Universidade de São Paulo

Instituto de Física de São Carlos

OOPS - *Object-Oriented Parallel System*

Um *framework* de classes para programação científica paralela.

Eloiza Helena Sonoda

Orientador: *Prof. Dr. Gonzalo Travieso*

Tese apresentada ao Instituto de Física de São Carlos da Universidade de São Paulo para obtenção do título de "Doutor em Ciências — Área: Física Aplicada".

São Carlos

2006

Sonoda, Eloiza Helena

OOPS - Object-Oriented Parallel System. Um framework de classes para programação científica paralela.

Eloiza Helena Sonoda - São Carlos, 2006.

Tese (Doutorado) - Área de Física Aplicada do Instituto de Física de São Carlos, 2006 - Páginas: 96.

Orientador: Prof. Dr. Gonzalo Travieso.

1. Programação Paralela; 2. Orientação a Objetos.

I.Título.

Dedico este trabalho à minha mãe Mari, pelo inestimável apoio na sua realização e por todo amor dedicado a mim e a meu filho.

Agradecimentos

Agradeço principalmente ao prof. Dr. Gonzalo Travieso pela sua orientação sempre tão competente, que me conduziu a um amadurecimento profissional e resultou em um trabalho que me satisfaz plenamente. Além disso, seu bom humor, amizade e compreensão permitiram que eu terminasse esse doutorado com muitas conquistas pessoais, e neste ponto não existem palavras para agradecer.

Agradeço aos colegas do grupo, tanto atuais quanto antigos, pelo clima agradável de companheirismo no cotidiano. Particularmente, agradeço ao Andrré e à Thaty pela sólida amizade entre nós.

Pelo apoio burocrático, agradeço às secretárias do grupo e a todo o pessoal do setor acadêmico e da biblioteca do IFSC.

Gostaria também de agradecer à minha família que sempre me apoiou. Agradeço especialmente aos meus pais, Mari e Luiz, pela ajuda tática que tornou possível a realização deste trabalho, e ao meu marido Rodrigo e nosso filho Bruno por serem uma fonte de estímulo para mim.

Agradeço à CAPES pelo suporte financeiro.

Sumário

1 Introdução 1

2 Arquiteturas Paralelas 7

2.1 Convergência de Arquiteturas Paralelas 10

2.2 Modelo de Programação Paralela 11

3 Trabalhos Relacionados 15

3.1 Atores 16

3.2 Smalltalk 17

3.3 PETSc 18

3.4 Ferramentas em C++ 19

3.4.1 Bibliotecas de Classes 19

3.4.2 Extensões à Linguagem 26

3.4.3 Discussões 28

4 OOPS 33

4.1 Proposta 34

4.1.1 Processadores Virtuais 37

4.1.2 Grupos de Processadores Virtuais 38

4.1.3 Topologias 38

4.1.4 Contêineres Distribuídos 39

4.1.5 Componentes Paralelos 40

4.2 Ferramentas Utilizadas 42

4.3 Desenvolvimento de Aplicações 44
4.4 Classes do OOPS 45
4.4.1 Classes Básicas 46
4.4.2 Classes Usadas no Tratamento de Erros 48
4.4.3 Classes que Modelam Topologias 50
4.4.4 Modos de Distribuição dos Contêineres 54
4.4.5 Classes de Contêineres Distribuídos 56
4.4.6 Intertopologia 59
4.4.7 Interoperabilidade das Classes do OOPS 60
4.5 Relacionamento com MPI 62
4.5.1 Inicialização da Máquina Virtual Paralela 62
4.5.2 Comunicador MPI 62
4.5.3 Comunicações entre os Processos MPI 63
4.6 Considerações 66

5 Exemplos 69
5.1 Início da Execução 69
5.2 Aplicando Topologias 70
5.3 Usando Contêineres Distribuídos 70
5.4 Trabalhando com Componentes Paralelos 74
5.5 Composições de Componentes Paralelos 77

6 Desempenho 81

7 Conclusões 85
7.1 Sugestões de Trabalhos Futuros 86
7.2 Considerações Finais 88

Lista de Figuras

2.1 Arquiteturas de memória compartilhada. 9
2.2 Arquitetura de memória distribuída. 9
2.3 Arquitetura híbrida. 9
2.4 Modelo de passagem de mensagem. 12

4.1 Diagrama da aplicação da topologia sobre o grupo. 39
4.2 Diagrama da distribuição do contêiner pelo grupo. 39
4.3 Diagrama da execução do componente paralelo sobre o contêiner. 40
4.4 Ilustração da composição seqüencial. 41
4.5 Ilustração da composição concorrente. 42
4.6 Esquema em camadas de um programa utilizando o OOPS. 44
4.7 Diagrama das classes básicas do OOPS. 48
4.8 Ilustração das diversas topologias apresentadas. 53
4.9 Diagrama da herança de classes que modelam topologias. 54
4.10 Diagrama da herança de classes que modelam modos de distribuição. 55
4.11 Aplicação de diferentes modos de distribuição. 56
4.12 Ilustração das atualizações dos elementos fantasmas. 58
4.13 Diagrama das classes que modelam os contêineres. 59
4.14 Diagrama da interoperabilidade das classe do OOPS. 61

5.1 Diagrama de um estêncil e da dependência dos dados. 75

6.1 Sobrecarga do OOPS. 82

6.2 *Speedup* alcançado pelas versões paralelas do Mandelbrot. 83

Resumo

Neste trabalho foi realizado o projeto e o desenvolvimento do *framework* de classes **OOPS** — ***Object-Oriented Parallel System***. Esta é uma ferramenta que utiliza orientação a objetos para apoiar a implementação de programas científicos concorrentes para execução paralela. O OOPS fornece abstrações de alto nível para que o programador da aplicação não se envolva diretamente com detalhes de implementação paralela, sem contudo ocultar completamente aspectos paralelos de projeto, como particionamento e distribuição dos dados, por questões de eficiência e de desempenho da aplicação. Para isso, o OOPS apresenta um conjunto de classes que permitem o encapsulamento de técnicas comumente encontradas em programação de sistemas paralelos. Utiliza o conceito de processadores virtuais organizados em grupos, aos quais podem ser aplicadas topologias que fornecem modos de comunicação entre os processadores virtuais, e contêineres podem ter seus elementos distribuídos por essas topologias, com componentes paralelos atuando sobre eles. A utilização das classes fornecidas pelo OOPS facilita a implementação do código sem adicionar sobrecarga significativa à aplicação paralela, representando uma camada fina sobre a biblioteca de passagem de mensagens usada.

Abstract

This work describes the design and development of the **OOPS (Object Oriented Parallel System)** class framework, which is a tool that uses object orientation to support programming of concurrent scientific applications for parallel execution. OOPS provides high level abstractions to avoid application programmer's involvement with many parallel implementation details. For performance considerations, some parallel aspects such as decomposition and data distribution are not completely hidden from the application programmer. To achieve its intents, OOPS encapsulates some programming techniques frequently used for parallel systems. Virtual processors are organized in groups, over which topologies that provide communication between the processors can be constructed; distributed containers have their elements distributed across the processors of a topology, and parallel components use these containers for their work. The use of the classes supplied by OOPS simplifies the implementation of parallel applications, without incurring in pronounced overhead. OOPS is thus a thin layer over the message passing interface used for its implementation.

1
Introdução

Tradicionalmente, cientistas investigavam os fenômenos naturais através de experimentação ou de análises teóricas. Porém, nas últimas décadas a modelagem e a simulação computacionais de sistemas físicos se tornaram um terceiro paradigma de investigação, representando uma abordagem vital em diversas linhas de pesquisa em ciência, medicina e engenharia [1]. Exemplos são estudo de clima global, modelagem de desastres naturais, estudos de seqüência de DNA, *docking* molecular, simulações hidrodinâmicas, entre outros. Essas diversas aplicações apresentam uma grande demanda computacional e há o contínuo desejo de melhorias nos modelos computacionais e de aumento na precisão das aproximações numéricas para representar os sistemas de forma mais realista. Desta maneira, indiferentemente da capacidade dos sistemas disponíveis, há sempre necessidade por maior poder computacional.

Os avanços na tecnologia de semicondutores permitiram o aumento na freqüência das operações e a integração de um número maior de componentes em um *chip*. A arquitetura de computadores transforma o potencial bruto da tecnologia em capacidade e desempenho dos sistemas computacionais [2]. Assim a grande demanda por desempenho coloca exigências cada vez maiores na arquitetura e faz com que arquiteturas paralelas sejam cada vez mais necessárias para atendê-las [3].

Os avanços tecnológicos têm possibilitado que o crescimento do desempenho

dos processadores nas últimas décadas tenha seguido a conhecida lei de Moore [3], resultando em uma duplicação deste desempenho a cada ano e meio. A definição de Almasi e Gottlieb [4] representa bem os elementos de uma arquitetura paralela: *"um computador paralelo é um conjunto de elementos de processamento que se comunicam e cooperam para resolver rapidamente grandes problemas"*. Esta definição é extensa o bastante para incluir supercomputadores paralelos com centenas ou milhares de processadores, *clusters* de computadores pessoais e *workstations* com múltiplos processadores.

Existem vários sistemas paralelos disponíveis, e o panorama atual sugere que a sua presença se torne ainda mais comum por causa do aumento do desempenho e da disponibilidade dos processadores, diminuição de seu custo e melhorias nas tecnologias de interconexão. Esses sistemas representam atualmente uma boa opção para obter desempenhos significativamente melhores do que com sistemas seqüenciais, e as aplicações mais exigentes são escritas como programas concorrentes para processamento paralelo [2].

Assim, para satisfazer o desejo de poder computacional, o processamento paralelo exerce um papel importante e tem se tornando bastante atraente, pois seu uso possibilita extrair maior desempenho do sistema computacional. Entretanto a responsabilidade de desenvolver aplicações que consigam explorar o paralelismo de forma eficiente acaba sendo do programador da aplicação, e por esse motivo por várias vezes o paralelismo se mantém inacessível ou subutilizado pela falta de ferramentas adequadas para seu uso. Um número maior de aplicações científicas poderiam se beneficiar das vantagens de multiprocessamento, porém o maior empecilho para tal reside na dificuldade do desenvolvimento da aplicação paralela [5].

A tarefa de desenvolver programas paralelos é bastante árdua, pois introduz fontes adicionais de complexidade em relação à já complexa programação seqüencial, tais como o gerenciamento explícito das execuções em vários computadores, a coordenação das comunicações entre eles, a distribuição dos dados e o acesso a dados distribuídos, e a preocupação com balanceamento de cargas e latência na comunicação. Os sistemas paralelos são capazes de fornecer grande poder computacional mas, em contrapartida, há uma grande dificuldade na programação de

aplicações que utilizem este poder de forma eficiente. Devido a isso, é importante que os desenvolvedores de aplicações paralelas deixem de se envolver em demasia com detalhes de implementação, sendo necessário que esta ocorra através de uma interface mais sofisticada que seja capaz de gerar programas paralelos escaláveis, eficientes e portáveis [3].

Nesse panorama, o desenvolvimento de ferramentas adequadas que auxiliem a programação paralela é muito importante e é neste âmbito que o *framework* de classes que propomos se insere, auxiliando a programação paralela de aplicações científicas.

Diferentes e variadas abordagens foram propostas para estimular e facilitar o desenvolvimento de aplicações paralelas. Fazem parte deste contexto as linguagens paralelas, os compiladores paralelizantes, as diretivas de paralelização e as bibliotecas. Alguns estudos como [6, 7] mostram que a *orientação a objetos* fornece bons fundamentos para a computação concorrente e/ou distribuída, sendo uma abordagem promissora e amplamente explorada em diversos domínios [8]. O paradigma de orientação a objetos fornece naturalmente ferramentas para reutilização e extensão de códigos, com herança e polimorfismo, sem necessariamente haver perda significativa de desempenho.

O projeto de componentes para uma *programação genérica reutilizável* tem se mostrado um desafio de grande interesse para desenvolvedores de *softwares*, como discutido por Szyperski [9]. A programação genérica prevê que *"componentes possam ser desenvolvidos separadamente e combinados arbitrariamente, sujeitos somente a interfaces bem definidas"* [10, 11]. Esta programação reutilizável permite que algumas funcionalidades complexas sejam obtidas da composição de objetos pré-existentes, e é apropriada para fins comerciais [12].

Neste trabalho de doutorado foi realizado o projeto e o desenvolvimento do OOPS — *Object-Oriented Parallel System*, um *framework* de classes para apoiar a programação de aplicações científicas concorrentes para execução paralela, explorando a possibilidade de desenvolvimento de plataformas extensíveis, tanto para auxiliar a programação de um determinado tipo de aplicação como para reutilizar códigos recorrentes.

Esta ferramenta fornece componentes referentes à computação científica pa-

ralela, com um conjunto de classes abstratas e concretas que permitam o encapsulamento das técnicas mais utilizadas em programação de aplicações paralelas. Pretende-se que o uso das abstrações de dados e de algoritmos paralelos fornecidos pelo OOPS acelere o desenvolvimento de aplicações científicas paralelas. Além disso espera-se incentivar a programação desse tipo de aplicação, a partir do momento em que é minimizado o envolvimento do programador da aplicação nos detalhes de uma implementação paralela com o fornecimento de ferramentas de alto nível de abstração, sem contudo ocultar totalmente os aspectos paralelos para garantir eficiência de código.

Uma gama ampla de classes projetadas foram implementadas para validar as interfaces e o projeto como um todo. Ainda outras classes estão em desenvolvimento, e as funcionalidades providas pelo OOPS podem ser estendidas de acordo com a necessidade ou desejo de experimentação.

Para apresentação desta tese o texto foi organizado da seguinte maneira: no capítulo 2 é apresentada uma breve discussão sobre arquiteturas paralelas para justificar o conceito de um multicomputador genérico [13]. Este modelo generaliza os diversos tipos de sistemas paralelos atualmente encontrados, visto que é possível verificar uma convergência das características das principais arquiteturas paralelas. Isto auxilia no projeto e na implementação dos programas pois com essa abstração o programador deixa de tratar de uma arquitetura específica durante o desenvolvimento da aplicação, aumentando sua portabilidade. O multicomputador é compatível com o modelo de programação de passagem de mensagens, o qual é utilizado neste trabalho.

Alguns trabalhos relacionados ao tema desta tese são apresentados no capítulo 3. O enfoque foi dado a ferramentas que auxiliam o desenvolvimento de aplicações concorrentes utilizando a orientação a objetos, principalmente aquelas que usam (ou se baseiam) na linguagem C++ [14], a qual foi utilizada no desenvolvimento do OOPS. Esta é uma linha de pesquisa com enfoques variados e pretende-se apenas fornecer um panorama das diversas ferramentas existentes, e com isso analisar os tipos de soluções propostas, situando o OOPS nesse panorama e o comparando com as outras abordagens.

As características gerais e maiores detalhes do *framework* OOPS são discutidos no capítulo 4. O OOPS consiste de um conjunto de classes que encapsulam técnicas comumente utilizadas em programação paralela. Seu objetivo é apoiar a programação de aplicações científicas para execução paralela fornecendo ferramentas com alto nível de abstração. Com o conjunto de classes disponibilizado até o momento, o OOPS enfatiza o uso de paralelismo de dados e apresenta também construções para paralelismo de tarefas. É destinado a aplicações que façam uso de contêineres de dados regulares, como vetores e matrizes. O OOPS usa o conceito de processadores virtuais que são organizados em grupos, e sobre os quais é possível construir topologias para viabilizar a comunicação entre os processadores. Fornece também contêineres que são coleções de dados e que podem ter seus elementos distribuídos por uma topologia. Dessa forma, componentes paralelos podem ser executados sobre os dados para a concretização da aplicação paralela. Neste contexto o programador da aplicação deve se envolver com alguns aspectos paralelos, como decisões sobre partição dos dados por exemplo, porém não necessita conhecer detalhes de sua implementação.

O capítulo 5 contém diversos exemplos de uso do OOPS para demonstrar como sua utilização pode facilitar o desenvolvimento de aplicações concorrentes. O capítulo 6 apresenta uma comparação de desempenho de um programa de teste escrito usando as construções do OOPS em relação a uma versão escrita diretamente em MPI, para avaliar a sobrecarga que o OOPS adiciona à biblioteca de passagem de mensagens.

Finaliza-se a tese com o capítulo 7 que apresenta algumas sugestões de trabalhos futuros e as conclusões que foram obtidas com a realização deste trabalho de doutorado.

2
Arquiteturas Paralelas

Conforme discutido no capítulo anterior, com os avanços tecnológicos recentes o processamento paralelo tem se destacado como uma importante ferramenta na solução de aplicações que exigem alta demanda de poder computacional.

A realidade atual de mercado fez com que os trabalhos em arquiteturas paralelas se direcionassem ao aproveitamento do potencial dos microprocessadores existentes, praticamente encerrando a época dos supercomputadores especializados.

Utilizando a taxonomia de Flynn [1], pode-se concluir que a grande maioria dos sistemas paralelos modernos se encaixa na categoria de sistemas MIMD, *Multiple Instruction Multiple Data*, em que cada elemento de processamento apresenta suas próprias instruções a executar em seu próprio conjunto de dados. A distinção principal entre os sistemas paralelos dedicados atuais se faz em relação ao acesso dos processadores à memória disponível no sistema, ou seja, se a memória é acessível diretamente a todos os processadores, a apenas um ou a um grupo pequeno deles. Dessa forma temos os seguintes tipos de arquiteturas paralelas mais comuns:

Memória compartilhada Neste tipo de arquitetura paralela, os processos compartilham uma memória comum e a comunicação entre eles se dá através da leitura e escrita das variáveis no espaço de endereçamento compartilhado. Existem dois tipo de sitemas de memória compartilhada, o SMP

(*Symmetric MultiProcessors*) e o NUMA (*Non-Uniform Memory Access*).

No SMP, todos os processadores são interligados a uma memória comum, ou a um conjunto de bancos de memória, por uma rede de interconexão, como ilustra a figura 2.1(a). O tempo de acesso à memória é igual para todos os processadores. A largura de banda no acesso à memória é um fator limitante para o aumento no número de processadores, fazendo com que sistemas desse tipo não sejam escaláveis [3].

Nos sistemas do tipo NUMA, figura 2.1(b), são mantidos os conceito de memória compartilhada e rede de interconexão. A diferença consiste em que a cada processador, ou a um grupo deles, é associada uma parte da memória comum, cujo acesso local é feito sem atraso. Porém se a posição requerida está em um nó remoto, sofre-se com a latência da rede (caso o requerimento não seja satisfeito pela *cache*). Para esse tipo de arquitetura, a localização dos dados na memória física pode representar um fator importante para o desempenho de uma aplicação, e com isso o programador deve se preocupar com a localidade dos dados [2].

Memória distribuída Neste tipo de arquitetura paralela, cada nó de processamento apresenta sua própria memória, inacessível aos outros nós exceto por trocas explícitas de mensagens entre eles, ilustrado na figura 2.2. Para a programação de sistemas de memória distribuída, a localidade dos dados é um fator bastante importante para o desempenho da aplicação.

Dois tipos de sistemas se encaixam nesta categoria, os chamados massiçamente paralelos (MPP — *Massively Parallel Processors*) nos quais a infraestrutura de conexão é fortemente acoplada e especializada, e os *clusters* que são formados por componentes e rede de interconexão comerciais.

Arquitetura híbrida ou mista É o tipo de arquitetura paralela em que o sistema de memória distribuída apresenta cada nó de processamento sendo um sistema do tipo SMP com um número pequeno de processadores, como ilustrado na figura 2.3.

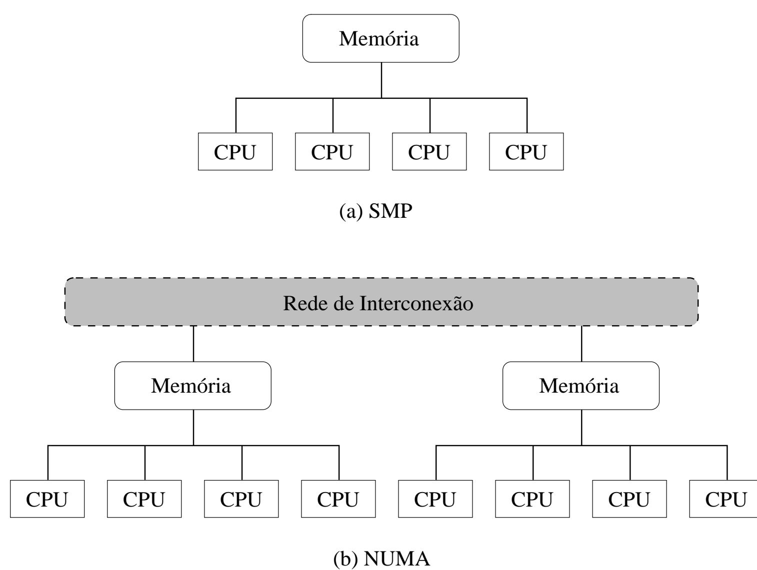

Figura 2.1: Ilustração de arquiteturas de memória compartilhada na qual todos os processos compartilham o mesmo espaço de endereçamento: (a) sistema tipo SMP (*Symmetric MultiProcessor*) e (b) sistema tipo NUMA (*Non-Uniform Memory Access*).

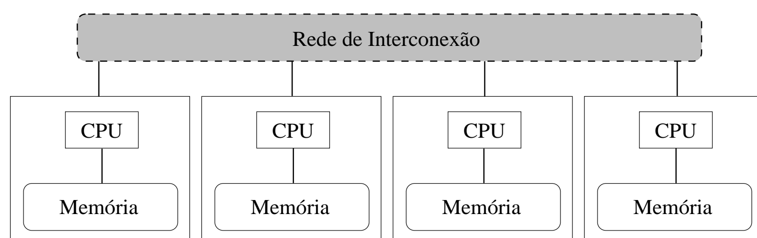

Figura 2.2: Ilustração da arquitetura de memória distribuída em que cada nó de processamento apresenta sua própria memória local inacessível aos outros nós exceto por trocas explícitas de mensagens entre eles.

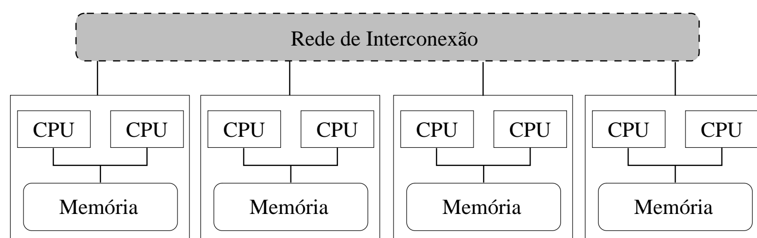

Figura 2.3: Ilustração da arquitetura híbrida na qual um sitema de memória distribuída é composto por nós de processamento do tipo SMP.

Esses tipos de arquiteturas paralelas relacionados acima podem ser representados na generalização chamada *multicomputador* [13], em que nós de processamento apresentam memória local e são interligados por uma rede de interconexão escalável. Esta generalização das arquiteturas paralelas é discutida na secção 2.1, a seguir.

Um tipo de arquitetura paralela não-dedicada comum e que não é modelada pelo multicomputador é a *Grid* [15, 16]. A *grid* é um sistema de memória distribuída, que envolve o compartilhamento de diferentes recursos (*hardware* e/ou *software*), situados em diferentes localidades, que são gerenciados por seus próprios administradores e interligados por LANs ou WANs. Comumente a interconexão é realizada pela Internet. Com este tipo de arquitetura é propiciado o uso dos recursos sobressalentes em diversos domínios, formando virtualmente um grande sistema paralelo acessível através de uma só interface. A *grid* não é modelada pelo multicomputador porque a latência no acesso a nós de processamento distintos pode ser extremamente diferente, fazendo com que o aspecto de localidade dos dados assuma outro matiz.

2.1 Convergência de Arquiteturas Paralelas

A rápida difusão de computadores no comércio, ciência e educação deve-se à primeira padronização de um modelo de máquina simples, o *computador de von Neumann* [17], que consiste de uma unidade central de processamento (CPU) conectada a uma unidade de armazenamento (memória). Este modelo simples permite aos programadores projetarem algoritmos para uma máquina abstrata e não para uma máquina específica. Um desenvolvimento semelhante era também necessário para a expansão de arquiteturas paralelas.

Examinando a evolução das principais arquiteturas paralelas, Culler, Sing e Gupta [2] indicam uma recente convergência para sistemas escaláveis, ou seja, passíveis de crescimento, através de uma máquina paralela genérica. Este modelo de máquina genérica, o *multicomputador* [13], deve ser simples para facilitar a compreensão e a programação, e realista para assegurar que programas desenvolvidos para ele executem com razoável eficiência em computadores paralelos

reais.

Assim, um multicomputador compreende um conjunto de nós de processamento interligados por uma rede de interconexão escalável, como um sistema de memória distribuída, ilustrado na figura 2.2. Cada um desses computadores executa seu próprio código, que pode acessar diretamente a memória local e enviar e receber mensagens pela rede. Na rede idealizada, o custo de emissão de uma mensagem entre dois nós é independente da localização do nó e de algum outro tráfego de rede, mas depende do comprimento da mensagem. As mensagens são usadas para a comunicação com outros computadores ou, equivalentemente, ler e escrever em memórias remotas (localizadas em outro nó de processamento).

Uma característica do modelo de multicomputador é que acessos à memória local são menos custosos que à memória remota, isto é, leitura e escrita são menos custosos que envio e recepção. Portanto é desejável por questões de desempenho que acessos a dados locais sejam mais freqüentes que acessos a dados remotos.

2.2 Modelo de Programação Paralela

Com as características do sistema computacional modeladas pelo multicomputador, são necessárias algumas considerações ao se desenvolver um programa paralelo para que um bom desempenho seja conseguido em uma máquina escalável, tais como:

Granulosidade A granulosidade de paralelização deve ser grossa, ou seja, a relação entre computações e comunicações deve ser grande. Isso se deve ao fato de que os processadores apresentam um alto desempenho nas computações enquanto que a comunicação entre os processos apresenta um custo crescente com o número de processadores.

Localidade O programa deve ser organizado de forma a favorecer os acessos a informações locais, visto que o custo de comunicação de dados a nós remotos é maior.

Escalabilidade É importante que o programa consiga se aproveitar do aumento no número de processadores disponíveis no sistema paralelo. Para isso, a

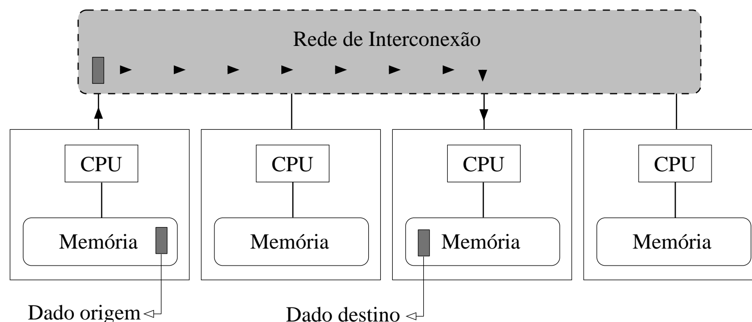

Figura 2.4: Modelo de passagem de mensagem.

aplicação deve ter a capacidade de se adaptar a essa quantidade de processadores de forma eficiente.

O modelo de *passagem de mensagens* pode ser adequadamente utilizado juntamente com a máquina genérica descrita anteriormente para satisfazer exigências da programação paralela, tendo sido empregado no desenvolvimento deste trabalho. Existem outros modelos de programação paralela que não serão discutidos aqui, como tarefas e canais, paralelismo de dados e memória compartilhada [13].

Este modelo de passagem de mensagens é amplamente utilizado. Os programas de passagem de mensagens criam múltiplas tarefas, cada uma delas encapsulando dados locais. Cada tarefa é identificada por um nome próprio e interage com outras através do envio e recepção de mensagens *para* e *de* outras tarefas, respectivamente. A figura 2.4 ilustra o modelo de passagem de mensagens, no qual o dado a ser comunicado sai do nó de processamento de origem, a mensagem passa pela rede de interconexão e chega ao seu destino.

Na abordagem de passagem de mensagens, uma coleção de programas pode ser escrita em uma linguagem seqüencial padrão complementada com chamadas para uma biblioteca com funções para envio e recepção de mensagens. Assim, uma computação compreende um ou mais processos que se comunicam através de chamadas de rotinas de uma biblioteca para a transmissão de mensagens entre eles. Esse modelo de programação é bastante apropriado para evidenciar as referências a dados presentes em memórias remotas. Para conseguir localidade, estas devem ocorrer com menor freqüência em relação à quantidade de computações e acessos à memória própria.

O paradigma de passagem de mensagem é muito atrativo devido à portabilidade de desempenho de aplicações que o utilizam, além de ter uma fácil implementação. Já que é compatível com a abstração do multicomputador, que modela de forma satisfatória tanto sistemas de memória compartilhada quanto de memória distribuída, a passagem de mensagem não se tornará obsoleta com a renovação ou alteração do sistema paralelo utilizado para a execução do programa.

Como citado no capítulo anterior, neste trabalho é proposto o desenvolvimento do *framework* de classes OOPS, descrito detalhadamente no capítulo 4, que intenciona auxiliar no desenvolvimento de aplicações científicas paralelas. Tanto a generalização do multicomputador como o modelo de programação de passagem de mensagens são usados no projeto deste *framework*.

3
Trabalhos Relacionados

Existem várias abordagens para viabilizar o desenvolvimento de aplicações paralelas, tais como as linguagens paralelas, os compiladores paralelizantes, as diretivas de paralelização e as bibliotecas. Neste capítulo é fornecido um panorama das ferramentas que têm como finalidade facilitar o projeto e o desenvolvimento de aplicações científicas concorrentes usando orientação a objetos, especialmente aquelas relacionadas à linguagem C++, visto que esta foi utilizada no desenvolvimento do *framework* OOPS.

Usamos as seguintes definições para os termos *seqüencial*, *concorrente*, *paralelo* e *distribuído*, que estão de acordo com o seu uso corrente, conforme apresentado por Briot, Guerraoui e Löhr [6]:

Seqüencial Um programa seqüencial é aquele que indica uma seqüência de operações a serem executadas uma por vez, ou seja, realizadas de forma seqüencial.

Concorrente Um programa concorrente especifica as atividades que podem ser executadas simultaneamente. Dessa maneira, indica *semanticamente* quais computações podem ser sobrepostas.

Paralelo A concorrência é uma propriedade semântica enquanto que o paralelismo é relacionado com a implementação. Assim, uma execução paralela é tal que, de acordo com a possibilidade, as atividades de um programa

concorrente são de fato executadas simultaneamente.

Distribuído Uma execução distribuída é aquela que se utiliza de mais de um sistema computacional para sua realização. Isso *não* implica necessariamente em paralelismo pois uma mesma linha de controle pode migrar de sistema sem contudo executar atividades de forma a se sobreporem.

A seguir, apresenta-se uma breve discussão de algumas ferramentas relacionadas a este trabalho para que seja possível analisar as diversas soluções propostas e comparar o OOPS com essas abordagens.

3.1 Atores

O modelo de atores proposto por Agha [18] é bastante importante e influente [19]. É um modelo de linguagem baseado em objetos e foi criado para simplificar a expressão de concorrência e distribuição.

Os atores diferem bastante dos modelos tradicionais principalmente em questões relacionadas à natureza da concorrência. É feita uma associação dos conceitos de *objetos* e *processos*, explorando as similaridades entre encapsulamento de dados em objetos com a memória particular dos processos, e o envio de mensagens a objetos com a passagem de mensagem entre processos. Este modelo apresenta objetos com sua própria linha de controle e que utilizam mensagens assíncronas enviadas entre eles. Em várias abordagens são denominados *objetos ativos*.

Atores são elementos que apresentam uma *caixa postal* e um *comportamento*. A caixa postal designa um *buffer* que pode armazenar uma seqüência linear e ilimitada de mensagens (chamadas *comunicações* e que são armazenadas pela ordem de chegada). O comportamento de um ator é definido por suas ações em resposta a uma comunicação. Um ator puro pode processar somente uma comunicação de sua caixa postal antes de finalizar [20].

Para responder a uma mensagem um ator: (a) pode enviar um número finito de comunicações a outros atores com nomes conhecidos de caixas postais; (b) pode criar um número finito de novos atores; e (c) deve designar um sucessor

com sua mesma identidade (nome da caixa postal) para processar a próxima comunicação recebida.

O comportamento desse sucessor é chamado de *comportamento substituto*, e não há restrições na relação entre o comportamento de um ator e seu comportamento substituto. O ator sucessor ocupa uma posição especial pois representa a continuação do processo original. O sucessor pode iniciar suas atividades respondendo à próxima comunicação da caixa postal assim que estiver criado, e pode executar simultaneamente enquanto seu antecessor está completando outras tarefas. Isso permite responder concorrentemente a uma seqüência de mensagens enviadas a uma caixa postal.

Supondo que um ator envia p comunicações para outros atores e cria q novos atores além de seu sucessor antes de morrer, essas $p+q+1$ ações são em princípio concorrentes, ou seja, o processamento de uma comunicação dispara $p+q+1$ tarefas concorrentes.

A granulosidade resultante é bastante fina, o que acaba reduzindo a eficiência do código nos sistemas atuais. A programação também é dificultada pela falta de ordem das mensagens enviadas entre os mesmos parceiros, já que nesta abordagem a ordem do envio das mensagens não é necessariamente igual à sua ordem de chegada.

O modelo de atores apresenta também outros pontos fracos, como dificuldade no reaproveitamento de código e a incompatibilidade dos aspectos de objetos que *não* têm similaridade com processos, tais como herança e replicação [6]. Isso se deve à diferença conceitual entre objetos e processos, pois o primeiro representa uma entidade de *encapsulação* enquanto que o último é uma entidade de *concorrência*.

3.2 Smalltalk

O Smalltalk-80 [21, 22] é um ambiente de programação orientado a objetos e todo o sistema é estruturado através de bibliotecas de classes, construídas sobre os conceitos de objeto, mensagem, classe e herança.

O Smalltalk oferece originalmente uma biblioteca de classes padrão bastante

rica. Esta apresenta, entre outras, classes para programação concorrente, tais como `Process` que modela processos com sua linha de controle, o escalonador de processos modelado pela classe `ProcessorScheduler`, e `Semaphore` para implementar semáforos que controlam a sincronização dos processos. O Smalltalk ainda oferece bibliotecas para comunicação remota (*sockets* e RPC) assim como para armazenamento e troca de objetos.

Outras plataformas utilizam essa linguagem e constróem abstrações de mais alto nível estendendo a biblioteca padrão. Exemplos são o Actalk [23] que introduz conceitos de objetos ativos ao Smalltalk, e o Simtalk [24] que trata de simulações distribuídas e concorrentes.

Mesmo sendo uma linguagem puramente orientada a objetos e contando com uma biblioteca padrão bem completa, o Smalltalk não se tornou tão popular quanto o C++ no desenvolvimento de *softwares* e ferramentas para programação. O Smalltalk é interpretado, e as diversas construções da linguagem têm implementações pouco eficientes devido à grande flexibilidade da linguagem.

3.3 PETSc

O PETSc (*the Portable, Extensible Toolkit for Scientific computation*) [25] é uma biblioteca numérica projetada para facilitar a escrita de aplicações científicas paralelas. O enfoque é dado para aplicações modeladas por *equações diferenciais parciais*.

A abordagem do PETSc é encapsular algoritmos matemáticos usando orientação a objetos e utilizar passagem de mensagens para tratar dos aspectos paralelos. É escrito em C e utiliza pacotes como MPI, BLAS e LAPACK.

O PETSc fornece várias das estruturas de dados e dos algoritmos numéricos necessários para a solução de equações diferenciais parciais. As comunicações são tratadas internamente nos objetos PETSc, sem que todos os aspectos paralelos sejam ocultados ao programador, o qual pode fazer combinações de fases seqüenciais e paralelas. A vantagem dessa organização é que a comunicação é intrinsecamente associada aos objetos de alto nível e às operações realizadas sobre eles, ao invés de utilizar um conjunto de envios e recebimentos e outras chamadas

MPI, que dificultam o entendimento da conexão entre uma chamada específica de passagem de mensagem e as operações matemáticas executadas nos dados.

Os três objetos abstratos básicos do PETSc são *conjunto de índices*, *vetores* e *matrizes*. Sobre essa base o pacote apresenta várias classes de objetos *solvers*, incluindo os lineares, não-lineares e *timestepping* [26]. Estes podem ser usados em aplicações escritas em Fortran, C e C++. A biblioteca é organizada em hierarquias, permitindo aos usuários empregar o nível de abstração que for mais apropriado para seu problema em particular.

Esta proposta é suficientemente genérica para ser utilizada em diversas aplicações científicas porém sua flexibilidade é limitada, por fornecer abstrações úteis especificamente para algoritmos que tratam de equações diferenciais parciais.

3.4 Ferramentas em C++

A linguagem C++ é bastante utilizada no desenvolvimento de diversas aplicações científicas e também de ferramentas para auxílio da programação paralela. Várias são as propostas encontradas para a viabilização da programação paralela com orientação a objetos usando C++, e dentro desta ampla gama de ferramentas, selecionamos somente algumas delas para possibilitar a análise dos tipos de soluções propostas. Esses trabalhos podem ser classificados em duas categorias, bibliotecas de classes e extensões à linguagem, como exposto a seguir.

A discussão das técnicas apresentadas é feita em conjunto no final desta secção.

3.4.1 Bibliotecas de Classes

ABC++ [27] É um sistema estruturado em dois níveis de abstração. O nível mais baixo permite criar linhas de controle, controlar seu escalonamento e sincronizar suas operações. Já o nível mais alto oferece dois modelos de programação: objetos ativos e regiões compartilhadas. Com os objetos ativos, cada objeto C++ apresenta sua própria linha de controle e pode estar localizado em qualquer processador do sistema. Em regiões compartilhadas, um mesmo objeto pode ser compartilhado por algumas linhas de controle distintas.

Active Expressions [28] Esta é uma biblioteca de classes para concorrência em C++. Oferece padrões comuns de comunicação e sincronização encapsulados em interfaces bem definidas, tudo em C++ sem utilizar outras ferramentas. Apresenta três principais abstrações, (a) *componentes ativos*, que encapsulam as funcionalidades do usuário em componentes que operam concorrentemente, (b) *expressões ativas*, que expressam os padrões de comunicação e sincronização entre os componentes ativos, e (c) *regiões compartilhadas*, que permitem compartilhamento de dados entre os componentes ativos.

ClassdescMP O Classdesc [29, 30] é um sistema que enfatiza o uso de *descritores de classes* (*class descriptors*), que são declarações da estrutura interna dos objetos extraídas pelo pré-processador e inseridas no programa para permitir serialização do objeto. Isso possibilita o envio de objetos para um *buffer* através dos operadores de inserção e extração, que posteriormente pode ser salvo em arquivo ou enviado pela rede (o conteúdo pode ser salvo de forma independente da arquitetura, opcionalmente, para por exemplo ser usado em um *cluster* de computadores heterogêneos).

O ClassdescMP é um ambiente para programadores C++ usando MPI. A inicialização e finalização do MPI são realizadas, respectivamente, pelos construtor e destruidor de uma classe fornecida pela biblioteca, denominada `MPISPMD`. Essa classe também permite acesso a dados comumente utilizados em programas MPI, como o número de processos envolvidos na computação ou o identificador do próprio processo. A classe `MPIbuf` modela os objetos serializáveis. Além dessas facilidades, permite ao usuário do sistema fazer chamadas diretamente para o MPI caso seja necessário por questões de desempenho.

C++// É um sistema que consiste de um pré-processador e uma biblioteca de classes para fornecer reusabilidade em programação concorrente [31]. O pré-processador apenas gera automaticamente algumas classes adicionais ao programa que serão usadas como *proxy* durante a execução, sem que para isso seja necessária nenhuma alteração do código. O conjunto de classes implementa objetos ativos, com sua própria linha de controle, e objetos passivos, que são objetos puramente

seqüenciais. Para os objetos ativos, o programador declara uma classe derivada de uma classe especial da biblioteca ou, na alocação do objeto, usa um operador definido pelo C++// para gerar o novo processo para o objeto.

O C++// é baseado em *reflexão* [32], sendo que a ação a ser tomada na requisição de uma execução é decidida com o uso dos *proxies* gerados pelo pré-processador.

DAPPLE (*DAta-Parallel Programming Library for Education*) [33] O DAPPLE é uma biblioteca de classes em C++ concebida para o ensino de programação paralela. O paralelismo ocorre através da distribuição de dados em um programa SPMD. Vetores e matrizes são classes base com paralelismo de dados e com os operadores aritméticos sobrecarregados, tais como permutações e reduções. São eles que representam a distribuição dos elementos pelos processadores virtuais. Ainda é fornecida a possibilidade de operar com apenas uma parte dos processadores virtuais, que nesse caso são chamados de processadores ativos.

DatTel A STL [34] utiliza o conceito de *templates* e é usada em aplicações para modelar estruturas de dados diversas. O DatTel (*Data-parallel Template Library*) [35] é uma biblioteca que pretende viabilizar a execução eficiente de programas que usam STL em máquinas paralelas. Atém-se ao paradigma de programação com *templates* do C++ clássico e usa paralelismo de dados.

Janus O *framework* Janus [36] fornece componentes genéricos para aplicações científicas baseadas em *malhas*. Uma característica importante é que ele trata de estruturas regulares e irregulares de maneira homogênea. É implementado usando STL e roda sobre MPI.

Visa simplificar a implementação de elementos finitos e outros métodos baseados em malhas. Pode ser usado tanto em aplicações simples, tal como método de diferenças finitas em grades retangulares, como em códigos mais complexos, por exemplo refinamento de malha adaptativo.

Este *framework* fornece representação de conjuntos, suas relações, e os dados que os definem. Os três conceitos fundamentais do Janus são *Domain*, *Relation*

e *Domain Function*. Um *Domain* é um conjunto finito representado por uma seqüência em que cada elemento do domínio tem uma única posição (índice) pela qual pode ser representado. As relações entre os domínios, modeladas por *Relation*, podem ser representadas por matrizes de adjacência, e as funções que atuam sobre os domínios são representados pelo *Domain Function*.

KeLP (*Kernel Lattice Parallelism*) [37] É um *framework* de classes em C++ para implementação de aplicações científicas que tenham necessidade de adaptação aos dados ou ao *hardware* em tempo de execução. Tem como objetivo a portabilidade das aplicações por diversas arquiteturas. É possível tratar de balanceamento de cargas, comunicação entre os processos e entrada e saída paralelas em alto nível. Encapsula comunicações expressando-as em termos de movimentação atômica de vetores. Apresenta paralelismo de tarefas, separando o paralelismo da computação numérica. Fornece ferramentas de alto nível para que o programador se concentre na aplicação e na matemática envolvida, e não em detalhes de paralelismo, como distribuição e comunicação.

As classes do KeLP podem ser especializadas através de herança ou por composição, de acordo com as necessidades do programa. É uma camada entre a aplicação e o nível baixo de comunicação (MPI). A classe `Region` representa um conjunto retangular de índices em n dimensões no qual o usuário do *framework* decompõe seu problema. A classe `XArray` modela os blocos de dados distribuídos pelos processadores na `Kelp Grid`. Para problemas do tipo estêncil, o KeLP utiliza células fantasmas para armazenar os dados.

OODFw (*Object-Oriented Distributed Framework*) Este *framework* apoia a programação por dados distribuídos em aplicações irregulares para alto desempenho [38]. Trabalha com fases seqüenciais e paralelas, seguidas de sincronização, e o particionamento dos dados e a comunicação entre os processos ficam implícitos no modelo. Apresenta objetos seqüenciais usados nas computações e objetos distribuídos que expressam paralelismo de dados. Os dados armazenados em objetos distribuídos são particionados automaticamente entre os processos existentes. Além da própria partição, cada processo apresenta uma cópia dos elementos vizi-

nhos, determinados pelo padrão de acesso aos dados. A atualização dos dados é realizada quando o usuário executa o método `Update` que atua sobre uma classe derivada de `UserData`. A classe `Subdomain` é usada para determinar índices de acesso à partição local.

OOMPI (*Object-Oriented MPI*) [39] Esta é uma biblioteca de classes que apenas encapsula as funcionalidades do MPI em uma *interface orientada a objetos*. O OOMPI se baseia nos mesmos conceitos do MPI, porém possibilitando envios e recepções com sintaxe mais simples, sem a especificação de tantos parâmetros como em MPI [40].

Para realizar as comunicações, o OOMPI utiliza *portos*. Esses objetos, que são instâncias da classe `OOMPI_Port`, contêm informações sobre o comunicador MPI e o *rank* do processo para o qual a mensagem será enviada. Todas as comunicações são definidas em termos de mensagens, neste contexto, objetos `OOMPI_Message`. Sempre que uma comunicação for realizada, uma mensagem será construída automaticamente e de forma transparente ao programador. É possível também transmitir um objeto em uma mensagem, o qual deve ser derivado da classe `OOMPI_User_type` pertencente à biblioteca e deve fornecer a implementação de um construtor que monta um tipo de dados MPI para a transmissão do objeto. O OOMPI permite ainda o envio e recepção de mensagens através de *streams* com operadores de inserção e extração, e também inclui a possibilidade de empacotamento de dados.

Representa uma *camada fina* adicionada entre o programa do usuário e o MPI, fornecendo as abstrações de classes sem promover perdas de desempenho ao sistema de passagem de mensagens, como mostrado em [41].

Overture [42, 43] É um *framework* para uso na programação de problemas de equações diferenciais parciais, inclusive os que envolvem geometrias complexas. Trabalha com malhas estruturadas (refinamento adaptativo de malhas), e pode usar composição e sobreposição destas para formar uma malha híbrida e assim modelar domínios mais complexos.

Inclui as bibliotecas A++/P++ para vetores seqüenciais e paralelos. Essas

bibliotecas utilizam classes que encapsulam o paralelismo e possibilitam a distribuição dinâmica de dados, no mesmo estilo do HPF [44].

POOMA O *framework* POOMA (*Parallel Object-Oriented Methods and Applications*) [3] tem como objetivo facilitar o desenvolvimento de aplicações científicas de larga escala, para executar de forma seqüencial ou paralela.

Este *framework* consiste de um conjunto de classes genéricas em C++ integradas e organizadas em várias camadas de abstração. Cada uma dessas camadas é construída sobre a camada inferior. A camada mais alta consiste de aplicações, tais como modelo de acelerador de partículas ou transporte de nêutrons por algoritmo de Monte Carlo [45]. Essa camada é construída sobre os objetos gerais do POOMA (como malhas, campos, partículas) e algoritmos paralelos (transformada de Fourier, interpoladores). Abaixo há a camada de abstração de paralelismo que gerencia tarefas paralelas como decomposição de domínios e balanceamento de carga. Finalmente, a camada mais baixa de todas trata de detalhes de computação, como contêineres genéricos e algoritmos usando STL.

O POOMA apresenta a classe `Field` que é um contêiner de dados distribuído usado para descrever um campo físico, e seus elementos podem estar sobre um tipo `Mesh` definido em POOMA para discretizar o domínio da simulação. Há também a classe `Index` que permite ao usuário escrever expressões do tipo estêncil. Para apoiar simulações com partículas, o POOMA fornece a classe `ParticleBase` que apresenta uma descrição mínima para um conjunto de partículas. A classe `Interpolator` implementa esquemas para interpolar dados entre a posição das partículas e os elementos de `Field`. Ainda trata da transferência de dados distribuídos via classe `DataConnect`. Essa classe permite que o POOMA compartilhe dados com entidades externas ao *framework*.

POOMA II O projeto POOMA está em curso com o POOMA II [46] atualmente. Este apresenta contêineres que facilitam o armazenamento e a computação de valores de matrizes através de acesso a elementos individuais, paralelismo de dados e computações do tipo estêncil. O POOMA II manipula automaticamente todas as comunicações entre processos para que o desenvolvedor não precise se

concentrar nos detalhes de implementação.

Conceitos de domínios, contêineres e *engines* são bastante importantes no seu contexto. Um `Domain` especifica um conjunto de índices permitidos em um contêiner, ou seja, a região na qual um contêiner pode definir valores. As classes `Array` e `Field` são contêineres de valores, a primeira funciona como um mapa dos índices de um domínio para seus valores enquanto que a segunda suporta múltiplos valores em cada uma de suas células. Foi introduzido o uso de `Engines`, separando claramente o uso do contêiner do armazenamento de seus dados. O `Engine` é a entidade que realmente armazena e computa dados para os contêineres.

Para distribuir dados por uma máquina paralela basta passar um objeto `Layout` na construção do `Engine`. Um *tag* é usado para especificar um *engine* em particular, como `Brick` por exemplo, que realiza uma distribuição dos dados por blocos.

SAMRAI (*Structured Adaptive Mesh Refinement Applications Infrastructure*) [47, 48] Este é um *framework* orientado a objetos cuja finalidade é oferecer tecnologia para apoiar o desenvolvimento de aplicações paralelas de refinamento de malha adaptativo (*Adaptive Mesh Refinement* — AMR), sem que haja conhecimento de programação paralela por parte do programador da aplicação. Permite ao pesquisador explorar novas aplicações e algoritmos através da composição e extensão de componentes existentes usando orientação a objetos.

Esse *framework* trata de malhas *estruturadas* e focaliza as dificuldades numéricas, algoritmicas e de *software* relacionadas ao uso de AMR em aplicações físicas.

O SAMRAI é dividido em uma coleção de *pacotes*, tais como *solvers*, *integration algorithms*, *mesh management* e *geometry*. Cada um desses pacotes apresenta um conjunto de elementos de *software* usados para construir uma aplicação AMR. Juntos, os pacotes formam um só ferramental de propósito geral com o qual as aplicações são construídas. Entre os diversos pacotes, é mantida dependência somente entre as interfaces para facilitar o uso, extensão e manutenção da biblioteca.

3.4.2 Extensões à Linguagem

Charm++ [49, 50] É uma linguagem orientada a objetos para programação paralela que consiste de C++ acrescido de algumas extensões. Variáveis globais e membros estáticos não são permitidos visto que suas implementações em sistemas paralelos apresentam problemas de eficiência.

Diferencia entre objetos concorrentes, chamados *chares*, e objetos seqüenciais, que são instâncias de uma classe normal de C++. O modelo de programação é direcionado à mensagem, ou seja, a execução dos objetos concorrentes é ativada por mensagens como em [51] e, enquanto um processo espera por alguma comunicação, outro processo pode executar no processador. As classes *chare* têm dados locais e métodos para comunicação de mensagens, e apresenta *pontos de entrada* ao invés de métodos públicos. Esses pontos de entrada determinam as operações a executar ao receber uma mensagem. Outros tipos de objetos são o *branched chare*, que permite a definição de formas de distribuição de dados e que pode ter métodos públicos além dos pontos de entrada, e o *branch-office chare* que são objetos replicados. Apresenta *chare containers*, tal como um vetor por exemplo, o qual pode ter seus elementos migrados entre os processadores durante o seu uso.

O Charm++ apresenta estratégias de balanceamento dinâmico de cargas, onde os *chares* são distribuídos pelo sistema automaticamente. O programador é quem deve decidir o que será executado em paralelo e deixar que o sistema de tempo de execução escolha em qual processador e em que momento.

COOL (*Concurrent Object-Oriented Language*) [52, 53] É uma extensão de C++ projetada para ser usada em sistemas de memória compartilhada. Fornece estruturas para concorrência, exclusão mútua e sincronização de tarefas. O programador fornece informações sobre o padrão de referência aos dados e o sistema de tempo de execução distribui as tarefas e objetos de modo que as tarefas estejam próximas dos objetos a que fazem referência (em hierarquia de memória).

DC++ O *Distributed C++* [54, 55] é uma linguagem para programação paralela em sistemas distribuídos fracamente acoplados. Estende C++ em três categorias de classes: as classes do tipo *gateway* que atuam como portas para comunicação e sincronização entre os processos, as classes cujas instâncias podem ser passadas entre os processos através dos *gateways* e as classes C++ convencionais.

A concorrência é alcançada com a criação de múltiplos processos e iniciando várias linhas de controle. O DC++ inclui suporte para concorrência com as linhas de controle, para comunicação e sincronização com portos explícitos, invocação remota de métodos e espera por futuros [56], e para localidade com domínios. O código é compilado para C++ que realiza chamadas para o sistema de tempo de execução DC++.

DPC++ (*Data Parallel C++*) A linguagem DPC++ [57] pode ser descrita como C++ acrescido de paralelismo de dados no estilo das extensões de *array* do Fortran 90. O paralelismo é alcançado através do uso de objetos paralelos. Introduz tipos de escalares, declarações de vetores paralelos e os operadores sobre esses vetores.

ICC++ (*Illinois Concert C++*) [58] É um dialeto de C++ no qual o programa pode ser executado seqüencial ou paralelamente. Acrescenta anotações para concorrência e usa análises de dependência pelo compilador para gerar a concorrência necessária. Dessa forma, o programador deve indicar os pontos de concorrência (tais como blocos e *loops*) e o sistema otimiza a granulosidade de acordo com o sistema paralelo disponível. Um bloco concorrente executa todas as operações simultaneamente de acordo com a análise de dependências; em um *loop* concorrente, as dependências carregadas por este [59] são apenas analisadas para variáveis isoladas ou vetores inteiros. O ICC++ fornece construções para lidar com *arrays* distribuídos, e viabiliza paralelismo de dados e de tarefas.

Mentat Na proposta do Mentat [60] o programador determina questões de granulosidade e particionamento, enquanto que o sistema de execução e o compilador

controlam comunicação e sincronizacão. A linguagem é baseada em C++ e o paralelismo é modelado por classes especiais, tal como a classe *Mentat*. Uma classe Mentat é usada principalmente em três situações, para objetos que: (a) possuem métodos demorados, (b) armazenam dados compartilhados ou (c) apresentam métodos com latência grande (realizando entrada e saída, por exemplo). Cada objeto Mentat (isto é, de uma classe Mentat) tem um espaço de endereçamento separado e uma linha de controle distinta.

As classes Mentat se classificam em *regulares* e *persistentes*, sendo que as persistentes podem ou não ser seqüenciais. Uma classe Mentat regular tem métodos puramente funcionais (sem estado) e seus objetos servem apenas para disparar novas linhas de controle. Uma classe Mentat persistente permite manter estado interno. Se a classe for persistente seqüencial, os métodos serão executados sobre seus objetos na ordem do programa; caso contrário, o compilador (e o sistema de execução) é livre para reordenar a execução dos métodos chamados sobre o objeto.

3.4.3 Discussões

As várias propostas supra apresentadas possuem características similares e por isso as discussões e considerações são feitas em conjunto nesta secção. Apesar de a proposta completa do OOPS ser apresentada formalmente no texto no próximo capítulo, adianta-se algumas de suas características para possibilitar a comparação com as outras abordagens de ferramentas e assim justificar algumas escolhas de projeto.

O desenvolvimento de *linguagens para concorrência* possui a vantagem de que a expressão da concorrência e da comunicação é mais simples do que se realizada por meio de *bibliotecas*. Além disso, o compilador pode executar otimizações e detecção de erros de forma mais extensiva do que quando se usam bibliotecas. Contudo há desvantagens que podem acabar por desencorajar sua utilização, tais como a necessidade de aprendizado da nova linguagem pelos programadores, perda de código existente e a complexidade que representa o desenvolvimento de uma linguagem e o ferramental que a acompanha, como compiladores e sistema

de tempo de execução. Toda essa complexidade desestimula a experimentação com novas funcionalidades e também dificulta o acompanhamento da tecnologia de compiladores das linguagens existentes.

A proposta do OOPS se encaixa na categoria de biblioteca para concorrência e é desenvolvido em C++, que é uma linguagem de uso bastante amplo. O OOPS é um *framework* que apresenta classes concretas e abstratas para serem utilizadas no desenvolvimento da aplicação paralela. Assim mantém-se a linguagem conhecida pelo programador da aplicação, o qual apenas deverá aprender algumas interfaces de classes e como elas irão cooperar ou se estender para resolver seu problema em particular. Dessa forma, novas técnicas de compilação e otimização desenvolvidas para linguagens tradicionais podem ser aproveitadas sem contudo haver a necessidade de modificações da ferramenta (apenas atualização do compilador usado). Além disso, códigos existentes podem ser encaixados na estrutura do *framework* sem haver a necessidade de total reescrita já que a linguagem de programação é mantida.

Algumas das ferramentas relacionadas apresentam um diferencial quanto à abordagem de integração de concorrência e orientação a objetos, pois usam o conceito de *objetos ativos*, que é como se denomina a união dos conceitos de objeto e processo. Nesta categoria estão as bibliotecas ABC++ e C++// por exemplo. O uso dessa solução acarreta nas dificuldades expostas na secção 3.1, que são problemas conceituais, de eficiência e dificuldade de reaproveitamento de código.

No OOPS a orientação a objetos é utilizada para organizar a estrutura da aplicação e encapsular detalhes de implementação como comunicação e sincronização, por exemplo. Dessa forma, é possível fornecer abstrações de alto nível para facilitar o desenvolvimento de aplicações paralelas. Os objetos lidam com processos, porém esses conceitos são mantidos separados [61].

O paralelismo de dados é outra característica recorrente no panorama de ferramentas apresentado. Seu uso é interessante para diversos problemas científicos e encoraja a programação para grandes quantidades de dados. Além disso, quando apropriado, gera algoritmos que correspondem de perto à estrutura matemática do problema. Entretanto, alguns algoritmos não podem ser adequadamente ex-

pressos unicamente por meio dele.

Algumas das ferramentas discutidas apresentam paralelismo de dados, tanto bibliotecas como linguagens. Destas, várias implementam objetos distribuídos e particionados automaticamente. Com o OOPS é possível realizar construções de paralelismo de dados. Por exemplo, contêineres de dados podem ser distribuídos com abstrações de alto nível pelos processadores virtuais disponíveis para a execução, porém espera-se que o programador da aplicação paralela decida qual o melhor particionamento para seu caso em particular. Mesmo sem necessitar se envolver com os detalhes de implementação da distribuição dos dados, a decisão da forma que ela deve ser realizada é deixada a cargo do usuário do *framework*, o qual tem maior capacidade de discernimento na escolha da melhor opção para sua aplicação. Esse aspecto do projeto do OOPS, de paralelismo explícito, aumenta a complexidade de programação mas possibilita alcançar melhores desempenhos para a aplicação.

Algumas ferramentas existentes são voltadas a *algoritmos específicos*. Janus, Overture e SAMRAI se encaixam nessa categoria, e são voltadas a algoritmos baseados em malhas (regulares ou não). Com esse tipo de solução é possível otimizar as implementações oferecidas já que o uso que será feito da ferramenta é conhecido, porém há a desvantagem de perder flexibilidade. O *framework* OOPS é projetado para ser mais abrangente, no sentido de oferecer facilidades para aplicações científicas regulares em geral que necessitem de processamento paralelo e utilizem contêineres como vetores e matrizes para armazenar seus dados. Para isso fornece classes que encapsulam técnicas comumente encontradas nesse tipo de aplicação.

Várias são as ferramentas, entre bibliotecas e linguagens, que implementam contêineres distribuídos e algoritmos paralelos que atuam sobre eles. Essa variedade de propostas mostra que há bastante interesse nessa abordagem, e que há diversas formas para fornecer essa facilidade. O OOPS fornece abstrações para contêineres distribuídos (vetores, matrizes, etc.) e também adiciona componentes paralelos para atuarem sobre os elementos desses contêineres.

O OOMPI é uma biblioteca que difere bastante dos outros trabalhos apresentados neste capítulo pois somente fornece uma interface orientada a objetos ao

MPI, diminuindo o nível de detalhamento necessário para as comunicações. Apesar de introduzir pouca sobrecarga e ser flexível, não adiciona construções de mais alto nível de abstração para auxiliar a programação paralela. Com o uso desta ferramenta, assim como do MPI, o código das comunicações fica entrelaçado com o código das computações, dificultando o entendimento de como as comunicações e as operações matemáticas se relacionam.

O OOPS utiliza orientação a objetos para encapsular códigos recorrentes em programação paralela, e assim fornece classes para facilitar a implementação das aplicações. Por exemplo, existem classes que modelam contêineres de dados distribuídos, topologias para a comunicação entre os processos e outras para diversos modos de distribuição dos dados.

Há outras similaridade do OOPS com algumas ferramentas apresentadas. Por exemplo, o Classdesc, assim como o OOPS, inicializa e finaliza o MPI em um construtor e um destruidor, respectivamente, sem a interferência do programador. Com isso a função de finalização do MPI (`MPI::Finalize()`), que deve ser executada por todos os processos paralelos, é chamada mesmo que haja um término do programa com erro, já que o destruidor da classe é chamado assim que o objeto sai do escopo. Essas ferramentas ainda têm em comum o acesso a valores como o identificador do processo e o número de processos participando de um grupo. Porém, o OOPS realiza a comunicação de objetos sem a necessidade de pré-processamento, simplesmente necessitando que o objeto a ser enviado seja derivado de uma classe fornecida na biblioteca e que sejam implementados alguns métodos especiais para serialização do objeto. Também não há o envolvimento do programador com o sistema de passagem de mensagem utilizado pelo OOPS como ocorre com o Classdesc.

Citamos também as semelhanças do *framework* KeLP com o OOPS. As diversas classes dessas ferramentas podem ser especializadas utilizando-se herança ou composição para satisfazer as necessidades da aplicação. A proposta do KeLP também apresenta algumas abstrações de alto nível para que detalhes laboriosos de implementação paralela fiquem ocultados do programador da aplicação, sem que isso elimine a interferência deste programador no paralelismo. O KeLP usa células adicionais em seus contêineres para armazenar dados fantasmas usados

em algortimos de estêncil assim como o OOPS.

4

OOPS

Utilizamos a definição apresentada por Gamma, Helm, Johnson e Vlissides [12] de que um *framework* é um conjunto de classes cooperativas que torna reutilizável o projeto para um determinado tipo de problema. Um *framework* apresenta classes concretas e abstratas definindo suas responsabilidades e colaborações, e um usuário deste *framework* o configura para sua aplicação particular, derivando e compondo instâncias de classes presentes nele. Um *framework* pré-define parâmetros como particionamento da aplicação em classes ou a forma como elas colaboram, para que o usuário se concentre nas características específicas de sua aplicação.

O desenvolvimento de *frameworks* está imerso no escopo da programação genérica [10, 11], a qual auxilia o desenvolvimento de *softwares* confiáveis e de forma mais rápida por causa da reutilização de código. A programação genérica abstrai algoritmos eficientes para serem utilizados em diversas situações, oferecendo a oportunidade de reutilização de código de forma flexível. Tornou-se bem conhecida com a biblioteca em C++ de Stepanov e Lee, *Standard Template Library* (STL) [34]. A STL é uma biblioteca de *templates* que trata de estruturas de dados e algoritmos fundamentais da ciência da computação. Outras bibliotecas de *templates* em C++ cujos projetos seguem o paradigma de programação genérica podem combinar expressividade, reutilização e eficiência, e essas características são de grande importância para muitas aplicações científicas. Um exemplo é o

Matrix Template Library (MTL) [62].

Para viabilizar a programação genérica é importante a utilização de interfaces bem definidas [1, 12, 10, 11]. Dessa forma os diversos componentes de um projeto podem ser modificados ou desenvolvidos separadamente sem que as alterações de implementação interfiram na sua utilização, já que suas interfaces são preservadas.

Conforme discutido no capítulo 2, é interessante utilizar o modelo de multicomputador e o paradigma de passagem de mensagens na programação concorrente para execução paralela a fim de se tentar atingir portabilidade de desempenho em diversos sistemas paralelos. O multicomputador generaliza os sistemas paralelos atuais de forma satisfatória, e a passagem de mensagens ajuda a garantir localidade tornando explícitas todas as operações não-locais.

A avaliação das diversas abordagens existentes para a programação de sistemas paralelos, entre as quais algumas foram relacionadas no capítulo 3, sugere que uma boa solução deveria incluir técnicas de orientação a objetos para organizar o programa e encapsular detalhes que podem ser gerenciados sem a interferência direta do programador. Porém, o controle do programador da aplicação em alguns casos é essencial para alcançar eficiência do código, mesmo que isso acarrete no aumento da complexidade.

4.1 Proposta

Como visto no capítulo 1, o desenvolvimento de aplicações paralelas é bastante complexo. Além da implementação das computações como em um programa seqüencial, é necessário o gerenciamento das execuções em vários computadores e a coordenação das suas comunicações. Torna-se assim importante o desenvolvimento de ferramentas adequadas para auxiliar a programação paralela.

A proposta deste trabalho consiste no projeto e no desenvolvimento do *framework* de classes OOPS, *Object-Oriented Parallel System*, para o apoio da programação concorrente de aplicações científicas para execução paralela. Esta ferramenta, que se insere na proposta de Travieso [63], utiliza orientação a objetos para fornecer abstrações de alto nível de forma que o programador não tenha a necessidade de se envolver diretamente em detalhes de implementação do

paralelismo da sua aplicação. Entretanto, como será visto adiante, é deixado a seu cargo decisões sobre alguns aspectos paralelos por questões de eficiência da aplicação. Espera-se incentivar a programação de aplicações paralelas visto que minimizamos as dificuldades de sua implementação sem deixar de utilizar a capacidade de escolha do programador para garantir desempenho do código. A ênfase do OOPS reside em aplicações científicas com alta demanda computacional. Ele fornece um conjunto de classes para encapsular técnicas comumente utilizadas na programação de aplicações científicas paralelas.

As principais características da proposta do *framework* OOPS são:

Desempenho O desempenho da aplicação é uma questão sempre bastante importante em se tratando de programação paralela. Isso porque a decisão pelo paralelismo no desenvolvimento de uma aplicação advém da necessidade de extrair maior desempenho do sistema computacional. Dessa forma, as decisões de projeto do OOPS enfatizam essa necessidade, pois a utilização do *framework* não deve adicionar uma sobrecarga excessiva à execução da aplicação, acrescentando apenas uma camada fina entre o programa do usuário e o sistema de comunicação. O OOPS pretende facilitar a etapa de desenvolvimento da aplicação paralela sem que seu uso acarrete em grande perda de desempenho quando comparado com a utilização direta do sistema de passagem de mensagens.

Aplicações científicas O foco do OOPS são aplicações científicas com necessidade de alto desempenho que se utilizem de contêineres regulares, tais como vetores e matrizes, para armazenar seus dados. A utilização do *framework* auxilia na implementação e organização do código da aplicação na medida em que suas abstrações facilitam a expressão dos aspectos paralelos e definem sua interoperabilidade. As diversas classes fornecidas pelo OOPS encapsulam técnicas comumente utilizadas por esse tipo de aplicação científica, como distribuição dos dados pelos processadores disponíveis, reduções envolvendo valores distribuídos ou comunicação em variadas topologias. Com o conjunto de classes disponibilizado até o momento, o OOPS enfatiza o uso de paralelismo de dados, porém apresenta também constru-

ções para paralelismo de tarefas com o uso de componentes e a possibilidade de composição concorrente destes.

Nível de abstração O OOPS fornece ferramentas com alto nível de abstração em relação aos aspectos paralelos da programação, tais como classes modelando diversas formas de distribuição de dados e estruturas de vizinhanças para as comunicações entre os processadores. Com isso é minimizado o envolvimento do programador com detalhes de implementação que ajudam a tornar a programação paralela complexa e passível de erros. Porém, o paralelismo é *explícito*, ou seja, alguns aspectos paralelos de projeto, como por exemplo a decisão do particionamento mais adequado dos dados, são deixadas para o programador da aplicação. Desta forma é usado o discernimento do programador e o seu conhecimento da aplicação para buscar um melhor desempenho do código paralelo.

Reutilização de código A reutilização de código é um aspecto importante a ser levado em consideração no desenvolvimento de qualquer aplicação para que não se desperdice esforços. O OOPS é uma biblioteca de classes com a qual o programador continua usando uma linguagem padrão, no caso C++, adicionando ao seu código chamadas para a biblioteca. Assim, código pré-existente pode ser facilmente reaproveitado, o que poderia ser proibitivo no caso de novas linguagens. Além disso, fragmentos específicos de código podem ser implementados por especialistas para garantir eficiência, por exemplo distribuição e captação de dados, e o usuário do *framework* os utiliza para sua aplicação em particular.

Orientação a objetos A orientação a objetos apresenta bons fundamentos para a programação concorrente. Com herança e polimorfismo, a orientação a objetos fornece mecanismos para reutilização e extensão de código, sem necessariamente promover perda de desempenho. Conforme discutido, a programação de aplicações paralelas é ainda mais complexa do que a programação seqüencial por acrescentar a necessidade de gerenciamento da execução em diversos processadores e das comunicações entre estes, fazendo com que a reutilização de código e o encapsulamento de detalhes se tornem

ainda mais essenciais para este tipo de aplicação. O OOPS utiliza orientação a objetos para organizar a estrutura da aplicação e encapsular detalhes de implementação paralela, mantendo os conceitos de objetos e processos ortogonais.

Portabilidade É necessário que uma ferramenta seja projetada visando a portabilidade da aplicação que a utiliza para não haver perda significativa de desempenho da sua execução em diferentes sistemas paralelos. Dessa forma, a utilização da ferramenta não se tornará obsoleta com a alteração ou renovação do sistema computacional. Para tentar atingir esse objetivo, o OOPS usa o conceito de multicomputador genérico para modelar o sistema computacional e passagem de mensagens para comunicação entre os processos.

Interface O OOPS apresenta uma interface bem definida, com a qual o programador implementa sua aplicação paralela. Os detalhes de implementação das suas classes são ocultados para que alterações ou manutenções na ferramenta não acarretem em mudança subseqüente do código do usuário.

O OOPS se baseia no conceito de processadores virtuais os quais podem ser organizados em grupos. Sobre esses grupos podem ser construídas topologias, que fornecem os mecanismos de comunicação entre os processadores. Contêineres podem ter seus elementos distribuídos em uma topologia, e componentes paralelos podem executar fazendo uso de contêineres distribuídos. Todos esses conceitos básicos do OOPS são discutidos a seguir.

4.1.1 Processadores Virtuais

O primeiro conceito importante utilizado pelo OOPS é o de *processadores virtuais*. Eles representam os elementos disponíveis para a execução paralela, e podem ser associados a *processos* na máquina paralela. Dessa maneira mais de um processador virtual pode estar executando em um mesmo computador ou em computadores completamente distintos em um dado momento, de forma transparente à aplicação. Vale ressaltar que não há dependência do número de processadores virtuais com o número de processadores físicos utilizados.

O número de processadores virtuais assim como sua distribuição pelos processadores físicos são especificados fora da aplicação no momento em que esta é colocada para executar, e permanece constante até o final da execução. Neste contexto, o termo “processador” se refere a “processador virtual”. A constância no número de processadores virtuais pode ser revista para expandir as facilidades do *framework* com a criação dinâmica de tarefas, como discutido nas sugestões de trabalhos futuros, no capítulo 7.

4.1.2 Grupos de Processadores Virtuais

Os processadores virtuais são organizados em *grupos*, que são coleções destes processadores. Um grupo pode informar o número de processadores presentes, ou seja, o tamanho tam do grupo, e também o número de identificação único de um processador nesse grupo, que varia de 0 a $tam - 1$.

Os grupos podem ser subdivididos arbitrariamente em subgrupos, nos quais os processadores virtuais recebem um outro identificador.

4.1.3 Topologias

Sobre os grupos de processadores virtuais podem ser aplicadas *topologias*. Uma topologia sempre é construída sobre um grupo de processadores virtuais e fornece modos de comunicação por passagem de mensagens entre eles. Assim uma topologia no OOPS define uma estrutura de vizinhanças na qual os processadores pertencentes a um grupo são inseridos em posições determinadas.

As comunicações realizadas no OOPS podem ser *ponto-a-ponto* ou *coletivas*. As comunicações ponto-a-ponto são realizadas através de parceiros, e permitem a transmissão e recepção de dados entre dois processadores. Já as comunicações coletivas se dão entre os diversos processadores de um grupo.

Como os grupos podem ser subdivididos, as comunicações coletivas em uma topologia podem ser realizadas envolvendo o grupo inteiro de processadores ou somente uma parte dele. Por exemplo, uma topologia de grade bidimensional pode realizar uma comunicação entre todos os elementos da grade, ou entre os elementos de uma linha ou coluna através da formação de subgrupos para as

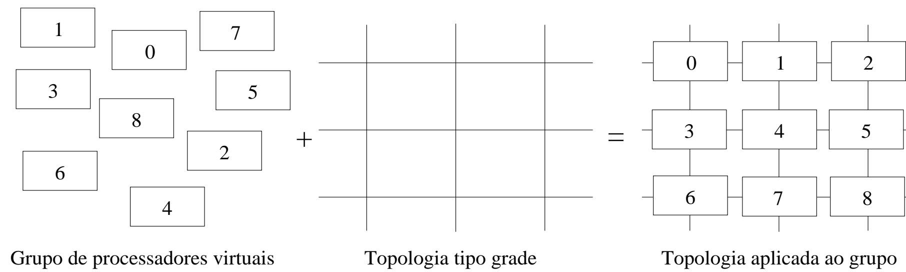

Figura 4.1: Diagrama da aplicação de uma topologia tipo grade bidimensional sobre um grupo de 9 processadores virtuais (os identificadores dos processadores variam de 0 a 8).

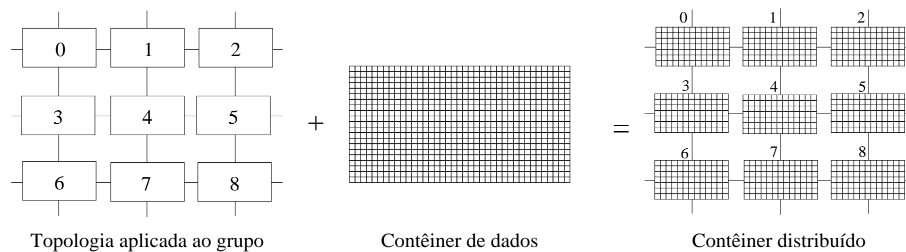

Figura 4.2: Diagrama da distribuição de um contêiner pelo o grupo de processadores virtuais na topologia tipo grade bidimensional.

linhas e colunas.

A figura 4.1 representa um diagrama de como uma topologia é aplicada a um grupo de processadores virtuais. Na figura, sobre um grupo de 9 processadores virtuais é aplicada uma topologia do tipo grade bidimensional, viabilizando a comunicação entre os processadores nesse arranjo. Note que as comunicações ponto-a-ponto são realizadas entre processadores vizinhos na topologia.

4.1.4 Contêineres Distribuídos

Um *contêiner distribuído* é aquele em que os elementos são distribuídos entre os diversos processadores de uma determinada topologia e que permite acesso aos seus elementos por todos esses processadores.

A figura 4.2 ilustra a distribuição de um contêiner pelo grupo de processadores virtuais na topologia tipo grade bidimensional. Cada processador do grupo recebe uma porção dos elementos do contêiner para realizar as computações concorrentemente sobre eles.

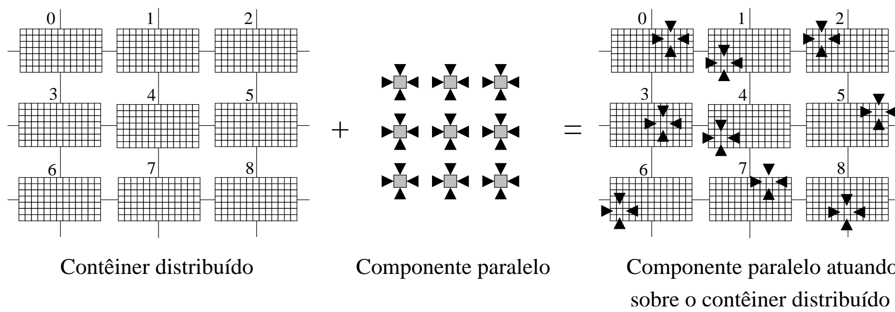

Figura 4.3: Diagrama da execução de um componente paralelo sobre o contêiner que foi distribuído pelo grupo de processadores virtuais na topologia tipo grade bidimensional.

4.1.5 Componentes Paralelos

Um *componente paralelo* fornece o código a ser executado em paralelo pelos processadores presentes em um grupo de processadores virtuais, e pode utilizar os contêineres distribuídos nesses processadores.

O diagrama da figura 4.3 representa a execução de um componente paralelo sobre o grupo de processadores virtuais na topologia tipo grade bidimensional. Este componente realiza as computações sobre os elementos do contêiner distribuído.

O uso dos componentes paralelos se dá por meio de *composições*, ou seja, componentes distintos podem ser combinados para formar uma aplicação paralela. As composições podem ser *seqüenciais* ou *concorrentes* [13].

Em uma composição seqüencial, componentes paralelos executam um após o outro fazendo uso do mesmo grupo de processadores virtuais. Para este caso, a troca de informações entre os diferentes componentes pode ocorrer através de variáveis e contêineres de dados, visto que esses valores se encontram nos processadores do grupo depois da execução de cada componente. Esse é o caso, por exemplo, de um cálculo de Hamiltoniano e em seguida sua diagonalização para encontrar seus autovalores, em que cada uma dessas tarefas representa um componente paralelo distinto. Após a execução do primeiro componente, o cálculo do Hamiltoniano, os dados estão armazenados nos contêineres distribuídos, e podem em seguida ser utilizados pelo segundo componente paralelo, a procura dos autovalores. A figura 4.4 ilustra essa situação, na qual dois componentes

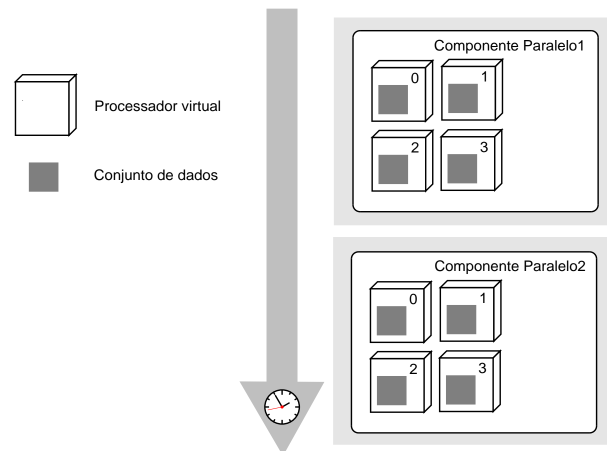

Figura 4.4: Ilustração da composição seqüencial. Dois componentes paralelos distintos atuam sobre o mesmo conjunto de dados utilizando o mesmo grupo de processadores virtuais.

distintos atuam sobre o mesmo conjunto de dados utilizando o mesmo grupo de processadores virtuais, um em seguida do outro.

Já na composição concorrente, onde componentes paralelos executam concorrentemente e utilizam grupos de processadores distintos, é necessária uma forma de comunicação de dados entre esses componentes. Isso ocorre no OOPS através de *intertopologias*, que fornecem modos de comunicação entre grupos diferentes de processadores virtuais. Pode-se exemplificar essa forma de composição com um sistema de aquisição e tratamento de dados em tempo real. Os componentes paralelos envolvidos são o sistema de aquisição e cada uma das etapas do tratamento de dados. Esses componentes devem executar concorrentemente, pois novos dados são capturados enquanto outros estão em diferentes estágios de tratamento. Cada um desses componentes paralelos se utiliza de um grupo próprio de processadores para sua execução, e deve passar seu conjunto de dados para o próximo componente assim que terminar para então receber do anterior o novo conjunto de dados a tratar. A composição concorrente é esquematizada na figura 4.5, na qual três componentes paralelos executam concorrentemente com seus próprios grupos de processadores virtuais e as ligações entre eles indicam as comunicações de dados.

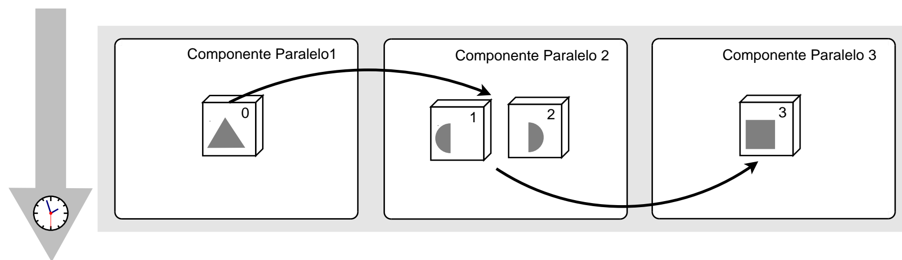

Figura 4.5: Ilustração da composição concorrente. Três componentes paralelos executam concorrentemente com seus próprios grupos de processadores virtuais e sobre diferentes conjuntos de dados em um determinado momento. As ligações entre os diferentes grupos de processadores virtuais indicam as comunicações entre eles.

4.2 Ferramentas Utilizadas

Linguagem de Programação A linguagem orientada a objetos utilizada para o desenvolvimento do *framework* OOPS é o C++ [14]. Esta é uma linguagem bastante difundida [64, 65], com a qual vários trabalhos relacionados foram desenvolvidos e que apresenta compiladores eficientes [6].

Sistema de Passagem de Mensagens Usamos o MPI, *Message Passing Interface* [40], como sistema de passagem de mensagens no desenvolvimento do OOPS. O MPI é um padrão de biblioteca de passagem de mensagens para sistemas paralelos e fornece uma base comum para o desenvolvimento de aplicações em plataformas distintas. Em virtude disso, ele incorpora aspectos de variados sistemas ao invés de adotar como padrão um sistema específico.

Pode-se utilizar a biblioteca MPI para especificar a comunicação entre conjuntos de processos formando um programa concorrente. O programa do usuário é escrito em uma linguagem seqüencial padrão, C++ por exemplo, com chamadas às funções da biblioteca MPI. Esta utiliza os protocolos de rede para enviar as mensagens entre os processos pela rede física.

Padrões como o MPI se ocupam de detalhes de baixo nível do funcionamento do programa, mais especificamente com o movimento de dados entre os processos concorrentes, o que torna a programação bastante complexa. Neste sentido, o MPI pode ser comparado às linguagens montadoras, onde o programador precisa

se ocupar com o movimento de dados entre as unidades de armazenamento (memória e registradores). São necessárias outras soluções que operem em um nível de abstração mais alto facilitando a programação paralela.

No MPI, um *processo* é o elemento básico de computação. A diferença de um processo MPI a um programa seqüencial é que aquele pode cooperar com outros processos MPI para a execução de uma tarefa. Os processos são interligados por meio de *comunicadores*, e é por meio destes que um processo pode enviar e receber *mensagens*. Uma mensagem significa a troca de informações entre os processos. Os dados enviados em mensagens têm tipos associados a eles (inteiro, número de ponto flutuante, etc.) e eventuais conversões de representação entre arquiteturas distintas serão realizadas automaticamente pelo MPI.

Dentro de um comunicador, os processos recebem um número de identificação, chamado *rank*, que é usado para determinar o parceiro da comunicação. As mensagens levam ainda um valor de *tag* para diferenciá-las de outras que possam estar circulando simultaneamente pelo sistema, inclusive pelo mesmo comunicador.

São possíveis comunicações *ponto a ponto*, ou seja, que envolvem um par de processos, e comunicações *coletivas*, que envolvem um grupo de processos. Mensagens de envio e recepção são exemplos de comunicações ponto a ponto. Entre as comunicações coletivas estão o *broadcast*, no qual um dado pode ser enviado a todos os processos, a barreira, que sincroniza os processos, e as reduções, em que são realizadas somatórias de valores ou procura por um máximo, entre outros.

A figura 4.6 ilustra em camadas um programa que utiliza o *framework* OOPS. Nesta figura, o OOPS é representado como uma camada fina entre o programa do usuário e o MPI para ressaltar que sua utilização não deve representar uma grande sobrecarga para a aplicação paralela para não comprometer seu desempenho.

Documentação Uma ferramenta como um *framework* deve apresentar uma documentação preparada cuidadosamente para viabilizar seu uso. Esta é sempre uma boa prática de programação visto que auxilia no entendimento e manutenção de um programa e se torna mais crítica no caso em que há a necessidade de um aprendizado prévio à sua utilização, como no caso de ferramentas para

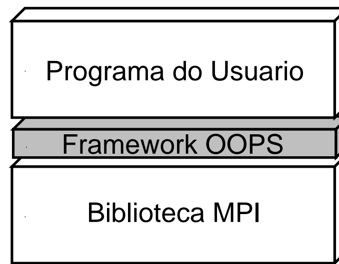

Figura 4.6: Esquema em camadas de um programa utilizando o *framework* OOPS. O OOPS representa uma camada fina entre o programa do usuário e a biblioteca MPI de forma a não representar uma grande sobrecarga para a aplicação e não comprometer seu desempenho.

programação.

Escolhemos o sistema Doxygen [66] para a preparação da documentação das classes do OOPS. Esta é uma ferramenta que auxilia no desenvolvimento de um manual concomitantemente à escrita do programa. Pode gerar arquivos em HTML ou LaTeX por exemplo, e também visualizações das dependências entre os vários elementos da biblioteca ou diagramas de herança, tudo automaticamente. Basta inserir blocos especiais no código do programa, que são compreendidos como comentários pelo compilador, mas de forma que o Doxygen consegue extrair informações para gerar a documentação.

Para uma documentação gráfica mais detalhada que a fornecida pelo Doxygen, utilizamos ferramentas de UML [67]. O UML provê uma estrutura gráfica para a organização de construções de projetos, adicionando uma modelagem visual bastante expressiva, evidenciando as colaborações entre classes, seus componentes e a estrutura do *framework*.

4.3 Desenvolvimento de Aplicações

Para o desenvolvimento de aplicações paralelas utilizando o OOPS, o programador da aplicação deve primeiramente encontrar a concorrência inerente ao seu problema. Ele deve verificar nos algoritmos a melhor forma de explorar o possível paralelismo da aplicação, estruturando seu projeto de forma a se beneficiar dos pontos de concorrência encontrados.

A partir de então, o programador deve verificar quais as abstrações fornecidas

pelo *framework* que ele utilizará para a escrita de seu programa, sendo esta etapa uma transição do algoritmo para o código. Ele se envolve diretamente com os aspectos paralelos de projeto, verificando quais estruturas oferecidas pelo OOPS modelarão sua aplicação, tomando decisões de particionamento de dados e padrões de comunicação dos processadores virtuais, por exemplo. Em suas decisões, o programador da aplicação deve sempre ponderar suas escolhas em relação ao desempenho de seu código.

O algoritmo deve ser implementado como um conjunto de componentes paralelos que fazem uso de estruturas de dados que serão implementadas como contêineres distribuídos. Deve ser feita uma análise de dependência de dados e balanceamento de cargas nas operações realizadas pelos componentes paralelos nos contêineres distribuídos e, assim, optar tanto pela topologia que será aplicada ao grupo de processadores virtuais (associados ao componente em questão), quanto pelo modo de distribuição dos elementos do contêiner, que sejam mais apropriados para manter localidade dos dados. O programador deve determinar se será usada composição concorrente ou seqüencial dos componentes paralelos que atuarão na aplicação, e caso seja composição concorrente, ele decide pela estrutura de comunicação entre os componentes para a construção da intertopologia.

Com essas diretrizes em mente, o programador escreve seu código utilizando os mecanismos de implementação fornecidos pelo OOPS. Ele deve verificar as especificações das interfaces das classes que utilizará, como estas classes facilitam a expressão do paralelismo da aplicação e pré-definem algumas de suas interações. O OOPS permite que o usuário se mantenha em um nível mais alto de abstração do que se utilizasse diretamente uma biblioteca de passagem de mensagens para o desenvolvimento da aplicação paralela, sendo que detalhes de implementação paralela são encapsulados nas classes que fornece.

4.4 Classes do OOPS

O *framework* OOPS fornece um conjunto de classes para facilitar a programação científica paralela. Apresentamos diversas delas nesta secção para possibilitar

o entendimento do seu funcionamento e de como se relacionam. Seus usos são exemplificados na secção seguinte.

4.4.1 Classes Básicas

Existe um conjunto de classes básicas do OOPS que são necessárias à biblioteca. Algumas dessas classes são indispensáveis a todo programa que utilize o *framework*. Outras são classes abstratas, que não podem ser instanciadas mas que servem de base para a implementação de características particulares, como por exemplo as topologias.

Nessa categoria de classes básicas estão:

`OOPS::Main` É necessário inicializar e finalizar a máquina virtual paralela. Isso é feito na classe `OOPS::Main` através de seus construtor e destruidor, respectivamente. Como citado no capítulo 3, isso faz com que a função de finalização do MPI seja chamada por todos os processos paralelos mesmo que haja um término do programa com erro. Um dos métodos dessa classe, `OOPS::Main::executeOn()`, é o ponto de entrada do programa paralelo.

`OOPS::Group` O grupo de processadores virtuais citado na secção 4.1.2 é modelado com a classe `OOPS::Group`. Esta apresenta as funcionalidades pertinentes a um grupo, como informações de seu tamanho (`OOPS::Group::size()`) e identificador do processador virtual (`OOPS::Group::myID()`). Também pode formar subgrupos através dos métodos `OOPS::Group::subGroup()` e `OOPS::Group::split()`.

`OOPS::Sendable` Para viabilizar a comunicação de objetos utilizando os métodos das classes do OOPS que tratam de comunicação, o *framework* fornece a classe base `OOPS::Sendable`. Todos os métodos de comunicação do OOPS são sobrecarregados para os tipos de dados básicos da linguagem e também para a `OOPS::Sendable`. Esta classe apresenta os métodos virtuais `OOPS::Sendable::pack()` e `OOPS::Sendable::unpack()` que devem ser implementados por suas classes derivadas.

`OOPS::Partner` Esta classe modela um parceiro em uma comunicação. Recebe em seu construtor o grupo e o identificador do processador virtual que será seu parceiro nas comunicações ponto-a-ponto. Apresenta os métodos `OOPS::Partner::send()` e `OOPS::Partner::recv()` para o envio e recepção de mensagens entre os parceiros.

`OOPS::Workgroup` É modelado um grupo de trabalho de processadores virtuais com esta classe. Um `OOPS::Workgroup` recebe um grupo de processadores virtuais no seu construtor e fornece diversos métodos de comunicação coletiva. Entre eles citamos a barreira (`OOPS::Workgroup::barrier()`), o *broadcast* (`OOPS::Workgroup::bcast()`), procura por máximo e mínimo (`OOPS::Workgroup::max()` e `OOPS::Workgroup::min()`, respectivamente), o somatório e o produtório de valores (métodos `OOPS::Workgroup::sum()` e `OOPS::Workgroup::prod()`, respectivamente), e dois métodos menos triviais, um para a distribuição e outro a arrecadação de frações contíguas de vetores (`OOPS::Workgroup::scatter()` e `OOPS::Workgroup::gather()`) bastante similares aos equivalentes em MPI.

`OOPS::Topology` Como visto, para viabilizar as comunicações entre os processadores virtuais pertencentes a um grupo é necessário aplicar uma topologia. A classe `OOPS::Topology` é uma classe abstrata que é a base para todas as possibilidades de topologias que se pode projetar. Esta classe fornece métodos para comunicações coletivas com todos os processadores presentes no grupo, equivalentes aos disponibilizados por `OOPS::Workgroup` citados acima, e suas derivadas fornecem a estrutura das comunicações ponto-a-ponto no arranjo que modelam. Esta classe apresenta também o método `OOPS::Topology::group()` que retorna um ponteiro para o grupo sobre o qual a topologia é construída.

`OOPS::Distribution` Os contêineres distribuídos fornecidos pelo OOPS apresentam modos de distribuição de seus elementos pelo grupo de processadores em uma topologia. A classe `OOPS::Distribution` é a base para esses modos de distribuição. Fornece métodos virtuais para conversão de índices locais para globais e vice-versa, `OOPS::Distribution::localToGlobal()`

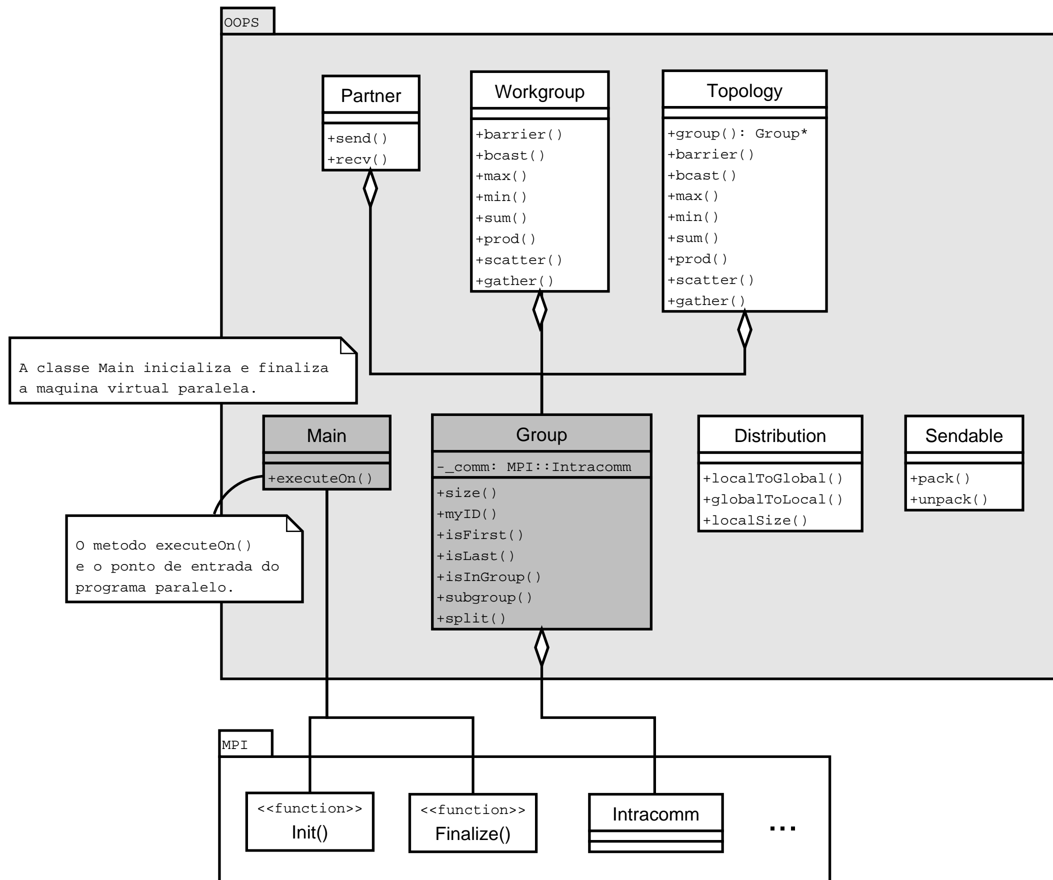

Figura 4.7: Diagrama das classes básicas do OOPS. Somente as classes `OOPS::Main` e `OOPS::Group` se relacionam diretamente com MPI. As ligações mostram que a topologia, o parceiro e o grupo de trabalho são construídos sobre um grupo de processadores virtuais.

e `OOPS::Distribution::globalToLocal()`, respectivamente, assim como um método para retornar o tamanho local do contêiner no processador (`OOPS::Distribution::localSize()`).

A figura 4.7 mostra um diagrama contendo as classes básicas do OOPS. As classes `OOPS::Partner` e `OOPS::Workgroup` podem ser usadas para o desenvolvimento de topologias e de intertopologias.

4.4.2 Classes Usadas no Tratamento de Erros

O tratamento de erros no OOPS é feito através de exceções. Para isso algumas classes foram criadas e são usadas caso o programa entre em um ponto de erro. As exceções são lançadas e o programador da aplicação decide quais ações devem

ser tomadas de acordo com suas especificações.

As classes do OOPS usadas no tratamento de erros são:

`OOPS::Error_NotPositive` Sinaliza erro para quando um valor estritamente positivo é esperado para uma variável.

`OOPS::Error_Negative` Classe que indica erro de um valor negativo passado. Este erro é sutilmente diferente do anterior, sendo usado quando é possível um valor positivo ou nulo para a variável em questão.

`OOPS::Error_OutOfRange` Se for realizada uma consulta fora de uma variação possível, tal como um índice possível para um vetor, essa operação resulta no envio da classe `OOPS::Error_OutOfRange`.

`OOPS::Error_NotEqual` Sinaliza a não igualdade de valores. Essa classe é utilizada em comparações de variáveis que caso não sejam iguais não é possível realizar uma operação. Por exemplo, a multiplicação de uma matriz por um vetor somente é executada quando o número de colunas da matriz é igual ao tamanho do vetor.

`OOPS::Error_NotAllowed` Sinaliza uma situação não permitida, como por exemplo a tentativa de comunicação sem um grupo válido na topologia.

`OOPS::Error_NotPossible` Indica a impossibilidade de realizar uma operação com os parâmetros passados, tal como formar uma grade completa de processadores apresentando um número de linhas fixo que não seja divisor do número de processadores.

Para facilitar o uso dessas classes para o tratamento de erros, o OOPS apresenta algumas funções *inline* parametrizadas prontas para serem usadas em testes que podem realizar o envio de exceções. Podemos citar as seguintes funções parametrizadas no tipo T de suas variáveis:

`OOPS::verifyPositive(const T &x)` que caso a variável x não seja positiva envia uma exceção to tipo `OOPS::Error_NotPositive`.

`OOPS::verifyRange(const T &x, const T &beg, const T &end)` que verifica se a variável x tem o valor entre *beg* e *end*, ou seja, se $beg \leqslant x \leqslant end$. Em caso negativo, envia a exceção `OOPS::Error_OutOfRange`.

`OOPS::verifyEqual(const T &x, const T &y)` averigua a igualdade das variáveis x e y, e em caso negativo, envia a exceção `OOPS::Error_NotEqual`.

entre outras.

Vale ressaltar que cabe ao usuário do OOPS recolher essas exceções lançadas e adequadar o tratamento do erro ocorrido de acordo com as especificações e necessidades de sua aplicação em particular.

4.4.3 Classes que Modelam Topologias

Conforme visto, a comunicação dos processadores virtuais em um grupo se dá através de topologias. É possível projetar diversas topologias tendo como base a classe básica `OOPS::Topology`, que herdam seus métodos de comunicação coletiva supra citados, e que implementam métodos para comunicação ponto-a-ponto e coletivas no arranjo.

Algumas possibilidades de padrões de comunicações encontradas comumente estão implementadas no OOPS, e outras podem ser adicionadas ao *framework* conforme a necessidade.

Citamos as seguintes topologias:

`OOPS::TopologyPlain` Esta topologia não apresenta uma estrutura de vizinhança entre os processadores. As comunicações ponto-a-ponto são realizadas entre qualquer par de processadores virtuais de um grupo usando simplesmente o indentificador do processador parceiro na comunicação através dos métodos `OOPS::TopologyPlain::send()` e `OOPS::TopologyPlain::recv()` para envio e recepção de dados, respectivamente. Este é o modo em que o MPI opera em suas comunicações ponto-a-ponto. As comunicações coletivas são realizadas envolvendo todos os processadores presentes no grupo. A figura 4.8(a) ilustra esta topologia.

`OOPS::TopologyPipe` Na topologia tipo duto ou tubo, os processadores virtuais do grupo são alinhados sequencialmente, de acordo com seu identificador p tendo dois vizinhos, um antecessor designado por $p-1$ e um sucessor identificado por $p+1$, com os quais pode realizar comunicações ponto-a-ponto. Porém o envio se dá em uma direção preferencial, ou seja, o envio é feito para o sucessor através do método `OOPS::TopologyPipe::toNext()` e a recepção de dados do antecessor com `OOPS::TopologyPipe::fromPrevious()`, conforme ilustrado na figura 4.8(b) onde as setas ligando os processadores têm uma só direção. Note que o primeiro processador virtual, com identificador zero, não apresenta antecessor e o último, com identificador igual ao tamanho do grupo menos um, não tem sucessor. As comunicações nesses extremos não têm efeito, ou seja, não são realizadas, e para esses casos é definido um parceiro nulo. As comunicações coletivas nessa topologia também envolvem todo o grupo de processadores virtuais.

`OOPS::TopologyPipeRing` Esta topologia modela um duto circular, figura 4.8(c). Herda as características da `OOPS:TopologyPipe` acrescentando que os extremos são ligados, ou seja, o processador virtual anterior ao primeiro é o último e o próximo ao último é o primeiro.

`OOPS::TopologyLinear` A topologia linear também herda as características de `OOPS::TopologyPipe` acrescentando a possibilidade de comunicação nas duas direções através dos métodos `OOPS::TopologyLinear::fromNext()` e `OOPS::TopologyLinear::toPrevious()`. Dessa forma um processador virtual nesse arranjo pode enviar e receber de seus vizinhos, como ilustrado na figura 4.8(d) na qual as setas ligando os processadores virtuais apresentam duas direções.

`OOPS::TopologyRing` Na topologia em anel modelada por `OOPS::TopologyRing`, todas as funcionalidades de `OOPS::TopologyLinear` são mantidas, só que com os extremos ligados como em `OOPS::TopologyPipeRing`, ou seja, é permitida a comunicação em qualquer direção em um arranjo circular. Essa topologia é ilustrada na figura 4.8(e).

`OOPS::TopologyGrid` Nesta topologia é modelada uma estrutura tipo grade bidimensional. Com este arranjo para os processadores virtuais são viabilizadas as comunicações ponto-a-ponto com os processadores ao norte, sul, leste, oeste, nordeste, noroeste, sudeste e sudoeste, como apresentado na figura 4.8(f) com as ligações entre os processadores virtuais (diversas ligações foram suprimidas da figura para facilitar a visualização). Os métodos fornecidos para as comunicações de envio e recepção de dados são por exemplo `OOPS::TopologyGrid::toNorth()`, `OOPS::TopologyGrid::fromNorth()`, `OOPS::TopologyGrid::toNW()`, e `OOPS::TopologyGrid::fromNW()`, etc. Os processadores nas bordas da grade não apresentam os vizinhos em todas essas direções, e, assim como nas topologias `OOPS::TopologyPipe` e `OOPS::TopologyLinear`, comunicações nas direções sem vizinhos não têm efeito. No modo padrão a grade é formada de maneira a ser o mais quadrada possível, sendo o número de linhas menor ou igual ao de colunas, havendo também a possibilidade de especificar o número de linhas da grade. Apresenta ainda outros métodos para fornecer informações sobre suas linhas e colunas. Por exemplo os métodos `OOPS::TopologyGrid::rowSize()` que retorna o tamanho da linha da grade, ou seja, quantos processadores há em uma linha, `OOPS::TopologyGrid::rowPosition()` retornando a linha que o processador ocupa na grade, `OOPS::TopologyGrid::row()` que retorna a linha inteira de processadores como um objeto da classe `OOPS::TopologyLinear` para comunicações nesta subtopologia, e os equivalentes para as colunas.

`OOPS::TopologyTorus` Também é fornecida a topologia modelando um toróide no OOPS. A classe `OOPS::TopologyTorus` apresenta todas as características da topologia tipo grade com a adição de vizinhos para os processadores das bordas, como nos arranjos circulares. Dessa forma, essa topologia é uma grade circular, ilustrada na figura 4.8(g). Na figura, assim como ocorre para a ilustração da topologia tipo grade, algumas ligações entre os processadores virtuais que indicam a possibilidade de comunicação ponto-a-ponto não são apresentadas para facilitar o entendimento.

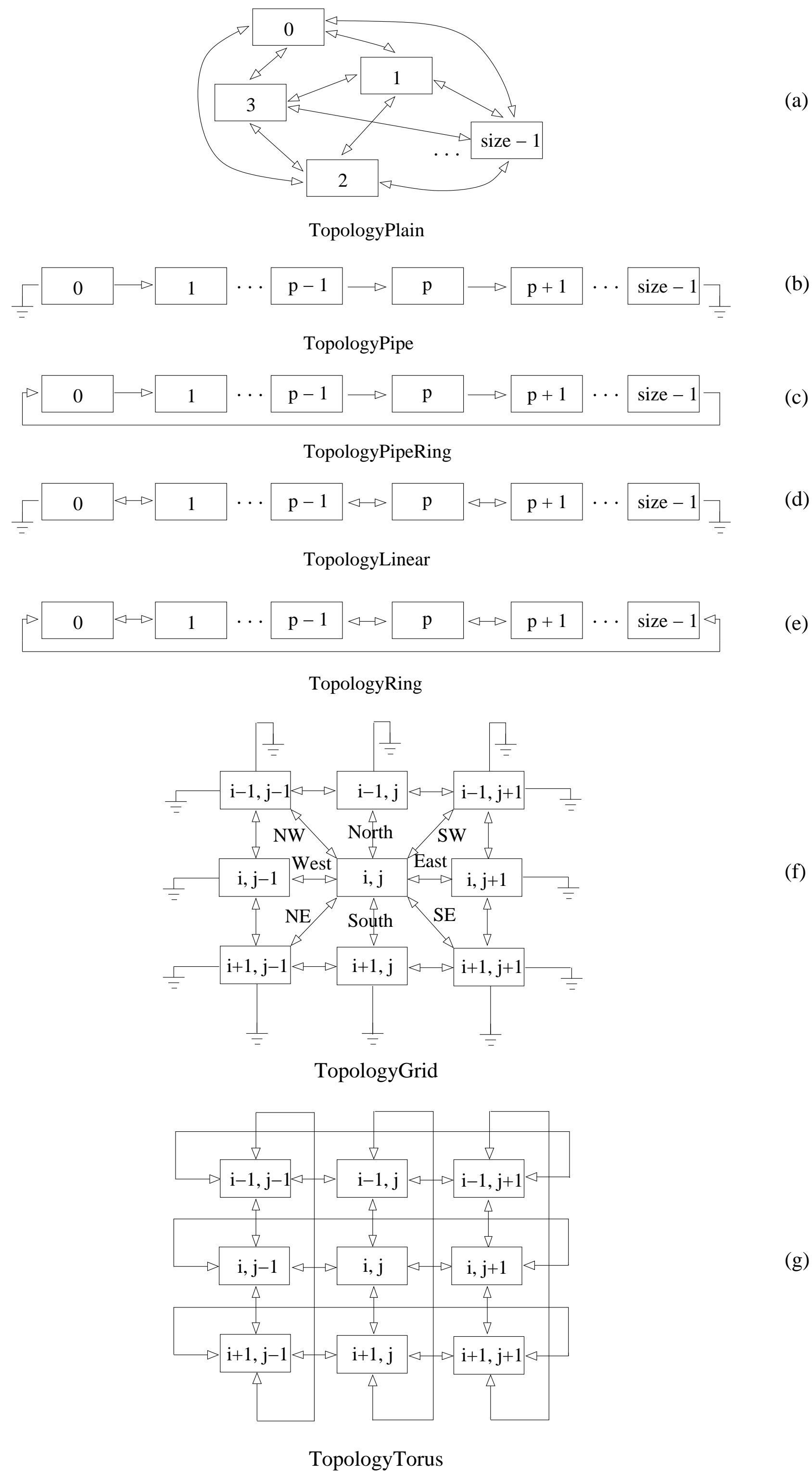

Figura 4.8: Ilustração das diversas topologias apresentadas. É possível ver as estruturas de vizinhanças usadas nas comunicações ponto-a-ponto entre os processadores vituais.

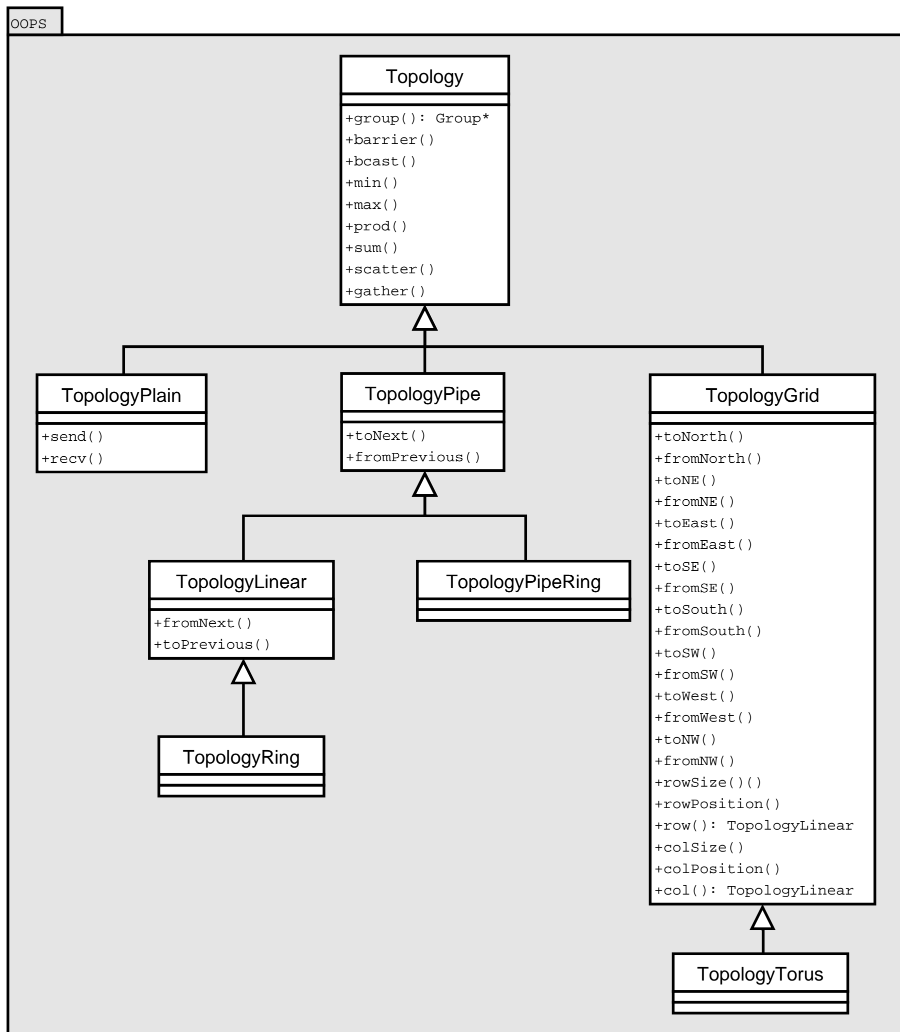

Figura 4.9: Diagrama da herança de classes que modelam topologias. As setas na figura partem da classe derivada para a classe base.

Poderiam ainda ser fornecidas outras topologias modelando grades tri-dimensionais, hipercubo ou árvores binárias, por exemplo.

A figura 4.9 mostra um diagrama de herança para as topologias fornecidas pelo OOPS. Das topologias presentes na figura somente `OOPS::Topology` não pode ser instanciada por se tratar de uma classe abstrata.

4.4.4 Modos de Distribuição dos Contêineres

No OOPS existe a necessidade do programador decidir a forma de particionamento dos seus dados. Devem ser especificados os modos de distribuição para cada dimensão do contêiner distribuído da mesma forma que em HPF. Na sec-

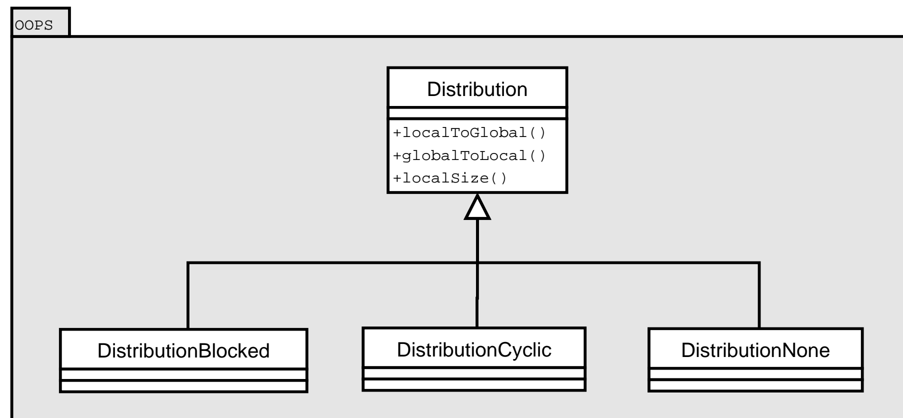

Figura 4.10: Diagrama da herança de classes que modelam modos de distribuição dos contêineres. As setas na figura partem da classe derivada para a classe base.

ção 5 que traz exemplos do uso das classes do OOPS, se torna mais claro como as combinações dos modos de distribuição podem acarretar em localidade dos dados.

No presente momento, algumas distribuições comuns estão disponíveis, e implementam os métodos da classe base `OOPS::Distribution`. São elas:

`OOPS::DistributionBlocked` Esse modo de distribuição indica que a distribuição dos dados deve ser feita em blocos consecutivos de tamanhos iguais, e caso o número de processadores não for divisor do tamanho da dimensão do contêiner, então alguns processadores receberão um elemento a mais que os outros.

`OOPS::DistributionCyclic` Há a possibilidade de distribuir os elementos do contêiner ciclicamente, ou seja, blocos consecutivos de tamanhos especificados são distribuídos pelos processadores virtuais de forma cíclica. No modo padrão o bloco é de tamanho 1. Este modo de distribuição é interessante para quando não há dependência dos dados.

`OOPS::DistributionNone` Também é possível especificar que uma dimensão não deve ser distribuída com o modo de distribuição `OOPS::DistributionNone`.

A figura 4.10 apresenta o diagrama de herança das classes de modos de distribuição, e a figura 4.11 ilustra o resultado de diferentes modos de distribuição de um contêiner em duas dimensões.

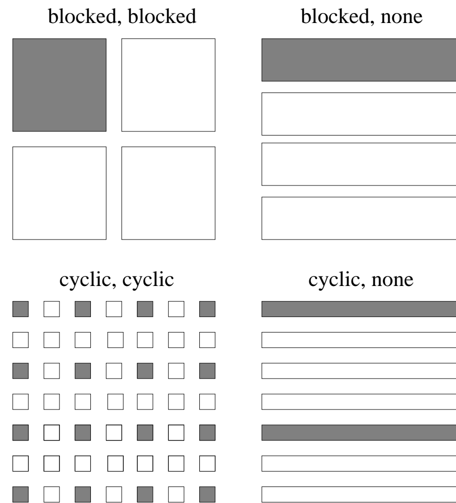

Figura 4.11: Diferentes modos de distribuição aplicados em um contêiner em duas dimensões para um grupo de quatro processadores virtuais. Variadas combinações são possíveis, alterando a distribuição dos dados pelos processadores virtuais.

4.4.5 Classes de Contêineres Distribuídos

Diversas aplicações paralelas utilizam coleções de dados distribuídos, e por esse motivo, disponibilizar contêineres distribuídos é importante. Vetores e matrizes são muito comuns em aplicações científicas, e o OOPS fornece contêineres para facilitar o uso e a distribuição de seus elementos.

Com o conhecimento das dependências dos dados e com o intuito de manter a localidade destes, o programador da aplicação decide sobre o melhor particionamento de dados utilizando combinações apropriadas da topologia do grupo de processadores virtuais e do modo de distribuição dos contêineres. Deixar este tipo de decisão para o programador aumenta a complexidade da programação, porém também aumenta a chance de atingir melhores desempenhos para a aplicação [2, 13]. Conforme acreditamos, manter o paralelismo explícito e contar com o discernimento do programador na escolha do particionamento dos dados é um fator crucial para alcançar uma maior eficiência do código.

Os contêineres do OOPS são parametrizados no tipo T do elemento armazenado, onde este deve ser um tipo básico ou derivado de `OOPS::Sendable`, e estão abaixo relacionados:

`OOPS::Vector<T>` Esta classe modela um vetor unidimensional, de dimensão es-

pecificada pelo usuário, assim como sua distribuição por uma dada topologia. Apresenta diversos métodos usados freqüentemente com esse tipo de estrutura, como OOPS::Vector<T>::size() que retorna seu tamanho global, OOPS::Vector<T>::localSize() para informar o tamanho local do vetor no processador virtual, acesso a elementos através de índices locais com OOPS::Vector<T>::local(), indexação através do operador [] também com índices locais, somatória dos elementos com OOPS::Vector<T>::sum(), entre outros. Também fornece métodos que realizam conversão de índices, tanto local para global OOPS::Vector<T>::localToGlobal(), como global para local OOPS::Vector<T>::globalToLocal().

Um vetor pode apresentar em sua estrutura local um conjunto de elementos a mais em suas bordas para armazenar elementos consecutivos pertencentes a processadores vizinhos, elementos estes chamados comumente de *fantasmas* ou *sombras*. Essa medida visa facilitar algoritmos do tipo estêncil, como exemplificado na secção 5.4 e em [3]. O tamanho deste bloco de elementos pode ser especificado na construção do contêiner e para realizar a busca dos valores atuais nos vizinhos, esta classe fornece o método OOPS::Vector<T>::syncGhosts().

OOPS::VectorRepl<T> Para quando há a necessidade de replicar um vetor por todos os processadores virtuais de um grupo foi implementada a classe OOPS::VectorRepl. Para evitar problemas de coerência de dados entre as diversas cópias replicadas, para esse tipo de vetor não são fornecidos métodos para modificação de seus elementos após a inicialização dos dados. Ou seja, é vedada a possibilidade de escrita sendo fornecido acesso apenas à leitura. Assim como o vetor distribuído, apresenta um método para retornar seu tamanho (OOPS::VectorRepl<T>::size()), indexação através de [], e o retorno da somatória de seus elementos, entre outros.

OOPS::Matrix<T> Matrizes bidimensionais distribuídas são modeladas por esta classe no OOPS. Para este contêiner é necessário que o usuário especifique os mesmos parâmetros de construtor do que para o vetor distribuído, mas nas duas dimensões (a menos da topologia). Apresenta métodos para

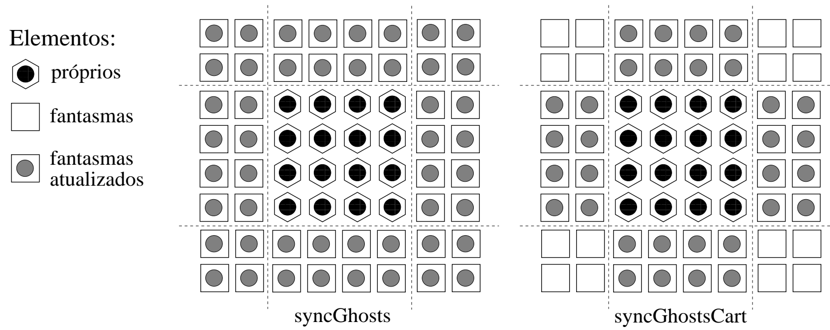

Figura 4.12: Ilustração das atualizações dos elementos fantasmas em uma matriz distribuída utilizando os métodos OOPS::Matrix<T>::syncGhosts() e OOPS::Matrix<T>::syncGhostsCart().

cada dimensão da matriz correlatos aos do vetor, retornando a dimensão das linhas e das colunas globais com OOPS::Matrix<T>::rowSize() e OOPS::Matrix<T>::colSize(), e OOPS::Matrix<T>::localRowSize() e OOPS::Matrix<T>::localColSize() para as dimensões locais. O acesso a elementos é realizado através do método OOPS::Matrix<T>::local() com o uso de índices locais. Visto que a matriz envolve duas dimensões, ocorre o retorno de um objeto do tipo OOPS::Matrix<T>::Index na conversão de índice local para global em OOPS::Matrix<T>::localToGlobal(), assim como de global para local com OOPS::Matrix<T>::globalToLocal().

Da mesma forma que ocorre com o vetor, a matriz distribuída no OOPS fornece a facilidade de armazenar elementos fantasmas em suas bordas e buscar seus valores atuais nos processadores remotos. Além do método OOPS::Matrix<T>::syncGhosts() que busca elementos em todos os vizinhos, existe o método OOPS::Matrix<T>::syncGhostsCart() que atualiza somente os elementos nas direções cartesianas, conforme ilustrado na figura 4.12. Esse último método é particularmente útil para diminuir as comunicações quando o estêncil utilizado envolve somente os dados nessas direções.

Para a leitura e escrita dos dados dos contêineres tipo OOPS::Vector<T> e OOPS::Matrix<T>, são fornecidos os métodos load() e store(), ou a sobrecarga dos operadores de extração ($\gg$) e inserção ($\ll$), respectivamente. Esses métodos realizam uma escrita padrão dos elementos, colocando-os em seqüência em for-

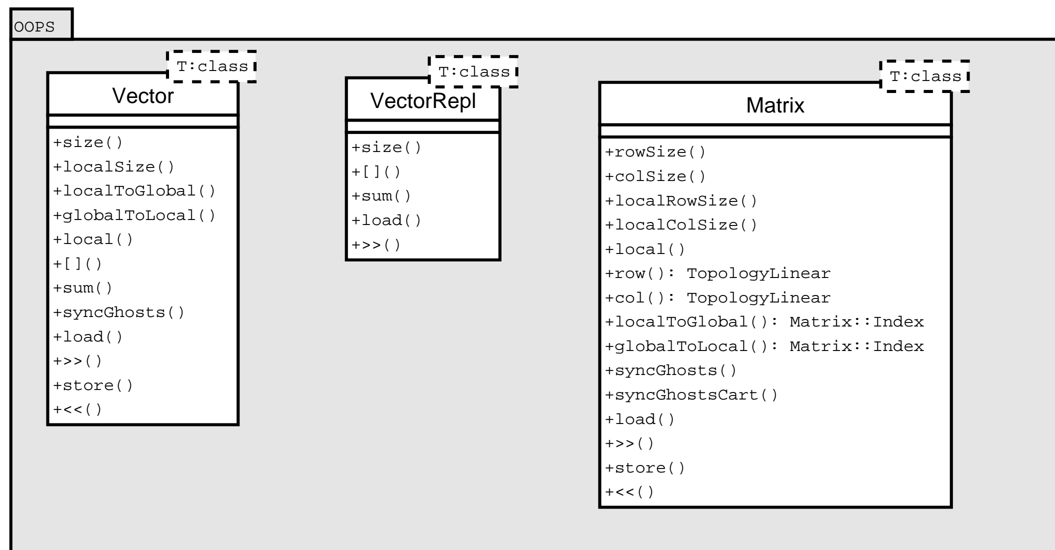

Figura 4.13: Diagrama das classes que modelam os contêineres. Todos os contêiners do OOPS são parametrizados no tipo T de seus elementos.

mato ASCII. Porém, caso o programador da aplicação queira adequar a entrada e saída de seus dados de acordo com uma formatação particular de seus arquivos, é possível passar um ponteiro para uma função na construção do contêiner a qual deve realizar a leitura ou escrita dos dados do arquivo da forma desejada. Dessa maneira, os métodos do OOPS realizam a distribuição ou captação dos elementos distribuídos usando a função do usuário para a leitura e escrita dos dados. Para o contêiner `OOPS::VectorRepl<T>` isso somente é válido para os métodos de leitura.

Ainda diversos outros métodos podem ser introduzidos na interface desses contêineres sem prejuízo aos métodos já existentes, somente aumentando a gama de funcionalidades providas. Como exemplos podemos citar o produto externo para o vetor, a inversão e transposta para a matriz.

4.4.6 Intertopologia

Conforme descrito anteriormente, para realizar comunicações entre os componentes paralelos em composições concorrentes, é necessário viabilizar a comunicação entre grupos distintos de processadores. Essa comunicação ocorre no OOPS através de intertopologias, modeladas a partir da classe `OOPS::InterTopology`:

`OOPS::InterTopology` Classe base para construção de intertopologias. Suas de-

rivadas devem fornecer modos de comunicação entre processadores pertencentes a grupos distintos.

Neste contexto citamos também a função parametrizada no tipo T da classe do componente, `OOPS::Execute<T>()`, que é disponibilizada para auxiliar o programador da aplicação na inicialização dos diversos componentes paralelos. Cada classe que modela um componente deve apresentar um método estático denominado `T::executeOn()` recebendo tanto um ponteiro para o grupo que irá executar o componente como um ponteiro para a intertopologia que deverá ser utilizada na comunicação ente eles. Esse método `T::executeOn()` será chamado por `OOPS::Execute<T>()`.

4.4.7 Interoperabilidade das Classes do OOPS

Nas secções anteriores, diversas classes presentes no OOPS, assim como alguns de seus métodos, foram apresentadas. É importante também discutir a forma que elas interoperam para formar, juntamente com o código do usuário, a aplicação paralela. A figura 4.14 apresenta um diagrama que ilustra as classes do OOPS e seus relacionamentos. Nesta figura os blocos que simbolizam as classes não foram expandidos mostrando seus membros e métodos para evidenciar somente suas ligações. Para simplicidade da ilustração, os contêineres distribuídos foram representados por uma só classe genérica e fictícia `OOPS::Distributed Container` assim como as classes de tratamento de erro, representadas por `OOPS::Error Treatment`. Além disso, somente as classes base das topologias e dos modos de distribuição dos contêineres foram representadas na figura.

As classes `OOPS::Main` e `OOPS::Group` são as únicas que tratam diretamente dos aspectos do MPI. Com isso, a dependência do OOPS com a biblioteca de passagem de mensagens utilizada ficou bastante restrita. `OOPS::Group` modela o grupo de processadores virtuais, e sobre esse grupo são construídas as formas de comunicações entre os processadores. Assim, podemos ver pela figura que `OOPS::Partner`, `OOPS::Workgroup`, `OOPS::Topology` e `OOPS::InterTopology` são classes que apresentam dependência com `OOPS::Group`. Todas as comunicações foram implementadas com sobrecarga no tipo do dado de envio para os tipos

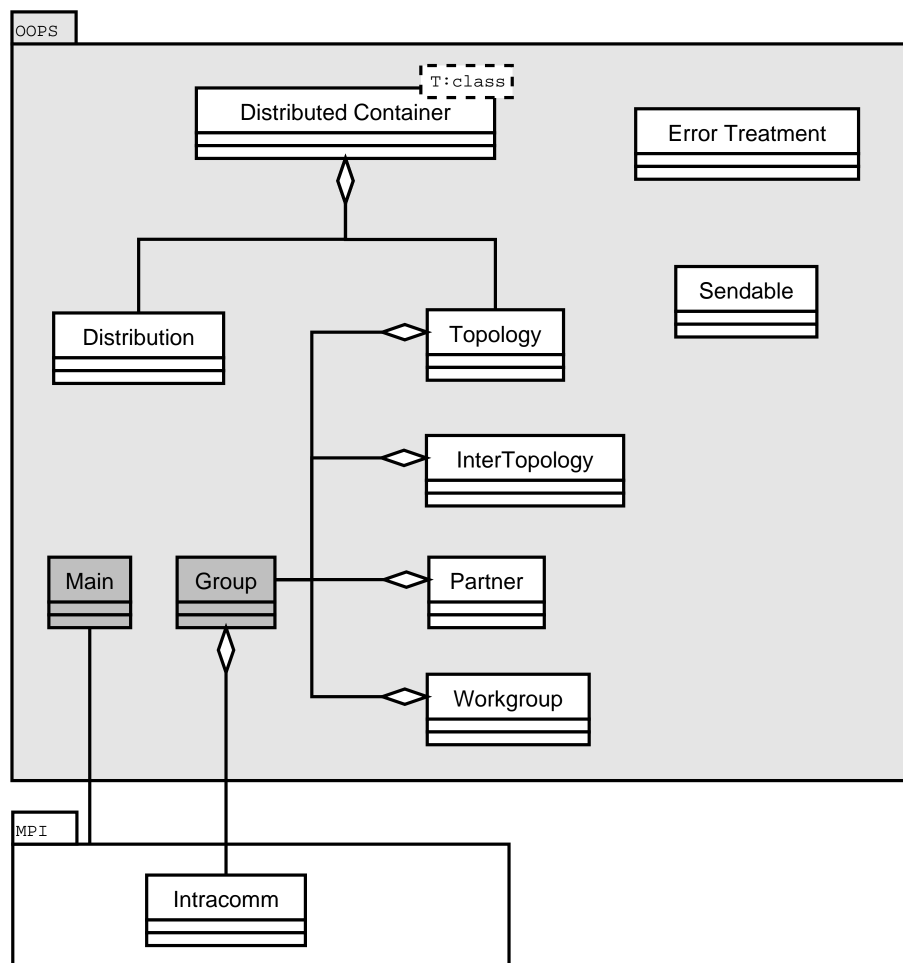

Figura 4.14: Diagrama da interoperabilidade das classes do OOPS.

básicos e para objetos `OOPS::Sendable`.

Os contêineres distribuídos necessitam de uma topologia e um modo de distribuição para sua construção. As possíveis combinações destes decidem o particionamento dos dados e, consequentemente, a localidade dos dados. Estes contêineres são parametrizados no tipo T de seus elementos.

Todo o tratamento de erros no OOPS é realizado através de exceções, indicado pela classe `OOPS::Error Treatment`. As exceções lançadas devem ser tratadas pelo usuário do *framework* de acordo com suas especificações.

4.5 Relacionamento com MPI

Para evidenciar a forma de relacionamento do *framework* OOPS com a biblioteca de passagem de mensagens, o MPI, colocamos nessa secção alguns exemplos das implementações de classes e métodos que foram apresentados acima.

4.5.1 Inicialização da Máquina Virtual Paralela

Conforme visto, a inicialização e finalização da máquina virtual paralela com os processos MPI se dão dentro do construtor e destruidor da classe `OOPS::Main`. Dessa forma:

```
1  // Construtor
2  Main::Main(int &argc, char **&argv) {
3    MPI::Init(argc, argv);
4  }
5  // Destruidor
6  Main::~Main() {
7    MPI::Finalize();
8  }
```

4.5.2 Comunicador MPI

A classe `OOPS::Group` apresenta um intracomunicador MPI como um de seus membros. Há uma relação direta entre os processadores virtuais do OOPS e os processos MPI.

```
class Group {
  MPI::Intracomm _comm;
  int _id; // Identificador do processador virtual
  int _n;  // Tamanho do grupo
  ... continuação da declaração da classe ...
};
```

Um dos construtores da classe OOPS::Group, apresentado no quadro a seguir, duplica o comunicador MPI::COMM_WORLD em _comm, e utiliza os métodos MPI::Intracomm::Get_rank() e MPI::Intracomm::Get_size() para ajustar os outros membros da classe.

```
// Construtor
Group::Group() {
  _comm = MPI::COMM_WORLD.Dup();
  _id = _comm.Get_rank();
  _n = _comm.Get_size();
}
```

4.5.3 Comunicações entre os Processos MPI

Por questão de escolha de implementação e para facilidade de manutenções, somente as classes OOPS::Main e OOPS::Group realizam chamadas diretamente ao MPI. Dessa forma restringimos a um pequeno número de classes do *framework* a dependência da biblioteca de passagem de mensagens utilizada. É possível trocar a biblioteca de passagem de mensagens usada pelo OOPS sem alterações em diversas de suas classes.

Como discutido, para viabilizar as comunicações entre os processadores virtuais no OOPS, é necessário utilizar uma topologia. Porém é um grupo do OOPS que realmente executa chamadas de comunicação do MPI já que é a classe que apresenta um intracomunicador do MPI, como mostrado acima. Assim, as comunicações são realizadas através de métodos privados de OOPS::Group, ou seja, não fazem parte de sua interface, e a classe base OOPS::Topology é declarada sua amiga, lhe sendo permitido, portanto, acessar métodos privados do grupo.

Podemos ver essa relação no fragmento de código a seguir da declaração da classe OOPS::Group e a implementação dos métodos apresentados:

```
class Group {
  ... declaração de membros ...
  void send(const int &x, int partner, int tag) const;
  void send(const int *v, int n, int partner, int tag) const;
  void max(int &x) const;

  friend class Topology;
public:
  ... continuação da declaração da classe ...
};

void Group::send(const int &x, int partner, int tag) const {
  if (_comm == MPI::COMM_NULL)
    throw Error_NotAllowed();
  _comm.Send(&x, 1, MPI::INT, partner, tag);
}

void Group::send(const int *v, int n, int partner, int tag) const {
  if (_comm == MPI::COMM_NULL)
    throw Error_NotAllowed();
  _comm.Send(v, n, MPI::INT, partner, tag);
}

void Group::max(int &x) const {
  if (_comm == MPI::COMM_NULL)
    throw Error_NotAllowed();
  int y;
  _comm.Allreduce(&x, &y, 1, MPI::INT, MPI::MAX);
  x = y;
}
```

com o membro `_comm` sendo do tipo MPI::Intracomm, como mostrado anteriormente. Note que os métodos `Send` e `Allreduce` no quadro acima fazem parte do escopo do MPI. Vemos também que antes das chamadas para os métodos de comunicação do MPI, é testada a existência do comunicador MPI. Em caso negativo é sinalizado um erro do tipo `OOPS::Error_NotAllowed`.

Comunicações coletivas As comunicações coletivas fazem parte da interface de `OOPS::Topology` e são herdadas por todas as topologias derivadas. Veja que no trecho de código a seguir, o membro `_group` é um ponteiro para `OOPS::Group` e dessa forma a topologia não realiza chamadas diretamente ao MPI:

```
class Topology {
  const Group *_group;
```

```
3  protected:
4    ...
5    void send(const int &x, int partner, int tag) const;
6    void send(const int *v, int n, int partner, int tag) const;
7    ...
8  public:
9    ...
10   void max(int &x) const;
11   ... continuação da declaração da classe ...
12 };
13
14 void Topology::send(const int &x, int partner, int tag) const {
15   _group->send(x, partner, tag);
16 }
17
18 void Topology::send(const int *v, int n, int partner, int tag) const {
19   _group->send(v, n, partner, tag);
20 }
21
22 void Topology::max(int &x) const {
23   _group->max(x);
24 }
```

Comunicações ponto-a-ponto As comunicações ponto-a-ponto nas topologias do OOPS apresentam uma gama de parceiros possíveis. Conforme visto, para `OOPS::TopologyPlain`, é possível especificar qualquer processador virtual pertencente ao grupo como parceiro de uma comunicação. Entretanto, para as outras possibilidades de topologias, como por exemplo `OOPS::TopologyPipe`, os parceiros são pré-definidos como os processadores adjacentes no arranjo, conforme visto anteriormente, seu antecessor e seu sucessor.

Assim temos:

```
1  class TopologyPipe : public Topology {
2  protected:
3    int _previous, // Identificador do antecessor
4      _next;       // Identificador do sucessor
5    int _tag_toNext,      // valor de tag para comunicação com sucessor
6      _tag_from_Previous; // valor de tag para comunicação com antecessor
7  public:
8    TopologyPipe(const Group *procs);
9    void toNext(const int &x) const;
10   void toNext(const int *v, int n) const;
11   ... continuação da declaração da classe ...
12 };
```

```
13
14  // Construtor
15  TopologyPipe::TopologyPipe(const Group *procs) : Topology(procs) {
16    _previous = _group->myID() - 1;
17    _next = _previous + 2;
18    if (_group->isFirst())
19      _previous = PARTNER_NULL;
20    else if (_group->isLast())
21      _next = PARTNER_NULL;
22
23    _tag_toNext = (_group->myID() * 100) + (_next * 10);
24    _tag_fromPrevious = (_previous * 100) + (_group->myID() * 10);
25  }
26
27  void TopologyPipe::toNext(const int &x) const {
28    Topology::send(x, _next, _tag_toNext);
29  }
30
31  void TopologyPipe::toNext(const int *v, int n) const {
32    Topology::send(v, n, _next, _tag_toNext);
33  }
```

No fragmento de código acima vemos que os membros `_next`, `_previous`, `_tag_toNext` e `_tag_fromPrevious` da classe `OOPS::TopologyPipe` são ajustados na construção do objeto. O método `OOPS::Group::myID()`, linhas 23 e 24, retorna o identificador do processador virtual dentro do grupo, e os métodos nas linhas 18 e 20, `OOPS::Group::isFirst()` e `OOPS::Group::isLast()`, retornam um valor booleano verdadeiro para o primeiro e o último processadores virtuais, respectivamente. Uma chamada para `OOPS::TopologyPipe::toNext()` já tem definido tanto o parceiro como o *tag* da comunicação passados para o método `OOPS::Group::send()` que realiza de fato a chamada MPI. Para topologias envolvendo comunicações em mais direções, o valor de *tag* leva em consideração estas direções.

4.6 Considerações

Para um programador desenvolver uma aplicação paralela com código eficiente e se aproveitar da disponibilidade de sistemas paralelos são necessárias ferramentas para apoiar a programação concorrente.

Neste capítulo foi apresentado o OOPS, um *framework* de classes que pretende

auxiliar o desenvolvimento de aplicações científicas paralelas. O OOPS é uma ferramenta direcionada a aplicações científicas com necessidade de alto desempenho que se utilizem de contêineres de dados regulares para armazenar seus dados. Enfatiza o uso de paralelismo de dados e apresenta construções para paralelismo de tarefas. Espera-se que o *framework* seja de fácil uso e aprendizado, situando-se em um nível de abstração intermediário, entre a biblioteca de passagem de mensagens e o modelo matemático. O OOPS mantém o paralelismo explícito por questões de desempenho da aplicação, e por isso várias decisões ficam a cargo do usuário do OOPS, como a topologia usada ou o modo de distribuição de um contêiner.

Algumas operações ainda precisam ser adicionadas ao *framework*. Por exemplo, quando ocorrer tentativas de operações utilizando contêineres com distribuições distintas, espera-se uma sinalização de erro. Neste caso, o programador da aplicação deve redistribuir explicitamente um dos operandos, alinhando os contêineres para depois realizar a operação. Uma redistribuição de um contêiner é uma operação cara computacionalmente pela grande quantidade de comunicação envolvida, tornando o uso de redistribuições automáticas desaconselhável. Seria também interessante que houvesse um método para conversão de vetores distribuídos para replicados e vice-versa. Essas e outras possíveis extensões somente aumentam a gama de funcionalidades do OOPS, não alterando a interface atual.

5 Exemplos

Neste capítulo exemplificamos o uso do OOPS na escrita de programas concorrentes. Com isso pretendemos demonstrar o uso do *framework* sem contudo fornecer uma revisão completa das suas funcionalidades.

5.1 Início da Execução

Como visto na secção 4.1.1, o número de processadores virtuais disponíveis para uma aplicação paralela é designado no momento em que ela é colocada para executar. Assim o OOPS cria um grupo com todos esses processadores e executa o método `executeOn()` da classe `OOPS::Main`. Esse método recebe um ponteiro para esse grupo criado pelo OOPS (`allProcs` de `OOPS::Group`) e é o ponto de entrada do programa paralelo. A classe `OOPS::Main` é o componente paralelo principal e o usuário do *framework* deve fornecer a implementação do seu método `executeOn()`.

Abaixo, um trecho de código exemplifica uma versão paralela do "Hello, world!" usando o OOPS:

```
1 #include <iostream>
2 #include <oops>
3 void OOPS::Main::executeOn(const OOPS::Group *allProcs) {
4   if (allProcs->myID() == 0)
5     std::cout << "Hello, world!" << std::endl;
6 }
```

nesse código como nos próximos, é incluído o arquivo `<oops>` que apresenta os cabeçalhos da biblioteca, e as definições são incluídas no *namespace* `OOPS`. Como já descrito, o método `myID()` de `OOPS::Group` retorna o identificador do processador virtual dentro do grupo. Assim, com esse código, somente o processador 0 imprime o texto.

5.2 Aplicando Topologias

Para a comunicação entre os processadores de um grupo é necessário construir uma topologia sobre esse grupo, conforme descrito.

No trecho de código a seguir é declarada uma topologia tipo duto para viabilizar as comunicações entre os processadores.

```
1 #include <iostream>
2 #include <oops>
3 void OOPS::Main::executeOn(const OOPS::Group *allProcs) {
4   OOPS::TopologyPipe topo(allProcs);
5   int x = 0;
6   topo.fromPrevious(x);
7   std::cout << "x = " << x << " and I am ID = "
8             << topo.group()->myID() << std::endl;
9   x++;
10  topo.toNext(x);
11 }
```

Nesse trecho de código, a variável x é passada como um *token* pelos processadores. Cada processador irá imprimir o valor recebido e incrementá-lo antes do envio para o processador seguinte na topologia. Como pode ser visto na linha 4, a topologia é criada sobre um grupo. No caso acima, a topologia tipo duto é construída sobre `allProcs`. Na linha 8 apresentamos um método da topologia que retorna um ponteiro para o grupo de processadores sobre o qual foi aplicada.

5.3 Usando Contêineres Distribuídos

O exemplo seguinte calcula concorrentemente o produto escalar de dois vetores lidos de arquivos, e o resultado é escrito na saída padrão somente pelo

processador 0. No código, a topologia aplicada ao grupo de processadores é `OOPS::TopologyLinear` já apresentada.

São declarados vetores de elementos do tipo *double*, que utilizam a topologia linear assim como o modo de distribuição `OOPS::DistributionBlocked` para realizar a distribuição de seus elementos em blocos.

Na linha 13 são declarados dois arquivos para leitura que são usados pelos métodos equivalentes `load` e $\gg$ (sobrecarga do operador de extração) de `OOPS::Vector<T>` para realizar a leitura dos dados do arquivo especificado e realizar a distribuição adequada destes. A definição de `OOPS::Vector<T>::operator*` leva em consideração a distribuição do vetor para realizar o produto dos argumentos elemento a elemento, e retorna o resultado em um vetor com a distribuição adequada. Na linha 18 é declarado um arquivo para escrita usado pelo operador de inserção ($\ll$) na linha seguinte para escrever os elementos do vetor em arquivo. O método `OOPS::Vector<T>::sum()` realiza a somatória dos elementos do vetor.

```
1 #include <iostream>
2 #include <fstream>
3 #include <oops>
4 const int N = 10000;
5
6 void OOPS::Main::executeOn(const OOPS::Group *allProcs) {
7   try {
8     OOPS::TopologyLinear topo(allProcs);
9     OOPS::DistributionBlocked block;
10
11    OOPS::Vector<double> v1(N, block, topo), v2(N, block, topo),
12      v(N, block, topo);
13    std::ifstream file1("vector1.dat"), file2("vector2.dat");
14    v1.load(file1);
15    file2 >> v2;
16
17    v = v1 * v2;
18    std::ofstream out("result.dat");
19    out << v;
20
21    double s = v.sum();
22    if (topo.group()->isFirst())
23      std::cout << "Scalar product = " << s << std::endl;
24  }
25  catch(OOPS::Error_NotPositive e) {
26    ... código ...
27  }
```

```
28   catch(...) {
29     std::cout << "\nOOPS!!\n" << std::endl;
30   }
31 }
```

Note que é necessário fornecer um tratamento de erros adequado para cada possibilidade de erro a ser gerada no programa. Nos próximos exemplos iremos subtrair essa indicação por questão de simplicidade do código.

A seguir, é apresentado um trecho de código para exemplificar o uso de matrizes distribuídas. Este exemplo é parecido com o anterior, porém neste é realizado o produto de uma matriz por um vetor. Duas topologias são construídas com o grupo `allProcs` nas linhas 7 e 8, uma linear e outra tipo grade. Para a construção da topologia tipo grade foi utilizado o construtor da classe que recebe como segundo argumento o número de linhas que a grade deverá ter. O vetor `a` é um vetor replicado para manter localidade dos dados, já que todos os seus elementos multiplicarão os elementos de uma linha da matriz `m`. Por esse mesmo motivo, a matriz apresenta suas linhas divididas em blocos mas não há divisão nas colunas, com a utilização dos modos de distribuição `DistributionBlocked` e `DistributionNone`. O resultado da multiplicação é armazenado em arquivo, neste exemplo com o uso do método `OOPS::Vector<T>::store()`.

```
1  #include <fstream>
2  #include <oops>
3  const int N = 1000;
4
5  void OOPS::Main::executeOn(const OOPS::Group *allProcs) {
6    try{
7      OOPS::TopologyLinear topo_linear(allProcs);
8      OOPS::TopologyGrid topo_grid(allProcs, allProcs->size());
9      OOPS::DistributionBlocked block;
10     OOPS::DistributionNone none;
11     OOPS::VectorRepl<double> a(N, topo_linear);
12     OOPS::Vector<double> b(N, block, topo_linear);
13     std::ifstream file_vec("vector.dat");
14     file_vec >> a;
15
16     OOPS::Matrix<double> m(N, N, block, none, topo_grid);
17     std::ifstream file_mat("matrix.dat");
18     file_mat >> m;
19
20     b = m * a;
```

```
21     b.store("result.dat");
22   }
23 }
```

O próximo trecho de código exemplifica o uso de distribuição cíclica, modelado pela classe `OOPS::DistributionCyclic`. Este modo de distribuição é particularmente útil para quando não existe a necessidade de comunicação dos dados entre vizinhos e a quantidade de cálculos realizada sobre os elementos de um contêiner é muito diferente para cada elemento, e tende a ter cargas parecidas em regiões de índices próximos. No exemplo implementamos o cálculo do fractal de Mandelbrot, onde para cada ponto z do plano complexo se realiza a iteração $z \leftarrow z^2 + c$; se essa iteração divergir, a quantidade de iterações necessária para determinar essa divergência é usada para determinar a cor na imagem fractal. Já que a quantidade de iterações pode variar bastante de ponto para ponto, a carga computacional é bastante distinta para os elementos da matriz.

```
1 #include <fstream>
2 #include <oops>
3 int compute_mandel(const double x, const double y) {
4   // retorna o valor de mandelbrot para o complexo (x + iy)
5 }
6
7 void OOPS::Main::executeOn(const OOPS::Group *allProcs) {
8   try{
9     OOPS::TopologyGrid grid(allProcs);
10     OOPS::DistributionCyclic cyclic;
11     int n_real, n_imag;
12     double br, ur, bi, ui;
13     readArgs(br, ur, bi, ui, n_real, n_imag); // deve ser implementado o método
14                                               // para leitura dos dados
15     double rdelta = (ur - br) / n_real;
16     double idelta = (ui - bi) / n_imag;
17     OOPS::Matrix<int> mandel(n_imag, n_real, cyclic, cyclic, grid);
18     for (int i = 0; i < mandel.localRowSize(); i++)
19       for (int j = 0; j < mandel.localColSize(); j++) {
20         OOPS::Matrix<int>::Index gi = mandel.localToGlobal(i,j);
21         mandel.local(i,j) = compute_mandel(br+(gi.col()*rdelta)+rdelta/2,
22                                            ui-(gi.row()*idelta)-idelta/2);
23       }
24     std::ofstream outFile("result.dat");
25     outFile << mandel;
26   }
27 }
```

Vemos o uso dos métodos de `OOPS::Matrix<T>` que retornam os tamanhos locais das linhas e colunas, respectivamente `OOPS::Matrix::localRowSize()` e `OOPS::Matrix<T>::localColSize()`. Os elementos distribuídos da matriz são acessados através do método `OOPS::Matrix<T>::local()`, usando índices locais. Verifica-se também a utilização de um índice global, que é um objeto da classe `OOPS::Matrix<T>::Index`, e alguns de seus métodos, como `row()` e `col()`, que informam os valores de linha e coluna do índice global.

5.4 Trabalhando com Componentes Paralelos

Várias aplicações científicas apresentam soluções tipo estêncil, por exemplo as que tratam de equações diferenciais parciais. Nesta secção, exemplificamos uma solução usando o OOPS para resolver o problema de Poisson, de acordo com o capítulo 16 de [3].

No caso bidimensional, temos as seguintes equações para o problema de Poisson:

$$\begin{aligned} \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} &= f(x,y) \qquad \text{no interior} \\ u(x,y) &= g(x,y) \qquad \text{no contorno} \end{aligned} \tag{5.1}$$

Trataremos do caso homogêneo, ou seja, tomaremos $f(x,y) = 0$. Para a solução numérica, usamos uma matriz para representar $u(x,y)$ como uma malha discretizada, representada por $u_{i,j}$, e utilizamos o método de Jacobi, que é um algoritmo iterativo simples para resolver este problema. Seguindo as aproximações de [3] para a equação 5.1, chegamos na seguinte equação para o Jacobi:

$$u_{i,j}^{t+1} = \frac{1}{4}\left(u_{i-1,j}^{t} + u_{i,j-1}^{t} + u_{i+1,j}^{t} + u_{i,j+1}^{t}\right) \tag{5.2}$$

onde t indica o número da iteração, e i e j indicam a posição do elemento na matriz.

Com a equação (5.2) o novo valor para um elemento é calculado em função do valor de seus elementos vizinhos no passo anterior e o cálculo é repetido iterativamente até uma condição de término ser satisfeita.

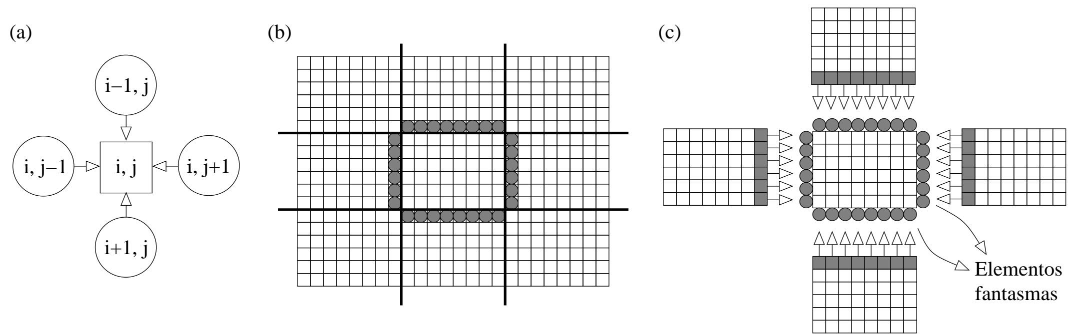

Figura 5.1: (a) Diagrama de um estêncil que utiliza os valores dos seus vizinhos para calcular seu novo valor. (b) Esquema da dependência dos dados para os cálculos em um processador. (c) Elementos de um processador adicionado de uma borda para armazenamento dos dados de seus vizinhos, e que é denominada fantasma ou sombra.

Um esquema da relação acima pode ser visto na figura 5.1(a). Como o conjunto de vizinhos requisitados é fixo e com acesso regular, é fácil realizar uma pré-busca de valores presentes em outros processadores, figura 5.1(b). Esses valores necessários para os cálculos podem ser armazenados na mesma estrutura, facilitando o algoritmo. Para isso, adiciona-se bordas nas laterais para o armazenamento desses dados que são chamadas de sombras ou fantasmas, cujo tamanho é determinado pelo estêncil utilizado como esquematizado na figura 5.1(c).

O quadro a seguir mostra as ferramentas que o OOPS fornece para auxiliar nesse problema. Neste exemplo temos as seguintes condições de contorno (valores para $g(x, y)$ na equação 5.1): o terço do meio de elementos na primeira linha terá seus vizinhos superiores em um valor fixo T_s, enquanto que os outros elementos na fronteira terão vizinhos com valor fixo T_i, assim como os valores iniciais dos elementos $u_{i,j}$. Duas matrizes distribuídas por blocos em uma topologia tipo grade (`a` e `b`) são usadas para representar u em duas iterações sucessivas. O último parâmetro do construtor das matrizes indica o tamanho da faixa adicional para armazenar os dados fantasmas (linhas 12 e 13), no caso o tamanho é 1. As atualizações dos fantasmas são realizadas nas chamadas de `OOPS::Matrix<T>::syncGhostsCart()` conforme já descrita. Veja que os valores de contorno para a fronteira de u são armazenados na borda especificada para os fantasmas. Depois de inicializadas as matrizes, começa-se o ciclo de cálculo até que a maior diferença (em valor absoluto) entre o valor novo e o antigo seja

menor que o limite `diff_limit`. No exemplo abaixo, podemos também ver o uso do método `OOPS::TopologyGrid::max()` na linha 45 que é uma operação de redução procurando um máximo global entre os valores máximos calculados localmente em cada processador da grade.

```
1 #include <fstream>
2 #include <oops>
3 void OOPS::Main::executeOn(const OOPS::Group *allProcs) {
4   try{
5     OOPS::TopologyGrid grid(allProcs);
6     OOPS::DistributionBlocked block;
7     int M, N;
8     double Ts, Ti, diff_limit;
9     readArgs(M, N, Ts, Ti, diff_limit); // deve ser implementado o método
10                                        // para leitura dos dados
11    int ghost = 1;
12    OOPS::Matrix<double> a(M, N, block, block, grid, ghost),
13      b(M, N, block, block, grid, ghost);
14
15    // Inicialização das matrizes
16    for (int i = -ghost; i < (a.localRowSize() + ghost); i++)
17      for (int j = -ghost; j < (a.localColSize() + ghost); j++) {
18        OOPS::Matrix<double>::Index gi = a.localToGlobal(i,j);
19         if ((gi.row() == -1) && (gi.col() > N/3) && (gi.col() < 2*N/3)) {
20          a.local(i,j) = Ts; b.local(i,j) = Ts;
21         }
22         else {
23           a.local(i,j) = Ti; b.local(i,j) = Ti;
24         }
25      }
26
27    double max_diff;
28    // Início do ciclo de cálculos iterativos
29    do {
30      b.syncGhostsCart();
31      for (int i = 0; i < b.localRowSize(); i++)
32        for (int j = 0; j < b.localColSize(); j++)
33          a.local(i,j) = (b.local(i,j-1) + b.local(i,j+1) +
34                          b.local(i-1,j) + b.local(i+1,j)) / 4;
35      a.syncGhostsCart();
36      max_diff = 0;
37      for (int i = 0; i < b.localRowSize(); i++)
38        for (int j = 0; j < b.localColSize(); j++) {
39          b.local(i,j) = (a.local(i,j-1) + a.local(i,j+1) +
40                          a.local(i-1,j) + a.local(i+1,j)) / 4;
41          double diff = fabs(b.local(i,j) - a.local(i,j));
42          if (diff > max_diff)
43            max_diff = diff;
```

```
44        }
45      grid.max(max_diff);
46    } while (max_diff > diff_limit);
47
48    std::ofstream result("result.dat");
49    result << b;
50  }
51 }
```

5.5 Composições de Componentes Paralelos

Como discutido anteriormente na apresentação da proposta do OOPS, secção 4.1, os componentes paralelos podem executar seqüencial ou concorrentemente.

A composição seqüencial não adiciona nenhum elemento que não tenha sido apresentado anteriormente nos exemplos, visto que não há necessidade de comunicação entre os componentes paralelos. No trecho de código a seguir apresentamos dois componentes paralelos `foo()` e `bar()` que utilizam uma mesma matriz distribuída para seus cálculos.

```
1 #include <fstream>
2 #include <oops>
3 void OOPS::Main::executeOn(const OOPS::Group *allProcs) {
4   try{
5     OOPS::TopologyGrid topo_grid(allProcs);
6     OOPS::DistributionBlocked block;
7     int N;
8     OOPS::Matrix<double> m(N, N, block, block, topo_grid);
9     foo(m);
10    bar(m);
11    std::ofstream result("result.dat");
12    result << m;
13  }
14 }
```

Para a composição concorrente é necessário dividir os processadores virtuais em grupos para atuarem nos diferentes componentes paralelos, e utilizar inter-topologias para a comunicação entre esses grupos viabilizando a troca de dados entre os componentes paralelos. A composição concorrente de componentes paralelos adiciona a possibilidade de paralelismo de tarefas ao OOPS.

O trecho de código a seguir exemplifica esta situação. Novamente utilizamos dois componentes paralelos `foo` e `bar`, que apresentam um método `executeOn()` com o código a ser executado. Usando objetos da classe `OOPS::Partner`, é montada uma intertopologia tipo duto para a comunicação entre os grupos que atuam nos dois componentes. A troca de dados entre eles ocorre nas chamadas dos métodos `toNext()` e `fromPrevious()` nas linhas 32 e 37. Nas linhas 51 e 52 estão as chamadas para a função parametrizada no tipo do componente `OOPS::Execute<T>()`, que recebe como parâmetros o grupo para atuar no componente e a intertopologia usada para a comunicação entre eles.

```
1 #include <oops>
2 class InterTopologyPipe : public OOPS::InterTopology {
3   OOPS::Partner *next, *prev;
4 public:
5   InterTopologyPipe(const OOPS::Group *g) : OOPS::InterTopology(g) {
6     int n = g->myID() + 1;
7     int p = g->myID() - 1;
8     if (g->isFirst())
9       p = OOPS::PARTNER_NULL;
10    if (g->isLast())
11      n = OOPS::PARTNER_NULL;
12    next = new OOPS::Partner(g, n);
13    prev = new OOPS::Partner(g, p);
14  }
15  void toNext(int &x) const { next->send(x); }
16  void fromPrevious(int &x) const { prev->recv(x); }
17 };
18
19 class foo {
20 public:
21  static void executeOn(const OOPS::Group *g, const InterTopologyPipe *it);
22 };
23
24 class bar {
25 public:
26  static void executeOn(const OOPS::Group *g, const InterTopologyPipe *it);
27 };
28
29 void foo::executeOn(const OOPS::Group *g, const InterTopologyPipe *it) {
30  int x;
31  ... continuação do código ...
32  it->toNext(x);
33 }
34
35 void bar::executeOn(const OOPS::Group *g, const InterTopologyPipe *it) {
```

```
36   int x;
37   it->fromPrevious(x);
38   ... continuação do código ...
39 }
40
41 void OOPS::Main::executeOn(const OOPS::Group *allProcs) {
42   try{
43     int n = 2; int *IDs; IDs = new int[n];
44     IDs[0] = 0; IDs[1] = 2;
45     OOPS::Group *sub1 = allProcs->subGroup(n, IDs);
46     IDs[0] = 1; IDs[1] = 3;
47     OOPS::Group *sub2 = allProcs->subGroup(n, IDs);
48     delete IDs;
49
50     InterTopologyPipe *it = new InterTopologyPipe(allProcs);
51     OOPS::Execute<foo>(sub1, it);
52     OOPS::Execute<bar>(sub2, it);
53   }
54 }
```

Vale ressaltar que a possibilidade de composição concorrente de componentes paralelos, mostrada no exemplo anterior, viabiliza o uso de paralelismo de tarefas com o OOPS.

6

Desempenho

Para verificar a sobrecarga que a utilização do OOPS adiciona a um programa paralelo, comparamos o desempenho das execuções de duas versões de um programa de teste, uma escrita diretamente em MPI e outra usando as classes fornecidas pelo OOPS. Escolhemos o cálculo do fractal de Mandelbrot, discutido no capítulo anterior na secção 5.3, como programa de teste. Apesar de esta não ser uma aplicação adequada para paralelização por não ter necessidade de alto desempenho e também apresentar granulosidade muito fina, o cálculo do fractal de Mandelbrot foi escolhido por realizar cálculos muito simples, permitindo que o programador se concentre nos diversos aspectos paralelos.

O sistema computacional usado para as execuções foi um *cluster* Beowulf [68] com 8 nós de processamento com a seguinte configuração: processadores Pentium 4 de 3,2 GHz com 1,5 GBytes de memória RAM e interconexão FastEthernet. Foram utilizados *softwares* de distribuição livre, basicamente o sistema operacional Linux [69], com as ferramentas de programação que são distribuídas com o mesmo [70] (compilador gcc, make, entre outros) e a biblioteca MPI desenvolvida no Argonne National Laboratory (MPICH) [71].

Todas as execuções foram realizadas usando matrizes de 5000 por 5000 elementos, e foi feita a média de dez execuções para determinar cada ponto dos gráficos a seguir. O gráfico da figura 6.1 apresenta uma curva que é a razão entre o tempo de execução da versão em OOPS sobre o tempo de execução da versão

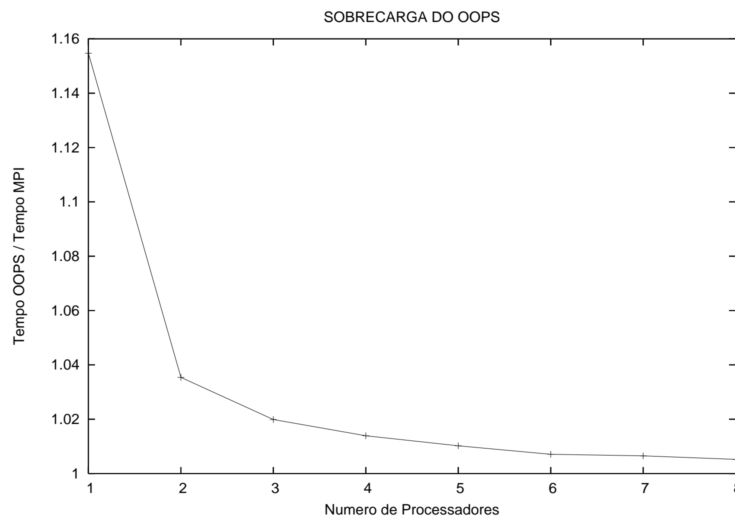

Figura 6.1: Sobrecarga adicionada ao programa paralelo para o cálculo do fractal de Mandelbrot utilizando o OOPS em relação a um programa escrito diretamente em MPI.

em MPI.

Verifica-se pelo gráfico que a utilização do OOPS adiciona uma sobrecarga em torno de 15% quando é utilizado somente um processador, e que esta sobrecarga diminui até menos de 1% para a execução com 8 processadores. Esta sobrecarga é verificada por causa dos diversos cálculos a mais que o OOPS realiza para o programa. Por exemplo, o OOPS trata do caso geral de uma distribuição cíclica para um tamanho qualquer de bloco de elementos para distribuição, enquanto que a versão em MPI é otimizada para bloco de tamanho 1. A queda na sobrecarga ao aumentarmos o número de processadores utilizados é devida à diminuição da importância desses cálculos frente ao tempo de comunicação dos dados.

Medimos também o *speedup* alcançado pelas versões paralelas do programa de cálculo do fractal de Mandelbrot. O *speedup* em P processadores é a razão do tempo tomado pela execução do programa seqüencial pelo tempo de execução do programa paralelo com P processadores, assim:

$$Speedup\,(P) = \frac{\text{Tempo seqüencial}}{\text{Tempo paralelo com } P \text{ processadores}}$$

A figura 6.2 apresenta o *speedup* das duas versões paralelas do programa, somente considerando o tempo tomado pelos cálculos, sem considerar o tempo de escrita da matriz em arquivo.

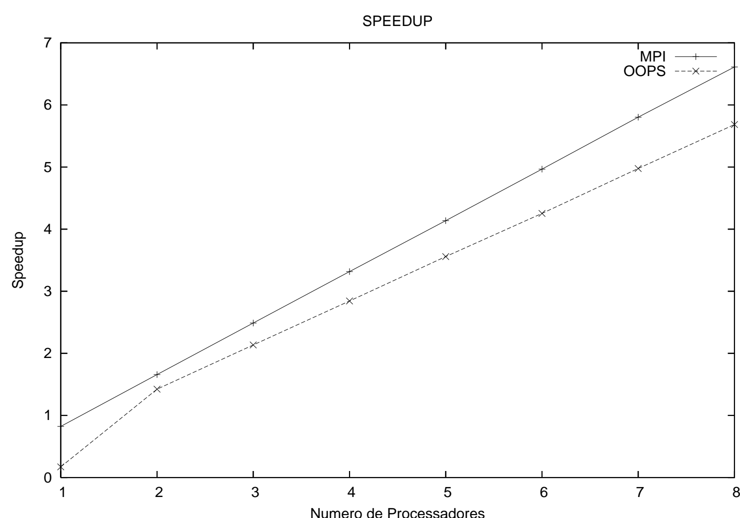

Figura 6.2: *Speedup* alcançado pelas versões paralelas em MPI e em OOPS para o cálculo do fractal de Mandelbrot.

Vemos que a inclinação da curva do *speedup* alcançado pelo programa em MPI é menor do que 1 e isso é devido à quantidade de cálculos realizados para a determinação dos índices locais e globais da matriz que não precisam ser realizados no programa seqüencial. A cada iteração o OOPS realiza cálculos mais complexos para a determinação dos índices do que o MPI (novamente por tratar do caso geral), e por isso vemos no gráfico que a curva do *speedup* alcançado com o programa em OOPS tem inclinação menor ainda.

Não se espera que a diferença no *speedup* alcançado pelo OOPS seja tão pronunciada em relação ao MPI quando a aplicação envolver uma quantidade maior de cálculos próprios em relação aos cálculos relacionados aos aspectos paralelos, como por exemplo o cálculo dos índices e dimensões locais e globais.

As classes fornecidas pelo OOPS facilitam a implementação de programas concorrentes sem contudo adicionar muita sobrecarga ao desempenho da aplicação, como visto. Neste sentido, o OOPS representa realmente uma camada fina entre o programa do usuário e o MPI.

7 Conclusões

A grande disponibilidade de sistemas paralelos e a alta demanda por desempenho computacional de diversas aplicações estimulam o desenvolvimento de aplicações paralelas. O uso de paralelismo é bastante atraente por possibilitar a extração de maior desempenho do sistema computacional.

Em contraponto, o desenvolvimento de programas paralelos eficientes que consigam alcançar bons desempenhos é bastante complexo. Com a programação paralela vários novos aspectos são introduzidos à programação seqüencial, tais como sincronização de comunicações entre processos, distribuição de dados, acesso a dados distribuídos, balanceamento de cargas e necessidade de localidade de dados, entre outros. Tudo isso dificulta sobremaneira a programação paralela e freqüentemente desestimula seu uso.

Diversas abordagens de ferramentas foram propostas para introduzir abstrações de alto nível e auxiliar na programação de aplicações paralelas, porém ainda não há um consenso das técnicas que devem ser utilizadas para tal. Com a análise do panorama de ferramentas existentes, neste trabalho de doutorado foi realizado o projeto e desenvolvimento um *framework* de classes, com características que foram consideradas apropriadas para facilitar a programação concorrente e ainda permitir que bons desempenhos sejam alcançados pelas aplicações que o utilizem.

O OOPS, *Object-Oriented Parallel System*, utiliza orientação a objetos para fornecer abstrações de programação e encapsular detalhes de implementação,

mantendo os conceitos de objetos e processos ortogonais. As comunicações entre os processadores virtuais se dão através de trocas de mensagens, evidenciando as operações não-locais. O enfoque é dado a aplicações científicas regulares que tenham demanda por alto desempenho e ênfase foi dada ao paralelismo de dados, apesar de apresentar construções para paralelismo de tarefas. Por se tratar de uma biblioteca de classes, pode facilmente ser estendida, e seu uso não acarreta nas dificuldades de reutilização de código e necessidade de aprendizado como no desenvolvimento de novas linguagens. Mantém-se o paralelismo explícito e o usuário do *framework* deve se envolver com aspectos paralelos, como particionamento dos dados, sem contudo se envolver com detalhes laboriosos de implementação. Dessa forma o conhecimento e discernimento do programador da aplicação são usados para conseguir um código com bom desempenho.

Como mostrado com um programa de teste no capítulo 6, o OOPS não acrescenta uma grande sobrecarga às aplicações desenvolvidas com ele em relação a implementações diretamente em MPI, e esta questão de desempenho é fundamental neste âmbito de programação paralela.

7.1 Sugestões de Trabalhos Futuros

Diversas classes estão implementadas no OOPS pelo momento, tendo sido apresentadas no capítulo 4. Novas funcionalidades podem ser acrescentadas provendo facilidades importantes a um *framework* como este. Além disso, melhorias podem ser feitas às implementações para alcançar maiores desempenho, confiabilidade e flexibilidade.

Mudanças na implementação não alteram em nada a interface do OOPS. Elas podem conferir maior desempenho ou tão somente servir para experimentação e pesquisa. Além disso, algumas extensões de funcionalidades poderiam tornar o *framework* mais atrativo ao uso.

A seguir colocamos algumas sugestões de adequações importantes ao OOPS. Outras extensões ainda não projetadas podem também vir a ser implementadas, caso haja interesse e demanda para tal.

Extensão das classes desenvolvidas Novos contêineres distribuídos, topolo-

gias do grupo de processadores virtuais, e modos de distribuição dos contêineres devem ser implementados para aumentar as funcionalidades do OOPS. Exemplos são contêineres distribuídos para arrays multidimensionais e árvores binárias, topologia modelando um hipercubo e um arranjo de árvore binária para a comunicação dos processadores virtuais, assim como modos de distribuição apropriados para essas topologias.

Verificação de facilidade de uso Deveria ser realizada uma análise de facilidade de uso do OOPS e da adequação de sua documentação. Para isso alunos de um curso de programação concorrente poderiam desenvolver aplicações utilizando as classes do OOPS e o resultado comparado com aplicações desenvolvidas usando MPI por um grupo de controle, levando em consideração o tempo de devolvimento de ambas.

Avaliação de desempenho Uma avaliação mais extensa de desempenho do OOPS e da sobrecarga adicionada ao MPI poderia ser realizada com a implementação de outras aplicações de teste. Estas aplicações deveriam ter características variadas no tocante à quantidade de comunicações e de computações realizadas.

Entrada e saída paralela A entrada e saída não estão sendo realizadas de forma adequada até o momento. É necessário que o OOPS forneça classes cujos métodos permitam que todos os processos enviem mensagens para a saída padrão e viabilizem a entrada e saída de arquivos paralelos.

Balanceamento de cargas e criação dinâmica de tarefas Poderiam ser desenvolvidas classes para modelar tarefas e balanceadores de carga, como na proposta de Travieso [63]. Em diversas situações as computações não são distribuídas equivalentemente entre os processos paralelos, fazendo com que alguns deles recebam uma carga de trabalho sensivelmente maior que outros. Torna-se importante a possibilidade de utilizar um sistema de balanceamento de cargas para minimizar este problema.

Estruturas de dados irregulares A integração de estruturas de dados irregulares tornaria o *framework* mais abrangente. Até o momento, o OOPS se

direciona a aplicações científicas regulares, que utilizam vetores e matrizes para armazenar seus dados. Com a modelagem de estruturas irregulares outras aplicações poderiam ser desenvolvidas com o OOPS, como aplicações de elementos finitos usando refinamento adaptativo de malhas.

Outras alterações e extensões que não foram sugeridas e ainda não vislumbradas podem ser realizadas sem prejuízo às implementações previamente existentes.

7.2 Considerações Finais

Argumentamos que a ferramenta que propomos com esse trabalho representa uma abordagem adequada ao desafio de desenvolver ferramentas para apoiar a programação concorrente para execução paralela. Soluções com abstrações de mais baixo nível permitem alcançar melhores desempenhos da aplicação, porém acarretam em uma maior dificuldade de programação. Já abordagens com nível mais alto de abstração que também atinjam bom desempenho se direcionam a uma gama restrita de aplicações.

No estágio atual de desenvolvimento, a utilização do OOPS se limita a aplicações que façam uso de contêineres regulares para armazenar seus dados. As classes já desenvolvidas tratam principalmente de paralelismo de dados, apesar de que paralelismo de tarefas é realizado com a composição concorrente de componentes paralelos.

Visto que o conjunto de classes do OOPS fornecido até o momento é coerente e apresenta todas as funcionalidades básicas necessárias para ser utilizado em algumas aplicações reais, é necessário que estas sejam implementadas para validar a interface, avaliar a facilidade de uso e direcionar o desenvolvimento do OOPS. Com a experiência conseguida o sistema pode ser ampliado de acordo com as necessidades de novas aplicações.

Referências Bibliográficas

[1] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[2] David E. Culler, Jaswinder Pal Sing, and Anoop Gupta. *Parallel Computer Architecture. A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[3] Jack Dongarra, Ian Foster, Geoffrey Fox, Willian Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of Parallel Computing*. Morgan Kaufmann, 2003.

[4] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamim/Cummings, 1994.

[5] Cherri M. Pancake and Donna Bergmark. Do Parallel Languages Respond to the Needs of Scientific Programmers? *IEEE Computer*, 23(12):13–23, 1990.

[6] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[7] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer*, 26(6):57–67, 1993.

[8] Luiz Fernando Capretz. A Brief History of the Object-Oriented Approach. *Software Engineering Notes*, 28(2), 2003.

[9] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[10] David. R. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of the Symbolic and Algebraic Computation, International Symposium (ISSAC'88)*, pages 13–25, 1988.

[11] James C. Dehnert and Alexander Stepanov. Fundamentals of Generic Programming. In *Generic Programming'98, International Seminar on Generic Programming, Selected Papers*, pages 1–11, 2000.

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

[13] Ian Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1995.

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison Weley, 2000.

[15] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Open Grid Service Infrastructure WG*. Global Grid Forum, 2002.

[16] Geoffrey Fox and Dennis Gannon. Computational Grids. *Computing in Science and Engineering*, 3(4):74–77, 2001.

[17] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. Pearson Education, 6th edition, 2002.

[18] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Artificial Intelligence. MIT Press, 1986.

[19] Actor-based Languages. `http://www.dekorte.com/docs/actors/`. visitado em 01/2006.

[20] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Aplications (OOPSLA'87)*, pages 168–182, 1987.

[21] Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation*. Computer Science. Addison-Wesley, 1983.

[22] Chamond Liu. *Smalltalk, Objects, and Design*. toExcel, 2000.

[23] Jean-Pierre Briot and Rachid Guerraoui. On the Use of Smalltalk for Concurrent and Distributed Programming. *Informatik Journal*, 1996.

[24] Jean Bèzivin. Some Experiments in Object-Oriented Simulation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Aplications (OOPSLA'87)*, pages 394–405, 1987.

[25] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[26] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matt Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 2.1.6, Argonne National Laboratory, 2003.

[27] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. C. Eigler, and G. R. Gao. ABC++: Concurrency by Inheritence in C++. *IBM Systems Journal*, 34(1):120–137, 1995.

[28] Mauricio De Simone. Active Expressions: A Language-Based Model for Expressing Concurrent Patterns. Tese de Mestrado, University of Waterloo, 1997.

[29] K. Russel and Duraid Madina. ClassdescMP: Easy MPI programming in C++. In Slot et al, editor, *Computacional Science*, LNCS 2660. Springer-Verlag, 2003.

[30] Russell Standish and Duraid Madina. Classdesc and Graphcode: support for scientific programming in C++. *submitted to SIAM Journal of Scientific Computing*, 2006.

[31] F. Belloncle D. Caromel and Y. Roudier. *Parallel Programming Using C++*. The MIT Press, 1996.

[32] Akinori Yonezawa and Takuo Watanabe. An Introduction to Object-Based Reflective Concurrent Computation. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 50–54, 1989.

[33] David Kotz. A Data-Parallel Programming Library for Education (DAPPLE). *Computer Science Education*, 6(2):141–159, 1996.

[34] Alexander A. Stepanov and Meng Lee. The Standard Template Library. Technical Report HPL-95-11, HP Laboratories, 1995.

[35] Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy. DatTeL: A Data-Parallel C++ Template Library. *Parallel Processing Letters*, 13(3):461–472, 2003.

[36] Jens Gerlach, Peter Gottschling, and Uwe Der. A Generic C++ Framework for Parallel Mesh Based Scientific Applications. In *Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'01)*, pages 45–54, 2001.

[37] S.B. Baden, P. Colella, D. Shalit, and B. Van Straalen. Abstract KeLP. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[38] Roxana Diaconescu and Reidar Conradi. A Data Parallel Programming Model Based on Distributed Objects. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02)*, pages 455–460, 2002.

[39] Jeffrey M. Squyres, Brian C. McCandless, and Andrew Lumsdaine. Object Oriented MPI: A Class Library for the Message Passing Interface. In

Proceedings of the Parallel Object-Oriented Methods and Applications (POOMA'96), 1996.

[40] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995.

[41] Peter W. Rijks, Jeffrey M. Squyres, and Andrew Lumsdaine. Performance Benchmarking of Object Oriented MPI (OOMPI) version 1.0.2. Technical report, University of Notre Dame - Department of Computer Science and Engineering, 1999. TR 99-14.

[42] Kyle K. Chand. Interoperability and Interchangeability in the Overture Framework. In *Proceedings of the SIAM Conference on Computational Science and Engineering*, 2005.

[43] D. L. Brown, William D. Henshaw, and Daniel J. Quinlan. Overture: An Object-Oriented Framework for Solving Partial Differential Equations on Overlapping Grids. In *Proccedings od SIAM Conference on Object Oriented Methods for Scientific Computing*, 1999.

[44] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 1997. Version 2.0.

[45] Julian C. Cummings, James A. Crotinger, Scott W. Haney, William F. Humphreyand, Steve R. Karmesin, John V. W. Reynders, Stephen A. Smith, and Timothy J. Williams. Rapid Application Development and Enhanced Code Interoperability Using the POOMA Framework. In *Proceedings of the SIAM Workshop on Object-Oriented Methods and Code Interoperability in Scientific and Engineering Computing (OO'98)*, 1998.

[46] Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William Humphrey, John Reynders, Stephen Smith, and Timothy J. Williams. Array Design and Expression Evaluation in POOMA II. In *Proceedings of the ISCOPE'98*, 1998.

[47] Andrew Wissink, David Hysom, and Richard D. Hornung. Enhancing Scalability of Parallel Structured AMR Calculations. In *Proceedings of the 17th*

ACM International Conference on Supercomputing (ICS03), pages 336–347, 2003.

[48] Andrew M. Wissink, Richard D. Hornung, Scott R. Kohn, Steve S. Smith, and Noah Elliott. Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing SC'01*, 2001.

[49] L. Kale and S. Krishnan. *Parallel Programming Using C++*, page 175. The MIT Press, 1996.

[50] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable Molecular Dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[51] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[52] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, 27(8):14–26, 1994.

[53] Rohit Chandra. *The COOL Parallel Programming Language: Design, Implementation and Performance*. Tese de Doutorado, Computer Science Department, Stanford University, 1995.

[54] H. Carr, R. Kessler, and M. Swanson. Distributed C++. *ACM SIGPLAN Notices*, 28(1), 1993.

[55] H. Carr, R. Kessler, and M. Swanson. Compiling Distributed C++. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 496–503, 1993.

[56] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming ABCL/1. In *Proceedings of the 1st ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 258–268, 1986.

[57] S. Gilbert. A MasPar implementation of Data Parallel C++. Technical report, London Parallel Applications Center (LPAC), 1992.

[58] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++ A C++ Dialect for High Performance Parallel Computing. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS)*, 1996.

[59] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[60] Andrew Grimshaw, Adam Ferrari, and Emily West. *Parallel Programming Using C++*, pages 383–427. The MIT Press, 1996.

[61] Jeong Lim and Ralph E. Johnson. The Heart of Object-Oriented Concurrent Programming. In *Proceedings of the ACM SIGPLAN Workshop on Object-based Concurrent Programming*, pages 165–167, 1988.

[62] J. G. Seik and A. Lumsdaine. The Matrix Template Library: A generic programming approach to high-performance numerical linear algebra. In D. Caromel, R. R. Oldehoeft, and M. D. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*. Springer-Verlag, 1998.

[63] Gonzalo Travieso. *Técnicas de Orientação a Objetos para Programação Paralela*. Tese de Livre Docência, Instituto de Física de São Carlos - Universidade de São Paulo, 2004.

[64] Bjarne Stroustrup. C++ in 2005. Extended foreword to Japanese translation of "The Design and Evolution of C++", 2005.

[65] Gabriela Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. In *Proceedings of the Symposium on Principles of Programming Languages (POPL ´06)*, 2006.

[66] Doxygen. Dimitri van Heesch, `http://www.doxygen.org/`. visitado em 01/2006.

[67] Meilir Page-Jones. *Fundamentos do Desenho Orientado a Objeto com UML*. Makron Books, 1st edition, 2001.

[68] William Gropp, Ewing Lusk, and Thomas Sterling, editors. *Beowulf Cluster Computing with Linux*. Cambridge, MA: MIT Press, 2nd edition, 2003.

[69] The Linux Kernel Archives. `http://www.kernel.org/`. visitado 01/2006.

[70] Free Software Foundation. The GNU Operating System, `http://www.gnu.org/`. visitado em 01/2006.

[71] 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.