Universidade Federal do Ceará
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

Uma Linguagem de Programação Paralela Orientada a Objetos para Arquiteturas de Memória Distribuída

Eduardo Gurgel Pinho

Fortaleza – Ceará
Junho de 2012

Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

Uma Linguagem de Programação Paralela Orientada a Objetos para Arquiteturas de Memória Distribuída

Autor
Eduardo Gurgel Pinho

Orientador
Prof. Dr. Francisco Heron de Carvalho Junior

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Fortaleza – Ceará
2012

Resumo

Em programação orientadas a objetos (POO) , a habilidade de encapsular interesses de software da dominante decomposição em objetos é a chave para alcançar alto nível de modularidade e diminuição de complexidade em projetos de larga escala. Entretanto, o paralelismo de memória distribuída tende a quebrar modularidade, encapsulamento e a independência de objetos, uma vez que as computações paralelas não podem ser encapsuladas em objetos individuais, os quais residem em um espaço de endereçamento único. Para reconciliar orientação a objetos e paralelismo em memória distribuída, esse trabalho introduz a PPOO (Programação Paralela Orientada a Objetos), um estilo de POO onde objetos são distribuídos por padrão. Como uma estensão do C++, uma linguagem consolidada em CAD, a linguagem PObC++ foi projetada e prototipada, incorporando as ideias da PPOO.

Abstract

In object-oriented programming (OOP) languages, the ability to encapsulate software concerns of the dominant decomposition in objects is the key to reaching high modularity and loss of complexity in large scale designs. However, distributed-memory parallelism tends to break modularity, encapsulation, and functional independence of objects, since parallel computations cannot be encapsulated in individual objects, which reside in a single address space. For reconciling object-orientation and distributed-memory parallelism, this work introduces OOPP (Object-Oriented Parallel Programming), a style of OOP where objects are distributed by default. As an extension of C++, a widespread language in HPC, the POb C++ language has been designed and protoyped, incorporating the ideas of OOPP.

Agradecimentos

Gostaria de agradecer a todas as pessoas que me incentivaram e apoiaram, possibilitando meu sucesso na obtenção do título de mestre. Especialmente ao meu orientador, professor Dr. Francisco Heron de Carvalho Júnior, que desde a graduação me orientou no desenvolvimento deste trabalho com muita competência e empenho.

Ao professor João Marcelo, pesquisador do CENAPAD-UFC(Centro Nacional de Processamento de Alto Desempenho), pelo apoio técnico para a utilização da estrutura do centro.

À UFC e à FUNCAP pela oportunidade e financiamento deste trabalho.

Aos meus pais Francisco das Chagas e Maria Helenilde pela dedicação e apoio.

Aos meus professores e colegas da UFC.

Aos que indiretamente contibuiram para esse trabalho: Inessa Sales, Iuri Fernandes, Lucas Abreu, Marco Diego Mesquita e Thiago Marcilon.

Sumário

Capítulo 1

Introdução

O melhor custo benefício de plataformas de computação paralela para computação de alto desempenho (CAD)[1], devido ao sucesso de plataformas de computação paralela de memória distribuída, tais como clusters [3] e grades computacionais [22], motivou o surgimento de novas classes de aplicações nos domínios das ciências computacionais e engenharias, os quais apresentam requisitos mais fortes de modularidade, abstração, segurança, produtividade e alto desempenho para as ferramentas de desenvolvimento de software [40]. Infelizmente, a programação paralela ainda é difícil de incorporar em plataformas usuais de desenvolvimento [7].

A tão sonhada paralelização automática é útil apenas em contextos restritos, assim como bibliotecas científicas pré-paralelizadas de propósito especial [19]. A programação baseada em esqueletos (*Skeletal programming*) [31], uma promissora alternativa que busca conciliar eficiência e alto nível de abstração, não tem atingido a aceitação esperada [14]. Atualmente, bibliotecas de suporte a passagem-de-mensagem que implementam o padrão MPI (*Message Passing Interface*) [20] são amplamente adotadas por programadores paralelos [2], oferecendo expressividade, portabilidade e eficiência sobre um domínio de plataformas de computação paralela. Entretanto, apresentam um nível de abstração e modularidade baixo para lidar com as aplicações emergentes de larga escala nos domínios de CAD.

[1]CAD é um domínio de aplicações com fortes requisitos de desempenho de computação, para alcançar um resultado em menor espaço de tempo, e/ou memória, que superam a capacidade de computadores individuais.

[2]Programador paralelo é o desenvolvedor de programas que executam em plataforma de computação paralela, tais como clusters, MPPs, multiprocessadores e processadores de múltiplos núcleos, com o objetivo de reduzir o tempo de computação de um problema pela orquestração de múltiplas unidades de processamento para solucionar um certo problema.

No contexto de aplicações corporativas, a programação orientada a objetos (POO) tem se consolidado como principal paradigma para alcançar a produtividade e a qualidade no desenvolvimento de software. A orientação a objetos é o resultado de duas décadas de pesquisa em técnicas e ferramentas de programação motivadas pela necessidade de lidar com níveis crescentes de complexidade de software, que teve início com a crise do software dos anos de 1960 [18].

Muitas linguagens de programação orientadas a objetos (OO) ou que utilizam conceitos de orientação a objetos têm sido desenvolvidas, tais como C++, Java, C#, Smalltalk, Ruby e Objetive-C. Apesar do sucesso na indústria do software, linguagens orientadas a objetos não são populares em CAD, dominada por linguagens procedurais tradicionais, como Fortran e C, como consequência do alto nível de abstração e modularidade oferecida por linguagens POO, cujo efeito sobre o desempenho bruto de execução é sensível em aplicações de computação intensiva. Quando o paralelismo entra em cena, a situação é pior, dado a carência de maneiras elegantes de incorporar explicitamente o paralelismo por passagem-de-mensagens a linguagens POO sem quebrar princípios importantes dessas linguagens, como a independência funcional e o encapsulamento dos objetos.

Essa dissertação tem por objetivo apresentar uma linguagem de programação que utilize conceitos de orientação a objetos unindo a práticas existentes na área de programação paralela por passagem de mensagens. PObC++ (uma sigla para Parallel Object C++) é uma linguagem que estende o C++, bem como introduz um novo estilo de programação paralela, suportado por essa linguagem, o qual denominamos OOPP (Programação Paralela Orientada a Objetos), onde objetos são intrinsecamente paralelos, ou seja, distribuídos em um conjunto de unidades de processamento de um computador paralelo de memória distribuída. Uma importante contribuição é a hierarquização da comunicação entre os objetos em dois níveis:

- *Intra-objeto*, para as interações comuns entre processos por passagem-de-mensagens, encapsuladas em objetos paralelos;
- *Inter-objeto*, para a coordenação usual de objetos, por meio de invocação de métodos.

A decisão de estender o C++ vem da sua grande aceitação em CAD. Entretanto, PPOO pode servir de modelo para extensões para outras linguagens OO, tais como Java e C#. A premissa principal que guia o projeto PObC++ é a preservação

dos princípios da orientação a objetos enquanto introduz-se o estilo de programação baseada em passagem-de-mensagens.

O Capítulo 2 discute o contexto atual sobre programação paralela por passagem-de-mensagem e programação orientada a objetos, como também sua integração. O Capítulo 3 apresenta as premissas e conceitos principais por trás da OOPP, mostrando como é suportado pela linguagem proposta, chamada PObC++. Esse capítulo termina apresentando a arquitetura geral do atual protótipo do compilador. O Capítulo 4 apresenta três casos de estudo de programação com PObC++, com o intuito de dar evidências sobre a expressividade, produtividade de desenvolvimento, e performance do PPOO. No Capítulo 5, apresentamos nossas considerações finais, descrevendo possíveis pesquisas futuras e idéias para outras iniciativas.

Capítulo 2

Paralelismo em Linguagens Orientadas a Objetos

Orientação a objetos é um mecanismo de abstração de dados bastante disseminado dentre as técnicas de construção de software. Seus principais conceitos foram introduzidos em meados dos anos 60 com a linguagem de programação Simula'67 [15, 16]. Porém, a primeira linguagem orientada a objetos que se tornou proeminente foi o Smalltalk [17], desenvolvido pela Xerox PARC nos anos 1970. Os desenvolvedores do Smalltalk adotaram o uso pervasivo de objetos como base de computação para a linguagem, sendo o primeiro a utilizar o termo programação orientada a objetos (OOP[1]). Durante os anos 1990, OOP se tornou a principal técnica de estruturação de programas, influenciado principalmente pelo crescimento de popularidade das interfaces gráficas de usuário (GUI), onde as técnicas de orientação a objetos foram extensivamente aplicadas. Entretanto, o interesse no modelo rapidamente ultrapassou o uso em GUI, a partir do momento que engenheiros de software e programadores reconheceram o poder dos princípios da orientação a objetos para lidar com o crescimento do tamanho e da complexidade do software. Hoje, as linguagens com conceitos de orientação a objetos mais representativas na indústria de software são C++, Java e C#.

Linguagens orientadas a objetos modernas são artefatos poderosos de estruturação de programas. Porém, frequentemente, a sintaxe enriquecida com açúcares sintáticos, a semântica complexa, e o conjunto abrangente de bibliotecas suportadas por essas linguagem escondem os princípios essenciais da OOP. Nesta seção, são apresentadas as características essenciais de linguagens imperativas

[1]Do inglês, *object-oriented programming*.

orientadas a objetos, uma vez que a preservação de seus conceitos fundamentais ao incorporarem o processamento paralelo de memória distribuída é o principal assunto abordado nesta dissertação.

2.1 Princípios da Programação Orientada a Objetos

Os diferentes paradigmas de programação propostos nas últimas décadas propuseram a quebra da complexidade de um software pela sua divisão em um conjunto de partes menores, chamados módulos, que podem ser mais facilmente compreendidos e implementados, tendo suas relações e interações especificadas de forma simples. A esse processo deu-se o nome de *modularização de software*. Cada módulo de software deve implementar um *interesse de software*, que é definido como uma parte conceitual de uma solução tal que a composição dos interesses possa definir a solução necessária pelo software [37]. O processo de modularização baseado em interesses é chamado de *separação de interesses*[2].

A noção concreta de módulo em uma linguagem ou ferramenta de programação depende do paradigma em questão. Por exemplo, programação orientada a objetos utiliza objetos para descrever interesses, enquanto programação funcional utiliza funções (em um sentido matemático). Tipos abstratos de dados (TADs) definem uma noção de módulo também adotada por várias linguagens, como Ada. Linguagens orientadas a objetos modernas são capazes de suportar diversas noções de módulo, incluindo objetos, funções e TADs. As seções seguintes apresentam os princípios da programação orientada a objetos.

2.1.1 Objetos

Na programação imperativa, um *objeto* é uma entidade do software em execução constituída pelas seguintes partes:

- Um *estado* definido por um conjunto de objetos internos chamados *atributos*;
- Um conjunto de subrotinas chamadas de *métodos*, que define o conjunto de *mensagens* válidas que o objeto pode aceitar, capazes de alterar o seu estado.

Os métodos de um objeto definem suas transformações de estado válidas, as quais definem seu sentido computacional. As assinaturas dos métodos de um objeto formam sua *interface*. Um programa orientado a objetos é composto de um conjunto

[2]do inglês, *separation of concerns*.

de objetos que coordenam suas tarefas por invocação mútua de seus métodos, cooperando a fim de implementar o interesse da aplicação.

2.1.2 Princípio do Encapsulamento

O princípio mais importante por trás da orientação a objetos (OO) é o *encapsulamento*, através do qual um objeto que conhece a interface de um outro objeto não precisa fazer suposições sobre detalhes de sua implementação (atributos e código-fonte dos métodos) para usar suas funcionalidades. Para isso, é suficiente que o objeto concentre-se na interface dos objetos aos quais depende. De fato, uma linguagem OO não permite que um objeto acesse o estado interno de um outro objeto, estaticamente ou dinamicamente, ao oferecer acesso apenas através dos métodos de sua interface.

O encapsulamento facilita que programadores concentrem-se em detalhes relevantes sobre a estrutura interna de uma implementação particular de um objeto. De fato, os detalhes de implementação e os atributos de um objeto podem ser completamente modificados sem afetar as partes do software que dependem do objeto, desde que sua interface, e seu comportamento sejam preservados. Nesse sentido, o encapsulamento é uma propriedade importante da orientação a objetos para lidar com diferentes graus de escala e complexidade de um software. O encapsulamento oferece aos programadores a possibilidade de trabalhar com níveis maiores de segurança, somente permitindo acesso sobre determinadas partes do estado do programa quando for essencial e seguro.

2.1.3 Classes de objetos

Uma *classe* é definida como um conjunto de objetos similares, que possuem um conjunto similar de *atributos* e *métodos*[3]. Classes podem também ser introduzidas como protótipos de objetos, especificando os atributos e métodos de objetos instanciados durante a execução.

Seja A uma classe com um conjunto α de atributos e um conjunto μ de métodos. Uma nova classe A' pode ser derivada de A, com um conjunto α' de atributos e um conjunto μ' de métodos, tal que $\alpha \subseteq \alpha'$ e $\mu \subseteq \mu'$. Isso é chamado de derivação por *herança* [43], onde A é uma *superclasse* (generalização) de A', e A' é uma *subclasse*

[3]Nós nos abstraimos de definir rigorosamente similaridade de métodos/atributos. Para nossos propósitos, poderíamos definir que dois atributos são similares se eles representam referências aos objetos da mesma classe ou variáveis do mesmo tipo, onde métodos similares possuem a mesma assinatura e implementação. Entretanto, similaridade pode ser definida em diferentes níveis de abstração. Por exemplo, é possível considerar que dois métodos são similares se eles possuem a mesma assinatura com implementações distintas.

(especialização) de A. Pelo *princípio da substituição*, um objeto de uma classe A' pode ser usado no contexto onde um objeto da classe A é requisitado, pois todos os estados internos e transformações de estados de A são válidos em A'.

Herança de classes pode ser simples ou múltipla. Na *herança simples*, uma classe derivada possui exatamente uma superclasse, enquanto que na *herança múltipla* uma classe pode ser derivada de um conjunto de superclasses. Linguagens orientada a objetos modernas, como Java e C#, aboliram a flexibilidade da herança múltipla, presente em C++, adotando o sistema de herança simples inicialmente proposto pelo Smalltalk. Para lidar com esses casos de herança múltipla, o projeto da linguagem Java introduziu a noção de *interface*. Em uma interface, são declarados os métodos que devem fazer parte dos objetos que a implementam. Interfaces definem a noção de tipo de objetos e suas classes.

2.1.4 Princípio da Abstração

Classes e herança trazem quatro diferentes mecanismos de abstração para a programação orientada a objetos [43]:

- *Classificação/Instanciação*: classes agrupam objetos com estruturas similares (métodos e atributos). Objetos representam *instâncias* de classes;
- *Agregação/Decomposição*: é a habilidade de ter objetos como atributos de outros objetos. Desse modo, um conceito representado por um objeto pode ser descrito pela orquestração de um conjunto de objetos representando suas partes constituintes, formando uma hierarquia de objetos que representa a estrutura por atrás do conceito;
- *Generalização/Especialização*: consequência do conceito de derivação de classes por herança, que torna possível reconhecer características comuns entre diferentes classes de objetos através de superclasses. Tal habilidade torna possível o *polimorfismo* de subtipos, típico em linguagens orientadas a objetos, onde uma referência de um objeto, ou variável, que é *tipada* com uma classe pode referir-se a um objeto de qualquer uma das suas subclasses;
- *Agrupamento/Individualização*: é a utilização de coleções de classes que agrupam de objetos com interesses comuns. Através do polimorfismo, tais coleções de classes relacionadas, por relações de herança, podem ser válidas.

2.1.5 Independência Funcional

Independência funcional é uma propriedade importante dos objetos a ser alcançada no projeto das classes a partir das quais são instanciados. É uma medida da independência entre os objetos que constituem uma aplicação. É particularmente importante para o propósito dessa dissertação. Independência funcional é calculada por duas medidas: *coesão* e *acoplamento*. Coesão mede o quão focado um objeto está em uma certa responsabilidade. Portanto, uma classe com alta coesão expressa um interesse único e bem definido. O acoplamento de uma classe mede seu grau de dependência em relação a outras classes. Baixo acoplamento significa que modificações em uma classe tendem a causar menores efeitos em outras classes dependentes. Além disso, o baixo acoplamento minimiza a propagação de erros em classes relacionadas. É possível concluir que independência funcional é melhor quando há alta coesão e baixo acoplamento.

2.1.6 O Princípio da Modularidade

Modularidade é uma maneira de gerenciar a complexidade do software, promovendo a divisão de sistemas complexos e de larga escala em partes mais simples e mais facilmente gerenciáveis. Existem critérios aceitos para classificar o nível de modularidade de um artefato de construção de programas: *decomponibilidade*, *componibilidade*, *inteligibilidade*, *continuidade*, e *proteção* [36].

A orientação a objetos promove a organização do software em *classes* onde os objetos que atuam na aplicação irão ser instanciados durante a execução. De fato, classes são as estruturas fundamentais de softwares de arquitetura orientada a objetos. Em um bom projeto, as classes capturam conceitos simples e bem definidos no domínio da aplicação, orquestrando-os para que a aplicação execute na forma de objetos (*decomponibilidade*). Classes promovem o reuso de partes de software, uma vez que o conceito capturado pela classe de objetos pode estar presente em diversas aplicações (*componibilidade*). Mecanismos de abstração tornam possível a reutilização somente de partes de classes que são comuns entre objetos em aplicações distintas. Encapsulamento e alto grau de independência funcional promovem independência entre classes, tornando possível entender o significado de uma classe sem examinar o código das outros classes envolvidas (*inteligibilidade*), além de evitar a propagação de modificações nos requisitos de uma implementação de uma dada classe para outras classes (*continuidade*). Finalmente, mecanismos de exceção tornam possível restringir o escopo do efeito de erros de execução (*proteção*).

2.2 Programação Paralela

A exploração da concorrência em um software para melhorar o seu desempenho vem da execução paralela de tarefas, independentes entre si ou não, para aproveitar melhor os recursos de computadores com múltiplas unidades de processamento. Nas últimas décadas, os computadores apresentaram um aumento exponencial de capacidade de processamento, enquanto a velocidade de acesso memória cresceu de forma apenas linear. Assim, a paralelização de tarefas trouxe novas possibilidades para alcançar maior desempenho das máquinas.

Para realizar a especificação de tarefas que serão executadas de forma paralela, é necessário a utilização de linguagens de programação, *frameworks*, bibliotecas de subrotinas (MPI, OpenMP [39]) ou outros artefatos voltados ao suporte a programação paralela. Porém, é preciso decidir de que forma a interação entre tarefas paralelas será realizada. As duas maneiras mais comuns são:

- Espaço de dados compartilhados;
- Troca de mensagens.

O uso de um espaço de dados compartilhados é uma maneira de interação entre processos através de variáveis acessíveis a todos os processadores. Programar para esse tipo de plataforma é geralmente mais simples, uma vez que existem apenas interações com variáveis armazenadas na memória compartilhada entre os processos. A maior dificuldade é garantir que acessos concorrentes não acarretem interferências entre processos [1], o que é menos comum em programação paralela, onde prioritariamente as tarefas acessam conjuntos disjuntos de dados na memória.

A troca de mensagens explícitas entre processos através de uma rede de comunicação abstrata é uma outra forma de promover interações entre processos, quando encontram-se localizados em unidades de processamento distintas, as quais não possuem uma memória compartilhada. Para a comunicação ocorrer, é necessário expressar o envio e o recebimento de dados através de primitivas de comunicação. Além de informar que dado será enviado, deve existir uma forma de identificação de processos para que tais mensagens possuam destino correto e não ambíguo. As bibliotecas de subrotinas mais comuns para a assim chamada programação paralela por passagem-de-mensagens são o MPI (*Message Passing Interface*) e a PVM (*Parallel Virtual Machine* [24]). A primeira é mais voltada para plataformas de computação paralela onde as unidades de processamento são homogêneas, tendo

por isso alcançado popularidade em *clusters*, enquanto a última foi desenvolvida para processamento paralelo em redes de estações de trabalho heterogêneas, as NOWs (*Network of Workstations*), precursoras dos *clusters* na década de 1990.

2.2.1 Programação Paralela em Espaço de Dados Compartilhados

Em arquiteturas de memória compartilhada, programação paralela é comumente descrita através do uso de processos mais leves chamados de *threads* que executam tarefas concorrentemente. Em geral, um *thread* só existe dentro de um processo e pode compartilhar recursos entre *threads* de um mesmo processo. Um *thread* é definida como um fluxo de instruções que executam independentemente de outros *threads* dentro de um programa. A implementação de um *thread*, em um sistema operacional, pode ser encarada como um processo ou como linhas de execução dentro de um processo. No segundo caso, essas linhas de execução compartilham espaço de endereçamento, mas podem executar em processadores diferentes. Assim, computações irão ocorrer paralelamente no programa em questão.

Nas seções seguintes, são apresentadas três maneiras bem conhecidas de trabalhar com memória compartilhada e *threads*. São elas: *Pthreads*, *OpenMP* (*Open Multi-Processing*) e o uso de objetos *thread* . Outras maneiras menos difundidas são ainda utilizadas, como o uso de Atores na linguagem Scala [26].

Programação com Pthreads

Historicamente, diferentes versões de interfaces de programação para uso de *threads* foram especificadas e implementadas por várias empresas de hardware. Cada versão possuía a sua própria concepção de *thread* e como deviam ser programadas. Nesse contexto, surgiu a especificação do *Pthreads* (*POSIX Threads*) para o sistema operacional UNIX em 1995. Desde então, a maioria das plataformas de computação paralela em memória compartilhada (multiprocessada) suportam esse padrão. Em *Pthread*, um *thread* é representado pela estrutura `pthread_t` e a partir dela é possível aplicar várias operações, tais como: criação de *threads* (*pthread_create*), junção de *threads* (*pthread_join*) e cancelamento de *threads* (*pthread_cancel*). Após um *thread* ser criado, é possível esperar por uma sincronização com outro *thread* e até cancelar um *thread* durante sua execução. Além da junção de threads, existem outros métodos de sincronização de *threads*, como o uso de semáforos binários (*mutex*) ou variáveis condicionais.

Na Figura 2.1, é apresentado um exemplo simples de criação e destruição de threads na linguagem C. Entre as linhas 5 e 11, está descrita a tarefa a ser executada

```
1   #include <pthread.h>
2   #include <stdio.h>
3   #define NUM 5
4
5   void *PrintHello(void *threadid) {
6     long tid;
7     tid = (long)tid;
8     printf("Hello World! Thread #%ld!\n", tid);
9     pthread_exit(NULL);
10  }
11
12  int main (int argc, char *argv[]) {
13    pthread_t threads[NUM];
14    int err;
15    long t;
16    for(t=0; t<NUM; t++) {
17      int rc
18      rc = pthread_create(&threads[t],
19                          NULL, PrintHello, (void *)t);
20      if (rc){
21        printf("ERRO\n");
22        exit(-1);
23      }
24    }
25    pthread_exit(NULL);
26  }
```

Figura 2.1: Exemplo de criação de threads

```
int main(int argc, char *argv[]) {
    int a,b,c;
     #pragma omp parallel
    {
      // Seção paralela.
      // Threads inicializadas

      ..........
      Variáveis 'a', 'b' e 'c' são compartilhadas.
      ..........

      // Threads finalizadas.
      // Fim da seção paralela.
    }
    return 0;
}
```

Figura 2.2: Exemplo de uma seção paralela

por todas as *threads* criadas, cuja função é simplesmente exibir na saída padrão uma mensagem com seu respectivo *rank*. Na função principal do programa, chamada de *main* na linguagem C, está a descrição de como definir as estruturas que irão representar um *thread* (linha 14) e como instanciar para executar a tarefa descrita anteriormente (linha 18). Por fim, na linha 24, temos a finalização de todas as *threads* criadas.

Vale ressaltar que *Pthreads* é um mecanismo de programação concorrente de propósito geral, não apenas voltado para os requisitos de programação paralela. O uso eficiente do paralelismo oferecido pelas threads quando executadas em processadores ou núcleos de processamento distintos é uma preocupação inerente do programador. Para os propósitos mais específicos da programação paralela, foi desenvolvido o OpenMP, explicado na próxima seção.

OpenMP

OpenMP [39] é uma interface portável para programação paralela em memória compartilhada, utilizada como uma extensão das linguagens C, C++ e Fortran. Foi criada em 1997, inicialmente somente para a linguagem Fortran. Encontra-se na versão 3.1. É composto de subrotinas, diretivas de compilação e variáveis ambientes que gerenciam o uso de múltiplas *threads* em seções de programas.

Em C/C++, são utilizadas cláusulas de compilação chamadas de #pragma que

```
int main(int argc, char *argv[]) {
    int N = 100000;
    int i, data[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        data[i] = 2 * i;

    return 0;
}
```

Figura 2.3: Exemplo de um laço for em C com OpenMP

descrevem o comportamento de uma certa seção paralela, especificam como as variáveis serão compartilhadas e até restrições de sincronização como seções críticas. Para criar uma seção de código paralela basta utilizar a cláusula parallel. A Figura 2.2 apresenta um exemplo muito simples de criação de seção paralela. É importante notar que implicitamente as variáveis a, b e c serão compartilhadas entre as *threads* inicializadas dentro do bloco que usa a diretiva #pragma omp parallel.

Outra construção comum é o uso da cláusula parallel for para paralelizar automaticamente um laço for estruturado. Assim, uma certa quantidade de *threads* criados na seção paralela irão dividir o trabalho de percorrer as iterações do laço. A Figura 2.3 demonstra o uso dessa cláusula. O vetor data será modificado por diferentes *threads* durante o processamento do laço. Muitas outras diretivas podem ser utilizadas para controlar diferentes aspectos do paralelismo, especialmente a sincronização entre as *threads*, tais como shared, critical, atomic, ordered, barrier, e outras.

Além das cláusulas, é possível utilizar subrotinas que configuram as seções paralelas. Tais funções podem ser usadas para definir número de *threads* a serem disparadas (o valor padrão é o número núcleos de processamento da plataforma), descobrir a identificação de um *thread* (rank) e até funções de bloqueio de seções paralelas, para sincronização.

Como já mencionado, o OpenMP foi desenvolvido para os requisitos de programação paralela, evitando os custos de criação e destruição de tarefas ao tornar essas operações transparentes ao programador.

Objeto Thread

Uma abordagem bastante usual em linguagens orientadas a objeto é a utilização de um um objeto autônomo para realizar uma *thread*. Dessa forma, caso o usuário queira definir o funcionamento de uma *thread* específica, basta utilizar a herança entre classes. Um objeto *thread* pode ter métodos para sincronização, junção de *threads*, cancelamento, adormecimento, etc.

Por exemplo, na linguagem Java, uma classe precisa herdar da classe Thread e implementar a função run para definir seu comportamento quando esse *thread* for disparado. Para controle de interferência entre *threads* que realizam invocações concorrentes à objetos de um programa, Java oferece recursos que permitem implementar tais objetos como monitores [27], como métodos sincronizados e variáveis condicionais, a fim de protegê-los de acessos concorrentes indevidos.

2.2.2 Programação Paralela por Passagem-de-mensagens com MPI

MPI [20] é uma especificação padrão para construção de bibliotecas de subrotinas de programação paralela por passagem-de-mensagens, desenvolvido em meados da década de 1990 por um consórcio que reunia representantes da indústria e da academia. Seu principal interesse era estabelecer uma interface de passagem-de-mensagens de implementação eficiente para arquiteturas paralelas de memória distribuída com um conjunto homogêneos de unidades de processamento, conseguindo assim sobrepor as várias interfaces proprietárias desenvolvidas por diferentes fabricantes de supercomputadores ao longo da década de 1980.

Foi observado que tal diversidade de soluções proprietárias resultava em alto custo para os usuários de supercomputadores paralelos devido a fraca portabilidade de programas entre arquiteturas de fabricantes distintos. Além disso, a carência de práticas comuns (padrões de programação paralela) impossibilitava a evolução técnica e a disseminação de técnicas de programação paralela. O suporte à construção de interfaces de alto nível e bibliotecas de subrotinas para computação científica e engenharia foi originalmente a principal premissa para o projeto do MPI. Entretanto, MPI tornou-se útil para o desenvolvimento de aplicações finais. A especificação é mantida desde o seu início pelo Fórum MPI [35].

MPI é atualmente o principal representante de programação paralela por passagem-de-mensagens. Talvez seja a única interface de programação paralela portável e de propósito geral que explore eficientemente o desempenho de plataformas de computação de alto desempenho. Desde o final da década de

1990, qualquer nova implantação de *cluster* ou MPP[4] tem suportado algum tipo de implementação do MPI. De fato, a maioria dos fabricantes de supercomputadores adotam o padrão MPI como sua principal interface de programação, oferecendo implementações otimizadas para suas arquiteturas. MPI é também considerado uma das principais razões do crescimento e da popularização da computação em *clusters*, dado a existência de implementações eficientes de código aberto para plataformas baseadas no sistema operacional Linux. Tornou-se popular até mesmo em computadores de memória compartilhada, especialmente em arquiteturas NUMA (*Non-Uniform Memory Access*), uma vez que sua grande aceitação por programadores de software paralelo é visto como uma maneira de reduzir a curva de aprendizado de programação paralela para essas arquiteturas.

Muitas implementações do MPI, têm sido desenvolvidas para suportar plataformas de computação de diferentes propósitos, muitas das quais de código-aberto. Algumas implementações populares de caráter portável são: MPICH [30], OpenMPI [23], e LAM-MPI [10]. Também existem versões de especificações do MPI em linguagens não suportadas oficialmente, tais como: Boost.MPI [21] (C++), MPI.NET [25] (C#) e MPJ [4] (Java) .

O MPI fórum suporta atualmente duas versões oficiais, para Fortran e C: MPI-1 e MPI-2. MPI-2 é uma extensão do MPI-1 com muitas inovações propostas pela comunidade de usuários do MPI. Centenas de subrotinas são suportadas, com diferentes propósitos, tais como:

- comunicação ponto-a-ponto (MPI-1);
- comunicação coletiva (MPI-1);
- escopos de comunicação (MPI-1);
- topologias de processos (MPI-1);
- tipos de dados (MPI-1);
- comunicação unilateral (MPI-2);
- criação diânmica de processos (MPI-2);
- entrada e saída paralela em arquivos (MPI-2).

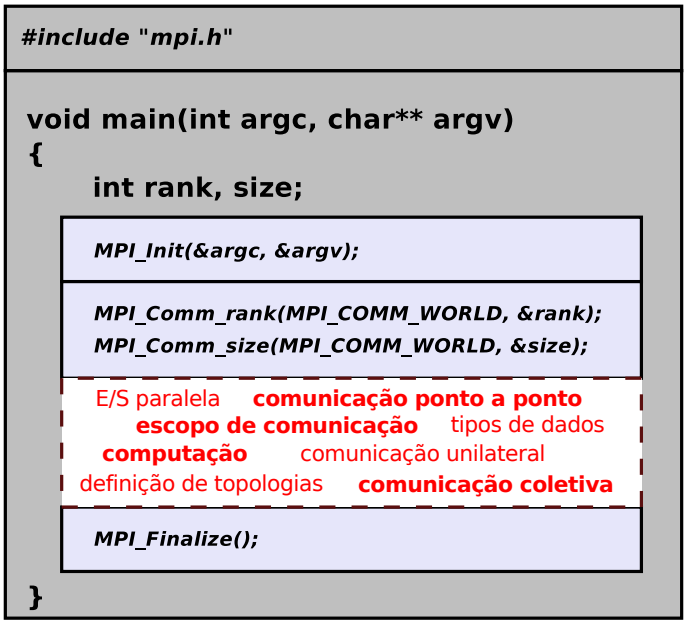


Figura 2.4: Programa MPI

Modelo de Programação do MPI

Na especificação original do padrão MPI, um programa paralelo é um único programa executável que é executado em cada unidade de processamento do computador paralelo, modelo de programação conhecido como SPMD (*Single Program Multiple Data*), onde cada processo executa a mesma computação sobre um subconjunto de uma estrutura de dados distribuída (*paralelismo de dados*) ou aplicada a diferentes valores de parâmetros de entradas (*varredura de parâmetros*). Cada processo é identificado por um inteiro distinto chamado *rank*, cujo valor varia de 0 a $size - 1$, onde *size* é o número de processos em execução.

Processos MPI podem executar computações distintas sobre diferentes estruturas de dados, usando *ranks* para distinguir processos com diferentes propósitos. Portanto, MPI também suporta MPMD (*Multiple Program Multiple Data*), uma generalização do SPMD onde o programa MPI consiste de diversos programas representando diferentes tipos de processos.

A estrutura completa de um programa MPI está ilustrada na Figura 2.4. O cabeçalho "mpi.h" contém os protótipos das subrotinas do MPI, assim como definições de constantes e declaração de tipos de dados. As chamadas para *MPI_Init* e *MPI_Finalize* marcam, respectivamente, o início e o fim do uso de subrotinas MPI. A primeira inicializa o ambiente MPI e a segunda libera recursos utilizados

[4] *Massively parallel processor.*

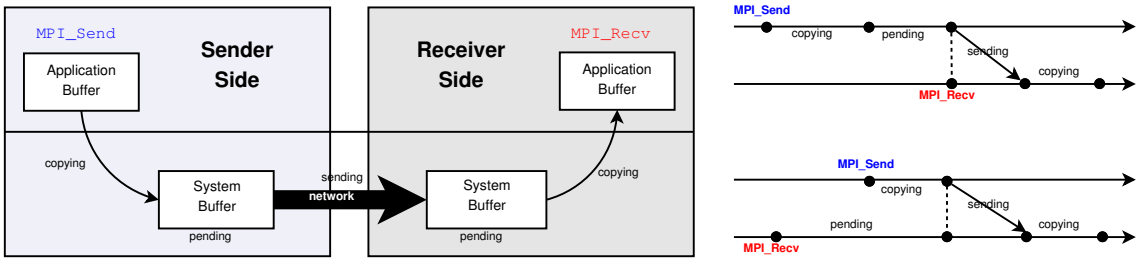


Figura 2.5: Semântica de Comunicação MPI Ponto-a-Ponto

durante a execução. As chamadas de *MPI_Comm_rank* e *MPI_Comm_size* tornam possível que um processo descubra sua identificação (*rank*) e número de processos em execução (*size*), respectivamente.

Em programas paralelos onde processos precisam se referir uns aos outros, isto é, processos que executam subrotinas de comunicação, os valores *rank* e *size* são essenciais. *MPI_COMM_WORLD* é uma constante do tipo *MPI_Comm*, representando o comunicador global (escopo de comunicação), que envolve todos os processos em execução.

Ainda na Figura 2.4, o código contido no retângulo pontilhado apresenta uma sequência de movimentações de dados efetuadas por chamadas de subrotinas MPI para realizar uma passagem de mensagem. Nas seções seguintes, nós apresentamos um visão geral dos subconjuntos de subrotinas do MPI.

Comunicação Ponto-a-Ponto

As subrotinas básicas de comunicação ponto-a-ponto são *MPI_Recv* e *MPI_Send*, onde o processo transmissor envia um "*buffer*" de dados, que contém um certo número de itens de um certo tipo de dados MPI, para um processo receptor. Em conjunto com *MPI_Init*, *MPI_Finalize*, *MPI_Comm_size*, e *MPI_Comm_rank*, eles formam o subconjunto básico de subrotinas do MPI, com poder de expressão para desenvolver programas paralelos por passagem-de-mensagens. MPI também oferece um conjunto rico de subrotinas de comunicação ponto-a-ponto, com várias semânticas de sincronização.

Existem dois modos principais de comunicação para subrotinas de comunicação ponto-a-ponto no MPI: *blocante* e *não-blocante*. Para entender a semântica dessas subrotinas, é importante conhecer alguns detalhes sobre como a comunicação no MPI funciona, o que está ilustrado na Figura 2.5. Do lado esquerdo, o *buffer de aplicação* e o *buffer de sistema* estão representados. O *buffer* de aplicação é passado pelo programador para a subrotina de comunicação (*MPI_Send*/*MPI_Recv*), onde o

dado a ser enviado ou recebido está acessível pelo programa. De fato, é acessível por uma variável regular do programa, o que é bastante conveniente para programas que executam computações sobre vetores multidimensionais, armazenados em endereços físicos contíguos na memória.

O *buffer* de sistema é um *buffer* interno onde o dado é copiado do *buffer* de aplicação pela operação de envio (lado do transmissor) ou pela operação de recebimento (lado do receptor). Entretanto, não é acessível pelo programa. Inconsistências podem ocorrer se o programador acessar o *buffer* de aplicação antes de ser copiado no *buffer* de sistema ou diretamente mandado para o receptor. No lado direito da Figura 2.5, a sequência de operações em uma comunicação é descrita, envolvendo a aplicação e os *buffers* de sistema. No primeiro cenário (topo), a operação de envio é executada antes da operação de recebimento. No segundo cenário, é o receptor que chama a subrotina de recebimento antes do transmissor. A semântica de cada subrotina de comunicação ponto-a-ponto do MPI depende de como cada subrotina faz uso desses *buffers*.

Em uma operação de comunicação blocante, o transmissor ou o receptor retorna o controle ao "chamador" quando o acesso para o *buffer* de aplicação pelo programa é seguro. No recebimento blocante, tal estado é alcançado quando o dado recebido tenha sido copiado para o *buffer* de aplicação. No envio blocante, é alcançado após o dado enviado ter sido copiado para o *buffer* de sistema ou diretamente enviado para o receptor. O envio blocante tem três modos: *síncrono* (*MPI_Ssend*), *bufferizado* (*MPI_Bsend*), e *imediato* (*MPI_Rsend*).

No envio síncrono blocante, não existe *buffer* de sistema. O dado é enviado para o receptor diretamente do *buffer* de aplicação. Assim, *MPI_Ssend* somente retorna o controle para a aplicação após o acoplamento das subrotinas de envio e recebimento, quando a comunicação se completa. No envio blocante *buffer*izado, existe um *buffer* de sistema explicitamente alocado pelo programador utilizando a subrotina *MPI_Attach_buffer*, cujo o espaço deve ser suficiente para comportar todas as chamadas pendentes à *MPI_Bsend*. Se o *buffer* estiver cheio em uma chamada a *MPI_Bsend*, ou seja, existem muitas chamadas pendentes, o comportamento da chamada será como um envio blocante síncrono. Em um envio blocante imediato, não existe *buffer* de sistema em ambos os lados (recebor e transmissor). Desse modo, quando um *MPI_Rsend* é executado, um *MPI_Recv* deve ter sido executado em um instante anterior de tempo.

Vale a pena salientar que na maioria das implementações do MPI , *MPI_Send*

é um envio blocante *buffer*izado, porém com um pequeno *buffer* pré-alocado pelo ambiente MPI. Por essa razão, programadores dizem que tal comportamento é síncrono e blocante para mensagens grandes (em geral, acima de 64KB). O documento oficial do MPI não especifica a semântica do *MPI_Send* a ser seguida pelos implementadores. Dessa forma, um programa que utiliza *MPI_Send* deve estar preparado para funcionar tanto no caso de envio síncrono quanto no caso de envio *buffer*izado.

Para cada subrotina blocante, existe seu correspondente não-blocante, cujo nome é composto pelo prefixo "*I*" (e.g. *MPI_Irecv*, *MPI_Isend*, *MPI_Issend*, *MPI_Ibsend*, *MPI_Irsend*). Subrotinas não blocantes não aguardam até que o acesso ao *buffer* de aplicação seja seguro antes de retornar o controle ao chamador da função. O controle é retornado com um manipulador do pedido, um valor do tipo *MPI_Request*. A comunicação ocorre no plano de fundo, sendo possível sobrepor computação útil com comunicação, minimizando problemas de sincronização e evitando certos estados de impasse na interação entre processos. O programador é responsável por evitar acessos errôneos ao *buffer* da aplicação até que a comunicação seja finalizada. Um conjunto de subrotinas existem para testar e esperar pelo término de um ou mais pedidos (*MPI_Test*, *MPI_Testall*, *MPI_Testany*, *MPI_Testsome*, *MPI_Wait*, *MPI_Waitall*, *MPI_Waitany*, *MPI_Waitsome*). Para isso, o programador deve usar os manipuladores de pedido retornados pelas subrotinas de comunicação não-blocante para se referir a operações pendentes.

Comunicação Coletiva

O MPI suporta subrotinas que encapsulam operações de comunicação envolvendo vários processos comumente usados em algoritmos paralelos. Existem operações para disseminação de dados de um processo para vários outros (*MPI_Bcast* e *MPI_Scatter*) e vice-versa (*MPI_Gather*). Nesse último caso, possivelmente executando uma operação no conjunto de dados recebidos (*MPI_Reduce*). Existem também operações onde todos os processos envolvidos disseminam dados para cada um dos outros (*MPI_Alltoall*, *MPI_Allgather*), e podem executar operações de agregação nos dados recebidos (*MPI_Allreduce*). Algumas operações são personalizadas (*MPI_Scatter*, *MPI_Alltoall*, *MPI_Allgather*, *MPI_Allreduce*), onde o transmissor escolhe diferentes pedaços de dados endereçados a diferentes processos. Reduções parciais também são suportadas (*MPI_Scan*, *MPI_Reducescan*).

Grupos e Communicadores

Todas as operações de comunicação discutidas acima executam em um contexto de um comunicador, um valor do tipo *MPI_Comm*. O comunicador básico é o valor constante *MPI_COMM_WORLD*, que envolve todos os processos em execução do programa paralelo. Utilizando-se da abstração auxiliar de *grupos de processos* do tipo *MPI_Group*, um programador MPI pode criar comunicadores arbitrários envolvendo qualquer subconjunto de processos. O principal uso de grupos e comunicadores é encapsular operações de comunicação em diferentes escopos. De fato, os comunicadores foram propostos para que fosse possível utilizar diferentes bibliotecas científicas paralelas em diferentes grupos de processos em um único programa paralelo evitando conflitos entre mensagens.

2.3 Programação Paralela Orientada a Objetos

Diferentes abordagens para unir programação orientada a objetos e programação paralela por passagem-de-mensagens são abordadas nesta seção. Desde o simples encapsulamento de bibliotecas MPI em objetos, como em Boost.MPI, MPI.NET e MPJ, passando por extensões sobre linguagens OO, como no POP-C++, Javaparty, ParoC++, Titanium, e até a proposta de novas linguagens, como Fortress, Chapel e X10 do projeto HPCS.

2.3.1 Interfaces MPI em Linguagens OO

Boost.MPI, MPI.NET e mpiJava [5] são tidos como interfaces orientada a objetos do padrão MPI, ao invés de meros invólucros das subrotinas MPI, como no caso da especificação C++ do MPI pelo MPI Fórum. Além da orientação a objetos verdadeira, outro aspecto importante que foi incorporado é a serialização semi-automática de estruturas complexas para envio e recebimento. É importante notar que o uso de objetos e métodos no lugar de estruturas e funções é apenas uma modernização do padrão estruturado do MPI, não oferecendo qualquer ganho de poder para expressar padrões de computação paralela.

Na Figura 2.6, um exemplo de uso da biblioteca Boost.MPI. Inicialmente dois objetos *env* e *comm* são criados representando o ambiente de execução e um comunicador respectivamente. No bloco condicional, vários métodos são utilizados para realizar uma comunicação simples ponto-a-ponto. As operações de comunicação são definidas como métodos do objeto comunicador . A finalização do ambiente é feita pela destruição do objeto *env* ao fim do bloco da função *main*.

```
#include <boost/mpi.hpp>
#include <iostream>
#include <string>
#include <boost/serialization/string.hpp>
namespace mpi = boost::mpi;

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;

  if (world.rank() == 0) {
    world.send(1, 0, std::string("Hello"));
    std::string msg;
    world.recv(1, 1, msg);
    std::cout << msg << "!" << std::endl;
  } else if(world.rank() == 1){
    std::string msg;
    world.recv(0, 0, msg);
    std::cout << msg << ", ";
    std::cout.flush();
    world.send(0, 1, std::string("world"));
  }
  return 0;
}
```

Figura 2.6: Simples programa utilizando Boost.MPI

2.3.2 POP-C++

POP-C++ [38] é uma extensão da linguagem de programação C++. Sua principal motivação é ser uma ferramenta para programar sistemas para ambientes distribuídos heterogêneos, como grades computacionais e plataformas P2P (*peer-to-peer*). Desenvolvido pelo consórcio *GRID & Cloud Computing Group*, essa linguagem foi projetada para tratar diferentes necessidades nessas plataformas, tais como: gerenciamento e descoberta de recursos computacionais, controle de dados remotos e segurança. Tendo sua inspiração no CORBA [44], POP-C++ permite a execução de objetos C++ em ambientes distribuídos através de um novo tipo de objeto, chamado **parallel object**.

Objetos paralelos são objetos que podem ser executados remotamente, coexistindo e cooperando com objetos sequenciais tradicionais durante a execução da aplicação. Entretanto, algumas restrições são necessárias, como a inexistência de dados de acesso público e de variáveis globais no escopo do objeto paralelo, além da impossibilidade de definir operadores e de retornar endereços de memória nos métodos da classe.

Para capturar os requisitos de recursos computacionais, é possível especificar a descrição do objeto. Os seguintes requisitos de recursos podem ser indicados:

- Poder computacional;
- Tamanho de memória;
- Banda larga disponível;
- Localidade (endereço);
- Protocolo de comunicação;
- Codificação de dados.

Na Figura 2.7, é apresentado um exemplo simples de especificação de processamento, memória e protocolo em uma classe de objetos paralelos. Quando esse objeto for executado, o sistema de execução procura um processo que esteja executando em uma plataforma computacional com tais características e o implanta. Caso não exista tal processo, ocorre uma falha.

```
parclass Hello {
 public:
 Hello(int a) @{ od.power(P);
                 od.memory(100,60);
                 od.protocol("socket"); };
```

Figura 2.7: Classe paralela *Hello*

2.3.3 Charm++

Charm++ [29] é uma extensão da linguagem C++ cujo modelo de execução é dirigido por mensagens. Seu desenvolvimento é feito pelo Laboratório de Programação Paralela da Universidade de Illinois desde 1993. Essa linguagem introduz um tipo de processo paralelo chamado *chare*[5] utilizando diferentes estratégias de escalonamentos. Em uma execução dirigida por mensagens, todas as computações são inicializadas em resposta ao recebimento de mensagens. Além disso, todas as chamadas de sistema em Charm++ são não-blocantes, fazendo com que acesso a dados remotos seja feito em estágios separados. Sendo assim, não há bloqueios para um outro processo atuar enquanto um dado remoto está sendo utilizado. Esse projeto procura resolver o problema de desenvolver programas paralelos sem uma modularização adequada. Para tal, impõe a utilização de um modelo dirigido por mensagens, mas não fornece maneiras viáveis de total controle da execução que será produzida a partir dos módulos descritos.

Para trabalhar com esse modelo, a linguagem propõe cinco tipos de objetos: sequencial, concorrente, replicado, vetorial e de comunicação, onde objetos sequenciais são os objetos normais conhecidos na linguagem C++. *Chares* são os objetos concorrentes, *chare groups* são replicações de objetos e *chare array* é um conjunto indexado de tarefas (vetor). Por fim, a estrutura *Message* foi criada para especificar um certo dado a ser enviado e suas funções de manipulação. É importante notar que existe uma distinção clara entre objetos concorrentes e sequenciais por causa do custo do envio de mensagens, que é significante na maioria dos computadores paralelos. Assim, existe um entendimento explícito sobre que partes do código são custosas para a execução paralela. O controle de execução do Charm++ é feito por balanceamento dinâmico de carga guiado pelas mensagens trocadas no sistema. Cada aplicação pode configurar o tipo de balanceamento de carga em sua execução. Alguns tipos são suportados pela linguagem, como por

[5]do arcáico ingles e atual *chore*, significa tarefa.

```
entry [sync]  void bar(int n, int k);
entry [local] void foo(int n, double array[n]);
```

Figura 2.8: Exemplos de entradas

```
public remote class A {
    public int x;
    public static int y;
    public void foo() { ... }
    public static void bar() { ... }
}
```

Figura 2.9: Classe remota *A*

exemplo: randômico, central e por passagem de *token*.

A comunicação entre *chares* é feita por invocação remota de métodos, as quais constitutem as portas de acesso aos objetos concorrentes. Os atributos dessas portas são passados por mensagens. Esses métodos de entrada de um processamento paralelo podem ser caracterizados com diferentes atributos, tais como: síncrono ou assíncrono, imediato, local, e outros. Cabe ao desenvolvedor especificar que tipo de características serão atribuídas a esse método. A Figura 2.8 apresenta alguns exemplos de descrição de métodos de entrada.

2.3.4 JavaParty

Na linguagem Java, é comum o uso de RMI [6] ou bibliotecas de comunicação para mover uma aplicação paralela para um ambiente distribuído. JavaParty é uma extensão das capacidades da linguagem Java para trabalhar com computação distribuída. Classes em JavaParty podem ser declaradas como remotas. Assim, objetos remotos são acessíveis, no ambiente JavaParty, em qualquer parte do sistema.

Uma classe remota descreve uma classe de objetos padrões em Java, mas com acesso remoto disponível. Assim, uma variável declarada em uma classe remota é uma instância de uma variável em um certo objeto remoto. Sintaticamente, não existe diferença no uso de objetos remotos e não-remotos. Além disso, classes remotas podem utilizar-se de monitores remotos, objetos replicados e sincronização de variáveis como já é feito na linguagem Java padrão.

O principal ganho ao usar JavaParty no lugar da linguagem Java é a possibilidade de trabalhar transparentemente com objetos remotos. Assim,

[6] Do inglês, *Remote Method Invocation*

aplicações desenvolvidas utilizando Java podem ser adaptadas para utilizar objetos remotos e habilitar a execução em ambientes distribuídos. Apesar de tornar a comunicação entre processos em algo implícito, o uso dessa linguagem não oferece melhora substancial na forma de modularizar programas paralelos.

2.3.5 Modelo PGAS

PGAS (*Partitioned Global Address Space*) é um modelo de programação paralela que foi proposto para o desenvolvimento de linguagens orientadas a objetos. Desde 2002, algumas linguagens que utilizam este paradigma foram criadas sob o programa HPCS (*High Productivity Computer Systems*) [33] da DARPA[7]: X10 [41], Chapel [12], Fortress [42], e Titanium [45].

O programa HPCS possui dois objetivos principais: impulsionar a performance de computadores paralelos e incrementar sua usabilidade. No modelo PGAS, o programador pode trabalhar com variáveis compartilhadas ou locais sem explicitamente enviar e receber mensagens. Dessa forma, as noções de objeto paralelo e comunicação entre objetos paralelos não existem, como nas outras abordagens até então apresentadas nesta dissertação. Os objetos interagem através do espaço de endereço particionado, apesar da localização ser possivelmente em nós distintos do computador paralelo.

Essa abordagem torna a programação paralela mais fácil, mas pode inserir sobrecargas de desempenho, já que os movimentos de dados na memória são controlados implicitamente pelo sistema. Cada linguagem possui sua própria maneira de expressar paralelismo de dados e de tarefas, através de diferentes formas como: invocação assíncrona de métodos, geração explícita e dinâmica de processos, paralelização de laços e vetores particionados.

[7] Agência de Pesquisa do Departamento de Defesa dos EUA

Capítulo 3

PObC++: Programação Paralela Orientada a Objetos

Neste trabalho, parte-se do pressuposto de que a principal razão para as dificuldades na conciliação entre orientação a objetos e programação paralela para memória distribuída reside na prática usual de misturar interesses e processos na mesma dimensão de decomposição de software [11]. Em programação paralela para memória distribuída, a maior parte dos interesses do software devem ser realizados cooperativamente por um grupo de processos. Consequentemente, um objeto individual em um programa paralelo, o qual deve realizar algum interesse da aplicação, deve ser também distribuído, isto é, composto por partes cada qual localizada em uma unidade de processamento do computador paralelo. Porém, na prática comum, um objeto está localizado no espaço de endereçamento de uma unidade de processamento individual, sendo necessário um grupo de objetos para expressar um único interesse do software paralelo.

O diagrama no lado esquerdo da Figura 3.1 ("POR PROCESSOS") ilustra a prática comum em programação paralela em linguagens OOP, onde objetos individuais executam em um espaço de endereçamento único e interesses paralelos são implementados por um grupo de objetos que comunicam-se por passagem-de-mensagens usando bibliotecas padrões de passagem-de-mensagens, tais como MPI ou *sockets*, ou invocação remota de métodos, como Java RMI. Nessa abordagem, não existe distinção clara entre mensagens para interação paralela e para coordenação entre objetos. No caso do uso de invocação remota de métodos para comunicação entre objetos paralelos, relações cliente-servidor não são apropriadas para comunicação entre processos pares que interagem cooperativamente, o que

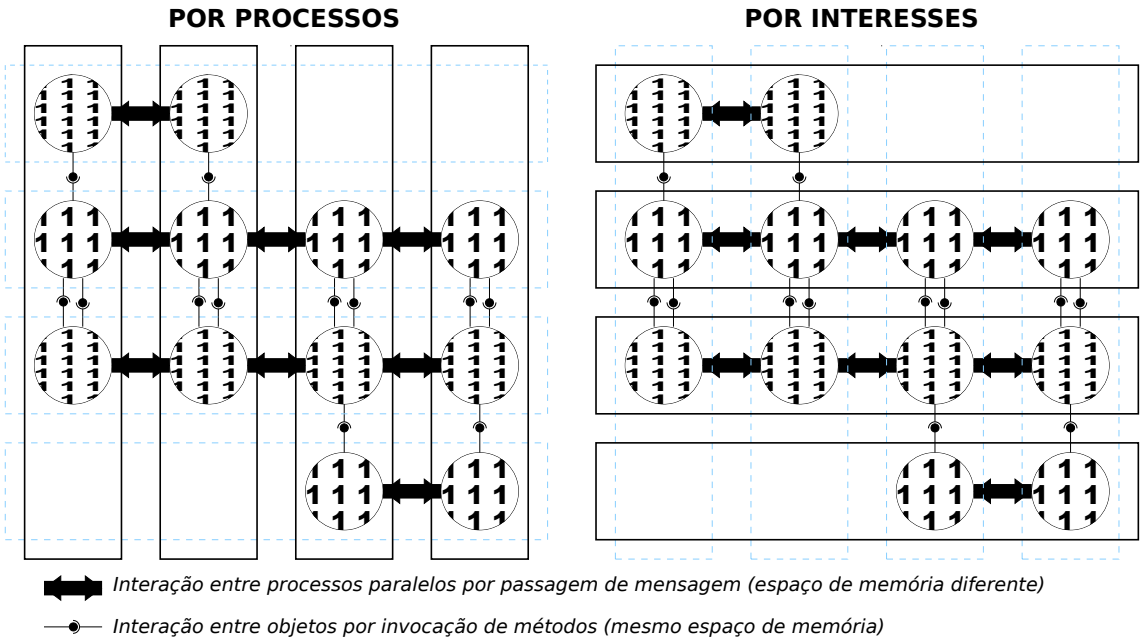


Figura 3.1: Processos vs. Perspectivas de Objetos

é comum em programação paralela. Por outro lado, o uso de bibliotecas de passagem-de-mensagens para a mesma finalidade é uma forma clandestina de comunicação entre os objetos, potencialmente violando o seu encapsulamento e compromentendo a sua independência funcional.

O diagrama no lado direito da Figura 3.1 ("POR INTERESSES") ilustra a prática mais adequada para programação paralela orientada a objetos (OOPP[1]). Objetos que antes cooperavam para implementar um interesse paralelo agora constituem unidades de um *objeto paralelo*, aqui referido como *p-objeto*. Interesses do software são agora encapsulados em um *p-objeto*, onde interações paralelas não são mais clandestinas, pois encontram-se encapsulados no *p-objeto*. De fato, interação paralela e coordenação de objetos são agora distinguíveis em diferentes níveis de uma hierarquia, levando à distinção entre comunicação *intra-objeto* e *inter-objeto*. Comunicação intra-objeto pode ser executada por passagem-de-mensagens, que é mais adequada para interação paralela entre pares de unidades. Por outro lado, comunicação inter-objeto pode ser implementada por invocação local de métodos.

A partir das considerações acima, argumenta-se que uma abordagem de decomposição completamente centrada em interesses promove uma maior *independência funcional* dos objetos que constituem um software paralelo, agora chamados objetos paralelos, eliminando o *acoplamento* adicional entre objetos que

[1]Utilizaremos a sigla do inglês para *Object-Oriented Parallel Programming*.

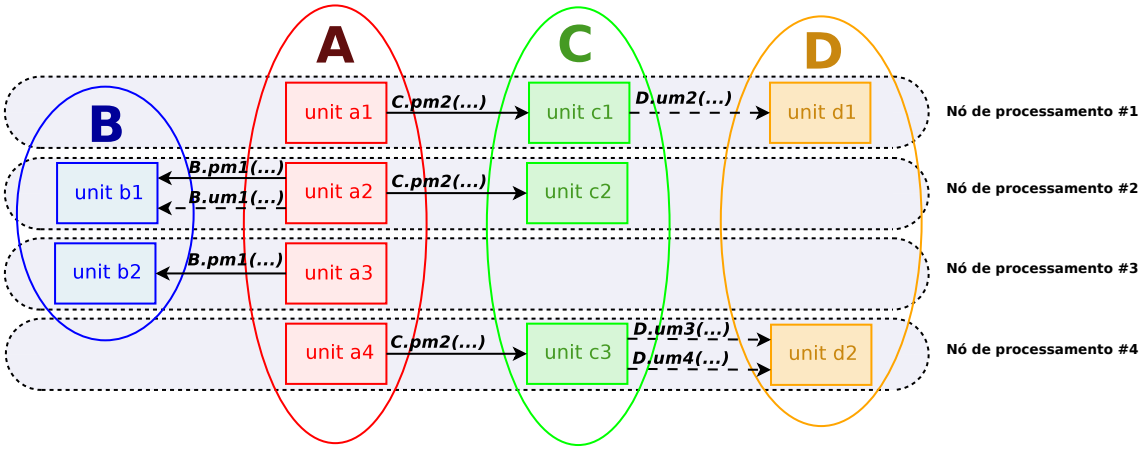


Figura 3.2: Invocação de Métodos Paralelos e de Unidades

resulta de uma abordagem de decomposição centrada em processos. Propõe-se então uma linguagem para OOPP, chamada PObC++, uma extensão paralela para a linguagem C++ implementada sobre a biblioteca MPI para habilitar a criação, comunicação e sincronização de processos. C++ é adotada porque é amplamente aceita e disseminada entre programadores interessados em computação de alto desempenho, principalmente em aplicações nos domínios das ciências computacionais e engenharias. Entretanto, os conceitos de OOPP introduzidos ao C++ podem ser facilmente também incorporados à Java ou C#, duas linguagens de programação amplamente utilizadas em aplicações de propósito geral.

PObC++ admite o estilo de programação paralela do padrão MPI, cuja importância em CAD foi discutida na Seção 2.2.2. Pode ser distinguido de outras alternativas de programação orientada a objetos paralela nos seguintes aspectos:

- Objetos mantêm a unidade da implementação de interesses, pois cada unidade de um *p-objeto* pode representar o papel de um processo com respeito a um interesse paralelo;
- Objetos enviam mensagens para outros objetos somente por invocação de métodos, evitando comunicações clandestinas através de passagem-de-mensagens por meio de interfaces de comunicação de mais baixo nível;
- Paralelismo totalmente explícito é suportado por uma noção explícita de processo e comunicação por passagem-de-mensagens intra-objeto, provendo total controle sobre interesses de processamento paralelo (balanceamento estático de carga, localidade dos dados, e outros).

As próximas seções introduzem os conceitos principais e abstrações por trás da OOPP, usando exemplos simples. PObC++ empenha-se em reconciliar programação por passagem-de-mensagens totalmente explícita, no estilo do MPI, com orientação a objetos, pela introdução de um conjunto mínimo de novos conceitos e abstrações. Por essa razão, decisões pragmáticas têm sido feitas para suportar o MPI sem quebra de princípios por trás da OOPP. Acredita-se que tal abordagem pode levar uma melhor curva de aprendizagem para novos usuários do PObC++. Trabalhos futuros irão estudar como aprimorar a OOPP, fazendo-a mais atrativa aos programadores de programas paralelos.

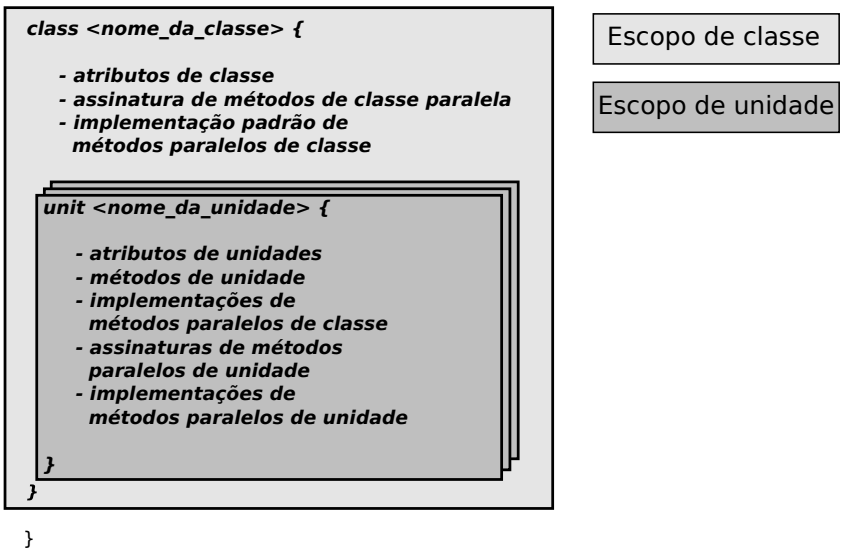
3.1 Objetos Paralelos

Um objeto paralelo (*p-objeto*) é definido por um conjunto de unidades, cada uma localizada em uma unidade de processamento de um computador paralelo de memória distribuída. Um *p-objeto* é um objeto no sentido puro da programação orientada a objetos, realizando algum interesse da aplicação e comunicando-se com outros objetos através de invocação de métodos. *P-objetos* distintos de uma aplicação podem estar localizados em subconjuntos distintos das unidades de processamento do computador paralelo, sobrepostos ou disjuntos.

O estado de um *p-objeto* (*estado global*) é definido por um conjunto cujos elementos são os estados de cada uma de suas unidades (*estados locais*). Estados locais são definidos exatamente como o estado de objetos simples (Seção 2.1).

Um *p-objeto* pode definir métodos paralelos e métodos de unidade. Métodos de unidade são aceitos por unidades individuais do *p-objeto*. Um método paralelo é aceito por um subconjunto das unidades do *p-objeto*. Sejam A e B *p-objetos* tais que as unidades de A, o *chamador*, executam uma invocação de um método paralelo m de B, o *chamado*. Cada unidade chamadora de A deve estar localizada no mesma unidade de processamento da unidade chamada de B. Em invocações de métodos paralelos, sincronização e comunicação entre unidades de um *p-objeto* chamado acontece por passagem-de-mensagens.

A Figura 3.2 ilustra a invocação de métodos paralelos (setas contínuas) e invocação de métodos de unidade (setas tracejadas). As chamadas ***um1***, de **A** para **B**, e ***um2***, ***um3***, e ***um4***, de **C** para **D**, ilustram invocações de métodos de unidade. O *p-objeto* **A** executa chamadas dos métodos paralelos ***pm1*** e ***pm2***, respectivamente aceitos pelos *p-objetos* **B** e **C**. Note que ambos, **B** e **C**, estão localizados em um subconjunto dos unidades de processamento onde **A** está localizado, de forma que



}

Figura 3.3: Classe Paralela

os pares de unidades envolvidos em uma chamada de método (a_2/b_1, a_2/c_2, a_3/b_2 e a_4/c_3) estão na mesma unidade de processamento. Assim, invocação de métodos são sempre locais, envolvendo unidades de *p-objetos* distintos que estejam na mesma unidade de processamento. Essa é a comunicação inter-objeto, que torna possível a coordenação entre os *p-objetos* para alcançar os interesses da aplicação. Por outro lado, comunicação entre processos está sempre encapsulada em um *p-objeto*, utilizando-se de passagem-de-mensagens entre as unidades. Esta é comunicação intra-objeto, que implementa padrões de comunicação entre processos em programas paralelos, os quais são ortogonais aos interesses do programa.

3.2 Classes de Objetos Paralelos

Uma classe de objetos paralelos (*p-classe*) representa um protótipo de um conjunto de *p-objetos* compostos por conjuntos comuns de unidades, métodos e atributos, distinguidos somente pelos seus estados de execução. A Figura 3.3 ilustra a estrutura de uma classe paralela, introduzindo a abstração de unidade e os escopos sintáticos possíveis em uma declaração de uma *p-classe*: *escopo da classe* e *escopo da unidade*. A Figura 3.4 exemplifica a sintaxe da linguagem para *p-classes*.

Unidades de uma *p-classe* podem ser *unidades singulares* ou *unidades paralelas*. Na instanciação de uma *p-classe*, somente uma instância de uma unidade singular irá ser executada em uma unidade de processamento, enquanto que um número arbitrário de instâncias de unidades paralelas pode ser executada, cada uma em

uma unidade de processamento distinto.

Um atributo pode ser declarado no escopo da unidade ou no escopo da classe. No primeiro caso, é chamado de *atributo de unidade*, cuja instância deve existir somente no espaço de endereçamento da unidade onde está sendo declarado. No segundo caso, é chamado de *atributo de classe*, que deve ter uma instância independente no espaço de endereçamento de cada unidade do *p-objeto*.

Métodos podem ser declarados no escopo da classe ou no escopo da unidade. No primeiro caso, constituem *métodos paralelos de classe*, que são aceitos por todas as unidades de um *p-objeto*. No outro caso, são *métodos de unidade singular*, aceitos por unidades individuais, ou *métodos paralelos de unidade*, aceitos por unidades paralelas. Métodos paralelos de unidade podem ser declarados somente no escopo de unidades paralelas, utilizando o modificador *parallel*. Declarações de métodos paralelos de classe e de métodos paralelos de unidades correspondem aos métodos paralelos de *p-objetos*, mencionados na seção anterior.

A implementação de um método paralelo de classe deve ser efetuada no escopo de cada unidade da *p-classe*, possivelmente distinta. Alternativamente, uma implementação padrão pode ser descrita no escopo da classe, a qual pode ser sobreposta pelas implementações específicas providas no escopo de uma ou mais unidades. Implementações padrão de métodos paralelos de classe possuem acesso somente aos atributos de classe, enquanto que métodos declarados no escopo de unidades, singulares ou paralelas, podem acessar atributos de classe e de unidade.

Na Figura 3.4(a), a *p-classe Hello1* declara quatro unidades chamadas a, b, c, e d. As primeiras duas são unidades singulares, ao passo que as duas últimas são unidades paralelas. Note que o uso da palavra-chave *parallel* para declarar unidades paralelas. *Hello1* possui um método paralelo de classe, chamado *sayHello*, sem implementação padrão. Portanto, deve estar implementado em cada unidade. Existe também um exemplo de método paralelo de unidade, chamado *sayBye*, declarado somente pela unidade d.

A *p-classe Hello2*, na Figura 3.4(b), tem a intenção de ilustrar atributos e métodos declarados em escopos de classe e de unidade. Por exemplo, uma cópia do atributo de classe i existe em cada unidade a, b, e c. São variáveis independentes, e podem ser acessadas e atualizadas localmente. Note que um atributo de unidade do tipo ponto flutuante de dupla precisão $n1$ é declarado em cada escopo de unidade. Tais declarações não possuem o mesmo efeito ao declarar $n1$ como um atributo de classe, pois $n1$ não pode ser acessado pelos métodos de classe se declarado no escopo

das unidades. $n2$ é outro atributo de unidade, mas acessível somente no escopo da unidade b. O método paralelo de classe *sayHello* agora possui uma implementação padrão, definida fora da classe, como recomendado por convenções de programação em C++, embora a sintaxe de C++ permita também a definição da implementação dentro do escopo da classe. A implementação padrão de *sayHello* é sobrecarregada por implementações especializadas nas unidades b e c, também definidas fora da declaração da classe. De fato, somente a unidade a irá executar a implementação padrão de *sayHello*. Por fim, o método *getMy_i* é um método de unidade singular no escopo de c. Desse modo, ele possui acesso ao atributo de classe i e ao atributo de unidade $n1$.

3.2.1 Herança

Herança entre *p-classes* em PObC++ é semelhante a herança entre classes C++. A única peculiaridade é o requerimento de *preservação das unidades*, restringindo que as unidades de uma subclasse são exatamente aquelas herdadas e da superclasse. A herança entre *p-classes* é ilustrada na Figura 3.5.

Adicionar ou remover unidades de uma superclasse pode violar o princípio de substituição de sistemas tipos que possuem subtipagem, o qual garante que o estado de um objeto de uma classe pode ser utilizado em qualquer contexto onde um objeto de uma de suas superclasses é esperado. Por exemplo, seja A uma *p-classe* com n unidades distintas e A' seja outra *p-classe* que herda de A introduzindo uma unidade adicional que seja distinta das anteriores. Assim, A' possui $n + 1$ unidades distintas. Agora, suponha que a *p-classe* B, possuindo n unidades distintas, declare um atributo de classe v do tipo A. Na execução, um *p-objeto* de B instancia um *p-objeto* de A através do atributo v, utilizando a chamada do construtor de cada unidade de A em cada uma das n unidades de B. Agora, suponha a substituição do *p-objeto* A pelo *p-objeto* A' através do atributo v. De acordo com o princípio de substituição de tipos, isso seria uma operação segura, uma vez que A' é um subtipo de A, como consequência da herança. Entretanto, isso não é possível no cenário descrito, já que o *p-objeto* A possui $n + 1$ unidades. Dessa forma, não é possível determinar uma unidade de processamento distinto do *p-objeto* B para alocar a unidade adicional.

3.2.2 P-Objetos Escaláveis e Unidades Paralelas

Escalabilidade é uma propriedade crucial a ser alcançada no desenvolvimento de algoritmos para implementação sobre um determinado computador paralelo [32].

```
class Hello1 {

 /* método paralelo de classe */
  public: void sayHello();

  unit a {
    /* implementação de método
       paralelo de classe */
    void sayHello() {
      cout << "Hello! I am unit a";
    }
  };

  unit b {
    /* implementação de método
       paralelo de classe */
    void sayHello() {
      cout << "Hello ! I am unit b";
    }
  };

  parallel unit c {
    /* implementação de método
       paralelo de classe */
    void sayHello()[Communicator comm] {
      int rank = comm.rank();
      cout << "Hello ! I am the "
           << rank << "-th unit c"
    }
  };

  parallel unit d {

    /* implementação de método
       paralelo de classe */
    void sayHello() {
      int rank = comm.rank();
      cout << "Hello ! I am the "
           << rank << "-th unit d";
    }
    /* implementação de método
       de unidade */
    parallel void
      sayBye()[Communicator comm] {
        int rank = comm.rank();
        cout << "Bye ! I am the "
             << rank << "-th unit d";
      }
 };

};
```

(a)

```
class Hello2 {

  // atributo de classe
  private: int i;

  /* método paralelo com implementação
     padrão fora da classe */
  public:
  void sayHello();

  unit a {
    // atributo de unidade
    public: double n1;
  };

  unit b {
    // atributos de unidade
    private: double n2;
    public: double n1;
  };

  unit c {
    public:
    // atributo de unidade
    double n1;
    /* método de unidade singular */
    int getMy_i() {
      return i++;
    }
  };
};

/* implementação de método
   paralelo padrão */
void Hello2::sayHello() {
    cout << "I am some unit of Hello";
}

/* implementações de
   métodos paralelos */
void Hello2::b::sayHello() {
    cout << "Hello ! I am unit b";
}

void Hello2::c::sayHello() {
    cout << "Hello ! I am unit c";
}
```

(b)

Figura 3.4: Exemplo Simples de *p-classes*

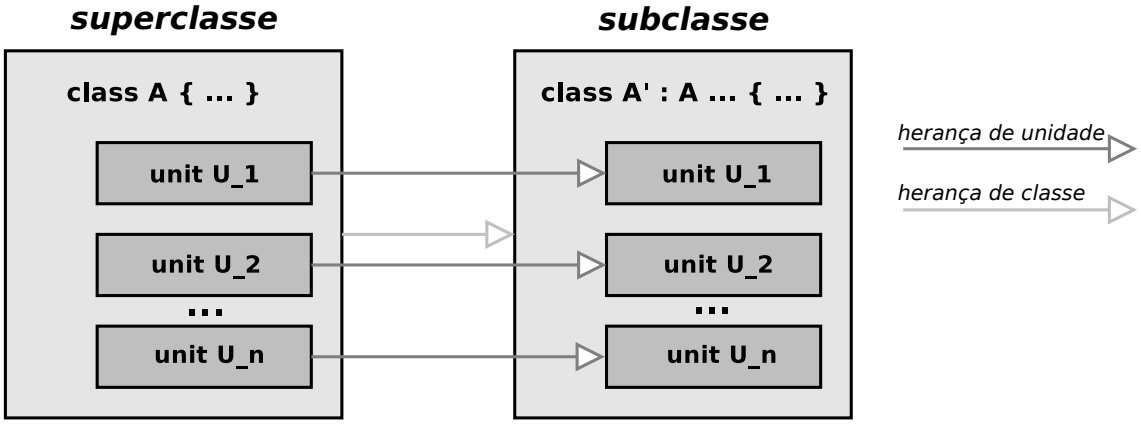


Figura 3.5: Herança Entre *p-classes*

A sintaxe de uma linguagem de programação paralela suporta programas paralelos escaláveis se possibilita a escrita de programas paralelos que possam executar em um número arbitrário de unidades de processamento sem recompilação. Na maioria das plataformas de programação paralela, um programa paralelo escalável é composto por um grupo de processos que executam o mesmo programa, identificados na forma de números inteiros, como os *ranks* do MPI, tornando possível distingui-los em um certo grupo homogêneo de processos. Esse é o estilo SPMD (*Single Program Multiple Data*) suportado pelas bibliotecas que implementam o modelo MPI (Seção 2.2.2). O número de processos deve ser definido exatamente antes da execução. Por essa razão, artefatos de programação paralela devem prover maneiras de especificar um número arbitrário de processos.

Como exemplificado anteriormente, unidades podem ser declaradas como unidades paralelas, usando o modificador *parallel*, de modo a ser possível descrever *p-objetos* escaláveis em POb$C{+}{+}$. Durante a execução de um método paralelo de unidade, a identificação de uma unidade simples e o número de unidades instanciadas para a unidades paralelas, assim como informações sobre organização topológica de tais unidades, podem ser descobertas pela invocação dos métodos de um ou mais *comunicadores* que podem ser fornecidos pelo chamador, como ilustrado no exemplo introduzido na Seção 3.2.3. Isso também é valido para métodos paralelos de classe.

Um *comunicador* é um tipo especial de objeto cuja interface contém métodos para comunicação e sincronização entre unidades de um *p-objeto* na execução de métodos paralelos. Detalhes sobre a semântica e o uso de comunicadores são apresentados na Seção 3.3.

```
class MatrixMultiplier {
   public: void distribute();
           int* collect();

   unit manager {
      private:
         int *a, *b, *c, n;
      public:
         void set_ab(int n_, int *a_, int *b_)
         { n = n_; a = a_; b = b_; }
   };

   parallel unit cell {
      private: int a = 0, b = 0, c = 0;
               int i, j, n;
               void calculate_ranks_neighbors
                     (CartesianCommunicator,
                      int, int, int*, int*,
                      int*, int*);
      public: parallel int* compute();
   };
};

class Main {
  public:
    int main();

  private:
    Communicator comm_data;
    CartesianCommunicator create_comm_compute();

  unit root {
     int main()[Communicator world_comm] {
        MatrixMultiplier::manager *mm
           = new MatrixMultiplier::manager();

        comm_data = world_comm.clone();
        create_comm_compute();

        mm->distribute[[comm_data]]();
        c = mm->collect[[comm_data]]();
     }
  };

  parallel unit peer {
     private:
        CartesianCommunicator comm_compute;

     int main()[Communicator world_comm] {
        MatrixMultiplier::cell *mm
           = new MatrixMultiplier::cell();

        comm_data = world_comm.clone();
        comm_compute = create_comm_compute();

        mm->distribute[[comm_data]]();
        mm->compute[[comm_compute]]();
        mm->collect[[comm_data]]();
     }
  };
};
```

(a)

```
void MatrixMultiplier::manager::distribute()
           [Communicator comm] {
   int fool_a, fool_b;
   comm.scatter(a-1, &fool_a,
                rankof(manager));
   comm.scatter(b-1, &fool_b,
                rankof(manager));
}

int* MatrixMultiplier::manager::collect()
           [Communicator comm] {
   comm.gather(-1, c, rankof(manager));
   return c + 1;
}

void MatrixMultiplier::cell::distribute()
           [Communicator comm] {
   comm.scatter(&a, rankof(manager));
   comm.scatter(&b, rankof(manager));
}

int* MatrixMultiplier::cell::collect()
           [Communicator comm] {
   comm.gather(&c, 1, rankof(manager));
   return &c;
}

void MatrixMultiplier::cell::compute()
            [CartesianCommunicator comm] {
   int west_1_rank, east_1_rank,
       north_1_rank, south_1_rank;
   int west_n_rank, east_n_rank,
       north_n_rank, south_n_rank;

   int i = comm.coordinates[0];
   int j = comm.coordinates[1];

   calculate_ranks_neighbors(comm, i, j,
           &west_n_rank, &east_n_rank,
           &north_n_rank, &south_n_rank);
   // initial alignment
   comm.isend(west_n_rank,0,&a,1);
   comm.isend(north_n_rank,0,&b,1);
   comm.recv(east_n_rank,0,&a,1);
   comm.recv(south_n_rank,0,&b,1);

   calculate_ranks_neighbors(comm, 1, 1,
           &west_1_rank, &east_1_rank,
           &north_1_rank, &south_1_rank);
   // start systolic calculation
   c += a * b;
   for (k=0; k < n-1; k++) {
     comm.isend(east_1_rank,0,&a,1);
     comm.isend(south_1_rank,0,&b,1);
     comm.recv(west_1_rank,0,&a,1);
     comm.recv(north_1_rank,0,&b,1);
      c += a * b;
   }
}
```

(b)

Figura 3.6: Exemplo Multiplicação de Matrizes

3.2.3 Um Exemplo Simples de Uma Classe Paralela

Na Figura 3.6(a), *p-objetos* da *p-classe MatrixMultiplier* possuem uma unidade nomeada *manager* e uma unidade paralela denominada *cell*, representando um grupo cooperativo de unidades organizadas em uma topologia de malha retangular bidimensional.

Através da invocação do método paralelo de classe *distribute*, a unidade *manager* distribui os elementos das duas matrizes quadradas ($n \times n$), chamadas de a e b, entre as unidades paralelas *cell*. Cada unidade de *cell* recebe um elemento de cada matriz. Após receber seus elementos correspondentes das matrizes de entrada, as unidades de *cell* executam um realinhamento dos elementos, requeridos pelo algoritmo sistólico de multiplicação de matrizes. Então, cada unidade de *cell* calcula seu elemento correspondente da matriz quadrada c, como consequência da invocação do método paralelo de unidade *compute*. Note que *compute* espera um comunicador cartesiano, já que unidades *cell* se comunicam em uma topologia de uma malha retangular bidimensional. Finalmente, a unidade *manager* deve coletar os resultados calculados por cada unidade *cell* pela invocação do método paralelo de classe denominado *collect*.

Essa é uma implementação simples de um algoritmo sistólico bem conhecido para uma multiplicação de matrizes [28], o qual será bastante útil nas discussão seguintes sobre comunicação e sincronização através de comunicadores.

3.3 Comunicação e Sincronização

Em OOPP, a ortogonalização entre *interesses* e processos, onde os primeiros estão agora encapsulados em *p-objetos*, resulta em uma separação clara entre dois tipos de mensagens:

- **comunicação inter-objetos**: mensagens trocadas entre objetos paralelos, implementando a orquestração dos interesses do software paralelo, concretamente realizados pelos *p-objetos*, a fim de implementar o interesse completo do software. Em geral, tais mensagens são realizadas por invocações de métodos, definindo uma relação cliente-servidor entre *p-objetos*;

- **comunicação intra-objeto**: mensagens trocadas entre as unidades de um *p-objeto*, normalmente por passagem-de-mensagens, definindo relações par-a-par entre unidades de um *p-objeto*. Tais mensagens definem as interações entre os processos da aplicação, requerido pelos algoritmos paralelos.

Em abordagens usuais de linguagens paralelas orientadas a objetos, não existe distinção clara entre esses tipos de mensagens. Como consequência, uma das abordagens seguintes é adotada:

- Comunicação e sincronização paralela é implementada pelo uso de invocação de métodos entre objetos, o que é inapropriado para programação paralela, uma vez que invocações de métodos conduzem a relações cliente-servidor entre pares de processos, ou pares de subconjuntos de processos. Porém, a maioria dos algoritmos paralelos assumem relacionamentos par-a-par entre eles;
- Passagem-de-mensagens entre objetos através de interfaces de baixo nível, as quais definem interações clandestinas entre objetos, resultando em baixa modularidade e alto grau de acoplamento entre os objetos que implementam um interesse paralelo.

A primeira abordagem é a principal fonte de dificuldades na abordagem ao problema do acoplamento $M \times N$ [8], onde dois conjuntos de N e M processos, possivelmente residindo em computadores paralelos distintos, desejam compartilhar alguma estrutura de dados cuja distribuição pode diferir nos dois conjuntos. Usando OOPP, tal acoplamento pode ser implementado por um *p-objeto*, cujas $M + N$ unidades são espalhadas entre todos os processos. Na segunda abordagem, objetos tendem a um grau menor de independência funcional. Assim, eles não podem ser analisados isoladamente, quebrando princípios importantes de modularidade por trás da orientação a objetos, discutidos na Seção 2.1.

3.3.1 Comunicação intra-objetos: Passagem-de-mensagens entre Unidades

A comunicação intra-objeto, entre unidades de um *p-objeto*, é implementada através de passagem-de-mensagens por meio de **comunicadores**, tal como no MPI. Em PObC++, um comunicador é um *p-objeto* primitivo, cujos métodos possibilitam a comunicação entre as unidades de um *p-objeto* na execução de métodos paralelos. A idéia de tratar comunicadores como objetos vem de interfaces orientadas a objetos do MPI, tais como MPI.NET [25] e Boost.MPI [21].

Seja A um *p-objeto*. Seja B um *p-objeto* que possui uma referência a A. A foi instanciado por B ou uma referência para A foi passada em uma chamada para algum método paralelo de B. Assim, B pode executar invocações a métodos paralelos de A. Em cada invocação paralela, B deve prover um comunicador, cuja

instanciação deve ser feita por B, usando a API do PObC++, ou recebido de outro *p-objeto*. Comunicadores podem ser reutilizados em invocações paralelas distintas, possivelmente aplicadas a métodos paralelos distintos de *p-objetos* também distintos.

A criação de comunicadores está ilustrada na Figura 3.6, onde dois comunicadores são criados: *comm_data*, pelas invocações dos métodos paralelos de classe *distribute* e *collect*; e *comm_compute*, por invocação do método paralelo de unidade *compute*. O comunicador *comm_data* agrupa todos os processos (unidades da *p-classe Main*). Por essa razão, é criado pela clonagem do comunicador global (*world_comm*), que é automaticamente passado para a função *main* de cada processo. O comunicador *comm_compute* é um *comunicador cartesiano*, criado em uma invocação ao método privado paralelo de classe *create_comm_compute*, cujo código é apresentado na Figura 3.7.

Assim como no MPI, um comunicador é sempre criado a partir de outro comunicador em uma operação coletiva envolvendo todos os membros do comunicador original. Por essa razão, *create_comm_compute* deve ser também executado pela unidade *root*. Note que um comunicador não é explicitamente passado no método *create_comm_compute*. Nesse caso, o comunicador do método paralelo que encerra (*world_comm*) é implicitamente passado para *create_comm_compute*, porém renomeado para *my_comm* no seu escopo.

A declaração implícita e passagem de comunicadores são apenas açúcar sintático [2] para simplificar o uso de comunicadores em programas paralelos simples. Eles são a razão do uso de colchetes para delimitar a declaração de comunicadores. De fato, existem duas situações onde é útil declarar explicitamente um identificador de comunicador:

- O método paralelo recebe mais de um comunicador;
- O programador faz uso de um identificador distinto para referenciar o comunicador, tal como no exemplo da Figura 3.6, onde *world_comm* é utilizado no lugar de *comm*.

Um exemplo e uma discussão sobre declaração implícita e passagem de comunicadores em métodos paralelos é apresentado na Seção 4.1.3, também justificando a notação usada com colchetes para delimitar declaração de comunicadores.

[2] Termo que caracteriza uma construção sintática que foi projetada para tornar mais fácil ler ou expressar algo.

Os operadores *rankof* e *ranksof* Nas Figuras 3.6 e 3.7, os operadores *rankof* e *ranksof* são utilizados para determinar o *rank* de unidades em um comunicador de um método paralelo de classe. O operador *rankof* deve ser aplicado ao identificador do tipo de uma unidade singular do *p-objeto*, retornando um inteiro que representa o *rank* da unidade no comunicador do método paralelo de classe. Consequentemente, *ranksof* deve ser aplicado ao identificador de unidades paralelas, retornando um vetor de inteiros. Eles são úteis e permitidos somente no código de métodos paralelos de classe de *p-objetos* que possuem unidades distintas. Na primeira chamada a *rankof* ou *ranksof*, comunicações acontecem para determinar os *ranks* das unidades do *p-objeto* no comunicador em uso. Por essa razão, na primeira chamada a um método paralelo de classe todas unidades envolvidas devem executar *rankof*/*ranksof* coletivamente. Na chamada subsequente, tal sincronização não é necessária, pois as unidades lembram os *ranks* previamente calculados. Além disso, pode-se especificar que comunicador deve ser utilizado para descoberta de *ranks*.

Assim como no MPI, comunicadores carregam informações topológicas sobre as unidades de um *p-objeto*. No caso padrão, unidades são organizadas linearmente, com *ranks* definidos por inteiros consecutivos de 0 a $M-1$, onde M é o número de processos. Alternativamente, unidades podem ser organizadas de acordo com uma *topologia cartesiana* em N dimensões, tendo P_i unidades em cada dimensão, para i de 0 a $N-1$. A topologia mais geral é a *topologia de grafos*, onde cada unidade é associada a um conjunto adjacente de unidades. Em geral, topologias de unidades de *p-objetos* seguem a estrutura de comunicação do algoritmo que seus métodos implementam. Tal informação pode ser útil para o sistema de execução fazer um melhor mapeamento de unidades às unidades de processamento da plataforma de computação paralela, na tentativa de balancear a carga das computações e minimizar custos de comunicação. No exemplo da Figura 3.6, o comunicador *comm_compute* possui uma topologia cartesiana com duas dimensões e ligações de arrodeio (*wraparounds*), onde *comm_data* possui uma topologia padrão linear.

As operações de comunicação suportadas por um comunicador são da mesma natureza daquelas que são suportadas por objetos comunicadores no Boost.MPI e no MPI.NET. Comunicadores também suportam operações para identificar unidades e suas localizações relativas na topologia do comunicador em questão. Essas operações são dependentes da topologia descrita pelo comunicador, como descrito a seguir:

- **Topologia Linear:** A operação *size* retorna o número de processos no grupo do comunicador; a operação *rank* retorna a identificação do processo, cujo

```
CartesianCommunicator
   Main:root::create_comm_compute()
           [Communicator my_comm]
{
   Group group_all = my_comm.group();
   Group group_peers
     = group_all.exclude(size_rank, rankof(root));
   my_comm.create(group_peers);

   /* o comunicador retornado é
      um comunicador nulo, pois root
      não está no group_peers  */

   return null;
}
```

(a)

```
CartesianCommunicator
   Main:peer::create_comm_compute() {
   Group group_all = comm.group();
   Group group_peers
     = group_all.exclude(rankof(root));
   Communicator comm_peers
     = comm.create(group_peers);

   int size = comm_peers.size();
   int dim_size = sqrt(size);
   int dims[] = {dim_size, dim_size};
   bool periods[] = {true, true};

   return
      new CartesianCommunicator
         (comm_peers, 2, dims,
                  periods, false);
}
```

(b)

Figura 3.7: Criação de comunicador

valor é um inteiro na faixa de 0 a $size - 1$;

- **Topologia de Grafo :** As operações *rank* e *size* possuem o mesmo significado como na topologia linear. Adicionalmente, a operação *neighbors* retorna um vetor de inteiros contendo os *ranks* das unidades adjacentes da unidade corrente. Similarmente, se a unidade corrente necessita saber os *ranks* das unidades adjacentes de outra unidade, usando seu *rank*, então é possível utilizar a operação *neighborsOf*. A operação *num_edges* retorna o número de arestas do grafo de unidades. Finalmente, a operação *edges* retorna uma matriz de adjacências representando o grafo.

- **Topologia Cartesiana:** As operações *rank*, *size*, *neighbors* e *num_edges* são também suportadas por esse tipo de comunicador, pois uma topologia cartesiana é um caso especial de topologia em grafo. Adicionalmente, a operação *dimensions* retorna um vetor de inteiros contendo o tamanho de cada dimensão, onde a operação *coordinates* retorna a coordenada da unidade em cada dimensão. O número de dimensões é a largura desses vetores. A operação *periodic* retorna um vetor de valores booleanos que dizem se uma dimensão possui aresta de arrodeio ou não. Finalmente, existem as operações *getCartesianRank*, que retorna o *rank* de uma unidade em um certo conjunto de coordenadas, e *getCartesianCoordinates*, que retorna as coordenadas de uma unidade com um dado *rank*. No exemplo da Figura 3.6, o método paralelo *compute* chama duas vezes o método de unidade *calculate_ranks_neighbors* para calcular os ranks dos quatro vizinhos da célula corrente na malha bidimensional no eixo x (oeste-leste) e no eixo y (norte-sul). O código de *calculate_ranks_neighbors* é mostrado na Figura 3.8. As coordenadas i e j da célula corrente são determinadas pela chamada ao método *coordinates*. Após o cálculo das coordenadas das células vizinhas, seus *ranks* são obtidos pela chamada ao método *getCartesianRank*.

Em OOPP, objetos não podem ser transmitidos através de comunicadores, mas apenas valores comuns de tipos de dados, primitivos e estruturados. Essa decisão não possui a intenção de meramente simplificar o esforço de implementação, pois a biblioteca Boost.MPI já oferece suporte para empacotamento e transmissão de objetos. De fato, a comunicação de objetos é uma fonte de problemas para a predição de tempo de execução, devido aos requisitos computacionais de empacotamento/desempacotamento de objetos (serialização). A predição de

```
void MatrixMultiplier::peer::calculate_ranks_neighbors
        (CartesianCommunicator& comm, int shift_x, int shift_y
         int* west_rank, int* east_rank, int* north_rank, int* south_rank)
{
   int dim_size_z = comm.dimensions(0);
   int dim_size_y = comm.dimensions(1);

   int i = comm.coordinates(0);
   int j = comm.coordinates(1);

   int west_coords[] = {(i-shift_x) % dim_size_x, j};
   int east_coords[] = {(i+shift_x) % dim_size_x, j};
   int north_coords[] = {i, (j-shift_y) % dim_size_y};
   int south_coords[] = {i, (j+shift_y) % dim_size_y};

   *west_rank = comm.cartesianRank(west_coords);
   *east_rank = comm.cartesianRank(east_coords);
   *north_rank = comm.cartesianRank(north_coords);
   *south_rank = comm.cartesianRank(south_coords);
}
```

Figura 3.8: Cálculo dos vizinhos

desempenho é um requisito importante em aplicações de CAD. Essa restrição permite otimizações de comunicação de primitivas ao trabalhar somente com valores de dados comuns (*raw data*).

A restrição descrita acima não é tão restritiva para a programação, uma vez que a transmissão de objetos pode ser facilmente implementada pelo empacotamento do estado do objeto em um *buffer*, enviando-o por passagem-de-mensagens para o destino, e desempacotando o estado enviado no objeto alvo. Entretanto, avalia-se ainda a possibilidade de introduzir primitivas para *migração* e *clonagem remota* de *p-objetos* para os casos de uso comum onde a transmissão de objetos é necessária. A migração é uma variação da clonagem remota, mas que a referência ao objeto no espaço de endereçamento original é perdida após a migração.

3.3.2 Comunicação Inter-objetos: Métodos Paralelos

Mensagens inter-objetos são implementadas como invocação de métodos de dois tipos (Seção 3.1):

- *Métodos de unidades*, definidos no escopo de uma unidade, com nenhum acesso a um comunicador.
- *Métodos paralelos*, declarados no escopo da classe (*método paralelo de classe*) ou no escopo de uma unidade (*método paralelo de unidade*);

Como descrito na Figura 3.2, uma *invocação de método paralelo* é realizado por um conjunto de *invocações locais de métodos* entre pares de unidades dos *p-objetos*

chamadores e dos chamados, localizados nas mesmos unidades de processamento. Consequentemente, não existe invocação remota de métodos. Métodos paralelos são os únicos pontos de comunicação e sincronização do programa paralelo, envolvendo unidades que residem em unidades de processamento distintas. É conveniente que as chamadas a métodos paralelos executados por unidades de um *p-objeto* completem-se conjuntamente, evitando sincronização excessiva entre unidades. Portanto, o número de chamadas ao mesmo método paralelo em cada unidade deve ser igual. Obviamente, tal restrição não pode ser forçada estaticamente. O programador deve garantir a coerência do uso de métodos paralelos, tal como na programação usando MPI. A flexibilidade de deixar a sincronização entre métodos paralelos ser explicitamente controlada pelo programador é vista como uma interessante oportunidade para investigar técnicas não-triviais de sincronização paralela.

A comunicação inter-objetos por invocação de métodos faz desnecessária a introdução do conceito de *intercomunicadores*, suportado pelo MPI, no PObC++. De acordo com o padrão MPI, intercomunicadores tornam possível a troca de mensagens entre grupos de processos em intracomunicadores locais disjuntos.

3.3.3 Um Exemplo de Comunicação e Sincronização

No exemplo da Figura 3.6, os métodos paralelos *distribute*, *collect* e *compute* executam operações de comunicação. Eles exemplificam operações de comunicação coletiva e ponto-a-ponto através de comunicadores. As duas primeiras operações, *scatter* e *gather*, executam padrões coletivos de comunicação para difusão e agregação de dados, respectivamente entre a unidade *manager* e as unidades *worker*, para a distribuição da entrada entre as unidades *cell* e para a agregação dos resultados calculados. Em *compute*, operações de comunicação ponto-a-ponto (*isend* e *recv*) são executadas para implementar a computação sistólica. Antes do laço principal, o realinhamento dos elementos das matrizes de entrada (a e b) é executado. Cada unidade *cell* comunica-se com suas unidades adjacentes na malha quadrada, nas direções esquerda, direita, acima e abaixo.

3.4 Instanciação

Um *p-objeto* é instanciado por outro *p-objeto* pela instanciação coletiva de cada uma de suas unidades em unidades de processamento diferentes, usando o operador *new* em C++, utilizando a identificação completa da unidade, com a forma: ⟨*classe*⟩::⟨*unidade*⟩. Isso é ilustrado no código da *p-classe Main* da Figura 3.6, onde as unidades *MatrixMultiplier::manager* e *MatrixMultiplier::cell* são instanciadas.

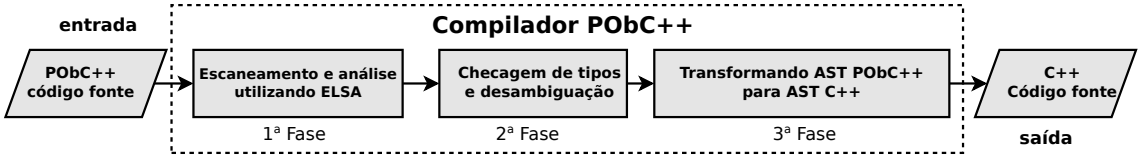


Figura 3.9: Fases do Compilador POb C++

Nenhuma comunicação acontece entre as unidades durante a instanciação. Como descrito anteriormente, comunicações ocorrem somente em invocações paralelas de métodos e o chamador é responsável em criar um comunicador apropriado e passar para o método paralelo que ele deseja invocar.

A instanciação de um *p-objeto* é uma operação indivisível. Entretanto, desde que não exista comunicação ou sincronização no processo de instanciação, unidades podem ser instanciadas em diferentes pontos de execução. O programador é responsável na garantia de que todas as unidades de um *p-objeto* estão propriamente instanciadas antes de invocar qualquer um dos seus métodos paralelos. É um erro de programação esquecer de instanciar uma ou mais unidades. Note que não há nenhum tipo de restrição ao número de unidades paralelas a ser instanciada sobre os unidades de processamento distintos. A identificação de cada unidade é definida pelo comunicador passado para cada método de invocação paralela e pode ser modificado entre invocações distintas.

3.5 Implementação

POb C++ é um projeto de código aberto hospedado em *http://pobcpp.googlecode.com*, liberado sob a licença BSD. O projeto é composto pelo compilador e pela sua biblioteca padrão.

3.5.1 Compilador

A fim de rapidamente produzir um protótipo de compilador confiável e rápido, um compilador *source-to-source* escrito em C++ foi desenvolvido pela modificação do ELSA, um analisador sintático C/C++ baseado no Elkhound. Elkhound [34] é um gerador de analisadores sintáticos que utiliza um algoritmo GLR, uma estensão ao conhecido algoritmo LR que trata gramáticas não-determinísticas e gramáticas ambíguas. Em particular, o algoritmo do Elkhound executa uma fase de desambiguação durante o processo de checagem de tipos a fim de gerar uma árvore sintática abstrata (AST[3]) válida. As fases do compilador da linguagem POb C++

[3]Sigla do inglês, *abstract syntax tree*.

estão descritas na Figura 3.9.

Na primeira fase, o compilador PObC++ recebe um código-fonte, realiza o escaneamento de símbolos e passa para a análise sintática, implementada pela modificação do ELSA. Depois, a segunda fase é executada, resolvendo possíveis ambiguidades através da checagem de tipos. Por fim, na terceira fase, acontece a transformação das ASTs PObC++ tipadas e sem ambiguiidades para ASTs C++. As modificações executadas durante a terceira fase transformam cada unidade de uma *p-classe* em uma classe C++ válida, incluindo informações adicionais para introspecção da *p-classe*. Essas transformações acontecem através de alterações no código de entrada.

Esses ajustes geram somente código C++ válido e não interferem com o resto do programa. De fato, somente o código contendo declarações de *p-classes* precisam ser compilados pelo compilador. Na figura 3.10, um exemplo simplificado de modificação produzida pelo compilador. Os atributos e métodos definidos na classe paralela *A* são passados as unidades *u1* e *u2*. As unidades definidas, *u1* e *u2*, tornaram-se classes com o identificador mantido que herda atributos e métodos da classe *Unit* do espaço de nomes *Pobcpp*, que é parte da biblioteca padrão. Além disso, métodos que possuam um comunicador como entrada são transformados em um método puramente C++ e as chamada a tais métodos são automaticamente completadas quando possível.

Códigos C++ podem ser utilizados sem modificações em programas desenvolvidos em PObC++. Assim, virtualmente qualquer biblioteca C/C++ pode ser integrada com programas em PObC++. Desde que o programador utilize o compilador de PObC++ para gerar C++, qualquer compilador pode ser usado para gerar código nativo de máquina. Tais aspectos são essenciais para promover uma integração de programas em PObC++ com bibliotecas cientificas e código legado escrito em C e C++. Essa é a motivação para o estudo de caso que será apresentado na Seção 4.2.

3.5.2 Biblioteca Padrão

A biblioteca padrão do PObC++ foi desenvolvida em conjunto com o compilador para facilitar a modificação de comunicadores, grupos e funções como *ranksof*. O compilador utiliza as declarações das classes e de seus métodos para gerar o código final em C++. Assim, é possível alterar a infraestrutura de comunicação utilizada, MPI nesse caso, por outra qualquer sem necessidade de alterar a linguagem e o

```
class A {
  public: int attribute;
  private: void par_func();
  unit u1 {
    private:
    void func1(int a) [Communicator c1] {
      int ranku2 = rankof(u2);
      func2(0);
    }
    void func2(int b) [Communicator c2] {
      func1(1);
    }
  };

  unit u2 {
  };
}
```

```
class A {
  class u1 : virtual public Pobcpp::Unit {
    public: int attribute;
    private: void par_func();
    private:
    void func1(int a, Communicator c1) {
      int ru2 = rankof<u2>(c1,
                           Unit_Type(this));
      func2(0, c1);
    }
    void func2(int b, Communicator c2) {
      func1(1, c2);
    }
  };
  class u2 : virtual public Pobcpp::Unit {
    public: int attribute;
    private: void par_func();
  };
}
```

Figura 3.10: Exemplo de geração de código

compilador. Para evitar ambiguidade, a biblioteca é definida dentro do escopo do espaço de nomes *Pobcpp*.

A implementação da biblioteca é simples e trabalha como uma interface para o MPI, através da Boost.MPI, na maioria das vezes. A chamada de mais funções para efetuar a mesma operação em C++ resulta em uma sobrecarga potencial. Entretanto, em geral, o tempo de comunicação domina o tempo das operações dos comunicadores em programas paralelos que justificam o uso de PObC++, tornando essa sobrecarga pouco significativa frente aos ganhos em termos de estruturação de software mesmo no contexto de computação de alto desempenho.

Capítulo 4

Estudos de Caso e Avaliação de Desempenho

Os estudos de caso apresentados neste capítulo procuram dar uma visão geral das técnicas de programação, expressividade e desempenho oferecidos pela linguagem PObC++ para desenvolvedores de programas paralelos.

O primeiro estudo de caso apresenta um integrador númerico paralelo que demonstra a técnica de programação paralela baseada em esqueletos. Além disso, apresenta uma avaliação de desempenho, comparando o programa em PObC++ com uma versão diretamente escrita em C++ e MPI. A avaliação de desempenho procura oferecer uma evidência de que o PObC++ não produz perda significativa no desempenho de um programa paralelo, usando a programação C++/MPI como referencial.

O segundo estudo de caso ilustra a integração do PObC++ com bibliotecas de computação científica, um importante requisito para a aceitação dessa linguagem entre programadores dos domínios da engenharia e das ciências computacionais. Por isso, foi desenvolvido uma interface OOPP para um subconjunto do PETSc, biblioteca de subrotinas amplamente utilizada, para solucionar sistemas algébricos em Fortran, C e C++. De acordo com seus desenvolvedores, o PETSc segue um modelo baseado em objetos, embora implementado sobre linguagens procedurais, como é o caso de Fortran e C. Por esse motivo, suporta matrizes, vetores e solucionadores como objetos, provendo uma interface para sua instanciação e gerenciamento. OOPP provê uma alternativa completamente orientada a objetos para o PETSc, fornecendo todos os benefícios da OOP para programadores PETSc.

O terceiro estudo de caso ilustra abstração e encapsulamento de interações

paralelas em *p-objetos*, além de uma avaliação de desempenho de uma implementação em C do algoritmo *bucketsort* desenvolvido pelo projeto *NAS Parallel Benchmarks* comparado a sua versão em PObC++.

Mais detalhes sobre casos de estudos são apresentados nas seções seguintes.

4.1 Integração Numérica Paralela

Um integrador númerico paralelo utilizando duas *p-classes*, *Farm* e *Integrator*, foi implementado. A primeira é uma classe abstrata, implementando o esqueleto *farm*, que implementa uma abstração para uma estratégia de paralelismo bastante difundida, incluindo seu padrão de comunicação inter-processos. A *p-classe Integrator* estende *Farm* com o intuito de implementar uma integração numérica paralela usando paralelismo baseado em *farm*. Foi reutilizado a implementação do algoritmo de integração numérica pela biblioteca NINTLIB, baseada no método de Romberg [9]. NINTLIB é uma implementação sequencial.

Programação baseada em Esqueletos O termo *esqueleto algorítmico* foi primeiramente cunhado por Murray Cole duas décadas atrás, para descrever o reuso de padrões de computação paralela cujas implementações podem ser otimizadas para plataformas específicas de computação paralela [13]. A programação baseada em esqueletos tem sido amplamente investigada pela comunidade acadêmica, sendo considerada uma abordagem promissora para programação paralela em alto nível [31]. Programação baseada em esqueletos é uma importante técnica da OOPP.

4.1.1 A p-classe *Farm* (Esqueleto)

A *p-classe Farm* é apresentada na Figura 4.1(a). Declara dois métodos paralelos, *synchronize_jobs* e *synchronize_results*, as quais são invocadas pelas unidades para transmitir os trabalhos, representados por *p-objetos* da *p-classe Job*, do gerenciador (unidade singular *manager*) para os trabalhadores (unidades paralelas *worker*), e os resultados dos trabalhos, representados por *p-objetos* da *p-classe Result*, calculados pelo método *work*, dos trabalhadores para o gerenciador. Apesar de *work* estar implementado em cada trabalhador, ainda é um método de unidade, e não um método paralelo, uma vez que, o trabalho de um trabalhador é um procedimento local em sua natureza, refletido pela ausência de comunicação entre os trabalhadores.

Os métodos *synchronize_jobs*, *synchronize_results*, e *perform_jobs* possuem implementações padrão na *p-classe Farm*. As implementações desses métodos são supostamente otimizadas para a plataforma alvo, liberando os programadores do

```
template<typename Job, typename Result>
class Farm {
   public:
   void synchronize_jobs();
   void synchronize_results();

   unit manager {
     private:
     Job* all_jobs;
     Result* all_results;

     public:
     void add_jobs(Job* job);
     Result get_next_result();
     Result* get_all_results();
     virtual void* pack_jobs(Job* jobs);
     virtual Result unpack_result(void*);
   };

   parallel unit worker {
     private:
     Job* local_jobs;
     Result* local_results;

     public:
     parallel void perform_jobs();
     virtual Result work(Job job);
     virtual Job unpack_jobs(void* jobs);
     virtual void* pack_result(Result* r);
    };
 };
```

(a)

```
class Integrator:
  public Farm<IntegratorJob,
              double> {
  unit manager {
    private:
      int inf, sup;
      int dim_num;
      int partition_size;

    public:
      Manager(int inf, int sup,
              int dim_num,
              int psize):
          inf(inf), sup(sup),
          dim_num(dim_num),
          partition_size(psize) { }

    public:
      void generate_subproblems();
      double combine_results();
  };

  parallel unit worker {
    private:
      int n_of_partitions;
      int next_unsolved_subp;
      double (*function)(double*);

    public:
      Worker(double (*f)(double*),
             int tol, int nop) :
          function(f),
          n_of_partitions(nop),
          next_unsolved_sub(0),
          tolerance(tol) { }
  };
 };
```

(b)

Figura 4.1: P-classe *Farm* (a) e p-classe *Integrator* (b)

conhecimento sobre detalhes da arquitetura paralela envolvida. De fato, programas paralelos especializados para outras arquiteturas que utilizem esse esqueleto podem se utilizar da *p-classe Farm*.

Os métodos *add_jobs*, *get_next_result* e *get_all_results* também possuem implementações padrão. *add_jobs* acrescenta trabalhos para a unidade singular *manager*, *get_next_result* retorna nulo se todos os resultados chegaram e *get_all_results* retorna todos os resultados obtidos. Esses métodos estão preparados para a concorrência entre os métodos *synchronize_jobs*, *synchronize_results*, e *perform_jobs*, cujas implementações foram desenvolvidas de forma segura para o uso de *threads*. Então, o usuário pode chamar os métodos em *threads* distintas para sobrepor computação e comunicação. Entretanto, de acordo com as características da plataforma paralela alvo, a implementação irá decidir quantos trabalhos deve ser recebidos por *add_jobs* antes de iniciar o envio das tarefas dos trabalhadores, bem como decidir quantos resultados devem ser calculados por um trabalhador antes do envio do resultado, ou um conjunto de resultados, para o gerenciador.

A fim de usar o esqueleto *farm*, o programador deve estender a *p-classe Farm*, utilizando herança. Então, é necessário implementar os métodos de unidade virtuais *pack_job*, *unpack_result*, *pack_result*, *unpack_job* e *work*. Os primeiros quatro métodos são necessários para empacotar/desempacotar o estado de trabalho/resultado para/de *buffer*, a fim de transmiti-los através do comunicador. Os tipos *Result* e *Job* devem ser definidos de acordo com a aplicação. Vale a pena ressaltar que objetos não podem ser transmitidos através de comunicadores, mas somente dados. Procedimentos de empacotamento e desempacotamento são necessários para manter a generalidade da *p-classe Farm*. O método *work* define a computação executada por cada trabalhador, cuja execução define a solução do problema. Por isso, é um método virtual em *Farm* e deve ser implementado pelas subclasses de *Farm* que implementam aplicações específicas.

4.1.2 A *p-classe Integrator*

A *p-classe Integrator*, como declarada no arquivo cabeçalho *integrator.h*, é apresentada na Figura 4.1(b). Além dos métodos virtuais herdados da *p-classe Farm*, é implementado também o método *generate_subproblems* do gerenciador, o qual particiona o intervalo de integração, gera objetos *Job*, e chama os métodos *add_jobs*, com o intuito de alimentar a lista de tarefas, e *combine_subproblems_results*, cuja função é chamar *get_next_result* ou

```
class IntegratorMain {
  public: int main();

  unit root {
    int main() {
      #pragma omp parallel
      {
      Integrator::manager *m
          = new Integrator::manager
                (0.0, 1.0, 5, 2);
      double res;
        #pragma omp sections
        {
        #pragma omp section
        m -> generate_subproblems();
        #pragma omp section
        m -> synchronize_jobs();
        #pragma omp section
        m -> synchronize_results();
        #pragma omp section
        res = m->combine_results();
        }
      cout >> "Result is ", res;
      }
    }
  };

(...)
```

```
(...)

  parallel unit peer {
    int main() {
      #pragma omp parallel
      {
        Integrator::worker *w
            = new Integrator::worker
                (function, 1D-5,16);
        #pragma omp sections
        {
            #pragma omp section
            w -> synchronize_jobs();
            #pragma omp section
            w -> perform_jobs();
            #pragma omp section
            w -> synchronize_results();
        }
      }
    }
  };
};
```

Figura 4.2: P-classe Main

get_all_results para capturar os resultados numéricos calculados pelos trabalhadores que serão adicionados.

4.1.3 Programa Principal

A classe principal do programa de integração está descrita na Figura 4.2. A implementação tenta explorar toda a concorrência disponível, criando várias *threads* utilizando OpenMP (como descrito na Seção 2.2.1). Esse é um exemplo de como paralelismo com múltiplas *threads* pode ser explorado dentro das unidades de um *p-objeto*.

O leitor pode ter notado que o comunicador não é passado pela invocação do método paralelo no código do *main*. Se um comunicador não é explicitamente utilizado, o comunicador da invocação paralela do chamador é implicitamente passado. No exemplo, tal comunicador é o comunicador global, que é recebido pelo método *main*. Isso somente é possível se o método paralelo onde acontece a invocação possui somente um comunicador, não havendo ambiguidade. Note que o identificador do comunicador do método *main* não é explicitamente informado. Em tais casos, o comunicador possui um identificador padrão chamado *comm*. De fato, existem duas situações onde é útil declarar explicitamente um identificador para um comunicador:

- O método paralelo recebe mais de um comunicador;
- O programador quer utilizar um identificador distinto para referenciar o comunicador, tal como no exemplo da Figura 3.6, onde *world_comm* foi usado no lugar de *comm*.

A declaração implícita e a passagem de comunicadores são apenas açúcares sintáticos para simplificar o uso de comunicadores em programas simples. Essa é a razão para o uso de colchetes para a delimitação da declaração de comunicadores. Programas simples que usam apenas o comunicador global não precisam declarar parâmetros de comunicador.

4.1.4 Avaliação de Desempenho

Um caso de teste é definido pelo programa, C++/MPI ou PObC++, pelo valor da dimensão (N) e pela quantidade de processadores P. A Tabela 4.1 resume os resultados obtidos pela execução do integrador numérico programado com PObC++, pela sua versão pura C++/MPI, no *cluster* CENAPAD-UFC[1], instalado no Departamento de Computação da Universidade Federal do Ceará, variando o número de processadores (P) e dimensões (N). As execuções que utilizam até 16 processos foram realizadas com processos em unidades de processamento distintas, mas as que utilizam acima de 32 processos utilizam vários processos na mesma unidade de processamento.

Os tempos de execução apresentados para cada par em $P \times N$, em segundos, constitutem a média das 40 execuções execuções com a remoção de no máximo 10

[1]CENAPAD-UFC possui 48 Blades Bull 500, onde cada Blade possui dois processadores Intel Westmere 6 cores X5650 EP, 6x4GB DDR3 1333MHz de memória RAM, e Infiniband QDR de 36 portas (*http://www.cenapad.ufc.br*). No momento do experimento, apenas 16 blades, cada qual com 12 núcleos, estavam disponíveis.

	número de dimensões(N)											
	6				7				8			
P	PObC++	C++	$+\delta$	$-\delta$	PObC++	C++	$+\delta$	$-\delta$	PObC++	C++	$+\delta$	$-\delta$
1	5,90(×0,9)	5,84(×0,9)	0,02	0,03	109,1(×0,9)	107,8(×0,9)	0,06	0,00	1971(×0,9)	1942(×0,9)	0,02	0,00
2	2,98(×1,8)	2,95(×1,8)	0,16	0,0	55,0(×1,8)	54,5(×1,9)	0,01	0,00	997(×1,9)	983(×1,9)	0,02	0,00
4	1,50(×3,6)	1,49(×3,6)	0,04	0,06	27,7(×3,7)	27,3(×3,8)	0,10	-0,07	500(×3,9)	493(×3,8)	0,02	0,00
8	0,76(×7,1)	0,76(×7,1)	0,04	0,13	13,8(×7,5)	13,9(×7,4)	0,03	0,09	252(×7,8)	247(×7,7)	0,14	0,00
16	0,38(×14,2)	0,39(×13,9)	0,05	0,13	6,9(×15,0)	7,0(×14,8)	0,02	0,06	125(×15,7)	124(×15,4)	0,09	0,01
32	0,20(×27,1)	0,20(×27,1)	0,15	0,15	3,6(×28,9)	3,5(×29,7)	0,15	0,18	64,9(×30,3)	63,4(×30,2)	0,14	0,12
seq	5,43s				104,1s				1921s			

Tabela 4.1: Sumário de resultado de desempenho (*Tempo de execução* e *Speedup*)

valores extremos (*outliers*) para alcançar intervalos de confiança a 90%. Após a remoção, analisamos os intervalos nos extremos e calculamos as diferenças relativas mínima e máxima em porcentagem, valores $-\delta$ e $+\delta$ na tabela, respectivamente.

Sejam $[x_1, x_2]$ e $[y_1, y_2]$ os intervalos para as versões PObC++ e C++/MPI, onde x_1 é o menor tempo encontrado para um caso de teste utilizando PObC++ e x_2 o maior tempo encontrado para o caso de teste utilizando C++/MPI e y_1 e y_2 calculado da mesma forma para o caso de teste com C++/MPI. Sendo $-\delta$ e $+\delta$ calculados da seguinte forma:

$$-\delta = ((x_2 - y_1)/y_1) \times 100 \qquad + \delta = ((y_2 - x_1)/x_1) \times 100$$

Esses valores estando muito próximos a $0,0\%$ indicam que não há diferença estatistica significante entre as amostras. O desvio padrão variou entre $0,07\%$ e $4,86\%$ para os experimentos utilizando PObC++ e entre $0,22\%$ e $4,84\%$ para C++/MPI.

A função integrada e o intervalo aplicado são

$$f(x_1, x_2, x_3, \cdots, x_n) = {x_1}^2 + {x_2}^2 + {x_3}^2 + \cdots + {x_n}^2$$

e $[0, 1]$, respectivamente. Na versão paralela, o parâmetro *dim_partition_size* define o número de partições do intervalo em cada dimensão, do mesmo tamanho, produzindo $num\_jobs = dim\_partition\_size^n$ subproblemas de integração. É necessário definir também o parâmetro de *number_of_partitions* (número de partições) requerido pelo procedimento *romberg_nd*, como um múltiplo de *num_jobs*, a fim de preservar a quantidade de computação executada pela versão sequencial. Arbitrariamente, nós definimos:

- *dim_partition_size* = 2
- *number_of_partitions* = 8
- n variando entre 6 e 8.

O experimento teve como objetivo coletar evidências empíricas sobre a equivalência de desempenho entre PObC++ e C++/MPI. Argumentamos que tal conclusão pode ser feita somente pela observação que o modelo de programação paralela das duas versões são equivalentes. De fato, a versão PObC++ é compilada para um programa C++/MPI que é virtualmente equivalente à versão C++/MPI,

```
class ParallelVec {
  parallel unit PVec {
    public:
    PVec();
    PetscErrorCode Create()
    [Communicator comm];
    PetscErrorCode CreateSeq(PetscInt)
    [Communicator comm];
    PetscErrorCode Destroy();
    private:
    Vec vec; // Petsc data
  };
};

PetscErrorCode ParallelVec::PVec::Create()
                    [Communicator comm] {
  return VecCreate(comm.get_mpi_comm(), &vec);
}
```

(a)

```
class ParallelKSP {
  parallel unit PKSP {
    public:
      PetscErrorCode Create() [Communicator comm];
      PetscErrorCode Solve(ParallelVec::PVec& b,
                           ParallelVec::PVec& x);
      PetscErrorCode SetOperators(ParallelMat::PMat& Amat,
                                  ParallelMat::PMat&
      /* Other methods: tolerance control,
         preconditioning, etc */
      (...)
    private:
      KSP ksp; // Petsc data
  };
};
```

(b)

Figura 4.3: P-classes *ParallelVec* (a) e *ParallelKSP* (b)

exceto por chamadas indiretas da interface do comunicador para as subrotinas da biblioteca MPI. A diferença essencial são as propriedades alcançadas pela orientação a objetos pelo PObC++, com ganhos potenciais em modularidade e abstração.

4.2 Interface OOPP para PETSc

PETSc é uma biblioteca para computação científica desenvolvida como um conjunto de estruturas de dados e subrotinas para solucionar sistemas algébricos lineares e não-lineares usando MPI. É amplamente utilizado por programadores científicos e de engenharia na solução numérica de equações diferenciais parciais (EDP), as quais descrevem fenômenos de seu interesse. PETSc segue um projeto baseado em objetos, porém sobre linguagens procedurais, tais como Fortran e C. Esta seção mostra uma interface OOPP para o PETSc, demonstrando o potencial de integração do PObC++ com bibliotecas de computação científica de

uso disseminado.

4.2.1 P-classes *ParallelVec* e *ParallelMat*

ParallelVec e *ParallelMat* são *p-classes* que representam as estruturas de dados *Vec* e *Mat* do PETSc, respectivamente. *Vec* representa um vetor distribuído através das unidades de processamento, enquanto *Mat* representa uma matriz distribuída. Na Figura 4.3(a), o código de *ParallelVec* é apresentado. O código de *ParallelMat* é analogamente definido.

ParallelVec e *ParallelMat* possuem unidades paralelas *pvec* e *pmat*, respectivamente. Vale ressaltar que o PETSc requer um comunicador MPI para a criação de vetores e matrizes. É algo frequentemente necessário em bibliotecas científicas paralelas que utilizam o MPI. Em PObC++, isso é suportado pela declaração do método paralelo *Create*. Então, um comunicador primitivo pode ser obtido pela chamado do método *get_mpi_comm* do comunicador *comm*, que pode ser passado para as sub-rotinas PETSc *VecCreate* e *MatCreate*. Essa é uma suposição válida, uma vez que PObC++ é implementado sobre o padrão MPI. Em *ParallelVec*, a chamada a *VecCreate* irá atribuir uma estrutura de dados *Vec* ao atributo privado *vec*, representando o vetor PETSc que guarda a partição local do vetor distribuído dentro do espaço de endereçamento da unidade.

4.2.2 A p-classe *ParallelKSP*

ParallelKSP declara unidades paralelas chamadas *ksp*, representando um solucionador paralelo de sistemas lineares usando o método de sub-espaço de Krylov. A Figura 4.3(b) esboça os elementos essenciais da interface de um *p-objeto* da *p-classe ParallelKSP*. Após a instanciação do solucionador, pela chamada ao construtor de *ParallelKSP*, o usuário deve executar o método *Create* para inicializar as estruturas de dados do solucionador através das unidades de processamento. Então, é necessário chamar *SetOperators* para prover os operadores de matrizes, *Amat* e *Pmat*, que são reusados através de subsequentes chamadas ao método *Solve*, onde os parâmetros do problema, vetores b e x, são informados. O atributo *ksp* é o solucionador propriamente dito do PETSc.

4.3 Ordenação Paralela

Esta seção apresenta uma *p-classe* abstrata chamada *Sort*, e duas *p-classes* derivadas que implementam os algoritmos de ordenação, *bucketsort* e *odd-even sort*. Essas *p-classes* são chamadas *BucketSort* e *OddEvenSort*, respectivamente, as quais

```
class Sort {
  public: void virtual sort() = 0;

  parallel unit worker {
    public:
      void set(int* items, int size);
    protected:
      int* items;
      int size;
    };
};
```
(a)

```
class BucketSort : Sort {
  public: void sort();

  parallel unit worker {
    private:
      void local_sort(int* items,
                      int size);
      void fill_buckets(int* key_array_1,
                        int* key_array_2,
                        int* bucket_ptr,
                        int* bucket_size)
  };
};
```
(b)

Figura 4.4: *Sort* (a) e *BucketSort* (b)

ilustram a abstração e o encapsulamento em OOPP, em relação a comunicação e sincronização entre as unidades de um *p-objeto*, uma vez que as sequências de comunicação entre unidades de dois ordenadores usam padrões de interação paralela distintos.

4.3.1 A p-classe *Sort*

O código fonte da *p-classe Sort* é apresentado na Figura 4.4(a), onde está declarado a função virtual paralela *sort*. Além disso, estão definidas unidades paralelas *worker* que declaram um vetor de inteiros a ser ordenado através dos trabalhadores, representado pelas variáveis *items* e *size*. Desse modo, qualquer especialização de *Sort* precisa implementar o método *sort* de acordo com o algoritmo escolhido.

```
void BucketSort::worker::sort() {
    /* Estruturas de dados principais */
    int* key_array = items; // vetor de chaves locais
    int* key_buff_send;    // vetor de chaves locais em seus baldes
    int* bucket_ptr;
    int* bucket_size;        // tamanho dos baldes
    int* key_buff_recv;    // vetor global de chaves

    /* Inicialização das estruturas */
    ...
    fill_buckets(key_array, key_buff_send, bucket_ptr, bucket_size);
    /* Determina tamanho global de cada balde */
    comm.allreduce(bucket_size, Operation<int>.Add, bucket_size_totals);
    /* Determina quantos itens locais serão enviados para cada processo */
    comm.alltoall(send_count, recv_count);
    /* Envia itens para cada processo */
    comm.alltoallflattened(key_buff_send,send_counts,key_buff_recv,outValues);
    /* Ordenação local */
    local_sort(key_buff_recv, size);
}
```

Figura 4.5: P-classe *BucketSort* (Método *sort*)

Note que *items* e *size* estão declarados como atributos de unidade, ao invés de atributos de classe. Isso acontece porque eles representam uma partição local do vetor a ser ordenado, os quais são inicialmente distribuídos através das unidades. No fim da execução, cada elemento do vetor *items* da unidade i é menor ou igual a qualquer elemento no vetor de itens da unidade j, desde que $i < j$. Em um projeto alternativo, *items* e *size* poderiam ser atributos de classe, onde é implicitamente suposto que todos os itens são conhecidos em todas as unidades, no começo e no fim do processo de ordenação, dando ao método de ordenação a responsabilidade de executar uma distribuição inicial dos itens através das unidades.

4.3.2 A p-classe *BucketSort*

Bucketsort é um algoritmo de ordenação eficiente $\Theta(n)$, desde que os itens estejam uniformemente distribuídos no vetor de entrada. Seja n o número de itens. Existe um conjunto $\{b_0, \ldots, b_{k-1}\}$ de baldes (*buckets*), onde $n >> k$. O algoritmo *bucketsort* possui os seguintes passos:

i. Os itens são distribuídos através dos conjunto de baldes, tal que $p \in b_i \wedge q \in b_j$ então $p < q$ se e somente se $i < j$;

ii. Os itens dentro de cada balde são localmente ordenados usando algum outro algoritmo de ordenação.

A *p-classe BucketSort* implementa a versão paralela do algoritmo *bucketsort*. Para isso, é definida por herança a partir da *p-classe Sort*, implementando o

método *sort*. Existem duas estratégias para implementar *BucketSort*, baseado em duas maneiras de paralelizar o algoritmo *bucketsort*, levando a duas alternativas de implementações do método *sort*:

i. Existe um trabalhador raiz (*root*) que inicialmente conhece todos os itens, executando o primeiro passo do algoritmo sequencial. Então, os baldes são distribuídos por todos trabalhadores usando o método *scatter* do comunicador *comm*. Os trabalhadores executam uma ordenação sequencial dentro dos baldes que recebem. Finalmente, os trabalhadores enviam os baldes ordenados de volta ao trabalhador raiz usando o método *gather* pelo mesmo comunicador.

ii. Os itens são inicialmente distribuídos entre os trabalhadores, de modo que todo trabalhador possui versões locais de cada um dos k baldes e distribui seus itens locais pelos seus baldes locais. Depois disso, os trabalhadores distribuem os baldes, concatenando versões locais do mesmo balde em um balde único no espaço de endereçamento de um único trabalhador, usando o método *alltoallflattened* do comunicador *comm*. Seja r e s os *ranks* de dois trabalhadores arbitrários e seja b_i e b_j os baldes guardados em r e s, respectivamente. Na distribuição final, $r < s$ se e somente se, $i < j$. Finalmente, os trabalhadores executam ordenação sequencial dentro de cada balde que receberam. Os itens estão agora ordenados nos trabalhadores.

A segunda estratégia é mais apropriada quando existe uma computação rápida para decidir para qual balde um item deve ser associado. Além disso, o conjunto de itens deve caber no espaço de endereçamento do trabalhador raiz (*root*). A computação é dominada pela ordenação sequencial executada por cada trabalhador. Entretanto, tal estratégia quebra a suposição da *p-classe Sort* de que todos os itens devem estar distribuídos através dos das unidades de processamento dos trabalhadores no processo inicial de ordenação.

A primeira estratégia é melhor se os trabalhadores não possuem espaço de endereçamento suficiente para armazenar todos os itens. Assim, os itens devem estar distribuídos através das unidades de processamento dos trabalhadores durante a execução, como requerido pela *p-classe Sort*. A Figura 4.5 esboça a implementação do método *sort* da *p-classe BucketSort* usando a segunda estratégia. O código fonte é baseado no núcleo IS (Integer Sorting) do projeto NPB (*NAS Paralell Benchmarks*) [2], que implementa o *bucketsort* para avaliar o desempenho de *clusters* e MPP's

```
void OddEvenSort::worker::sort() {
  /* Declaração e inicialização de variáveis locais */
  ...
  /* Ordena elementos locais usando a rotina quicksort */
  local_sort(items, nlocal);
  /* Determina o rank dos vizinhos */
  oddrank  = rank % 2 == 0 ? rank - 1 : rank + 1;
  evenrank = rank % 2 == 0 ? rank + 1 : rank - 1;
  /* Laço principal do algoritmo Odd-even */
  for (j = 0; j < nworkers-1; j++)
  {
    if (j%2 == 1 && oddrank != -1 && oddrank != nworkers)
    {   // Odd-Even
        comm.isend(oddrank, 0, items, nlocal);
        comm.receive(oddrank, 0, ritems, nlocal);
    }
    else if (evenrank != -1 && evenrank != nworkers)
    {   // Even-Odd
        comm.isend(evenrank, 0, items, nlocal);
        comm.receive(evenrank, 0, ritems, nlocal);
    }
    compareSplit(nlocal, items,
                 ritems, wspace,rank < status.MPI_SOURCE);
  }
  ...
}
```

Figura 4.6: *p-classe* OddEvenSort (Método *sort*)

(*Massive Parallel Processors*). Na seção 4.3.4, uma avaliação de desempenho é apresentada.

Existem dois pontos importantes para salientar sobre esse estudo de caso:

- Um programador que utilize OOPP deve tomar cuidado com suposições implícitas de *p-classes* abstratas sobre distribuicões de entrada e saída de estruturas de dados, quando derivar *p-classes* concretas dessas, tal como a suposição de que *Sort* requer que os itens estejam distribuidos através das unidades trabalhadoras;

- As duas soluções usam métodos de comunicação coletiva do comunicador, mas o usuário do ordenador não necessita ter conhecimento sobre as operações de comunicação executadas pelas unidades paralelas *workers*. É uma perspectiva completamente transparente por parte do usuário. De fato, é seguro trocar

a estratégia de paralelismo pela simples mudança de subclasses distintas da *p-classe Sort* que implementa outra estratégia de paralelização.

4.3.3 A p-classe *OddEvenSort*

O Odd-even é um algoritmo de ordenação baseado no conhecido algoritmo *bubblesort* [6] (ordenação da bolha). Assim, ele possui complexidade $\Theta(n^2)$, sendo essencialmente distinto do *bubblesort*. Enquanto que *bubblesort* executa uma sequência de passagens no vetor de itens até que eles estejam ordenados, comparando elementos vizinhos e trocando-os caso eles estejam na ordem errada, odd-even alterna dois tipos de passagens. Em passagens *par-ímpar*, cada elemento ímpar é comparado com seu próximo vizinho. Em passagens *ímpar-par*, cada elemento par é compara com seu próximo vizinho. Os elementos comparados são ainda trocados quando eles estão na ordem errada, tal como no *bubblesort*. *Odd-even* é um algoritmo paralelo intrinsicamente de memória compartilhada, uma vez que as passagens *ímpar-par* e *par-ímpar* podem ser executadas concorrentemente.

A *p-classe OddEvenSort* implementa uma variação distribuída do algoritmo de ordenação *odd-even*. Tal como *BucketSort*, é derivado por herança de *Sort*, implementando o método *sort*, onde as comparações, trocas e operações de comunicação/sincronização ocorrem. O método *sort* de *OddEvenSort* é descrito na Figura 4.6. Os itens (vetor *items*) são distribuidos para os trabalhadores, tal como na segunda implementação do *bucketsort*. Antes de tudo, os trabalhadores ordenam seus itens locais usando algum algoritmo de ordenação (e. g. *quicksort*). Então, as passagens da ordenação *odd-even* são alternadas através das unidades trabalhadoras. Nas passagens *ímpar-par*, os trabalhadores com *ranks* ímpares e seus respectivos proximos vizinhos trocam seus itens. Nas passagens *impar-par*, os trabalhadores com *ranks* pares e seus respectivos próximos vizinhos trocam seus itens. A troca de itens entre trabalhadores é executada através de chamadas a operações ponto-a-ponto no comunicador *comm*, *isend* e *recv*. Após uma passagem, cada par de trabalhadores envolvidos em uma troca possui conhecimento sobre os itens de ambos os trabalhadores. Seja n o número de itens. Então, o trabalhador com o menor *rank* copia os $n/2$ menores itens em seu vetor *items*, onde o outro trabalhador copia os $n/2$ maiores. O processo é repetido até que os itens estejam ordenados em todos os trabalhadores.

4.3.4 Avaliação de Desempenho

O objetivo principal desse experimento também é comparar um programa paralelo que utiliza MPI com sua versão implementada em POb$C{+}{+}$. Dessa vez, utilizamos a implementação em C com MPI do *Integer Sort* (IS) pertencente ao projeto NPB desenvolvido pela divisão de supercomputação avançada da NASA, desprezando portanto o peso dos recursos da orientação a objetos do $C{+}{+}$. Esse algoritmo foi desenvolvido com o objetivo de testar a velocidade de computação com inteiros e o desempenho de comunicação do sistema distribuído envolvido.

A p-classe *IntegerSort*

A *p-classe IntegerSort* herda da *p-classe* Sort as unidades paralelas *worker* e implementa a segunda estratégia descrita na Seção 4.3.2. Esta possui vários métodos, onde os mais importantes são *sort* e *rank*. O primeiro é um método paralelo que evoca o processo de ordenação propriamente dito a partir da entrada de um comunicador. O método *rank* calcula qual localização será estipulada para cada chave. Abaixo, o processo resumido de ordenação, onde N denota a quantidade máxima de chaves para distribuir entre B_{max} baldes com valor máximo V_{max}.:

i. Geração sequencial de N chaves aleatórias com valor máximo V_{max};

ii. Início de medição de tempo;

iii. Calcular para cada chave seu *rank*;

iv. Enviar chave para o processo correspondente ao *rank* e balde calculado;

v. Fim da medição de tempo.

Durante a computação, diversas chamadas de comunicação coletiva são executadas no programa original: *MPI_Allreduce*, *MPI_Alltoall*, *MPI_Alltoallv* e *MPI_Reduce*. Da mesma forma, na execução do método paralelo *sort* tais comunicações são executadas utilizando os métodos de *Communicator*.

Experimento

Para medir o desempenho dos programas, utilizamos três classes de problemas previamente definidas pelo programa original: B, C e D. Na tabela 4.2, a descrição de cada classe de problema é definida a partir de três variáveis: N, B_{max} e V_{max}. Um caso de teste é definido pelo programa, NPB original ou POb$C{+}{+}$, pela classe

Classe	N	B_{max}	V_{max}
B	2^{25}	2^{10}	2^{21}
C	2^{27}	2^{10}	2^{23}
D	2^{29}	2^{10}	2^{27}

Tabela 4.2: Valoração atribuída por cada classe de problema

	classes de problemas											
	B				C				D			
P	PObC++	C/MPI	$+\delta$	$-\delta$	PObC++	C/MPI	$+\delta$	$-\delta$	PObC++	C/MPI	$+\delta$	$-\delta$
1	6,61	5,92	0,13	-0,07	26,8	24,2	0,12	-0,06	-	-	-	-
2	3,34	3,04	0,19	-0,07	13,4	12,3	0,11	-0,06	-	-	-	-
4	1,70	1,56	0,15	-0,07	6,91	6,3	0,11	-0,06	132	111	0,21	-0,14
8	0,88	0,80	0,12	-0,03	3,6	3,3	0,07	-0,06	69,0	58,9	0,19	-0,13
16	0,47	0,43	0,14	-0,06	1,9	1,7	0,10	-0,05	37,7	33,3	0,14	-0,09
32	0,27	0,25	0,12	-0,03	1,1	1,0	0,08	-0,03	23,2	21,2	0,10	-0,06
64	0,19	0,18	0,17	-0,05	0,7	0,7	0,31	-0,02	15,0	14,0	0,08	-0,03

Tabela 4.3: Sumário de resultado de desempenho *Tempo de execução*

de problema e pela quantidade de processadores utilizados. 40 execuções para cada caso com eliminação de extremos (*outliers*) para a normalização da distribuição dos tempos de execução. Desta forma, aumentamos a confiança nos dados selecionados para descrever a média de tempo de execução. Após a eliminação, analisamos os intervalos nos extremos e calculamos as diferenças relativas mínima e máxima em porcentagem, valores $-\delta$ e $+\delta$ na tabela, respectivamente. Esses valores foram calculados da mesma forma como explicado na Seção 4.1.4, além de utilizar a mesma máquina descrita e as mesmas configurações de quantidade de unidades de processamentos e de núcleos. O desvio padrão variou entre $0,07\%$ e $3,72\%$ para os experimentos utilizando PObC++ e entre $0,97\%$ e $4,02\%$ para C/MPI. É importante notar que para a classe D, o núcleo IS restringe a execução para mais de quatro processos.

O resumo dos resultados que está na Tabela 4.3 é uma evidência de que um código PObC++ não introduz sérias despesas computacionais, pois a diferença de tempo entre as versões C e PObC++ é pequena e o *speedup* é bastante similar. Apesar de que é notória a perda de eficiência quando chegamos ao uso de 64 processadores, pois chegamos próximo do limite de tempo serial para ordenar a quantidade de chaves e cada processo fica responsável por uma mínima parcela. Outra nuância importante é a diferença de linguagem e consequentemente o compilador podendo influenciar, já que pode existir uma diferença de otimização do código de máquina

gerado. Então, além de levar em conta a introdução das noções dos *p-objetos*, estamos avaliando o uso do próprio C++ e da orientação a objetos. Finalmente, alega-se novamente que ganhamos modularidade, abstração, usabilidade e perdemos uma aceitável porcentagem de desempenho.

Capítulo 5

Conclusões

Devido ao crescente interesse em técnicas de Computação de Alto Desempenho na indústria do software, principalmente através do processamento paralelo, assim como o ganho em importância das aplicações científicas e de engenharia na sociedade moderna, a crescente complexidade dessas aplicações nos domínios de CAD tem atraído atenção de um número significante de pesquisadores em modelos, linguagens e técnicas de programação, interessados em resolver o problema desafiador que é reconciliar conhecidas técnicas para lidar com complexidade de software e aplicações corporativas de larga escala com requisitos de alto desempenho.

A programação orientada-a-objetos é considerada uma das principais respostas de projetistas de linguagens de programação para lidar com alta complexidade e escala de software. Desde de 1990, tal estilo de programação se tornou amplamente difundido entre programadores. Apesar do seu sucesso entre programadores de domínio de aplicações corporativas, linguagens OO não foram ainda realmente aceitas entre programadores dos domínios de CAD, dominada por cientistas e engenheiros. Essa não aceitação é usualmente explicada pelas sobrecargas de desempenho acarretadas por algumas características presentes em linguagens que suporta alto nível de abstração, modularidade e segurança, e pela complexidade adicional introduzida pelo suporte à programação paralela.

Existem muitos projetos de pesquisa e desenvolvimentos técnicos que têm tentado reconciliar programação paralela para memória distribuida e linguagens orientadas-a-objetos, algumas delas foram discutidas ao longo da Seção 2.3. Porém, argumentamos que tais tentativas quebram importantes princípios da orientação-a-objetos e/ou não alcançam o nível de flexibilidade, generalidade e alto desempenho de programação paralela usando o padrão MPI. Para resolver tais

problemas, esse trabalho inclui as seguintes contribuições:

- Uma alternativa de programação orientada a objetos onde objetos são paralelos por padrão, chamado de OOPP (Programação Paralela Orientada a Objetos);
- O projeto de uma linguagem baseada na OOPP, chamada de POb$C{+}{+}$, demonstrando a viabilidade da OOPP como um modelo prático;
- Um protótipo de um compilador para a linguagem POb$C{+}{+}$, que pode ser usado para validar o uso e a performance da OOPP;
- Uma comparação entre o desempenho de programas em POb$C{+}{+}$ e sua contraparte em $C{+}{+}$/MPI e C/MPI, que evidencia que orientação a objetos não traz sobrecargas significativas de desempenho;
- Discussões sobre técnicas de programação por trás da OOPP, usando um conjunto de estudos de casos.

Os resultados apresentados nessa proposta, incluindo o projeto, implementação e avaliação de desempenho de um protótipo da linguagem POb$c{+}{+}$, são bastante promissores. Os exemplos são evidências que a abordagem proposta pode conciliar coerentemente os estilos comuns de programação adotados por programadores paralelos e de programadores que utilizam orientação a objetos, tornando isso possível para um programador bem educado em programação paralela com MPI e OO, tomar vantagem rapidamente das características da OOPP. Além disso, os resultados de desempenho trazem evidências de uma sobrecarga tolerável, ou insignificantes de desempenho, apesar do ganho em modularidade e abstração quando comparado a programação direta com MPI.

Como trabalhos futuros, podemos enumerar:

- Definição de uma semântica formal para a linguagem POb$C{+}{+}$;
- Aplicação do OOPP em outras linguagens OO;
- Análise estática para verificar problemas de composição paralela;
- Desenvolvimento de um compilador para uso em produção;
- Análise de mais estudos de caso.

Referências Bibliográficas

[1] Andrews, G. *Concurrent Programming: Principles and Practice.* Addison Wesley, 1991.

[2] Bailey, D. H., Harris, T., Shapir, W., van der Wijngaart, R., Woo, A., and Yarrow, M. The NAS Parallel Benchmarks 2.0. Tech. Rep. NAS-95-020, NASA Ames Research Center, Dec. 1995. *http://www.nas.nasa.org/NAS/NPB.*

[3] Baker, M., Buyya, R., and Hyde, D. Cluster Computing: A High Performance Contender. *IEEE Computer 42*, 7 (July 1999), 79–83.

[4] Baker, M., and Carpenter, B. MPJ: A Proposed Java Message Passing API and Environment for High Performance Computing. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing* (London, UK, 2000), Springer-Verlag, pp. 552–559.

[5] Baker, M., Carpenter, B., Fox, G., Ko, S. H., and Li, X. mpiJava: A Java Interface to MPI. In *In Procedings of the First UK Workshop on Java for High Performance Network Computing.* (1998).

[6] Batcher, K. E. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference* (New York, NY, USA, 1968), AFIPS '68 (Spring), ACM, pp. 307–314.

[7] Bernholdt D. E., Nieplocha, J., and Sadayappan, P. Raising Level of Programming Abstraction in Scalable Programming Models. In *IEEE International Conference on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing (P-PHEC)* (2004), Madrid, Spain, IEEE Computer Society, pp. 76–84.

[8] Bertran, F., Bramley, R., Sussman, A., Bernholdt, D. E., Kohl, J. A., Larson, J. W., and Damevski, K. B. Data Redistribution and Remote Method Invocation in Parallel Component Architectures. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Apr. 2005), IEEE.

[9] Burkardt, J. NINTLIB - Multi-dimensional Quadrature. *http://people.sc.fsu.edu/ burkardt/f_src/nintlib/nintlib.html*.

[10] Burns, G., Daoud, R., and Vaigl, J. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium* (1994), pp. 379–386.

[11] Carvalho-Junior, F. H., Lins, R., Correa, R. C., and Araújo, G. A. Towards an Architecture for Component-Oriented Parallel Programming. *Concurrency and Computation: Practice and Experience 19*, 5 (2007), 697–719.

[12] Chamberlain, B. L., Callahan, D., and Zima, H. P. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[13] Cole, M. *Algorithm Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.

[14] Cole, M. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing 30*, 3 (2004), 389–406.

[15] Dahl, O.-J. *The roots of object orientation: the Simula language*. Springer-Verlag New York, Inc., New York, NY, USA, 2002, pp. 78–90.

[16] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. Some features of the SIMULA 67 language. In *Proceedings of the second conference on Applications of simulations* (1968), Winter Simulation Conference, pp. 29–31.

[17] Deutsch, L. P., and Goldberg, A. Smalltalk yesterday, today and tomorrow. *BYTE 16* (August 1991), 108–ff.

[18] Dijkstra, E. The Humble Programmer. *Communications of the ACM 15*, 10 (1972), 859–866.

[19] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., and White, A. *Sourcebook of Parallel Computing*. Morgan Kauffman Publishers, 2003, ch. 20-21.

[20] Dongarra, J., Otto, S. W., Snir, M., and Walker, D. A Message Passing Standard for MPP and Workstations. *Communications of ACM 39*, 7 (1996), 84–90.

[21] Douglas, G., and Matthias, T. Boost.mpi website, http://www.boost.org/doc/html/mpi.html, May 2010.

[22] Foster, I. *The Grid: Blueprint for a New Computing Infrastructure*, 1 ed. Morgan Kaufmann, Nov. 1998.

[23] Gabriel, E., Fagg, G., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R., Daniel, D., Graham, R., and Woodall, T. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 353–377.

[24] Geist, G., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. S. PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. *MIT Press, Cambridge* (1994).

[25] Gregor, D. G., and Lumsdaine, A. Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)* (Feb. 2008).

[26] Haller, P., and Odersky, M. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci. 410* (February 2009), 202–220.

[27] Hoare, C. A. R. Monitors: an operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.

[28] Johnsson, S. L., Harris, T., and Mathur, K. K. Matrix Multiplication on the Connection Machine. In *Proceedings of the 1989 ACM/IEEE Conference*

On Supercomputing (New York, NY, USA, 1989), Supercomputing '89, ACM, pp. 326–332.

[29] Kale, L. V., and Krishnan, S. CHARM++: A Portable Concurrent Object Oriented System based on C++. *SIGPLAN Not. 28*, 10 (Oct. 1993), 91–108.

[30] Karonis, N., Toonen, B., and Foster, I. MPICH-G2: A Grid-enabled Implementation of the Messaging Passing Interface. *Journal of Parallel and Distributed Computing 63*, 5 (2003), 551–563.

[31] Kuchen, H., and Cole, M. Algorithm Skeletons. *Parallel Computing 32* (2006), 447–626.

[32] Kumar, V., Grama, A., Gupta, A., and Karypis, G. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[33] Lusk, E., and Yelick, K. Languages for High-Productivity Computing - The DARPA HPCS Language Support. *Parallel Processing Letters*, 1 (2007), 89–102.

[34] McPeak, S., and Necula, G. Elkhound: A Fast, Practical GLR Parser Generator. In *Compiler Construction*, E. Duesterwald, Ed., vol. 2985 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 2725–2725.

[35] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing 8*, 3-4 (1994), 169–416.

[36] Meyer, B. *Object-Oriented Software Construction*, 1st ed. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[37] Milli, H., Elkharraz, A., and Mcheick, H. Understanding Separation of Concerns. In *Workshop on Early Aspects - Aspect Oriented Software Development (AOSD'04)* (Mar. 2004), pp. 411–428.

[38] Nguyen, T., and Kuonen, P. Programming the grid with pop-c++. *Future Gener. Comput. Syst. 23*, 1 (2007), 23–30.

[39] OpenMP Architecture Review Board. OpenMP: Simple, Portable, Scalable SMP Programming, 1997.

[40] Post, D. E., and Votta, L. G. Computational Science Demands a New Paradigm. *Physics Today 58*, 1 (2005), 35–41.

[41] Sarkar, V. X10: An Object Oriented Approach to Non-uniform Cluster Computing. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2005), ACM, p. 393.

[42] Steele, Jr., G. L. Parallel Programming and Code Selection in Fortress. In *Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2006), PPoPP '06, ACM, pp. 1–1.

[43] Taivalsaari, A. On The Notion Of inheritance. *ACM Comput. Surv. 28* (September 1996), 438–479.

[44] Yang, Z., and Duddy, K. CORBA: A Platform for Distributed Object Computing. *SIGOPS Oper. Syst. Rev. 30*.

[45] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. Titanium: A High-performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing (New* (1998), ACM, Ed., ACM Press.