UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Um Modelo de Paralelismo de Grão Fino para Objetos Distribuídos

por

RAFAEL BOHRER ÁVILA

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, abril de 1999

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Ávila, Rafael Bohrer

Um Modelo de Paralelismo de Grão Fino para Objetos Distribuídos / por Rafael Bohrer Ávila. — Porto Alegre: PPGC da UFRGS, 1999.
75 f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 1999. Orientador: Navaux, Philippe Olivier Alexandre.

1. Paralelismo de grão fino. 2. Objetos distribuídos. 3. DPC++. 4. Métodos concorrentes. 5. Monitores. I. Navaux, Philippe Olivier Alexandre. II. Título.

Dedicado a meu pai e minha mãe.

Agradecimentos

- à Universidade Federal do Rio Grande do Sul e ao Instituto de Informática, pela disponibilização da infra-estrutura e recursos técnicos;
- à CAPES, pelo suporte financeiro sem o qual este trabalho não poderia ser desenvolvido;
- à Ida, Ana Paula, Fabiana e todo o pessoal da biblioteca, pela incansável prestatividade;
- à Cristina e à Margareth, pelo excelente trabalho na resolução dos problemas da rede;
- aos amigos e colegas de curso, pelo companheirismo e pelos momentos de descontração imprescindíveis para o sucesso no desenvolvimento do trabalho;
- ao Prof. Philippe Navaux pela orientação, paciência, apoio e pela oportunidade de realizar esta pesquisa;
- aos meus familiares, pelo incentivo permanente e pela compreensão nos momentos de dificuldade.

A todos, citados e não citados (desculpem!), um especial agradecimento pela amizade que podemos cultivar. A vocês e a Deus, um sincero Muito Obrigado.

Sumário

Lista de Abreviaturas

MPP	*Massively Parallel Processors*
SMP	*Symmetric Multiprocessor*
SPMD	*Single Program—Multiple Data*
PVM	*Parallel Virtual Machine*
MPI	*Message Passing Interface*
POSIX	*Portable Operating System Interface*

Lista de Figuras

Lista de Tabelas

Resumo

Esta Dissertação apresenta um modelo de paralelismo de grão fino para utilização em aplicações baseadas em objetos distribuídos. A pesquisa é desenvolvida sobre o modelo de distribuição de objetos da linguagem DPC++, uma extensão de C++ concebida na Universidade Federal do Rio Grande do Sul. A motivação para o desenvolvimento deste modelo é a crescente disponibilidade de arquiteturas multiprocessadas e de tecnologias de comunicação de alto desempenho, o que permite o aproveitamento eficiente de um nível de concorrência de menor granularidade. O objetivo do trabalho é integrar de forma adequada e eficiente a utilização de tal nível de concorrência no modelo original de distribuição de objetos de DPC++, permitindo que as aplicações desenvolvidas com a linguagem possam explorar ao máximo o poder computacional oferecido pelas arquiteturas citadas.

Como principais características, o modelo proposto apresenta a capacidade de concorrência entre os métodos de um mesmo objeto distribuído e a introdução de um mecanismo de sincronização baseado na semântica de monitores. Os resultados obtidos com a implementação de uma aplicação de geração de fractais de Mandelbrot demonstram que, em termos de desempenho, o modelo apresentado efetivamente atinge seus objetivos. Além disso, a extensão à sintaxe original de programação de DPC++ revela importante contribuição no sentido de aumentar o poder de expressão da linguagem para o desenvolvimento de aplicações paralelas.

Palavras-chave: Paralelismo de grão fino, objetos distribuídos, DPC++, métodos concorrentes, monitores

TITLE: “A FINE-GRAIN PARALLELISM MODEL FOR DISTRIBUTED OBJECTS”

Abstract

This Thesis presents a fine-grain concurrency model for applications based on distributed objects. The basis for the development of this research is the model of distribution of objects presented by the language DPC++, an extension of C++ conceived at the Federal University of Rio Grande do Sul. The motivation for the development of this model is the growing availability of multiprocessor architectures and high-performance communication technologies, which allows for the efficient use of a finer grain of concurrency. The purpose of this work is to efficiently integrate such level of concurrency into the original model of distribution of objects of DPC++, making it possible for the applications developed with the language to thoroughly explore the computational power offered by the mentioned architectures.

As main features the proposed model presents the possibility of concurrency between the methods of a single distributed object and the introduction of a synchronisation mechanism based on the semantics of monitors. The results obtained with the implementation of a Mandelbrot fractal generation application show that, in relation to performance, the presented model effectively reaches its purposes. In addition, the extension to the original programming sintax of DPC++ reveals an important contribution towards a higher expressivity degree in the development of parallel applications.

Keywords: Fine-grain concurrency, distributed objects, DPC++, concurrent methods, monitors

1 Introdução

A grande popularização das redes locais de computadores contribuiu para o maior interesse na pesquisa em programação paralela, sendo as máquinas integrantes da rede utilizadas para compor uma máquina paralela virtual. Este tipo de arquitetura tornou-se, nos últimos anos, o método mais difundido na prática de computação paralela [BAK96]. Em comparação com os computadores construídos especificamente para o processamento distribuído, como as máquinas ditas massivamente paralelas (MPP) e as SMP — *Symmetric MultiProcessor* — as redes tornaram-se bastante atrativas em função de seu baixo custo e alta disponibilidade, o que estimula também a portabilidade das aplicações desenvolvidas.

Com o crescimento do interesse no processamento distribuído, começaram-se a desenvolver ferramentas com o objetivo de abstrair a arquitetura paralela usada e fornecer ao programador uma interface de programação uniforme, independentemente da organização ou características internas das máquinas empregadas. Pela natureza intrínseca da arquitetura de redes de computadores, o modelo de troca de mensagens ganhou especial atenção, surgindo bibliotecas de programação como PVM [GEI94] e MPI [MPI94]. Ferramentas para emulação de memória compartilhada sobre redes foram também desenvolvidas, como o pacote TreadMarks [TRE98].

As diferenças técnicas entre uma máquina virtual montada sobre uma rede e uma máquina massivamente paralela, por exemplo, com conexões proprietárias entre os nós e, portanto, mais eficientes, levam a um fator que deve ser levado em conta quando do projeto da aplicação paralela, de modo que esta apresente o desempenho adequado. Esse fator, chamado *grão de paralelismo*, pode ser entendido como a relação entre a quantidade de computação e de comunicação efetuada pelos nós de processamento. Dependendo da maior ou menor eficiência de comunicação entre os nós, essa relação tem que ser balanceada, quando do projeto da aplicação, de modo que a quantidade de processamento efetuada por um nó compense, em termos de desempenho, o tempo gasto em uma posterior operação de comunicação efetuada por ele.

A utilização do paradigma de troca de mensagens requer, freqüentemente, o emprego de um grão de paralelismo de tamanho médio a grosso, pois a arquitetura utilizada geralmente implica em custos relativamente elevados de comunicação entre os nós de processamento. É o caso, por exemplo, quando do uso de redes locais como máquinas paralelas, tipicamente conectadas por tecnologia Ethernet 10Mbits. Na utilização de memória compartilhada a comunicação é mais eficiente, pois é feita diretamente através de operações em memória, geralmente entre vários fluxos de execução de um mesmo processo, e portanto o grão de paralelismo pode ser mais fino. É importante salientar, então, que os recursos oferecidos por uma ferramenta de programação são fundamentais para que o grão de paralelismo possa ser corretamente adequado pelo usuário.

Um paradigma de programação bastante propício ao desenvolvimento de aplicações paralelas de granularidade média a grossa (tipicamente troca de mensagens) é o modelo de orientação a objetos. Oferecendo um nível de abstração mais alto que o de bibliotecas de comunicação como PVM e MPI, o modelo de orientação a objetos mostra-se bem adequado à computação distribuída. Conforme demonstrado por Cavalheiro e Navaux [CAV93], pode-se estabelecer uma analogia entre os dois sistemas:

- um *objeto* corresponde a um *processo*;

- os *atributos* de um objeto correspondem à área de *memória* de um processo;
- os *métodos* de um objeto correspondem ao *código executável* de um processo;
- uma chamada a um *método* de um objeto corresponde ao envio de uma *mensagem* a um processo.

Essa adequação levou ao surgimento de um modelo chamado de *objetos ativos* [ELL89], utilizado em muitas linguagens para programação distribuída da atualidade [ÁVI97]. Neste modelo os objetos da aplicação são instanciados em nós de processamento distintos, podendo então realizar suas funções em paralelo. A característica principal do modelo de objetos ativos é o encapsulamento de um objeto a uma entidade ativa do sistema, tipicamente um processo do sistema operacional. Este processo possui um canal de comunicação com o meio externo, por onde são recebidas e enviadas mensagens que representam as chamadas a métodos.

Um exemplo de linguagem de programação que adota o modelo de objetos ativos é a linguagem DPC++ — *Distributed Processing in C++* [CAV93]. DPC++ consiste em uma extensão de C++ que permite que uma aplicação seja facilmente construída e executada de forma distribuída sobre uma rede de computadores, aliando os benefícios da programação orientada a objetos com o ganho em desempenho advindo da execução distribuída.

Assim como DPC++, várias outras linguagens e ambientes para programação distribuída tiveram seu desenvolvimento motivado, em grande parte, pela disponibilidade de redes locais; em conseqüência, vistas as considerações feitas sobre o grão de paralelismo, os recursos de programação oferecidos em tais linguagens voltaram-se, principalmente, ao projeto de aplicações distribuídas, ou seja, com grão de paralelismo médio a grosso.

Atualmente, entretanto, o desenvolvimento de novas tecnologias de interconexão e de construção de máquinas paralelas tem tornado cada vez mais possível a exploração de um nível mais fino de paralelismo em aplicações concorrentes. Por um lado, nota-se uma tendência de convergência de redes de estações com as máquinas massivamente paralelas pelo desenvolvimento de tecnologias de comunicação de alto desempenho como Myrinet [BOD95], Fast Ethernet [IEE95] e SCI [IEE92]. Por outro lado, tem-se um crescimento da disponibilidade de máquinas SMP pela redução de seus custos de construção, de forma que computadores com dois, quatro ou oito processadores passam a fazer cada vez mais parte da realidade de instituições de pesquisa.

Em particular, tem-se dado especial atenção à pesquisa em arquiteturas que combinam as duas tecnologias, originando agregados (*clusters*) cujos nós são máquinas SMP conectadas por alguma das tecnologias rápidas citadas anteriormente. Exemplos de pesquisas nessa área são as máquinas SNOW [FRÖ98], do GMD, com 8 nodos conectados por tecnologias Myrinet e Fast Ethernet, e as máquinas PC1 e PC2 [HEI98], da Universidade de Paderborn, na Alemanha, com 32 e 96 nodos, respectivamente, conectados por tecnologia SCI. Ambos os projetos utilizam nodos Dual PentiumII, com 2 processadores. Além desses exemplos, o próprio Instituto de Informática da UFRGS conta atualmente com um agregado de 4 máquinas Dual Pentium conectadas por tecnologia Fast Ethernet.

A crescente disponibilidade desse tipo de arquitetura, portanto, tem motivado o desenvolvimento de pesquisas no sentido de extrair o máximo do poder computacional de tais máquinas paralelas, procurando explorar todos os níveis de paralelismo oferecidos e, ao mesmo tempo, oferecer ao programador um conjunto adequado de recursos de programação. O crescimento do interesse na exploração de paralelismo fino tem motivado, por exemplo, o surgimento de diversas implementações [PRO99, LEW98, LER99] do

padrão POSIX Threads [IEE95a], que define uma interface uniforme para programação portável com múltiplos fluxos de execução.

O modelo de objetos distribuídos proposto para DPC++ representa uma integração bastante adequada do paradigma de orientação a objetos com um ambiente distribuído, onde o grão de paralelismo é explorado em níveis médio a grosso. Em função disso, o modelo não apresenta características que reflitam o uso de um nível mais fino de concorrência. Contudo, considerando-se a crescente disponibilidade de arquiteturas que, por suas características, oferecem a exploração de paralelismo de granularidade fina, torna-se desejável que DPC++ possa fazer uso desses recursos. Para tanto, faz-se necessário que o modelo de objetos distribuídos apresentado pela linguagem seja estendido.

Esta Dissertação, portanto, propõe uma extensão ao modelo de distribuição de objetos de DPC++, tanto no nível de linguagem de programação quanto no nível operacional, que permita à linguagem explorar eficientemente um grão fino de paralelismo, sem, contudo, abdicar de suas características principais.

No restante do texto, o capítulo 2 apresenta um estudo dos principais mecanismos de expressão de paralelismo utilizados na prática da programação paralela, como embasamento teórico para a apresentação do modelo proposto. Na seqüência, o capítulo 3 apresenta a forma como tais mecanismos são utilizados em linguagens para programação paralela da atualidade. Em seguida, o capítulo 4 apresenta o modelo de distribuição de objetos empregado na linguagem DPC++, destacando suas principais características e fornecendo também um embasamento teórico para a definição da extensão proposta. O capítulo 5 apresenta a extensão propriamente dita, definindo um novo modelo de distribuição de objetos para DPC++. No capítulo 6 faz-se um detalhamento maior da implementação do modelo, seguido pelo capítulo 7 que apresenta resultados obtidos na implementação de uma aplicação utilizando os novos recursos introduzidos na linguagem. Por fim, o capítulo 8 apresenta as conclusões sobre o trabalho.

2 Mecanismos de expressão de paralelismo

A programação paralela revelou, ao longo dos anos, uma série de mecanismos de linguagem para a expressão de paralelismo das mais variadas formas. Com o desenvolvimento de novos conceitos no nível de programação e com o aparecimento e aperfeiçoamento de novas arquiteturas computacionais, vários mecanismos já existentes foram adaptados de modo a se encaixarem nos novos modelos de programação.

Este capítulo dedica-se, portanto, a fazer um apanhado geral dos principais mecanismos para expressão de paralelismo encontrados na literatura, baseando-se principalmente nas obras de Andrews [AND91] e Silberschatz [SIL91], dos quais a maioria dos mecanismos estudados foi obtida. Naturalmente, dentro do contexto do trabalho, o enfoque é dado sobre mecanismos de exploração de paralelismo fino.

2.1 Expressão de paralelismo

2.1.1 Paralelismo implícito *versus* paralelismo explícito

Uma das principais questões quando do projeto de linguagens ou novos mecanismos para expressão de paralelismo é sobre a forma de oferecê-lo ao programador, que pode ser implícita ou explícita.

Conforme defendido por autores como East [EAS95], o paralelismo implícito seria a forma mais natural de programação, pois os eventos do mundo real ocorrem efetivamente em paralelo. A serialização de tarefas é que seria, portanto, a exceção, e então seriam utilizados mecanismos para expressá-la, explicitamente, quando se fizesse necessário.

Entretanto, desde os primórdios da Informática, com o estabelecimento e difusão da arquitetura von Neumann, a execução de programas de computadores foi feita de forma seqüencial, e este modelo de programação tornou-se o padrão de fato. Com o advento da programação paralela, os costumes estabeleceram que o “normal” fosse a execução seqüencial, e o “diferente” a execução paralela. Assim surgiram os primeiros mecanismos para expressão explícita de paralelismo.

O paralelismo explícito, porém, obriga o programador a decidir *quando* e *onde* paralelizar o seu algoritmo, além da necessidade de estabelecer pontos de sincronização quando necessário, tirando sua atenção da resolução do problema. Buscando solucionar essa questão, surgiram pesquisas no sentido de tomar um programa escrito para executar de forma seqüencial e introduzir paralelismo onde possível, voltando à idéia do paralelismo implícito. Esta abordagem teria sucesso não fosse a grande complexidade envolvida em introduzir paralelismo em um programa seqüencial, por exemplo pela necessidade de se analisar as dependências de dados entre diferentes tarefas.

A filosofia seguida neste trabalho defende que a programação paralela deva ser encarada de forma mais natural, como um paradigma de programação, e não como um conjunto de primitivas a ser utilizado para aumentar o desempenho de um algoritmo. Assim, defende-se que a prática da programação paralela requer a utilização de uma ferramenta que incorpore o paralelismo dentro do seu modelo de programação, de forma que ele surja como uma conseqüência do projeto de um algoritmo, e não como um objetivo. Em outras palavras, que o paralelismo seja natural e, portanto, implícito.

2.1.2 Blocos paralelos

O *bloco paralelo* é o mecanismo mais intuitivo para expressão de paralelismo em um algoritmo. Dado um conjunto de tarefas a serem realizadas, a inclusão de tais tarefas em um bloco paralelo faz com que elas sejam executadas concorrentemente entre si. A estrutura geral de um bloco paralelo é:

início do bloco
  tarefa 1
  tarefa 2
  ...
  tarefa n
fim

A partir do início do bloco são criados n fluxos de execução independentes, de modo que cada um executa uma das tarefas. A execução do programa só prossegue, ao fim do bloco, depois que todas as tarefas forem completadas. Neste ponto, os n fluxos de execução extras são encerrados, e somente o fluxo original continua.

CC++ e ICC++ são exemplos de linguagens que oferecem o mecanismo de blocos paralelos. A sintaxe e algumas variações de semântica apresentadas pelas linguagens são melhor detalhadas no capítulo seguinte.

2.1.3 Laços paralelos

Uma forma também intuitiva de oferecer a expressão de paralelismo é incorporá-lo em comandos de repetição (como `for`), originando os *laços paralelos*. A estrutura de um laço paralelo é:

repita de 1 até n
  tarefa 1
  tarefa 2
  ...
  tarefa m
fim

Este comando cria um fluxo de execução independente para cada valor assumido pela variável iteradora, e executa-os de forma concorrente. Cada fluxo de execução, por sua vez, executa cada uma das tarefas de 1 até m. O processamento interno a um fluxo de execução, porém, é seqüencial. Como no caso dos blocos paralelos, a execução do programa só prossegue após o término da última tarefa de cada iteração paralela.

2.1.4 Expressão irregular de paralelismo

Diferentemente de blocos e laços paralelos, algumas linguagens disponibilizam uma forma de expressão irregular de paralelismo, ou paralelismo não estruturado. Este mecanismo é geralmente oferecido pela definição de uma palavra-chave que, prefixando uma chamada de procedimento, cria um novo fluxo de execução para essa tarefa. Utilizando a sintaxe empregada em CC++ e ICC++, a figura 2.1 ilustra a utilização de tal mecanismo.

No exemplo dado, após a execução do procedimento `a()`, é criado um novo fluxo de execução para o procedimento `b()`. O fluxo de execução original prossegue, imediatamente executando (em paralelo, portanto, com `b()`) o procedimento `c()`.

```
...
a();
spawn b();
c();
...
```

FIGURA 2.1 - Comando para expressão irregular de paralelismo.

2.1.5 Métodos concorrentes

Os mecanismos vistos até agora refletem uma característica bastante intuitiva de expressão de paralelismo, sem estarem restritos a um determinado modelo de programação. Assim, tais mecanismos tanto podem ser empregados em aplicações puramente procedurais quanto naquelas que seguem um paradigma específico, como a orientação a objetos.

Uma outra opção é integrar a expressão de paralelismo diretamente com o modelo de programação oferecido pela linguagem; assim, na orientação a objetos, surge o mecanismo de *métodos concorrentes*.

Algumas linguagens para programação paralela orientada a objetos oferecem a possibilidade de que os métodos de um mesmo objeto sejam executados de forma concorrente entre si. Essa característica dispensa, em conseqüência, a utilização de mecanismos extras como blocos ou laços paralelos, incorporando o paralelismo ao modelo de programação. A maior ou menor adequabilidade de utilização de um ou outro mecanismo recai, assim, sobre a forma como é modelado o algoritmo.

No nível de linguagem de programação, a concorrência entre os métodos pode ser oferecida de forma implícita ou explícita. Na forma implícita, assume-se que todos os métodos definidos para uma determinada classe são potencialmente concorrentes entre si. Na forma explícita, a linguagem deve prover um meio de identificar-se quais os métodos que devem ter sua execução paralelizada. Uma terceira possibilidade permite ainda que a classe como um todo seja identificada como paralela ou não; tal definição afeta, em conseqüência, todos os métodos da classe.

Dentre as linguagens que empregam métodos concorrentes podem ser citadas COOL e Panda. COOL adota a abordagem explícita, de forma que os métodos a serem executados em paralelo devem ser identificados pela palavra-chave `parallel`. Já a linguagem Panda segue a terceira abordagem, identificando uma classe paralela através de um mecanismo de herança.

2.2 Mecanismos para sincronização

A possibilidade de existência de múltiplos fluxos de execução operando sobre os mesmos dados, visto que os mecanismos apresentados são próprios de programação com memória compartilhada, levou à definição de mecanismos para a sincronização de tarefas. Esta seção relaciona alguns dos principais recursos de sincronização empregados na prática da programação paralela.

2.2.1 Mutexes

O termo *mutex* vem da expressão *mutual exclusion*, e consiste em uma forma bem simples e intuitiva de sincronização. Um mutex representa acesso exclusivo, por parte

de um fluxo de execução, a um trecho de um programa paralelo, sendo associado a duas operações básicas para adquiri-lo e liberá-lo, conforme a seguinte estrutura:

```
...
trecho que pode ser compartilhado
aquisição do mutex
    trecho que requer exclusividade de acesso
liberação do mutex
...
```

Um exemplo de linguagem que oferece esse mecanismo é Presto, que implementa a classe *Lock* que, por sua vez, oferece os métodos `lock()` e `unlock()`.

A implementação de mutexes aparece, em linguagens de programação, tanto na forma apresentada quanto integrada à orientação a objetos. Em linguagens como CC++, COOL e Java é possível definir um *método atômico*, de modo que um mutex implícito, associado ao objeto, é adquirido quando o método é chamado, sendo liberado ao seu final.

A linguagem Java permite, ainda, que sejam declarados blocos de exclusão mútua, onde o início do bloco marca a aquisição implícita de um mutex, e o fim do bloco sua liberação.

2.2.2 Semáforos

O mecanismo de *semáforos* permite um controle mais elaborado de sincronização, relacionando o acesso a seções críticas com o estabelecimento de certas condições no decorrer do algoritmo. Um semáforo oferece duas operações básicas, tradicionalmente chamadas de *p* e *v*, que constituem os protocolos de entrada e saída de uma seção crítica.

Um semáforo possui um contador interno, inicializado no momento de sua criação. Este valor é decrementado quando um processo executa uma operação *p* e incrementado quando de uma operação *v*. Se o resultado de uma operação *p* é negativo, então o processo é bloqueado, permanecendo nesse estado até que outro processo execute uma operação *v*.

Apesar de ser um mecanismo largamente utilizado na prática da programação paralela, somente a linguagem Panda implementa semáforos diretamente. Isto se deve ao fato de um semáforo poder ser facilmente implementado a partir de outros recursos de sincronização como mutexes e monitores.

2.2.3 Barreiras

Uma *barreira* também é um mecanismo simples de sincronização, e consiste no estabelecimento de um ponto, durante a execução de vários processos, onde cada um aguarda pela chegada de todos os demais. Assim que o último processo atinge o ponto determinado, cada um prossegue sua execução normalmente. O mecanismo de barreiras é útil, por exemplo, na inicialização de aplicações paralelas, onde é comum a necessidade do estabelecimento de condições iniciais antes que os processos comecem a sua execução.

2.2.4 Variáveis síncronas

A linguagem CC++ implementa um mecanismo semelhante ao de barreiras chamado de *variáveis síncronas*. Uma variável síncrona possui, inicialmente, um valor indeterminado. Uma tentativa de leitura do conteúdo da variável, neste período, resulta no bloqueio do processo que executou a operação. No momento em que um determinado

processo define o conteúdo da variável, ou seja, executa uma operação de escrita, todos aqueles que haviam sido bloqueados são liberados, e a partir desse momento a variável passa a ser uma constante.

2.2.5 Monitores

Um *monitor* é um mecanismo de sincronização bastante elaborado, que permite um controle ainda mais poderoso do que o de semáforos sobre recursos compartilhados. Um monitor é composto de variáveis internas, que representam o estado do recurso sendo controlado, e procedimentos para alterá-las e modificá-las (de forma semelhante à declaração de uma classe). Esses procedimentos são a única maneira pela qual um processo pode manipular o recurso representado.

De modo que as variáveis internas sejam mantidas em um estado consistente, dada a possibilidade de acesso concorrente por dois ou mais processos ao monitor, todos os procedimentos são implicitamente atômicos.

Durante a execução de um algoritmo paralelo, um processo pode encontrar o monitor em um estado tal que ele não pode seguir sua execução, por exemplo se o processo é um consumidor e o monitor representa um *buffer* que está vazio. Neste caso, o processo deve ser bloqueado e somente reiniciado quando o estado do monitor se tornar válido (ou seja, quando um item for produzido).

Com essa finalidade, os monitores definem *variáveis condicionais* que podem ser associadas a uma condição particular e usadas para bloquear e reiniciar processos. Tais variáveis são declaradas como parte dos dados internos do monitor e, portanto, somente são acessíveis pelos procedimentos definidos.

Duas operações básicas, *wait* e *signal*, são associadas a variáveis condicionais. Uma operação *wait* bloqueia o processo em execução, que é armazenado em uma fila mantida pela variável. Adicionalmente, uma operação *wait* libera o mutex implícito usado no monitor, de modo que outros processos possam ter acesso a ele. Uma operação *signal* tem o efeito complementar, ou seja, libera um processo previamente bloqueado em uma variável condicional. Se não há nenhum processo para ser desbloqueado, uma operação *signal* não tem qualquer efeito.

Note-se que, após a execução de um *signal*, existem potencialmente dois processos executando o mesmo procedimento do monitor, ou seja, aquele que foi liberado e aquele que executou a operação. Como esta é uma situação inválida, um dos dois processos deve permanecer bloqueado até que o outro termine o procedimento. Decidir qual abordagem seguir, no momento da implementação, é tradicionalmente um ponto de discussão, pois existem diversas alternativas [ALT91] e ambas as abordagens podem ser bem justificadas [MON91].

Podem ser definidas operações adicionais sobre variáveis condicionais, como *empty*, que testa se existe algum processo bloqueado, e *signal_all*, que libera todos os processos bloqueados ao mesmo tempo.

Devido à sua semelhança com a estrutura de um objeto, o mecanismo de monitores pode ser integrado de forma bastante adequada a esse paradigma de programação, como apresenta, por exemplo, a linguagem Java. Outras linguagens como COOL, Panda e Presto oferecem monitores como entidades separadas.

2.2.6 Sincronização implícita

De forma semelhante à utilização de paralelismo implícito, a sincronização entre diferentes processos pode ser imaginada com um caráter implícito, ou seja, o próprio modelo de programação ou uma análise feita pelo compilador podem determinar onde se fazem necessárias operações de sincronização em um algoritmo paralelo. Tal abordagem é seguida, por exemplo, pela linguagem ICC++, que oferece um modelo de consistência de objetos, conforme apresentado no capítulo seguinte. Essa característica, porém, pode ser bastante dificultada pela complexa análise exigida sobre o código da aplicação paralela. Além disso, parece mais natural que o estabelecimento de sincronismo no acesso a dados compartilhados seja tarefa do próprio desenvolvedor da aplicação.

2.3 Considerações no nível de implementação

Esta seção discute dois mecanismos que não afetam, necessariamente, o comportamento dos recursos de programação disponibilizados no nível de linguagem, mas cujo entendimento é bastante útil na consideração da implementação de tais recursos.

2.3.1 Fluxos de execução — *threads*

Dentre as diversas formas e níveis de paralelismo empregados em aplicações concorrentes, o conceito de múltiplos fluxos de execução, ou *threads*, é atualmente um mecanismo amplamente utilizado para a implementação de paralelismo de grão fino. Esta explanação também tem como objetivo estabelecer uma definição simplificada de *thread* na forma como ele é utilizado neste trabalho.

Um *thread* é sempre associado, em última análise, a um processo disponibilizado pelo sistema, e consiste em um fluxo independente de execução deste processo. Todos os *threads* de um processo compartilham os mesmos código executável e espaço de endereçamento, sendo este último geralmente utilizado para comunicação. Cada *thread* possui, entretanto, contexto e área de pilha independentes. A grande vantagem de utilização de *threads*, além do compartilhamento de memória, é o custo de sua criação, bastante reduzido em relação àquele de um processo.

O mecanismo de *threads* é largamente utilizado na atualidade, contando com uma interface padronizada (POSIX Threads) e sendo disponibilizado em diversos ambientes e bibliotecas para programação concorrente como *Aboelha* [FRÖ96] e *Athapascan0* [GIN97].

2.3.2 *Futures*

O *future* é um mecanismo introduzido por Halstead, Jr. [HAL85], projetado para aumentar o grau de concorrência de um algoritmo paralelo. Embora seja originalmente concebido como uma característica oferecida no nível de linguagem de programação, o mecanismo é aqui relacionado pelo fato de poder ser introduzido de forma transparente ao programador, sem alterar a sintaxe nem a semântica da linguagem.

Um *future* é aplicado a chamadas de procedimentos que retornam algum valor, que normalmente forçam o processo a esperar pelo seu término. Com o uso de *futures*, o processo não é bloqueado, mas prossegue sua execução em paralelo com a chamada do procedimento, até que chegue a um ponto onde o valor seja realmente necessário (por

exemplo, para ser atribuído a uma outra variável). Antes desse ponto, porém, o processo pode "ganhar tempo" desempenhando outras atividades. A variável que contém o *future*, ou seja, aquela utilizada para receber o valor de retorno do procedimento, pode inclusive ser utilizada como parâmetro na chamada de outros procedimentos, já que essa operação não implica, por si só, em ler o seu conteúdo. Assim que o valor de retorno da função é recebido, a variável assume o seu caráter normal. O processo é bloqueado se tenta ler o *future* antes que a resposta seja recebida, e desbloqueado quando do recebimento.

A utilização de *futures* pode contribuir de forma significativa para o grau de paralelismo apresentado por um algoritmo, dependendo, naturalmente, da forma como ele é projetado. Além disso, este mecanismo não interfere na semântica natural do programa, dispensando o uso de primitivas adicionais de sincronização. Note-se, porém, que a implementação de *futures* na forma apresentada não é uma tarefa muito simples, pois envolve, entre outros aspectos, uma análise detalhada do algoritmo para se determinar a ocorrência de operações de leitura. Algumas linguagens como ABC++, COOL e C++// oferecem o mecanismo de *futures* de forma modificada, com a declaração de variáveis especiais ou pelo uso de funções específicas (interferindo, portanto, na sintaxe de programação), o que torna sua implementação mais viável.

2.4 Avaliação dos mecanismos estudados

Percebe-se, dentre os mecanismos e linguagens apresentadas, que a utilização de determinado recurso é tanto mais adequada quanto melhor for sua integração com o modelo de programação oferecido. Assim, a utilização de blocos concorrentes em CC++ e ICC++ mostra-se bem adequada a um paradigma procedural, enquanto que o uso de monitores, em Java, reflete uma integração favorável ao desenvolvimentos de aplicações orientadas a objetos.

Considerando-se as colocações feitas sobre as formas explícita ou implícita de expressão de paralelismo e sincronização, pode-se observar que a maioria das linguagens opta pela abordagem explícita. Levando-se em conta que os costumes de programação estabeleceram o processamento seqüencial como "normal", e portanto a prática da programação paralela pode ser favorecida, no que diz respeito à sua compreensão e assimilação, pelo estabelecimento de primitivas adicionais para expressar tarefas "incomuns", essa opção não é mais que natural.

A programação paralela, entretanto, sofre de constantes evoluções e, no decorrer desse processo, muitas idéias e abordagens novas surgem na busca de aprimorar os conhecimentos já estabelecidos. Uma dessas correntes coloca a programação paralela como um outro paradigma de programação, e a assimilação de um novo paradigma requer, em função de seu próprio bem entendimento, uma introdução gradativa de seus conceitos nos modelos já existentes. No entendimento deste trabalho, uma ferramenta adequada de programação paralela é aquela, como já mencionado, que introduz o paralelismo como uma conseqüência de um modelo de programação que seja natural. Portanto, tanto mais se considera um recurso de programação paralela como adequado quanto mais ele estimula, por sua própria semântica, o entendimento do paralelismo de forma implícita.

O próximo capítulo apresenta diversas linguagens para programação paralela da atualidade, mostrando como os mecanismos estudados neste capítulo são integrados nas diferentes sintaxes e modelos de programação. A apresentação serve, também, como fonte de estudos para a definição do modelo de paralelismo fino proposto.

3 Linguagens para programação paralela

Este capítulo apresenta um estudo feito entre diversas linguagens para programação paralela orientadas a objetos da atualidade, buscando identificar a maneira como os mecanismos de expressão de paralelismo e sincronização, vistos no capítulo anterior, são oferecidos ao usuário e sua integração com o paradigma de programação. O estudo tem ainda como objetivos permitir uma avaliação do modelo de programação oferecido por DPC++ e justificar posteriormente a adoção das características do novo modelo.

À exceção de Java, todas as linguagens estudadas são baseadas em extensões de C++, principalmente por ser esta uma das mais populares e bem aceitas das atuais linguagens para programação orientada a objetos. Essa generalização permite também uma comparação mais pertinente com DPC++, que segue a mesma regra.

Um trabalho anterior [ÁVI97] apresenta uma comparação geral entre tais linguagens e DPC++, contribuindo significativamente para o desenvolvimento do modelo de paralelismo fino proposto.

Este capítulo procura concentrar sua atenção nas características relacionadas com a expressão de paralelismo de grão fino nas linguagens estudadas, portanto a apresentação não tem a intenção de ser completa. Informações mais detalhadas sobre as linguagens podem ser encontradas na bibliografia indicada e nas coletâneas apresentadas por Wyatt et al. [WYA92], Furmento et al. [FUR95] e Wilson e Lu [WIL96].

A próxima seção faz uma classificação das linguagens de acordo com o modelo de programação oferecido, o que influi diretamente nos mecanismos e na forma como estes são disponibilizados. Em seguida inicia-se a apresentação das linguagens e de suas principais características.

3.1 Modelos de programação

Conforme definido por Furmento et al. [FUR95], as linguagens para programação paralela orientada a objetos podem ser classificadas, de acordo com o modelo de programação adotado, em três grupos, apresentados a seguir.

3.1.1 Modelo de tarefas concorrentes

As linguagens deste grupo utilizam mecanismos básicos de expressão de paralelismo de forma explícita e direta, através de novos comandos ou recursos como herança, sem necessariamente integrá-los ao paradigma de orientação a objetos. Mecanismos de sincronização como mutexes e monitores são também freqüentemente disponibilizados de forma explícita.

3.1.2 Modelo de paralelismo de dados

Este modelo de programação procura atribuir, de forma geral, operações paralelas implícitas sobre estruturas de dados. A linguagem ICC++, por exemplo, permite a declaração de grupos de objetos, chamados *collections*, sobre os quais é possível executar uma invocação de método. Como resultado, o método é chamado em paralelo sobre todos os objetos do grupo.

Exemplos de linguagens que seguem esse modelo são pC++ [YAN96],

C** [LAR96] e Dome [FUR95]. A análise sobre linguagens desse grupo não será aprefundada por fugirem dos objetivos de DPC++, sobre a qual o trabalho se baseia.

3.1.3 Modelo de objetos ativos

Este grupo compreende as linguagens que implementam objetos associados a processos, definindo o conceito de objetos ativos [ELL89]. Essa abordagem leva as linguagens deste grupo a darem maior enfoque à programação distribuída.

Dentro do grupo de objetos ativos identificam-se os modelos ditos *mono-ativo* e *multi-ativo*. No modelo mono-ativo o fluxo de execução interna de um objeto distribuído é único, ou seja, não há concorrência interna. No modelo multi-ativo cada invocação a um objeto distribuído gera um novo fluxo de execução, possibilitando a concorrência interna.

3.2 CC++

CC++ (*Compositional C++*) [KES96] é uma linguagem desenvolvida pelo *California Institute of Technology*, destinada tanto a ambientes distribuídos como a arquiteturas SMP. CC++ segue o modelo de tarefas concorrentes, embora ofereça um mecanismo semelhante ao de objetos ativos. A declaração de *objetos globais*, associados a *objetos processadores*, define um ambiente de execução onde objetos podem ser remotamente instanciados. Essa característica, porém, não é a base do modelo de programação, como no modelo de objetos ativos.

3.2.1 Expressão de paralelismo

CC++ oferece, no nível de linguagem, a utilização direta dos mecanismos de blocos paralelos, laços paralelos e paralelismo não estruturado, como mostra a figura 3.1.

```
par
{
  tarefa_1();
  tarefa_2();
  ...
  tarefa_n();
};

parfor (i=0; i<n; i++)
{
  tarefa_1();
  tarefa_2();
  ...
  tarefa_m();
};

...
spawn b();
...
```

FIGURA 3.1 - Mecanismos de expressão de paralelismo em CC++.

3.2.2 Sincronização

Para sincronização de tarefas, CC++ disponibiliza a declaração de variáveis síncronas e métodos atômicos, conforma ilustra a figura 3.2.

```
void funcaoQualquer()
{
  sync int a;
  ...
  a == 0; // bloqueia o processo se a ainda está indefinida
  ...
};

class Semaforo
{
  public:
    atomic void p();
    atomic void v();
};
```

FIGURA 3.2 - Mecanismos de sincronização em CC++.

Mesmo que CC++ não implemente métodos concorrentes, a utilização de métodos atômicos pode ser útil quando do uso de objetos com os mecanismos de concorrência apresentados.

A semântica das variáveis síncronas não é integrada à de métodos atômicos, portanto o bloqueio de um processo em uma variável síncrona declarada dentro de um método atômico não libera o mutex implícito associado ao objeto. Essa característica deve ser levada em conta pelo programador para evitar a ocorrência de *deadlocks.*

3.3 COOL

A linguagem COOL (*Concurrent Object Oriented Language*) [CHA96] foi desenvolvida na Universidade de Stanford, sendo voltada à programação com memória compartilhada. COOL encaixa-se no modelo de tarefas concorrentes, mas procura integrar os mecanismos de expressão de paralelismo e sincronização com a orientação a objetos.

3.3.1 Expressão de paralelismo

O paralelismo é expressado, em COOL, por meio de *funções paralelas*, que podem ser tanto procedimentos normais como métodos de objetos. A figura 3.3 ilustra a utilização desse mecanismo.

3.3.2 Sincronização

Como mostra o exemplo, funções paralelas não podem retornar um valor que não seja um ponteiro do tipo `condH`. O valor retornado representa um evento que permite, posteriormente, verificar se a execução da função já foi completada ou mesmo aguardar pelo seu final. A utilização desse mecanismo pode ser comparada àquela de *futures*, mas

```
parallel condH funcaoQualquer();

class Exemplo
{
  public:
    parallel void foo();
    parallel condH* bar();
};
```

FIGURA 3.3 - Mecanismos de expressão de paralelismo em COOL.

COOL não permite que se possa associar um valor de retorno a um evento `condH`, devendo o programador utilizar-se de variáveis globais ou passar parâmetros por referência para esse fim.

COOL permite também a declaração de variáveis condicionais através da palavra-chave `cond`. Uma variável desse tipo disponibiliza as operações `release()`, que implementa um *wait*, `signal()`, `broadcast()`, que implementa um *signal_all()*, e `wait()`, que implementa um *wait* mas não libera o mutex associado.

Para métodos de objetos, COOL permite a adição das palavras-chave `mutex` e `nonmutex`. Métodos `mutex` implementam exclusão mútua normalmente; métodos `nonmutex` são não exclusivos entre si.

COOL ainda oferece um mecanismo de sincronização entre várias funções paralelas, como mostra a figura 3.4. A execução do programa só prossegue após o término de todas as funções contidas no bloco.

```
waitfor
{
  a.foo();
  b.bar();
  e.baz();
};
```

FIGURA 3.4 - Mecanismo de sincronização em COOL.

3.4 ICC++

ICC++ (Illinois Concert C++) [CHI96] é uma linguagem para programação paralela desenvolvida como parte do projeto *Concert* da Universidade de Illinois. Sua utilização é voltada à exploração de paralelismo de grão fino, enquadrando-se no modelo de tarefas concorrentes.

3.4.1 Expressão de paralelismo

Os mecanismos de expressão de paralelismo em ICC++ são os mesmos oferecidos em CC++, ou seja, blocos paralelos, laços paralelos e paralelismo não estruturado. A figura 3.5 mostra a sintaxe empregada na linguagem.

```
conc
{
  tarefa_1();
  tarefa_2();
  ...
  tarefa_n();
};

conc for (i=0; i<n; i++)
{
  tarefa_1();
  tarefa_2();
  ...
  tarefa_m();
};

...
spawn b();
...
```

FIGURA 3.5 - Mecanismos de expressão de paralelismo em ICC++.

Diferentemente de CC++, ICC++ utiliza a mesma palavra-chave (`conc`) para expressar blocos e laços paralelos, neste último como um prefixo. Sua utilização também é permitida em outros comandos iterativos como `while` e `do`.

3.4.2 Sincronização

ICC++ procura livrar o programador da tarefa de se preocupar com a sincronização sobre dados compartilhados, apresentando um modelo de consistência automática de dados. Assim, a sincronização é implicitamente incluída na execução do programa quando a dependência de dados a exige.

Tomando o exemplo dado, uma tarefa t_j depende de uma tarefa t_i se algum identificador que aparece em ambas recebe um valor em alguma delas; ou, se t_i contém alguma operação de desvio (`break`, `continue` ou `goto`). Em qualquer desses casos, garante-se que t_i executa completamente antes de t_j.

3.5 Java

Java [LEA96, OAK97] é uma linguagem relativamente recente que vem, a cada dia, conquistando mais adeptos, principalmente por sua ligação com a Internet. Independentemente disso, Java apresenta conceitos bastante interessantes no âmbito de programação paralela, podendo ser enquadrada no modelo de tarefas concorrentes.

3.5.1 Expressão de paralelismo

Um novo fluxo de execução, em Java, é criado pela instanciação de uma classe derivada de *Thread*, ou que implemente a interface *Runnable*. A figura 3.6 ilustra a declaração

e criação de um objeto *Thread*.

```
class Exemplo extends Thread
{
  public foo() ...;
  public bar() ...;
};
...
new Exemplo(); // criação de um novo thread
```

FIGURA 3.6 - Expressão de paralelismo em Java.

A superclasse *Thread* oferece os métodos `start()`, `stop()`, `suspend()` e `resume()`, que permitem controle direto sobre a execução do *thread*.

A execução sucessiva dos métodos de um objeto *Thread* tem caráter seqüencial, mas diferentes *threads* podem, naturalmente, executar os métodos de um mesmo objeto qualquer de forma concorrente entre si.

3.5.2 Sincronização

A sincronização entre diferentes *threads*, em Java, é feita pelo uso de monitores. Todo e qualquer objeto em Java possui os métodos `wait()`, `notify()` (*signal*) e `notifyAll()` (*signal_all*), de modo que cada um possui uma única variável condicional implícita. A declaração explícita de variáveis condicionais não é permitida.

Métodos declarados `synchronized` têm comportamento atômico e identificam os procedimentos de acesso ao monitor, como mostra a figura 3.7.

```
class Semaforo
{
  public synchronized void p() {...};
  public synchronized void v() {...};
};
```

FIGURA 3.7 - Sincronização de métodos em Java.

Java permite, ainda, a utilização explícita de blocos com exclusão mútua, também através da palavra-chave `synchronized`, como mostra a figura 3.8. O bloco recebe um parâmetro que identifica o objeto cujo mutex implícito será usado no estabelecimento da exclusão mútua.

```
synchronized (this)
{
  counter--;
};
```

FIGURA 3.8 - Bloco atômico em Java.

3.6 Panda

A linguagem Panda [ASS93], desenvolvida pela Universidade de Kaiserslautern, Alemanha, é uma linguagem de programação associada a um ambiente de execução paralela (*kernel*), e é uma das poucas linguagens paralelas baseadas em C++ que seguem o modelo de objetos multi-ativos.

3.6.1 Expressão de paralelismo

De forma semelhante a Java, um objeto ativo em Panda é definido por herança da classe *UserThread*, como mostra a figura 3.9.

```
class Exemplo:  UserThread
{
  public:
    void code();
    void foo();
    void bar();
};
...
new Exemplo(); // criação de um novo thread
```

FIGURA 3.9 - Expressão de paralelismo em Panda.

Os métodos de um objeto ativo são, implicitamente, potencialmente concorrentes. O método `code()` é usado como construtor para a parte ativa de um objeto, consistindo no fluxo de execução original.

3.6.2 Sincronização

A sincronização em Panda é obtida pelo uso de objetos que implementam semáforos e sinais. Adicionalmente, o pré-compilador da linguagem pode ser instruído a adicionar um semáforo a todos os métodos de uma classe, tornando-os atômicos, fazendo o objeto comportar-se como um monitor.

3.7 Presto

A linguagem Presto [FUR95] foi desenvolvida pela Universidade de Washington e baseia a expressão de paralelismo pela utilização explícita de *threads*, encaixando-se também no modelo de tarefas concorrentes.

3.7.1 Expressão de paralelismo

A base de execução de um programa em Presto é a classe *Thread* que, quando instanciada, cria um novo fluxo de execução. Um objeto dessa classe pode, então, ser utilizado para a invocação concorrente de métodos de outros objetos, como mostra a figura 3.10.

```
class Exemplo
{
  public:
    void foo (int a);
    void bar ();
};
...
e = new Exemplo ();
t = new Thread ("exemplo");
t->start (e, e->foo, 5); // chama e->foo(5) de forma concorrente
```

FIGURA 3.10 - Expressão de paralelismo em Presto.

3.7.2 Sincronização

Presto disponibiliza as classes *Lock*, *Spinlock*, *Monitor* e *Condition* para o estabelecimento de sincronização entre invocações concorrentes. As duas primeiras classes oferecem os métodos `lock()` e `unlock()`, sendo que um processo que seja bloqueado por um objeto *Spinlock* não é desescalonado, ao contrário do que ocorre com a classe *Lock*. Um objeto da classe *Monitor* possui os métodos `entry()` e `exit()`. Objetos *Condition* oferecem os métodos `wait()` e `signal()`, e podem ser associados a um monitor.

A figura 3.11 ilustra o estabelecimento de uma seção crítica através de um objeto da classe *Lock*.

```
Lock l;
...
l.lock();
  ...  // seção crítica
l.unlock();
```

FIGURA 3.11 - Mecanismo de sincronização em Presto.

Presto oferece ainda um mecanismo explícito de *futures*; todo objeto *Thread* possui um método `willjoin()` que indica que a próxima invocação de método possui um valor de retorno. Um outro método `join()` faz com que o valor seja obtido, bloqueando o processo se necessário. A figura 3.12 ilustra esse mecanismo.

```
t->willjoin();
t->start(...);
...
ObjAny val = t->join();
```

FIGURA 3.12 - Uso explícito de *futures* em Presto.

3.8 Considerações finais

Nota-se que a maioria das linguagens aqui apresentadas adota o modelo de tarefas concorrentes para expressão de paralelismo, possivelmente procurando facilitar o

entendimento da linguagem ao programador, visto que esse modelo é o mais intuitivo. O modelo de objetos ativos, porém, é adotado por diversas outras linguagens, como ABC++ [O'FA96], C++// [CAR96], CHARM++ [KAL96], UC++ [WIN96], entre outras. Tais linguagens não estão relacionadas na apresentação justamente por não apresentarem recursos para exploração de paralelismo fino, seguindo o modelo mono-ativo e portanto dedicando-se ao processamento distribuído. Outras linguagens como Concurrent C++ e LITP-C++ [FUR95] foram omitidas por apresentarem uma sintaxe de programação muito diferente daquela usualmente empregada em C++, fugindo dos objetivos principais do trabalho.

A tabela 3.1 resume os recursos oferecidos pelas linguagens em relação à expressão de concorrência de granularidade fina e sincronização.

TABELA 3.1 - Resumo das características das linguagens apresentadas.

Linguagem	Concorrência	Sincronização
CC++	`par`, `parfor`, `spawn`	variáveis `sync` métodos `atomic`
COOL	métodos `parallel`	`waitfor` métodos `mutex` e `nonmutex` variáveis `condH`
ICC++	`conc`	implícita
Java	classe *Thread*	monitores
Panda	métodos concorrentes	pré-compilador semáforos e sinais
Presto	classe *Thread*	classes *Lock, Spinlock, Monitor* e *Condition*

Este capítulo apresentou diversas linguagens paralelas da atualidade que empregam a orientação a objetos como paradigma de programação. A maneira pela qual paralelismo e sincronização são oferecidos ao usuário depende do modelo de programação e das características próprias adotadas por cada uma. Esta apresentação permite, posteriormente, um melhor entendimento e justificativa para a adoção das características do modelo de paralelismo fino proposto.

O próximo capítulo apresenta o modelo de distribuição de objetos de DPC++, enfocando suas principais características no nível de linguagem de programação e no nível operacional. Na seqüência é apresentado o modelo proposto.

4 O modelo de distribuição de objetos de DPC++

Este capítulo faz uma apresentação do modelo de distribuição de objetos [CAV94] empregado na linguagem DPC++, com o objetivo de fornecer um embasamento teórico do contexto sobre o qual o trabalho é realizado e melhor ilustrar a motivação para o seu desenvolvimento.

4.1 Características gerais de DPC++

DPC++ é uma extensão da linguagem C++ desenvolvida no Instituto de Informática da Universidade Federal do Rio Grande do Sul. Seu objetivo é "unir as facilidades da programação segundo o paradigma de orientação a objetos com os benefícios do processamento distribuído, oferecendo uma ferramenta que apresente facilidades de programação para ambientes distribuídos" [CAV93].

A figura 4.1 mostra uma visão geral de uma aplicação DPC++. Do ponto de vista do programador, uma aplicação DPC++ em nada difere de uma aplicação orientada a objetos convencional, sendo composta por diversos objetos que executam suas tarefas individualmente e comunicam-se através de chamadas de métodos. A diferença entre uma aplicação DPC++ e uma aplicação convencional está no fato de que, em DPC++, dois objetos podem ser instanciados em nodos distintos, potencialmente paralelizando a execução de suas tarefas. Em outras palavras, uma aplicação DPC++ é distribuída entre os nodos de processamento, sendo que um objeto é a unidade de distribuição.

Naturalmente, nem todos os objetos de uma aplicação DPC++ são distribuídos[1]. A decisão sobre quais objetos executar de forma distribuída cabe ao programdor no momento do projeto e da codificação da aplicação.

A transparência dos mecanismos utilizados na distribuição de uma aplicação DPC++ é uma das principais características da linguagem, ficando o programador livre da utilização de primitivas explícitas de comunicação ou sincronização entre nodos distintos, podendo concentrar sua atenção no desenvolvimento da aplicação. Um dos aspectos positivos de DPC++, em função dessa característica, é que ele favorece o desenvolvimento de aplicações orientadas a objetos em geral, nas quais a exploração de paralelismo não é a principal finalidade.

Durante o processo de compilação da aplicação distribuída, DPC++ trata de adicionar as primitivas necessárias para a comunicação entre objetos distribuídos através da rede de comunicação entre os nodos. Uma chamada remota de método é processada como um mecanismo de RPC, sendo os parâmetros e os valores de resposta enviados em mensagens através da rede.

Seguindo o modelo de objetos ativos, um objeto distribuído em DPC++ possui um espaço de endereçamento próprio e uma interface de acesso, que corresponde ao conjunto de métodos públicos especificados na declaração da classe. A comunicação entre objetos distribuídos ocorre somente pela interface de acesso definida, ou seja, pela chamada de métodos. O espaço de endereçamento de um objeto distribuído não é acessível externamente. Objetos não-distribuídos instanciados localmente fazem parte do espaço de

[1] O termo *objeto distribuído* refere-se a um objeto capaz de ser instaciado remotamente, e não significa que um mesmo objeto seja particionado, ou "distribuído", entre os nodos da rede, como se pode vir a entender.

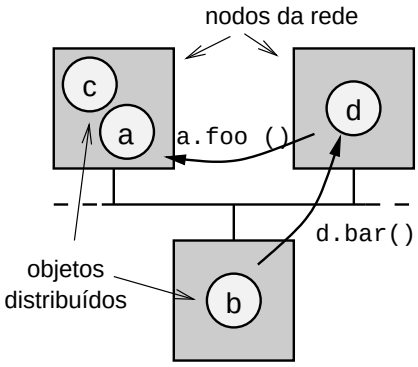

FIGURA 4.1 - Visão geral de uma aplicação DPC++.

endereçamento de um objeto distribuído, sendo chamados de objetos *passivos*.

Embora não seja exigido pelo modelo de objetos ativos, os objetos distribuídos em DPC++, assim como na maioria das linguagens que seguem esse modelo [ÁVI97], têm fluxo de controle interno único, de forma que apenas um método é executado por vez, sendo as requisições simultâneas enfileiradas e atendidas na ordem em que foram recebidas. Isto mantém o comportamento observado em aplicações seqüenciais tradicionais, e portanto agiu como fator motivador para a adoção generalizada dessa característica.

4.2 Nível de linguagem de programação

4.2.1 Declaração de classes distribuídas

O nível de linguagem de DPC++ é bastante conciso, introduzindo uma quantidade mínima de modificações sintáticas, o que faz com que a linguagem possa ser facilmente assimilada.

A distribuição de uma aplicação DPC++ é determinada pela identificação das classes distribuídas, ou seja, classes cujas instâncias serão objetos distribuídos. A declaração de tais classes é feita através da palavra-chave `dclass`, conforme ilustra o exemplo na figura 4.2.

```
dclass Server
{
  private:
    int numTasks;

  public:
    Server ();
    int requestTask ();
}
```

FIGURA 4.2 - Exemplo de declaração de uma classe de objeto distribuído.

4.2.2 Criação e término de objetos distribuídos

Objetos distribuídos são criados, à medida em que a aplicação é executada, por declaração estática ou por criação dinâmica, através do operador `new`. Seguindo a semântica normal de C++, o método construtor de um objeto distribuído é executado quando da sua criação. Analogamente, a execução do método destrutor, seja de forma automática ou pelo operador `delete`, declara o término da execução de um objeto distribuído.

Em ambos os casos, a sintaxe usada é a mesma de C++, o que contribui para o melhor entendimento do programa e fácil assimilação de DPC++.

4.2.3 Sincronização

A interação entre objetos distribuídos, como já mencionado, ocorre somente por meio de chamada de métodos. Portanto, a sincronização entre eles é obtida através desse mecanismo. A característica síncrona ou assíncrona da comunicação depende do tipo de retorno declarado para o método. Para métodos cujo tipo de retorno é `void`, a chamada é feita de forma assíncrona, permitindo que o objeto que originou a invocação possa continuar seu processamento imediatamente. Quando um método possui um valor de retorno, o objeto chamador é bloqueado até que a informação desejada seja devolvida pelo objeto remoto. Uma terceira modalidade de comunicação é disponibilizada pelo uso da palavra-chave `confirmation` na declaração de um método. Neste caso, o objeto que chamou o método também é bloqueado, mas somente até que o recebimento da mensagem seja confirmado pelo nodo remoto. Este tipo de método tem a finalidade de introduzir um certo grau de tolerância a falhas em uma aplicação DPC++.

Essas duas palavras-chave são as únicas alterações introduzidas por DPC++ à sintaxe padrão de C++. Algumas características, entretanto, como a utilização de variáveis de classe e passagem de ponteiros como parâmetros, não são suportadas.

4.3 Nível operacional

4.3.1 Estrutura de uma aplicação DPC++

O modelo de objetos distribuídos de DPC++ define o comportamento de uma aplicação em seu nível operacional. A estrutura funcional de uma aplicação DPC++ é apresentada na figura 4.3, que introduz dois novos elementos:

- o *Diretório,* reponsável pela criação de objetos distribuídos em uma aplicação DPC++;
- o *cluster*, que é a implementação de um objeto distribuído.

O Diretório é um objeto distribuído criado automaticamente no início da aplicação. Sempre que a criação de um novo objeto distribuído é solicitada, essa requisição é enviada ao Diretório. Este, aplicando técnicas de balanceamento de carga, escolhe um nodo apropriado e efetua a instanciação do objeto desejado. O endereço do novo objeto é então devolvido àquele que solicitou a criação, e a aplicação prossegue sua execução.

O *cluster* é o elemento responsável por introduzir a característica ativa em um objeto distribuído, suprindo-o com capacidade de processamento e área de memória. Em termos práticos, um *cluster* é um processo do sistema operacional. Além de manter o próprio

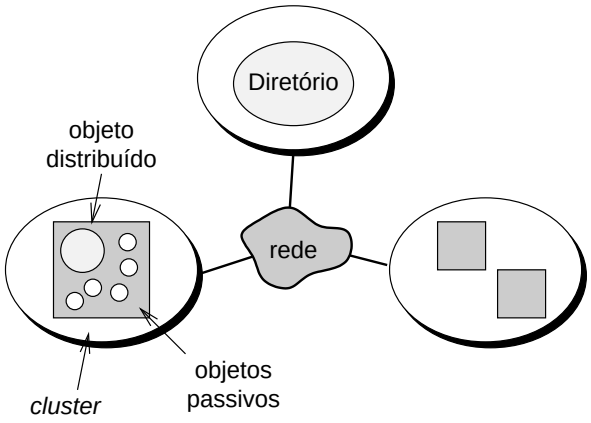

FIGURA 4.3 - Estrutura funcional de uma aplicação DPC++.

objeto distribuído, o espaço de endereçamento de um *cluster* é usado para a manutenção dos objetos passivos criados no decorrer da aplicação.

4.3.2 Estrutura de um objeto distribuído

A figura 4.4 mostra a organização interna de um objeto distribuído no modelo de DPC++.

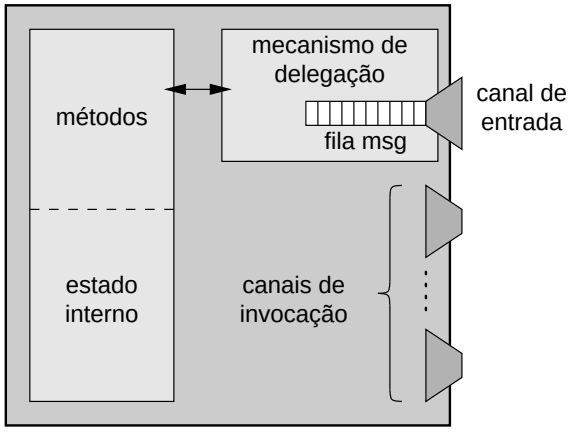

FIGURA 4.4 - Organização interna de um objeto distribuído.

Além dos métodos e do estado interno, definido por seus atributos, um objeto distribuído apresenta um mecanismo de delegação, associado a um canal de entrada, e vários canais de invocação. Esse mecanismo de delegação é responsável por gerenciar requisições remotas para execução de métodos, recebidas através do canal de entrada. Cada requisição contém uma identificação do método a ser executado e os parâmetros necessários para a execução. O mecanismo de delegação extrai esses elementos e executa a invocação. Se for necessário enviar uma reposta ao objeto remoto (para métodos não-`void`), ela é enviada através de um dos canais de invocação. Cada um desses canais permite a comunicação com um objeto distribuído.

4.3.3 Objetos procuradores

A comunicação entre objetos distribuídos não ocorre de forma direta. Ao invés disso, cada objeto distribuído comunica-se com outro através de um *objeto procurador*.

Um objeto procurador é um objeto passivo que faz parte do espaço de endereçamento de um objeto distribuído e que consiste em uma "imagem" de outro objeto distribuído. A finalidade de um procurador é encapsular os procedimentos de envio e recebimento de mensagens entre objetos distribuídos, incluindo a comunicação com o objeto Diretório.

Quando um objeto distribuído cria outro objeto distribuído, o resultado é, na verdade, a criação de um objeto procurador. Este, por sua vez, envia ao Diretório a solicitação de criação daquele objeto. A resposta recebida, que consiste no endereço do objeto criado, é guardada pelo procurador para o envio de futuras requisições.

Esse procedimento é completamente transparente ao objeto distribuído que originou a criação, pois a interface de acesso do procurador é exatamente igual à do objeto distribuído que ele representa. A implementação de cada método do procurador consiste em criar e enviar uma mensagem ao objeto real, requisitando a execução do método correspondente.

A figura 4.5 ilustra a comunicação entre dois objetos distribuídos através de um procurador, mostrando o caminho virtual que se intenciona e o caminho realmente tomado pela invocação.

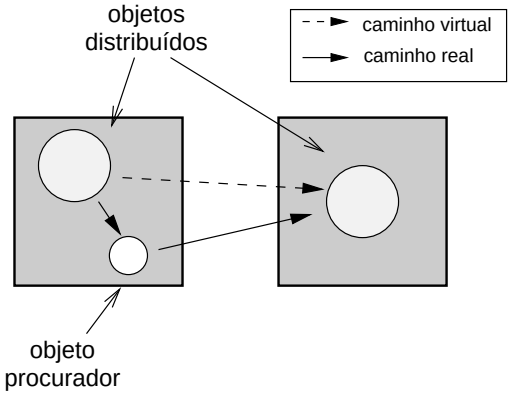

FIGURA 4.5 - Comunicação entre objetos distribuídos através de um objeto procurador.

4.4 Considerações finais

O paradigma de programação orientada a objetos mostra-se bastante adequado a ambientes distribuídos, principalmente porque permite um dimensionamento adequado do grão de paralelismo a ser empregado na aplicação. Desta forma, a utilização da orientação a objetos no modelo de distribuição de DPC++ atende aos objetivos da linguagem no sentido de proporcionar um modelo de programação adequado ao desenvolvedor de aplicações distribuídas.

Este capítulo apresentou, assim, o modelo de distribuição de objetos empregado na linguagem DPC++, destacando suas principais características e fornecendo um embasamento para a apresentação da extensão proposta, feita a seguir.

5 O modelo de paralelismo de grão fino proposto

Vistos o modelo original de distribuição de objetos em DPC++ e a forma como a expressão de paralelismo é disponibilizada em linguagens da atualidade, este capítulo apresenta o modelo proposto de paralelismo de grão fino que é tema desta pesquisa.

A motivação para a extensão do modelo de objetos distribuídos de DPC++ surgiu, principalmente, da crescente disponibilidade de sistemas computacionais que oferecem a exploração de um nível mais fino de paralelismo. Como já citado, o modelo original de DPC++ mostra-se bastante adequado à utilização em ambientes distribuídos, considerando o nível médio a grosso de paralelismo oferecido em tais ambientes; porém, o surgimento de tecnologias rápidas de interconexão como Fast Ethernet e SCI, aliado ao barateamento da construção de máquinas multiprocessadas, levou à implementação de arquiteturas cujo potencial de exploração de paralelismo fino não pode ser ignorado. E, para que as aplicações DPC++ possam tirar proveito também desse potencial, a extensão do modelo faz-se necessária.

A principal meta desta proposta, portanto, é introduzir no modelo de objetos distribuídos de DPC++ a capacidade de exploração de paralelismo de grão fino de forma eficiente, procurando-se, adicionalmente, firmar um compromisso entre respeitar as características principais da linguagem e manter fidelidade às idéias expostas no capítulo anterior em relação à adequação da ferramenta de programação paralela.

Assim como o modelo original de DPC++, a extensão proposta também pode ser analisada em termos de nível de linguagem e nível operacional. Sua apresentação, portanto, seguirá essa classificação, conforme a seguir.

5.1 Nível de linguagem

5.1.1 Introdução de concorrência entre métodos

Como visto nos mecanismos e linguagens apresentados anteriormente, existem diversas formas pelas quais se pode introduzir a expressão de paralelismo em uma linguagem ou biblioteca de programação concorrente. Adicionalmente, a maneira como um ou outro mecanismo é integrado à linguagem e ao modelo de programação, no caso a orientação a objetos, é de grande importância para que sua utilização por parte do programador seja eficiente e de assimilação natural. Conforme afirmado por Wyatt et al. [WYA92], a não ser que concorrência, sincronização e comunicação sejam cuidadosamente integradas, uma linguagem paralela orientada a objetos pode se tornar ineficiente e difícil de ser utilizada.

Nesse sentido, baseando-se nos estudos apresentados nos capítulos anteriores, decidiu-se pela introdução de concorrência entre os métodos de um mesmo objeto distribuído como a forma mais adequada de explorar paralelismo de grão fino em DPC++.

Adicionalmente, tal expressão de concorrência é implícita [ÁVI99], ou seja, não requer que os métodos sejam declarados com algum tipo de modificador, objetivando conferir um caráter mais natural à semântica do mecanismo. O assunto, já comentado anteriormente, é melhor explorado na análise feita ao final do capítulo.

A declaração de uma classe distribuída mantém a mesma sintaxe já apresentada no modelo original, como mostra o exemplo da figura 5.1.

No exemplo dado, os métodos `getCurrentSpeed()` e `accelerate()` podem ser

```
dclass Car
{
  private:
    int speed;

  public:
    Car ();
    int getCurrentSpeed ();
    void accelerate ();
}
```

FIGURA 5.1 - Declaração de uma classe distribuída no novo modelo.

executados em paralelo se forem simultaneamente chamados. É justamente por essa característica que o modelo proposto busca seu principal objetivo, qual seja explorar eficientemente o grau de paralelismo fino disponibilizado por uma máquina paralela. Este nível de paralelismo se encaixa adequadamente no modelo de programação, visto que pode ser bem dimensionado na implementação de um único método.

A execução concorrente ocorre somente quando os métodos são chamados a partir de outros objetos distribuídos; se a chamada é feita internamente, ou seja, a partir de um outro método do mesmo objeto, a execução segue o comportamento seqüencial tradicional, objetivando melhor controle, por parte do programador, quando da implementação da classe.

Naturalmente, somente os métodos públicos de uma classe podem ser executados diretamente de forma concorrente, pois os demais não fazem parte da interface de acesso ao objeto. Métodos não públicos podem ser executados de forma concorrente entre si como conseqüência de sua invocação por parte daqueles constantes da interface definida na classe.

5.1.2 Criação e destruição de objetos distribuídos

Os mecanismos de criação e destruição de objetos distribuídos permanecem inalterados no novo modelo, mantendo a mesma sintaxe de utilização já empregada em DPC++, através de alocação automática ou criação dinâmica pelos operadores `new` e `delete`.

A modalidade de execução dos métodos construtores e destrutores é sempre síncrona. Além disso, a invocação do método destrutor de um objeto distribuído aguarda pelo término de todas as invocações de métodos ainda em andamento, visto a possibilidade de concorrência interna. Essa característica é necessária para que se possa projetar corretamente a aplicação, e pode ser usada também como mecanismo de sincronização entre objetos distribuídos.

5.1.3 Sincronização

A possibilidade de execução concorrente dos métodos de um mesmo objeto distribuído estabelece também a possibilidade de condições de corrida sobre os atributos de tal objeto, o que pode levar o algoritmo a estados inconsistentes. Para resolver esse problema, o novo modelo oferece um mecanismo de sincronização por monitores.

A semântica do monitor é integrada ao objeto distribuído, de forma semelhante à feita em Java. Os procedimentos de acesso ao monitor são representados por um subcon-

junto dos métodos públicos da classe, identificados pelo modificador `atomic`, já que esses procedimentos devem ter comportamento atômico. Diferentemente de Java, entretanto, o modelo proposto permite a declaração de variáveis condicionais através da palavra-chave `cond`. As operações sobre variáveis condicionais são oferecidas com a mesma sintaxe de chamadas de métodos em objetos estáticos, como, por exemplo, em `empty.wait()`.

A figura 5.2 ilustra a utilização do mecanismo de sincronização proposto na implementação de um *buffer* limitado, onde vários processos produzem e consomem elementos de forma concorrente.

```
class BoundedBuffer
{
  private:
    Data buffer[MAXITEMS];
    int count = 0, in = 0, out = 0;
    cond empty, full;

  public:
    atomic void produce (Data item) {
      while (count == MAXITEMS)
        full.wait();
      buffer[in] = item;
      in = in ⊕ 1; count++;
      empty.signal();
    };

    atomic Data consume () {
      Data item;

      while (count == 0)
        empty.wait();
      item = buffer[out];
      count--; out = out ⊕ 1;
      full.signal();

      return (item);
    };

    int getItemCount () {
      return (count);
    };
}
```

FIGURA 5.2 - Implementação de um *buffer* limitado usando a nova sintaxe.

A classe `BoundedBuffer` apresenta dois métodos principais, `produce()` e `consume()`, que implementam as tarefas de produzir e consumir um item, respectivamente. Pela característica de paralelismo implícito recém apresentada, esses dois métodos seriam potencialmente concorrentes entre si; porém, por atuarem ambos sobre a variável `count`, que controla o número de elementos no *buffer*, estabelecendo portanto uma condição de corrida, os dois devem ser declarados com o modificador `atomic`, de

forma que, em um determinado instante de tempo, no máximo um processo esteja atuando sobre o *buffer.*

Um terceiro método, `getItemCount()`, é disponibilizado para que se possa saber a quantidade de elementos no *buffer* em um dado momento. Como este método simplesmente lê o valor da variável `count`, sua declaração é feita normalmente, podendo, portanto, ser executado em paralelo com os demais.

Quando da execução do método `produce()`, por exemplo, o algoritmo verifica se existe a possibilidade de produção de mais um item (ou seja, se o *buffer* não está cheio). Em caso negativo, o processo é bloqueado através da execução de um *wait*. Conforme especifica a semântica de monitores, essa operação libera o mutex associado ao objeto, permitindo que, futuramente, outro processo venha a consumir um elemento, liberando aquele bloqueado anteriormente.

A semântica adotada para a operação `signal()` é a chamada *signal and continue*, conforme definido por Andrews [ALT91]. A principal justificativa é a própria semântica da operação. Mesmo que um novo processo esteja sendo explicitamente recolocado em execução, parece mais intuitivo que o processo que executou a operação prossiga até o final do método. Por outro lado, a condição pela qual o processo que estava bloqueado foi liberado pode não mais ser verdadeira quando ele finalmente for escalonado para execução, o que força a utilização de um `while` ao invés de um simples `if` quando a condição é testada. A justificativa final para adoção da semântica *signal and continue* é que as demais exigem que o processo sendo liberado seja colocado imediatamente em execução, o que requer influência sobre o escalonador, nem sempre possível.

O exemplo dado ilustra bem o objetivo de oferecer uma sintaxe de programação adequada e com bom poder de expressão. A possibilidade de declaração de variáveis condicionais permite uma especificação completa do funcionamento do algoritmo, identificando claramente as condições sob as quais os processos são bloqueados ou liberados.

Considerando-se a sintaxe da linguagem Java, por exemplo, que não oferece a declaração de variáveis condicionais, não se pode saber, quando da execução de um *signal*, se o processo sendo liberado havia sido bloqueado ao tentar produzir ou consumir, o que dificulta o entendimento do algoritmo.

Comparando o mecanismo proposto com outra forma de sincronização, as figuras 5.3 e 5.4 mostram a implementação de um semáforo utilizando, respectivamente, a sintaxe proposta para DPC++ e o mecanismo de variáveis síncronas apresentado por CC++. Como este último não é integrado ao modelo de orientação a objetos, o bloqueio de um processo por uma leitura em uma variável síncrona não libera o mutex adquirido implicitamente pelo método declarado `atomic`. Isso força a utilização de um método auxiliar, dificultando o projeto e o entendimento do algoritmo. Além disso, o controle de quais processos bloquear ou desbloquear deve ser feito pelo usuário (através de uma fila, no exemplo dado), o que, em DPC++, é automaticamente gerenciado pelo mecanismo de variáveis condicionais.

5.2 Nível operacional

A extensão proposta, em seu nível operacional, modifica a estrutura interna de um *cluster* e de um objeto distribuído pela introdução de duas características principais: a utilização de múltiplos fluxos de execução, que implementam a concorrência entre os métodos, e a possibilidade de um *cluster* abrigar mais de um objeto distribuído.

```
class Semaphore
{
  private:
    cond locked;
    int count;

  public:
    atomic void p () {
      count--;
      if (count < 0)
        locked.wait();
    };

    atomic void v () {
      count++;
      locked.signal();
    };
}
```

FIGURA 5.3 - Implementação de um semáforo em DPC++.

5.2.1 Nova estrutura de um objeto distribuído

A figura 5.5 mostra a estrutura interna de um objeto distribuído segundo o novo modelo. As modificações em relação ao modelo original concentram-se no mecanismo de delegação responsável pelo recebimento de requisições remotas para execução de métodos.

Ao receber uma nova requisição, o mecanismo de delegação imediatamente cria um novo fluxo de execução para atendê-la. Se diversas requisições são recebidas sucessivamente, elas passam a ser tratadas em paralelo pelos diversos fluxos de execução criados. Como uma requisição corresponde a uma chamada de método, este mecanismo é o que efetivamente implementa a execução concorrente dos métodos de um objeto distribuído.

Como cada requisição é tratada por um novo fluxo de execução, o fluxo inicial está sempre pronto a receber uma nova requisição, dispensando, portanto, o emprego de uma fila de mensagens[1].

5.2.2 Nova estrutura de um *cluster*

Assim como em um objeto distribuído, a estrutura interna de um *cluster*, mostrada na figura 5.6, é alterada no novo modelo. Um *cluster* passa, agora, a ser capaz de abrigar mais de um objeto distribuído e, em conseqüência, os objetos passivos por eles criados.

Diferentemente do objeto distribuído, a modificação na estrutura do *cluster* não é reflexo de nenhuma característica do nível de linguagem; ao contrário, cada objeto distribuído continua a ser completamente independente em relação aos demais. A modificação do *cluster* tem por objetivo um melhor aproveitamento dos recursos utilizados e também

[1] Apesar de haver uma pequena latência de criação de um novo fluxo de execução, os subsistemas de comunicação utilizados freqüentemente implementam o armazenamento de mensagens, portanto a possibilidade de perda de requisições pode ser desconsiderada, e a fila de mensagens, ao menos conceitualmente, pode ser eliminada.

```
class Semaphore
{
  private:
    int count;
    Queue<sync int *> q;

    atomic void check (sync int *go) {
      count--;
      if (count < 0)
        q.push (go);
      else
        *go = 1;
    };

  public:
    void p () {
      sync int passed;

      check (&passed);
      passed == 1;
    };

    atomic void v () {
      count++;
      if (count <= 0)
        *q.pop() = 1;
    };
}
```

FIGURA 5.4 - Implementação de um semáforo em CC++.

possibilitar uma implementação mais eficiente do modelo de distribuição de objetos, conforme explicado na análise feita no final do capítulo.

Uma possível desvantagem do fato de diferentes objetos distribuídos ocuparem o mesmo espaço de endereçamento está na possibilidade de interferência entre eles. Porém, as vantagens oferecidas por essa característica, conforme apresentado posteriormente, superam de maneira convincente essa possibilidade. Além disso, cabe salientar que, em algoritmos corretamente escritos, essa interferência não ocorre.

5.2.3 Criação de objetos distribuídos

O fato de um *cluster* poder abrigar mais de um objeto distribuído implica que o objeto Diretório, ao criar um novo objeto, não precise necessariamente disparar um novo *cluster*, mas somente criar o objeto em um *cluster* já existente. Esse fato permite, também, um aprimoramento no desempenho da aplicação distribuída, a ser visto no final do capítulo. A decisão de criar ou não um novo *cluster* cabe, portanto, à técnica de balanceamento de carga empregada no objeto diretório.

Uma abordagem simples, empregada na implementação descrita no próximo capítulo, consiste na utilização de um único *cluster* em cada nodo, disparados no início

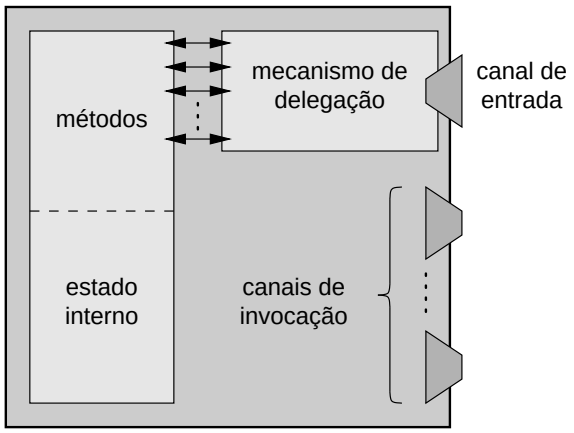

FIGURA 5.5 - Estrutura interna de um objeto distribuído no novo modelo.

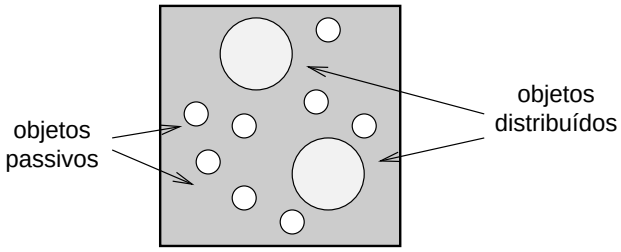

FIGURA 5.6 - Estrutura interna de um *cluster* no novo modelo.

da aplicação, de modo que o objeto Diretório não precise se preocupar em criar novos *clusters.* Esta abordagem também traz benefícios, apresentados na análise a seguir.

5.3 Análise do modelo proposto

O modelo de paralelismo de grão fino proposto tem como principal objetivo fazer com que as aplicações DPC++ possam eficientemente explorar tal nível de concorrência nas máquinas paralelas da atualidade. Essa exploração eficiente é atingida, basicamente, por dois fatores: a execução paralela dos métodos de um objeto distribuído, que por si só reduz o tempo necessário para a execução do algoritmo, e a possibilidade de sobreposição de operações de computação e comunicação. Ao permitir que, por meio de fluxos de execução adicionais, um objeto distribuído possa prosseguir com suas tarefas mesmo quando da necessidade de espera do complemento de uma operação de comunicação, o modelo reduz o tempo total gasto pelo objeto, contribuindo também para o desempenho da aplicação. As demais características do modelo, entretanto, trazem diversos outros benefícios.

Baseando-se nas conclusões sobre os estudos realizados e idéias levantadas em relação à expressão de paralelismo, apresentados inicialmente, optou-se pela implantação de concorrência entre os métodos de um mesmo objeto distribuído como a forma mais adequada de se introduzir paralelismo fino em DPC++. Como já mencionado, essa opção dispensa o uso de mecanismos adicionais, como blocos ou laços paralelos, e integra-se perfeitamente ao paradigma de programação.

Seguindo a idéia de estimular uma abordagem mais natural à programação paralela, a concorrência entre os métodos tem caráter implícito. Visto que a orientação a objetos, de uma forma geral, não impõe condições de exclusividade ou de precedência na execução ou declaração de métodos, não há, senão por razões de costumes, porque assumir que um objeto deva executar apenas um método de cada vez. Pelo contrário, um objeto pode naturalmente ser visto como um elemento capaz de executar uma certa quantidade de tarefas, não necessariamente de forma seqüencial.

A opção pelo uso do mecanismo de monitores para sincronização deve-se à perfeita integração desse mecanismo com o modelo de orientação a objetos e pelo poder de expressão que ele oferece, permitindo que os algoritmos paralelos sejam implementados de forma clara e simples. Além disso, o mecanismo de monitores é flexível o bastante para permitir a implementação de outros recursos de sincronização como mutexes e semáforos.

Uma das metas a serem atingidas no nível de linguagem diz respeito, conforme levantado anteriormente, à disponibilização de uma ferramenta adequada à programação paralela. A definição de métodos implicitamente concorrentes com sincronização explícita busca atingir essa meta, considerando que a sincronização de acesso sobre dados compartilhados é parte do paradigma de programação e, portanto, tarefa do programador. Esse fato pode ser exemplificado tomando-se, ainda, o exemplo do *buffer* limitado. A figura 5.7 mostra (a) a definição já apresentada da classe `BoundedBuffer` e (b) uma definição hipotética invertendo a semântica da especificação de sincronização dos métodos.

```
class BoundedBuffer
{
  public:
    atomic void produce ();
    atomic Data consume ();
    int getItemCount ();
}
```

(a)

```
class BoundedBuffer
{
  public:
    void produce ();
    Data consume ();
    par int getItemCount ();
}
```

(b)

FIGURA 5.7 - Comparação de semânticas para a implementação do *buffer* limitado.

No item (b) utiliza-se a palavra-chave `par` como modificador para o método `getItemCount()`, definindo que esse método pode ser executado em paralelo, e assume-se que os métodos sem identificação têm execução atômica.

A intenção de execução concorrente do método `getItemCount()` fica bem clara, em função do hábito de assumir o processamento seqüencial como normal. Porém, a declaração dos métodos `produce()` e `consume()` não deixa claro o caráter atômico de sua execução, já intrinsecamente suposto, por parte do programador, pela natureza paralela do algoritmo. Em outras palavras, dentro de um contexto de programação paralela torna-se mais natural assumir a concorrência do que a atomicidade implícita entre os métodos de um objeto.

O que se deseja, portanto, com a sintaxe definida é, dentro de um paradigma de programação paralela, oferecer uma ferramenta adequada ao programador.

O modelo procura, também, manter uma das principais características de DPC++ que é aproveitar a orientação a objetos para facilitar o desenvolvimento de aplicações distribuídas. A implementação de algoritmos paralelos pode ser mais ou menos facilitada,

em uma linguagem de programação, em função da identificação entre a natureza do algoritmo e os recursos oferecidos pela linguagem. Assim ocorre que certos algoritmos sejam mais facilmente implementados em uma linguagem do que em outra, como mostrado pela implementação do semáforo em CC++ e com a sintaxe proposta.

Antes de tentar abranger toda e qualquer classe de algoritmos paralelos, porém, o modelo proposto para DPC++ procura não só oferecer uma sintaxe simples e com grande poder de expressão para a implementação de concorrência mas também manter sua utilidade no desenvolvimento de aplicações em geral, onde o paralelismo não é o objetivo principal, e sim uma conseqüência do modelo de programação. O programador, desta forma, pode ocupar-se mais com os problemas intrínsecos do algoritmo e menos com questões de criação de processos ou de sincronização.

Em seu nível operacional, o modelo proposto traz, além dos objetivos principais de aproveitamento eficiente de recursos computacionais, uma série de benefícios. No modelo original, a criação de um novo objeto distribuído implica, em última análise, no disparo de um novo processo. Esse procedimento requer que o código executável do processo seja carregado do sistema de arquivos, que o processo seja criado, disparado, e que a inicialização do objeto distribuído seja executada. No novo modelo, a latência de criação de um novo objeto distribuído é consideravelmente reduzida, pois equivale ao disparo de um novo fluxo de execução (tarefa realizada pelo *cluster* quando a requisição é recebida) ao invés de toda a carga de um processo.

O fato de um *cluster* poder abrigar mais de um objeto distribuído implica em diversas vantagens. Considerando-se, como mencionado anteriormente, que cada nodo execute somente um *cluster*, decorre que, para que este último seja capaz de criar objetos de quaisquer das classes distribuídas, o processo que o implementa deve incorporar o código executável de todas essas classes; em conseqüência, esse processo pode ser o mesmo a ser disparado em todos os nodos, encaixando-se em um modelo de processamento SPMD (*Single Program Multiple Data*). Por ser um modelo freqüentemente adotado por bibliotecas e ambientes para programação paralela, como, por exemplo, MPI, essa característica contribui significativamente para a portabilidade de DPC++.

Outra vantagem decorrente dessa característica é relacionada à comunicação entre objetos distribuídos. Como todos os objetos de um mesmo *cluster* compartilham o mesmo espaço de endereçamento, esse recurso pode ser utilizado para implementar, de forma mais eficiente, a comunicação entre tais objetos. Também o código executável e os dados estáticos de uma classe distribuída são compartilhados por todas as suas instâncias, reduzindo a utilização de memória.

O consumo de recursos é bastante beneficiado pelo modelo proposto. Supondo cada objeto distribuído implementado por um processo independente, o número de objetos em um mesmo nodo seria limitado, na prática, por fatores de degradação do sistema. O uso de múltiplos fluxos de execução (*threads*), que consomem menos recursos e podem ser gerenciados no nível de usuário, permite a existência de um número maior de objetos sem acarretar no saturamento do ambiente de execução.

Como um último e importante aspecto positivo, visto que as principais extensões estão limitadas ao domínio de um *cluster*, o novo modelo não afeta a estrutura geral das aplicações DPC++, mantendo respeito às principais características da linguagem sem isentar-se de atingir seus principais objetivos.

Em resumo, a execução concorrente de métodos, auxiliada pela disponibilização de mecanismos de sincronização por monitores, é o meio escolhido para que uma aplicação DPC++ possa efetivamente aproveitar o paralelismo de grão fino oferecido por máquinas

paralelas. No nível operacional, o modelo baseia-se na criação de múltiplos *threads* como forma de implementação dessa característica. A esse respeito, o próximo capítulo detalha alguns aspectos mais importantes da implementação do novo modelo, de modo que se possa ter uma idéia mais concreta de seu funcionamento.

6 Implementação do modelo proposto

Este capítulo apresenta um detalhamento maior do modelo proposto em termos de implementação, com o objetivo de proporcionar um melhor esclarecimento e melhor fixação das idéias apresentadas no capítulo anterior.

6.1 O subsistema de execução — DECK

Com vistas a garantir melhor portabilidade e estrutura de implementação para a linguagem, o grupo de trabalho em DPC++ decidiu definir uma sub-camada de *software* que ofereça, independentemente do ambiente de execução, uma interface uniforme dos serviços básicos usados em uma aplicação distribuída. Desta forma surgiu DECK — *Distributed Executive Communication Kernel*.

6.1.1 Objetos DECK

A camada DECK define conceitualmente 4 objetos que representam a implementação de seus serviços mais básicos:

- *threads*, que permitem o disparo de fluxos de execução adicionais;
- *semáforos*, usados para sincronização entre vários fluxos de execução em um mesmo nodo;
- *mensagens*, usadas para comunicação, oferecendo serviços de empacotamento de dados como suporte a ambientes heterogêneos;
- *portas de comunicação*, para a troca de mensagens entre diferentes nodos de processamento.

A implementação de DPC++, portanto, é baseada na utilização de tais objetos, além do serviço de nomeação oferecido por DECK, no qual um servidor de nomes permite o cadastramento de portas de comunicação sob nomes bem conhecidos.

Além do oferecimento desses serviços, DECK propõe a implementação de funções mais complexas, como comunicação em grupo e suporte a tolerância a falhas. O anexo A apresenta uma descrição mais completa da camada e de um protótipo simplificado de DECK, implementado para a realização dos experimentos com o modelo.

6.2 Inicialização da aplicação

Como mencionado no capítulo anterior, a abordagem escolhida para criação de *clusters* é a de disparar todos no início da aplicação, o que proporciona latência reduzida na criação de novos objetos distribuídos e permite o emprego de um modelo de processamento SPMD. Desta forma, uma aplicação distribuída consiste de um único processo que é disparado em todos os nodos simultaneamente.

Note-se que esse processo inclui o código de todas as classes envolvidas na aplicação, incluindo o objeto Diretório, objetos procuradores e rotinas de inicialização.

A efetiva instanciação ou execução de um ou de outro depende, entretanto, da localidade, em relação aos nodos, e da própria seqüência de tarefas executadas pela aplicação.

O seguinte algoritmo descreve os passos tomados em cada nodo a partir do início da aplicação:

inicializar DECK
inicializar clusters
se nodo = 0
    criar objeto Diretório
    executar rotina `main()` *do usuário*
fim se
aguardar pelo término da aplicação
finalizar os clusters
finalizar DECK

O primeiro passo tomado em cada nodo é a inicialização da camada de serviços DECK; em seguida, passa-se à inicialização dos *clusters.* Esta tarefa, em comparação com o modelo original, pode ser entendida como o disparo de um *cluster* no nodo atual. Porém, como já existe um processo em execução, a tarefa é simplificada pela simples criação de um fluxo de execução adicional.

À exceção do nodo 0, todos os demais, a partir deste momento, apenas aguardam pelo término da aplicação. No nodo 0, efetua-se a criação do objeto Diretório e, em seguida, inicia-se a rotina principal definida pelo usuário. O final dessa rotina marca o fim da aplicação, quando os *clusters* são destruídos e a camada DECK finalizada.

O Diretório é sempre criado no nodo 0, de forma que o usuário possa ter controle sobre a localidade desse objeto, bastando, para isso, alterar a ordem de configuração das máquinas da rede. Esse controle pode ser útil para a realização de experimentos em relação ao desempenho da aplicação.

6.3 Implementação dos elementos do modelo

6.3.1 *Cluster*

A implementação de um *cluster* concentra as duas principais características de nível operacional do novo modelo, que são a utilização de múltiplos fluxos de execução e a existência de vários objetos distribuídos em seu espaço de endereçamento.

Visto que a função do mecanismo de delegação é a mesma em qualquer objeto distribuído, esse mecanismo é concentrado, na implementação, em um único elemento chamado de *dispatcher*. A figura 6.1 mostra a esquematização da implementação de um *cluster.*

O *dispatcher* consiste na parte ativa de um *cluster,* possuindo um fluxo de execução inicial associado a uma porta de comunicação. Esses dois elementos são criados na inicialização do *cluster.* A porta de comunicação é cadastrada no servidor de nomes de DECK, de modo que seja bem conhecida do restante da aplicação. Neste ponto o *cluster* não possui nenhum objeto distribuído em seu espaço de endereçamento, e o *dispatcher* está apto a receber requisições remotas.

O ciclo de vida do *dispatcher* é definido pelo seguinte algoritmo:

repita até (fim da aplicação)

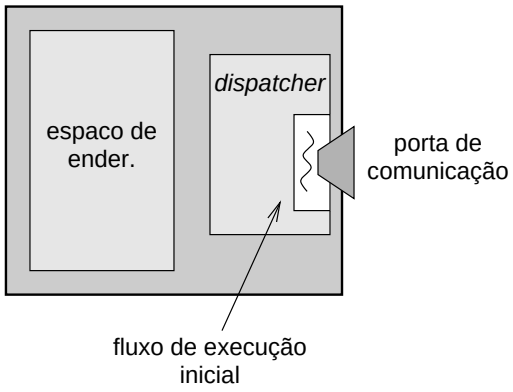

FIGURA 6.1 - Estrutura da implementação de um *cluster.*

receber requisição (req);
disparar thread para tratar requisição (req);
fim repita

Ao receber uma requisição, o *dispatcher*, agindo como o mecanismo de delegação de um objeto distribuído, imediatamente dispara um novo *thread* para tratá-la, e volta a escutar a porta de comunicação.

A figura 6.2 ilustra o processo de criação de um objeto distribuído. O *dispatcher* recebe do objeto Diretório uma mensagem solicitando a criação de um objeto da classe *A*, e cria um novo *thread* para atendê-la. Este *thread*, por sua vez, cria uma instância local da classe *A* e devolve ao Diretório uma referência a essa instância, finalizando sua execução. Esse objeto recém criado é o próprio objeto distribuído.

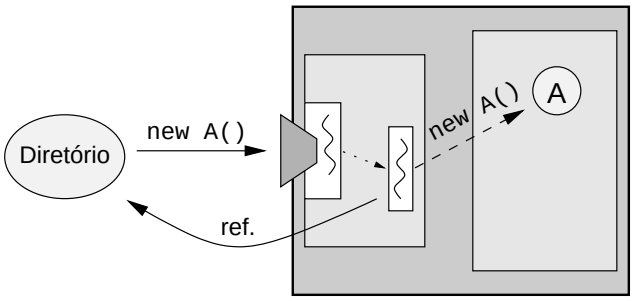

FIGURA 6.2 - Processo de criação de um objeto distribuído.

A referência ao objeto, bem como o endereço do *dispatcher* no qual ele foi instanciado, é devolvida pelo Diretório e armazenada pelo objeto procurador que originou o pedido de criação. Futuramente, quando da invocação de métodos no objeto *A*, essa referência deve ser incluída na mensagem de requisição, que é enviada ao *dispatcher* do nodo correspondente. A figura 6.3 ilustra o processo.

É dessa forma que um *cluster* é capaz de manter, em seu espaço de endereçamento, vários objetos distribuídos. Devido à referência aos objetos e ao emprego de múltiplos fluxos de execução no *dispatcher*, cada objeto distribuído pode ser identificado individualmente e o mecanismo de delegação de todos os objetos distribuídos de um *cluster* pode ser unificado sem prejuízo ao modelo. A figura 6.4 mostra uma última situação em que o *dispatcher* trata várias requisições simultâneas.

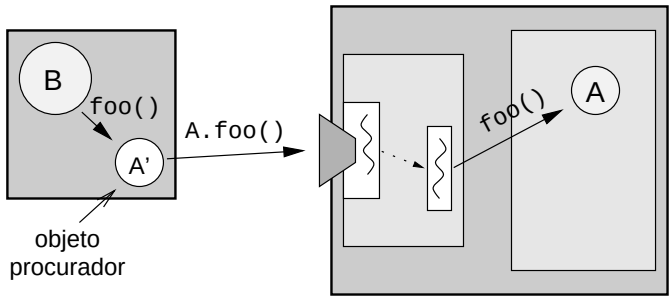

FIGURA 6.3 - Invocação de um método em um objeto distribuído.

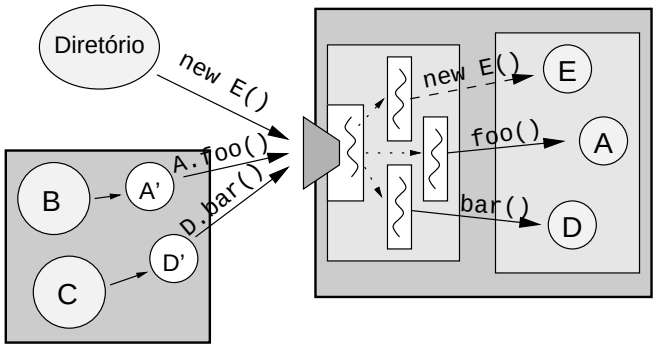

FIGURA 6.4 - Tratamento de múltiplas requisições pelo *dispatcher.*

6.3.2 Objetos distribuídos

A implementação de um objeto distribuído é feita, basicamente, pelo *cluster* e pelos objetos procuradores, pois são estes elementos que tornam ativa uma instância normal de um objeto. Entretanto, algumas características comuns à implementação de qualquer objeto distribuído motivaram a definição de uma classe básica *DistributedObject* que concentra a implementação de tais características.

Um exemplo de característica comum é o semáforo utilizado para implementar um método atômico em DPC++. Apesar de a efetiva utilização do semáforo depender da implementação de cada objeto distribuído, a declaração, criação e destruição do mesmo pode ser implementada na classe básica, de modo que um método desta classe possa ser simplesmente herdado por um objeto final. Isso reduz também a complexidade da análise feita pelo compilador, que teria que introduzir tais procedimentos no código de todos os objetos distribuídos.

Além de concentrar características comuns, a utilização de uma classe básica para a implementação de objetos distribuídos permite que, futuramente, novos elementos possam ser adicionados sem que seja necessário alterar a implemetação do modelo propriamente dito.

6.3.3 Objetos procuradores

Os objetos procuradores, como descrito no modelo de DPC++, são responsáveis pela comunicação direta com objetos distribuídos remotos. Na prática, sua função consiste em montar mensagens com os parâmetros necessários à invocação de um método e enviá-las ao objeto em questão. Conforme mostrado anteriormente, um procurador guarda o

endereço do *cluster* (ou seja, da porta de comunicação do *dispatcher*) e a referência local de um objeto distribuído. Essas informações, somadas às dos demais objetos procuradores utilizados por um objeto distribuído, constituem os canais de invocação do referido objeto, conforme definido pelo modelo.

De forma semelhante aos objetos distribuídos, os objetos procuradores em DPC++ possuem diversas características em comum, e portanto essas características são também reunidas em uma classe básica *ProxyObject*, que é herdada por todos. Por exemplo, todos os objetos procuradores precisam, durante sua criação, criar uma referência ao objeto Diretório. Esse procedimento é implementado no construtor da classe básica, bastando, posteriormente, que cada objeto procurador execute também o construtor herdado.

Se o método possui um valor de retorno, o procurador ainda aguarda pelo recebimento da resposta, e a repassa ao objeto distribuído local. Para receber essa resposta, o procurador envia, junto com a mensagem, a identificação de uma porta de comunicação à qual aquela deve ser dirigida. Devido à capacidade de execução concorrente dos métodos de um objeto distribuído, porém, a implementação dos objetos procuradores requer especial atenção com respeito ao recebimento de respostas remotas.

A implementação de métodos que possuem um valor de retorno exige a adição de canais de comunicação extras. No modelo original, após a invocação de um método remoto que retorne um valor, o recebimento dessa resposta pode ser feito por uma única porta de comunicação, dedicada a essa finalidade. Isso porque tem-se a certeza de que não há qualquer outra invocação em curso, visto que um objeto distribuído emprega somente um único fluxo de execução. No novo modelo, considerando que um objeto distribuído pode ter, em um dado momento, múltiplos fluxos de execução ativos, é possível que mais de um deles tenha originado uma invocação remota sobre a qual aguardam uma resposta. Se somente um porta de comunicação for utilizada para esse fim, não há garantia de que cada fluxo de execução receberá sua resposta correspondente, o que pode causar erros na execução do algoritmo.

A princípio, imaginou-se que a utilização de uma porta de comunicação para cada método que retorna um valor seria uma solução para o problema. Essa alternativa, porém, não é suficiente para a situação ilustrada na figura 6.5. O exemplo mostra dois objetos distintos *B* e *C* que fazem simultaneamente a invocação do mesmo método em um objeto *A*, gerando, neste último, a criação de dois fluxos de execução concorrentes. Estes, pela implementação do método, comunicam-se com um quarto objeto *D*, do qual esperam uma resposta. Neste ponto, mesmo que o objeto *A* utilize uma porta de comunicação para cada método, não se garante a ordem correta de recebimento das respostas, visto que se referem ao mesmo método.

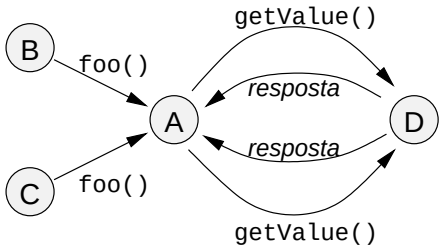

FIGURA 6.5 - Problema no recebimento de respostas em um objeto distribuído.

A solução definitiva para o problema, adotada na implementação, consiste em utilizar uma porta de comunicação distinta para cada invocação de método que retorna um

valor. Desta forma garante-se que cada resposta será recebida pelo fluxo de execução correspondente, mantendo o correto funcionamento do algoritmo.

6.3.4 Objeto Diretório

O objeto Diretório, como visto, é criado no nodo 0 durante o processo de inicialização da aplicação distribuída. Exceto pela maneira como é criado, o objeto Diretório segue o mesmo comportamento de qualquer objeto distribuído, ou seja, possui métodos que são acessados remotamente por meio de objetos procuradores e que podem ser executados concorrentemente entre si.

A criação do objeto Diretório diferencia-se da maneira usual por ser feita de forma direta, ou seja, o Diretório é localmente instanciado no nodo 0 no início da aplicação, e não pelo seu objeto procurador. O modo pelo qual o Diretório é encontrado pelos procuradores é explicado posteriormente.

```
class Directory:  public DistributedObject
{
  private:
    ...
    atomic int getANode ();

  public:
    Directory ();
    ~Directory ();
    dpc_do_addr_t createDistributedObject (constructor_id, parm);
};
```

FIGURA 6.6 - Declaração da classe que implementa o objeto Diretório.

O Diretório possui, além de construtor e destrutor, um único método público, `createDistributedObject()`, como mostra a figura 6.6. Esse método é chamado pelos demais objetos procuradores para criar um novo objeto distribuído. O primeiro parâmetro identifica o construtor a ser utilizado na instanciação do objeto. O segundo consiste em uma mensagem que encapsula os parâmetros exigidos pelo construtor indicado; de outra forma seria necessário incluir o próprio objeto Diretório no processo de compilação de uma aplicação DPC++, para que se pudesse identificar cada parâmetro individualmente.

De posse desses argumentos, o Diretório escolhe um nodo para a criação do objeto e simplesmente repassa a identificação do construtor e a mensagem contendo os parâmetros ao *cluster* correspondente, que se encarrega da instanciação e devolve a referência ao objeto. Ao receber essa resposta, o Diretório novamente repassa esse dado, juntamente com o endereço do *cluster*, ao objeto procurador que solicitou a criação.

Como mencionado, esse método pode ser executado concorrentemente por diversos outros objetos distribuídos, o que contribui para otimizar o desempenho da aplicação. Algumas variáveis internas do Diretório, porém, relacionadas à implementação da política de balanceamento de carga, devem ter acesso sincronizado, e portanto a classe define um método atômico `getANode()` que realiza essa tarefa e retorna a identificação de um nodo apropriado.

A respeito da política de balanceamento de carga, a implementação feita adota uma abordagem bastante simples, fazendo uma escolha circular entre os nodos de processa-

mento disponíveis. A maneira pela qual o Diretório é implementado permite, entretanto, a implantação de uma política mais elaborada, dispensada nesta implementação inicial para não desviar a atenção dos objetivos do trabalho.

6.3.5 Objeto procurador do Diretório

O objeto procurador do Diretório é implementado de forma especial, não sendo derivado da classe *ProxyObject* como os demais. O motivo é justamente a localização do objeto Diretório.

Cada objeto procurador localiza o seu objeto distribuído correspondente por meio do endereço do *cluster* onde tal objeto está instanciado e por uma referência local. Definiu-se que o Diretório será sempre instanciado no nodo 0, portanto seu endereço é o do *cluster* desse nodo. Porém, a referência local ao Diretório não é conhecida dos demais nodos, e seria preciso que, durante a inicialização de DPC++, mensagens fossem enviadas entre os nodos de forma a se obter esse dado, o que aumentaria o custo de carga da aplicação distribuída. Esse procedimento, porém, torna-se desnecessário pela forma como o modelo é implementado.

O objeto procurador do Diretório, ao ser criado, obtém, pelo servidor de nomes, o endereço do *cluster* 0, ao qual suas mensagens serão enviadas. O conteúdo das mensagens, entretanto, não inclui a referência local ao objeto Diretório. Pela identificação do método, o *cluster* é capaz de reconhecer que a mensagem se destina àquele objeto, e assim a invocação pode ser realizada corretamente. Naturalmente, quando o Diretório é criado, o *cluster* guarda uma referência para poder realizar tais invocações.

6.4 Considerações finais

Este capítulo apresentou alguns dos detalhes mais importantes da implementação do modelo de paralelismo proposto, com o objetivo de fornecer uma visão mais concreta de alguns de seus aspectos. O próximo capítulo apresenta alguns resultados experimentais obtidos com a implementação de uma aplicação distribuída que faz uso das extensões propostas.

7 Resultados práticos e medidas

Este capítulo apresenta os resultados obtidos na implementação de uma aplicação distribuída seguindo o novo modelo de distribuição de objetos proposto para DPC++. A aplicação em questão consiste na geração de fractais de Mandelbrot, conforme detalhado a seguir.

7.1 Fractais de Mandelbrot

Os fractais de Mandelbrot [MAN82] consistem em imagens abstratas geradas através de operações matemáticas aplicadas sucessivamente sobre os pontos de uma figura. Formalmente, os fractais ou *conjunto* de Mandelbrot são calculados no plano dos números complexos, onde cada número $c = a + bi$ equivale, em termos computacionais, a um pixel de uma figura. Seja c um ponto qualquer no plano complexo; iniciando-se por um valor $z_0 = 0 + 0i\ (= 0)$, pode-se gerar uma seqüência de pontos, chamada de *órbita*, utilizando-se a fórmula

$$z_{n+1} = z_n^2 + c$$

Se algum ponto da órbita possui módulo (distância à origem) maior que 2.0, diz-se que a órbita *escapou*. O conjunto de Mandelbrot é, então, definido como o conjunto de pontos cujas órbitas nunca escapam.

Quando da implementação do algoritmo de geração de fractais de Mandelbrot, entretanto, duas restrições se fazem necessárias. A primeira é que é necessário discretizar os pontos a serem calculados de modo a corresponderem a pixeis de uma figura. A segunda é que, para que o algoritmo tenha um fim, deve-se estabelecer um limite para o número de iterações efetuadas sobre um ponto. Se a órbita não escapa até o número máximo de iterações ser alcançado, considera-se que o ponto faz parte do conjunto. Essas restrições fazem com que a imagem gerada seja apenas uma aproximação do conjunto real; a qualidade da figura é assim determinada pela resolução utilizada e pelo número máximo de iterações efetuadas sobre um ponto.

Para se obter o efeito estético desejado, define-se a cor de cada ponto da figura em função do número de iterações necessário para calculá-lo, repetindo as cores de forma circular, se necessário.

Dado o limite de tamanho 2.0 para o módulo de um ponto, a aplicação da fórmula sobre os pontos pertencentes à região determinada pelos pontos $(-2, -2)$ e $(2, 2)$ é suficiente para se obter o conjunto de Mandelbrot completo, mostrado na figura 7.1. É usual, entretanto, limitar o cálculo a uma sub-região dessa área, de modo a obter-se um efeito de "zoom" que proporciona a geração de figuras diferentes. Do ponto de vista computacional, esse procedimento altera o custo de processamento do algoritmo, pois diferentes pontos do fractal escapam em mais ou menos iterações, quando não exigem a execução do número total. A escolha de uma ou outra região e um maior ou menor efeito de "zoom" podem provocar mudanças significativas no desempenho do algoritmo.

Com estas considerações, podem-se identificar três fatores que influem, primariamente, no custo de processamento de um fractal de Mandelbrot:

- as dimensões (ou resolução) da figura que se deseja obter pois, naturalmente, uma figura maior contém um número maior de pontos a serem calculados;

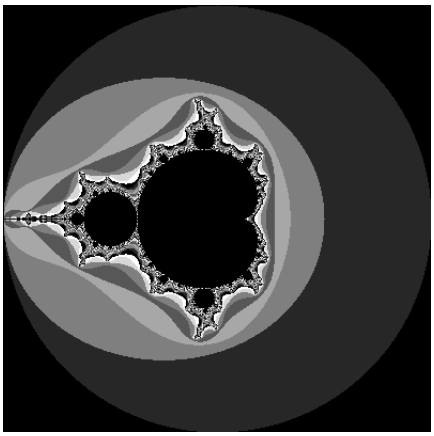

FIGURA 7.1 - O conjunto de Mandelbrot completo.

- a região do fractal a ser calculada;
- o número de iterações sobre cada ponto.

Na prática, o número de iterações deve representar um equilíbrio entre o tempo de processamento e a qualidade da figura que se deseja obter, em função das dimensões e da região de cálculo. De forma geral, pontos mais internos ao fractal necessitam de um número maior de iterações para serem alcançados; portanto, um efeito de "zoom" muito grande implica no aumento do número de iterações necessário para que se obtenha uma imagem razoável.

7.2 Implementação seqüencial

A implementação seqüencial do algoritmo segue a abordagem mais intuitiva, e consiste em realizar, para cada ponto da imagem, o número de iterações necessárias, definindo sua cor final. A figura 7.2 mostra os resultados obtidos para diferentes tamanhos de fractais, tomando-se o conjunto completo como região de cálculo, ou seja, $(-2, -2)$ a $(2, 2)$.

Todos os experimentos foram feitos em uma rede composta de 4 computadores PC conectados por tecnologia Ethernet 10 Mbits rodando o sistema operacional Linux versão 2.0.36. Cada nodo possui dois processadores PentiumPro 200MHz com 64M de memória RAM.

7.3 Implementação paralela

7.3.1 Divisão em sub-regiões

A idéia mais intuitiva de paralelização do algoritmo de Mandelbrot é dividir a figura que se deseja calcular em sub-regiões que podem ser calculadas concorrentemente. Como não existe dependência entre o cálculo de um ponto e de outro, essa idéia pode ser implementada de maneira relativamente simples e eficiente.

A implementação paralela traz, assim, outros dois fatores que influenciam no desempenho do algoritmo:

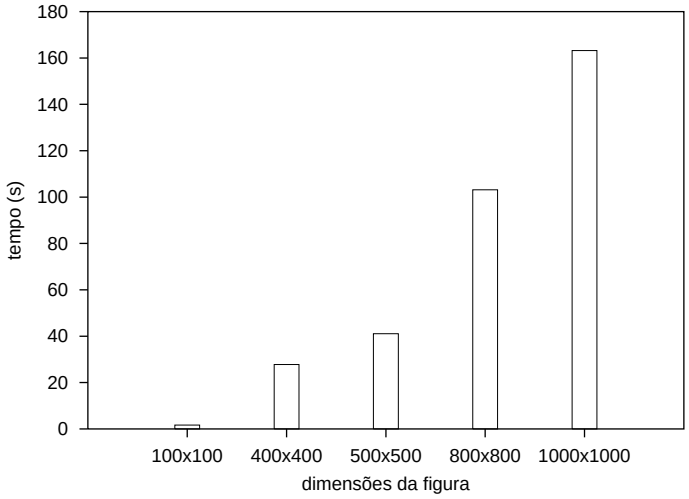

FIGURA 7.2 - Tempo de execução para a geração de diferentes tamanhos de fractais.

- o número de processadores utilizados no cálculo.
- o tamanho das sub-regiões nas quais a figura é dividida;

Os cinco fatores resultantes serão analisados, posteriormente, em sua influência sobre o desempenho da implementação feita em DPC++, principalmente em relação ao paralelismo de grão fino.

A simples divisão de um número pré-determinado de sub-regiões para cada processador não garante, necessariamente, um bom desempenho para a aplicação, pois a variação de tempo de processamento entre os pontos de uma ou outra região podem rapidamente levar um processador a ficar ocioso enquanto que outros tenham que calcular o número total de iterações. Por esse motivo, torna-se mais eficiente dividir a figura em um número maior de partes e fazer os processadores as calcularem por demanda, requisitando novas sub-regiões de cálculo à medida em que completam regiões recebidas anteriormente.

7.3.2 Estruturação da aplicação em DPC++

A abordagem de paralelização recém descrita pode ser implementada de forma bastante adequada utilizando-se o paradigma de orientação a objetos. Em DPC++, a aplicação é baseada em duas classes distribuídas, *MandelMaster*, que distribui as regiões de cálculo e organiza as partes já calculadas para formar a figura, e *MandelWorker*, que implementa um objeto que faz o cálculo de uma sub-região. A figura 7.3 mostra uma visão funcional da aplicação implementada em DPC++.

O número de *workers* pode ser variado de acordo com o grau de paralelismo desejado e com a disponibilidade de máquinas para a execução da aplicação. O tamanho de cada sub-região de cálculo é determinado pelo número de pontos que a compõe.

A figura 7.4 mostra a interface do objeto *master.* Em sua inicialização são fornecidas as dimensões da figura completa e o número de pontos de cada sub-região. Estas são logicamente numeradas de 0 a $n - 1$, onde n é o número total de sub-regiões, para posterior identificação por parte dos *workers.*

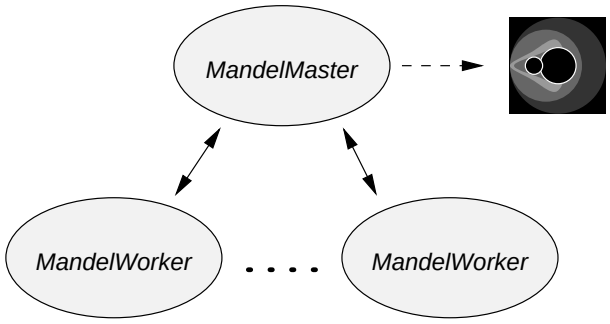

FIGURA 7.3 - Estrutura funcional da implementação em DPC++.

```
dclass MandelMaster
{
  private:
    ...
    cond allPartsReceived;
    atomic void incPartsIn ();

  public:
    MandelMaster (...);
    atomic unsigned long getNewPart ();
    void partDone (unsigned long partnum, filename_t filename);
    atomic void savePicture (filename_t filename);
};
```

FIGURA 7.4 - Declaração da classe *MandelMaster*.

Durante a execução da aplicação, os objetos *worker* obtêm, através do método `getNewPart()`, a identificação de uma sub-região a ser calculada. Terminado o cálculo, um *worker* sinaliza que a sub-região já está calculada através do método `partDone()`, indicando o número que a identifica e o nome de um arquivo no qual os respectivos resultados foram armazenados. O *master* então se encarrega de adicionar tais resultados à figura propriamente dita, guiando-se pelo número da sub-região. Quando não há mais sub-regiões a serem calculadas, o *master* retorna uma indicação especial e os *workers* finalizam sua execução.

Um objeto *worker* recebe, em sua criação, uma referência ao *master*, além da região de cálculo, as dimensões da figura, o tamanho das sub-regiões e o número de iterações a serem efetuadas. As dimensões da figura e o tamanho das sub-regiões são necessários para que o *worker* possa identificar a região a ser calculada a partir do número obtido do *master*. Um *worker* disponibiliza somente um método público `start()`, que faz com que ele inicie seu trabalho.

O grau de paralelismo apresentado pela implementação pode ser dividido em dois níveis distintos:

- de granularidade *grossa*, pela execução paralela dos diversos objetos *worker*;
- de granularidade *fina*, pela execução paralela dos métodos do objeto *master*.

Concentrando-se no paralelismo de grão fino, objetivo principal do trabalho, a seguinte análise pode ser feita a respeito da influência exercida pelos cinco fatores anteriormente citados.

O grau de paralelismo entre os métodos do objeto *master* é basicamente determinado por outros dois fatores: o tamanho de cada sub-região, que faz variar o tempo necessário para ler os resultados do arquivo gerado pelo *worker* e atualizar a figura, e a freqüência com que os métodos são chamados. Enquanto que o tamanho de cada sub-região é diretamente fornecido, a freqüência de acesso ao objeto *master* depende dos demais fatores:

- *diminui* em relação direta com o tamanho das sub-regiões de cálculo e com o número de iterações;
- *aumenta* com o número de *workers*;
- varia *irregularmente* com a definição da região de cálculo do fractal e com as dimensões da figura.

TABELA 7.1 - Influência sobre a freqüência de acesso ao objeto *master*.

Fator	Influência
tamanho das sub-regiões	$\downarrow$
nº iterações	$\downarrow$
nº *workers*	$\uparrow$
tamanho da figura	$\neq$
região do fractal	$\neq$

A tabela 7.1 oferece uma visão melhor desse quadro. Percebe-se que a influência do tamanho das sub-regiões de cálculo na eficiência da utilização de paralelismo fino não pode ser precisamente determinada; quanto maior o seu valor, menor será a freqüência de acesso ao método `partDone()`, porém tanto maior será o tempo necessário para atualizar a figura. A primeira conseqüência tende a diminuir o grau de concorrência interna no objeto *master*, enquanto que a segunda tende a aumentá-lo.

Dada a diversidade de fatores de influência, algumas restrições podem ser feitas para efeito de maior clareza e concreticidade nos resultados obtidos.

Nota-se que as variações do tamanho da figura e da região de cálculo do fractal exercem influência indeterminada sobre o desempenho da aplicação, o que dificulta a análise final dos resultados. A razão para esta influência irregular deve-se ao fato de que, mantendo-se os demais parâmetros fixos, a variação desses parâmetros somente modifica a localização, no plano complexo, dos pontos a serem calculados. Como o cálculo de um único ponto não segue uma regra determinada, em termos de tempo de processamento, este efeito acaba por refletir-se no desempenho da aplicação. Para que os resultados possam ser mais objetivamente analisados, portanto, a região de cálculo será fixada no intervalo $(-2, -2)$ a $(2, 2)$, ou seja, de modo a abranger o fractal em sua totalidade. Igualmente, as dimensões da figura calculada serão fixadas em 1000×1000, valor obtido experimentalmente que proporciona a obtenção de resultados adequados em um tempo de processamento razoável.

Outro fator a ser restrito é o número de iterações aplicado a cada ponto. De acordo com a análise feita, este fator somente exerce influência negativa sobre a freqüência de chamamento dos métodos do objeto *master*. Considerando-se que esse fator regula a proximidade da figura calculada em relação à real, conclui-se que, de acordo com a região de cálculo e dimensões escolhidas para a figura, existe um número mínimo de iterações que proporciona um resultado satisfatório. Assim, de acordo com as experimentações realizadas, fixou-se o número de 500 iterações na obtenção dos resultados.

Os experimentos foram realizados, portanto, variando-se os dois fatores restantes, a saber o número de objetos *worker* e o tamanho das sub-regiões de cálculo. Sob outra ótica, pode-se também observar que estes são os dois fatores introduzidos pela implementação paralela.

Para que se possa observar de forma adequada o efeito do paralelismo de grão fino sobre a aplicação, os tempos de execução e índices de *speedup* e eficiência são apresentados, além das variações seqüencial e paralela, para as execuções com objetos seqüenciais, ou seja, sem o uso de concorrência interna. Os tempos de execução são dados em segundos. Sejam t_s, t_d e t_c, respectivamente, as medidas de tempo para execução seqüencial, distribuída (com objetos seqüenciais) e completa (usando concorrência interna); os índices de *speedup* são dados por

$$S_d(n) = \frac{t_s}{t_d(n)}$$

$$S_t(n) = \frac{t_s}{t_c(n)}$$

onde n é o número de *workers* utilizado, S_d é o *speedup* obtido isoladamente pela distribuição da aplicação, e S_t representa o ganho total obtido pela utilização do modelo completo. Um terceiro índice S_p dado por

$$S_p(n) = \frac{t_d(n)}{t_c(n)}$$

representa o ganho obtido, em relação à execução distribuída, pelo uso de métodos concorrentes. As medidas de eficiência são determinados pela fórmula

$$E(n) = \frac{S(n)}{n}$$

onde $S(n)$ é uma das variações de *speedup* apresentadas. As medidas de tempo utilizadas nos cálculos são médias aritméticas de 10 execuções da mesma aplicação. Este valor proporciona resultados confiáveis, visto que as máquinas utilizadas encontravam-se dedicadas aos experimentos realizados.

7.3.3 Efeito da distribuição da aplicação

As figuras 7.5 e 7.6 mostram as medidas de tempo (t_d) e *speedup* (S_d) obtidos com diferentes tamanhos de sub-regiões, variando-se o número de objetos *worker*. Esses resultados correspondem à execução da aplicação sem a utilização de concorrência interna aos objetos, para que se possa melhor analisar o ganho obtido isoladamente pela distribuição da aplicação.

As duas figuras mostram claramente a influência do grão de paralelismo sobre o desempenho da aplicação. Para tamanhos de sub-regiões muito pequenos, o custo de

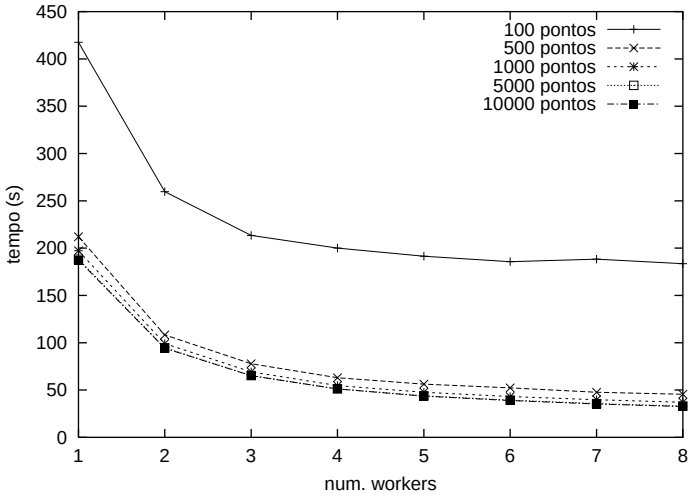

FIGURA 7.5 - Tempos de execução utilizando objetos seqüenciais.

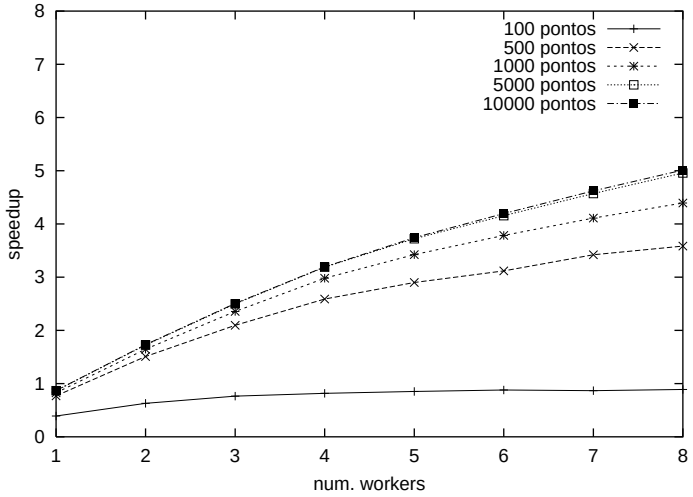

FIGURA 7.6 - Ganho obtido com a distribuição da aplicação.

comunicação entre os objetos distribuídos não compensa a divisão do cálculo, o que resulta em baixos índices de *speedup*. Analisando-se a eficiência da distribuição, mostrada na figura 7.7, conclui-se que o algoritmo passa a apresentar um desempenho satisfatório para tamanhos de sub-regiões entre 500 e 1000 pontos. Nesta faixa, a aplicação apresenta eficiência média em torno de 70%.

7.3.4 Influência do paralelismo de grão fino

O gráfico da figura 7.8 mostra o ganho em desempenho (S_p) obtido pela introdução de concorrência entre os métodos do objeto *master.*

Conforme discutido anteriormente, o tamanho das sub-regiões de cálculo exerce influências antagônicas sobre o paralelismo fino. Conclui-se ser este o motivo do comportamento relativamente irregular da curva de desempenho em relação ao número de objetos *worker.*

Os resultados obtidos indicam que o melhor equilíbrio entre o tamanho das sub-

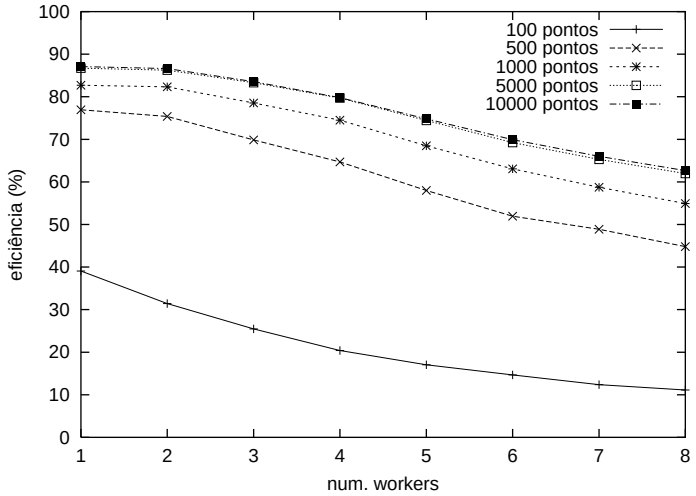

FIGURA 7.7 - Eficiência do algoritmo com objetos seqüenciais.

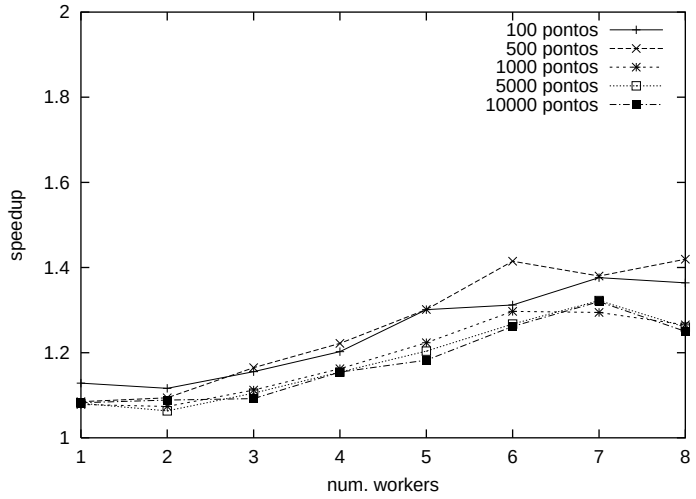

FIGURA 7.8 - Ganho obtido pela concorrência interna ao objeto *master.*

regiões e a freqüência de acesso ao objeto *master*, fatores de influência sobre o paralelismo fino, é conseguido para sub-regiões com tamanho em torno de 500 pontos, caso em que o *speedup* obtido com o uso de métodos concorrentes chega a cerca de 1.4, para um valor máximo teórico de 2.

Como previsto, um número maior de objetos *worker* aumenta a concorrência de acesso ao *master*, e em conseqüência também aumenta o desempenho da aplicação. O gráfico mostra os melhores resultados com a utilização de 6 a 7 objetos *worker*, indicando uma possível saturação do sistema a partir deste ponto, onde as curvas apresentam tendência de queda.

Observa-se que o ganho obtido é sempre maior que 1, não acarretando, em nenhum caso, perda de desempenho em relação à execução com objetos seqüenciais. Este fato se verifica mesmo com a utilização de apenas 1 objeto *worker*, pois ainda assim as chamadas ao método `partDone()` podem ser sobrepostas.

A eficiência do uso de métodos concorrentes é mostrada na figura 7.9, onde se ob-

serva um aproveitamento médio em torno de 60% do valor máximo teórico. Para regiões de 500 pontos, onde o melhor desempenho é obtido, a eficiência chega a cerca de 72% com a utilização de 6 objetos *worker.*

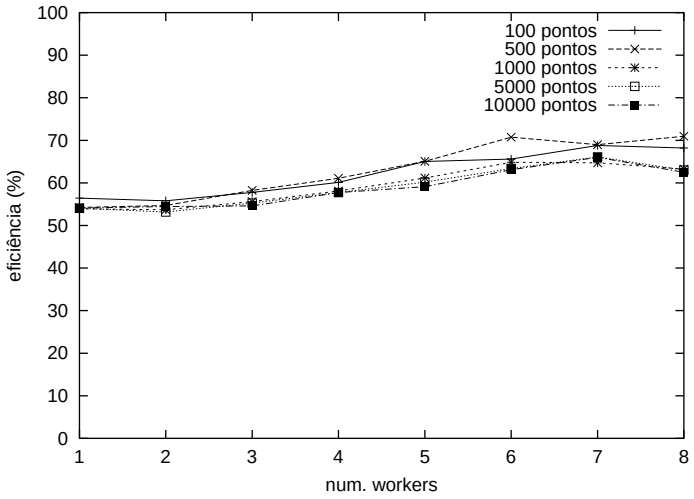

FIGURA 7.9 - Eficiência do algoritmo em relação ao paralelismo fino.

Os resultados apresentados comprovam a eficácia do modelo proposto em aumentar o desempenho de uma aplicação DPC++. Cabe salientar que, mesmo com a necessidade de sincronização introduzida pelo objeto *master* durante todo o processo de cálculo do fractal, o uso de métodos concorrentes mostrou-se efetivo no aumento do desempenho da aplicação.

Além do ganho em desempenho, outro ponto favorável que se pode observar é a naturalidade proporcionada pela sintaxe de programação em relação à modelagem da aplicação. A não utilização de mecanismos explícitos de expressão de paralelismo contribui para que o funcionamento da aplicação possa ser facilmente compreendido por um usuário não familiarizado com o algoritmo; adicionalmente, o uso da variável condicional `allPartsReceived` permite que o término da atividade dos *workers* possa ser detectado de forma simples e objetiva. Este benefício, também muito importante, reforça a conclusão que os objetivos do trabalho foram efetivamente alcançados.

8 Conclusão

A busca por ferramentas e ambientes de programação cada vez mais eficientes e adequados às necessidades dos usuários tem continuamente motivado pesquisas nessa área. A programação paralela, em particular, oferece um campo de trabalho bastante interessante por ser uma prática relativamente nova. Impulsionada nos últimos anos pelo aprimoramento das técnicas de construção de máquinas paralelas e pelo surgimento de novas tecnologias de comunicação, a programação paralela vem evoluindo constantemente e tornando-se cada vez mais presente nas atividades relacionadas à Ciência da Computação. O crescente interesse na pesquisa em agregados de máquinas multiprocessadas é um importante fator motivador para o desenvolvimento e utilização de ferramentas de programação paralela, oferecendo a exploração de um nível de paralelismo até então de pouca disponibilidade. As diversas linguagens para programação paralela da atualidade oferecem diferentes formas e níveis de exploração de concorrência, dedicando-se em maioria à exploração do processamento distribuído, principalmente devido ao uso de redes locais como máquinas virtuais. Tal foi também a motivação para o surgimento de DPC++, linguagem sobre a qual a pesquisa é baseada.

O modelo de concorrência apresentado nesta pesquisa difere da maioria das linguagens paralelas por adotar uma forma de expressão implícita de paralelismo. Embora o usual seja assumir o processamento seqüencial como normal, o modelo procura conferir um caráter mais natural à programação paralela, objetivando uma linguagem com grande expressividade e que possa eficientemente explorar o nível fino de concorrência oferecido pelo ambiente de execução.

O paralelismo de grão fino é explorado, no modelo proposto, pela possibilidade de concorrência entre os métodos de um objeto. Comparada com outros mecanismos, esta alternativa mostra-se adequada porque dispensa comandos adicionais, mantendo a linguagem mais próxima da orientação a objetos tradicional, e oferece semântica que estimula o processamento paralelo. Essa característica é implementada por meio de múltiplos fluxos de execução, ou *threads*, de modo que cada *thread* seja responsável pela execução de um método individualmente. O uso de *threads* permite não só a execução paralela dos métodos de um objeto mas também a sobreposição de operações de comunicação e computação. Enquanto que a primeira vantagem é observada em arquiteturas multiprocessadas, a segunda se faz presente também em sistemas que contam com apenas um processador. Tal característica permite que um objeto siga o seu processamento mesmo quando da realização de operações de comunicação, conseqüentemente contribuindo igualmente para o seu desempenho.

O modelo adota o mecanismo de monitores como recurso de sincronização entre métodos concorrentes. Tal funcionalidade confere à linguagem um excelente poder de expressão para a introdução de sincronização em uma aplicação paralela, sendo versátil o bastante para a resolução de vários tipos de problemas ou mesmo para a implementação de outros mecanismos, como mutexes ou semáforos. Além disso, as semelhanças estrutural e semântica entre um monitor e um objeto permitem uma integração bastante adequada do mecanismo ao modelo de programação, novamente contribuindo para a naturalidade da linguagem.

As extensões introduzidas pelo novo modelo no nível operacional de DPC++ trazem uma série de vantagens em termos de desempenho e economia de recursos, como é o caso da criação de objetos distribuídos e de sua manutenção em um mesmo *cluster*.

Em resumo, a efetiva introdução do modelo proposto à linguagem DPC++ potencialmente permite um desempenho ainda melhor de suas aplicações, explorando todos os níveis de concorrência oferecidos pelo ambiente de execução, aliado a uma sintaxe de programação com maior poder de expressão.

A implementação do modelo na prática comprova sua capacidade de exploração de paralelismo de grão fino, representando o alcance dos objetivos principais do trabalho. A aplicação de geração de fractais de Mandelbrot implementada, mesmo com a utilização freqüente de operações de sincronização, demonstrou eficiência média de cerca de 60% em uma máquina com 2 processadores, chegando a um índice de *speedup* de cerca de 1.4, o que equivale a mais de 70% de eficiência de exploração de paralelismo fino. Os resultados mostram ainda que em nenhum caso houve perda de desempenho em relação à execução distribuída.

Além dos resultados práticos, a própria estruturação da aplicação implementada permite dizer que, no nível de linguagem de programação, o trabalho também alcançou seus objetivos. A naturalidade com que a concorrência e a sincronização são atingidas dentro da aplicação levam a concluir que se conseguiu uma integração bastante adequada de tais funcionalidades com o paradigma de programação orientada a objetos. Comparando a sintaxe proposta com outras linguagens para programação paralela da atualidade, nota-se que DPC++ permite que o programador, envolvido em um ambiente intrinsecamente paralelo, possa utilizar-se desse recurso como um verdadeiro paradigma de programação, e assim desenvolver sua aplicação de forma mais simples e intuitiva.

Considera-se, por outro lado, que a programação paralela ainda tem muitas arestas a serem aparadas até que se torne uma prática tão usual como a programação seqüencial tradicional. Acredita-se, porém, que o desenvolvimento de pesquisas como a realizada neste trabalho pode contribuir significativamente para a obtenção de tal condição. Os algoritmos elaborados durante o desenvolvimento do trabalho mostraram-se bastante favorecidos pelas características adotadas no modelo proposto; o uso futuro de DPC++, porém, é que vai determinar na prática o quanto a linguagem pode ser útil no desenvolvimento de aplicações reais. Sendo a conclusão positiva ou negativa, tem-se a certeza de que mais um passo será dado na direção de uma ferramenta de programação adequada.

As atividades futuras no projeto DPC++ incluem a efetiva integração do modelo de concorrência proposto à linguagem, adaptando o compilador para a nova sintaxe e introduzindo as extensões ao modelo original. Também estão previstas a reestruturação do nível operacional de DPC++, de modo a utilizar os serviços da camada de execução DECK, e a integração de outras atividades de pesquisa sobre o projeto. Por fim, prevê-se a realização de experimentos com aplicações DPC++ sobre agregados de máquinas multiprocessadas conectadas por tecnologias de alto desempenho como Fast Ethernet e Myrinet, as quais encontram-se em fase de implantação no Instituto de Informática.

Anexo 1 A camada de serviços DECK

Conforme mostrado no capítulo 6, a implementação do novo modelo de distribuição de objetos de DPC++ baseia-se em uma camada de *software* que oferece serviços básicos como comunicação e sincronização. Esta camada, chamada DECK — *Distributed Executive Communication Kernel* — atua entre a implementação do modelo propriamente dita e o ambiente de execução utilizado, com o objetivo de conferir melhor estruturação e portabilidade a DPC++. Este anexo descreve a organização interna de DECK e uma implementação bastante simplificada utilizada na obtenção dos resultados apresentados anteriormente.

A.1 Descrição da camada DECK

A.1.1 Contextualização de DECK no ambiente de execução

A camada DECK exerce uma função intermediadora entre DPC++ e os recursos oferecidos pelo ambiente de execução. A configuração final do ambiente depende da forma como DECK é implementado, como mostrado na figura A.1. A situação (a) corresponde a uma implementação de DECK diretamente sobre o sistema operacional; em (b), supondo um ambiente UNIX, DECK é implementado sobre os pacotes POSIX *threads* e MPI.

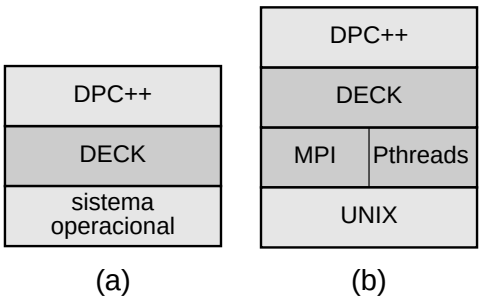

FIGURA A.1 - Variações de implementação da camada DECK.

As diferenças de configuração influem sobre o desempenho e a portabilidade de DECK. Em teoria, a situação (a) oferece melhor desempenho, enquanto que a situação (b) confere maior portabilidade à implementação da camada.

A.1.2 Organização interna

Internamente, conforme ilustra a figura A.2, a camada DECK é dividida em duas sub-camadas, uma de serviços básicos, dependente da configuração do sistema, e outra que oferece serviços mais elaborados, como comunicação em grupo e suporte a tolerância a falhas.

A camada básica é responsável pelos serviços de comunicação, sincronização e gerência de *threads*. A camada superior assume, entre outros, o serviço de nomeação de portas de comunicação.

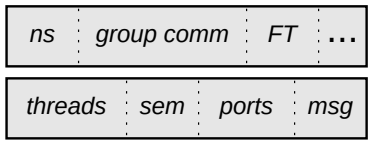

FIGURA A.2 - Estrutura interna da camada DECK.

A seção seguinte descreve a implementação de tais serviços conforme utilizada para a realização dos experimentos com o novo modelo de distribuição de objetos de DPC++. Esta implementação, contudo, não inclui todos os serviços previstos para DECK, mas somente aqueles necessários ao modelo. Uma descrição mais completa dos serviços oferecidos pela camada é apresentada por Barreto et al. [BAR98].

A.2 Implementação dos serviços utilizados por DPC++

Esta seção apresenta uma implementação simplificada de DECK, incluindo somente os serviços de comunicação, sincronização, gerência de *threads* e nomeação. A implementação é feita diretamente sobre o sistema UNIX, utilizando o padrão Pthreads (POSIX *threads*) e o mecanismo de *sockets* para comunicação.

As aplicações DECK seguem o modelo de execução SPMD, ou seja, o mesmo processo é disparado simultaneamente em todos os nodos. Como parâmetros de inicialização, o processo recebe o número do nodo onde foi disparado, o número total de nodos e uma lista contendo os endereços IP de todos os nodos. O número de cada nodo é dado pela ordem em que aparece na lista; para n nodos, a numeração varia de 0 a $n - 1$.

A.2.1 Funções de propósito geral

Antes dos serviços propriamente ditos, esta seção apresenta algumas funções de propósito geral dentro de DECK, conforme definido pela interface apresentada na figura A.3.

```
int deck_init (int *argc, char ***argv);
int deck_done ();
int deck_node ();
int deck_numnodes ();
```

FIGURA A.3 - Funções de propósito geral em DECK.

A função de inicialização da camada DECK recebe referências aos argumentos `argc` e `argv` da rotina `main()` da aplicação, de onde são extraídos alguns parâmetros usados no processo. Ainda são disponibilizadas funções de finalização da camada e outras relativas à identificação dos nodos.

A.2.2 Gerência de *threads*

A gerência de *threads*, nesta implementação de DECK, inclui somente as funções de criação e de *yield* da CPU, que faz com que o *thread* seja desescalonado. As funções

são diretamente mapeadas nos equivalentes Pthreads e UNIX. Os *threads* são criados no estado *detached*, para melhor aproveitamento de recursos, e com escopo de escalonamento em nível de sistema. A figura A.4 mostra a interface de programação definida.

```
typedef struct
{
  ...
} deck_thread;

int deck_thread_create (deck_thread *t, void *start, void *parm);
void deck_thread_yield ();
```

FIGURA A.4 - Interface de *threads* em DECK.

A.2.3 Semáforos

A implementação dos semáforos de DECK é feita utilizando-se os mecanismos de *mutex* e variáveis condicionais oferecidos em Pthreads. Um semáforo utiliza, além do contador, um *mutex* para sincronizar o acesso a essa variável e uma variável condicional que efetivamente bloqueia e libera o *thread.* A interface de programação é apresentada na figura A.5.

```
typedef struct
{
  ...
} deck_sem;

int deck_sem_create (deck_sem *s, int init_value);
int deck_sem_destroy (deck_sem *s);
int deck_sem_p (deck_sem *s);
int deck_sem_v (deck_sem *s);
```

FIGURA A.5 - Interface de semáforos em DECK.

A.2.4 Mensagens

Um objeto mensagem consiste de uma área de armazenamento alocada dinamicamente quando a mensagem é criada. Além disso, a estrutura que define uma mensagem inclui variáveis para o controle de empacotamento e desempacotamento de dados. A interface de programação é mostrada na figura A.6.

O empacotamento e desempacotamento de dados é feito através da indicação de tipos pré-definidos (caracteres, inteiros, reais, etc.). Estes incluem portas de comunicação e as próprias mensagens.

Além das funções de criação, destruição, empacotamento e desempacotamento de dados, existem ainda duas funções que permitem que uma mesma mensagem seja reaproveitada em várias operações de comunicação.

```
typedef struct
{
  ...
} deck_msg;

int deck_msg_create (deck_msg *m, unsigned long size);
int deck_msg_destroy (deck_msg *m);
int deck_msg_pack (deck_msg *m, int type, void *datum, int size);
int deck_msg_unpack (deck_msg *m, int type, void *datum, int size);
int deck_msg_clear (deck_msg *m);
int deck_msg_reset (deck_msg *m);
```

FIGURA A.6 - Interface de mensagens em DECK.

A.2.5 Portas de comunicação

As portas de comunicação são implementadas através do mecanismo de *sockets* utilizando datagramas. A criação de uma porta de comunicação cria um novo *socket*, cujo descritor é armazenado na estrutura da porta, juntamente com o endereço da máquina onde a operação foi executada. A figura A.7 mostra a interface de programação.

```
typedef struct
{
  ...
} deck_port;

int deck_port_create (deck_port *p);
int deck_port_destroy (deck_port *p);
int deck_port_send (deck_port *p, deck_msg *m);
int deck_port_recv (deck_port *p, deck_msg *m);
```

FIGURA A.7 - Interface de portas de comunicação em DECK.

Portas de comunicação podem ser empacotadas em mensagens e enviadas a nodos remotos. Uma porta recebida através de uma mensagem somente aceita o envio de mensagens. Além disso, o descritor do *socket* não pode ser utilizado, pois refere-se a um recurso do nodo remoto; neste caso, um novo *socket* é criado e destruído a cada envio de mensagem. O endereço previamente armazenado é utilizado para localizar o nodo de destino.

A.2.6 Servidor de nomes

Este é o único serviço não básico implementado nesta primeira versão de DECK. O servidor de nomes permite a associação de nomes a portas de comunicação. Um nome é um *string* de caracteres com um tamanho máximo pré-definido. Uma vez associada a um nome, uma porta de comunicação pode ser obtida mediante apresentação dessa identificação. Esta é a maneira pela qual um *thread* pode entrar em comunicação com outro em um nodo remoto. A figura A.8 mostra a interface de acesso ao servidor de nomes de DECK.

```
int deck_ns_bind (char *name, deck_port *p);
int deck_ns_fetch (char *name, deck_port *p);
```

FIGURA A.8 - Interface de acesso ao servidor de nomes.

Durante a inicialização de DECK, um *thread* é criado à parte, no nodo 0, para atender às requisições de nomeação. O recebimento de tais requisições é feito através de uma porta de comunicação especial, bem conhecida do restante do sistema. O servidor de nomes executa, até a finalização de DECK, um *loop* que consiste em receber uma requisição e executar o serviço solicitado, de forma semelhante ao *dispatcher* de um *cluster* em DPC++. Entretanto, apenas um fluxo de execução é utilizado.

A.3 Exemplo de utilização

O exemplo mostrado na figura A.9 ilustra a utilização de DECK na implementação de uma aplicação simples de troca de mensagens entre dois nodos distintos.

```
#include <deck.h>
int main (int argc, char **argv)
{
  deck_port p;
  deck_msg m;
  char msg[256];

  deck_init (&argc, &argv);

  if (deck_node () == 0) // receiver
  {
    deck_port_create (&p);
    deck_ns_bind ("Teste", &p);
    deck_msg_create (&m, 256);
    deck_port_recv (&p, &m);
    deck_msg_unpack (&m, DECK_OTHER, msg, 256);
    printf (msg);
    deck_msg_destroy (&m);
    deck_port_destroy (&p);
  };

  if (deck_node () == 1) // sender
  {
    deck_msg_create (&m, 256);
    sprintf (msg, "Hello, world.");
    deck_msg_pack (&m, DECK_OTHER, msg, 256);
    while (deck_ns_fetch ("Teste", &p) != DECK_EOK)
      deck_thread_yield ();
    deck_port_send (&p, &m);
    deck_msg_destroy (&m);
  };

  deck_done ();
  return (0);
};
```

FIGURA A.9 - Exemplo de utilização de DECK.

Bibliografia

[ALT91] ALTERNATIVE signaling disciplines. In: ANDREWS, Gregory R. **Concurrent programming**: principles and practice. Redwood City, CA: Benjamin/Cummings, 1991. p.305–307.

[AND91] ANDREWS, Gregory R. **Concurrent programming**: principles and practice. Redwood City, CA: Benjamin/Cummings, 1991. 637p.

[ASS93] ASSENMACHER, H. et al. Panda—suporting distributed programming in C++. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 7., 1993, Kaiserslautern, Germany. **Proceedings...** Berlin: Springer-Verlag, 1993. p.361–383. (Lecture Notes in Computer Science, v.707).

[ÁVI99] ÁVILA, Rafael B.; NAVAUX, Philippe O. A. Um modelo de expressão implícita de concorrência aplicado à programação orientada a objetos. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 3., 1999, Porto Alegre. **Anais...** Porto Alegre: UFRGS, 1999. p.225–229.

[ÁVI97] ÁVILA, Rafael Bohrer. **A comparative study on DPC++ and other concurrent object-oriented languages**. Porto Alegre: CPGCC da UFRGS, 1997. (TI-673).

[BAK96] BAKER, Louis; SMITH, Bradley J. **Parallel programming**. New York, NY: McGraw-Hill, 1996. 381p.

[BAR98] BARRETO, Marcos E.; NAVAUX, Philippe O. A.; RIVIÈRE, Michel P. DECK: a new model for a distributed executive kernel integrating communication and multithreading for support of distributed object oriented application with fault tolerance support. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 4., 1998, Neuquén, AR. **Anales...** Neuquén: Universidad Nacional de Comahue, Facultad de Economia y Administración, Departamento de Informática y Estadística, 1998. v.2, p.623–637.

[BOD95] BODEN, N. et al. Myrinet: a gigabit-per-second local-area network. **IEEE Micro**, Los Alamitos, v.15, n.1, p.29–36, Feb. 1995.

[CAR96] CAROMEL, Denis; BELLONCLE, Fabrice; ROUDIER, Yves. C++//. In: WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. p.257–296.

[CAV93] CAVALHEIRO, Gerson G. H.; NAVAUX, Philippe O. A. DPC++: uma linguagem para processamento distribuído. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES—PROCESSAMENTO DE ALTO DESEMPENHO, 5., 1993, Florianópolis. **Anais...** Florianópolis: SBC, 1993. v.2, p.732–744.

[CAV94] CAVALHEIRO, Gerson G. H. **Um modelo para linguagens orientadas a objetos distribuído**. Porto Alegre: CPGCC da UFRGS, 1994. Dissertação de Mestrado.

[CHA96] CHANDRA, Rohit; GUPTA, Anoop; HENNESSY, John L. COOL. In: WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. p.215–256.

[CHI96] CHIEN, Andrew A.; DOLBY, Julian T. ICC++. In: WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. p.343–382.

[EAS95] EAST, Ian. **Parallel processing with communicating process architecture**. London: UCL Press, 1995. 119p.

[ELL89] ELLIS, Clarence A.; GIBBS, Simon J. Active objects: realities and possibilities. In: KIM, Won; LOCHOVSKY, Frederick H. (Eds.). **Object-oriented concepts, databases, and applications**. New York, NY: ACM Press, 1989. p.561–572.

[FRÖ96] FRÖHLICH, Antônio A. M. et al. A concurrent programming environment for the i486. In: INTERNATIONAL SYMPOSIUM ON SYSTEMS RESEARCH, INFORMATICS AND CYBERNETICS—INFORMATION SYSTEMS ANALYSIS AND SYNTHESIS, 5., 1996, Orlando, US. **Proceedings...** [S.l.: s.n.], 1996.

[FRÖ98] FRÖHLICH, Antônio Augusto Medeiros. **SNOW project**. Disponível por WWW em http://www.first.gmd.de/~guto/snow (fev. 1998).

[FUR95] FURMENTO, Nathalie; ROUDIER, Yves; SIEGEL, Günther. **Parallélisme et distribution en C++**: une revue des langages existants. Valbonne, FR: I3S, Université de Nice Sophia-Antipolis, 1995. (RR 95-02).

[GEI94] GEIST, Al et al. **PVM**: parallel virtual machine. Cambridge, MA: MIT Press, 1994.

[GIN97] GINZBURG, I. **Athapascan0b**: intégration efficace et portable de multiprogrammation légère et de communication. Grenoble, FR: Institut National Polytechnique de Grenoble (INPG), 1997. Tese de Doutorado.

[HAL85] HALSTEAD JR., Robert H. Multilisp: a language for concurrent symbolic computation. **ACM Transactions on Programming Languages and Systems**, New York, v.7, n.4, p.501–538, Oct. 1985.

[HEI98] HEISS, Hans-Ulrich. **AG Heiß—project Arminius**. Disponível por WWW em http://www.uni-paderborn.de/cs/heiss/arminius (dez. 1998).

[IEE92] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **IEEE standard for scalable coherent interface (SCI)**, IEEE 1596-1992. New York, NY, 1992.

[IEE95] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **Local and metropolitan area networks-supplement—media access control (MAC) parameters, physical layer, medium attachment units and repeater for 100Mb/s operation, type 100BASE-T (clauses 21–30)**, IEEE 802.3u-1995. New York, NY, 1995.

[IEE95a] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **Information technology—portable operating system interface (POSIX), threads extension [C language]**, IEEE 1003.1c-1995. New York, NY, 1995.

[KAL96] KALÉ, Laxmikant V.; KRISHNAN, Sanjeev. CHARM++. In: WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. p.175–214.

[KES96] KESSELMAN, Carl. CC++. In: WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. p.91–130.

[LAR96] LARUS, James R.; RICHARDS, Brad; VISWANATHAN, Guhan. C**. In: WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. p.297–342.

[LEA96] LEA, Douglas. **Concurrent programming in Java**: design principles and patterns. Reading, MA: Addison-Wesley, 1996.

[LER99] LEROY, Xavier. **LinuxThreads library**. Disponível por WWW em http://pauillac.inria.fr/~xleroy/linuxthreads (jan. 1999).

[LEW98] LEWIS, Bil; BERG, Daniel J. **Multithreaded programming with Pthreads**. Mountain View: SUN Microsystems, 1998. 382p.

[MAN82] MANDELBROT, B. B. **The fractal geometry of nature**. New York, NY: W. E. Freeman and Company, 1982.

[MON91] MONITORS. In: SILBERSCHATZ, Abraham; PETERSON, James Lyle; GALVIN, Peter B. **Operating system concepts**. 3.ed. Reading, MA: Addison-Wesley, 1991. p.168–175.

[MPI94] MPI FORUM. **The MPI message passing interface standard**. Knoxville: University of Tennessee, 1994.

[OAK97] OAKS, Scott; WONG, Henry. **Java threads**. Sebastopol, CA: O'Reilly & Associates, 1997. 254p.

[O'FA96] O'FARRELL, William G. et al. ABC++. In: WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. p.1–42.

[PRO99] PROVENZANO, Christopher Angelo. **Pthreads**: general information. Disponível por WWW em http://www.mit.edu/people/proven/pthreads.html (jan. 1999).

[SIL91] SILBERSCHATZ, Abraham; PETERSON, James Lyle; GALVIN, Peter B. **Operating system concepts**. 3.ed. Reading, MA: Addison-Wesley, 1991. 696p.

[TRE98] TREADMARKS distributed shared memory (DSM) system. Disponível por WWW em http://www.cs.rice.edu/~willy/TreadMarks/overview.html (dez. 1998).

[WIL96] WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. 758p.

[WIN96] WINDER, Russel et al. UC++. In: WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. p.629–669.

[WYA92] WYATT, Barbara B.; KAVI, Krishna; HUFNAGEL, Steve. Parallelism in object-oriented languages: a survey. **IEEE Software**, Los Alamitos, p.56–66, Nov. 1992.

[YAN96] YANG, Shelby X. et al. pC++. In: WILSON, Gregory V.; LU, Paul (Eds.). **Parallel programming using C++**. Cambridge, MA: MIT Press, 1996. p.507–546.