo tamanho do tipo dos elementos;

4. *Total:* o usuário definiu tipos descontínuos. Descontinuidade será suportada, e a função `get_extent()` retornará a soma dos tamanhos dos campos que compõem os tipos.

Os tipos de dados são gerenciados em *EPOS* pela abstração envelope. As características dos tipos utilizados determinam a complexidade do envelope a ser uti-

lizado. Caso tipos descontínuos sejam utilizados, uma mensagem pode ser composta por uma coleção de *buffers*. É necessário, portanto, que o envelope possua um vetor para armazenar os *buffers* que o compõem. Caso apenas tipos contínuos sejam utilizados, um ponteiro substitui o vetor.

As características do tipo de dados também alteram a função que determina seu tamanho (*extent*). Caso tipos descontínuos sejam utilizados, o tamanho é a somatória dos tamanhos dos campos que o compõem o tipo. Caso apenas tipos contínuos sejam utilizados, o tamanho é o produto do número de elementos pelo tamanho de cada elemento. Caso apenas os tipos pré-definidos sejam utilizados a função se resume a verificar o tamanho do tipo utilizado em um vetor de constantes. Caso apenas o tipo `MPI_Byte` seja utilizado, o tamanho é constante e igual a 1.

Os tipos de dados influenciam alguns aspectos de configuração do EPOS. Caso a rede seja heterogênea e tipos pré-definidos diferentes de `MPI_Byte` sejam utilizados será necessário converter os dados para notação de rede. A abstração `Envelope` encarrega-se das conversões de forma transparente. Caso descontinuidade seja suportada, a classe envelope conterá um vetor para armazenar mais de um *buffer*. Caso contrário, apenas um ponteiro para um *buffer* de dados é suficiente. Caso o ambiente de execução seja heterogêneo, o aspecto de cenário *heterogeneous* de *EPOS* será habilitado, e os dados serão convertidos para notação de rede de forma transparente, utilizando envelopes *Typed*.

O sistema de gerência de tipos é configurado através de um *script* que analisa o código da aplicação. Para cada nível de complexidade definido acima, há um arquivo de cabeçalho com as definições necessárias para suportá-lo. O script verifica os símbolos da *MPI* referenciados pela aplicados e define qual o arquivo de cabeçalho adequado. Esta configuração não afeta outras unidades da implementação da *MPI*.

O *script* de configuração usado é listado a seguir:

```
TEST3_EXP="MPI_(((UNSIGNED)?CHAR|SHORT|LONG)|UNSIGNED) \
|INT|FLOAT|(LONG_)?DOUBLE|PACKED)"
```

```
TEST2_EXP="MPI_Type_contiguous"
TEST1_EXP="MPI_H?(VECTOR|INDEXED|STRUCT)"

rm -f ../Datatypes.h

(egrep -q $TEST1_EXP $1 && # test 1
  ln -s Datatypes/Total.h ../Datatypes.h) ||
  (egrep -q $TEST2_EXP $1 &&
    ln -s Datatypes/Contiguous.h ../Datatypes.h) ||
    (egrep -q $TEST3_EXP $1 &&
      ln -s Datatypes/Basic.h ../Datatypes.h) ||
      ln -s Datatypes/None.h ../Datatypes.h
```

4.4 Comunicação Coletiva

As operações coletivas da *MPI* são baseadas na comunicação ponto a ponto. A família de comunicação coletiva exige uma função ponto a ponto confiável e que possa alcançar qualquer nodo da rede. Não há outras restrições, e portanto a comunicação coletiva pouco afeta a configurabilidade da família de comunicação ponto a ponto.

Cada uma das topologias descritas anteriormente necessitam de algoritmos específicos de roteamento. Na topologia circular, por exemplo, o remetente avalia se deve enviar a mensagem ao antecessor ou ao sucessor, buscando o menor número de *hops*. Todo nodo que receber uma mensagem que não seja destinada a ele a repassa na mesma direção.

Por exemplo, na figura 4.2 o nodo 7 envia uma mensagem ao nodo 2, em uma rede homogênea com 8 nodos. Pela esquerda são necessários $7 - 2 = 5$ *hops* e pela direita serão necessários $2 - 7 + 8 = 3$ *hops*. Assim sendo, o nodo 7 enviará a mensagem para o nodo 8. O nodo 8 identifica que a mensagem não é destinada a ele, e a repassa também pela direita, para o nodo 1, que a repassará ao nodo 2. Desde que

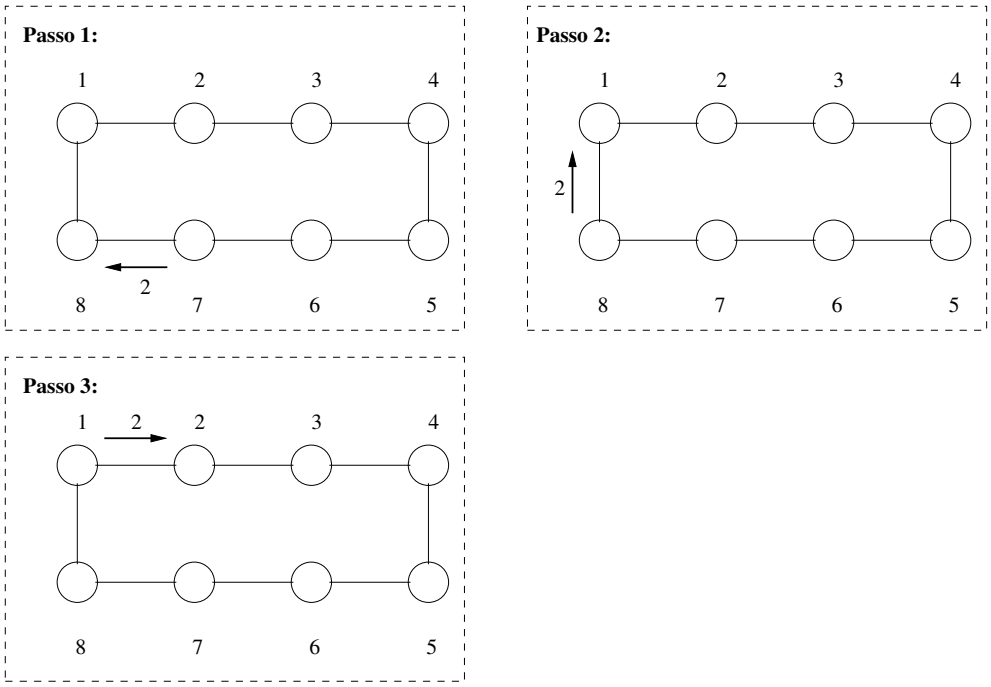

Figura 4.2: Envio de Mensagem em Topologia Circular

não haja outra transmissão ocorrendo entre os nodos envolvidos, a latência será igual a 3 vezes a latência de uma linha de conexão. Mas o nodo 7 já estará livre após enviar a mensagem ao nodo 8, e poderá continuar seu processamento.

Entretanto, caso existam outras comunicações envolvendo algum destes nodos, é possível que o outro caminho fosse mais adequado. Mas é mais *provável* que a rota envolvendo menos nodos seja a melhor, a menos que seja possível prever o comportamento dos outros nodos com base na aplicação. É o caso das operações coletivas da *MPI*.

Nas operações coletivas, o nodo raiz (*root*) sempre envia, recebe ou envia e recebe informações de todos os outros nodos. Considerando que a execução esteja sincronizada (o que pode ser garantido com `MPI_Barrier`), qualquer nodo pode prever as comunicações que envolvem os outros nodos. Desta forma, o algoritmo não deve levar em conta apenas a latência, mas também a interferência na rede. Além disso, o tempo que um nodo específico levará para completar a operação não é tão importante. É importante que todos os nodos completem a operação o quanto antes, especialmente se a aplicação

estiver sincronizada.

A figura 4.3 esboça um algoritmo para a operação coletiva `MPI_Scatter` onde o nodo 1 é o raiz. Na operação `MPI_Scatter` as mensagens sempre se originam do nodo raiz, o qual se torna um gargalo. Não é possível fazer nenhum tipo de *pipeline*, pois o nodo raiz precisa transmitir $N - 1$ mensagens.

O algoritmo consiste em dividir a rede em duas sub-redes, excluindo o nodo raiz: nodos 2-5 e nodos 6-8. A seguir, enviar alternadamente para um nodo de cada sub-rede, começando do mais distante e terminando com o mais próximo. A primeira mensagem é enviada para a rede que tiver o maior número de nodos.

É importante ressaltar que as mensagens enviadas aos nodos mais distantes terão mais *hops* que as enviadas aos nodos mais próximas, e portanto devem ser enviadas antes. Como dito anteriormente, o objetivo é que todos os nodos completem a operação o mais rápido possível, e não apenas algum nodo específico.

Ao observar a figura 4.3 pode-se perceber que o algoritmo da operação `MPI_Scatter` consiste apenas na seqüência de envio de mensagens: 5, 6, 4, 7, 3, 8, 2. Nos algoritmos de operações coletivas, não importa qual a rota da mensagem, pois essa é uma preocupação do sistema de comunicação ponto a ponto. Apenas a ordem dos processos, que leva em conta a topologia e o número de *hops* de cada rota, é importante.

Caso a topologia seja diferente, é necessário alterar apenas a ordem de envio das mensagens. Caso a topologia seja linear, a ordem de envio das mensagens é: 8, 7, 6, 5, 4, 3, 2, 1. Nenhuma outra alteração é necessária na implementação da operação.

Ainda que a topologia seja completa, nenhuma outra modificação seria necessária. Qualquer ordem de envio poderia ser utilizada, mas a latência seria a mesma: não seria possível fazer um *pipeline*, pois o nodo raiz continuaria sendo um gargalo. E, não importando qual a topologia da rede, apenas a ordem dos processos seria alterada. Isso ocorre porque a *topologia natural da aplicação* (`MPI_Scatter`) *é a topologia linear*.

A topologia de cada operação coletiva depende da existência ou não de um gargalo. O gargalo, quando existe, é o nodo raiz. Toda operação coletiva implica em uma seqüência de operações ponto a ponto entre nodos. No caso da operação

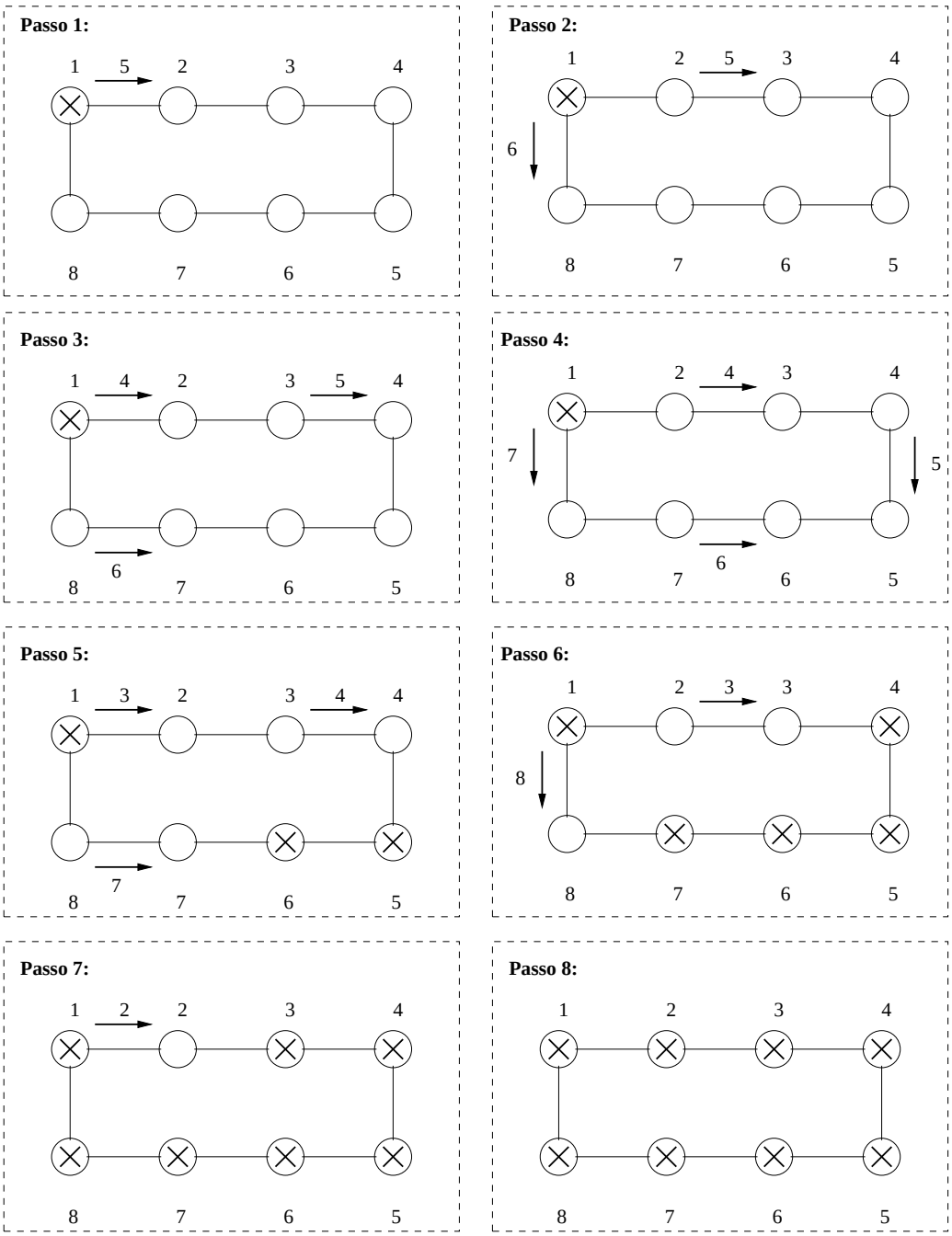

Figura 4.3: MPI_Scatter em Topologia Circular

`MPI_Barrier`	`MPI_Gather` + `MPI_Bcast`
`MPI_Bcast`	árvore
`MPI_Gather`	linear
`MPI_Scatter`	linear
`MPI_Allgather`	circular
`MPI_Alltoall`	N `MPI_Bcast`
`MPI_Reduce`	árvore
`MPI_Allreduce`	`MPI_Reduce` + `MPI_Bcast`
`MPI_Reducescatter`	`MPI_Reduce` + `MPI_Scatter`

Tabela 4.1: Topologia das operações coletivas

`MPI_Scatter`, todas as operações ocorrem entre o nodo raiz e os demais, e portanto não é possível otimizar o algoritmo com um *pipeline*.

Entretanto, na operação `MPI_Bcast` a mesma mensagem é passada a todos os nodos, e portanto o nodo que receber a mensagem no primeiro passo pode repassá-la no segundo passo, e assim sucessivamente. No passo i, $2^{(i-1)}$ operações concorrentes podem acontecer, reduzindo a complexidade do algoritmo de `O(N)` a `O(log N)`, ou seja, com uma *topologia em árvore*.

Todas as operações coletivas da *MPI* tem topologia linear ou em árvore ou são uma combinação de outras operações coletivas. `MPI_Allgather`, possui topologia *em anel* ou *circular*. A única diferença entre as topologias linear e circular é a ordem dos processos, e portanto a topologia circular pode ser vista como uma extensão da linear. As topologias de todas as operações coletivas, com base na implementação *MPICH*, são apresentadas na tabela 4.1.

4.4.1 Topologia da Aplicação

Como foi visto na seção 2.2.1, aplicações *estruturadas* têm, por definição, uma topologia natural. Todas as operações coletivas, como foi demonstrado

na seção anterior, são regulares. Todas têm sua topologia natural (linear ou árvore). Entretanto, a topologia da rede nem sempre é similar a topologia da aplicação e é necessário emular a topologia da aplicação, respeitando as características da topologia da rede.

Conforme foi discutido na seção anterior, na implementação das operações coletivas, a única informação pertinente à topologia é a ordem em que as mensagens serão enviadas: o número de mensagens será sempre o mesmo. A topologia da aplicação consiste em uma seqüência de N processos caso seja linear, e $logN$ caso seja em árvore. A aplicação não faz restrições quanto a ordem dos processos. A topologia da rede faz, como pode ser observado na figura 4.3. Os nodos mais distantes devem ser os primeiros da seqüência. Assim, o impacto da topologia de rede no algoritmo das operações coletivas limita-se a ordem das comunicações ponto a ponto que as compõem.

Entretanto, o algoritmo de algumas operações coletivas pode afetar a ordem dos processos. É o caso da operação `MPI_Alltoall`, que pode ser implementada como uma seqüência de operações `MPI_Bcast` alternando o nodo raiz. Caso todos os nodos enviem suas mensagens na mesma ordem, haverá sempre um nodo que será um gargalo. Uma possível solução para o problema consiste em deslocar a ordem de envio pelo número do nodo raiz. Assim, cada nodo seguirá uma ordem, e não haverá colisões na rede.

A ordem dos processos pode ser vista como um vetor cujo valor inicial é definido pela topologia de rede. Seu valor é alterado pelas extensões habilitadas, que são definidas pela topologia de aplicação. Por exemplo, caso a topologia de aplicação seja circular e a de rede seja hipercubo, a extensão circular será habilitada para a topologia linear. A ordem inicial será a adequada para operações lineares sobre hipercubos, alterada pela extensão circular. Mas caso uma ordem específica para topologia circular sobre rede em hipercubo seja definida, ela será a utilizada.

Com esta abordagem, é possível implementar as operações coletivas da *MPI* de forma independente de topologia, como uma composição de *extensões* sobre a ordem e uma operação ponto a ponto entre o nodo e cada elemento do vetor. Descrever uma nova topologia consiste apenas em definir a ordem inicial do vetor, e todas as operações coletivas são suportadas na nova topologia de rede. Esta abordagem permite um aumento

na reutilização de código em comparação com a abordagem tradicional.

Topologias complexas podem ser descritas como composições de topologias mais simples. Em primeiro lugar, o vetor é iniciado com base na topologia de mais alto nível da rede. Em seguida, cada elemento é substituído pelo vetor que representa a topologia imediatamente inferior, e assim sucessivamente, como pode ser visto na figura 4.4. No exemplo, a topologia superior é linear, onde cada nodo corresponde a uma sub-rede com topologia circular. À direita da figura podemos observar que no segundo passo o nodo 1 da topologia superior é substituído pelos nodos que compõem a sub-rede, na ordem adequada a topologia circular.

Conforme foi demonstrado, a representação de topologias através de um vetor permite simplificar a implementar das operações coletivas. Entretanto, não é obrigatória a utilização de um vetor elementar para armazenar a ordem. Na seção a seguir, é apresentada uma forma orientada a objetos e flexível para a armazenagem e recuperação da ordem dos nodos, através do conceito de um *iterador*.

4.4.2 O Iterador de Topologias

Nas seções anteriores as diferenças entre os algoritmos de operações coletivas e topologias foram resumidas a um vetor com a ordem de envio de mensagens e um conjunto de operações lineares. Como foi mostrado na seção 2.1.6, um vetor pode ser eficientemente representado por um contêiner, e acessado através de um iterador. Assim sendo, é possível representar o vetor da topologia com um contêiner. A implementação das operações coletivas consiste em acessar e incrementar sucessivamente o iterador, percorrendo todos os nodos conforme o algoritmo. Assim, chega-se ao conceito de um Iterador de Topologias.

O Iterador de Topologias pode ser a única forma de acesso a rede por parte da aplicação. Ele abstrai a topologia da rede, independentemente de sua complexidade, e a oferece uma interface simples: os operadores `*`, `++` e `--`.

Cada topologia de aplicação é representadas por *contâineres*, com interface similar às oferecidas pela biblioteca padrão de C++. Cada *contâiner* oferece um

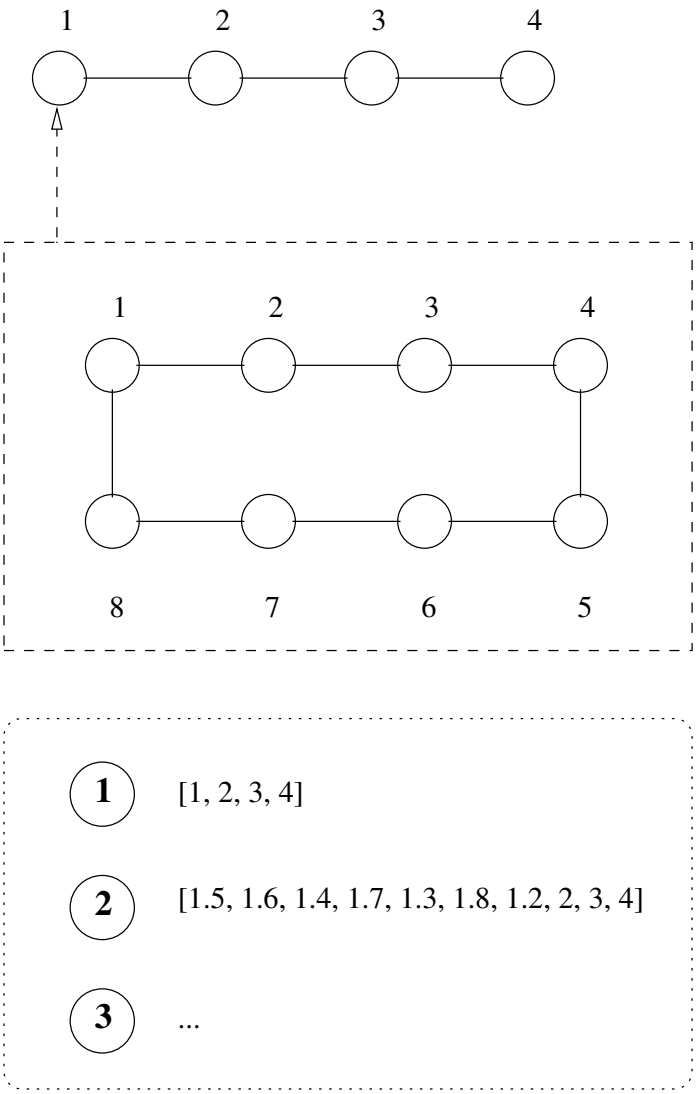

Figura 4.4: Exemplo de composição de topologias

Iterador para acesso padronizado a seus dados. O padrão ISO C++ estabelece a interface de contânieres e iteradores.

As operações coletivas podem ser implementadas utilizando apenas os iteradores, de forma que sejam parametrizadas quanto a topologia. Ao definir a topologia de rede e as extensões adequadas, as operações são instanciadas. A metaprogramação estática é utilizada neste caso para eliminar chamadas de funções virtuais. A utilização de Iteradores de Topologia não implica em perda de performance. Para verificação, um mesmo programa foi implementado com e sem o Iterador, habilitando uma extensão. Os códigos *assembly* gerados nas duas implementações foram idênticos, constatando que o Iterador não prejudica a performance da aplicação.

Para suportar as operações coletivas a topologia de rede deve fornecer dois contâineres, que representam a ordem de comunicações para as topologias linear e em árvore. O restante da implementação das operações é independente de características da rede.

O uso de Iteradores fornece maior portabilidade do que as implementações convencionais de operações coletivas. Basta definir dois vetores, ao invés de escrever várias páginas de código. Caso algum erro seja cometido pelo programador, o resultado será uma ordem incorreta de operações, que poderá resultar em perda de performance. Por outro lado, nas implementações convencionais, erros podem resultar em instabilidade do sistema.

4.4.3 Direção das Operações Coletivas

As operações `MPI_Gather` e `MPI_Scatter` tem comportamento semelhante. Ambas consistem em uma seqüência de operações envolvendo os demais nodos. As duas operações diferem apenas em um ponto: a direção da comunicação. Em `MPI_Gather`, o nodo *root* recebe mensagens dos demais. Em `MPI_Scatter`, o nodo *root* envia mensagens aos demais.

A direção das mensagens é uma característica importante das operações coletivas. Ela define qual operação ponto a ponto será usada pelo nodo *root* e a sua

Nome	Nodo Raiz	Demais Nodos
`Straight`	Envio	Recebimento
`Reverse`	Recebimento	Envio
`Bidirectional`	Envio e Recebimento	Envio e Recebimento!

Tabela 4.2: Direções das Operações Coletivas

Operação	Topologia	Direção
`MPI_Bcast`	`tree`	`straight`
`MPI_Gather`	`linear`	`reverse`
`MPI_Scatter`	`linear`	`straight`
`MPI_AllGather`	`linear`	`bidirectional`
`MPI_Alltoall`	`linear`	`bidirectional`
`MPI_Reduce`	`tree`	`reverse`

Tabela 4.3: Sentido das Operações Coletivas

complementar, que será usada pelos demais nodos. Por exemplo, em `MPI_Gather` o nodo *raiz* utilizará a operação de recebimento e os demais utilizarão a operação de envio. Em `MPI_Scatter` é o oposto.

Isolar o sentido das mensagens das outras características permite utilizar a mesma implementação para operações coletivas opostas. Os três sentidos suportados estão listados na tabela 4.2. O sentido do nodo *root* para os demais é considerado direto (*Straigth*) e o sentido oposto é considerado reverso (*Reverse*). Bidirecional (*Bidirectional*) é utilizado em operações como `MPI_Allgather`, onde a cada operação o envio de mensagens é mútuo.

Na tabela 4.3 é possível ver o sentido de cada operação coletiva.

4.4.4 Independência de API

Conforme apresentado na seção anterior, a direção de uma operação coletiva define a operação que será executada em cada nodo, que pode ser envio, recebimento ou ambos.

Entretanto, estas funções não precisam ser da *MPI*. Podem ser de qualquer sistema que ofereça operações ponto a ponto, como *PVM*, por exemplo. Portanto, a implementação das operações coletivas apresentada deve poder ser portada para outros sistemas de troca de mensagens.

Para tornar a implementação de operações coletivas independente de *API* de programação, a classe *Direction*, que representa a direção da mensagem, será parametrizada. Seu parâmetro é o conjunto das operações ponto a ponto que deve utilizar. Um conjunto é definido para as operações bloqueantes e outros para as operações imediatas da *MPI*. Outros conjuntos podem ser definidos para outros sistemas de trocas de mensagens. As operações que compõem o conjunto são encapsuladas por objetos de função.

Desta forma, a mesma análise feita para as operações coletivas da *MPI* pode ser feita para operações de outras *API*s, promovendo a reutilização de código.

4.4.5 Operações de Redução Global

O comportamento das operações de redução global depende da *API* de comunicação que é usada, e portanto são suportadas através de um *aspecto* que altera os objetos de função responsável pelas operações ponto a ponto. Este *aspecto* executará as operações de redução a cada iteração.

4.4.6 Resumo das Operações Coletivas

As operações coletivas da *MPI* foram decompostas em três entidades:

- Topologia de Aplicação;
- Direção das Mensagens;

- Funções.

A topologia de aplicação define a ordem com que os nodos irão comunicar-se e a direção das mensagens define a natureza da comunicação. As funções são parâmetros para a direção que permitem portar as operações coletivas para outras *API*s. Por ser metaprogramada, a arquitetura apresentada não prejudica a performance.

Através de aspectos que atuam como *adaptadores*, as 3 entidades podem ser ”coladas”. Por exemplo, caso uma operação requisite a topologia circular, um aspecto sobre a topologia linear pode fornecê-la.

Esta modelagem de operações coletivas segue os princípios da programação genérica. A direção das mensagens é um algoritmo genérico, que itera sobre um contâiner (a topologia) através de seu iterador. A cada iteração, ele executa uma função (a função de comunicação ponto a ponto). As entidades são semelhantes às oferecidas pela biblioteca padrão de C++ para outros domínios, e portanto encaixam-se de forma natural na linguagem.

4.5 Comparação de Desempenho

Para avaliar o desempenho da implementação proposta, um protótipo foi implementado e comparado a MPICH sobre GM. GM é o sistema de troca de mensagens para redes Myrinet. O sistema de comunicação de EPOS foi emulado sobre o GM para a comparação. A figura 4.5 mostra a latência de ambas as implementações executando uma aplicação de *pingpong* MPI. O binário gerado pela implementação proposta foi significativamente menor: a aplicação ligada com a implementação possui 20k, contra 400k utilizando MPICH-GM.

A latência do protótipo foi 5% maior, mas melhorias na emulação do sistema de comunicação podem reduzí-la.

O protótipo oferecia a maioria dos serviços ligados a MPI, e todos referentes a aplicação testada (ping-pong). Estima-se que uma implementação completa, incluindo a emulação do EPOS sobre o GM, exigiria menos de 5000 linhas de código.

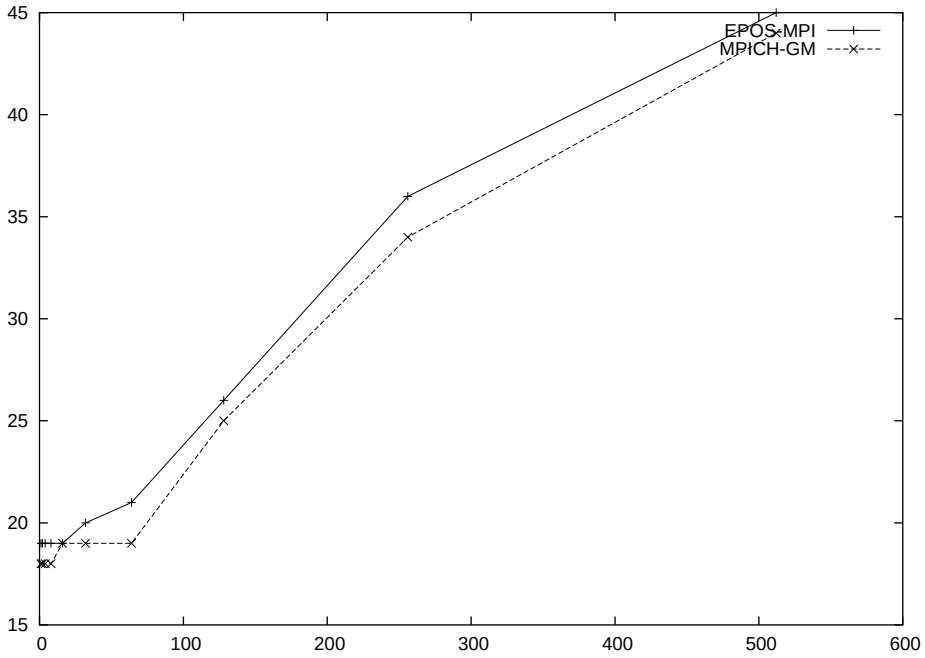

Figura 4.5: Desempenho de uma aplicação ping-pong MPI

Apenas a parte dependente da biblioteca GM (ADI) da MPICH, composta basicamente pelas operações ponto-a-ponto, possui mais de 30.000 linhas. A parte independente de ADI possui mais de 90.000 linhas, incluindo as funções do protótipo e algumas funcionalidades da MPI-2. Através das técnicas apresentadas, implementar a MPI inteira exigiu muito menos esforço do que apenas adaptar a MPICH.

Capítulo 5

Conclusão

Este texto apresentou um sistema de comunicação baseado em programação genérica. O sistema é implementado com base em *templates* que têm como parâmetro o cabeçalho da aplicação. Com esta característica as filas de mensagens, que dependem do cabeçalho, podem ser gerenciadas pelo sistema de comunicação. Com base nisso, várias funcionalidades de rede que dependem das filas, como *rendezvous* e comunicação imediata, também tornaram-se responsabilidade do sistema de comunicação. Desta forma, ele cumpre o papel deste tipo de sistema: alivia as camadas superiores de código de comunicação.

Uma implementação de MPI foi desenvolvida com base neste sistema de comunicação. Graças aos recursos avançados e a interface simples do sistema, a implementação exigiu pouco esforço e poucas linhas de código. De fato, foi mais fácil e rápido desenvolver uma implementação inteira do que apenas adaptar uma já existente, o que demonstra as vantagens do sistema proposto sobre os convencionais. As rotinas de comunicação ponto a ponto não possuem qualquer funcionalidade: apenas habilitam os recursos necessários e traduzem a interface MPI para a do sistema. Assim, a MPI foi implementada como a sua essência: A **Interface** Padrão de Troca de Mensagens.

O sistema proposto também torna mais fácil aproveitar peculiaridades do *hardware* de rede. Por exemplo, algumas redes de alto desempenho possuem um processador próprio e podem operar como um multiprocessador assimétrico com

o processador principal, implementando a maioria do que é necessário para um sistema de comunicação conformante com o padrão MPI. Isto seria impraticável com uma implementação *middleware*.

Há, entretanto, uma limitação do sistema: ele suporta apenas um protocolo a cada tempo. Geralmente, esta restrição não limita sua aplicação em computação de alto desempenho, pois nestes casos geralmente apenas um protocolo é usado. Mas em pesquisas futuras será investigada a possibilidade de utilizar múltiplos protocolos no sistema de forma eficiente.

As operações coletivas foram decompostas em 3 entidades: topologia, direção e funções. Através da análise das semelhanças e diferenças entre as operações coletivas, foi possível isolar em classes as diferenças entre as operações. Por exemplo `MPI_Gather` e `MPI_Scatter` possuem a mesma topologia, mas em sentidos opostos. Esta decomposição permite reutilizar grande parte do código, sem influenciar o desempenho. Ao isolar a parte independente da rede (direção e aspectos) da parte dependente (topologia) facilita o porte das operações. Apenas duas topologias precisam ser implementadas: linear e árvore. Sua diferença consiste na possibilidade ou não de utilizar um *pipeline*, ou seja, se as informações enviadas para cada nodo são idênticas ou não. As outras topologias utilizadas são baseadas nas duas fornecidas (através de aspectos) caso uma versão especializada não seja fornecida.

A análise das operações coletivas também pode servir de base para trabalhos futuros. A análise também pode ser aplicada a aplicações paralelas, para que ao analisar suas características de comunicação algoritmos apropriados de roteamento (e talvez também de mapeamento) possam ser escolhidos.

Ao implementar direção como um algoritmo genérico, onde as funções de comunicação são parâmetros de classe, a implementação das operações coletivas torna-se independente de API. Elas podem ser facilmente portadas para outros sistemas de troca de mensagens, sem exigir modificação nos algoritmos das operações.

Este trabalho demonstra a importância de sistemas de comunicação configuráveis, como o de EPOS. Por adaptar-se às características da aplicação (no caso a MPI), o sistema permite que ela seja implementada como apenas um mapeamento entre

APIs. A complexidade das implementações convencionais advém principalmente da falta de flexibilidade e funcionalidade dos sistemas de comunicação convencionais, bem como de suas decisões de projeto.

O projeto também demonstra a aplicabilidade de técnicas de engenharia de software para sistemas operacionais e computação de alto desempenho. Técnicas recentes permitem desenvolver software em alto nível obtendo desempenho similar aos desenvolvidos apenas com *smart programming*.

Referências Bibliográficas

[1] Lothar Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conference on Advanced Systems Engineering*, Heidelberg, Germany, June 1999.

[2] D. Beuche. The PURE family of object-oriented operating systems for deeply embedded systems. In *Proc. of ISORC'99, St Malo, France*, May 1999.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[4] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[5] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[6] Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of AOSD 2003*, pages 50–59. AOSD, 2003.

[7] Constantinos A. Constantinides, Atef Bader, Tzilla H. Elrad, P. Netinant, and Mohamed E. Fayad. Designing an Aspect-Oriented Framework in an Object-Oriented Environment. *ACM Computing Surveys*, 32(1), March 2000.

[8] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[9] Rossen Dimitrov and Anthony Skjellum. Impact of latency on applications' performance. In *Proceedings of MPIDC 2000*, March 2000.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995. version 1.1.

[11] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, 1997.

[12] Real-Time Message Passing Interface (MPI/RT) Forum. *Document for the Real-Time Message Passing Interface (MPI/RT-1,1)*, 2001. version 1.1.

[13] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley Pub Co, 1st edition, feb 1995.

[14] Antônio Augusto Fröhlich, Philippe Olivier Alexandre Navaux, Sérgio Takeo Kofugi, and Wolfgang Schröder-Preikschat. Snow: a parallel programming environment for clusters of workstations. In *Proceedins of the 7th German-Brazilian Workshop on Information Technology*, Maria Farinha, Brazil, September 2000.

[15] Antônio Augusto Medeiros Fröhlich. Scenario adapters: Efficiently adapting components. *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, July 2000.

[16] Antônio Augusto Medeiros Fröhlich. *Application-Oriented Operating Systems*. PhD thesis, GMD-FIRST, June 2001.

[17] Antônio Augusto Medeiros Fröhlich, Gilles Pokam Tientcheu, and Wolfgang Schröder-Preikschat. Epos and myrinet: Effective communication support for parallel applications running on clusters of commodity workstations. *Proceedings of the 8th International Conference on High Performance Computing and Networking*, pages 417–426, May 2000.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[20] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[21] Jens Gerlach. *Domains Engineering and Generic Programming for Parallel Scientific Computing*. PhD thesis, Technische Universitt Berlin, March 2002.

[22] Maciej Golebiewski, Markus Baum, and Rolf Hempel. High performance implementation of mpi for myrinet. *Lecture Notes in Computer Science*, 1557:510–521, February 1999.

[23] Antonio González, Miguel Valero-García, and Luis Díaz de Cerio. Executing algorithms with hypercube topology on torus multicomputers.

[24] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[25] William Gropp and Ewing Lusk. *MPICH Working Note: The implementation of the second generation MPICH ADI*. Argone National Laboratory - Mathematics and Computer Science Division.

[26] William Gropp and Ewing Lusk. *MPICH Working Note: The Second-Generation ADI for the MPICH Implementation of MPI*. Argone National Laboratory - Mathematics and Computer Science Division.

[27] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[28] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428. ACM Press, 1993.

[29] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed rendering for scalable displays. In *In Proceedings of Supercomputing*, 2000.

[30] Parry Husbands and James C. Hoe. Mpi-start: delivering network performance to numerical applications. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–15, San Jose, U.S.A., November 1998.

[31] Lars Paul Huse. Collective communication on dedicated clusters of workstations. In *PVM/MPI*, pages 469–476, 1999.

[32] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[33] Gabriele Kotsis. Interconnection topologies and routing for parallel processing systems.

[34] Steven S. Lumetta, Alan M. Mainwaring, and David E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of Supercomputing'97*, Sao Jose, USA, November 1997.

[35] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. Technical report, Rensselaer Polytechnic Institute Computer Science Department, september 1993.

[36] Inc. Myricom. The gm message passing system, 1999.

[37] Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. The design and implementation of zero copy mpi using commodity hardware with a high performance network. In *Proceedings of the 12th international conference on Supercomputing*, pages 243–250. ACM Press, 1998.

[38] Hirotaka Ogawa and Satoshi Matsuoka. Ompi:optimizing mpi programs using partial evaluation. *Conference on High Performance Networking and Computing Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–18, November 1996.

[39] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[40] L. Prylli and B. Tourancheau. Protocol design for high performance networking: a myrinet experience, 1997.

[41] Loic Prylli and Bernard Tourancheau. BIP: a New Protocol Designed for High Performance Networking on Myrinet. In *Proceedings of the International Workshop on Personal Computer based Networks of Workstations*, Orlando, USA, April 1998.

[42] Wolfgang Schröder-Preikschat. PEACE - A Software Backplane for Parallel Computing. *Parallel Computing*, 20(10):1471–1485, 1994.

[43] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems*, Paderborn, Germany, October 1998.

[44] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.

[45] Richard W. Stevens. *UNIX Network Programming*. Prentice Hall, 1998.

[46] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, June 1997.

[47] Hong Tang and Tao Yang. Optimizing threaded mpi execution on smp clusters. In *Proceedings of the 15th international conference on Supercomputing*, pages 381–392. ACM Press, 2001.

[48] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717. Springer, April 1997.

[49] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using mpi's derived datatypes to improve i/o performance. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–10, San Jose, U.S.A., November 1998.

[50] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 conference on Supercomputing*, page 3. IEEE Computer Society Press, 2000.

[51] Michael VanHilst and David Notkin. Using c++ templates to implement role-based designs. Technical report, University of Washington Departament of Computer Science and Engineering, july 1995.

[52] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[53] S. Wheat et al. PUMA: An Operating System for Massively Parallel Systems. In *Proceedings of the Twenty-Seventh Anual Hawaii International Conference on System Sciences*, Maui, U.S.A., January 1994.