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

Uma Plataforma CCA para Aplicações Científicas usando Conectores

Gisele Azevedo de Araújo

Fortaleza-Ceará
16 de abril de 2010

Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação

Gisele Azevedo de Araújo

Uma Plataforma CCA para Aplicações Científicas usando Conectores

Trabalho apresentado ao Programa de Doutorado em Ciência da Computação do Departamento de Computação da Universidade Federal do Ceará como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: ***Ricardo Cordeiro Corrêa, DC/UFC***

Fortaleza-Ceará
16 de abril de 2010

A689n Araújo, Gisele Azevedo de
Uma plataforma CCA para aplicações científicas usando conectores / Gisele Azevedo de Araújo. – Fortaleza, 2010.
153 f. il.; color. enc.

Orientador: Prof. Dr. Ricardo Cordeiro Corrêa
Área de concentração: Sistemas distribuídos
Tese (doutorado) - Universidade Federal do Ceará, Centro de Ciências. Departamento de computação, Fortaleza, 2010.

1. Sistemas operacionais distribuídos (computadores) 2. Análise de componentes principais I. Corrêa, Ricardo Cordeiro (orient.) II. Universidade Federal do Ceará – Pós-Graduação em Ciência da Computação. III.Título

CDD 005

Uma Plataforma CCA para Aplicações Científicas Usando Conectores

Gisele Azevedo de Araújo

Tese apresentada ao Curso de Mestrado e Doutorado em Ciência da Computação da Universidade Federal do Ceará, como parte dos Requisitos para a obtenção do Grau de Doutor em Ciência da Computação.

Composição da Banca Examinadora:

Prof. Dr. Ricardo Cordeiro Corrêa (DC/UFC) (DC/UFC) (Presidente)

Prof. Dra. Alba Cristina Magalhães de Melo (UNB)

Prof. Dra. Maria Cristina Silva Boeres (UFF)

Prof. Dr. Francisco Heron de Carvalho Júnior (DC/UFC)

Prof. Dr. Joaquim Bento Cavalcante Neto (DC/UFC)

Aprovada em 16 de abril de 2010.

Sumário

Resumo i

Abstract ii

1 Introdução 1

2 Computação de Alto Desempenho 8

2.1 Taxonomia do Paralelismo 9

2.2 Soluções Arquiteturais 11

2.3 Modelos 13

2.3.1 Modelo Arquitetônico: Cliente/Servidor 13

2.3.2 Modelo Arquitetônico: Ponto-a-ponto 14

2.3.3 Modelo Fundamental: Dirigido a eventos 15

2.4 Programação Paralela e Distribuída 18

2.4.1 Tarefa e Processo 18

2.4.2 Problema $M \times N$ 18

2.4.3 Comunicação em Sistemas Distribuídos e Paralelos e Sincronismo 19

2.4.4 MPI 20

2.4.5 Objetos Distribuídos 21

2.4.6 *Socket* 22

2.4.7 *Clusters* 23

2.4.8 RPC 24

2.4.9 CORBA 26

2.4.10 Grades (do inglês *Grids*) 27

2.5 Medidas de Desempenho (Eficiência e Aceleração) 29

2.5.1 Abordagem de Medição de Desempenho 29

3 Modelos de Componentes e Plataformas para Computação de Alto Desempenho 31

3.1 O Conceito de Componente 31

3.2 O Modelo de Componentes CCA 34

3.2.1 Elementos Básicos: Portas e Conexões 35

3.2.2 Criação de Componentes e Realização de Conexões 38

3.2.3 Paralelismo no CCA . . . 39
3.3 Plataformas CCA . . . 41
3.3.1 Ccaffeine . . . 41
3.3.2 XCAT . . . 42
3.3.3 DCA . . . 43
3.3.4 SCIRun . . . 44
3.3.5 MOCCA . . . 45
3.3.6 Decaf . . . 45
3.4 O Modelo de Componentes *Hash* (#) . . . 46
3.4.1 Premissas do # . . . 46
3.4.2 Conectores . . . 51
3.4.3 Espécies de Componentes . . . 52
3.4.4 Semântica de Composição . . . 52
3.4.5 Ambiente para Construção de Sistemas de Programação # . . . 53
3.5 O Modelo de Componentes Fractal . . . 55

4 Expansão do Modelo de Conexão do CCA **58**
4.1 Implementação da Comunicação . . . 58
4.2 Conectores Endógenos no CCA . . . 62
4.2.1 Conectores Endógenos - Conexão Direta . . . 63
4.2.2 Conectores Endógenos - Conexão Indireta . . . 63
4.3 Conectores Exógenos no CCA . . . 65
4.4 Considerações Finais . . . 67

5 Plataforma *Forró* **69**
5.1 Visão Geral da Plataforma *Forró* . . . 69
5.2 Arquitetura da Plataforma *Forró* . . . 72
5.2.1 Elementos da Plataforma . . . 73
5.3 Modelo de Implantação do *Forró* . . . 77
5.3.1 Componentes e Portas . . . 77
5.3.2 Conexões com Conectores Diretos . . . 77
5.3.3 Conexões com Conectores Indiretos . . . 78
5.4 Implementação e Funcionamento da Plataforma *Forró* . . . 78
5.4.1 Implementação dos Elementos do Forró . . . 78
5.4.2 Componentes Adicionais . . . 80
5.4.3 Funcionamento dos Elementos do *Forró* . . . 82

6 Estudos de Caso **91**
6.1 Considerações Iniciais . . . 91
6.2 Estudo de Caso: Problemas de Otimização sobre Conjuntos Independentes em Grafos . . . 92
6.2.1 Descrição do Problema . . . 92
6.2.2 Desenvolvimento da Aplicação . . . 94

6.3 Estudo de Caso: Equações Lineares 99
6.3.1 Descrição 99
6.3.2 Desenvolvimento da Aplicação 100
6.4 Estudo de Caso: Simfra 105
6.4.1 Descrição 105
6.4.2 Desenvolvimento da Aplicação 106
6.5 Estudo de Caso: Ordenação de Inteiros 109
6.5.1 Descrição do Problema 109
6.5.2 Desenvolvimento da Aplicação 114
6.6 Considerações Finais 116

7 Conclusão **117**
7.1 Contribuições 117
7.2 Produtos Gerados 118
7.3 Perspectivas e Trabalhos Futuros 119

Referências e Bibliografia **121**

Agradecimentos

A DEUS pela oportunidade e pelo privilégio que me foi dado em participar deste Doutorado e aprender com mais profundidade sobre um tema que me interessava há algum tempo.

À minha mãe pelo amor incondicional, pelo incentivo e apoio em todos os momentos.

Ao meu orientador Prof. Ricardo Cordeiro Corrêa pelo incentivo, compreensão e presteza no auxílio às atividades e discussões sobre o andamento desta Tese de Doutorado.

À minha irmã pelo carinho, pelo apoio emocional e suporte logístico.

Ao meu pai e meu irmão caçula pela presença e incentivo.

Ao meu esposo pelo amor, pela paciência de ler várias vezes o texto, incentivo e em tolerar a minha dedicação à Tese.

Ao professor Heron pelo incentivo, idéias de aplicações, sugestões de conferências, ajudas nos artigos e pelas sugestões de melhoria nas avaliações da Qualificação e da Proposta.

Ao professor Joaquim Bento pelos conselhos e sugestões de melhoria nas avaliações da Qualificação e da Proposta.

À professora Cláudia pelo apoio emocional e ajuda durante o Doutorado.

A todos os professores do Programa que de alguma forma contribuiram para a realização desse trabalho.

Ao Daniel (LIA), Juliana (LIA), Débora (LIA) e Marcos (Computação Gráfica) pela ajuda, pela espontaneidade e alegria na troca de informações e prontidão nas dificuldades do dia-a-dia.

E, finalmente, aos demais idealizadores, coordenadores e funcionários da Universidade Federal do Ceará.

Resumo

Este trabalho apresenta o *Forró*, uma *plataforma* de componentes para aplicações científicas compatível com o modelo de componentes *Common Component Architecture* (CCA) usando conectores endógenos e exógenos. As principais contribuições do presente trabalho foram: proposta de conceitos ***link*** e ***espaço de conexão*** para componentes distribuídos; e, proposta de utilização de conectores endógenos, conectores exógenos e ***dutos*** no CCA, os quais ainda não tinham sido utilizados em nenhuma outra plataforma CCA. O presente trabalho se dividiu em três etapas. A primeira etapa foi a proposta de novos conceitos em componentes distribuídos. A segunda etapa foi a implementação de uma plataforma CCA e conectores - exógenos e endógenos. Finalmente, a terceira etapa consistiu na modelagem, execução e testes com aplicações de CAD usando a plataforma proposta, chamada *Forró*. Os resultados dos estudos de caso foram satisfatórios em relação ao tempo de execução. Baseado neste estudo é possível afirmar que a plataforma proposta pode apresentar-se como uma solução viável para composição de aplicações a partir de métodos ou código legado.

Abstract

This work presents *Forró*, a component platform for scientific applications CCA model compliant using endogenous and exogenous connectors. The main contributions of this study were: concepts proposal of ***link*** and ***Connecting Space*** for distributed components, and proposal of use endogenous connectors, exogenous connectors and ***ducts*** in CCA model, which had not been used in any other platform CCA. This work was divided into three stages. The first stage was the proposal of new concepts in distributed components. The second stage was the implementation of a CCA platform and exogenous and endogenous connectors. Finally, the third stage was modeling, implementing and testing *High Performance Computing* (HPC) applications using the proposed platform, called *Forró*. The results of case studies were satisfactory regarding the runtime. Based on this study it can be said that the proposed platform can present itself as a possible solution for applications composition from legacy code or methods.

Lista de Figuras

2.1 Taxonomia do paralelismo. 10
2.2 Estrutura de memória fisicamente compartilhada. 12
2.3 Estrutura de memória fisicamente distribuída. 13
2.4 Arquitetura de *Cluster*. 24
2.5 Recursos de grade vinculados a aplicação de Teleciência do neurocientista Mark Ellisman (http://www.npaci.edu/Alpha/telescience.html). 28

3.1 Exemplo de uma conexão de componentes usando CCA. 35
3.2 Diagrama do padrão CCA de projeto *Uses/Provides* 38
3.3 Exemplo de interação de componentes paralelos CCA. 39
3.4 Diagrama do Padrão SCMD. 40
3.5 Representação de Processos para Componentes #. 48
3.6 Separação da computação nos processos envolvidos. 50
3.7 Arquitetura *Hash*. 54

4.1 Um exemplo de conexão "clandestina" entre dois componentes. 59
4.2 Elementos do novo modelo de conexão do *Forró*. 61
4.3 Ilustração de um conector endógeno. 62
4.4 Uma conexão direta entre dois componentes CCA. 63
4.5 Uma conexão indireta através de conectores endógenos e ***links*** especializados. 64
4.6 Conector indireto através de conectores endógenos implementando um canal de uma aplicação orientada a eventos. 65
4.7 Componentes conectados por um conector exógeno. 66
4.8 Conectores exógenos e conector indireto através de conectores endógenos implementando um canal de uma aplicação baseada em pulsos. 68

5.1 Plataforma *Forró*. 71
5.2 Arquitetura da Plataforma *Forró*. 72
5.3 Arquitetura de um componente *Forró*. 73
5.4 Elementos da plataforma *Forró*. 74
5.5 Um conector indireto típico no *Forró* formado por quatro componentes. . 78
5.6 Diagrama de classes *Forró* e *interfaces* CCA. 79

5.7 Associação entre componentes na plataforma *Forró* e os módulos em código nativo. 82
5.8 Passo Inicial da comunicação entre dois componentes remotos. 84
5.9 Passo 1 da comunicação entre dois componentes remotos. 85
5.10 Passo 2 da comunicação entre dois componentes remotos. 85
5.11 Passo 3 da comunicação entre dois componentes remotos. 86
5.12 Passo 4 da comunicação entre dois componentes remotos. 86
5.13 Representação da invocação de um método. 87
5.14 Uma conexão indireta através de conectores endógenos e ***links*** especializados no *Forró*. 88
5.15 Aplicação MPI. 89
5.16 Uso de biblioteca paralela. 89
5.17 Modelo baseado em eventos. 90

6.1 Fluxograma do método geral utilizado para obter limites inferiores e superiores para $\alpha(G, k)$. 95
6.2 Componentes, conectores e portas para determinação de limites inferiores e superiores para o problema Subconjunto k-Particionável Máximo (SkPM). 98
6.3 Pontos que a plataforma pode trazer alguma sobrecarga. 99
6.4 Classes e componentes do Estudo de Caso 1 - Equações Lineares. 100
6.5 Ambiente integrado para análise de elementos finitos. 106
6.6 Tipos de visões do Simfra. 106
6.7 Classes e componentes do Estudo de Caso 2 - Simfra. 107
6.8 *Ping-pong* assíncrono. 112
6.9 *Ping-pong* assíncrono. 113
6.10 *Ping-pong* assíncrono. 113
6.11 *Benchmark* paralelo de ordenação de inteiros na classe A. 114
6.12 *Benchmark* paralelo de ordenação de inteiros na classe B. 115
6.13 *Benchmark* paralelo de ordenação de inteiros na classe C. 115

Lista de Tabelas

2.1 Invocação de método remoto. 26

3.1 Comparação das características dos tipos de códigos: monolítico, baseado em bibliotecas e baseado em componentes. 34
3.2 Implementação do método `setServices` do "Componente 1" na qual é declarada uma porta *uses*. 37
3.3 Implementação do método `setServices` do "Componente 2". O "Componente 2" fornece uma porta *provides* para qualquer componente utilizar. Neste exemplo, o "Componente 2" pode ser considerado "Servidor". 37
3.4 Implementação da *interface* `GoPort` do "Componente 1". 37
3.5 Implementação do método `methodA` do "Componente 2" 37
3.6 Características do Ccaffeine 42
3.7 Características do XCat 43
3.8 Características do DCA 44
3.9 Características do SCIRun 44
3.10 Características do MOCCA 45
3.11 Características do Decaf 46

5.1 Código do componente encapsulador de um componente nativo em C++. 81

6.1 Trecho de código do componente encapsulador `ComponentExample1` no *Forró*. 102
6.2 Comandos de preparação e inicialização do ambiente local. 103
6.3 Comandos de preparação e inicialização. 103
6.4 Comandos de execução no ambiente *Forró*. 104
6.5 Componente e porta Java `MarchingCubes` que encapsula o código C++ do Simfra. . . 108
6.6 Comandos de execução no ambiente *Forró*. 109
6.7 Tamanhos de Problemas dos *Benchmarks* Paralelos NAS 110

Acrônimos

ADL *Architecture Description Language*

API *Application Programming Interface*

CAD Computação de Alto Desempenho

CCA *Common Component Architecture*

CFD *Computational Fluid Dynamics*

CIM Conjunto Independente Máximo

CIPM Problema do Conjunto Independente de Peso Máximo

CORBA *Common Object Request Broker Architecture*

DCA *Distributed CCA Architecture*

DIMACS *Second DIMACS Implementation Challenge for Cliques, Coloring, e Satisfiability*

DMMP *Distributed-Memory, Message-Passing*

DMSV *Distributed-Memory, Shared-Variable*

FEMOOP *Finite Element Method - Object Oriented Programming*

GMMP *Global-Memory, Message-Passing*

GMSV *Global-Memory, Shared-Variable*

GUI *Graphical User Interface*

HCL *Hash Configuration Language*

HOCC *Hash Overlapping Composition Calculus*

HPC *High Performance Computing*

HPE *Hash (#) Programming Environment*

IDL *Interface Descriptor Language*

I/O *Input/Output*

JNI *Java Native Interface*

JNDI *Java Naming and Directory Interface*

MEF Método de Elementos Finitos

MISD *Multiple Instruction, Single Data*

MIMD *Multiple Instruction, Multiple Data*

MOM *Message-Oriented Middleware*

MPI *Message Passing Interface*

MPMD *Multiple Process, Multiple Data* ou *Multiple Program, Multiple Data*

MVC *Model-View-Controller*

NAS *Numerical Aerodynamic Simulation*

NPB *NAS Parallel Benchmarks*

OMG *Object Management Group*

OpenGL *Open Graphics Library*

PRMI *Parallel Remote Method Invocation*

RMI *Remote Method Invocation*

RPC *Remote Procedure Call*

SCMD *Single Component/Multiple Data*

SIDL *Scientific Interface Definition Language*

SIMD *Single Instruction, Multiple Data*

SISD *Single Instruction, Single Data*

SkPM Subconjunto k-Particionável Máximo

SOAP *Simple Object Access Protocol*

SPMD *Single Process, Multiple Data* ou *Single Program, Multiple Data*

STL *Standard Template Library*

UC *Unidade de Controle*

UCP Unidade Central de Processamento (em inglês *Central Processing Unit*)

UCPs Unidades Centrais de Processamento (em inglês *Central Processing Units*)

UDDI *Universal Description, Discovery and Integration*

UP unidade de processamento

Capítulo 1

Introdução

Esta tese tem como contexto a área da programação baseada em componentes direcionada a aplicações de Computação de Alto Desempenho (CAD). Esta área entra em um campo que está na fronteira atual delimitada pelo desafio inicial de construir e utilizar artefatos para fazer contas até a disseminação do uso de computadores em Ciências, Engenharias e todas outras áreas do conhecimento. Devido à vastidão de detalhes nesta evolução, a seguir tem-se apenas um breve resumo do desenvolvimento desde o desafio humano de fazer "contas", interligando as evoluções da Computação, dos artefatos e do pensamento "algorítmico".

O homem já utilizava dispositivos mecânicos para simplificar cálculos milhares de anos atrás. Por exemplo, o ábaco provavelmente existiu na Babilônia cerca de 3.000 anos antes da nossa era. Daí até 1640, os inventos andaram a passos muito lentos. Até que em 1642, B. Pascal [1, pág.07] construiu um dispositivo mecânico com operações fundamentais de adição e subtração. Em 1671, G. W. Leibniz [1, pág.07] desenvolveu uma máquina que conseguia não só somar e subtrair, mas também multiplicar e dividir. Em 1822, C. Babbage [1, 2, pág.13,pág.12], construiu uma pequena máquina diferencial e em 1832, desenvolveu o princípio da máquina analítica, precursora dos computadores atuais. Em 1869, W. Jevons [1, pág.272], construiu uma máquina para resolver problemas lógicos. Após alguns anos, o trabalho nos dispositivos de calcular se intensificou. Na década de 40, os cálculos necessários para a balística durante a segunda guerra mundial impulsionaram ainda mais o desenvolvimento de dispositivos de cálculos avançados. Em 1890, H. Hollerith [3, pág.86], incorporando ideias da máquina analítica de Babbage, usou uma máquina eletromecânica que utilizava cartões perfurados para analisar informações do censo. Por volta de 1944, H. Aiken [3, pág.155] completa uma máquina eletromecânica, chamada Mark I, a qual combina várias máquinas estatísticas do tipo Hollerith. Por volta de 1945, o ENIAC [4, pág.47], identificado como o primeiro computador eletrônico digital, foi finalizado e disponibilizado para testes. Do ponto de vista de *software*, o seu desenvolvimento começou mais tardiamente. Por exemplo, em 1951, G. Hopper [5, pág.92] inventou a noção de um compilador para um computador chamado UNIVAC. Nos tempos atuais o desenvolvimento se dá em um ritmo acelerado, principalmente após a década de 60, quando a Arpanet, uma precursora da Internet de hoje, começou a ser construída.

Na década de 80, houve a ascensão do computador pessoal. A partir daí, as aplicações de *hardware* e *software* se popularizaram e se desenvolveram em uma velocidade muito grande, disseminando-se em diversos ramos das sociedades atuais.

Uma vez que as empresas de supercomputadores romperam com uma barreira técnica, as empresas de microprocessadores puderam adotar rapidamente os elementos de sucesso dos projetos dos supercomputadores poucos anos mais tarde. O segundo fator, e talvez mais importante foi o surgimento de um próspero mercado de computadores pessoais e de negócios com as exigências de desempenho cada vez maior. Utilizações de computador, tais como gráficos em 3D, interfaces gráficas, multimídia e jogos foram fatores determinantes neste mercado. Como resultado, surgiu uma área chamada Computação de Alto Desempenho (CAD) [6, pág.18], que é, em termos gerais, o ramo da Ciência da Computação que se concentra em executar, explorando o máximo do desempenho, uma ampla gama de sistemas em computadores pessoais até grandes sistemas de processamento paralelo.

O campo de CAD obteve destaque através de avanços em tecnologias de eletrônica e integrada a partir dos anos 1940. Desde então houveram alguns avanços como *hyperthreading* em processadores Intel, computação em cluster e em grades. Para explorar plenamente esses avanços, os pesquisadores e profissionais da indústria começaram a projetar sistemas de *software* paralelo e/ou distribuído e algoritmos para lidar com grandes e complexos problemas de engenharia e problemas científicos [7, pág.25 - xxiii].

Um princípio que aparece naturalmente nessa disciplina é o uso do paralelismo para desenvolver programas que possam ser divididos em partes menores que possam ser executadas simultaneamente. A promessa de paralelismo tem fascinado pesquisadores por pelo menos três décadas. No passado, os esforços de computação paralela e os investimentos mostraram que no final, a computação em um único processador sempre prevalecia. No entanto, verificamos que a computação de propósito geral de computação está dando um passo irreversível em direção a arquiteturas paralelas. Essa mudança na direção do paralelismo não está baseada em novas arquiteturas de paralelismo, ao invés disso, baseia-se nas arquiteturas tradicionais [8, pág.7]. Um desafio que surge naturalmente é o contínuo aprimoramento das tecnologias de comunicação e do conhecimento em algoritmos paralelos, fazendo a área de CAD evoluir. É dessa forma que o desenvolvimento da área de CAD envolve a adequada combinação de arquiteturas de computadores, linguagens de programação e algoritmos. A alta complexidade dos problemas e a integração de diversas áreas do conhecimento são características comuns às aplicações de ciências computacionais atuais, na busca da viabilização do processamento de alto desempenho.

Ainda na busca do alto desempenho, houve também uma explosão no crescimento de sistemas distribuídos, o que tornou imperativo entender como superar as seguintes dificuldades: presença de *hardware* e *software* heterogêneos e a falta de aderência dos padrões existentes. Um sistema distribuído [9, pág. 15] é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens.

Com o crescimento da disponibilidade de sistemas distribuídos e sistemas paralelos, surgiu o problema de reaproveitamento (reuso) do código existente nestes novos sistemas. Para reusar os códigos existentes, surgiram as seguintes dificuldades: incompatibilidades de linguagens, falta da padronização na comunicação e falta de acoplamento das interfaces são dificuldades enfrentadas para o reuso de código [10]. Estas características estão sendo necessárias, especialmente, na computação científica de alto desempenho, onde simulações de alta fidelidade e multi-física são cada vez mais complexas e requerem, frequentemente, equipes multidisciplinares de pesquisa. Por exemplo, uma abordagem clássica utilizada em simulações de sistemas de comunicações móveis sem fio é fazer uma separação do sistema em dois níveis de acordo com a variabilidade temporal dos parâmetros envolvidos. Em particular, utiliza-se a separação em nível de enlace e em nível sistêmico para alcançar as metas estabelecidas com o simulador. A simulação ao nível do enlace é feita por engenheiros. Já a modelagem dos fenômenos físicos presentes na simulação sistêmica é feita por físicos [11].

Para ajudar nas dificuldades citadas do reuso e heterogeineidade em aplicações genéricas surgiu a programação usando componentes é uma solução de Engenharia de *Software*. A base da programação usando componentes é o encapsulamento de unidades de funcionalidades bem especificadas, com estabelecimento de regras a serem seguidas para a interação entre as unidades. A construção de um programa para uma determinada aplicação consiste no adequado estabelecimento de um conjunto de tais unidades, chamadas de *componentes*, designadas a partir das especificações de suas funcionalidades, e da correta interação existente entre essas unidades. Em um sistema baseado em componentes, conectores são utilizados para combinar componentes. Os conectores precisam ter uma semântica que os torne simples em termos de construção e utilização. Ao mesmo tempo, sua semântica deve ser rica o suficiente para dotá-los de propriedades desejáveis, tais como generalidade e reutilização. Observe-se que os detalhes de implementação não são relevantes para a construção da aplicação, bastando, conforme mencionado, estabelecer adequadamente as suas especificações.

Devido ao fato de as aplicações de CAD terem alcançado um estágio em que o seu desenvolvimento trata todos os desafios oriundos das suas características já mencionadas (heterogeneidade, multidisciplinaridade etc), o uso da programação por componentes vem tornando-se um ponto de convergência entre CAD e Engenharia de *Software*. Os trabalhos [12, 13, 14] são exemplos desta abordagem. Porém, essa convergência não é total em razão da dicotomia entre a abrangência das soluções usualmente propostas em Engenharia de *Software* versus as particularidades, sobretudo quanto à eficiência, das aplicações de CAD.

Em [15], afirma-se que a ideia da utilização de plataformas de componentes para lidar com a complexidade do desenvolvimento de aplicações interdisciplinares de CAD está se tornando cada vez mais popular. Estes sistemas permitem que os programadores acelerem o desenvolvimento através de abstrações de alto nível, permitem reuso de código, bem como fornecem *interfaces* de componentes bem especificadas que facilitam a tarefa de interação da equipe.

Na presente tese, propomos expandir o domínio de aplicações de programação por

componentes para CAD ao incluir uma classe de aplicações ainda inexplorada. Trata-se da classe de aplicações de otimização combinatória [16], a qual possui algumas características que a diferem da classe de aplicações de simulações numéricas. Otimização combinatória é um termo que descreve as áreas de programação matemática que estão preocupadas com a solução de problemas de otimização tendo uma representativa estrutura discreta ou combinatorial [17]. De uma forma geral, um problema de otimização combinatória é definido como o problema de encontrar um elemento de um conjunto finito e enumerável que maximiza ou minimiza uma função definida nesse conjunto. A dificuldade reside no fato de, apesar de finito, o conjunto em questão é, em geral, grande demais para que a simples enumeração de todos os seus elementos seja viável. A alternativa, então, é explorar propriedades estruturais do conjunto de forma a reduzir a quantidade de elementos a serem explicitamente examinados na enumeração. Em geral, a estrutura do conjunto é conhecida apenas parcialmente, de forma que, apesar de reduzido, um número considerável de elementos ainda precisa ser enumerado. Devido a esse fato, muitos algoritmos apresentam um paralelismo, seja na possibilidade de enumeração simultânea de diferentes soluções, seja na própria determinação das propriedades estruturais acima mencionadas. No entanto, há uma dificuldade na exploração desse paralelismo potencial que decorre de a enumeração evoluir de maneira imprevisível. Como consequência, a repartição dos dados das possíveis soluções a serem analisadas deve ser continuamente ajustada à medida que a enumeração avança. Neste trabalho, a classe de problemas de otimização escolhido foi a classe de problemas de conjuntos independentes [18, pág.69].

O CCA (do inglês *Common Component Architecture*) é um modelo de componentes para CAD que pode ser visto como um modelo de componentes de baixo nível em uma inclusão de modelos que dêem sustentação a conexões mais sofisticadas. O modelo CCA é utilizado em várias aplicações de simulação numérica, como por exemplo em [19, 20]. O CCA apresenta ainda algumas deficiências como por exemplo não estabelece como as conexões são realizadas. Então, nesta tese propomos expandir o modelo de conexão do CCA e propomos a sua utilização em aplicações de otimização combinatória. Esta expansão tem o objetivo de incluir tipos de conectores que ainda não haviam sido utilizados até o momento em plataformas CCA. Dentre esses tipos de conectores, destaca-se a inclusão de *conectores exógenos* no modelo, com os quais podemos descrever modelos de computação úteis na nova classe de aplicações. Um *conector endógeno* conecta componentes distintos ou não. O termo "endógeno" ressalta o fato de que essas interações se dão por meio de invocações de métodos originadas por um dos componentes que estão sendo conectados. Um *conector exógeno* estabelece uma sequência de interações entre componentes. Essas interações são associadas a invocações de métodos e originadas internamente no conector, justificando o uso do termo "exógeno".

Por conseguinte, verificamos que quando o próprio componente efetua o fluxo de controle, o fluxo de dados e a computação são altamente acoplados, mesmo que em um grau variante dependendo do nível de indireção na passagem de mensagens. De qualquer maneira neste tipo de conexão, o controle e computação estão juntos. Com o objetivo de sugerir uma melhoria em plataformas CCA, pensamos em minimizar o

acoplamento e adicionar características para a conexão. Com este intuito, sugerimos o uso de conectores endógenos e exógenos como uma forma de especificar e controlar a interação dos componentes em uma plataforma CCA.

A preocupação principal da pesquisa reportada nesta tese é a busca por uma abordagem de desenvolvimento de aplicações baseadas em componentes que, além de oferecer liberdade ao desenvolvedor para criar suas aplicações a partir da composição de componentes, pudesse facilitar a tarefa de adaptação e combinação dessas funcionalidades de forma a garantir um bom desempenho.

Com essa preocupação em mente, fizemos um levantamento na literatura para identificar os aspectos mais importantes para o suporte à integração de sistemas que requerem alto desempenho. Apesar dessa proposta ter sido inicialmente investigada para resolver este problema de integração destes sistemas, uma análise sobre os requisitos e características estabelecidas para a plataforma revelou a possibilidade de generalizá-la para o desenvolvimento de aplicações distribuídas e paralelas de CAD em geral.

Portanto, com base na análise das características de CAD descritas no Capítulo 2 e nos estudos sobre os modelos de componentes e plataformas existentes descritas no Capítulo 3, é proposta nesta tese uma plataforma de componentes denominada plataforma *Forró*.

A escolha pela abordagem baseada em componentes se deu em função da capacidade de reuso, interoperabilidade e extensão inerentes nesse tipo de aplicação de otimização combinatória. Feita a escolha, por ser uma alternativa baseada em componentes e na padronização de *interfaces* de comunicação, a proposta de uso de uma plataforma de componentes mostrou-se adequada para permitir que diferentes aplicações de um dado domínio pudessem ser instanciadas e combinadas de maneira organizada.

As principais contribuições da tese, neste ponto da expansão do modelo de conexões do CCA, tiveram como origem as discussões em [21, 22, 23, 24], as quais fazem parte da tese.

Neste cenário, o principal objetivo da presente tese é propor uma adaptação viável do CCA, principalmente, para uma classe de aplicações de alto desempenho ainda inexplorada pelo CCA, a qual se trata da classe de aplicações de otimização combinatória. Os pesquisadores do CCA, pela origem nas Engenharias e nas Ciências, estão centrados em aplicações numéricas, envolvendo muitas operações matriciais.

As principais contribuições do presente trabalho foram: proposta de conceitos (inspirado em [25]) ***link*** e ***espaço de conexão*** para componentes distribuídos; e, proposta de utilização de conectores endógenos, conectores exógenos e ***dutos*** no CCA, os quais ainda não tinham sido utilizados em nenhuma outra plataforma CCA. Um ***espaço de conexão*** é definido por um certo conjunto de conectores, parametrizados pela mesma implementação de ***dutos***. As conexões realizadas pelos conectores desse ***espaço de conexão*** garantem uma série de propriedades globais. Um ***espaço de conexão*** pode ser usado, por exemplo, para implementar um certo modelo de computação.

Para realizar as contribuições, implementamos uma plataforma compatível com o modelo de componentes CCA para aplicações distribuídas. A plataforma adota uma abordagem que usa ***espaço de conexão*** (definido no Capítulo 5) e conectores endógenos

e exógenos para garantir a interoperabilidade de sistemas heterogêneos. Esta abordagem foi inspirada nas recentes pesquisas em plataformas e modelos de componentes, como por exemplo [26, 27]. A escolha da linguagem Java para implementação da plataforma se deve à facilidade de reuso e à independência de *hardware*.

Finalizando, o presente trabalho se dividiu em três etapas. A primeira etapa foi a proposta de novos conceitos em componentes distribuídos (explicada nos Capítulos 4 e 5). A segunda etapa foi a implementação de uma plataforma CCA e conectores - exógenos e endógenos (explicada no Capítulo 5). Finalmente, a terceira etapa (explicada no Capítulo 6) consistiu na modelagem, execução e testes com aplicações de CAD usando a plataforma proposta, chamada *Forró*.

Organização da Tese

Este texto está organizado da seguinte forma: o Capítulo 2 descreve os conceitos de Computação de Alto Desempenho (CAD), o Capítulo 3 mostra o conceito de componentes e o estado da arte em modelos e plataformas (ou *frameworks* no contexto CCA); o Capítulo 4 apresenta o uso de conectores exógenos e endógenos no CCA; o Capítulo 5 detalha a contribuição do presente trabalho apresentando a plataforma *Forró* proposta; o Capítulo 6 descreve o estudo de caso. Em seguida, o Capítulo 7 apresenta a conclusão contendo as contribuições e as perspectivas; e, por último, são apresentados dois exemplos no Apêndice.

Dois capítulos são dedicados à contextualização do tema desenvolvido. Dedicamos o Capítulo 2 a uma descrição, em uma forma esquemática, do panorama atual dos principais conceitos de CAD. Nessa descrição, procuramos relacionar as principais soluções arquiteturais entre si. A abordagem seguida não é a de fornecer um levantamento detalhado do estado-da-arte, mas de enfatizar a diversidade e de destacar os elementos suficientes para motivar as contribuições que apresentaremos nos capítulos seguintes. Seguindo a mesma linha, discutimos algumas linguagens e ferramentas de *software* largamente empregadas em CAD sob a perspectiva da programação. Finalmente, mostramos seu impacto sobre as aplicações do ponto de vista do desempenho. Esta última discussão é essencial para dar o devido destaque à questão essencial da área de CAD.

Para dar continuidade à contextualização, passamos no Capítulo 3 a apresentar os modelos de componentes atualmente empregados em CAD. Iniciamos esse capítulo com uma exposição de uma noção de componente que, apesar de apresentada de forma abstrata, nos permite descrever o seu emprego nas aplicações de nosso interesse de uma forma genérica. Ainda nesse mesmo capítulo, passamos a uma discussão um pouco mais específica, pois apresentamos três modelos de componentes em detalhes de forma a tornar mais precisos os desafios envolvidos e os meios que têm sido empregados para enfrentá-los.

Após os capítulos de contextualização, passamos aqueles que incluem as contribuições desta tese. O primeiro deles é o Capítulo 4, no qual discutimos questões referentes ao emprego de conectores no modelo de componentes CCA. Essas questões surgem em torno

de dois aspectos principais. O primeiro, naturalmente, é o desempenho das conexões, sempre relevante nas aplicações de CAD. O segundo é a capacidade de expressar as interações entre componentes presentes nas diversas aplicações de CAD. Nesse aspecto, lançamos nesta tese um maior destaque a particularidades de aplicações em otimização combinatória. Propomos, então, o emprego de conectores endógenos e exógenos, em conexões diretas e indiretas, para implementar diferentes modelos de computação paralela ou distribuída. O segundo deles é o Capítulo 5, no qual mostramos as características da implementação da plataforma CCA. Em seguida, mostramos o terceiro deles que é o Capítulo 6, no qual discutimos os estudos de caso que realizamos utilizando a plataforma CCA. Por último, no Capítulo 7, mostramos as contribuições, enumeramos os produtos gerados e discutimos as perspectivas futuras de desenvolvimento da plataforma *Forró*.

Capítulo 2

Computação de Alto Desempenho

Computação de Alto Desempenho (CAD) engloba sistemas que devem ser cuidadosamente concebidos de modo que cumpram rigorosos requisitos, principalmente, requisitos de desempenho [28]. O desempenho alcançado por uma aplicação quando executada em um determinado sistema computacional é uma medida que depende de dois fatores essenciais [28]. O primeiro deles é o poder computacional potencial do sistema em questão, o qual pode ser entendido como a capacidade de processamento que o sistema oferece. O segundo fator é a capacidade da aplicação explorar eficientemente o poder potencial disponível. Ao longo deste capítulo, traçamos um panorama dos principais aspectos que podemos identificar como conceitos dominantes no trato das duas questões levantadas.

Um breve histórico da evolução das arquiteturas de CAD:

- Supercomputadores - computadores que eram maiores, mais rápidos e mais complicados para os computadores disponíveis na época - nos anos 80;
- *Clusters* - coleção de computadores interconectados autônomos trabalhando juntos como um recurso único e integrado - nos anos 90;
- Computação em grade - computação e a infra-estrutura de gerenciamento de dados que fornece o apoio eletrônico para uma comunidade global em qualquer área - nos anos 90;
- *Cloud Computing* - utilização da memória e da capacidade de cálculo de computadores e servidores compartilhados e interligados através da Internet (princípio da computação em grade) - nos anos 2000.

Neste capítulo, são abordados, como forma de revisão, os conceitos relativos à Computação de Alto Desempenho (CAD) associados a este trabalho. Na Seção 2.1, mostramos uma visão geral sobre a taxonomia do paralelismo e apresentamos os principais paradigmas. Na Seção 2.2, descrevemos as soluções arquiteturais. Em seguida, na Seção 2.4, explicamos os principais conceitos usados na programação paralela e distribuída. Finalmente, na Seção 2.5, descrevemos conceitos relacionados ao desempenho.

2.1 Taxonomia do Paralelismo

Começamos nossa exposição por um levantamento das soluções comumente adotadas pelos sistemas atuais para alcançar alto poder computacional. Duas vias têm sido privilegiadas. A primeira delas é o investimento no desenvolvimento dos circuitos eletrônicos. É evidente que esse desenvolvimento propicia um aumento no desempenho dos sistemas. No entanto, há limites físicos para o que se pode esperar por essa via [29]. Por essa razão, é essencial explorar uma segunda via, aquela que busca soluções arquiteturais para levar o desempenho para além do que a tecnologia de circuitos eletrônicos torna disponível. Tais soluções apoiam-se em um princípio básico: paralelismo. O termo traduz a ideia de realizar, simultaneamente, operações que sejam independentes. Para isso, existe a necessidade de replicação de unidades de processamento. Unidades de armazenamento também podem ser replicadas.

O termo unidade de processamento (UP) é empregado acima com um significado bastante amplo, denotando qualquer artefato de *hardware* capaz de realizar algum processamento. Posto dessa forma, o conceito de paralelismo torna-se aplicável em diversos níveis. É fato comum, atualmente, a presença de várias unidades lógico-aritméticas em um mesmo microprocessador, assim como a utilização de diversos microprocessadores em um mesmo sistema computacional.

Situação semelhante pode ser observada quanto às unidades de armazenamento. Mesmo guardando as suas particularidades (quando comparadas às unidades de processamento), o conceito de replicação também pode ocorrer em diferentes níveis. A fim de traçar um panorama mais objetivo da presença do paralelismo, utilizamos uma taxonomia para guiar a exposição.

Em geral, os computadores digitais podem ser classificados em quatro categorias, de acordo com a multiplicidade de instrução e fluxos de dados. Michael J. Flynn introduziu este regime ou taxonomia para classificar as organizações de computador em 1966. A classificação de Flynn tornou-se padrão e é amplamente utilizada. Flynn criou as abreviaturas *Single Instruction, Single Data* (SISD), *Single Instruction, Multiple Data* (SIMD), *Multiple Instruction, Single Data* (MISD) e *Multiple Instruction, Multiple Data* (MIMD) para as quatro classes de computadores. Em 1988, E. Johnson [30] propôs subdivisões para a classe MIMD baseadas na estrutura de memória e no mecanismo de comunicação utilizado, sobre as quais falamos posteriormente. Na Figura 2.1, podemos observar as categorias de Flynn e as categorias de Johnson.

O elemento essencial do processo de computação é a execução de uma sequência de instruções sobre um conjunto de dados. As organizações de computadores são caracterizadas pela variedade de *hardware* fornecido para o serviço de instrução e fluxos de dados. Estão listadas abaixo estão as quatro organizações de máquina de Flynn [31, pag.17]:

- SISD (do inglês *Single Instruction, Single Data* (SISD)): fluxo de instrução única/ fluxo de dados único

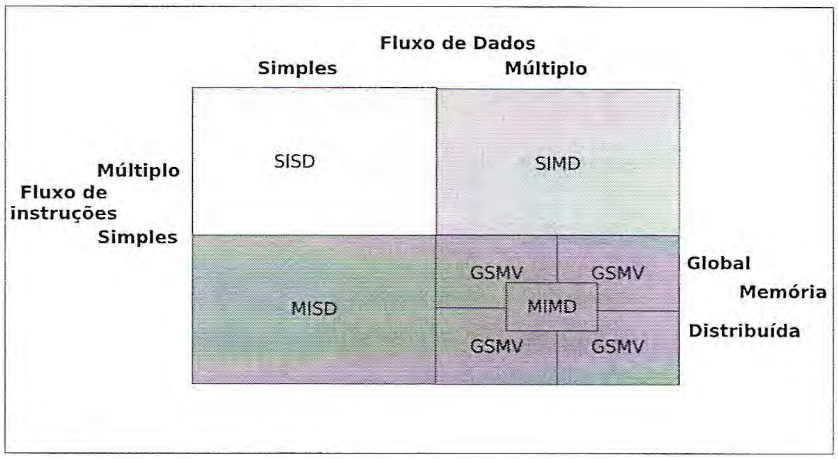


Figura 2.1: Taxonomia do paralelismo.

- SIMD (do inglês *Single Instruction, Multiple Data* (SIMD)): fluxo de instrução única / fluxo de dados múltiplo
- MISD (do inglês *Multiple Instruction, Single Data* (MISD)): fluxo de instrução múltipla / fluxo de dados único (sem aplicação real [31, pag.17])
- MIMD (do inglês *Multiple Instruction, Multiple Data* (MIMD)): fluxo de instrução múltipla / fluxo de dados múltiplo

A organização SISD representa a maioria dos computadores disponíveis hoje. As instruções são executadas sequencialmente, mas, podem ser sobrepostas em suas fases de execução. A maioria dos sistemas uniprocessados possui *pipeline*. Um computador SISD pode ter mais de uma UP. Todas as UPs estão sob a supervisão de uma *Unidade de Controle* (UC).

Computadores da classe SIMD possuem um conjunto de UPs que operam de forma paralela e síncrona, executando uma mesma operação em cada pulso do relógio global (sincronismo é melhor explicado posteriomente ainda neste Capítulo). O paralelismo está no fato de a instrução comum ser efetuada sobre dados diferentes nas diversas UPs.

Abstratamente, o MISD [32] é um *pipeline* de múltiplas unidades funcionais executando independentemente operando em um único fluxo de dados, esperando resultados entre uma unidade funcional e a próxima unidade funcional.

De acordo com [30], a categoria MIMD abrange uma grande classe de computadores e, atualmente, esta categoria é a mais difundida para CAD. Esta classe merece uma descrição mais detalhada por ser o enfoque desta tese.

Na classe MIMD [31], encontram-se classificados a maioria dos sistemas com múltiplos processadores e sistemas de computadores. Um computador MIMD intrínseco implica em interações entre os processadores porque todos os fluxos de memória são derivados do mesmo espaço de dados compartilhado por todos os processadores.

A classe MIMD pode ser resumida como segue:

- Cada processador executa sua própria sequência de instruções.
- Cada processador trabalha em uma parte diferente do problema.
- Cada processador comunicam dados para outros processadores.
- Processadores podem ter que esperar por outros processadores ou esperar para acessar dados.

Em 1988, conforme citado anteriormente, E. Johnson propôs classificações para subdividir a classe MIMD. Segundo [30], as classificações foram baseadas em características relativas à estrutura de memória (global ou distribuída) e ao paradigma de programação utilizado (variáveis compartilhadas ou troca de mensagens).

As quatro categorias criadas são [31]:

- GMSV (do inglês *Global-Memory, Shared-Variable* (GMSV)): Memória Global / variáveis compartilhadas.
- GMMP (do inglês *Global-Memory, Message-Passing* (GMMP)): Memória Global / passagem de mensagens (sem aplicação real).
- DMSV (do inglês *Distributed-Memory, Shared-Variable* (DMSV)): Memória Distribuída / variáveis compartilhadas.
- DMMP (do inglês *Distributed-Memory, Message-Passing* (DMMP)): Memória Distribuída / passagem de mensagens.

Os computadores da classe GMSV são conhecidos como multiprocessadores. No outro extremo, as máquinas da classe DMMP são denominadas multicomputadores. Um multicomputador pode consistir até mesmo de computadores completos, totalmente independentes e conectados apenas por uma rede local. Esse tipo de multicomputador recebe hoje em dia a denominação popular de *cluster* [33]. Finalmente, a classe DMSV, a qual está se tornando popular na perspectiva de aliar a facilidade de implementação de arquitetura distribuída com a facilidade de programação do ambiente de variável compartilhada. Os computadores desta classe são também conhecidos como computadores de memória compartilhada distribuída.

2.2 Soluções Arquiteturais

Nesta seção, são apresentadas as soluções arquiteturais utilizadas nos sistemas que podem ser enquadradas na classe MIMD, ressaltando os aspectos importantes que levam a ganhos de desempenho. Em [34], com a replicação de unidades de processamento e memória em um computador, podem ser formadas duas organizações básicas a seguir:

- **Estrutura de memória fisicamente compartilhada** (ver Figura 2.2), cada posição da memória possui endereço compartilhado por todas as Unidades Centrais de Processamento (em inglês *Central Processing Units*) (UCPs). Esse sistema também é chamado de sistema fortemente acoplado. Em um sistema de memória fisicamente compartilhada, a comunicação entre UCPs ocorre através de operações de leitura e escrita na memória compartilhada. Toda a coordenação e a sincronização entre os processadores também são realizadas através da memória global A plena exploração do potencial de paralelismo envolve um gerenciamento eficiente dos pontos de sincronismo no acesso a posições de memória comuns ao processamento sendo executado em cada UCP.

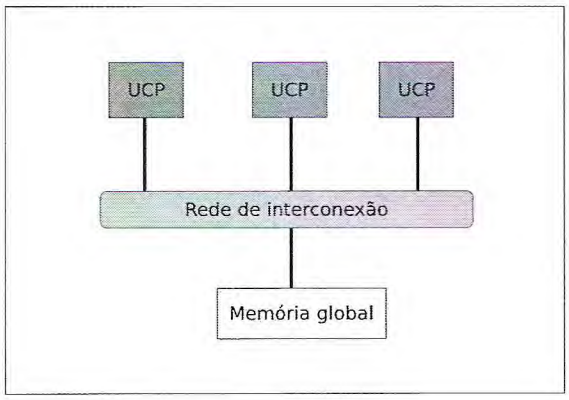


Figura 2.2: Estrutura de memória fisicamente compartilhada.

- **Estrutura de memória fisicamente distribuída** (ver Figura 2.3), cada UCP tem sua própria memória local. Os espaços de endereçamento das memórias locais são independentes. Dessa forma, cada UCP não pode endereçar uma posição de memória que não seja da sua própria memória local. Como consequência, as comunicações entre as UCPs são feitas por troca de mensagens através da rede de interconexão por um comando de envio no processador que está enviando a mensagem e um comando de recebimento no processador que irá receber a mensagem. As unidades de processamento de um sistema de troca de mensagens podem ser conectadas de várias formas que vão desde estruturas de interconexão local a redes dispersas geograficamente [35]. Novamente, é o gerenciamento das interações entre as UCPs que determinam a eficiência na exploração do paralelismo potencial de um sistema de memória distribuída. Neste caso, esse fato se traduz por um bom gerenciamento das comunicações.

O conceito de memória *cache* teve sua introdução com Wilkes em 1965 [36] ao distinguir um segundo tipo de memória além da memória principal. O objetivo é introduzir uma memória situada próximo ao processador, pequena e de alta velocidade de acesso, que concentre a maior parte dos acessos realizados em uma computação [36].

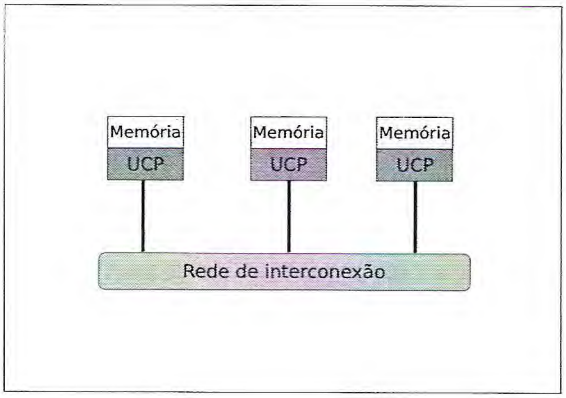


Figura 2.3: Estrutura de memória fisicamente distribuída.

2.3 Modelos

Esta seção dá uma visão de modelos usados correntemente na programação paralela e distribuída. Nessa descrição, as UPs são trocadas pela noção de processo. Um *processo* [37, pag.83] é uma entidade abstrata que executa tarefas.

2.3.1 Modelo Arquitetônico: Cliente/Servidor

Nesta subseção, descrevemos a arquitetura cliente-servidor. Esta arquitetura é definida por um servidor, oferecendo um conjunto de serviços, que atende a pedidos sobre esses serviços e clientes, os quais para que o serviço seja executado, enviam uma solicitação para o servidor através de uma biblioteca de troca de mensagens. Este modelo é baseado na distribuição de funções entre os dois tipos de processos independentes e autônomos: servidor e o cliente. Um cliente é qualquer processo que solicita serviços específicos a partir de um processo de um servidor. Um servidor é um processo que fornece serviços solicitados ao cliente. Os processos Cliente e Servidor podem residir no mesmo computador ou em diferentes computadores ligados por uma rede [38].

Um cliente é uma máquina que acessa recursos compartilhados de rede ou requer algum serviço a ser executado fornecidos por outra máquina (chamada de servidor). Um servidor é a máquina que atende as solicitações de uma máquina cliente.

O termo cliente/servidor é na realidade um conceito lógico. Os componentes cliente e servidor podem não existir em *hardware* físico distinto. Uma única máquina pode ser tanto um cliente e um servidor, dependendo da configuração do *software*. A tecnologia cliente/servidor é um modelo para a interação entre a execução de processos de software simultaneamente. O termo arquitetura refere-se a estrutura lógica e as características funcionais de um sistema, incluindo a forma como eles interagem uns com os outros em termos de *hardware*, *software* e as ligações entre eles.

Neste modelo, uma coleção de rotinas que são embutidas em uma aplicação para o envio, a recepção e outras operações de troca de informação. O servidor ou rejeita ou

faz o pedido e envia uma resposta para o cliente. Em outras palavras, cada nó oferece serviços a outros nós.

A aplicação distribui o trabalho entre a máquina local (cliente) e o servidor (back-end), dependendo da capacidade do cliente e produtos de servidor. Sistemas cliente/servidor são, frequentemente, muito eficientes porque minimizam o tráfego de rede, e porque cada parte do aplicativo pode ser otimizado para sua função.

Um exemplo de cliente pode ser a aplicação que apresenta os dados ao usuário. Como regra, o cliente não executa todas as funções de banco de dados, em vez disso, o cliente envia pedidos de dados a um servidor, depois, formata e exibe os resultados [39].

Os componentes da arquitetura cliente/servidor deve obedecer a alguns princípios básicos, se eles são para interagir apropriadamente. Estes princípios devem ser uniformemente aplicáveis a cliente, servidor e componentes de *middleware* de comunicação. Geralmente, esses princípios geram arquitetura cliente/servidor constitua a base sobre a qual a maioria da geração corrente de sistemas cliente/servidor são construídos. Alguns dos principais princípios são os seguintes: independência de *hardware*, independência de *software*, acesso a serviços, distribuição de processos, normas e independência de *hardware*.

A topologia Cliente/Servidor referem-se ao *layout* físico da rede cliente/servidor em que todos os clientes e servidores são conectados uns aos outros. Isso inclui todos as estações de trabalho (clientes) e os servidores. Entre os possíveis projetos e estratégicas topológicas utilizadas no cliente / servidor são as seguintes:

- um único cliente, um único servidor
- vários clientes, um único servidor
- vários clientes, vários servidores

Em termos gerais, existem três tipos de sistemas cliente / servidor:

- Duas camadas (do inglês *Two-tier*):
- Três camadas (do inglês *Three-tier*)
- N-camadas (do inglês *N-tier*).

2.3.2 Modelo Arquitetônico: Ponto-a-ponto

Nesta subseção, descrevemos a arquitetura ponto-a-ponto, na qual todos os processos interagem com regras semelhantes.

Um ponto (em inglês "*peer*") é um computador que se comporta como um cliente no modelo cliente/servidor e também contém uma camada adicional de *software* que permite executar funções de servidor. O computador "*peer*" pode responder às solicitações de outros pares. O escopo dos pedidos e respostas, e como eles são executados, são específicos do aplicativo [40].

Nesta arquitetura, os "pontos" trabalham de forma cooperativa para desempenhar atividades computacionais, ou seja, cada nó envia e recebe mensagens explicitamente. Não há processo clientes ou servidores. Cada processo mantém controle de seus recursos e o modo de interação com outros processos. Nesta arquitetura, estudamos uma biblioteca de troca de mensagens, a qual é uma coleção de rotinas que são embutidas em uma aplicação para o envio, a recepção e outras operações de troca de informação.

Normalmente, haverá um pedido de acesso aos recursos que pertencem a outros pares. O pedido pode ser para obter informações sobre os conteúdos e arquivos, ou para um arquivo a ser lido ou copiado, cálculos a serem realizados, ou uma mensagem de arquivo a ser transmitida aos outros.

A arquitetura ponto-a-ponto permite:

- Externalidades importantes: por agregar recursos através de interoperabilidade a baixo custo, o conjunto é ainda maior do que a soma de suas partes.
- Menor custo de propriedade e compartilhamento de custos, por utilizar infra-estrutura existente e por eliminar ou distribuir custos de manutenção;
- Privacidade/anonimato, por incorporar estes requisitos na concepção e algoritmos de sistemas ponto-a-ponto e aplicações, e por permitir que seus pares tenham um maior grau de controle autônomo sobre seus dados e recursos;

2.3.3 Modelo Fundamental: Dirigido a eventos

Nesta subseção falamos de um modelo para tratar eventos entre processos. Neste modelo cada nó reage a mensagens, mudando estado e gerando novas mensagens. Neste modelo é importante, por exemplo, sabermos em que ordem ocorreram os eventos/mensagens. Todo computador tem um relógio que pode ser consultado pelas aplicações. Porém, quando os processos fazem a consulta, sempre existirá um diferença de tempo entre computadores.

O conceito de tempo é fundamental para o nosso modo de pensar. É derivado do conceito mais básico, o qual é a ordem na qual os eventos ocorrem. Afirmamos que algo aconteceu às 15:15hs. se ocorreu após ler o nosso relógio 15:15hs e antes de ler 15:16hs. O conceito de temporal ordenação dos acontecimentos permeia nosso pensamento sobre os sistemas. Por exemplo, em um sistema de reserva de linha aérea que especifique que um pedido de reserva deverá ser concedida se foi feita antes do vôo estar cheio. No entanto, vemos que este conceito deve ser cuidadosamente reexaminados quando consideramos eventos em um sistema distribuído [41].

Em um sistema distribuído, às vezes é impossível dizer qual de dois eventos ocorreu em primeiro lugar. A relação "Aconteceu antes" é, portanto, apenas uma ordenação parcial dos eventos no sistema. Descobrimos que os problemas frequentemente surgem porque as pessoas não estão plenamente conscientes desse fato e suas implicações [41].

Ordenação Parcial

A maioria das pessoas provavelmente diria que um evento a aconteceu antes de um evento b se a aconteceu em tempo anterior do que b. Poderia se justificar esta definição em termos das teorias físicas de tempo. Entretanto, se um sistema é para atender a uma especificação corretamente, então esta especificação deve ser dada em termos de eventos observáveis dentro do sistema. Se a especificação é, em termos de tempo físico, então, o sistema deve conter relógios reais. Mesmo que um sistema contenha relógios reais, há ainda o problema que tais relógios não são perfeitamente precisos e não mantém precisamente o tempo físico. Portanto, define-se a relação "aconteceu antes de" sem usar relógios físicos.

Para definir um sistema usando a modelagem de eventos com precisão. Supomos que o sistema é composto de um conjunto de os processos. Cada processo é composto por uma sequência de eventos. Dependendo da aplicação, a execução de um subprograma em um computador poderia ser um evento, ou o execução de uma instrução única máquina pode ser um evento. Assumimos que os eventos de um processo formam uma sequência, onde a ocorre antes b nesta sequência se a acontece antes de b. Em outras palavras, um único processo é definido para ser um conjunto de eventos com um ordenação a priori total. Este parece ser o que é geralmente entendido por um processo. Poderia ser trivial para estender a definição para permitir que um processo se dividir em subprocessos distintos.

Assumimos que o envio e o recebimento de uma mensagem é um evento em um processo. Podemos então definir a relação "aconteceu antes", denotada por $\rightarrow$ como se segue:

Relação $\rightarrow$: a relação $\rightarrow$ sobre o conjunto de eventos de um sistema é a menor relação satisfazendo as três condições: (1) Se a e b são eventos no mesmo processo, e a vem antes de b, então $a \rightarrow b$(2) Se a é o envio de uma mensagem por um processo e b é a recepção da mesma mensagem por outro processo, em seguida, então $a \rightarrow b$. (3) Se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$. Dois eventos distintos a e b são ditos ser concorrentes $a \nrightarrow b$ e $b \nrightarrow a$.

Nós assumimos que $a \nrightarrow a$ para qualquer evento a. Isto implica que $\rightarrow$ é uma ordenação parcial não-reflexiva no conjunto de todos os eventos do sistema.

Relógio lógico

Introduzindo relógios ao sistema, começamos com um ponto de vista abstrato em que o relógio é apenas um maneira de atribuir um número a um evento, onde o número é pensado como o momento em que ocorreu o evento. Mais precisamente, definimos um relógio C_i para cada processo P_i ser uma função que atribui um número $C_i(a)$ para qualquer evento a neste processo. O sistema de relógios completo é representada pela função C que atribui a qualquer evento b o número $C(b)$, onde $C(b) = C/(b)$ se b é um evento no processo P_j. Até agora, não fazemos nenhuma hipótese sobre a relação dos números $C_i(a)$ com o tempo físico, então, podemos pensar que os relógios C_i mais como

relógios lógicos do que relógios físicos. Eles podem ser executados por contadores com nenhum mecanismo de tempo real.

A definição acima deve ser baseada na ordem em que os eventos ocorrem. A condição razoável mais forte é que se um evento a ocorre antes de outro evento b, então a deverá acontecer mais cedo do que b. Mais formalmente, esta condição segue abaixo:

Condições do relógio Para quaisquer eventos a e b: se $a \nrightarrow b$ então $C(a) < C(b)$.

Suponha que os processos são algoritmos, e os eventos representam determinadas ações durante a sua execução. Mostramos como introduzir relógios para os processos que satisfazem a condição do relógio. O relógio do processo P_i é representado por um registro C_i, de modo que $C_i(a)$ é o valor contido por C_i durante o evento a. O valor de C_i mudará entre os eventos, desta forma mudar C_i não se constitui um evento sozinho.

Para garantir que o sistema de relógios satisfaz a condição de relógio, garantimos que ele satisfaz as condições $C1$ e $C2$. Condição $C1$ é simples: os processos só precisa obedecer a regra de implementação a seguir:

R1. Cada processo P_i incrementa C_i entre quaisquer dois eventos sucessivos.

Para encontrar a condição $C2$, é necessário que cada mensagem m contenha um *timestamp* T_m que iguala o tempo no qual a mensagem foi enviada. Sobre receber uma mensagem com um *timestamp* T_m, um processo deve avançar seu relógio ser posterior a T_m. Mais precisamente, temos a seguinte regra:

R2. (a) Se um evento a é o envio de uma mensagem m para o processo P_i, então a mensagem m contém um *timestamp* $T_m = Ci(a)$. (b) Sobre receber uma mensagem m, o processo P_i atribui C_i maior ou igual ao seu valor presente e maior do que T_m.

Em R2.(b) consideramos o evento que representa o recibo de mensagem m ocorre depois de atribuir de C_j. Obviamente, $R2$ garante que C2 é satisfeita. Portanto, as regras de implementação mais simples Rl and $R2$ implica que a condição de relógio é satisfeita, portanto elas garante um sistema correto de lógicos.

Ordenação total dos eventos

Pode-se usar um sistema de relógios que satisfazendo a condição de relógio aplicar uma ordenação total no conjunto de todos os eventos do sistema. Simplesmente ordenando os eventos pelos tempos em que eles ocorrem. Para quebrar os vínculos, usamos qualquer ordenação total arbitrária dos processos. Mais precisamente, definimos uma relação $\Rightarrow$ como segue: se a é um evento em processo de P_i e b é um evento em processo P_j, então $a \Rightarrow b$ se e somente se (i) $C_i(a) < C_j(b)$ ou (ii) $C_i(a) = C_j(b)$ e $P_i < P_y$. É fácil ver que isto define uma ordem total, e que a condição d relógio implica que se $a \rightarrow b$ então $a \Rightarrow b$. Em outras palavras, a relação $\Rightarrow$ é uma forma de completar o "aconteceu antes"ordenação parcial para uma ordenação total.

A ordenação $\Rightarrow$ depende do sistema de relógios C_i. e não é exclusivo. Diferentes opções de relógios que satisfazem a condição de relógio produzem diferentes relações $\Rightarrow$. Dado qualquer relação de ordenação total $\Rightarrow$ que se estende $\rightarrow$, existe um sistema de relógios

que satisfaz a condição do relógio que dá a relação que isso. É apenas uma ordenação parcial $\rightarrow$, que é unicamente determinada pelo sistema de eventos.

Ser capaz de ordenar totalmente os eventos podem ser muito útil na implementação de um sistema distribuído. Na verdade, a motivo para aplicação de um sistema correto de relógios lógicos é a obtenção de uma ordem total.

2.4 Programação Paralela e Distribuída

Esta seção explica algumas linguagens e conceitos necessários na programação paralela e distribuída.

2.4.1 Tarefa e Processo

Conforme dito anteriormente, um *processo* é uma entidade abstrata que executa tarefas. Neste contexto, um programa paralelo é composto de processos de cooperação múltipla. Processos, mesmo que estejam em uma mesma UP não compartilham espaço de endereçamento. Cada um dos processos de cooperação executa um subconjunto de tarefas no programa [37, pag.83].

Uma *tarefa* [37, pag.82] é a menor unidade que um programa paralelo pode explorar; uma tarefa individual é executada em exatamente um processo, e a concorrência em um mesmo processo com memória compartilhada é explorada através de tarefas.

2.4.2 Problema $M \times N$

O problema $M \times N$ [42] é o problema de escalonar N tarefas em M máquinas. O problema de escalonamento é determinar a ordem de executar as tarefas em cada máquina. O objetivo é minimizar o tempo decorrido para completar todas as tarefas.

As seguintes hipóteses são assumidas:

1. Existem N tarefas que requerem processamento em M máquinas. ($Tarefaj = 1, 2, ..., N$; Máquina $m = A, B, ..., M$).

2. O processamento da tarefa j na máquina m é chamado de uma *operação* e é designado m_j.

3. Toda tarefa requer processamento em toda máquina e nenhuma tarefa é processada mais de uma vez por qualquer máquina. Existirão $(N \times M)$ operações em um problema $N \times M$. Se uma determinada tarefa não requer processamento em uma máquina particular a operação correspondente terão um tempo de processamento zero.

4. Uma máquina pode processar somente uma tarefa em qualquer tempo dado. Além disso, uma tarefa não pode ser processada por mais de uma máquina em qualquer tempo dado.

5. Há somente uma máquina de cada tipo.

6. Uma operação uma vez iniciada deve ser completada sem interrupção.

7. Todas as tarefas são consideradas de igual importância. Portanto, não há datas de vencimentos, prioridades, ou ordem de avanço.

8. Tempos de processamento são assumidos serem conhecidos sem erro.

9. Tempos de processamento são independentes de outros tempos de processamento e também, são independentes da ordem na qual são processados.

10. Tempo de iniciação e o tempo requerido para transportar tarefas entre máquinas é zero.

Este problema pode ser visto como um caso especial de um problema de escalonamento generalizado [42].

2.4.3 Comunicação em Sistemas Distribuídos e Paralelos e Sincronismo

Nesta subseção compreendemos a relação entre comunicação em sistemas distribuídos e paralelos e sincronismo.

Redes de computadores e computação distribuída tem, ultimamente, atraído muita atenção [43]. Isso é devido, em parte, pela disponibilidade de processadores de baixo custo que fazem a construção viável de tais redes. Frequentemente, uma tarefa particular pode ser decomposta em processos disjuntos que se comunicam (por exemplo, quando não há memória compartilhada) em diferentes maneiras. A decomposição escolhida dita como os objetivos são realizados. Por exemplo, processos acoplados fortemente por usar protocolos de comunicações síncronas pode decrescer uma saída global do sistema porque seu paralelismo potencial é reduzido. Por esta razão, o uso de protocolos de comunicação assíncrona se mostra mais adequado. Tais protocolos permitem um processo continuar executando enquanto uma mensagem está sendo entregue. Isto tende a aumentar o desempenho dos processos de uma rede para outra rede de comunicação. Infelizmente, uma consequência desta abordagem é que um nenhum único processo pode ter conhecimento completo do estado global do sistema, porque qualquer informação que um processo obtém de mensagens reflete um estado anterior dos processos de envio, não o estado corrente. Isto torna o projeto e a análise de programas distribuídos muito difícil. Um dos aspectos de construção de programas distribuídos é a sincronização.

Por outro lado, para compreender as comunicações interprocessos é essencial para entender a sua influência no desempenho de aplicações paralelas. Segundo [44, 45], muitas interações em programas paralelos ocorrem em padrões bem definidos envolvendo dois ou mais processos.

A principal consideração é que o desempenho das primitivas de comunicação de computadores paralelos é crítica para o desempenho global do sistema, ou seja, os padrões de comunicação afetam a computação pela taxa de comunicação.

Comunicações Síncronas e Assíncronas

Comunicações por sua natureza requerem, no mínimo, dois processos, um para enviar uma mensagem e um processo para receber esta mensagem, por isso, tem-se que especificar o grau de cooperação necessária entre os dois processos.

Se o remetente está pronto para enviar, mas o receptor não está pronto para receber, o remetente é bloqueado e, da mesma forma, se o receptor está pronto para receber antes do remetente está pronto para enviar, o receptor é bloqueado. A ação de se comunicar sincroniza as sequências de execução dos dois processos [35]. Alternativamente, o remetente pode ser autorizado a enviar uma mensagem e continuar sem bloqueio.

As comunicações são assíncronas quando não existe uma dependência temporal entre as sequências de execução dos dois processos. O receptor poderia estar executando qualquer declaração, quando uma mensagem é enviada, e só mais tarde verificar o canal de comunicações por mensagens.

A decisão sobre comunicação assíncrona e síncrona é baseada na capacidade do canal de comunicações para armazenar mensagens, e na necessidade de sincronização. Na comunicação assíncrona, o remetente pode enviar várias mensagens sem o receptor removê-las do canal, de modo que o canal deve ser capaz de armazenar um número muito grande de mensagens. Dado que qualquer *buffer* é finito, eventualmente, o remetente será bloqueado, mensagens serão descartadas ou o remetente recebe um erro e tenta mais tarde.

Comunicações síncronas e assíncronas são similares a chamadas telefônicas e mensagens de email, respectivamente. Uma chamada telefônica sincroniza as atividades do chamador e da pessoa que responde. Se as pessoas não podem sincronizar, o resultado da chamadas será ocupado e ou sem sinal. Por outro lado, qualquer número de mensagens podem ser enviado por e-mail, e os receptores podem optar por verificar os e-mails recebidos a qualquer momento.

Comunicação assíncrona exige um *buffer* de mensagens enviadas e não recebidas, e este *buffer* deve ser guardado em algum lugar.

2.4.4 MPI

Nesta subseção, explicamos a abordagem de troca de mensagens e estudamos a principal biblioteca de troca de mensagens, chamada *Message Passing Interface* (MPI) [46]. O MPI [46] é uma coleção de rotinas que são embutidas em uma aplicação para o envio, a recepção, rotinas de comunicação ponto-a-ponto e operações coletivas para movimentação de dados, computação global e sincronização.

O objetivo do MPI [46] é fornecer uma biblioteca padrão de rotinas para escrever programas portáveis e eficientes de trocas de mensagens. O MPI não é uma linguagem,

mas uma especificação de uma biblioteca de rotinas que podem ser chamadas a partir de programas.

Talvez a melhor maneira de introduzir os conceitos de MPI que poderia inicialmente parecer estranho é mostrar como eles surgiram como extensões necessárias de conceitos bastante familiar. Consideramos o que é talvez a operação mais elementar em uma biblioteca de passagem de mensagens, enviar um bloqueio. Na maioria dos sistemas atuais de transmissão de mensagens, ele é muito parecido com este [47]:

`send (destino, tipo, endereço, comprimento)`, no qual

- Destino é o identificador do processo do processo para que esta mensagem é enviada (geralmente um inteiro).
- Tipo é um inteiro arbitrário restringir o recebimento da mensagem.
- Endereço é um local de memória, significando o início do buffer contendo os dados a serem enviados.
- Comprimento é o comprimento em bytes da mensagem.

Um requisito importante em todos os sistemas de trocas de mensagem é garantir um espaço de comunicação segura no qual as mensagens não-relacionadas são separadas uma da outra. Por exemplo, mensagens de biblioteca podem ser enviadas e recebidas, sem interferência de outras mensagens geradas no sistema.

No MPI, onde não existe uma máquina virtual, usar apenas uma *tag* de mensagem não é suficiente para distinguir com segurança a biblioteca de mensagens de usuário mensagens. O conceito de comunicador é introduzido no MPI para atingir este requisito de comunicação segura.

Um grupo em MPI é um objeto que pode ser acessado através de uma *handle*. Grupos de tarefas fornecem contextos através dos quais as operações MPI podem ser restritas somente aos membros de um grupo em particular. Os membros de um grupo são atribuídos a identificadores únicos dentro do grupo chamado *ranks*. Um grupo é um conjunto ordenado de *ranks* que sejam contíguos e começam do zero.

Para se comunicarem, os processos MPI podem enviar mensagens de maneira síncrona ou assíncrona.

A norma MPI tem evoluído com o trabalho de MPI-2, a qual o MPI foi estendido para adicionar mais funcionalidades, incluindo: processos dinâmicos, apoio cliente-servidor, comunicação unidirecional, *Input/Output* (I/O) paralelo, e funções de comunicação coletiva não-bloqueantes.

2.4.5 Objetos Distribuídos

Nesta subseção, explicamos a programação de arquiteturas de objetos distribuídos, onde não há distinção entre clientes e servidores. Cada entidade distribuível [48] é um objeto que fornece serviços para outros objetos e recebe serviços de outros objetos. Os objetos

se comunicam através de um sistema de middleware chamado requisitor de objetos. Contudo, arquiteturas de objetos distribuídos são mais complexas para projetar que sistemas cliente-servidor.

Um objeto tem um conjunto de atributos que juntos representam o estado do objeto. Atributos não necessitam ser acessíveis a partir de outros objetos; eles podem ser privados ou ocultos [48].

Os dados armazenados em atributos ocultos podem somente ser acessados e modificados por operações: outros objetos não podem se tornar dependentes de estruturas de dados internas de um objeto e podem ser alterados sem afetar outros objetos.

Esta característica é particularmente importante se os objetos são concebidos e mantidos em uma configuração distribuída, possivelmente em diferentes organizações. Objetos podem exportar um conjunto de operações que revelam o estado dos atributos ou modificam seus valores. Outros objetos podem requerer execução de um pedido de uma operação exportada.

Segundo [49], um modelo sofisticado para o compartilhamento de recursos é baseado no conceito de objetos distribuídos. Um objeto distribuído representa e encapsula um recurso que ele usa para fornecer serviços a outros objetos.

O paradigma orientado a objeto é muito adequado para um modelo de sistema distribuído. Os serviços podem ser vistos como operações que um objeto exporta. Como esses objetos são implementados não é importante e o tipo de um objeto é, portanto, definido por suas *interfaces*.

2.4.6 *Socket*

Nesta subseção, explicamos uma abstração, chamada *Socket*, que representa uma porta de comunicação bidirecional associada a um processo e damos um exemplo de seu funcionamento.

Um *socket* [50] é um ponto de comunicação. Dois processos podem se comunicar através da criação de *sockets* e o envio de mensagens entre eles. Há uma variedade de diferentes tipos de *sockets*, diferindo na forma como o espaço de endereçamento dos *sockets* está definido e do tipo de comunicação que é permitida entre *sockets*. Um tipo de *socket* é unicamente determinado por um tipo de tripla <domínio, tipo, protocolo>. Para que um *socket* remoto seja alcançado, deve ser possível atribuir um nome a ele. A forma que este nome é determinado pelo domínio da comunicação ou do endereço da família à qual o socket pertence. Existe também um tipo abstrato ou o estilo de comunicação associados a cada *socket*.

Finalmente, há um protocolo específico que é utilizado com o socket. Um *socket* pode ser criado com a chamada de sistema *socket* especificando o domínio, tipo de *socket* e protocolo. O domínio de comunicação ou família de endereço para o qual o *socket* pertence especifica um determinado formato de endereço. Todas as operações posteriores em um *socket* interpretará um endereço fornecido de acordo com o formato específico. Um protocolo é um conjunto de convenções de comunicação para controlar a troca de

informação entre duas partes [50].

Um exemplo de definição de *socket* abaixo:

```
socket_descriptor = socket(domain, type, protocol)
int socket_descriptor, domain, type, protocol;
```

Essa chamada retorna um pequeno inteiro positivo chamado de descritor de *socket* que pode ser usado com um parâmetro para referenciar o *socket* em chamadas de sistema subsequente. Os descritores de *sockets* são similares aos descritores de arquivo retornados pela chamada aberta de sistema. Cada chamada aberta ou de *socket* retornará o menor inteiro não usado portanto um dado número denota ou um arquivo aberto, ou nenhum dos dois (mas nunca ambos). Descritores de arquivo ou de *sockets* podem ser usados em muitas chamadas de sistema. Por exemplo, chamada de sistema de fechamento é usada para destruir *sockets*.

Existem basicamente cinco tipos de *socket* disponíveis:

- *Socket Stream*: provê duas vias, em sequência, confiável, e o fluxo de dados não-duplicados, sem limites de registro. Um *stream* funciona muito como uma conversa telefônica. O tipo do *socket stream*, no domínio da *Internet*, usa o TCP (do inglês *Transmission Control Protocol*).

- *Socket* datagrama: suporta um fluxo bidirecional de mensagens. Em *socket* datagrama pode receber mensagens em uma ordem diferente da sequência em que as mensagens foram enviadas. Limites de registros são preservadas. O *socket* datagrama, no domínio da Internet, usa o protocolo UDP (do inglês *User Datagram Protocol*).

- *Socket* de pacote sequencial - fornece uma conexão de duas vias, em sequência e confiável para datagramas com um comprimento máximo fixo. Nenhum protocolo para este tipo foi implementado para toda a família de protocolos.

- *Socket raw*: fornece acesso a protocolos de comunicação subjacentes.

2.4.7 *Clusters*

Um *cluster* [51, 52] é um tipo de processamento paralelo ou distribuído de sistema, que consiste de uma coleção de computadores interconectados autônomos trabalhando juntos como um recurso único e integrado. A arquitetura típica de um cluster é mostrado na Figura 2.4.

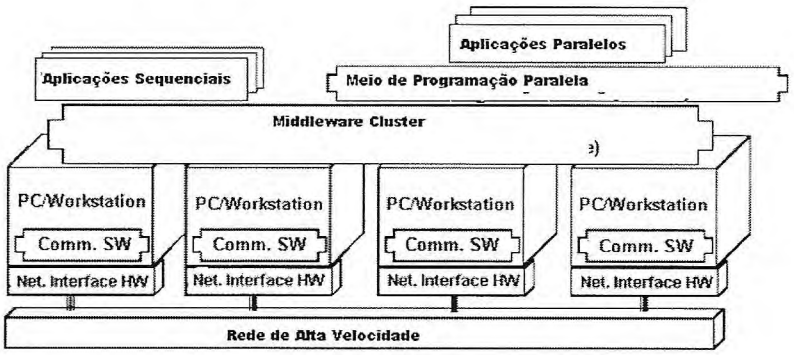


Figura 2.4: Arquitetura de *Cluster*.

Em um *cluster* [53], todos os subsistemas que o compõem são supervisionadas dentro de um único domínio administrativo, geralmente residem em uma única sala e gerenciado como um único sistema de computador. Os nós de computador constituintes são capazes de operar independentemente tarefas completas, e são de um tipo normalmente utilizado para cargas de trabalho individual autônomo. Os nós podem incorporar um microprocessador simples ou múltiplos microprocessadores em um multiprocessador simétrico (SMP) de configuração. A rede de um *cluster* é dedicada à integração dos nós e é separado do ambiente do *cluster* (mundo externo). Um *cluster* pode ser empregado em muitos modos, incluindo mas não limitados a: alta capacidade e desempenho sustentado em um problema único, de alta capacidade, ou transferência em um trabalho ou processo de trabalho, alta disponibilidade através de redundância de nós, ou de banda larga através da multiplicidade de discos e de acesso ao disco ou canais de I/O.

2.4.8 RPC

A ideia de chamadas de procedimento remoto [54] (denominado em inglês *Remote Procedure Call* (RPC)) é bastante simples. RPC é baseado na observação de que as chamadas de procedimento são bem conhecidas e bem compreendidas mecanismo de transferência de controle e de dados dentro de um programa em execução em um único computador. Assim, propõe-se que este mesmo mecanismo seja ampliado para fornecer a transferência de controle e de dados através de uma rede de comunicação. Quando um procedimento remoto é invocado, o ambiente de chamada for suspenso, os parâmetros são passados através da rede para o ambiente onde o processo está executando (o que nós referimos como o receptor), e do processo desejado é executado lá. Quando o procedimento termina e produz os seus resultados, os resultados são passados apoio ao ambiente de chamada quando a execução continua como se estivesse voltando de uma chamada de uma única máquina simples. Embora o ambiente de chamada é suspenso, outros processos em que a máquina pode (eventualmente) ainda executar (dependendo dos detalhes do paralelismo do ambiente e da aplicação) [54].

Há muitos aspectos atrativos para essa ideia. Um deles é a semântica clara e simples: estes devem fazê-lo mais fácil de construir cálculos distribuídos, e para obtê-los corretamente. Outro atrativo é a eficiência: as chamadas de procedimento parecem simples o suficiente para a comunicação a ser bastante rápida. O terceira atrativo é a generalidade: em computações de máquina única, os procedimentos são frequentemente o mecanismo mais importante para a comunicação entre as partes do algoritmo [54].

RMI

Invocação do método remoto (ou RMI do inglês *Remote Method Invocation* (RMI)) é a invocação de um método em um objeto remoto. Um objeto remoto é um objeto cujos métodos podem ser chamados a distância através de uma *interface* remota. Um método remoto é um método definido em uma interface remota, é invocado através dessa interface. Qualquer objeto, mesmo um objeto local, pode ser pensado como um servidor e seus usuários como seus clientes. Um objeto local é essencialmente um servidor local, um objeto remoto é um servidor remoto.

Para ser acessível através de RMI, um objeto remoto deve:

- Implementar uma *interface* remota;
- Serem exportados para o sistema RMI.

Um objeto remoto é acessado através de um *stub* remoto. Um *stub* é um objeto remoto que implementa a mesma interface remota(s) como o objeto remoto que se refere. Sua classe é gerada a partir do objeto remoto correspondente pelo sistema RMI em tempo de compilação.

Um *stub* remoto só pode ser obtido como resultado de outra invocação de método remota. O cliente usa o *stub* remoto como uma instância da *interface* implementada pelo objeto remoto. O *stub* remoto não é o próprio objeto remoto, nem é uma instância da classe remota. Um stub *remoto* é realmente um *proxy* para o objeto remoto.

Um *stub* remoto também possui todas as propriedades usuais de um objeto Java: estado, métodos e referências externas, embora apenas os métodos (nem Estado ou referências externas) são de interesse para o cliente RMI.

Existem diferenças fundamentais entre a programação em uma única máquina e a programação distribuída. Chamar um método remoto via chamada de método remoto (RMI), não é exatamente o mesmo que chamar um método em um objeto local, mesmo que seja usada a mesma sintaxe. Sintaxe é o conjunto de regras que regem o arranjo das palavras.

Conforme dito, a sintaxe de uma invocação de método remoto é idêntico ao da sintaxe de uma chamada de método local, como vemos no exemplo abaixo:

Tabela 2.1: Invocação de método remoto.

```
try {
        result = remoteInterface.invoke(arguments);
} catch ( RemoteException ex)
}
```

Um método em um objeto local pode modificar os objetos passados como parâmetros, ou ele pode modificar algum outro objeto para o qual tanto ele quanto o invocador tem acesso (por exemplo, dados estáticos). Em ambos os casos, as alterações são visíveis para o invocador.

Um método remoto não pode usar esta técnica para se comunicar com o seu invocador. Um objeto remoto pode ainda alterar os seus parâmetros ou outros objetos, mas, essas modificações não são visíveis para o invocador. Um objeto remoto pode se comunicar com o seu invocador através de valores de retorno ou exceções. Isto tem consequências importantes para a concepção de métodos remotos.

Quando a passagem de parâmetros é por valor, os dados são copiados. O emissor e o receptor têm diferentes instâncias de dados. Se o método chamado modifica um parâmetro, o invocador não perceberá nenhuma mudança.

Quando a passagem de parâmetros é por referência, uma referência para o valor original é passada. O emissor e o receptor ambas se referem aos mesmos dados. Se o método chamado modifica um parâmetro, a mudança no seu valor será percebido pelo invocador.

Ao invocar métodos remotos, o RMI passa os tipos primitivos e os tipos objetos por valor, exceto para objetos remotos exportados, que são passados por referência. "Passar por valor" para os valores de tipo de objeto é implementado como cópia completa. "Passar por referência" para exportar objetos remotos é implementado por referências remotas - *stubs* remotos.

2.4.9 CORBA

CORBA (do inglês *Common Object Request Broker Architecture* (CORBA)) [55], é uma norma aprovada pelo OMG (do inglês *Object Management Group* (OMG)) para permitir a interoperabilidade entre aplicações em ambientes heterogêneos distribuídos, sem verificar onde as aplicações estão localizados. Em outras palavras, CORBA é uma infra-estrutura de computação distribuída aberto objeto. Seu objetivo é automatizar muitas tarefas comuns de programação de rede, como registro de objetos, localização e ativação. Esta automação de que normalmente são funções da rede é feito com um *software* intermediário chamado de ORB (do inglês *Object Request Broker*). Corba se situa na máquina entre camada de dados e a camada de aplicação (ou seja, um nível inferior ao nível da aplicação de camada 7 do modelo OSI) e trata, de forma transparente, as mensagens de solicitação de clientes (que pode ser utilizador ou servidor de objetos) e servidores (ou seja, as

implementações que prestam serviços específicos).

O paradigma CORBA é baseado em uma combinação de duas metodologias existentes. A primeira, a computação distribuída cliente-servidor, é baseado em parte em sistemas de passagem de mensagens mais comumente encontrados em ambientes baseados em *UNIX*. A segunda metodologia é a programação orientada a objeto. Uma ORB desempenha o papel de uma interface de programação de aplicação de chamada de procedimento remoto (RPC) orientado a objeto. Presta serviços comuns, tais como passagem de mensagens básicas e uma comunicação tipo "RPC" entre clientes e servidores, serviços de diretórios, localização e transparência de máquina. CORBA é baseado em um modelo de comunicação ponto-a-ponto refTari.Bukres.Fundamentals.CORBA.Perspective.2001.

Um objeto CORBA tem uma interface e uma implementação. A interface não é vinculada a uma linguagem de programação específica de implementação, mas é escrito em um linguagem especial chamada IDL (do inglês *Interface Descriptor Language* (IDL)), que, por sua vez, se traduz em diferentes construções nas línguas através de mapeamentos de aplicação diferentes línguas. Isto torna possível chamar uma implementação de objeto escrito em uma determinada língua (por exemplo, Cobol) de um programa cliente escrito em outro idioma (por exemplo, Smalltalk).

Em resumo, CORBA fornece várias vantagens sobre os actuais sistemas distribuídos. Do ponto de vista de desenvolvimento de software, os desenvolvedores podem usar CORBA para distribuir aplicações em redes cliente-servidor. Ao invés de ter centenas de milhares de linhas de código em execução em computadores com terminais burros. menor, as aplicações mais robustas que se comunicam entre servidores de arquivos e estações de trabalho são agora necessárias. CORBA mantém a distribuição de aplicações simples, uma arquitetura de *plug-and-play* é usado para distribuir as aplicações cliente-servidor. O programador então pode escrever aplicativos que funcionam de forma independente em várias plataformas e redes [55].

Outras soluções para o problema de integração são distribuídos DCOM da Microsoft (*Distributed Component Object Model*) e DCE (*Distributed Computing Environment*).

2.4.10 Grades (do inglês *Grids*)

A Grade [56, pág. 09] é a computação e a infra-estrutura de gerenciamento de dados que fornece o apoio eletrônico para uma comunidade global em negócios, governo, pesquisa, ciência e entretenimento. O termo "grade" [57, pág. 66] (do inglês *grid*) foi cunhado em meados dos anos 1990 para designar a (então) infraestrutura proposta de computação distribuída para a ciência avançada e engenharia. Muito progresso já foi feito na construção dessa infra-estrutura e em sua extensão e aplicação de problemas de computação comercial. Enquanto isso o termo "*Grid*" também tem sido confundido por ocasião de englobar tudo. desde redes avançadas e *clusters* de computação para inteligência artificial. tem também surgido uma boa compreensão dos problemas que as tecnologias de grade visam, e pelo menos um primeiro conjunto de aplicações para as quais estejam aptos.

Em outras palavras, podemos definir uma grade como um sistema que coordena a distribuição de recursos usando protocolos de propósito geral, abertos e padrão e *interfaces* para entregar qualidades de serviço não triviais. Examinando os elementos-chave desta definição, temos:

- Coordena recursos distribuídos: uma grade integra e coordena recurso e usuários que se situam em diferentes domínios de controle, por exemplo, diferentes unidades administrativas da mesma empresa e / ou diferentes empresas, e aborda as questões de segurança, de política, e assim por diante que surgem nessas configurações. Caso contrário, estamos lidando com um sistema de gestão local.

- Usando protocolos de propósito geral, abertos e padrão e *interfaces*. A grade é construída a partir de múltiplos protocolos e *interfaces* que resolvam esses fundamental questões como autenticação, autorização e descoberta de recursos.

- Entrega de qualidade de serviço não triviais. A grade permite aos seus constituintes recursos a serem utilizados de forma coordenada para oferecer várias qualidades de serviços relacionados, por exemplo, tempo de resposta, disponibilidade.

Grades, ilustradas na Figura 2.5, a integram redes, comunicação, computação e informação para fornecer uma plataforma virtual para a computação e gestão de dados da mesma forma que a *Internet* se integra recursos para formar uma plataforma virtual de informações. Grade está transformando a ciência, negócios, saúde e sociedade.

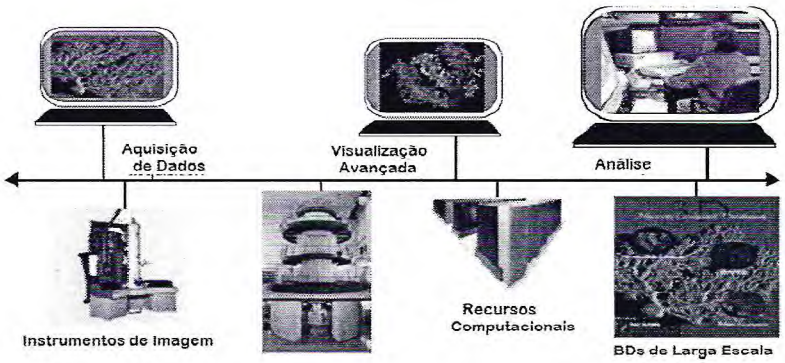


Figura 2.5: Recursos de grade vinculados a aplicação de Teleciência do neurocientista Mark Ellisman (http://www.npaci.edu/Alpha/telescience.html).

Grades em grande escala são intrinsecamente distribuídas, heterogêneas e dinâmicas. Elas prometem efetivamente ciclos e armazenamentos infinitos, bem como o acesso a instrumentos, dispositivos de visualização e assim por diante sem considerar as localizações geográficas.

A realidade é que, para alcançar esta promessa, sistemas complexos de *software* e serviços devem ser desenvolvidos, para permitir o acesso de uma forma amigável, permitir

que os recursos sejam utilizados de forma eficiente, e aplicar as políticas que permitam que as comunidades de usuários coordenar os recursos de uma forma estável e que promova o desempenho. Se os usuários acessam a Grade para usar um recurso (um único computador, arquivos de dados, etc), ou para usar vários recursos em agregados como um "computador virtual coordenado", a Grade permite aos usuários uma interação com os recursos de maneira uniforme, fornecendo uma plataforma abrangente para computação global e gerenciamento de dados.

O elemento principal de qualquer grade é a sua rede - redes que interligam recursos geograficamente distribuídos e permitem que eles possam ser utilizados em conjunto para apoiar a execução de uma única aplicação [56, pág. 16]. Redes que conectam recursos na Grade, prevalecem mais do que os computadores como seu armazenamento de dados associado [56, pág. 23]. Embora os recursos computacionais possam ser de qualquer nível de poder e capacidade, algumas das Grades mais interessantes para os cientistas envolvem os nós que são eles próprios máquinas de alto desempenho paralelo ou clusters. Tal Grade com 'nós' com alto desempenho fornecem os principais recursos para a simulação, análise, mineração de dados e outras atividades de computação intensiva.

2.5 Medidas de Desempenho (Eficiência e Aceleração)

Esta seção descreve conceitos relacionados ao desempenho, estabelece medidas de desempenho computacional e os pontos importantes associados a essas medidas. Em particular, damos destaque a algumas estratégias de melhoria e medidas de desempenho que são utilizadas para avaliação de sistemas e processadores.

2.5.1 Abordagem de Medição de Desempenho

Esta subseção descreve conceitos de medição associados ao desempenho em processamento paralelo, considerando que neste contexto, explorar o paralelismo é uma abordagem cada vez mais comum para melhorar o desempenho dos sistemas de computadores. Em termos de *hardware*, isso normalmente significa fornecer múltiplos processadores simultaneamente ativos. Em termos de *software*, isso normalmente significa a estruturação de um programa como um conjunto de subtarefas amplamente independentes [58].

No mundo sequencial, o desempenho de um sistema geralmente pode ser adequadamente caracterizado em termos da taxa de instrução da processador único e a exigência de tempo de execução do *software* em um processador de taxa unitária (o que chamamos de sua demanda de serviço). No mundo paralelo, as medidas são consideravelmente mais complexas. No domínio do *hardware*, é preciso se preocupar não apenas com a taxa de instrução de um processador, mas também com fatores como o número de processadores. No domínio de *software*, devemos nos preocupar não só com as demandas de serviços, mas também com fatores como a estrutura do *software*.

Na avaliação de um sistema paralelo, duas medidas de desempenho de interesse particular são speedup e eficiência. Aceleração é definida para cada n número de processadores como a razão entre o tempo decorrido durante a execução de um programa em um único processador (O tempo de execução em um único processador) para o tempo de execução quando n processadores estiverem disponíveis. Como segue [58]:

$$S(n) = \frac{T_t}{T_n}$$

A eficiência é definida como a utilização média dos n processadores alocados. Ignorando "entrada e saída", a eficiência de um único sistema de processador é 1. Aceleração neste caso é 1. Em geral, a relação entre a eficiência e a aceleração é dada por

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

Se a eficiência permanece 1 (um) quando os processadores são adicionados, temos uma aceleração linear. (Tecnicamente, a aceleração para ser linear exige só que $S(n) = \alpha w$ para alguma constante α, $0 < \alpha \leq 1$, mas usamos a definição mais rigorosa $\alpha = 1$.) Este é o caso ideal, como melhorias na aceleração pode ser obtida sem custo de eficiência. Aceleração linear não é viável, em geral, por causa da contenção de recursos compartilhados, o tempo necessário para a comunicação entre os processadores e entre processos, e a inabilidade da estrutura do *software* para que um número arbitrário de processadores pode ser mantido proveitosamente ocupado. Minsky e Papert observou evidência de que a "aceleração típica" tem a forma $S(n) = \log n$ [59]: outros estudos têm fornecido evidências de que acelerações muito maiores do que as acelerações típicas podem ser alcançadas [60].

Capítulo 3

Modelos de Componentes e Plataformas para Computação de Alto Desempenho

Este capítulo apresenta um painel de incorporação de conceitos de Engenharia de *Software* à produção de software para Computação de Alto Desempenho (CAD), atendendo a uma demanda crescente das aplicações por reusabilidade em um ambiente heterogêneo.

Na Seção 3.1 descrevemos o conceito de componente adotado neste trabalho. Na Seção 3.2, mostramos o modelo de componentes CCA (do inglês *Common Component Architecture* (CCA)). Após isso, na Seção 3.3 apresentamos algumas das principais plataformas CCA. Na Seção 3.4, mostramos o modelo de componentes *Hash* (#). Por fim, descrevemos as principais características do modelo de componentes Fractal na Seção 3.5.

3.1 O Conceito de Componente

O conceito de *reuso* surgiu em Engenharia de *Software* motivado pelo desafio de se obter rapidamente um *software* complexo a partir da integração de partes independentes, frequentemente menores e menos complexas [61]. Nesse contexto, o conceito de componente foi imaginado para identificar essas partes menores de maneira um pouco mais precisa, dando a elas características necessárias para tornar a integração possível. Assim, dizemos que [62]:

> ***Componente*** é uma unidade de código definida e implementada de forma autônoma, que pode ser acoplada a outros componentes através de um ou mais pontos de acesso às suas funcionalidades específicas.

Essa simples ideia é uma maneira para:

- Construir aplicações por composição, possivelmente em tempo de execução, usando unidades bem testadas e bem "comportadas".
- Separar interesses de computação e comunicação.

- Manipular atividades dinâmicas em execução e distribuídas globalmente.
- Incorporar códigos otimizados, o qual ainda é um desafio.

Ao lado disso, a programação baseada em componentes oferece muitas vantagens sobre a programação orientada a objetos e a programação estruturada, entre elas podemos citar:

- Componentes fornecem uma especificação precisa das entradas necessárias de outros componentes ou elementos do sistema;
- Componentes bem projetados têm o potencial de encapsular bem o paralelismo;
- Permite a integração dos componentes implementados em diferentes linguagens em aplicações multi-componentes.
- Componentes podem ter estado;
- Componentes podem ser substituídos, adicionados ou suprimidos a partir de uma aplicação em tempo de execução;
- Um componente pode ser facilmente deslocado para um local remoto, sem a necessidade de recompilação de outras partes de aplicação, em especial outros componentes que interagem com ele diretamente.

Um ponto central na construção de um programa complexo pela integração de componentes é a interação entre esses componentes. A interação entre componentes é geralmente feita através de conectores. Segue abaixo a definição de conector:

Conector é uma unidade que intermedia as interações entre dois ou mais componentes, estabelecendo as regras que governam essas interações e especificando todos os mecanismos empregados [63, 64]. Tipicamente, um conector efetua uma conexão entre pontos de acesso de componentes.

Note que a definição deixa espaço para o uso de conectores que trabalhem em conjunto com a finalidade de combinar os dados ou fluxos de controle em interações complexas entre os componentes, normalmente fornecendo algumas propriedades locais ou globais para o padrão de comunicação (como a sincronização e balanceamento de carga). Para atingir esse objetivo, algum processamento ou armazenamento de informação pode ser exigido, o que motiva a identificação de duas categorias distintas de conectores [65]:

Conector direto fornece uma conexão direta simplesmente através de invocações de métodos entre pontos de acesso sendo conectados.

Conector indireto é a composição de uma ou mais ligações que permitem algumas combinações de fluxos de dados e de controle para prover serviços de interação mais complexos. Em geral, um conector indireto pode usar outros conectores, diretos ou indiretos, ou mesmo componentes na sua implementação.

Historicamente, os conceitos mencionados acima foram designados para refletir a ideia de que os conectores são destinados a encapsular interação ou comunicação, enquanto componentes são destinados para encapsular computação. Este princípio admite duas variações principais[66, 67]:

Conector endógeno conecta componentes distintos ou não. O termo "endógeno" ressalta o fato de que essas interações se dão por meio de invocações de métodos originadas por um dos componentes que estão sendo conectados.

Conector exógeno estabelece uma sequência de interações entre componentes. Essas interações são associadas a invocações de métodos e originadas internamente no conector, justificando o uso do termo "exógeno".

Neste ponto, duas questões devem ser abordadas: a primeira é como projetar e construir componentes; a segunda, como executar um conjunto de componentes adequadamente conectados para atender a um objetivo específico.

Com relação à primeira questão, busca-se um padrão que os componentes devem seguir para que possam ser conectados e, assim, construir uma aplicação composta por componentes que interagem entre si. Com tal padrão bem definido, ao qual chamamos de *modelo de componentes* [68], podemos moldar cada componente, durante a sua construção, de forma que o mesmo disponibilize o que os demais componentes possam necessitar para interagir com ele. Por outro lado, uma resposta à segunda questão nos fornece o mecanismo para usar os componentes já construídos. Trata-se do que se convencionou chamar *plataforma* (do termo em inglês *framework*), que é um sistema que permite a instanciação, o acoplamento e a execução de componentes [13].

Neste contexto, uma arquitetura de componentes é uma especificação de um conjunto de *interfaces* e regras de interação que governam a comunicação entre componentes e outras ferramentas necessárias, tais como repositórios e ferramentas de composição [10]. Uma arquitetura de componentes consiste de duas partes:

- Um modelo de componentes, que implementa um conjunto de comportamentos requeridos;
- Uma plataforma:
  - É um meio de execução;
  - Define um conjunto de regras para a maneira na qual os componentes podem ser instanciados e compostos;
  - Provê um conjunto de serviços básicos usados por componentes para executar e interagir.

Recentemente, com o crescente interesse na construção de aplicações de CAD através da integração de partes de *sofware* construídas independentemente, o conceito de componente passou a fazer parte do vocabulário de CAD. Alguns termos herdados da

Engenharia de *Software* ganharam uma interpretação particular, em virtude da cultura e das especificidades de CAD. Este é o caso, por exemplo, do termo *framework*. Ao longo deste texto, mantemos os significados usualmente adotados em CAD. Ao leitor interessado, sugerimos [69, 70] para os significados em Engenharia de *Software*.

A seguir, passamos a descrever três modelos de componentes utilizados em CAD. Estes são, possivelmente, os mais utilizados atualmente. Damos destaque ao modelo CCA, pois trata-se do modelo sobre o qual se baseia a plataforma *Forró* proposta nesta tese. Para cada um dos modelos considerados, apresentamos um breve apanhado de algumas plataformas existentes.

3.2 O Modelo de Componentes CCA

O CCA é um modelo de componentes para executar aplicações de alto desempenho. O CCA emprega uma filosofia de projeto minimalista para simplificar a tarefa de incorporar *software* existente de alto desempenho [71].

O padrão CCA não especifica uma plataforma única, mas ao invés disso, especifica um conjunto mínimo de serviços necessários para ser compatível com o CCA. O modelo CCA foi projetado para prover um meio rápido e simples para cientistas gerenciarem a complexidade de simulações científicas de larga-escala. O mesmo modelo é usado para a implementação de *plataformas* sequenciais, paralelas e distribuídas.

A ideia central do CCA é construir aplicações pela composição de componentes [72]. O padrão do projeto CCA contém uma composição interativa dos componentes [73]. Na tabela abaixo vemos uma comparação feita pelo grupo do CCA [15]:

Tabela 3.1: Comparação das características dos tipos de códigos: monolítico, baseado em bibliotecas e baseado em componentes.

Características	Código Monolítico	Biblioteca	Componentes
Suporte para *workflows* específicos e fluxos de informação	Alta	Baixa	Baixa
Extensibilidade ao nível de usuário	Baixa	Alta	Alta
Flexibilidade para *workflows* e fluxos de informação	Baixa	Alta	Alta
Facilidade de incorporação de código (reuso de código)	Baixa	Média	Alta (reuso pode reduzir)
Facilidade de experimentação	Baixa	Média	Alta
Amplitude do "ecossistema" corrente para "*plugins*"	Baixa	Alta	Baixa (mas crescente)
Facilidade de simulações acopladas	Baixa	Média	Alta

3.2.1 Elementos Básicos: Portas e Conexões

Uma *porta* de um componente CCA é um ponto de acesso que disponibiliza definições concretas da maneira de interagir desse ou com esse componente. Posto de outra forma, mas ainda no campo abstrato, uma porta é um ponto de acesso com *interface* bem definida, especificando as maneiras possíveis de comunicação entre componentes CCA. Normalmente, um componente CCA tem múltiplas portas, correspondendo a diferentes pontos de acesso. Uma porta CCA é um recurso que pode ser exportado ou importado por um componente. Um componente exporta uma porta sua para fazer com que ela seja conhecida externamente ao componente. Por outro lado, ao importar uma porta de outro componente, um componente passa a ter acesso às funcionalidades especificadas nessa porta importada.

O modelo CCA estabelece dois tipos de porta, denominados *provides* e *uses*. Uma porta *provides* desempenha o papel de ponto de entrada de um componente. Uma porta *uses*, ao contrário, contém as funcionalidades que um componente pode necessitar de outro componente, desempenhando assim os pontos de saída do referido componente. Uma conexão é uma associação de uma porta *provides* e uma porta *uses*, o que só é possível se as duas portas envolvidas na conexão corresponderem a uma mesma interface. Observe que uma conexão conecta um ponto de saída (porta *uses*) a um ponto de entrada (porta *provides*), conforme ilustrado na Figura 3.1. O modelo CCA não impõe o uso de conectores para realização das conexões.

Algumas observações decorrem diretamente da discussão acima sobre portas e conexões. Um componente pode exportar portas *provides* ou *uses*, mas somente importa portas *provides*. No entanto, uma porta *provides* só pode ser importada se ela estiver conectada a uma das portas *uses* exportadas. Dessa forma, depois que um componente exporta uma porta *uses*, ela pode ser conectada a uma porta *provides*. Posteriormente, o primeiro componente importa essa porta *provides*, tendo assim acesso ao segundo componente. Na figura 3.1, o componente B exporta uma porta *provides* que contém funcionalidades que o componente A necessita e o componente A importa esta porta *provides* conectada a sua porta *uses* exportada.

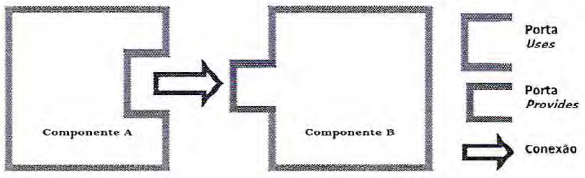


Figura 3.1: Exemplo de uma conexão de componentes usando CCA.

Com o intuito de tornar a discussão um pouco mais concreta, vejamos alguns detalhes da especificação CCA, e de sua utilização. Um componente deve implementar a *interface* `Component`, a qual inclui um método denominado `setServices`, usado para comunicação

entre o componente e uma plataforma. Esse método é chamado por uma plataforma no momento em que esta cria uma instância de um componente. Este método é o meio através do qual o componente exporta as portas que ele espera fornecer ou usar de outros componentes.

Um dos objetivos de `setServices` é fornecer a um componente um objeto do tipo `Services`, o qual guarda informações relativas às suas portas. Existem três métodos principais que permitem ao componente interagir com o objeto `Services`, a saber:

- `addProvidesPort`: adiciona portas *provides*;
- `registerUsesPort`: registra portas *uses*;
- `getPort`: busca uma porta *uses* ou *provides* previamente registrada.

Em termos gerais, a conexão ou associação entre dois componentes CCA envolve as seguintes etapas:

1. O componente que fornece o serviço registra sua disponibilidade na plataforma;
2. O componente que usará um serviço registra o seu pedido para a plataforma;
3. O componente usuário e o componente fornecedor são conectados;
4. O componente usuário faz a chamada ao método que precisa para com o serviço que foi concedido.

Neste processo, os componentes são instanciados como grandes caixas pretas. Para conectar uma porta, o usuário deve interagir com um *framework* e definir que outra porta deverá ser conectada a porta inicialmente escolhida. No entanto, a maneira na qual as portas são conectadas umas às outras não é sequer determinada pelo CCA, ou seja, o modelo de componentes CCA não define características próprias de uma conexão entre componentes.

Mostramos um exemplo com dois componentes simples: o componente "Componente 1" e o componente "Componente 2". O "Componente 1" possui somente uma porta *provides* e o "Componente 2" somente com uma porta *uses*. As duas portas têm a mesma *interface*. Nas tabelas 3.2 e 3.3 é mostrada a implementação do método `setServices` de ambos respectivamente. A implementação de um método do componente com porta *uses* é mostrada na Tabela 3.4. Nesse método, é usado o método `getPort` do objeto `Services`. Na Tabela 3.5, é mostrada também a implementação da porta *provides*.

Tabela 3.2: Implementação do método `setServices` do "Componente 1" na qual é declarada uma porta *uses*.

```
public void setServices(Services s) throws CCAException {
        String portUsesName = "comp1usesPortComp2";
        TypeMap map1 = s.createTypeMap();
        s.registerUsesPort(portUsesName, "uses", map1);
        services = s;
}
```

Tabela 3.3: Implementação do método `setServices` do "Componente 2". O "Componente 2" fornece uma porta *provides* para qualquer componente utilizar. Neste exemplo, o "Componente 2" pode ser considerado "Servidor".

```
public void setServices(Services s) throws CCAException {
        GoPort portSayhello2 = new MyPort();
        String portName = "comp2providesPortcomp1";
        TypeMap mapPort = s.createTypeMap();
        s.addProvidesPort(portSayhello2, portName, "provides", mapPort);
        services = s;
}
```

Tabela 3.4: Implementação da *interface* `GoPort` do "Componente 1".

```
private final class MyPort implements GoPort {
    public int go() {
        try {
            Comp1UsesComp2Port portNativeTmp = (Comp1UsesComp2Port)
            services.getPortNonblocking("comp1usesPortComp2");
            portNativeTmp.methodA();
        } catch (CCAException e) {
        return 0; }
}
```

Tabela 3.5: Implementação do método `methodA` do "Componente 2"

```
private final class Component2Port implements InterfaceComponent2Port {
    . . .
    public void methodA() {
                :
    }
}
```

Na Figura 3.2, mostramos os passos gerais da criação e do uso da conexão descrita, são eles:

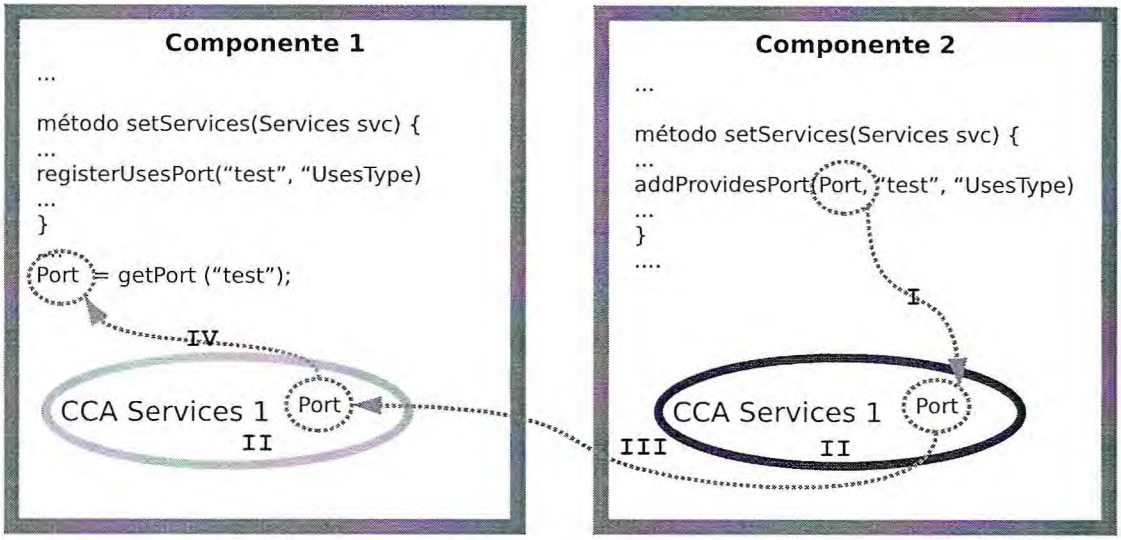


Figura 3.2: Diagrama do padrão CCA de projeto *Uses/Provides*

1. Em (I), "Componente 2" registra uma porta "test" chamando o método `addProvidesPort`;
2. Em (II), a plataforma cria os objetos `Services` dos "Componente 1" e "Componente 2" que armazena as portas registradas;
3. Em (III), quando é solicitada, a plataforma conecta o objeto `Services` do "Componente 2" ao objeto `Services` do "Componente 1";
4. Finalmente, em (IV), o "Componente 1" obtém a porta com uma chamada do método `getPort`.

3.2.2 Criação de Componentes e Realização de Conexões

Uma conexão CCA ocorre nos seguintes passos:

1. O usuário solicita a criação de componentes a uma plataforma CCA escolhida;
2. A plataforma CCA cria os componentes;
3. Durante a criação, os componentes através do método `setServices`, o qual expõem à plataforma, adicionam as portas *provides* que disponibilizam aos outros componentes e registram as portas *uses* que necessitam;

4. O usuário solicita à plataforma CCA a conexão (ou em inglês *binding*) de pares de portas: porta *uses* - porta *provides*;

5. A plataforma CCA efetua as conexões solicitadas;

6. O usuário solicita que a plataforma CCA inicie a aplicação através de uma porta `GoPort`;

7. A plataforma CCA inicia a aplicação;

8. Durante a execução da aplicação, os componentes que registraram portas *uses*, executam as chamadas das portas *uses* solicitadas e utilizam as conexões montadas.

`BuilderService` é uma porta contendo métodos que instanciam componentes ou conectam portas. Esta porta deve ser implementada por plataformas compatíveis com o modelo CCA para compor componentes e, de forma programada, criar/destruir componentes e fazer/desfazer conexões entre portas. Alguns dos métodos expostos pela `BuilderService` para o ciclo de vida do componente são `createInstance` para criar uma instância de um componente, e `destroyInstance` para excluir uma instância de um componente do escopo da plataforma [74]. Para propósitos de composição de aplicações com componentes, o `BuilderService` fornece um método `connect` para conectar de uma porta *uses* para uma porta *provides*, e fornece um método `disconnect` para desconectar uma conexão já existente [74].

3.2.3 Paralelismo no CCA

No CCA, a comunicação dentro de um componente paralelo ocorre por conta do próprio componente [10] (ver Figura 3.3).

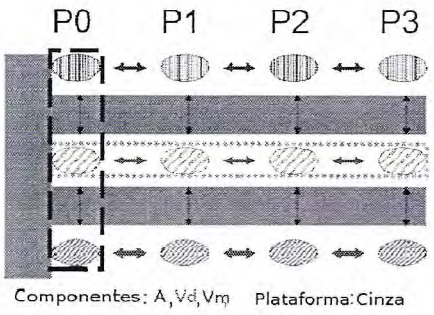


Figura 3.3: Exemplo de interação de componentes paralelos CCA.

Na Figura 3.3, os componentes no Vd, Vm e A se comunicam entre si utilizando portas da plataforma CCA. No entanto, dentro do mesmo componente os processos se comunicam através de camada de comunicação escolhida pelo componente. Por exemplo,

nesta Figura 3.3 o componente A poderia usar MPI para a comunicação entre os 4 (quatro) processos sobre o qual é distribuído, enquanto o componente Vd poderia usar memória compartilhada.

O padrão identificável mais comum na computação paralela é um padrão de dados múltiplos e programa único (SPMD), onde um programa idêntico é executado em todos os processos participantes, e os dados são decompostos entre os processos. Muitas vezes, simulações paralelas têm uma forma estritamente SPMD (por exemplo, uma simulação de combustão feito com o CCA ([75],2003 apud [76],2006, pág.05)). No entanto, às vezes a computação paralela consiste em programas SPMD múltiplos, divididos em máquinas paralelas iguais ou diferentes, que são acoplados com uma transferência de dados complexos (por exemplo, aplicações climáticas ([77], 2001 apud [76], 2006, pág.05). ([78], 2005 apud [76], 2006, pág.05)), ou um caso degenerado, no qual vários processos estão em comunicação com um único processo (por exemplo, para visualização ou registro de dados ([79], 1996 apud [76], 2006, pág.05)).

O CCA leva a extensão lógica do paradigma SPMD para o paradigma SCMD [76, pág.5]. O modo de programação SCMD trata cada componente como uma entidade caixa preta separada que é necessário ser instanciado e configurado de forma idêntica em todos os processos participantes. A Figura 3.4 representa o SCMD. Nesta figura, do ponto de vista da nomenclatura: existe um "*cohort*" de um componente (como o *Solver*), um conjunto de instâncias ligadas por setas horizontais pontilhadas representando a comunicação entre processos dentro de um único componente de memória distribuída. Um "*cohort*" de componentes é um conjunto de componentes que são identicamente instanciados em processos separados. Cada membro do grupo é ligado a diferentes instâncias de nós que residem no mesmo processo. Os nós (componentes distintos) são conectados por setas verticais que representam chamadas de função dentro do processo. Nós são agrupados no mesmo espaço de endereçamento do processo, representado por caixas retangulares. Nós são conectados usando o padrão de projeto *uses/provides* do CCA. Um "*cohort*" abrange todos os processos participantes e, juntos, os seus membros encapsulam o algoritmo paralelo que representa o componente (ver Figura 3.4).

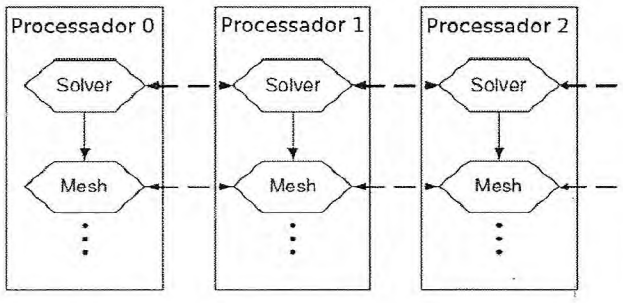


Figura 3.4: Diagrama do Padrão SCMD.

De outra forma, o paralelismo no CCA ocorre de forma "clandestina". Surge desta observação uma questão sobre como legalizar comunicações ponto-a-ponto ou coletivas que sigam o padrão MPI. Esta questão é discutida e resolvida nos Capítulos 4 e 5.

3.3 Plataformas CCA

Devido à vasta gama de pesquisa de componentes científicos que o CCA abrange, várias plataformas tem sido desenvolvidas com diferentes capacidades e limitações. Em última instância todos os recursos serão combinados, quer ampliando uma ou mais plataformas existentes, ou, pela especificação de pontes entre plataformas. Trabalhos recentes têm se concentrado na segunda abordagem, uma vez que permite que usuários acessem recursos de outras plataformas enquanto continuam o trabalho com a plataforma com a qual se sentem mais confortáveis. A principal divisão é entre plataformas diretamente conectadas e as plataformas distribuídas. Com algumas exceções, plataformas diretamente conectadas não têm a capacidade de gerenciar componentes distribuídos em uma vasta área de rede, enquanto as plataformas distribuídas não possuem a habilidade de conectar componentes paralelos não-isocardinal (por exemplo, para abordar o problema $M \times N$ - já visto no Capítulo 2). É importante notar que uma plataforma distribuída suporta componentes distribuídos, mas isso não implica que a própria plataforma tenha código distribuído. A seguir, descrevemos resumidamente as principais plataformas existentes CCA que serviram de inspiração para a plataforma que implementamos. Para comparar as principais plataformas, usamos uma tabela com as seguintes características: memória distribuída, memória compartilhada, componentes distribuídos, arquitetura, suporte ao problema $M \times N$ (definido no Capítulo 2) e suporte a linguagens de programação.

3.3.1 Ccaffeine

Ccaffeine [26] é uma ferramenta desenvolvida no Laboratório Nacional da Sandia (Califórnia, EUA), e, provavelmente é a plataforma mais usada para aplicações científicas complexas. Ccaffeine suporta componentes paralelos executando no mesmo espaço de memória, e pode usar ou não Babel (a versão "clássica" do Ccaffeine). Ele ainda não suporta componentes distribuídos. Aplicações baseadas em Ccaffeine são construídas com um sistema de *software* em camadas. As camadas estão estruturadas para apoiar três fases de desenvolvimento de aplicações de componente paralelo: montagem interativa, depuração e execução de produção para otimização.

Ccaffeine define uma linguagem simples baseada em linha de comando para criar e configurar aplicações de componentes. A implementação da GUI (do inglês *Graphical User Interface*) do Ccaffeine segue o padrão modelo-visão-controle (ou em inglês *Model-View-Controller* (MVC)) [26].

Ccaffeine disponibiliza um ambiente de alto desempenho para muitos modos interativos de *debug* e produção de muitas aplicações CCA baseadas em componentes. Estruturas de aplicação podem ser testadas interativamente e ajustadas durante

execuções de produção. O sistema de geração de código automático é limitado a aplicações SPMD, mas com pequenos ajustes pode ser usado para gerar aplicações MPMD (do inglês *Multiple Process, Multiple Data* ou *Multiple Program, Multiple Data*) baseadas em *scripts* de entradas múltiplas [80].

Além disso, Ccaffeine possui uma linguagem interativa usada para configurar sessões batch e interativas. Ele também permite que a GUI se comunique através de *socket* [81].

Abaixo segue uma tabela com as características escolhidas para comparação com outras plataformas:

Tabela 3.6: Características do Ccaffeine

Características	**Possui**	**Quais**
Memória Distribuída	Não	-
Memória Compartilhada	Sim	-
Componentes Distribuídos	Não	-
Arquitetura	SPMD e MPMD (restrito)	-
Suporte ao Problema $M \times N$	Sim	-
Suporte a Linguagens de Programação	Sim	Todas, Python é a mais adequada
Portas Coletivas	Não	-

3.3.2 XCAT

XCAT [82] é uma plataforma distribuída que suporta apenas componentes Java e não suporta componentes paralelos. XCAT utiliza o padrão de grade de serviços *web* para interações de componentes distribuídos e é uma plataforma de produção. XCAT-C++ [83] é a versão C++ do XCAT, que também fornece componentes como serviços *web*. Esta plataforma está atualmente em fase de transição para usar SOAP (do inglês *Simple Object Access Protocol*) como base para comunicações entre componentes e usar a biblioteca multi-protocolo Proteus [84]. XCAT-C++ tem o seu próprio gerador de *skeleton* e *stub* que utiliza descrições SIDL (do inglês *Scientific Interface Definition Language*) dos componentes. O XCAT-C++ não dá suporte a componentes paralelos ou ao problema $M \times N$ (definido no Capítulo 2), mesmo em caso de componentes não-distribuídos.

Algumas das principais características da arquitetura do XCAT são: `ComponentId`, `Services`, `Exceptions`, `BuilderService` e *interface* de *script*. Nesta plataforma, cada componente tem um único `ComponentID`. Além disso, foram adicionados alguns métodos ao XCAT: `getPortRef` e `setPortRef`. O método `getPortRef` obtém uma referência remota para uma porta *provides* que pode ser armazenada em *cache* quando uma conexão é feita entre uma porta *uses* local e uma porta *provides* remota. O `setPortRef` atribui uma referência para uma *cache* remota de porta *provides* quando uma conexão é feita entre uma porta *uses* local e um porta remota *provides*. `DisconnectProvider` desconecta porta *uses* a partir de uma conexão local que utiliza a porta. O método `disconnectUser` notifica uma porta *provides* remota a qual tem sido desconectada de uma porta *uses* [74].

Como o XCAT é uma plataforma distribuída de componentes, o método *createInstance* é capaz de criar instâncias de componentes nas locações remotas usando o protocolo GRAM (do inglês *Globus Resource Allocation Manager*) [85].

Para propósitos de rápida prototipagem, XCAT fornece uma *interface* para o `BuilderService` usando *scripts* Jython [82]. Jython é uma implementação Java puro da linguagem de *script* Python, e fornece uma *interface* para codificar escrita em Java.

Abaixo segue uma tabela com as características escolhidas para comparação com outras plataformas:

Tabela 3.7: Características do XCat

Características	**Possui**	**Quais**
Memória Distribuída	Sim	-
Memória Compartilhada	Não	-
Componentes Distribuídos	Sim	-
Arquitetura	Serviços Web	-
Suporte ao Problema $M \times N$	Sim	-
Suporte a Linguagens de Programação	Sim	C, C++, Fortran, Java e Python
Portas Coletivas	Sim	-

3.3.3 DCA

DCA [15] (do inglês *Distributed CCA Architecture*) é uma plataforma de pesquisa da Universidade de Indiana para explorar a semântica da invocação de método remoto paralelo - PRMI (do inglês *Parallel Remote Method Invocation*) - e trabalhar com o problema $M \times N$ (definido no Capítulo 2). DCA fornece uma *interface* similar ao MPI (do inglês *Message Passing Interface*) para transferências paralelas e manipula redistribuição de dados de um vetor de uma dimensão. Devido ao DCA envolver a experimentação de semântica do PRMI, ele não é uma plataforma de produção. DCA possui uma implementação parcial de um analisador SIDL.

DCA é uma plataforma distribuída CCA baseada em uma *interface* muito similar ao MPI. DCA usa construções MPI para resolver desafios de uma plataforma distribuída, tal como redistribuição de dados. Além disso, muitas implementações MPI são projetadas para aumentar a escalabilidade e o desempenho. Então, como o DCA tem fundamentação MPI, isto é uma vantagem. Outra vantagem é fornecer uma implementação alternativa de uma plataforma para testar o padrão CCA, especialmente nos aspectos relacionados com a distribuição de componentes paralelos [86].

Abaixo segue uma tabela com as características escolhidas para comparação com outras plataformas:

Tabela 3.8: Características do DCA

Características	Possui	Quais
Memória Distribuída	Sim	-
Memória Compartilhada	Não	-
Componentes Distribuídos	Sim	-
Arquitetura	SPMD	-
Suporte ao Problema $M \times N$	Sim	-
Suporte a Linguagens de Programação	Sim	C, C++, FORTRAN-95, JAVA, MATLAB, Python
Portas Coletivas	Sim	-

3.3.4 SCIRun

SCIRun [87] é uma plataforma distribuída que suporta componentes paralelos usando um subconjunto restrito de interações $M \times N$, e manipula dados de redistribuição automática.

SCIRun é um ambiente científico de solução de problemas que permite a construção interativa e *steering* de computações científicas de larga-scala [88, 89, 90, 91, 87].

Em SCIRun, entradas e parâmetros computacionais podem ser modificados interativamente, e os resultados destas mudanças fornecem respostas imediatas ao usuário. SCIRun é projetado para facilitar computação científica de larga-scala e visualização em uma larga abrangência de máquinas *desktop* para supercomputadores.

SCIRun constrói pontes entre diferentes modelos de componentes. Portanto, pode-se combinar um conjunto ímpar de ferramentas computacionais para criar poderosas aplicações com componentes cooperativos de diferentes fontes. SCIRun suporta computação distribuída, ou seja, componentes criados em computadores diferentes podem trabalhar em conjunto através de uma rede de alto desempenho e compor aplicações.

Abaixo segue uma tabela com as características escolhidas para comparação com outras plataformas:

Tabela 3.9: Características do SCIRun

Características	Possui	Quais
Memória Distribuída	Sim	-
Memória Compartilhada	Sim	-
Componentes Distribuídos	Sim	-
Arquitetura	SPMD	-
Suporte ao Problema $M \times N$	Sim	-
Suporte a Linguagens de Programação	Sim	C++, Java
Portas Coletivas	Não	-

3.3.5 MOCCA

MOCCA [92] é uma plataforma de componentes distribuídos compatível com CCA. MOCCA baseia-se na plataforma de metacomputação *H2O* [93]. A versão atual, chamada MOCCA_Light [92, 94], é uma implementação em Java puro da plataforma CCA e permite construir aplicações de componentes em recursos distribuídas disponíveis através de *H2O*.

Novas funcionalidades de MOCCA incluem o suporte experimental para manipulação de múltiplos componentes e a parametrização do número de portas e conexões. Além disso, existe também o esforço para a descoberta de serviços utilizando provedores JNDI (do inglês *Java Naming and Directory Interface*) incluídos em *H2O*. Estas características ainda são experimentais, por isso, a implementação e a API (do inglês *Application Programming Interface*) podem mudar em lançamentos futuros.

Em MOCCA, cada componente é executado como um *pluglet* separado implantado no `kernel` do *H2O*. Por usar o mecanismo *H2O* de implantação de *pluglets* no `kernel`, MOCCA permite fácil instalação e criação de instâncias de componentes em recursos compartilhados distribuídos.

Abaixo segue uma tabela com as características escolhidas para comparação com outras plataformas:

Tabela 3.10: Características do MOCCA

Características	**Possui**	**Quais**
Memória Distribuída	Não	-
Memória Compartilhada	Sim	-
Componentes Distribuídos	Não	-
Arquitetura	-	-
Suporte ao Problema $M \times N$	Não	-
Suporte a Linguagens de Programação	Sim	C, C++, Fortran 77 ou 90, Java e Python
Portas Coletivas	Não	-

3.3.6 Decaf

Decaf [95] é uma plataforma, originalmente desenvolvida para compreender a especificação CCA para verificar o SIDL e o Babel. Decaf é uma plataforma diretamente conectada e usa Babel como seu sistema em tempo de execução. O mais recente desenvolvimento investigado do Decaf é a ligação entre ele e as plataformas distribuídas SCIRun e XCAT-C++. Essa plataforma é uma prova da aplicabilidade do Babel em CCA.

As duas classes mais importantes da implementação do Decaf são: `framework` e `Services`, considerando que a única exigência do CCA é utilizar as *interfaces* SIDL. Decaf foi implementado em C++ e usa a biblioteca padrão do C++ (STL, do inglês *Standard Template Library*).

A classe `framework` do Decaf mantém três mapas. O primeiro mapa, conhecido como instância *d*, é um mapa de nomes únicos de instâncias para um par de tuplas. Os *alias*

são criados pelo método `getServices` que cria serviços que são serviços de objetos que não estão associados com uma instância de componente em particular.

O segundo mapa, denominado conexão d, armazena o `ConnectionID` baseado em uma *string* única com o nome de uma instância de um componente e o nome da porta.

Por último, o mapa *alias* d é um mapa de nomes de instâncias para os nomes das classes que representam. Este mapa é apenas acessado e modificado pelo método `getServices` [95].

Segue uma tabela com as características escolhidas para comparação com outras plataformas.

Tabela 3.11: Características do Decaf

Características	Possui	Quais
Memória Distribuída	Sim	-
Memória Compartilhada	Sim	-
Componentes Distribuídos	Sim	-
Arquitetura	SPMD	-
Suporte ao Problema $M \times N$	Sim	-
Suporte a Linguagens de Programação	Sim	C++, C, Java, Python e Fortran 77
Portas Coletivas	Não	-

3.4 O Modelo de Componentes *Hash* (#)

Nesta seção são mostradas as características do modelo # (lê-se em inglês *Hash*). O modelo # proposto em [96] move a programação paralela da perspectiva baseada em processo à programação orientada a interesses ortogonais.

A organização desta seção está definida a seguir. Na primeira subseção 3.4.1, descrevemos as premissas do modelo #. Na subseção 3.4.2, definimos a decomposição em interesses e os conectores e o seu tratamento. Na subseção 3.4.3, mostramos as espécies de componentes. Em seguida, na subseção 3.4.4 formalizamos a semântica de composição de componentes #. Ao final, na Subseção 3.4.5, mostramos o ciclo de vida de componentes #, a arquitetura # e ambiente de programação *Hash*.

3.4.1 Premissas do

O modelo # propõe uma noção de componentes que são paralelos e mostra como eles podem ser combinados para formar novos componentes e aplicações [97]. No modelo #, os componentes paralelos são chamados de componentes # e possuem a habilidade de serem implantados em um *pool* de nós computacionais (por exemplo: *clusters* ou grades). Para isso, um componente # é formado por um conjunto de "partes", chamadas unidades, cada uma implantada em um nó computacional, as quais interagem para executar uma tarefa paralela.

Separação de interesses [98] é um paradigma de que tenta formalmente separar o algoritmo básico de interesses de propósitos especiais tais como sincronização, concorrência, distribuição, persistência, restrições de tempo-real, recuperação de falhas e controle de localização. Este paradigma de programação consiste em separar interesses antes de compô-los para produzir *softwares* [99]. Alguns interesses especiais existem para preencher requisitos especiais da aplicação (tempo-real, persistência e distribuição) ou para gerenciar e otimizar o algoritmo computacional básico (controle de localização, concorrência, por exemplo) [98]. Um componente # é capaz de tratar de um interesse envolvendo unidades de um conjunto de processos envolvidos, de tal forma que cada unidade representa o papel do processo naquele interesse, isto será explicado na próxima subseção 3.4.2. Sob a perspectiva de processos, cada unidade constitui uma fatia de um processo pertencente a uma aplicação que faz uso do componente #.

O modelo de componentes # tenta aumentar os níveis de modularidade e abstração de programas paralelos por visualizá-los de uma perspectiva orientada a componentes. É importante enfatizar que o modelo # não define conectores específicos de sincronização. Estes são definidos pelos sistemas de programação #.

Sistema de Programação #: define a natureza concreta de um componente #. Para tanto, estes devem definir espécies de componentes apropriados ao ambiente de computação alvo, associadas a requisitos de um certo nicho de aplicação [100].

Um sistema de programação # define a natureza dos componentes #, ou seja, ele define um conjunto de espécies de componentes. Todo componente # em um sistema de programação # pertence a uma espécie (conceito que será definido nas próximas subseções).

Um sistema de programação baseado em componentes é compatível com o modelo de componentes # caso eles possuam as seguintes características [97]:

- Componentes são construídos a partir de um conjunto de partes, chamadas unidades, cada um supondo ser implantado em um nó de uma plataforma de execução de computação paralela;
- Componentes podem ser combinados para formar novos componentes e aplicações por meio de composição usando sobreposição, um tipo de composição hierárquica;
- Cada componente pertence a um conjunto finito de um tipo de componente suportado.

No senso do modelo #, um componente é a realização de um interesse funcional ou não, tais como [101]: computações paralelas e sequenciais, alocação de processos em processadores, depuração paralela, operações de entrada e saída, serviços de grade, funcionalidades de bibliotecas científicas de propósito específico, estratégias de balanceamento de carga estáticas e dinâmicas, entre outros. Especificação da configuração e da computação são separados em componentes compostos e simples, respectivamente.

No contexto do #, os conceitos de interesses e processos usados são descritos a seguir:

Interesses: são unidades primárias da decomposição de um *software*. Eles podem ser funcionais, descrevendo computações, ou não-funcionais, descrevendo aquilo que afeta computações [101]. Por exemplo, um interesse pode ser sincronização, concorrência, distribuição, persistência ou recuperação de falhas [98]. Estes interesses podem ser divididos em interesses funcionais e não-funcionais. Para exemplificar interesses não-funcionais, podemos citar: localização física de um processo e desempenho. Para os interesses funcionais, um exemplo seria uma parte de um *software* com um objetivo específico [101].

Processo: sob a perspectiva do modelo #, os processos que compõem um programa paralelo podem ser decompostos em várias fatias (do inglês *slices*) segundo algum critério e fatias de diferentes processos podem se agrupar em um interesse comum.

O modelo # aproxima-se mais da abordagem usada em Engenharia de *software* ao estabelecer que os interesses são o foco principal do projeto da aplicação. Ortogonalmente a isso, está a concepção da visão em processos como uma consequência [100].

A Figura 3.5 fornece uma noção intuitiva dos componentes #, considerando a estrutura básica de programas paralelos, como um conjunto de processos se comunicando por troca de mensagens. Para isso, é utilizado um programa paralelo que calcula $\mathbf{A} \times \hat{\mathbf{x}} \bullet \mathbf{B} \times \hat{\mathbf{y}}$, onde $\mathbf{A}_{m \times n}$ e $\mathbf{B}_{m \times k}$ são matrizes e $\hat{\mathbf{x}}_{n \times 1}$ e $\hat{\mathbf{y}}_{k \times 1}$ são vetores.

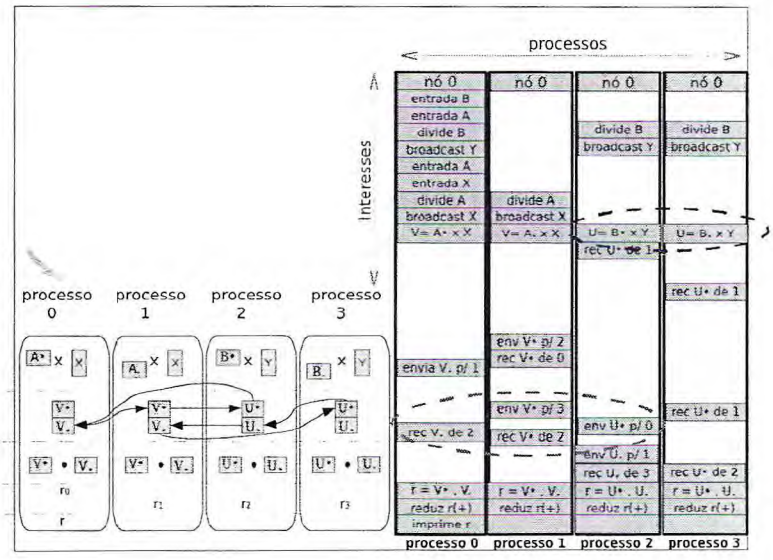


Figura 3.5: Representação de Processos para Componentes #.

Para isto, o programa paralelo é formado por N processos coordenados em dois grupos, denominados p e q, com processos de M e P, respectivamente. Na Figura 3.5, $M = P = 2$, $p = \{\text{processo 0, processo 1}\}$ e $q = \{\text{processo 2, processo 3}\}$. Na primeira fase, os

processos em p calculam $\mathbf{v} = \mathbf{A} \times \mathbf{x}$, enquanto os processos em q calculam $\mathbf{u} = \mathbf{B} \times \mathbf{y}$, onde $v_{m\times 1}$ e $u_{m\times 1}$ são vetores intermediários.

Na Figura 3.5, os processos que constituem o programa paralelo descrito acima é fatiado, de acordo com a noção de interesse de *software*.

Exemplo de Decomposição em Interesses

Em [100], tem-se um exemplo mais detalhado sobre a decomposição em interesses, que demonstra a ideia do fatiamento de processos em interesses. Neste item, este exemplo é mostrado de forma resumida. No exemplo considera-se o seguinte problema: sejam A e B duas matrizes quadradas $n \times n$ e $\mathbf{x}$ e $\mathbf{y}$, dois vetores $1 \times n$ (onde $n > 0$). Para calcular este resultado, considera-se um algoritmo paralelo para este cálculo em quatro processadores (P_0, P_1, P_2 e P_3) distintos. Ao lado disso, o processador P_0 executa o processo "raiz" (em inglês "root"), ou seja, se responsabiliza por iniciar a computação e recolher o resultado final r. A distribuição inicial dos dados ficará da seguinte forma: o processador P_0, após inicializar as matrizes e os vetores, armazenará a metade superior (linha 0 até a linha $n/2-1$) da matriz A consigo e enviará a metade inferior de A (linha $n/2$ a linha $n-1$) ao processador P_1 (por enquanto, vamos abstrair a forma de envio de dados entre processadores). P_0 enviará, também, o vetor $\mathbf{x}$ inteiro ao processador P_1 e manterá uma cópia consigo. De posse da matriz $\mathbf{B}$ e o vetor $\mathbf{y}$, P_0 enviará as metades superior e inferior da matriz $\mathbf{B}$ para os processadores P_2 e P_3 , respectivamente. Já o vetor $\mathbf{y}$ será enviado por inteiro aos processadores P_2 e P_3. Após realizada a distribuição descrita, configura-se o seguinte cenário: P_0 de posse da metade superior de $\mathbf{A}$ ($\mathbf{A}_0$) e do vetor $\mathbf{x}$; P_1 de posse da metade superior de $\mathbf{A}(\mathbf{A}_1)$ e do vetor $\mathbf{x}$; P_2 de posse da metade inferior de $\mathbf{B}$ ($\mathbf{B}_0$) e do vetor $\mathbf{y}$; finalmente, P_3 de posse da metade superior de $\mathbf{B}$ ($\mathbf{B}_1$) e do vetor $\mathbf{y}$.

A distribuição mostrada objetiva paralelizar o cálculo da multiplicação das matrizes nos processadores. Conforme esta distribuição, o processador P_0 calcula $\mathbf{v}_0 = \mathbf{A}_0 \times \mathbf{x}^T$ e P_1 calcular $\mathbf{v}_1 = \mathbf{A}_1 \times \mathbf{x}^T$. O vetor $\mathbf{v}$ é o resultado dessa multiplicação, e o mesmo está distribuído em $P_0(\mathbf{v}_0)$ e $P_1(\mathbf{v}_1)$. O mesmo ocorre para P_2 e P_3 onde é criado o vetor $\mathbf{u}$ que é dividido em $\mathbf{u}_0 = \mathbf{B}_0 \times \mathbf{y}^T$ e $\mathbf{u}_1 = \mathbf{B}_1 \times \mathbf{y}^T$. Com estas operações, foi calculado o produto de matrizes por vetores de forma paralela, gerando dois vetores intermediários $\mathbf{u}$ e $\mathbf{v}$.

Considerando o objetivo, o passo seguinte é o cálculo do produto vetorial $r = \mathbf{u} \times \mathbf{v}$. O cálculo citado deve ser efetuado de forma paralela nos processadores. Para isto, é necessário uma redistribuição dos vetores apresentados na Figura 3.6. A nova divisão será da seguinte forma: dividir os vetores resultantes em cada processador, ou seja, $\mathbf{v}_0$ que está localizado em P_0 será dividido pela metade em $\mathbf{v}_0 - superior$ e $\mathbf{v}_0 - inferior$ (superior para a metade superior e inferior para a metade inferior). Analogamente, nos outros processadores temos que P_1 irá gerar $\mathbf{v}_1 - superior$ e $\mathbf{v}_1 - inferior$, P_2 apresentará $\mathbf{u}_0 - superior$ e $\mathbf{u}_0 - inferior$ e P_3 com $\mathbf{u}_1 - superior$ e $\mathbf{u}_1 - inferior$. A nova configuração mostrada possível a redistribuição dos dados nos diversos processadores. P_0 envia a P_1

$\mathbf{v}_0 - inferior$ e P_1 envia a P_2 $\mathbf{v}_1 - superior$.

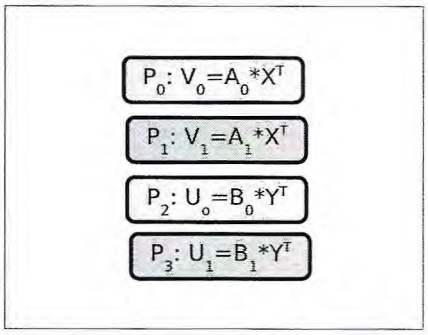


Figura 3.6: Separação da computação nos processos envolvidos.

Depois da redistribuição mostrada dos vetores *inferior* e *superior*, cada processador deve efetuar a multiplicação dos vetores e armazenar os dados em variáveis locais que representam uma fração do r. Na conclusão das operações, deve ser feito o somatório dos valores locais no processador P_0, pois o resultado final é a soma das partes, ou, $r = r_0 + r_1 + r_2 + r_3$.

Segundo [100], este exemplo pode ser dividido em vários interesses funcionais. à válido ressaltar que esses interesses trabalham sobre dados diferentes. A explicação horizontal da separação funcional feita, concentrando-nos no processador P_0, é mostrada a seguir:

1. `Inicializar`: Este interesse pertence apenas a P_0. Ele se encarrega de inicializar as estruturas **A**, **B**, **x** e **y** com valores inteiros aleatórios, para fins de teste.

2. `Distribuir`(**A**,**x**): Este interesse é responsável em distribuir as respectivas metades da matriz **A** e o vetor **x** (completo) entre os processadores P_0 e P_1.

3. `Distribuir`(**B**,**y**): Este interesse é similar ao último interesse citado, ele distribui as respectivas metades da matriz **B** e o vetor **y** entre P_2 e P_3. é um interesse relativo a P_0, P_2, P_3 e P_0, pois este processador é quem inicia as estruturas de dados, sem efetuar computação.

4. `MultMatVet`(**A**,**x**) e MultMatVet(**B**,**y**):

   - `MultMatVet`(**A**,**x**): Este interesse multiplica **A** e $\mathbf{x}^T$ e armazena o resultado em **v**:
   - `MultMatVet`(**B**,**y**): Este interesse multiplica **B** e $\mathbf{y}^T$ e armazena o resultado em **u**.

5. `Redistribuir`(**u** ,**v**): Este interesse reparte os vetores **u** e **v** entre os todos os processadores, sem a criação de novas estruturas de dados.

6. `MultVetVet(u, v)`: Este interesse calcula para cada processador um escalar r.

7. `Acumular(r)`: Este interesse envia o resultado calculado no interesse anterior em cada processador para P_0. P_0, além de enviar seu resultado a ele mesmo, soma os resultados obtidos nos outros processadores, e dele mesmo, no escalar r.

Resumindo, a multiplicação de dois vetores é um interesse comum a todos os processos. Cada processador necessita de uma rotina que efetue essa operação. A criação e distribuição das matrizes e vetores através dos processos é um interesse que diz respeito a apenas ao processo P_0. Receber o vetor **x** é um interesse que diz respeito a apenas os processos P_0 e P_1. Receber o vetor **y** é interesse dos processos P_2 e P_3.

Conforme[100], com esta visualização horizontal do programa, por interesses, o # vai de encontro à visão verticalmente centrada, nos processos, comum na programação paralela usual. Portanto, o modelo # propõe uma visão baseada em interesses e a programação paralela sob a perspectiva #, é orientada a interesses. Esta visão baseada em interesses permite a localização de diferentes tipos de informação nos programas, fazendo-os fáceis de escrever, entender, reusar e modificar [98].

3.4.2 Conectores

Com o foco em interesses, programadores podem construir componentes # combinando-os através da sobreposição (em inglês *overlapping*). O compartilhamento de código entre os componentes é feito através de um mecanismo de fusão entre os interesses internos aos mesmos [100].

Os tipos de conectores suportados podem ser considerados com uma característica que diferencia modelos e arquiteturas de componentes [22].

Em adicional, os componentes # que implementam conectores possuem uma coordenação exógena e são baseados em eventos, ou seja, são sincronizados a partir de eventos que disparam suas computações. Coordenação exógena estabelece que as primitivas que causam e afetam a interação de uma entidade com outras residem em outras entidades [102].

Tratamento Uniforme de Conectores No #, conforme a definição de componente, existe um tratamento uniforme entre os conceitos de componente e conector sob a forma de um único conceito, o componente #.

Este tratamento uniforme torna possível que uma plataforma ou infra-estrutura de componentes # suporte qualquer tipo de conector que possa ser programado como um componente #, inclusive os que implementam a relação par-a-par (em inglês *peer-to-peer*) entre componentes. Este tratamento de conectores como componentes # é a consequência direta do fato de que estes estão implantados em um conjunto de máquinas, através de suas unidades a quais implementam os papéis envolvidos no conector [100].

3.4.3 Espécies de Componentes

Plataformas usuais de componentes definem apenas uma espécie de componente, que se destina a resolver alguns interesses funcionais, com um conjunto fixo de conectores, tomadas como entidades distintas em relação aos componentes. O modelo de componentes de uma plataforma de componentes é definido pelos componentes e as regras de composição com outros componentes [103]. Para entender melhor, segue a definição de espécie:

Espécie: uma espécie de componente agrupa componentes # que são definidos em termos das mesmas unidades da composição de um software [100].

A espécie define a natureza concreta dos componentes # que a ela pertence e seus modelos de implantação. Conectores podem ser definidos com espécies de componentes #.

Sistemas de programação # são diferenciados devido ao suporte para muitos tipos de componentes, cada um especializado para resolver determinado tipo de interesses, funcionais ou não-funcionais existentes. Em [97], encontrou-se as seguintes utilizações principais das espécies de componentes:

- Conectores são tomados como espécies específicas de componentes, tornando possível para um programador desenvolver conectores específicos para a utilização das suas aplicações ou bibliotecas de conectores para reuso.
- Espécies de componentes podem ser utilizadas como abstrações para definir blocos de aplicações em domínios específicos das ciências e engenharia, proveniente de especialistas nestes domínios.
- No contexto de CAD, garantir a interoperabilidade na implementação dos plataformas computacionais existentes baseados em componentes é considerado um problema difícil.

3.4.4 Semântica de Composição

Esta subseção formaliza a semântica de composição de componentes #. Em alguns trabalhos anteriores, a composição por sobreposição foi formalizada através de um *calculus* de termos, chamados HOCC [104] (do inglês *Hash Overlapping Composition Calculus*) , e teoria das instituições [97]. Nesta tese, como mostrado em [97], para fornecer uma explicação mais intuitiva e simplificada da semântica de composição é usada uma linguagem chamada HCL [105] (do inglês *Hash Configuration Language*).

A configuração é uma especificação de um componente #, que pode ser abstrato ou concreto. Conceitualmente, em um sistema de programação #, um componente # é sintetizado em tempo de compilação ou de tempo de inicialização usando as informações de configuração, combinando partes de *software* cuja natureza depende da espécie de

componente. No ambiente HPE (do inglês *Hash (#) Programming Environment*), unidades de um componente # são classes C#.

Teoricamente, um componente abstrato # especifica o interesse de todos os seus componentes #. Seus parâmetros de espécies, delimitada por colchetes, determinam o contexto de uso para as quais os componentes concretos # devem ser especializados.

3.4.5 Ambiente para Construção de Sistemas de Programação

Para permitir uma construção e configuração visual de componentes #, foi criado um ambiente para construção de sistemas de programação conforme a arquitetura # proposta em [106]. Neste ambiente, os programadores poderiam configurar componentes e tipos de programas baseados em espécies de componentes.

Ciclo de Vida de Componentes

O ciclo de vida dos componentes # é controlado pela plataforma de implementação, implementando os padrões de projeto e abstrações subjacentes ao modelo de componentes. O modelo de componentes # também suporta composição hierárquica, mas as suas configurações são estáticas, como em Haskell# [107] evitando *overheads* de desempenho de tempo de execução adicional subjacentes ao ambiente em tempo de execução [21]. No caso do HPE, os seguintes estágios são definidos em [100]:

Descoberta: estágio no qual o desenvolvedor faz uma requisição ao ambiente sobre o conjunto de componentes disponíveis no momento.

Configuração: neste estágio, uma vez escolhidos os componentes com os quais o desenvolvedor irá trabalhar, o mesmo irá configurá-los de acordo com as suas necessidades, compondo-os por sobreposição para formação de novos componentes #.

Publicação: o desenvolvedor torna um componente ou uma aplicação disponível a outros desenvolvedores. Uma vez publicado, um componente pode ser descoberto e usado como um componente interno em uma nova configuração.

Implantação: neste estágio, o desenvolvedor implanta um novo componente ou aplicação em alguma plataforma de computação para executá-lo.

Produção: finalmente, neste estágio, o desenvolver torna o sistema disponível para execução e monitoração em tempo de execução.

Arquitetura

A arquitetura #, utilizada na plataforma HPE, é formada por três módulos distintos: o `Back-End`, o `Front-End` e o `Core`, os quais possuem responsabilidades diferentes em relação aos estágios do ciclo de vida de componentes # [21]. Os três módulos (Figura 3.7) são:

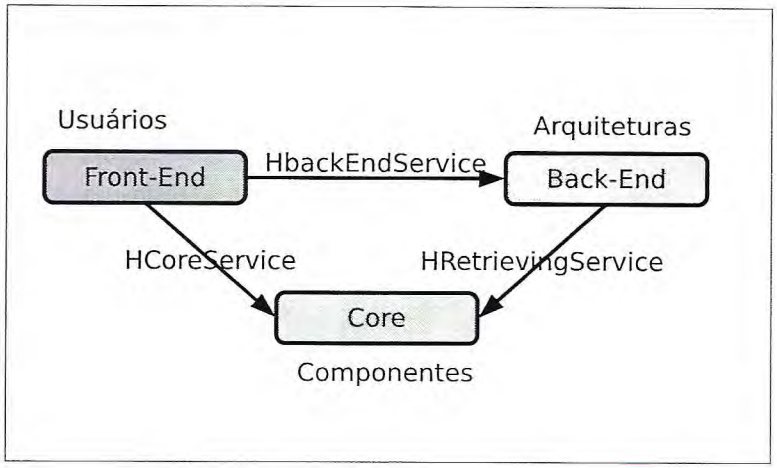


Figura 3.7: Arquitetura *Hash*.

`Front-End`: este módulo permite aos programadores criar configurações de componentes # e controlar seu ciclo de vida;

`Core`: tem como função oferecer serviços de configuração de componentes # e aplicações, através da sobreposição de componentes #.

`Back-End`: é um módulo que gerencia a infra-estrutura de componentes onde os componentes # são implantados e as plataformas de execução onde executam.

As *interfaces* entre estes três componentes foram implementados como serviços *web* (ou em inglês *Web Services*) para promover a independência das mesmas, sobretudo na questão de localização e de plataforma de desenvolvimento. Por exemplo, a partir de um `Front-End` um usuário pode ligar para qualquer `Core` e/ou `Back-End` de interesse que podem ser descobertos usando serviços UDDI (do inglês *Universal Description, Discovery and Integration*). O `Back-End` do HPE foi implementado através da extensão da plataforma CLI/Mono, enquanto o `Front-End` e do `Core` foram implementadas em Java, utilizando o padrão de projeto MVC.

HPE

O ambiente de programação *Hash*, chamado HPE (do inglês *Hash (#) Programming Environment*), é um sistema de programação # baseada em uma arquitetura proposta

para infra-estruturas (em inglês *framework*) a partir das quais plataformas de programação para domínios de aplicação específica podem ser instanciadas [21].

O sistema de tipos proposto em HPE garante certo nível abstração aos programadores em relação aos detalhes da arquitetura sobre a qual um componente encontra-se implantado [100].

A plataforma HPE se baseia na arquitetura #. Esta plataforma roda sob o Eclipse e provê uma base para implementação de ambientes de programação orientado a componentes #. O ambiente # foi instanciado sobre a plataforma citada [100].

A plataforma # especializa arquitetura do modelo de componentes # para suportar as seguintes espécies do HPE: qualificadores, arquiteturas, ambientes, estruturas de dados, computações, sincronizadores e aplicações [21]. Em [100], estas espécies são definidas resumidamente, seguem abaixo as definições:

Arquiteturas: estas espécies descrevem arquiteturas computacionais onde os componentes serão implantados.

Ambientes: descrevem a tecnologia de *software* utilizada para permitir o paralelismo na arquitetura escolhida.

Estruturas de dados: representam estruturas de dados, possivelmente paralelas, manipuladas por uma ou mais computações em uma aplicação.

Computações: espécies que especificam a computação paralela, denotando um interesse funcional necessário em aplicações.

Sincronizadores: estas espécies representam o meio pelo qual computações sincronizam o seu estado, ou comunicação e procedimentos.

Aplicações: representam uma computação relativa a uma aplicação final, a qual inclui o código principal.

Qualificadores: por último, espécies que representam características não-funcionais de um componente # a qual é relevante para a sua semântica ou desempenho.

3.5 O Modelo de Componentes Fractal

Fractal é um modelo de componentes modular e extensível que pode ser usado junto com várias linguagens de programação [108].

O modelo de componentes de Fractal é hierárquico, pois os componentes podem ser aninhados em componentes compostos (o que originou o nome do modelo de componentes "Fractal"); reflexivo, pois os componentes têm potencialidades completas de introspecção e de intercessão; e, aberto, pois os serviços extra-funcionais associados a um componente. Introspecção [109] é o conceito de prover *interfaces* de componentes separadas para

observação de comportamento. Intercessão [109] é o conceito de fornecer *interfaces* de componentes distintas para mudança de comportamento.

Componentes compostos (para ter uma visão uniforme das aplicações em vários níveis de abstração), componentes compartilhados (para modelar recursos), potencialidades de introspecção (para monitorar um sistema executando), e potencialidades da configuração e de reconfiguração (para executar e reconfigurar dinamicamente uma aplicação) são as principais características do Fractal. Outro objetivo do modelo de Fractal é ser aplicável a vários tipos de *softwares*. Infelizmente, as características avançadas do modelo de Fractal têm um custo que não é sempre compatível com os recursos limitados de ambientes restritos [108].

Visando alcançar estes objetivos contraditórios, o modelo de componentes Fractal não é definido com uma especificação extensa e fixa, que todos os componentes Fractal devem seguir, mas, como um sistema extensível de relações entre os conceitos bem definidos e APIs correspondentes que os componentes Fractal podem ou não podem executar, dependendo do que podem ou querem oferecer a outros componentes.

A especificação do Fractal define quatro níveis do conformidade. No nível 0, nada é imperativo e mesmo os objetos simples são componentes Fractal. O nível 1 adiciona o requisito que todos os componentes têm que prover a *interface* `Component`, que permite a introspecção do componente (por exemplo, descobre todas as interfaces de um componente). No nível 2, todas as interfaces dos componentes têm que estender a *interface* `Interface`, o que permite a introspecção da interface. O nível final 3 é o mesmo que o nível 2. Para todos os níveis acima mencionados, há o nível X.1, que adiciona o requisito que todos os componentes têm que fornecer *interfaces* definidas de controle. Além disso, há os níveis 3.2 e 3.3, que adicionam regras sobre fábricas de componentes [108].

Como descritos acima, as implementações Fractal podem adotar somente conceitos selecionados de Fractal. Consequentemente, a família de especificações Fractal com implementações compatíveis pode ser enorme. Para criar um meta-modelo para uma implementação Fractal específica não é difícil. Porém, devido à modularidade da especificação, para projetar e desenvolver um meta-modelo que permita descrever qualquer implementação Fractal e controlar seus metadados não é uma tarefa direta [108].

Existem algumas plataformas que usam o modelo de componentes Fractal, entre os quais, podemos citar:

- Proactive [110], uma plataforma baseada em um *middleware* (modelo de programação e ambiente) para computação distribuída, móvel, orientada a objetos e paralela;
- DREAM [111], uma plataforma para construção de MOM (do inglês *Message-Oriented Middleware*) reconfiguráveis;
- GoTM [112] uma plataforma baseada em Fractal para construir serviços de transação de *middleware*;

- JULIA [108], uma implementação Java do modelo Fractal, uma plataforma de tempo de execução pequeno, mas eficiente, que depende de uma combinação de interceptores para a programação de recursos reflexivos dos componentes.
- GCM (do inglês *Grid Component Model*) [113], que baseia-se na especificação formal, e propõe uma extensão adaptada para grade computacional. As características essenciais propostas pelo GCM incluem: suporte para a reflexão, estrutura hierárquica, modelo de extensibilidade, apoio a adaptatividade e interoperabilidade.

Capítulo 4

Expansão do Modelo de Conexão do CCA

Este capítulo apresenta o uso de conectores exógenos e endógenos no CCA (do inglês *Common Component Architecture*). Na Seção 4.1, descrevemos uma visão geral do modelo de implementação de comunicação do CCA em relação aos conectores. Na Seção 4.2, mostramos os conectores endógenos e o uso destes conectores no modelo CCA em conexões diretas e indiretas. Na Seção 4.3, detalhamos os conectores exógenos e mostramos seu uso no modelo CCA. Finalmente, apresentamos as considerações finais.

Para o presente trabalho, escolhemos o modelo de componentes CCA e não o # porque tínhamos o objetivo de englobar também aplicações distribuídas e o # é mais adequado a aplicações paralelas. Do modelo Fractal, usamos a ideia de aninhamento como inspiração para construir componentes do tipo conector e estes componentes serem utilizados para compor aplicações.

4.1 Implementação da Comunicação

A especificação do modelo de conexão do CCA concentra-se na conexão de componentes através de suas portas, sem estabelecer os detalhes das comunicações envolvidas nessas conexões. Algumas considerações sobre essas comunicações são essenciais para a discussão sobre conectores. Tais considerações são feitas sob a luz de dois aspectos fundamentais. O primeiro aspecto é, naturalmente, o desempenho. O segundo aspecto, menos explorado na literatura especializada, é a capacidade que o modelo de conexão apresenta de expressar modelos de computação paralela ou distribuída, e até mesmo de expressar combinações desses modelos.

Do ponto de vista da implementação, estabelecer um modelo de componentes implica que as construções descritas por modelos de computação paralela ou distribuída devem ser implementadas com uma combinação adequada de construções do modelo de componentes. Como consequência, essa capacidade de expressão de um modelo de componentes poderia agir como um modelo unificador dos modelos de computação

paralela ou distribuída. Com o objetivo de introduzir este aspecto no modelo CCA, impomos a restrição adicional, nem sempre atendida nas aplicações CCA descritas na literatura, de que comunicações dentro de componentes não são permitidas (supomos aqui que a comunicação entre componentes diferentes já estão restritas às conexões entre portas pelo próprio modelo CCA original). Essa restrição significa que, ao longo deste capítulo (e mesmo nos capítulos seguintes), trabalhamos com componentes CCA que ocupam um único espaço de endereçamento de memória. Portanto, o único paralelismo possível intra-componente é proveniente do uso de múltiplas tarefas, com memória compartilhada. Assim sendo, uma aplicação distribuída só pode ser construída pela composição de diferentes componentes, em diferentes espaços de endereçamento.

Vale a pena insistir ainda um pouco mais neste aspecto, visto que muitas aplicações hoje existentes não satisfazem essa restrição. Há dois casos que merecem ser comentados. O primeiro deles diz respeito aos inúmeros códigos legados paralelos de CAD para sistemas de memória distribuída nos quais comunicações são feitas através de troca de mensagens. Algumas adaptações desses códigos ao modelo CCA propõem encapsular toda uma computação distribuída em um mesmo componente, mantendo as trocas de mensagens efetuadas no interior desse componente. Imagine, por exemplo, um código distribuído que realize a multiplicação de duas matrizes, usando MPI para as comunicações. Conforme as restrições adotadas essas comunicações são "ilegais" ou "clandestinas", pois ocorrem intra-componente (veja ilustração na Figura 4.1). Segundo esse ponto de vista, a transformação desse código distribuído em componentes CCA deve ser feito de maneira a ter apenas comunicações "legais".

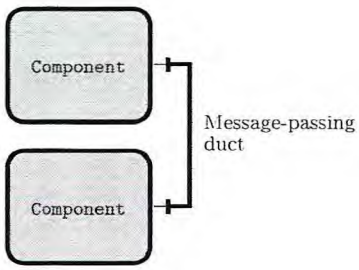


Figura 4.1: Um exemplo de conexão "clandestina" entre dois componentes.

O segundo caso que merece comentário são os conceitos de "componentes compostos" ou "componentes distribuídos" que aparecem em outros modelos de componentes para CAD, tais quais os modelos Fractal e #. à primeira vista, esses são conceitos não cobertos pela forma que usamos o modelo CCA. No entanto, é possível imaginar que o modelo que estamos considerando poderia ser visto como um modelo de baixo nível em um hierarquia de modelos de componentes. Em tal hierarquia, níveis mais altos poderiam definir componentes compostos, formados internamente por componentes ou conectores CCA. Essa parece ser uma via bastante promissora, embora fora do escopo desta tese.

Para concluir as considerações sobre o aspecto de desempenho das comunicações no

modelo CCA, recorremos novamente à experiência acumulada com as aplicações que aparecem na literatura. Conforme mencionado no Capítulo 1 desta tese, essas são quase que exclusivamente aplicações de simulação numérica, como por exemplo [114, 19]. Constatamos nas mesmas que a conexão direta é um imperativo de desempenho quando os componentes envolvidos na referida conexão compartilham um mesmo espaço de endereçamento. Esse fato nos obriga a manter esse tipo de conexão no modelo de conexão expandido que propomos. Porém, incluímos em nosso modelo, a possibilidade de composição de conexões diretas.

O segundo aspecto sobre comunicações no modelo CCA está relacionado à interoperabilidade de modelos de computação paralela ou distribuída (ver exemplos na Subseção 4.2.2). Por exemplo, supondo uma situação na qual um conjunto de componentes que residem nos nós de um *cluster* se comunicam entre si usando uma plataforma leve de passagem de mensagens durante a execução de um fragmento paralelo de uma simulação. As conexões entre esses componentes residindo no *cluster* devem implementar um determinado algoritmo de comunicação global, síncrono. Alguns destes componentes se comunicam assincronamente com outros componentes residentes em locais fora do *cluster* (para visualização de dados ou atualização de dados da simulação, por exemplo) por meio de uma invocação de método remoto [115]. As conexões neste caso devem implementar um outro modelo de comunicação, possivelmente sobre outra plataforma de troca de mensagens. Demandas semelhantes de integração de modelos aparecem em aplicações de otimização combinatória, conforme tratado no Capítulo 6.

A implementação da indireção que torna as comunicações "legais" é baseada no padrão MPI e é feita pela composição de um ou mais componentes CCA de um tipo especial implementado pela *interface* `Connector`, a qual estende a interface `Component`.

Os principais elementos da expansão do modelo de conexão do CCA (ver Figura 4.2) que propomos nesta tese são:

- Configuração
  - `Link`
  - ***Duto*** ou `Duct`
  - `ConnectingSpace`
- Conexão
  - `Conector`

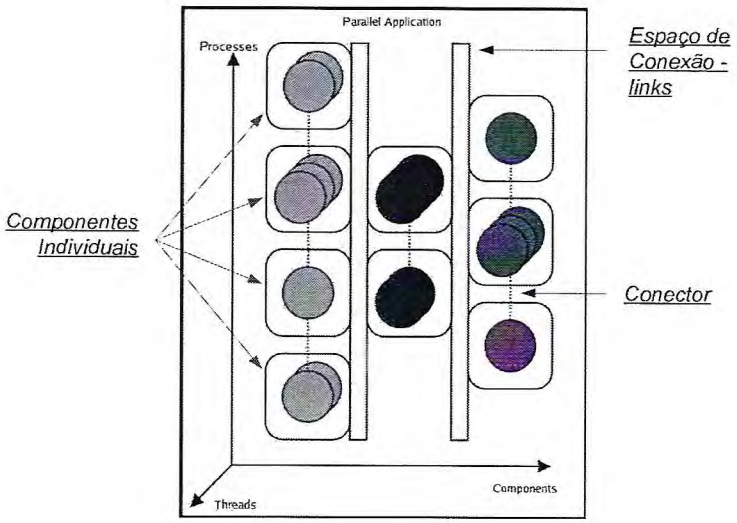


Figura 4.2: Elementos do novo modelo de conexão do *Forró*.

O elemento essencial do modelo de conexão do *Forró*, a plataforma CCA que implementamos neste trabalho, é a especificação de conectores. Conforme explicado no Capítulo 3, conector é uma unidade que intermedia as interações entre dois ou mais componentes. Além disso no *Forró*, todo conector inclui um ***duto***, que é usado para suportar o fluxo de dados e controle entre componentes [63, 64].

O conceito ***duto*** adotado neste trabalho foi:

Duto: elemento que dá suporte aos fluxos de comunicação envolvidos na conexão, sejam eles de dados ou controle. Um ***duto*** deve implementar mecanismos de envio e recepção de mensagens, incluindo mecanismos de comunicação coletiva. Pelo imperativo de alto desempenho, sua *interface* é naturalmente inspirada no MPI (embora sua implementação possa ser realizada sobre qualquer plataforma de comunicação).

O tipo `Connector` é parametrizado por um ***duto***. Também definimos uma porta especial, chamada `Link`, para permitir o acesso às funcionalidades do ***duto*** de um conector. A *interface* `Connector` inclui métodos para configurar o seu ***duto*** e criar ***links***. O ***Connecting Space*** (ou ***espaço de conexão***) consiste no conjunto de conectores indiretos que compartilham uma infraestrutura de comunicação e funcionalidades globais de conexão (mais detalhes na Seção 5.2).

Dentro da noção de conectores mostrada no Capítulo 3, os conceitos conector direto e conector indireto (ou em inglês *linked*) são:

- **Direto**: invocação de método direto na porta *provides*, ou seja, atribuição direta de porta *provides* à porta *uses*.

- ***Indireto*** ou ***Linked***: através de um *link*, que requer um componente porta *uses*, ou seja, um conectore traduz CCA (chamada de método) para MPI (*links*), gerando comunicação.

Estes tipos de conectores foram projetados para refletir a ideia na qual os conectores são destinados a encapsular interação ou comunicação, enquanto componentes são significativos para encapsular computação. Existem duas variações principais de modelos de componentes que respeitam este princípio. A primeira variação é discutida na Seção 4.2 e a segunda variação é discutida na Seção 4.3.

As características que caracterizam conectores são suas *interfaces*, tipos, a semântica, as restrições, evolução e propriedades não-funcionais [116]. Baseado nestas características, escolhemos para uso pioneiro em uma plataforma CCA, dois tipos de conectores - conectores endógenos e exógenos [24]. Estes conectores encapsulam a interação entre os componentes de alto nível dos modelos de componentes. Além dos conectores, introduzimos o uso do conceito ***duto*** no modelo CCA.

4.2 Conectores Endógenos no CCA

Segundo a definição de conector endógeno apresentada no Capítulo 3 e ilustrada na Figura 4.3, seu emprego está associado a situações em que componentes encapsulam juntos computação e controle, ao mesmo tempo que executam sua computação também iniciam chamadas a métodos da porta e gerenciam seus retornos. Ajudando assim, na modularidade dos componentes. Assim sendo, conectores endógenos se enquadram naturalmente no modelo CCA, ou seja, são utilizados como conectores responsáveis pela conexão gerenciando as interações entre uma porta *uses* de um componente a uma porta *provides* de um (mesmo ou distinto) componente. Este tipo de conector pode ser direto ou indireto. Em ambos os casos, o conector não gera invocações de métodos de porta por si só.

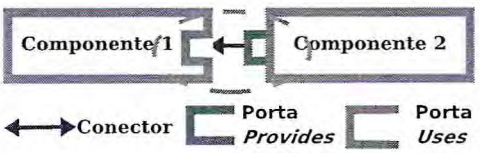


Figura 4.3: Ilustração de um conector endógeno.

4.2.1 Conectores Endógenos - Conexão Direta

Até mesmo modelos de componentes que não utilizam conectores de forma explícita muitas vezes têm operadores de composição que podem ser interpretados como conectores em diferentes níveis de abstração. Por exemplo, chamadas diretas de método entre os componentes podem ser consideradas como conectores no nível de código. Por conseguinte, podemos considerar como uso de um conector endógeno em uma conexão direta: uma conexão cujo ***duto*** é o procedimento de chamada produzido pelo compilador.

A Figura 4.4 ilustra uma conexão direta entre dois componentes CCA. representando uma conexão direta usando um conector endógeno. De uma forma geral, podemos considerar uma conexão simples entre dois componentes como um conector endógeno de forma direta.

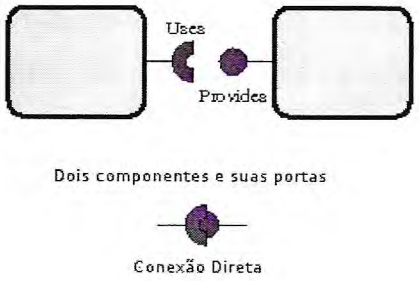


Figura 4.4: Uma conexão direta entre dois componentes CCA.

Para isto, supomos que o componente que contém a porta *uses* é também um conector responsável pela conexão gerenciando as interações, no qual o ***duto*** é o procedimento de chamada produzido pelo compilador, entre a própria porta *uses* e uma porta *provides* do mesmo ou de outro componente. As interações ocorrem da seguinte forma: o componente/conector endógeno contendo a porta *uses* invoca um método de uma determinada porta *provides* para transmitir uma invocação originada em uma porta *uses* de um componente.

4.2.2 Conectores Endógenos - Conexão Indireta

Uma conexão indireta por meio de conectores endógenos se distingue da conexão direta pelo fato de a conexão ser realizada com o uso de indireções em ambos os extremos. De uma forma geral, o lado da porta *provides*, um componente `ProvidesPortIndirection` é responsável por essa indireção, enquanto do lado da porta *uses*, a indireção fica a cargo do componente `UsesPortIndirection`. Em ambos os casos, a indireção consiste na tradução entre o tipo de porta sendo conectado com uma porta do tipo `Link` do conector na forma ilustrada na Figura 4.5. Em outras palavras, entre os componentes `ProvidesPortIndirection` e `UsesPortIndirection` e os conectores

endógenos a conexão é feita por portas do tipo `Link`. Os componentes responsáveis pelas indireções são dispensáveis em alguns casos. Ressalte-se o estabelecimento de conexões diretas para a implementação de uma conexão indireta. Outro aspecto neste tipo de conexão é a possibilidade de execução *multithread*, ou seja, um conector pode executar diferentes *threads* simultaneamente. Portanto, contendo diferentes *threads*, um conector endógeno em conexão indireta pode adicionar concorrência à execução.

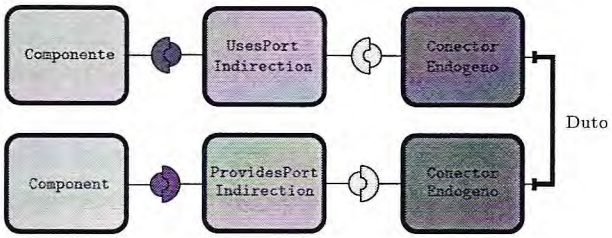


Figura 4.5: Uma conexão indireta através de conectores endógenos e ***links*** especializados.

Algumas observações podem ser feitas com respeito a esta solução. Primeiro, presume-se que a porta do tipo `Link` utilizada seja especializada em comunicação por passagem de mensagens, o que significa que esta porta inclui métodos com correspondência um-para-um com funções de comunicação. A segunda observação é que um conector endógeno em conexão indireta conecta uma porta *uses* com uma outra porta *provides* do mesmo ou de outro componente.

Dada as observações, a intervenção de um conector endógeno em conexão indireta ocorre da seguinte forma: quando uma mensagem é enviada de um componente para outro, o remetente invoca o método de envio de uma porta `Link`, enquanto que o receptor deve invocar o método de recebimento de outra porta `Link`. O desempenho de uma conexão indireta depende da implementação das indireções. O papel do componente `ProvidesPortIndirection` é continuamente invocar o método de recebimento na porta `Link`. Quando esse método retornar, uma invocação do método correspondente é produzida.

Além das características citadas, um conector endógeno em uma conexão indireta e os conectores exógenos (será mostrado na Seção 4.3) possibilitam a representação de diversos modelos de computação. Isto acontece na conexão indireta porque os conectores podem implementar modelos de computação. é importante observar que um conector implementado como indireção pode interceptar as mensagens trafegando pelas portas `Link` e incluir computação sobre elas. Isso permite, inclusive, garantir certas propriedades globais a essas comunicações. Finalmente, é importante ressaltar que nos conectores endógenos em conexão indireta pode ser incluído alguns tipos de controle (algumas características de sincronização global, etc). Esse fato é explorado no Capítulo 5.

Como exemplo da característica acima descrita, vamos mostrar o modelo de computação dirigido a eventos descrito formalmente em [117]. Este modelo de computação dirigido a eventos é usado para projetar e analisar algoritmos paralelos em

sistemas com comunicação baseada em passagem de mensagens.

O formalismo deste modelo de computação pressupõe que a computação é realizada em um sistema composto por um conjunto de nós, interligados por um conjunto de canais de comunicação bidirecional ponto-a-ponto. Neste sistema, um nó é capaz de executar computações sequencialmente e interagir com os vizinhos somente por envio ou recepção de mensagens através de canais incidentes ao mesmo. Não há chamada de métodos nesse modelo. Por esse motivo, a sua implementação no modelo CCA ilustra bem o poder de expressividade dos conectores.

Neste modelo, a computação distribuída realizada é completamente descrita por um estado inicial global (incluindo um estado inicial para cada nó e nenhuma mensagem em trânsito), as computações locais realizadas por cada nó, e as interações entre os nós.

Por outro lado, as computações locais em um nó formam uma sequência de eventos, cujo primeiro membro pode ser espontâneo (o que significa uma computação local que não depende da recepção de qualquer mensagem). Cada evento não-espontâneo desencadeia uma reação de um nó quando o mesmo recebe uma mensagem. Cada evento desencadeia uma reação diferente. Mais precisamente, o recebimento de uma mensagem afeta o estado local de um nó através da execução do método `event()`, que encapsula as ações de uma computação em particular associada a este nó. Esse método leva a mensagem como entrada. Além de alterar o estado local deste nó, a execução do `event()` produz como resultado um conjunto de mensagens (possivelmente vazias) que são enviadas (se houver), com o objetivo de induzir eventos em outros nós. Cada nó é encapsulado em um componente e os canais são implementadas com conectores, conforme mostra a Figura 4.6. Os eventos ocorrem de maneira assíncrona e são tratados pelo sistema a medida que ocorrem.

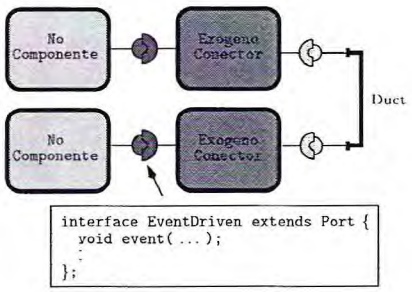


Figura 4.6: Conector indireto através de conectores endógenos implementando um canal de uma aplicação orientada a eventos.

4.3 Conectores Exógenos no CCA

A segunda variação determina que os conectores são a origem de invocações dos métodos de uma porta. A consequência mais importante disto é que o controle é feito por

conectores, que são necessariamente indiretos. A segunda variação é denominada conector exógeno (introduzida no Capítulo 3). Basicamente, um conector exógeno se distingue do conector endógeno devido às invocações de métodos de uma porta serem originadas internamente no conector, em contraste aos conectores endógenos, onde invocações de métodos de uma porta são originadas externamente em algum componente. Este tipo de conector pode invocar outro conector no caso de conexão indireta. Note-se que a idéia de conector exógeno pode parecer, à primeira vista, incompatível com o modelo CCA, porque o modelo CCA não foi concebido para permitir conexões entre portas *provides*. Para conexões, o modelo CCA não estabelece detalhes de implementação nem características, apenas estabelece uma *interface* sem métodos e nem atributos. Devido a este contexto aberto, propomos uma implementação usando componentes de um tipo especial chamado de conector. Como um conector é também um componente, então, uma conexão entre duas portas *provides* se dá através de conexões entre portas *uses* e portas portas *provides*. Portanto, não desobedecendo o modelo CCA.

Além das características acima, um conector exógeno, assim como um conector endógeno em uma conexão indireta, possibilita a representação de diversos modelos de computação.

A Figura 4.7 mostra um exemplo de um conector exógeno. O conector denominado `Conector` chama um método `metodo1` em um componente `Componente1` e o método `metodo2` do componente `Componente2`, mas não interagem diretamente com os outros. As chamadas de métodos podem ser acompanhadas por fluxo de dados entre `Componente1` e `Conector` ou entre `Componente2` e `Conector`, mas novamente não entre `Componente1` e `Componente2` diretamente.

A principal diferença entre um conector e um componente é que o primeiro destina-se à composição e o segundo destina-se à computação. Porém, existem componentes conectores que executam o papel de conectores.

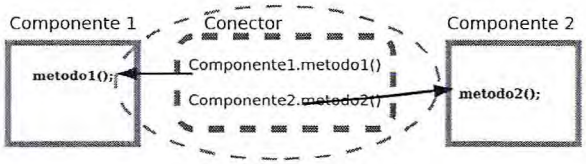


Figura 4.7: Componentes conectados por um conector exógeno.

Nossa motivação principal para a utilização de conectores exógenos é encapsular o controle no modelo de componentes CCA, com o objetivo de especificar mais detalhadamente a conexão entre componentes CCA. No CCA, os componentes encapsulam dados e funções, e são, portanto, flexíveis. Usando conectores exógenos para encapsular o controle esperamos melhorar a comunicação entre componentes do CCA. Os conectores exógenos objetivam conectar portas *provides* entre si (ao contrário do conector endógeno, que conecta uma porta *uses* a uma porta *provides* em uma conexão

direta ou conecta uma porta *provides* a uma porta *provides* em uma conexão indireta). Ao lado disso, introduzimos o uso de conectores exógenos no CCA como operadores de composição.

Os conectores exógenos no CCA servem de gatilho para dar início a uma execução de um componente. Por exemplo, uma porta de um conector exógeno pode fazer o componente simulador iniciar o cálculo dos resultados de outro componente.

Para ilustrar a representação de um modelo de computação por um conector exógeno no CCA, mostramos a seguir um exemplo de uma aplicação no modelo de computação baseada em pulsos que utiliza conectores exógenos.

O modelo de computação baseada em pulsos [41] é complementar à computação baseada em eventos e é motivado por aplicações nas quais o estado inicial evolui para um estado final em fases, como em muitos algoritmos iterativos e não é síncrono.

A evolução em fases no modelo de computação baseada em pulsos faz com que esta abordagem baseada em pulsos seja adequada para descrever vários algoritmos numéricos paralelos. Para modelar tal comportamento, um mecanismo que gera uma sequência de pulsos deve ser fornecido para gerenciar a evolução em cada nó. Cada pulso produzido por tal mecanismo inclui alguma computação em cada nó e produz uma transição de estados neste nó. Um conjunto de eventos está associado a cada pulso, pelo menos um evento por nó.

Por razões técnicas, um pulso pode incluir eventos vazios, ou seja, eventos que não possuem nem entrada ou saída. e que não executam ações. Os pulsos são indexados sequencialmente, e um evento é associado a um pulso indexado $l+1$, se o evento ocorrer entre os pulsos l e $l+1$.

Se dois eventos ocorrem de tal forma que há uma sequência de invocações de método principal de um evento para outro, então o evento que está na origem desta sequência é associado a um pulso menor do que aquele associado com o destino da sequência. A descrição da implementação CCA abaixo ajuda a compreender melhor o funcionamento deste modelo (ver mais detalhes em [118]).

As principais diferenças entre aplicações baseadas em eventos e aplicações baseadas em pulsos é o que controla as transições de estado local dos nós. No caso das aplicações baseadas em pulsos, o controle da computação é uma responsabilidade do mecanismo de geração de pulso. Esta é a razão pela qual conectores exógenos são empregados na forma mostrada na Figura 4.8.

Para que um conector exógeno conecte duas portas *provides*, a porta *uses* de cada um destes conectores é uma extensão de uma porta *uses* de conectores baseados em eventos, incluindo o método `pulse`, usado pelo conector para gerar um pulso no componente `node`.

4.4 Considerações Finais

Utilizar conectores endógenos para construir e combinar componentes é a nossa sugestão de melhoria para encapsulamento e composição. Componentes compostos construídos

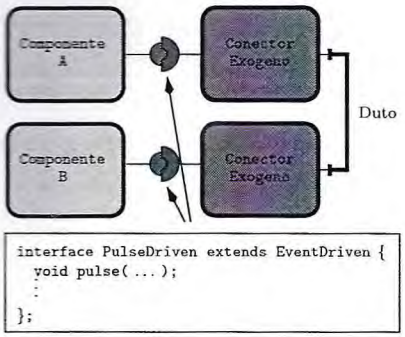


Figura 4.8: Conectores exógenos e conector indireto através de conectores endógenos implementando um canal de uma aplicação baseada em pulsos.

por conectores endógenos e exógenos fazem uma abordagem de composição para possível construção de um sistema. Um benefício da conexão exógena é que os componentes são fracamente acoplados, pois o controle é originado e encapsulado por conectores, ao contrário de conectores ADL (do inglês *Architecture Description Language*) que não originam ou encapsulam controle. Um resultado imediato do uso destes conectores é que os sistemas se tornam mais modulares e, portanto, mais fáceis para manter e reconfigurar.

Capítulo 5

Plataforma *Forró*

Este capítulo apresenta o *Forró*, uma plataforma de componentes baseada no modelo CCA para a construção de aplicações CAD. Como visto no Capítulo 3, segundo [103], uma plataforma de componentes é definida por um modelo de componentes, um modelo de conexão e um modelo de implantação. Conforme citado, tomamos o modelo CCA apresentado no Capítulo 3 como o modelo de componentes da plataforma *Forró*. Conforme mencionado no capítulo anterior, para o presente trabalho, escolhemos o modelo de componentes CCA e não o # porque tínhamos o objetivo de englobar também aplicações distribuídas e o # é mais adequado a aplicações paralelas. Nesse sentido, o *Forró* é uma plataforma CCA e, no decorrer do capítulo, descrevemos, entre outros aspectos, a implantação do modelo de conexão adotado no *Forró*. Nessa descrição, detalhamos a nossa proposta de ***espaço de conexão***, conceito criado para dar sustentação à integração de diferentes modelos de programação paralela ou distribuída. Na Seção 5.1 descrevemos uma visão geral da plataforma *Forró*. Na Seção 5.2, mostramos a arquitetura da plataforma *Forró* e dos componentes *Forró*. Após isso, apresentamos o modelo de conexão do *Forró*, detalhando a contribuição do presente trabalho descrevendo as inovações propostas para o modelo CCA, as quais estão relacionadas à inclusão da noção de conectores, e mostramos o modelo de implantação do *Forró*. Em seguida, descrevemos a implementação e o funcionamento da plataforma na Seção 5.4.

5.1 Visão Geral da Plataforma *Forró*

O detalhamento da implementação da plataforma *Forró* é o assunto das próximas seções. Antes desse detalhamento, porém, nós destacamos algumas de suas propriedades, como por exemplo, as inovações face a outras plataformas CCA existentes.

- *Implementação em Java.* Essa escolha justifica-se, sobretudo, por duas das características dessa linguagem: a adequação ao processamento distribuído e produtividade no desenvolvimento. Naturalmente, pode-se questionar o seu impacto no desempenho, por tratar-se de uma linguagem interpretada. Uma série de

elementos da implementação reduzem a sobrecarga devido ao uso da linguagem Java a níveis aceitáveis. Para uma análise sobre essa sobrecarga, é preciso, em primeiro lugar, identificar as suas potenciais fontes. Elas intervêm em três momentos: na etapa de especificação de configuração (criação de componentes e estabelecimento das conexões); na execução da computação em cada componente; e na operação das conexões. No primeiro caso, embora a sobrecarga devido ao Java possa ser grande, ela não é significativa, pois a restrição de alto desempenho se aplica à execução da aplicação somente. Neste primeiro caso, a diferença de tempo em que as tarefa terminam e o momento no qual elas são iniciadas (em inglês conhecido como *wall clock time*), não são aumentadas no momento de configuração. Nos outros dois casos, a sobrecarga pode ser evitada quase que inteiramente durante a execução da aplicação, *wall clock time* é aumentado apenas com invocação de métodos. Quanto à computação de cada componente, ela pode ser realizada fora da máquina virtual do Java, portanto livre da sua sobrecarga. Para a operação das conexões, a implementação prevê um mecanismo através do qual, apesar de todo componente ser definido como um componente Java, a plataforma permite a criação de uma imagem em linguagem nativa desse componente e de suas portas.

- *Configuração simultânea de diversas aplicações.* As aplicações são identificadas, permitindo que a plataforma possa distinguir as solicitações de criação de componentes e estabelecimento de conexões provenientes das diferentes aplicações. Somente um ambiente *Forró* precisa estar ativo em cada unidade computacional, mesmo que este seja simultaneamente usado por diversas aplicações. As suas execuções não interferem entre si.

- *Comunicações ponto-a-ponto.* As conexões indiretas do modelo CCA, baseadas em invocações de métodos, são implementadas sobre um mecanismo de comunicação ponto-a-ponto. Esse mecanismo é construído em três camadas, permitindo uma vasta gama de combinações de propriedades globais e locais das conexões. As *interfaces* dessas camadas são compatíveis com MPI.

- *Conexões eficientes.* Como é habitual nas plataformas CCA, *Forró* é capaz de efetuar conexões diretas de baixa sobrecarga. Além disso, o *Forró* já inclui uma implementação de ***duto*** MPI, permitindo conexões indiretas através desse ***duto*** em linguagem nativa, sem criar novas indireções e com baixa sobrecarga. Algumas *interfaces* são acrescentadas ao modelo CCA para manipular conectores, segundo modelo de conexão apresentado no Capítulo 4. São elas `Link` e `Duct`, cujas implementações conjuntas permitem a integração de diferentes modelos de computação. Conforme sugerido pelo modelo de conexão do Capítulo 4, essas *interfaces* incluem métodos de comunicação coletiva compatíveis com a *interface* MPI.

- *Aplicações paralelas ou distribuídas.* Com o uso de conexões indiretas previstas no modelo de conexão apresentado no Capítulo 4, aplicações paralelas, baseadas

em memória compartilhada ou distribuída, podem ser configuradas. Porém, uma instância de um componente da aplicação executa em um único local, como mencionado no Capítulo 4. Portanto, não existe o conceito de componente composto.

- *Criação de* ***espaços de conexão***. Conceitualmente, um ***espaço de conexão*** é definido por um certo conjunto de conectores, parametrizados pela mesma implementação de ***dutos***. As conexões realizadas pelos conectores desse ***espaço de conexão*** garantem uma série de propriedades globais. Um ***espaço de conexão*** pode ser usado, por exemplo, para implementar um certo modelo de computação. Uma mesma aplicação pode conter diferentes espaços de conexão.

Na Figura 5.1 é apresentada uma visão geral da plataforma *Forró*. Conforme se observa, a estrutura do ambiente *Forró* está organizada em camadas de *interface*, controle e domínio, seguindo o padrão arquitetural MVC [119].

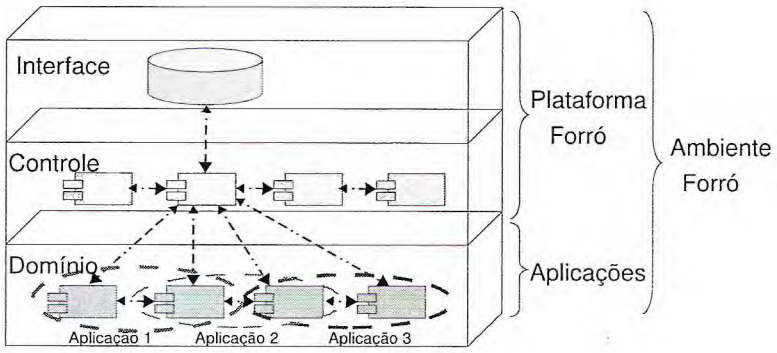


Figura 5.1: Plataforma *Forró*.

Na camada de *interface*, o usuário dispõe de comandos para configurar a plataforma, criar/destruir componentes, conectar/desconectar componentes e iniciar aplicações. Na camada de controle, os componentes próprios do *Forró* (veremos com mais detalhes nas próximas seções) gerenciam e configuram as aplicações pela conexão de componentes usando o modelo CCA, conforme citado. Na camada de domínio, se localizam as instâncias de componentes de uma ou mais aplicações.

Segundo o modelo de conexão do Capítulo 4, vale ressaltar uma característica importante, que é a compatibilidade desse modelo de conexão com o CCA, a qual decorre, essencialmente, do fato de que todos conectores do modelo serem implantados como componentes CCA. Além de permitir o reuso e a interoperabilidade com outras plataformas compatíveis com o CCA, um ponto importante que não pode ser perdido de vista é o desempenho. Nesse sentido, o modelo de conexão do *Forró* permite limitar a sobrecarga imposta pela plataforma na mesma medida que qualquer outra plataforma CCA. Teremos a oportunidade de retornar a esse ponto ainda neste capítulo.

Usualmente, conectores podem trabalhar em conjunto com o propósito de combinar fluxos de dados ou controle de maneira a permitir interações mais ricas e complexas entre componentes, tipicamente garantindo certas propriedades locais ou globais ao padrão de comunicação (tais como sincronismo e balanceamento de carga). Para atingir tal meta, um conector típico incorpora em suas atribuições algum processamento ou armazenamento de informações. Por exemplo, um ***espaço de conexão*** pode garantir balanceamento de carga através de um algoritmo de controle de tráfego entre os componentes baseado em algum conhecimento sobre o estado desses componentes. Essas são noções que ficarão mais claras à medida que avançarmos na exposição dos conectores previstos no modelo.

5.2 Arquitetura da Plataforma *Forró*

A arquitetura de alto nível da plataforma *Forró* é mostrada na Figura 5.2, a qual explica a composição da plataforma pela implementação do *Forró* com núcleo CCA, *Application Programming Interface* (API) para comunicação com o usuário, componentes *Forró* e código nativo.

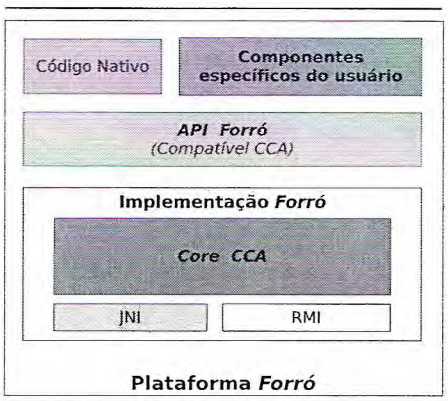


Figura 5.2: Arquitetura da Plataforma *Forró*.

Relacionando a Figura 5.2 com o padrão *Model-View-Controller* (MVC) citado anteriormente, podemos dizer que:

- A API *Forró*, a qual é baseada nos comandos CCA, representa a camada de *interface* com usuário. Essa API foi inspirada nos comandos do CCaffeine;
- Implementação *Forró*, a qual tem como núcleo as *interfaces* do modelo CCA, representa a camada de controle;

- Componentes *Forró*, que encapsulam os componentes do usuário, e Código Nativo, que encapsula o código legado do usuário com o qual se deseja interagir, representam a camada de domínio.

Na Figura 5.3 é mostrada a arquitetura de um componente *Forró*. A figura citada mostra que a comunicação entre o componente *Forró* e a plataforma é feita através das funcionalidades disponibilizadas pela especificação do CCA. Também ilustra que a comunicação entre componentes *Forró* é feita através das portas *uses* e *provides*. Além disso, mostra que para enviar e/ou receber dados de códigos nativos, o componente usa o *Java Native Interface* (JNI).

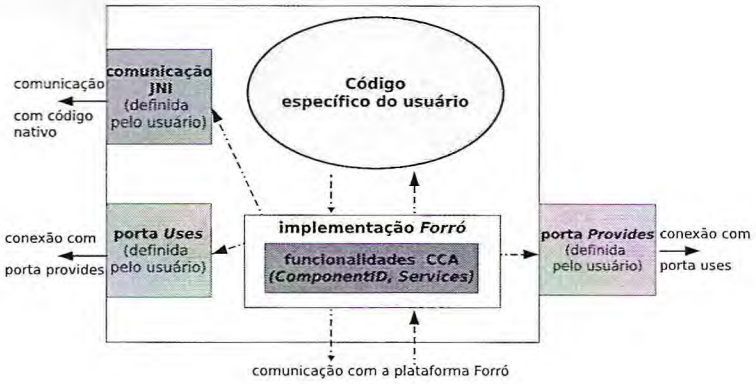


Figura 5.3: Arquitetura de um componente *Forró*.

5.2.1 Elementos da Plataforma

Em uma visão mais detalhada da composição da plataforma *Forró* (ver Figura 5.4), podemos dizer que a plataforma é composta por uma API de comunicação com o usuário, elementos de comunicação/configuração e componentes gerenciadores. A API é constituída por um conjunto de classes chamado *Frontend*. É importante ressaltar que não está implementada a capacidade para descoberta de recursos.

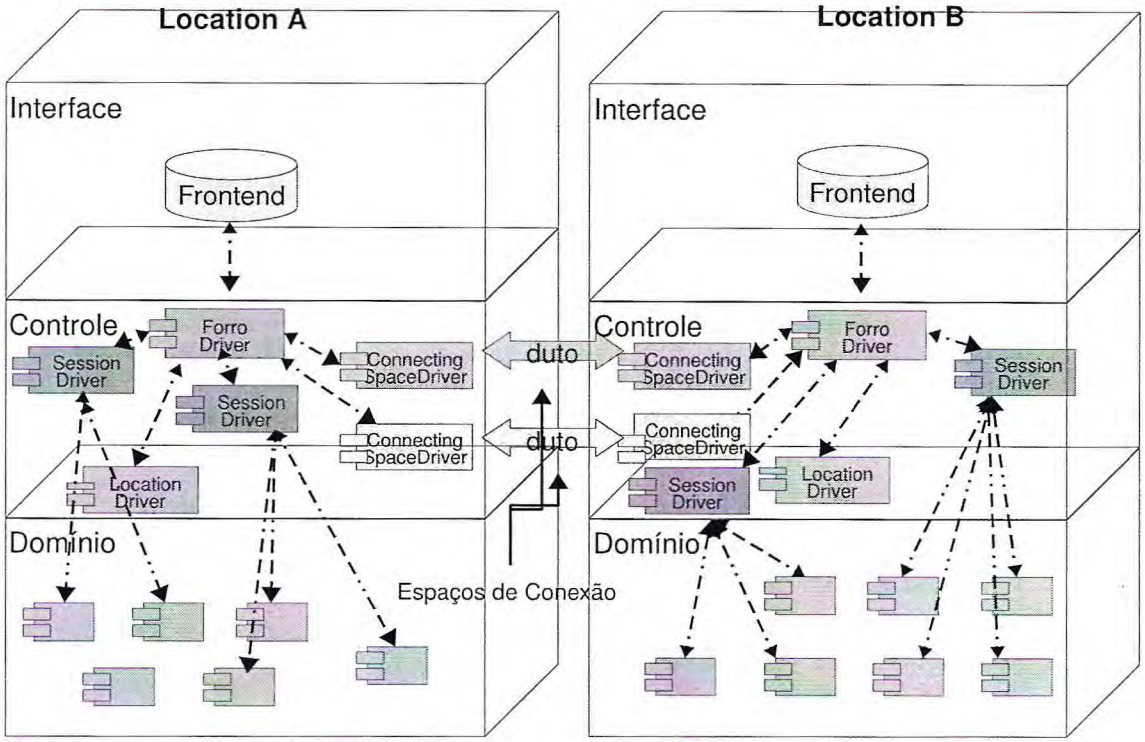


Figura 5.4: Elementos da plataforma *Forró*.

Os elementos de configuração são:

- `Link`:
  - Conecta pares ***locations*** em um mesmo ***Connecting Space***, referidas como pontos extremos do ***link***, permitindo que as portas dos componentes das ***locations*** se comuniquem de acordo com um padrão específico. Um ***link*** define o padrão de comunicação entre as ***locations*** que une.
  - Um tipo `Link` é parametrizado por um `Duct`, o qual ele usa para realizar as comunicações. Um `Link` pode incluir algum processamento, o que permite implementar propriedades globais ao conjunto de `Links` de um mesmo ***Connecting Space***. Essas propriedades globais são independentes do `Duct`.
  - Possui as mesmas características do *group* do MPI.
  - Provê métodos de comunicação ponto-aponto e coletiva.
- ***Duct*** (ou ***duto***)
  - Elemento que dá suporte aos fluxos de comunicação de um ***link***, seja ele de dados ou controle.

- Mais concretamente, um elemento `Duct` (implementação do ***duct***) no *Forró* é uma infraestrutura de comunicação que implementa a interação entre dois extremos, fornecendo serviços de baixo nível para segurança, autenticação, entre outros.
- Não há garantia de nenhuma propriedade global entre ***ducts*** diferentes além das especificadas pelos serviços oferecidos pela *interface* `Duct`.
- Possui as mesmas características do *communicator* do MPI.
- Permite a criação de *links*.

- ***Connecting Space***
  - Conjunto de conectores indiretos que compartilham uma infraestrutura de comunicação e funcionalidades globais de conexão.
  - Mais concretamente, ***espaço de conexão*** é um conjunto de ***locations*** com espaços de endereços físicos disjuntos e um conjunto de ***links*** que conectam pares de ***locations***.
  - Cada ***espaço de conexão*** estabelece a quantidade de pontos extremos que cada um dos seus ***links*** pode possuir.
  - Todas as conexões realizadas através dos ***links*** parametrizados de um ***Connecting Space*** possuem as propriedades globais estabelecidas pelo mesmo.
  - É composto por ***dutos***.
  - Possui propriedades coletivas de comunicação.
  - É o mais alto nível de abstração de comunicações coletivas ou puramente ponto-a-ponto.
  - Ordenação de entrega de mensagens (FIFO, ordem causal, etc).
  - Comunicação coletiva e roteamento.
  - Registro de estado global.
  - Questões de sincronização.

Os elementos de implementação são:

- ***Location***: recurso computacional (máquina ou processador, por exemplo) com habilidade de executar alguma computação, ou seja, capaz de executar componentes, possivelmente, concorrentemente.
- ***Session*** (ou ***sessão***): instância de uma aplicação, a qual pode conter um conjunto de instâncias de componentes que foram criados e associados à mesma. Toda aplicação é identificada por sua ***session***, permitindo à plataforma gerenciar diferentes aplicações simultâneas sem interferência entre elas.

Novas *interfaces* acrescentadas ao CCA:

`Link`: porta contendo métodos de comunicação ponto-a-ponto, formando um subconjunto das funções estabelecidas no MPI. Dentre esses métodos, estão os métodos de envio e recepção de mensagem, assim como de comunicação coletiva. Esta *interface* é uma extensão de `Port`. Uma implementação de `Link` pode acrescentar propriedades globais aos métodos de comunicação.

`Duct`: extensão de `Link`, incluindo métodos adicionais para registro e inicialização. Observe que um `Duct` é um caso especial de `Link`, fato que pode ser aproveitado para eliminar uma indireção em uma conexão indireta. Para eliminar indireção em conector indireto, não se pode usar conector direto porque um conector direto somente pode ser utilizado entre componentes na mesma máquina.

`Connector`: componente que deve conter uma porta *provides* do tipo `Link`. Sendo um componente, esta *interface* estende `Component`.

Os componentes gerenciadores são:

- `ForroDriver`: principal componente gerenciador da plataforma *Forró*. O `ForroDriver` controla todas as sessões que se executam em uma `location`. `ForroDriver` controla a comunicação de várias aplicações, pois cada aplicação está associada a uma sessão. Existe um `ForroDriver` para cada `location` (recurso computacional). O `ForroDriver` recebe todas as chamadas de configuração e repassa para outro `ForroDriver` caso seja em outra `location`. Na momento da execução das aplicações, o `ForroDriver` não interfere, logo, em tempo de execução é descentralizado.

- `SessionDriver`: componente gerenciador de uma única ***sessão***. Este componente controla as instâncias de componentes criados na ***sessão*** especificada.

- `ConnectingSpaceFactory`: componente gerenciador das conexões em um determinado modelo de computação. Este componente descreve um modelo da computação empregado quando `locations` que pertençam ao mesmo interagem através dos `links` que as conectam.

- `LocationDriver`: componente gerenciador de uma determinada ***location***.

Os elementos acima formam um cenário sobre o qual os componentes de uma aplicação podem ser criados e conectados entre si. Uma instância de um componente é criada em uma ***location*** específica e em uma determinada ***sessão***. Uma ***location***, por sua vez, pode pertencer a um ou mais ***Connecting Space***.

5.3 Modelo de Implantação do *Forró*

5.3.1 Componentes e Portas

Conforme o modelo CCA, os componentes *Forró* interagem através das conexões de suas portas. Uma porta pode ser implementada de diferentes maneiras, dependendo da conexão que se deseje realizar. Uma das possibilidades de implementação de uma porta é usando um componente CCA. Detalhes sobre esta possibilidade são dados mais adiante. Por enquanto, basta-nos mencionar que o *Forró* mantém tabelas de componentes e de suas portas. Seguindo a especificação CCA, a tabela de componentes é criada e gerenciada conforme esses componentes são criados. Por outro lado, as portas dos componentes devem ser informadas pelos próprios componentes à plataforma.

Qualquer componente que necessitar dos métodos disponibilizados registram a necessidade através de portas *uses*. No momento da execução, os componentes que necessitam de uma porta *uses* (componentes "usuários") invocam os métodos dos componentes "provedores" através das portas *uses* declaradas. Componentes *Forró* tem a capacidade de obter informações na lista de métodos que são suportados por qualquer outro componente *Forró*.

No *Forró*, a manipulação das portas *uses* e *provides* dos componentes foi implementada via tabelas no objeto `Services`. As adições e as exclusões das portas para os componentes são feitas diretamente nestas tabelas.

5.3.2 Conexões com Conectores Diretos

O modelo de implantação de conexões distingue as duas categorias de conectores previstos no modelo de conexão. Primeiramente, vejamos como é implantada uma conexão com conectores diretos.

Uma conexão é estabelecida pela obtenção de uma porta *provides* da tabela de portas associadas ao componente. Com o intuito de produzir uma conexão que não incorra na necessidade de indireções nas chamadas de método, a conexão deve realizar uma atribuição direta da porta *provides* à porta *uses*. Esse mecanismo é possível graças à especificação do CCA, que determina que um componente requisite à plataforma a porta *provides* conectada a sua porta *uses*. Em resposta a uma tal requisição, a plataforma deve responder com a porta que o próprio componente atribuirá à porta *uses*. Dessa forma, basta que a plataforma guarde em sua tabela de portas, as portas *provides* informadas pelos componentes. A tabela de mapeamento é guardada na sessão e não no componente. A cada requisição de um componente pela porta conectada a uma porta *uses*, a plataforma pode responder com a própria *provides*. O processo de conexão indireta, detalhada a seguir, ilustra como, em certos casos, esse procedimento não pode ser utilizado.

5.3.3 Conexões com Conectores Indiretos

Para esclarecer todas as formas de conectores indiretos, e suas interações, começamos detalhando a forma de um conector indireto. Na Figura 5.5, os elementos envolvidos em uma conexão no *Forró* aparecem de acordo com o papel desempenhado. A conexão indireta no *Forró* é feita por conectores endógenos.

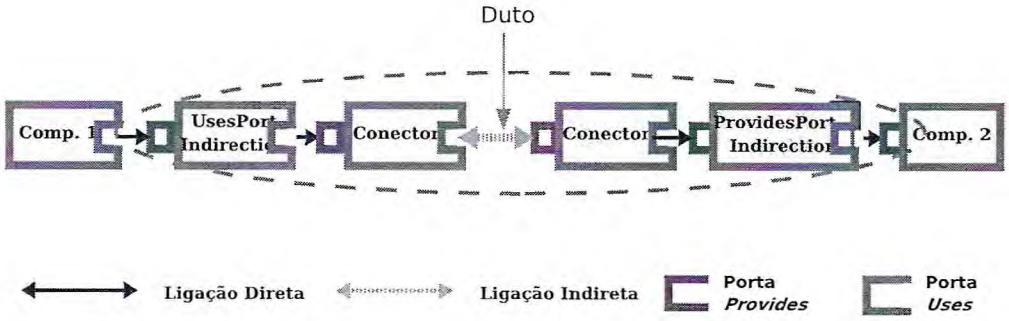


Figura 5.5: Um conector indireto típico no *Forró* formado por quatro componentes.

Na Figura 5.5, identificamos os quatro componentes envolvidos em um conector indireto. Dois desses componentes são do tipo `Connector`, o que os identifica como componentes que podem se ligar via um ***duto***. Além destes, há dois outros componentes, cada um responsável pela interação com uma porta em uma conexão. O componente `UsesPortIndirection` é conectado via conector direto, com uma porta *uses*. Pela descrição mencionada acima sobre conectores diretos, o componente `UsesPortIndirection` é membro da tabela de portas manipulada pela plataforma, fazendo o papel de porta *provides*.

Do outro lado do conector indireto, encontra-se o `ProvidesPortIndirection`, componente possuindo uma porta *uses* que deve ser conectada, novamente por um conector direto, ê porta *provides* de um componente.

5.4 Implementação e Funcionamento da Plataforma *Forró*

5.4.1 Implementação dos Elementos do Forró

O *Forró* foi implementado usando a linguagem de programação Java, com uma única máquina virtual por ***location***. As ***locations*** são criadas pelo usuário através de linha de comando no ambiente *Forró*. A escolha de Java como linguagem de programação foi devida às seguintes características: multi-plataforma, implementação rápida e vários códigos legados existentes.

Basicamente, para ser compatível com o CCA é necessário implementar as *interfaces* definidas pelo padrão CCA.

Todas as *interfaces* do modelo CCA são implementadas em Java (ver Figura 5.6) da seguinte forma:

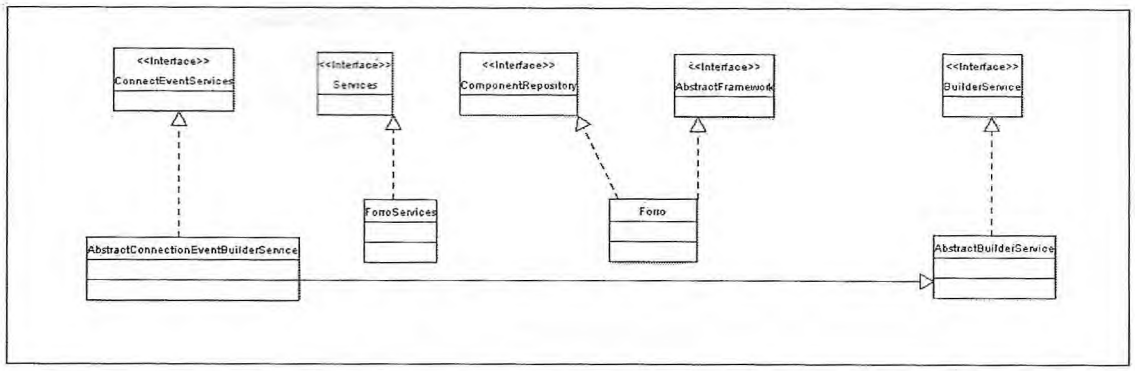


Figura 5.6: Diagrama de classes *Forró* e *interfaces* CCA.

BuilderService: cria instâncias de componentes e conecta portas. A *interface* CCA `BuilderService` é implementada pela classe `ForroSessionDriver`. `ForroSessionDriver` é responsável pela criação de componentes e conexão de portas em uma determinada ***sessão***. `ForroSessionDriver` contém as seguintes listas de: conexões, propriedades das conexões, componentes, propriedades de componentes e propriedades das portas dos componentes. Estas listas são associadas a uma específica ***location*** e a uma específica ***sessão***.

Services: a *interface* `Services` é implementada pela classe `ForroServices`. O objeto `ForroServices` contém a lista local de portas em cada componente.

ConnectionEventService: informa o componente das conexões feitas. A *interface* `ConnectionEventService` é implementada pela classe abstrata `AbstractConnectionEventBuilderService` que possui uma lista de eventos contendo o tipo do evento e o conjunto de *listeners*. A classe `AbstractConnectionEventBuilderService` é estendida pela classe `ForroSessionDriver`.

AbstractFramework: permite o componente se comportar com uma plataforma. `AbstractFramework` é implementada pela classe `Forro`.

ComponentRepository: é um repositório no qual os componentes são encontrados. A classe `Forro` implementa pela *interface* `ComponentRepository`.

5.4.2 Componentes Adicionais

`ConnectingSpaceFactory`: componente que fornece portas *provides* que são ***links*** cujo comportamento coletivo implementa um modelo de computação. Esse modelo de computação especifica características que dependem ou são relacionadas ao sincronismo de computações que envolvem as ***locations***. Um componente pode se comunicar através de qualquer ***Connecting Space*** que possua a ***location*** que o contém. Para isso, basta utilizar um conector com uma porta `Link` fornecida pelo `ConnectingSpaceFactory` correspondente. `ConnectingSpaceFactory` é distribuído entre as ***locations*** contidas em determinado ***Connecting Space***.

Identificação Única dos Elementos

A identificação de todos elementos da plataforma *Forró* é feita por nomes únicos e `ComponentIDs` (para o caso de componentes) atribuídos pelo `ForroDriver`. O `ComponentID` identifica unicamente qualquer componente instanciado no *Forró*.

Comunicação com o Usuário

A comunicação com o usuário é feita por uma API textual, chamada *Frontend*. A API é composta por várias classes e utiliza alguns padrões de implementação. Essa API fornece as operações típicas de uma plataforma CCA. Conforme já foi mencionado, um dos pontos fortes do *Forró* é também sua simplicidade. Há apenas oito operações "primitivas" fornecidas ao usuário:

- `repository init <localhost> <localsession>`: inicializa um repositório criando uma ***location*** uma ***sessão***. É o comando inicial de qualquer aplicação no ambiente *Forró*;
- `create <component>`: cria uma instância de um componente;
- `remove <component>`: remove uma determinada instância de um componente;
- `create connectingspace`: cria uma instância de um ***espaço de conexão***;
- `link`: cria um `Link` entre ***locations***. `Link` inclui, como parâmetros, as ***locations*** que são seus pontos extremos. A ordem dos parâmetros estabelece as identidades dos pontos extremos no ***link***;
- `connect`: cria uma conexão direta entre dois componentes ou entre dois ***espaços de conexão***;
- `go`: executa uma porta `GoPort` de uma instância de um componente;
- `quit`: finaliza a execução da ***sessão*** corrente.

Comunicação entre Componentes e Código Nativo

Para a comunicação com outras linguagens, desenvolveu-se no *Forró* a seguinte estrutura: um gerenciador nativo do *Forró*, o `NativeDriver` e um componente encapsulador do componente nativo original (ver Código 5.1). Com estes dois elementos, o *Forró* consegue disponibilizar os métodos para os componentes Java e para os componentes nativos.

Tabela 5.1: Código do componente encapsulador de um componente nativo em C++.

```
...
public class JNINativeComponent1Port implements NativeCDriverPort {
        /* Load the library to communicate with C process */
        static {
                System.loadLibrary("C1");
        }
        /** Native Method of a C component 1 **/
        public native int methodA();
        ...
}
```

No componente encapsulador, tem-se somente chamadas nativas dos método (ver Figura 5.7). O repasse ao métodos originais é feito através do JNI. Se um componente nativo precisar se comunicar com outro componente nativo, esta chamada não será repassada ao *Forró* do lado Java. A chamada será feita diretamente, se os componentes estiverem em máquinas distintas. O gerenciador nativo apenas registra os componentes e as portas disponibilizadas do lado nativo.

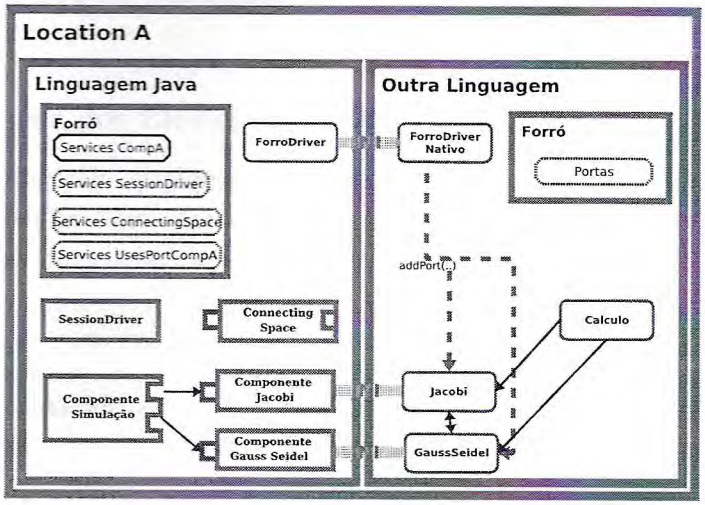


Figura 5.7: Associação entre componentes na plataforma *Forró* e os módulos em código nativo.

Na Figura 5.7, do lado da linguagem nativa temos três códigos nativos `Jacobi`, `GaussSeidel` e `Calculo`, e o `ForroDriverNativo`. Do lado Java, temos os componentes encapsuladores `ComponenteJacobi` e `ComponenteGaussSeidel`. Além disso, temos os *drivers* e um componente `ComponenteSimulação` que usa os métodos de Jacobi e GaussSeidel de forma transparente, ou seja, sem se preocupar em que linguagem estes métodos estão implementados.

5.4.3 Funcionamento dos Elementos do *Forró*

Para executar uma aplicação no ambiente *Forró*, é preciso definir o comportamento dos componentes e a composição da rede. O comportamento dos componentes deve ser configurado através das conexões entre as portas *uses* e *provides* associadas aos componentes da aplicação. A composição da rede deve ser feita através da criação dos seguintes elementos: repositórios, ***espaços de conexão*** e ***links*** entre as máquinas.

A execução no ambiente *Forró* pode ser dividida em duas fases: preparação do ambiente e execução de comandos. Na preparação do ambiente, as seguintes tarefas são necessárias: geração de bibliotecas nativas, atribuição de variáveis de ambiente e inicialização da plataforma *Forró*. Na execução de comandos, os seguintes passos são realizados: criação de repositório, criação de componentes, criação de ***espaço de conexão***, criação de `links`, conexão de componentes e execução de portas `GoPort`.

Criação de Instâncias

Para a criação de uma instância, obtém-se a lista de argumentos e a lista das classes dos argumentos. Em seguida, a chamada do método é montada, o componente é criado, o

objeto *Services* é atribuído, a instância é registrada e as propriedades do componente são atribuídas.

Identificação Única dos Elementos

O identificador único de qualquer componente no *Forró* é também conhecido como `ComponentID`. Além do `ComponentID`, nome de uma instância de um componente deve ser único dentro de uma determinada `location`.

Criação de *Espaço de Conexão*

Um ***espaço de conexão*** exige alguns elementos para serem criados, já que deve ser composto por ***links*** parametrizados por um mesmo ***duct***. A seguir, são listadas as etapas a serem cumpridas para criação dos componentes e conexões que formam um ***espaço de conexão***:

1. A criação da infra-estrutura de comunicação, a qual dá sustentação aos ***dutos***, deve ser iniciada. Para cada ***location*** do ***espaço de conexão*** sendo criado, deve-se criar uma instância do componente `InfrastructureFactory`. Naturalmente, espera-se que a referida ***location*** esteja registrada nessa infra-estrutura. Cada um desses componentes possui uma porta *provides* do tipo `DuctFactoryPort`.

2. Em cada ***location*** do componente `InfrastructureFactory`, existe a possibilidade de criação de ***links*** parametrizados pelo `Duct` correspondente. Para isso, é preciso instanciar um componente do tipo `ConnectingSpaceFactory` referente ao ***espaço de conexão*** que se deseja criar. A porta *uses* desse componente, do tipo `DuctFactoryPort`, deve ser conectada à porta *provides* do `InfrastructureFactory`. Através dessa porta, um `Duct` (mais concretamente, uma porta do tipo `Duct`) pode ser obtido para ser usado como parâmetro dos ***links*** que vierem a ser criados. O componente `ConnectingSpaceFactory` possui uma porta *provides* do tipo `LinkFactoryPort`. A criação de ***locations*** é feita pelo `ForroDriver`.

3. A solicitação de criação de um ***link*** nesse ***espaço de conexão*** faz aparecer uma porta *provides* do tipo `LinkFactoryPort` em cada `ConnectingSpaceFactory` das ***locations*** envolvidas nesse ***link***. Toda porta do tipo `Link` obtida dessa `LinkFactoryPort` efetua suas comunicações pelo respectivo ***link*** do ***espaço de conexão***. As identificações de origem e destino das mensagens nos métodos da *interface* `Link` dizem respeito às identidades dentro do próprio ***link***.

Criação de Conexão Indireta

Conforme o modelo de conexão adotado, uma conexão indireta é feita com conectores possuindo portas do tipo `Link`. As etapas a seguir devem ser seguidas para realizar uma conexão indireta:

1. Um componente do tipo `Connector` referente ao tipo de conector que se deseja utilizar é instanciado em cada ponto extremo da conexão. Cada um desses conectores possui uma porta *uses* do tipo `LinkFactoryPort`.

2. A porta `LinkFactoryPort` de cada conector criado deve ser conectada ao `ConnectingSpaceFactory` correspondente. Ao se realizar essa conexão, o *Forró* invoca o método `setServices` do conector para que este solicite uma porta `Link` e registre-a como porta *provides*.

3. As portas dos componentes envolvidos e o componente UsesPortIndirection (Figura 5.5) na conexão indireta podem ser conectadas, por conexão direta, às portas `Link` dos conectores.

Comunicação entre Componentes

As interações entre os componentes ocorrem de acordo com a configuração e as conexões da ***sessão*** estabelecidas entre os componentes pelo `ForroDriver`. Conforme explicado, temos dois tipos de conexão direta e indireta.

A seguir, ilustramos passo-a-passo, internamente no *Forró*, como funciona a criação de uma conexão indireta:

1. O usuário solicita ao *Forró* uma conexão da porta *uses* `UsesPortA` com a porta *provides* `ProvidesPortB`. O *Forró* solicita à `SessionDriver` conexão das portas através da chamada `BuilderService.connect(UsesPortA, ProvidesPortB)` como mostrado na Figura 5.8.

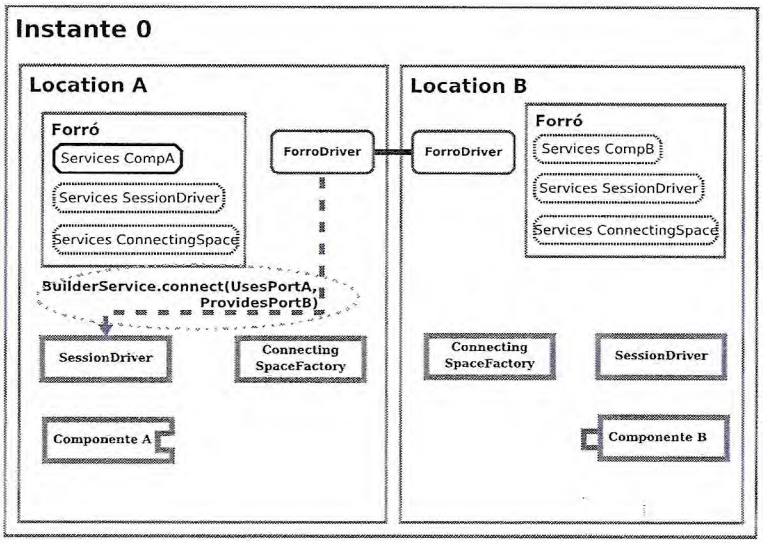


Figura 5.8: Passo Inicial da comunicação entre dois componentes remotos.

2. O *Forró* cria uma instância de um componente com a mesma *interface* da porta *uses* `UsesPortA` (ver Figura 5.9).

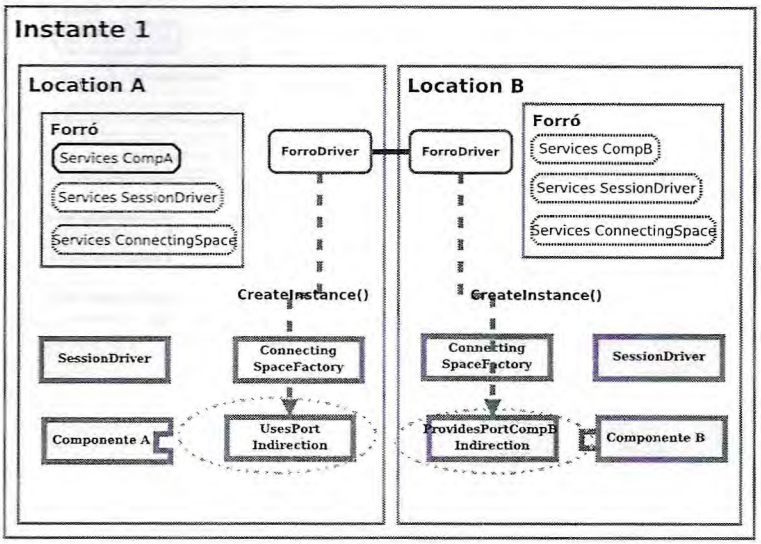


Figura 5.9: Passo 1 da comunicação entre dois componentes remotos.

3. O *Forró* adiciona as portas *uses* e *provides* do componente criado (ver Figura 5.10).

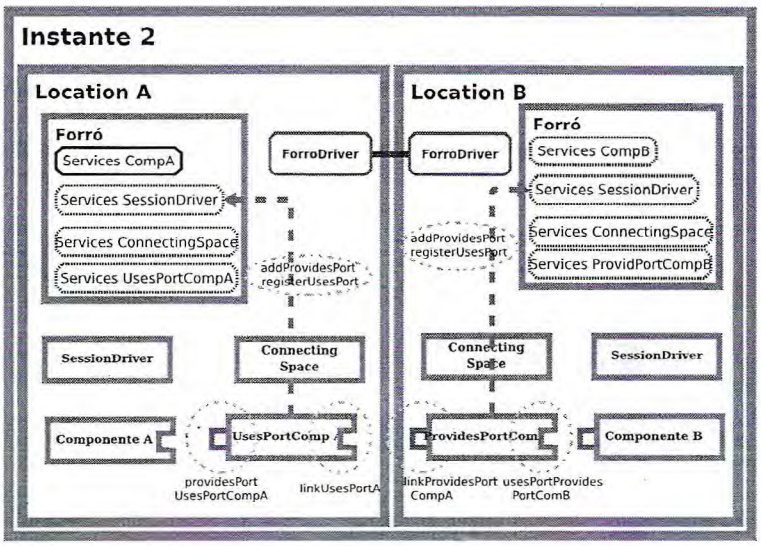


Figura 5.10: Passo 2 da comunicação entre dois componentes remotos.

4. O *Forró* conecta a instância criada com o componente `ComponentA` (ver Figura 5.11).

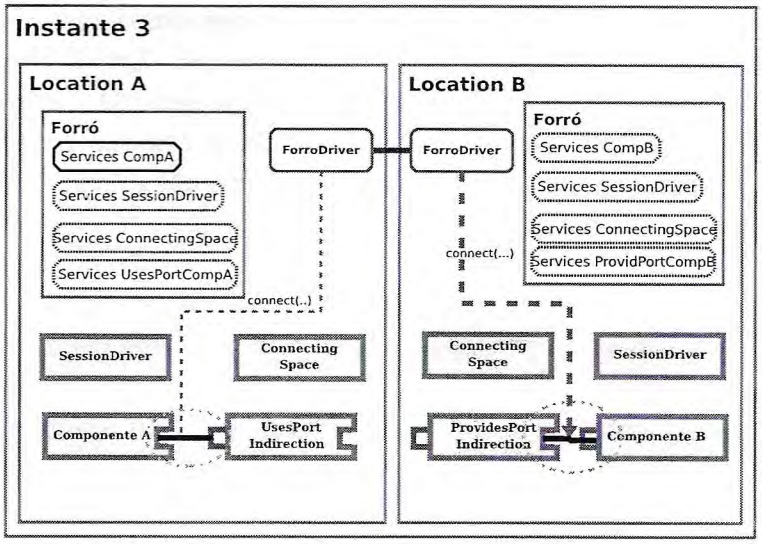


Figura 5.11: Passo 3 da comunicação entre dois componentes remotos.

5. Na ***Location*** B. o *Forró* conecta a nova instância ao `ConnectingSpaceDriver` do componente `ComponentB` e `ConnectingSpaceDriver` se conecta ao componente `ProvidesPortComponentB`. Em seguida, o componente `ProvidesPortComponentB` se conecta ao componente `ComponentB`. Desta forma, a conexão remota está montada entre o componente `ComponentA` e o componente `ComponentB` (ver Figura 5.12).

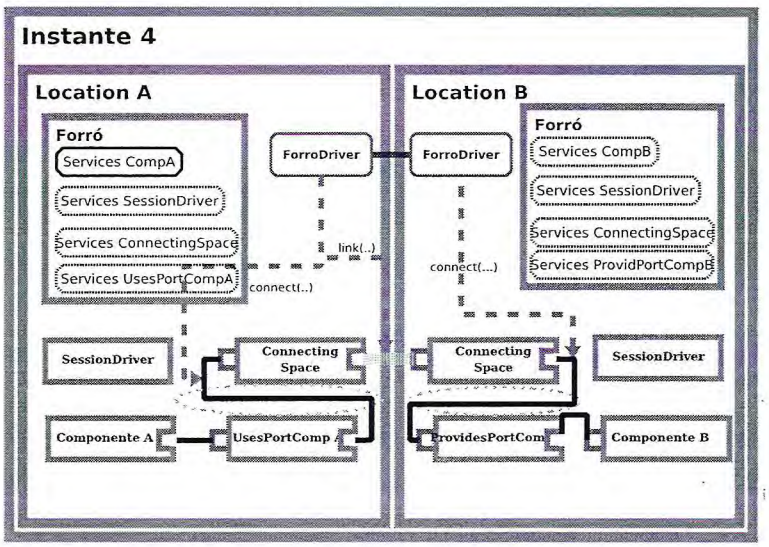


Figura 5.12: Passo 4 da comunicação entre dois componentes remotos.

Invocação de Método nas Conexões Indiretas Para invocar um método `methodM` implementado em um componente `ComponentB`, o método deve estar disponível em uma porta *provides* `PortX`. Suponha que exista um certo componente `ComponentA` que precisa desta porta. Quando o `ComponentA` invoca o `methodM` utilizando uma porta *uses* conectada com a porta `portX`, a chamada é transformada em uma mensagem pelo componente `UsesPortIndirection` correspondente. Em seguida, a mensagem é encaminhada pelo ***link*** associado, que por sua vez encaminha a chamada para o ***link*** do `ComponentB` utilizando o `Duct`. Nesse ponto, o conector pode responder a uma invocação de recebimento de mensagem em sua porta `Link` de parte da `ProvidesPortIndirection`. O resultado do método parte em direção ao `Component` seguindo procedimento similar, em sentido inverso.

Um fato que merece destaque é o funcionamento dessas conexões quando a própria aplicação é construída sobre algum modelo de computação baseado em trocas de mensagens. Nesse caso, as portas `Link` envolvidas podem ser os próprios ***dutos*** diretamente, removendo assim várias das indireções.

A Figura 5.13 ilustra um cenário típico no qual um método de um componente é invocado no *Forró* em `locations` distintas.

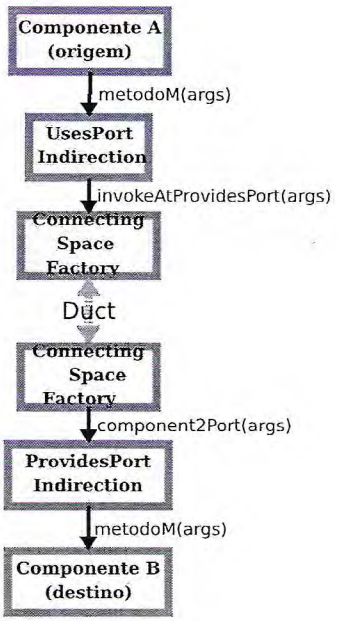


Figura 5.13: Representação da invocação de um método.

`LocationDriver`: gerenciador de `locations`. o `LocationDriver` provê três tipos de métodos, são eles: métodos de configuração, de conexão e de interação do modelo. Os métodos de configuração são responsáveis por iniciar componentes na `location`, incluir uma `location` em um `Connecting Space` e criar um `link` desta `location` para uma outra `location` em um `Connecting Space`. Os métodos de conexão consistem

em pedidos de conexões diretas de portas. Os métodos de configuração e conexão são sincronizados para evitar que sejam executados simultaneamente com as interações dos componentes.

Este componente é criado a partir de uma classe abstrata contendo o método `setServices`, que não pode ser sobrescrito. O objeto `Services` fornecido para essa tem um comportamente especial, conforme descrito mais adiante.

`InfrastructureFactory`: componente encarregado da infra-estrutura de comunicação, fornecendo portas to tipo `Duct` a componentes do tipo `ConnectingSpaceFactory`. Este componente também é criado a partir de uma classe abstrata que implementa a sua porta *provides*.

Os componentes e conexões do *Forró* que implementam uma conexão da forma ilustrada na Figura 4.5 são mostrados na Figura 5.14.

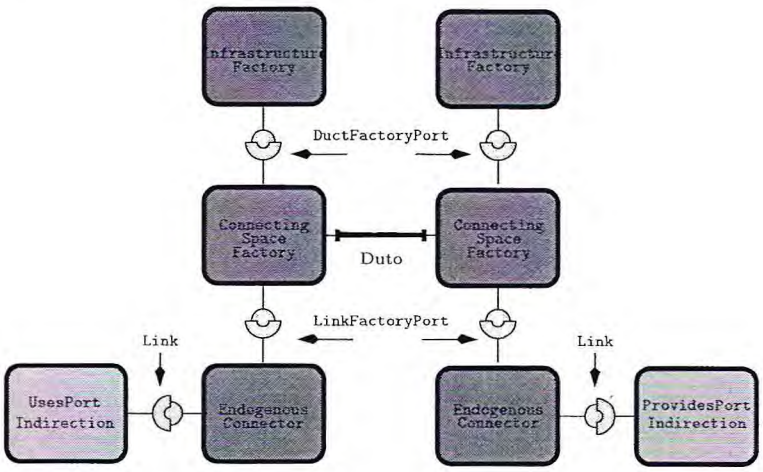


Figura 5.14: Uma conexão indireta através de conectores endógenos e ***links*** especializados no *Forró*.

Na Figura 5.14 temos uma conexão indireta disponibilizada pela seguinte estrutura:

1. Componente `UsesPortIndirection`: que direciona as invocações de métodos em uma determinada porta `Uses` por um `link` para um conector endógeno.

2. Conector `EndogenousConnector`: que recebe o `link` e o repassa para o espaço de conexão `ConnectingSpaceFactory`.

3. Espaço de conexão `ConnectingSpaceFactory`: que recebe o `link` e o repassa para outro espaço de conexão `ConnectingSpaceFactory`.

4. Espaço de conexão `ConnectingSpaceFactory`: que recebe o `link` e o repassa para o conector `EndogenousConnector`.

5. Conector `EndogenousConnector`: que recebe o `link` e o redirecciona as invocações de métodos para uma determinada porta `Provides` por um `link` para componente um conector `UsesPortIndirection`.

Os componentes `InfrastructureFactory` são elementos de configuração que estabelecem propriedades da infra-estrutura.

Nesta implementação atual, a capacidade de interpretação de uma linguagem de descrição de *interface* (do inglês IDL) não está disponível ainda.

Podemos ver outros exemplos de comunicação nas Figuras 5.15, 5.16 e 5.17.

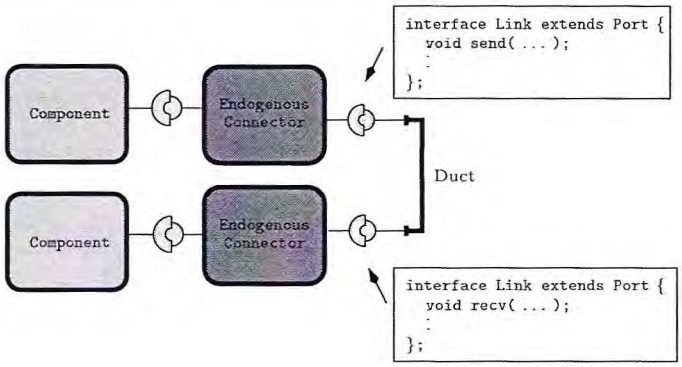


Figura 5.15: Aplicação MPI.

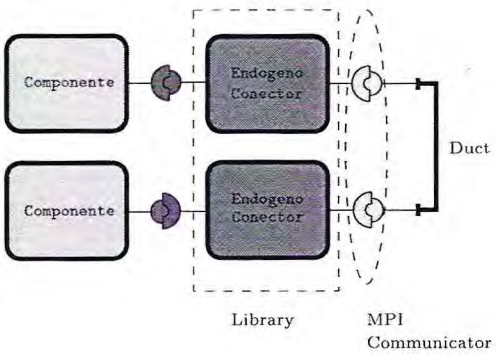


Figura 5.16: Uso de biblioteca paralela.

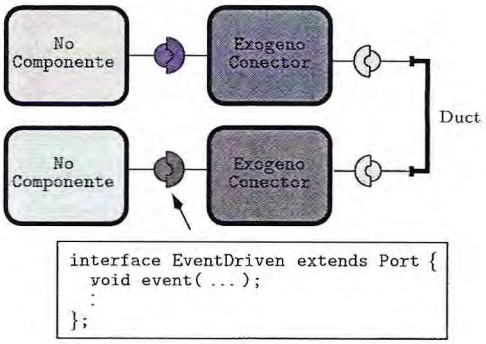


Figura 5.17: Modelo baseado em eventos.

Capítulo 6

Estudos de Caso

Este capítulo apresenta o desenvolvimento de estudos de caso com o objetivo de avaliar a especificação de componentes e conectores e também avaliar o peso da plataforma.

O primeiro estudo de caso usa o *Forró* em um problema de otimização sobre Conjunto Independente Máximo (CIM) em grafos. O segundo e o terceiro exemplos são, respectivamente: um código nativo para solução de equações lineares; e, um sistema que auxilia no estudo de fraturas em dutos e pavimentos chamado Simfra. O quarto exemplo é uma aplicação de referência que ordena inteiros.

Os estudos de caso foram escolhidos porque representam aplicações bem distintas, possuindo públicos alvos diferentes, possibilitando uma boa representação. Primeiramente, na Seção 6.1 descrevemos as considerações iniciais. Na Seção 6.2, mostramos o estudo de caso que mostra problemas de otimização sobre conjuntos independentes em grafos. Finalmente, a Seção 6.6 apresenta as considerações finais deste capítulo.

6.1 Considerações Iniciais

Este capítulo apresenta os estudos experimentais que foram realizados no contexto desta tese. Antes mesmo de descrever esses estudos, é necessário ressaltar a importância da realização de estudos dessa natureza para a avaliação de uma nova plataforma de componentes. Retomando o histórico da área de Engenharia de *Software*, é comum observar afirmativas feitas na literatura com comprovação incipiente em relação à teoria que as sustenta, à transferência de novos métodos; ou técnicas imaturas da academia para a indústria com comprovação também incipiente da sua eficácia e eficiência em relação aos existentes.

Segundo [120], estudos de caso são usados para monitorar projetos, atividades ou atribuições. Através destes estudos, é possível observar um atributo ou estabelecer relacionamentos entre diferentes atributos. Estudos de caso podem ser organizados para comparar resultados de projetos similares, um utilizando uma tecnologia atual e outro uma nova tecnologia.

A proposta inicial deste estudo de caso foi avaliar todas atividades que compreendem a composição de aplicações a partir da plataforma *Forró*. Partiu-se, então, para a identificação de características estratégicas para a composição de aplicações. Para aplicar os conceitos propostos por este trabalho foi escolhido um estudo de caso descrito na próxima seção.

6.2 Estudo de Caso: Problemas de Otimização sobre Conjuntos Independentes em Grafos

As motivações da escolha de problemas de otimização sobre conjuntos independentes em grafos como estudo de caso foram, principalmente, a necessidade de alto desempenho, possibilidade de paralelismo e a existência de códigos legados.

6.2.1 Descrição do Problema

Um grafo [121, pág.3] é uma estrutura matemática que é usada para modelar relações binárias entre elementos de um conjunto enumerável (usualmente, finito). Esses elementos são representados pelos vértices do grafo G, enquanto uma aresta uv de G, definida pelos vértices u e v, indica que esse par de vértices pertence à relação. O fato de uv ser uma aresta indica a existência de alguma forma de dependência entre os elementos representados pelos vértices u e v. Muitos problemas de otimização combinatória que ocorrem com frequência em diversas aplicações envolvem a determinação de conjuntos de vértices dois a dois independentes de um grafo. Neste estudo de caso, apresentamos uma implementação com componentes CCA para um desses problemas.

Para tornar a discussão mais específica, seja $G = (V, E)$ um grafo não-direcionado, simples, não-vazio, conexo e constituído de n vértices (V) e m arestas (E) [121, pág.3]. Na linguagem usualmente adotada na literatura sobre grafos, diz-se que $u \in V$ e $v \in V$ são adjacentes quando $uv \in E$. Um conjunto independente de G é um subconjunto $W \subseteq V$ de vértices tal que, para todo par u e v de vértices distintos em W, $uv \notin E$. Por essa propriedade, constata-se que um conjunto independente não pode conter vértices que sejam adjacentes em G. Dessa forma, um conjunto independente expressa a ideia de um conjunto de elementos que não formam pares na relação binária expressa pelo conjunto de arestas.

Há uma grande variedade de funções que podem ser definidas com relação aos subconjuntos de vértices de um grafo. Uma dessas funções, que podemos chamar de $f : 2^V \rightarrow \mathbb{N}$, indica o tamanho de um conjunto independente (a notação 2^V é usada para representar o conjunto de todos os subconjuntos de V). Outra função semelhante é $\omega : 2^V \rightarrow \mathbb{R}$, que mede o peso de cada subconjunto de V, a qual pressupõe a existência de um peso $\omega(v)$ associado a cada vértice de V. A função de peso mais comum, e que também será usada neste estudo de caso, é $\omega(W) = \sum_{v \in W} \omega(v)$. Tratamos a seguir de alguns problemas de otimização enunciados sobre conjuntos independentes e as funções

ℓ e ω. Passemos, então, a uma breve exposição desses problemas, na qual denominamos um vetor x, binário, de tamanho n e indexado pelos vértices de G, de vetor de incidência de um subconjunto W de V se $x[v] = 1$, se $v \in W$, e $x[v] = 0$ em caso contrário. Observe que um vetor de incidência de $W \subseteq V$ traz em si a informação do seu tamanho, visto que $\ell(W) = \sum_{v \in V} x[v]$. Em particular, x é um vetor de incidência de um conjunto independente de G se e somente se $uv \in E$ implica $x[u] + x[v] \leq 1$ [122]. Observe que essa exigência faz com que $x[u] = x[v] = 0$, $x[u] = 1$ ou $x[v] = 1$. Para tornar a notação mais simples, escrevemos $x(W)$ para representar $\sum_{v \in W} x[v]$, $W \subseteq V$.

Comecemos pelo problema de Conjunto Independente Máximo (CIM), o qual consiste na determinação de um conjunto independente que maximize $x(V)$. Em termos matemáticos,

$$\alpha(G) = \max \{x(V) \mid x \in \mathbb{B}^n, x[u] + x[v] \leq 1, u, v \in V, uv \in E\}, \tag{6.1}$$

em que $\mathbb{B}^n$ é o conjunto de vetores binários de tamanho n. O valor $\alpha(G)$ é chamado de número de estabilidade de G. O problema CIM é considerado computacionalmente difícil pois pertence a classe NP-Difícil [123], fato esse que está ligado à possibilidade de existência de um número exponencial de conjuntos independentes em um grafo (esse número pode chegar a $3^{n/3}$ [124]). Ainda tomando-se a função ℓ, podemos definir uma generalização do problema CIM, a qual chamamos de problema Subconjunto k-Particionável Máximo (SkPM) [125]. Cada instância desse problema é identificada por um parâmetro $1 \leq k \leq n$ (além do próprio grafo G) e consiste na determinação do maior subconjunto $W \subseteq V$ que aceita uma partição $W = W_1 \cup W_2 \cup \ldots W_k$ em k conjuntos independentes (mais precisamente, W_i, para todo $i \in \{1, 2, \ldots, k\}$, deve ser um conjunto independente),

$$\alpha(G, k) = \max \{x(V) \mid x \in \mathbb{B}^n, \exists W_1, W_2, \ldots, W_k, x[u] + x[v] \leq 1\}, \tag{6.2}$$

em que $u, v \in W_i, uv \in E, i = 1, 2, \ldots, k$.

O problema CIM corresponde ao caso particular em que $k = 1$. A atribuição de pesos aos vértices de G, e a consequente maximização da função ω, leva à definição do problema de Problema do Conjunto Independente de Peso Máximo (CIPM) na forma

$$\alpha_\omega(G) = \max \left\{ \sum_{v \in V} \omega(v) x[v] \mid x \in \mathbb{B}^n, x[u] + x[v] \leq 1, uv \in E \right\}. \tag{6.3}$$

Fazemos uma breve pausa na explanação para definir uma notação complementar. O complemento de G é denotado por $\bar{G} = (V, \bar{E})$, em que $uv \in \bar{E} \Leftrightarrow uv \notin E$, para todo $u, v \in V$, $u \neq v$. Tomamos um membro v de V para as definições que seguem. A notação $\bar{N}(v)$ representa o conjunto $\{u \in V \mid uv \in \bar{E}\}$. Uma orientação de G é uma função $\sigma : E \to V$ tal que $\sigma(uv) \in \{u, v\}$. Uma orientação σ é acíclica se todo subconjunto $U \subseteq V$ não-vazio possui (pelo menos) um vértice $s \in U$ tal que $\sigma(sv) = v$, para todo $v \in U$ e $sv \in E$. Uma orientação acíclica de $\bar{G}$ define a vizinhança-emanante u, $\bar{N}^+(u) = \{v \in \bar{N}(u) \mid \sigma(uv) = v\}$, e a vizinhança-atraída de v, $\bar{N}^-(v) = \bar{N}(v) \setminus (\{v\} \cup \bar{N}^+(v))$.

As notações $\bar{N}^+[v] = \bar{N}^+(v) \cup \{v\}$ e $\bar{N}^-[v] = \bar{N}^-(v) \cup \{v\}$ são usadas para indicar a inclusão de um vértice em sua própria vizinhança. Nós escrevemos $G^+(v)$ no lugar de $G[\bar{N}^+[v]] = (\bar{N}^+[v], E[\bar{N}^+[v]])$ para designar o subgrafo de G induzido por $\bar{N}^+[v]$. $V^+(v) = N^+(v) \cup \bar{N}^+(v)$, $V^+[v] = V^+(v) \cup \{v\}$. Antes de fechar esta subseção, um último comentário sobre a definição do problema SkPM. Um fato que se destaca quando analisamos esse problema é que uma instância do próprio problema pode ser vista como uma combinação adequada de k instâncias do problema CIM. Para chegar a essa tal combinação, observamos inicialmente que $W \cup \{r\}$, com $W \subseteq V$ e $r \in V$, é um conjunto independente de G cujo vértice minimal (com respeito a $\prec$) é r se e somente se W é um conjunto independente de $G^+(r)$. O termo representante é usado em referência ao vértice minimal r de um conjunto independente e $\bar{N}^+[r]$ é o conjunto de vértices que podem ser representados por $r \in V$ (note que r pode representar a si mesmo). O método de resolução que descrevemos em seguida baseia-se na ideia de que toda solução do problema SkPM pode ser descrita por dois representantes e, para cada um desses representantes, o conjunto independente que ele representa.

6.2.2 Desenvolvimento da Aplicação

Conforme citado, o desenvolvimento e a implementação dos estudos de caso possui três etapas principais (modelagem, implementação e execução da aplicação), as quais são explicadas nas próximas subseções.

Modelagem

Seguindo a observação acima, uma estratégia para a resolução do problema SkPM envolve a resolução de diversas instâncias do problema CIPM. As instâncias envolvidas correspondem a determinar um conjunto independente $W \subseteq \bar{N}^+[r]$ para cada vértice $r \in V$. Uma vez obtidos esses conjuntos independentes, escolhem-se k dentre eles de forma a maximizar o número de vértices escolhidos. Nessa estratégia, a escolha dos k conjuntos independentes é realizada com a ajuda de um vetor λ, cuja entrada $\lambda[v]$ funciona como um regulador que faz com que cada subproblema somente contribua para a solução final se ele possui um conjunto independente de peso maior que o valor estipulado por esse regulador. A contagem do número total de vértices escolhidos é feita levando em conta que, mesmo que um dado vértice apareça em mais de um conjunto independente, a contagem total deve ser capaz de contar cada vértice apenas uma vez. Com esse propósito, os vértices recebem pesos na forma

$$\max \left\{ -\lambda[r] + \sum_{v \in \bar{N}^+[r]} (1 - \mu[v]) x^r[v] \mid x^r \in \mathbb{B}^{|\bar{N}^+[r]|}, x^r[u] + x^r[v] \leq 1., u, v \in \bar{N}^+[r], uv \in E \right\}. \tag{6.4}$$

em que o vetor $\mu \in \mathbb{R}^{|\bar{N}^+[r]|}$, indexado pelos vértices de $\bar{N}^+[r]$, tem o papel de penalizar a escolha do vértice v em mais de um conjunto independente. Se $\mu[v] = 0$, para algum

$v \in \bar{N}^+[r]$, então v contribui com uma unidade no caso de $x[v] = 1$. Por outro lado, se $\mu \geq 1$, então v não contribui para o valor da solução.

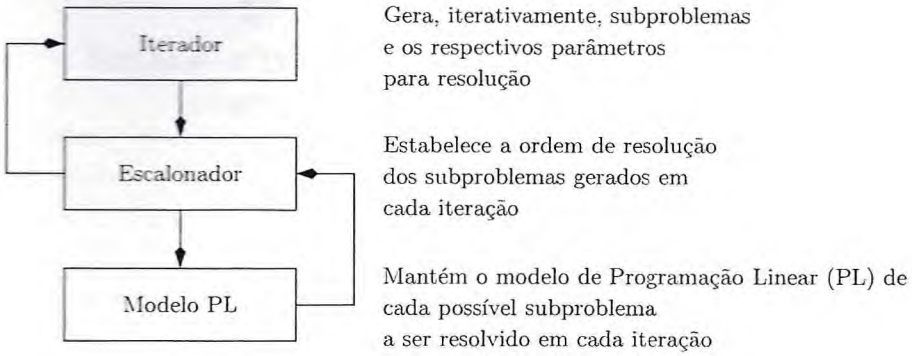


Figura 6.1: Fluxograma do método geral utilizado para obter limites inferiores e superiores para $\alpha(G, k)$.

Há dois passos empregados no método geral de solução ilustrado na Figura 6.1. O primeiro passo é a relaxação lagrangiana, a qual é aplicada com relação à restrição (6.1). Essa relaxação permite obter subproblemas de CIPM independentes de forma iterativa e está representada pelo bloco "Iterador". O segundo passo, conhecido como geração de planos de corte, é aplicado a esses subproblemas. As restrições desses problemas derivam unicamente da estrutura do grafo, o que significa que permanecem constantes no decorrer de todas as iterações geradas pelo bloco "Iterador". Por essa razão, os modelos de programação linear dos subproblemas são construídos no início do processo e guardados no módulo "Modelo PL" [126]. A variação que ocorre em cada subproblema de uma iteração para a seguinte está nos coeficientes da função objetivo. Portanto, quando um subproblema é escalonado para execução em uma certa iteração, é preciso configurar a função objetivo no seu modelo e usar a função de resolução desse modelo. Esta descrição deixa claro que a interação entre as duas abordagens é de responsabilidade do bloco "Escalonador", que recebe a lista de subproblemas a serem resolvidos em cada iteração, e escalona as funções de resolução dos respectivos modelos, em uma ordem apropriada.

O algoritmo final tem por objetivo obter limites, inferior e superior, para o valor de $\alpha(G, k)$. O limite inferior obtido corresponde ao valor da melhor solução obtida durante o algoritmo. Para o limite superior, pode-se observar que definindo uma variável $x[v]$ para

indicar se v é o representante de um conjunto independente, para todo $v \in V$, segue que

$$\begin{aligned}
\alpha(G,k) = \max \quad & \sum_{v \in V} x^v(\bar{N}^+[v]) \\
\text{Sujeito a} \quad & x^v[w] + x^v[w'] \leq 1, && v \in V, ww' \in E[\bar{N}^+(v)] && (6.5)\\
& x^v[w] \leq x^v[v], && v \in V, w \in \bar{N}^+(v) && (6.6)\\
& x^v[v] \leq x[v], && v \in V && (6.7)\\
& x(V) \leq k && && (6.8)\\
& x[v] \leq 1, && v \in V && (6.9)\\
& \sum_{u \in \bar{N}^-[v]} x^u[v] \leq 1, && v \in V && (6.10)\\
& x^v[w] \in \{0,1\}, && v \in V, w \in \bar{N}^+[v]. && (6.11)
\end{aligned}$$

Devido a (6.8) e (6.9), toda solução ótima contém exatamente k representantes. Se $x[v] = 1$, então v é o representante do conjunto independente de $G^+(v)$ descrito por $x[v]$ e o vetor de incidência x^v definido por (6.5)–(6.6) e (6.11). Caso contrário, $x[v] = 0$ e x^v não contribui para a função objetivo. A restrição (6.6) visa lidar com vértices isolados em $G[\bar{N}^+(v)]$. As desigualdades (6.10) impõem a condição que cada vértice pertença a um conjunto independente, no máximo.

$$\alpha(G) = \max\{x^r(\bar{N}^+[r]) \mid x^r \in STAB(G^+(r)), r \in V\}. \tag{6.12}$$

Implementação

A implementação do método ilustrado na Figura 6.1 através do uso de componentes CCA é motivado por alguns fatos. Em primeiro lugar, os diferentes componentes (os quais passaremos a descrever em breve, e estão ilustrados na Figura 6.2) são destinados a realizar procedimentos que podem ser reutilizados e recombinados. Dessa forma, esses mesmos componentes podem compor outras abordagens para o mesmo problema ou para outros problemas envolvendo conjuntos independentes em grafos. A segunda motivação especial é a natural decomposição, com a consequente paralelização em memória compartilhada, que o uso do conector exógeno permite. Esta é uma forte motivação quando levamos em conta a atual disponibilidade da tecnologia de múltiplos núcleos. Além disso, a possibilidade de implementações distribuídas que a plataforma *Forró* torna possível pelo uso de conectores indiretos. A estrutura de componentes, portas e conectores é apresentada na Figura 6.2, cujas descrições detalhadas estão a seguir.

Componentes – os componentes utilizados correspondem aos módulos identificados na Figura 6.1, com as especificidades necessárias para o caso particular do problema SkPM.

`Subgradient`: execução do método do subgradiente para o problema SkPM. Este método gera uma sequência de multiplicadores λ e μ iterativamente. Em cada iteração, os

subproblemas devem ser resolvidos, regulados por λ para que, a partir das soluções obtidas, um limite superior seja determinado. Este componente é o componente possuindo uma `GoPort` que permite dar início à execução.

`Scheduler`: mantém os representantes adicionados em uma estrutura de dados de maneira a ordenar a execução dos subproblemas respectivos de acordo com as suas prioridades. Quando adicionado pelo subgradiente, um representante deve receber como prioridade o valor de λ e os coeficientes da função objetivo para a iteração corrente; quando adicionado pelo `solver`, a prioridade corresponde ao limite superior corrente.

`B&C`: componente que contém os modelos de programação linear dos subproblemas, além de um método *branch-and-bound* [118] para a solução exata do problema CIPM. Esses modelos podem ser resolvidos concorrentemente, por não compartilharem memória.

Portas – as portas utilizadas para conectar os componentes descritos acima também são identificadas na Figura 6.2. Alguns dos métodos dessas portas são descritas a seguir:

`LPModel`: a configuração dos modelos de programação inteira dos subproblemas são realizados pelo `SubgradientComponent`, visto que este é o componente que realiza a decomposição do problema SkPM, gerando assim os subproblemas CIPM. Através desta porta, o método `build` permite a definição do modelo de cada subproblema e alterar os pesos da função objetivo. Além disso, a execução do método iterativo no `SubgradientComponent` depende de valores produzidos pela solução dos subproblemas. Esta porta contém métodos que tornam possível a consulta a valores de variáveis e limites inferiores ou superiores.

`LPScheduler`: permite ao `SubgradientComponent` incluir um subproblema no grupo sendo manipulado pelo `SchedulerComponent`. O método `add` é destinado a esse fim, tendo como parâmetros uma identificação do subproblema e uma prioridade inicial atribuída a esse subproblema. Outro método disponível nesta porta é `schedule`, permitindo ao conector exógeno requerer um subproblema.

`LPSolver`: define a interação entre o conector exógeno e o `B&CComponent`, usando para isso o método `solve`, o qual dispara o algoritmo B&C para um subproblema especificado.

Conectores – além dos conectores endógenos (diretos), merece destaque especial o conector exógeno `ExogenousConnector` que aciona a resolução de um subproblema. O subproblema a ser acionado é obtido do `SchedulerComponent` via método `schedule`.

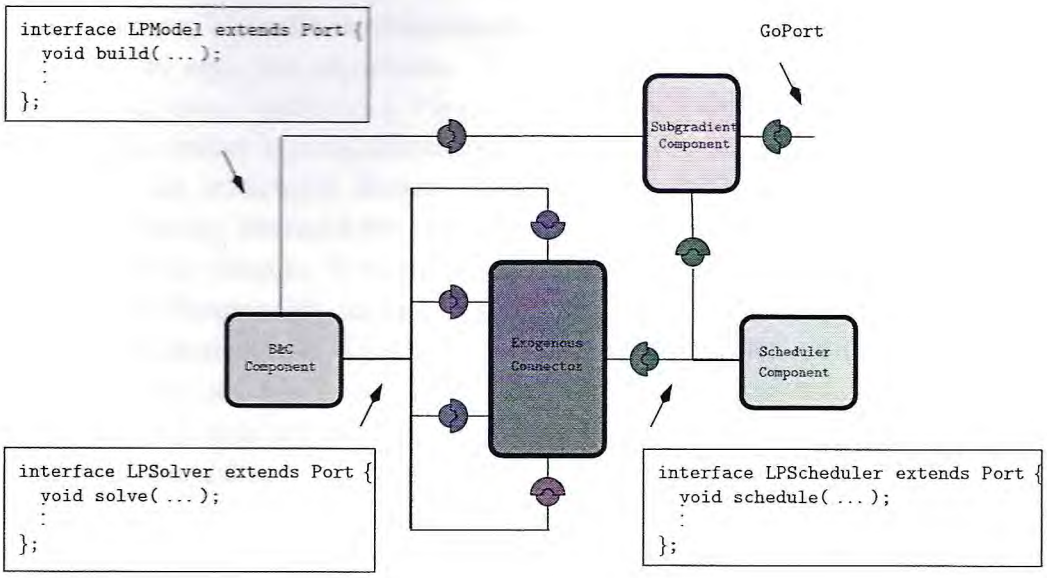


Figura 6.2: Componentes, conectores e portas para determinação de limites inferiores e superiores para o problema SkPM.

Execução

Usamos alguns experimentos computacionais com o objetivo de comprovar a viabilidade dos métodos propostos, tanto da plataforma *Forró*, quanto da aplicação descrita neste capítulo. Alguns resultados desses experimentos são apresentados e analisados a seguir. A implementação utilizada nos experimentos inclui componentes CCA em Java, com os métodos implementados na linguagem C. Em particular, o componente `B&CComponent` utiliza o CPLEX 11 e sua interface em C. As instâncias do problema SkPM incluem grafos gerados aleatoriamente e outros retirados do conjunto de testes contidos no DIMACS (do inglês *Second DIMACS Implementation Challenge for Cliques, Coloring, e Satisfiability*) [127].

Uma medida que ocupa lugar de destaque em aplicações usuais do CCA encontradas na literatura é a sobrecarga que o uso da plataforma CCA impõe. Nessas aplicações é prática corrente exigir que essa sobrecarga seja muito pequena, não devendo ultrapassar 5%, para que os aprimoramentos trazidos pelo uso de componentes sejam considerados vantajosos. Trata-se de uma medida facilmente mensurável, visto que a conversão em componentes não traz mudanças aos algoritmos ou métodos de resolução. Vale ressaltar que esse limite é difícil de ser respeitado em muitos casos dado o peso que as comunicações associadas às conexões entre componentes possuem. Quando passamos a observar as aplicações consideradas nesta tese, essa questão precisa ser relativizada, mesmo considerando que ainda está sujeita a uma certa controvérsia. O desempenho, nesse contexto de aplicações de otimização combinatória, continua sendo uma preocupação maior. No entanto, há outros aspectos que se ressaltam, ganhando importância relativa.

Neste contexto, o desafio de desempenho continua centrado nas implementações dos componentes, ou seja, nos algoritmos. Tomando o caso das implementações descritas neste estudo de caso, conforme a Figura 6.3, podemos identificar os pontos em que a plataforma pode trazer alguma sobrecarga, que são as conexões. Várias dessas conexões são efetuadas por conectores diretos, além de serem usadas em situações em que há algum processamento interno a um componente cujo tempo de execução é muito superior ao tempo de uso da conexão. O exemplo mais claro é uma conexão com uma porta do tipo `LPSolver` do `B&CComponent`, na qual a execução do método `solve` é dominante. Ainda nessa mesma implementação, é preciso mencionar o comportamento do conector exógeno. A sobrecarga imposta por esse conector nas interações entre `SchedulerComponent` e `B&CComponent` é devida ao escalonamento das *threads* que ele contém, que no entanto é imposta pelo paralelismo.

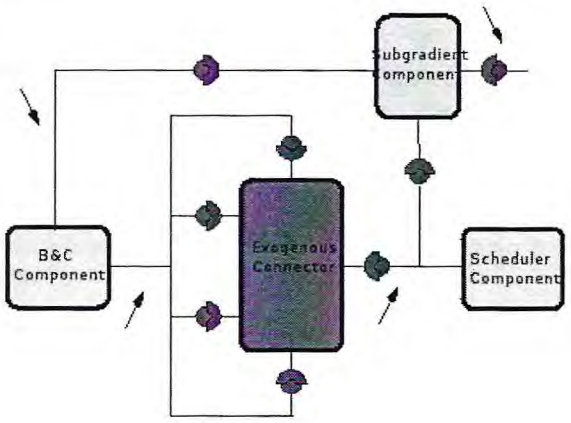


Figura 6.3: Pontos que a plataforma pode trazer alguma sobrecarga.

6.3 Estudo de Caso: Equações Lineares

Este estudo de caso é um cenário de métodos de solução de equações lineares representando um exemplo simples de código nativo.

6.3.1 Descrição

Como exemplo de código nativo, escolhemos duas classes nativas para solução de equações lineares: uma classe que encapsula o método Gauss-Seidel [128, pág.103] e outra encapsula o método Jacobi [128, pág.103]. O método de Gauss-Seidel é um método iterativo para resolução de sistemas de equações lineares. É semelhante ao método de Jacobi (e como tal, obedece ao mesmo critério de convergência). Para convergir, uma condição suficiente é que a matriz seja estritamente diagonal dominante. Com isto, fica garantida a convergência da sucessão de valores gerados para a solução exata de um sistema linear.

6.3.2 Desenvolvimento da Aplicação

O desenvolvimento dos estudos de caso envolveu três etapas principais (modelagem, implementação e execução da aplicação), em diferentes áreas de conhecimento, que abrangem as funcionalidades do *Forró*, descritas no Capítulo 5.

Modelagem

Esta etapa de modelagem de componentes e classes para o ambiente *Forró* foi desenvolvida com a ajuda dos exemplos CCA. O modelo foi construído no *software* Eclipse. Esta etapa abrange a seleção de classes e componentes. A seleção de classes e componentes consistiu na modelagem de classes nativas e componentes encapsuladores, citado no Capítulo 5, que contém os métodos nativos e os disponibiliza para outros componentes *Forró*.

As classes também possuem outros métodos. Para executar esta etapa, o programador deve ter um conhecimento das principais classes e de como são os componentes no ambiente *Forró*. A Figura 6.4 apresenta a representação do conteúdo das classes `C1` e `C2` e componentes encapsuladores `ComponentExample1` e `ComponentExample2`, incluindo a *interface* `Component` própria do CCA.

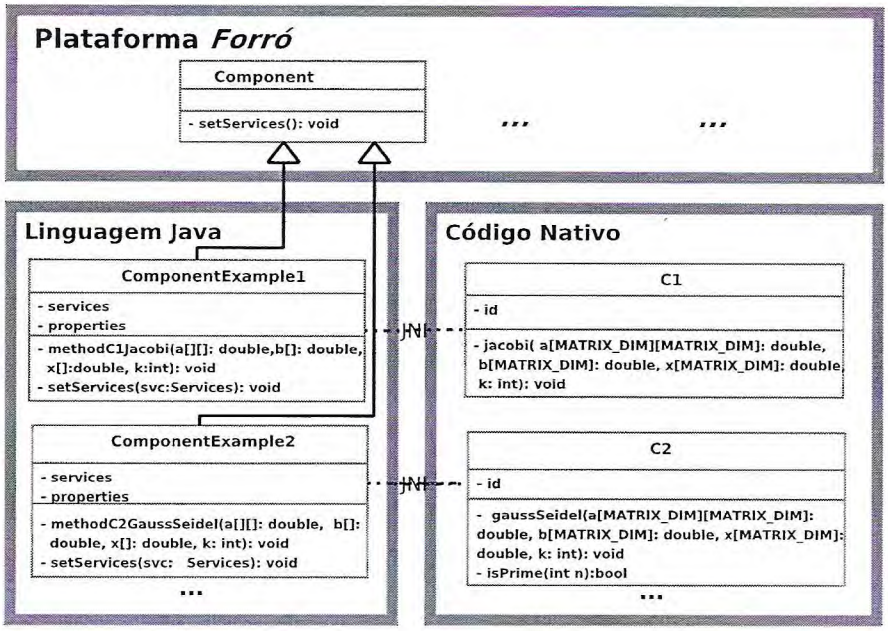


Figura 6.4: Classes e componentes do Estudo de Caso 1 - Equações Lineares.

Para compor o cenário da aplicação nativa, também foram utilizadas as seguintes classes: `C3`, para executar os métodos e imprimir a equação e as soluções e `ForroDriverNativo`, para gerenciar comunicações remotas.

Após a modelagem das classes e componentes, foi feita a implementação das mesmas, a qual está descrita na Subseção 6.3.2.

Implementação

Para implementar uma aplicação para funcionar no ambiente *Forró*, o usuário necessita apenas registrar/adicionar as portas *uses* e *provides* com as configurações que deseja utilizar. É importante ressaltar que o *Forró* não oferece suporte ainda à SIDL.

Nesta etapa de implementação de componentes e classes para o ambiente *Forró*, trabalhamos na otimização das classes e dos componentes para não sobrecarregar o código e a plataforma *Forró*, tentando ser eficiente.

Esta etapa foi desenvolvida também no Eclipse. O código nativo, após ter sido modelado no Eclipse, foi salvo no formato ".cpp", pois usamos a linguagem C++ como exemplo de linguagem nativa. A linguagem C++ foi escolhida por ser também uma linguagem orientada a objetos por ser eficiente.

Conforme visto no Capítulo 5, o grande benefício da utilização do *Forró* com relação a esta etapa é a capacidade de se poder utilizar diferentes linguagens para interação com a plataforma. Como citado anteriormente, a linguagem padrão da plataforma é Java, porém a plataforma interage com diferentes linguagens. Desta forma, os exemplos C++ escolhidos tem o objetivo de representar esta liberdade. A modelagem do código foi realizada em paralelo ao desenvolvimento da classe nativa *Forró* que encapsula a plataforma do lado nativo.

Para implementação da aplicação nativa no ambiente *Forró* a partir dos métodos, conforme apresentado na Figura 6.4, utilizamos a classe `C1`, responsável pelo método de Jacobi e a classe `C2`, responsável pelo método de Gauss-Seidel.

Entre as vantagens da plataforma *Forró* estão a reutilização de código com o objetivo de um aumento de produtividade para os desenvolvedores. Para isso, a plataforma requer a implementação de componentes encapsuladores, que disponibilizam as propriedades específicas e os métodos do código nativo para outros componentes. Através desse mecanismo, os métodos e as propriedades são disponibilizados para o desenvolvimento de aplicações com poucas linhas de código, conforme o trecho de código apresentado nas Tabelas 6.1. O conector usado foi o conector endógeno.

Tabela 6.1: Trecho de código do componente encapsulador `ComponentExample1` no *Forró*.

```
public class ComponentExample1 extends JNINativeComponent1Port
        implements Component {
        Services services;
        ...
        /*** Método Nativo escrito em C++ ***/
        static native int lineq_jacobi(double a[][],double b[],double x[],int k);
        static { System.loadLibrary("C1"); }
        public void setServices(Services s) throws CCAException { ... }
        private final class MyPort implements GoPort { ... }
        ...
}
```

O código apresentado na Tabela 6.1 demonstra o ganho de produtividade obtido com o uso da classe nativa, ao implementar as funcionalidades a serem disponibilizadas para outros componentes.

Execução

Conforme citado no Capítulo 5, para executar uma aplicação no ambiente *Forró*, definimos o comportamento dos componentes e a composição da rede. Também conforme citado no Capítulo 5, a execução no ambiente *Forró* é dividida em duas fases: preparação do ambiente e execução. A fase de preparação deste estudo de caso foi feita usando comandos do Linux na máquina local e na máquina remota. Estes comandos podem ser vistos nas Tabela 6.2 e 6.3.

Tabela 6.2: Comandos de preparação e inicialização do ambiente local.

killall rmiregistry
cd /home/gisele/Gisele/doctimp/nativeforro/ports g++ -g -fPIC -combine ../framework/Port.cpp ../framework/ForroC.cpp ../framework /ForroCDriver.cpp C2.cpp C3.cpp Component2Port.cpp -shared C1.cpp -o libC1.so g++ -fPIC -shared C2.cpp -o libC2.so g++ -g -fPIC -combine -shared C3.cpp -o libC3.so mv *.so /home/gisele/Gisele/doctimp/Forro/class cd /home/gisele/Gisele/doctimp/Forro/class
export LD_LIBRARY_PATH=/home/gisele/Gisele/doctimp/Forro/class export PATH=$PATH:/home/gisele/Gisele/doctimp/Forro/class:/home/gisele/Gisele/doctimp /Forro/class/forrocore
rmic -d /home/gisele/Gisele/doctimp/Forro/class -classpath /home/gisele/Gisele/doctimp /Forro/class framework.ForroDriver rmiregistry &
java -classpath '/home/gisele/Gisele/doctimp/jar/rmi.jar:/home/gisele/Gisele/doctimp/Forro /class' -Djava.rmi.server.codebase=file:/home/gisele/Gisele/doctimp/Forro/class/ -Djava.security.policy=file:/home/gisele/Gisele/doctimp/Forro/permission.policy framework.ForroDriver -Djava.net.preferIPv4Stack=true

Tabela 6.3: Comandos de preparação e inicialização.

(a) *Host* Local

Comandos Locais
scp -r /home/gisele/Gisele/doctimp/Forro gisele200.19.177.58:/home/gisele ssh 200.19.177.58

(b) *Host* Remoto

Comandos Remotos
killall rmiregistry rmiregistry &
export LD_LIBRARY_PATH=/home/gisele/Forro/class export PATH=$PATH:/home/gisele/Forro/class:/home/gisele/Forro/class /forrocore cd /home/gisele/Forro/class
java -classpath '/home/gisele/jar/rmi.jar:/home/gisele/Forro/class' -Djava.rmi.server.codebase=file:/home/gisele/Forro/class/ -Djava.security.policy=file:/home/gisele/Forro/permission.policy framework.ForroDriver -Djava.net.preferIPv4Stack=true

Preparado o ambiente, é preciso estabelecer os comandos para configurar o ambiente e a conectividade entre as instâncias dos componentes. Como citado, estes comandos

Tabela 6.4: Comandos de execução no ambiente *Forró*.

repository init localhost localsession
repository init 200.19.177.58 localsession
create component components.ComponentExample1 comp1 localhost
create component components.UsesPortComponent1 usesportcomp1 localhost
create component components.ProvidesPortComponent2 providesportcomp2 200.19.177.58
create component components.ComponentExample2 comp2 200.19.177.58
create connectingspace framework.BlockingConnectingSpaceDriver connectingspace1 localhost rmi
create connectingspace framework.BlockingConnectingSpaceDriver connectingspace1 200.19.177.58 rmi
connect localsession:localhost/comp1 comp1ToUsesPortComp1UsesPort usesportcomp1 usesPortComp1ProvidesPort
connect localsession:localhost/usesportcomp1 usesPortComp1ToConnectingSpace1UsesPort connectingspace1 connectingspace1ProvidesPort
connect localsession:localhost/connectingspace1 connectingspace1ToConnectingSpace2UsesPort localsession:200.19.177.58/connectingspace1 connectingspace2ProvidesPort
connect localsession:200.19.177.58/connectingspace1 connectingspace2ToProvidePortComp2 UsesPort localsession:200.19.177.58/providesportcomp2 providesPortComp2ProvidesPort
connect localsession:200.19.177.58/providesportcomp2 providesPortComp2ToComp2UsesPort localsession:200.19.177.58/comp2 comp2ProvidesPort
link novolink connectingspace1 200.19.177.58 localhost
connect localsession:localhost/usesportcomp1 usesPortComp1ToConnectingSpace1UsesPort localsession:200.19.177.58/providesportcomp2 providesPortComp2ProvidesPort
go comp1 sayHello1

seguem o padrão CCA. Para executar a aplicação deste estudo de caso foram usados os comandos listados na Tabela 6.4.

Nos comandos da Tabela 6.4 são criados: dois repositórios, um na máquina local e um na máquina com número IP `200.19.177.58`; quatro componentes, duas instâncias locais e duas instâncias remotas (onde uma instância local e uma remota são componentes que fazem parte de uma conexão indireta): dois espaços de conexão do tipo RMI: conexões entre os componentes: um `link` entre as duas máquinas; e finalmente, a execução da porta `GoPort` da instância `comp1` do componente `ComponentExample1`. .

Os comandos apresentados na Tabela 6.4 criam uma aplicação distribuída composta por dois componentes: um local, chamado `ComponentExample1` e um componente remoto, o `ComponentExample2`. As principais vantagens do uso do ambiente *Forró* nesta aplicação foram a distribuição dos componentes e a composição das conexões feitas pelo usuário. Além disso, existe a vantagem nas conexões do *Forró* com a sobreposição de computação e comunicação.

6.4 Estudo de Caso: Simfra

As principais motivações da escolha do Simfra como estudo de caso foram a necessidade de alto desempenho, multidisciplinaridade, códigos legados extensos e a necessidade de *hardware* específico.

6.4.1 Descrição

O Simfra é um sistema computacional para análise de integridade estrutural de dutos e pavimentos integrando diferentes unidades especializadas (instrumentação, computação gráfica e processamento de alto desempenho) distribuídas geograficamente. A aplicação Simfra surgiu de um projeto intitulado "Simulação e Visualização de Fraturas em Dutos e Materiais Betuminosos" [129, 130]. A idéia geral do projeto foi criar um sistema computacional integrando a criação de modelos de dutos e pavimentos, a simulação do comportamento estrutural desses modelos e o processo de visualização dessas simulações em um ambiente gráfico interativo. O sistema oriundo do projeto citado, o Simfra, é um pré-processador e pós-processador que pode ser integrado com diversos programas de análise de elementos finitos existentes (*SADISTIC*, *Finite Element Method - Object Oriented Programming* (FEMOOP), por exemplo).

O Simfra também pode ser definido como um ambiente gráfico para a criação de dutos e pavimentos e visualização de resultados de análise estrutural desses modelos, baseados em técnicas de visualização científica. O uso de técnicas de visualização permite uma melhor compreensão e interpretação dos resultados dessas simulações, que são fundamentais para todas as áreas envolvidas com petróleo.

Resumindo, o Simfra desenvolve uma metodologia para prever o comportamento estrutural com o cálculo de tensões e deslocamentos, ou seja, a evolução de trincas em meios viscoelásticos usando um código de Elementos Finitos.

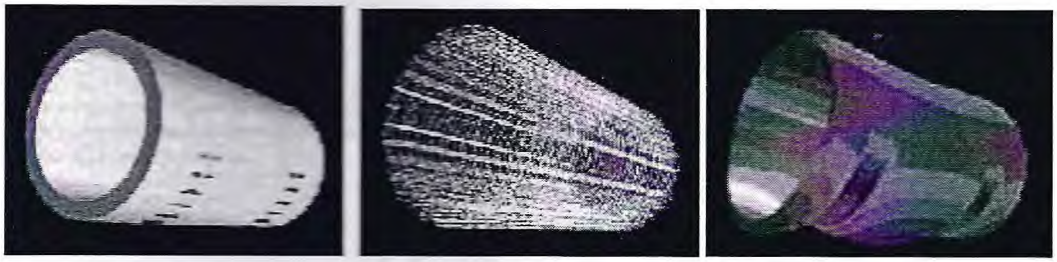
Nesse contexto, o Simfra envolve as seguintes etapas (ver Figura 6.5):

- Pré-processamento, onde são criados, as visões tridimensionais de dutos e pavimentos. A fase de pré-processamento que engloba a criação da visão geométrica, a definição das condições de contorno e definição de materiais, a comunicação com o programa de análise pelo Método de Elementos Finitos (MEF);
- Análise pelo MEF;
- Pós-processamento, no qual os resultados são avaliados.

O Simfra resolve problemas no estudo de determinados fenômenos por meio de mecânica computacional no estudo de Fratura em Dutos e Pavimentos. Para resolver os problemas citados, o Simfra possui três visões, são elas: geométrica, discreta e visual (ver Figura 6.6).

As restrições destas visões são: alto desempenho, alta heterogeneidade, reuso de código legado e múltiplos níveis de interação de software.

Figura 6.5: Ambiente integrado para análise de elementos finitos.



(a) Visão Geométrica. (b) Visão Discreta. (c) Visão Visual.

Figura 6.6: Tipos de visões do Simfra.

6.4.2 Desenvolvimento da Aplicação

Modelagem

A modelagem do Simfra, como nas Equações Lineares, foi realizada no Eclipse. A grande diferença desta etapa para a modelagem das Equações Lineares é o fato de que o Simfra era uma aplicação complexa e não apenas métodos separados, sendo dezenas de métodos funcionando em conjunto que necessitariam ser divididos e representados por componentes encapsuladores.

Analisamos as principais classes, indicadas pelos especialistas, do Simfra para avaliar o que seria importante para ser transformada em componente e quais seriam as portas dos componentes escolhidos. A aplicação Simfra é implementada em C++ e utiliza rotinas gráficas do *Open Graphics Library* (OpenGL) juntamente com a biblioteca multi-plataforma Fox para a criação da interface gráfica. O Simfra possui 45 classes de implementação. Após a avaliação de um dos integrantes da equipe de Computação Gráfica responsável pelo Simfra, selecionamos as seguintes classes candidatas: `SIMFRA3D`, `FEM` e `MarchingCubes`. Com a ajuda dos especialistas em Computação Gráfica, escolhemos as classes e o encapsulamento no *Forró* poderiam ser distribuídos e/ou paralelizados. Entre as classes candidatas foi selecionada a classe `MarchingCubes` para ser transformada em componente. O critério de escolha foi a quantidade e a complexidade dos parâmetros dos métodos a serem disponibilizados. A classe `MarchingCubes` apresentou o menor número

e a menor complexidade dos parâmetros dos métodos a serem disponiblizados. A classe `MarchingCubes` atua na etapa de pós-processamento. Nessa etapa final da simulação, as informações numéricas de deslocamentos e tensões vindas do programa de análise ganham um caráter visual no Simfra.

Conforme mencionado, o restante da aplicação foi encapsulado em um componente chamado `Simfra` (ver Figura 6.7).

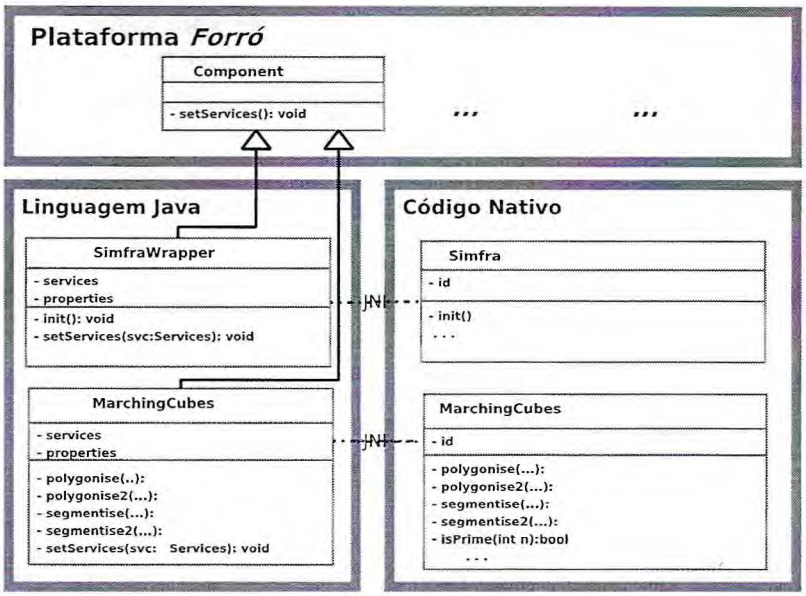


Figura 6.7: Classes e componentes do Estudo de Caso 2 - Simfra.

Implementação

As classes e os componentes também foram simplificados para não sobrecarregar o modelo, porém, tentando mais uma vez ser eficiente. A classe `MarchingCubes` foi encapsulada no componente `MarchingCubes` como mostra a Tabela 6.5.

Tabela 6.5: Componente e porta Java `MarchingCubes` que encapsula o código C++ do Simfra.

```
// Componente que encapsula as funcionalidades do algoritmo Marching Cubes.
public class MarchingCubes implements Component. Port {
    ...
    public static native Vertex VertexInterp(float isolevel, Vertex p1,Vertex p2,
        float valp1, float valp2);
    public static native Vertex VertexInterp2(float isolevel,int p1,int p2,
        float valp1. float valp2. float[] valinterp, GlobalObjects[] obj);
    public static native int Polygonise(Vertex n[], float t[]. float isolevel.
        TRIANGLE[] triangles);
    public static native int Polygonise2(int n[],float t[],float isolevel.
        TRIANGLE[] triangles.GlobalObjects[] obj);
    public static native int Segmentise(int n[], float t[], float isolevel,
        GlobalObjects[] obj);
    public static native int Segmentise2(int n[], float t[], float isolevel,
        GlobalObjects[] obj);
    ...
    public static native public void setServices(Services s) throws CCAException {...}
    ...
};
```

Os métodos `Polygonise`, `Polygonise2`, `Segmentise` e `Segmentise2` são os métodos que foram selecionados para serem disponibilizados na porta *provides* do componente `MarchingCubes`, pois, são os métodos que eram chamados por outras classes. Então, foi feito outro arquivo *header* (arquivo que contém os protótipos dos métodos) para redirecionar os métodos da classe para os métodos do componente `MarchingCubes`. Também foi modificado o código do`Simfra` para que no momento da criação da aplicação Simfra fossem criados os componentes `MarchingCubes` e `Simfra`.

Execução

Conforme citado no Capítulo 5, a fase de preparação do estudo de caso Simfra foi feita usando comandos do Linux na máquina local e na máquina remota. Estes comandos podem ser vistos na Tabela 6.6 e na Tabela 6.3, mostrada neste estudo de caso.

Tabela 6.6: Comandos de execução no ambiente *Forró*.

killall rmiregistry
cd /home/gisele/Gisele/doctimp/SIMFRA_APP
g++ -g -fPIC -combine Vertex.cpp Node.cpp -shared Simfra.cpp -o libSimfra.so
g++ -g -fPIC -combine Vertex.cpp Node.cpp -shared marchingCubes.cpp -o libmarchingCubes.so
mv *.so /home/gisele/Gisele/doctimp/Forro/class
cd /home/gisele/Gisele/doctimp/Forro/class
export LD_LIBRARY_PATH=/home/gisele/Gisele/doctimp/Forro/class
export PATH=$PATH:/home/gisele/Gisele/doctimp/Forro/class:/home/gisele/Gisele/doctimp/Forro/class/forrocore
rmic -d /home/gisele/Gisele/doctimp/Forro/class -classpath /home/gisele/Gisele/doctimp/Forro/class framework.ForroDriver
rmiregistry &
java -classpath '/home/gisele/Gisele/doctimp/jar/rmi.jar:/home/gisele/Gisele/doctimp/Forro/class' -Djava.rmi.server.codebase=file:/home/gisele/Gisele/doctimp/Forro/class/ -Djava.security.policy=file:/home/gisele/Gisele/doctimp/Forro/permission.policy framework.ForroDriver -Djava.net.preferIPv4Stack=true

6.5 Estudo de Caso: Ordenação de Inteiros

A principal motivação da escolha de uma aplicação de referência (ou em inglês *benchmark*) foi a necessidade de avaliação do peso do *Forró*. Além disso, usamos o NetPipe para testar o ambiente para assegurar a confiabilidade do ambiente de testes.

6.5.1 Descrição do Problema

O programa de Simulação Numérica Aerodinâmica (do inglês *Numerical Aerodynamic Simulation* (NAS)), localizado no Centro de Pesquisa da NASA, é um descobridor de computação de alto desempenho para a NASA, com foco em dinâmicas de fluido computacional e disciplinas relacionadas com aerosciência [131]. Este programa criou um conjunto de aplicações de referência (ou em inglês *benchmark*) como um esforço em grande escala para o avanço do acrodinâmica computacional.

Este conjunto de aplicações de referência (do inglês *benchmarks*) foi desenvolvido para avaliar o desempenho de computadores paralelos. Estas aplicações de referência são compostas por cinco *benchmarks* de núcleos paralelos e três *benchmarks* de simulações. Os *benchmarks* de simulações combinam várias computações de uma forma que se assemelha a ordem real de execução de determinados códigos de aplicações importantes de dinâmicas de fluido computacional (do inglês *Computational Fluid Dynamics* (CFD)). Juntos, eles simulam a computação e as características de movimentação de dados em CFD de larga

escala.

A principal característica destas aplicações de referência é a sua especificação na qual todos os detalhes destas aplicações são especificadas apenas algoritmicamente. Desta forma, muitas das dificuldades associadas com abordagens tradicionais de aplicações de referência em sistemas altamente paralelos são evitados.

Definição

Desde as especificações de 1991 do *NAS Parallel Benchmarks* (NPB) 1.0, a velocidade computacional e os tamanhos da memória tem crescido e correspondentemente portanto os tamanho representativos do problema também. Muitas aplicações computacionais da aerociência rotineramente usa milhões de pontos de grade. NPB 1.0 especifica dois tamanhos de problemas para cada cada *benchmark* - classe "A" and a maior classe "B". Para enfatizar o foco na supercomputação, foi adicionada uma classe "C" para todos os *benchmarks* NAS.

Os tamanhos dos problemas das classes "A", "B" e "C" e os tipos de *benchmarks* são dadas na Tabela 6.7.

Tabela 6.7: Tamanhos de Problemas dos *Benchmarks* Paralelos NAS

Código *Benchmark*	Classe A	Classe B	Classe C
Embaralhamento Paralelo (EP)	2^{28}	2^{30}	2^{32}
Multigrade (MG)	256^3	256^3	512^3
Gradiente Conjugado (CG)	14000	75000	150000
3-D FFT PDE (FT)	$256^2 \times 128$	512×256^2	512^3
Ordenação de Inteiros (IS)	2^{23}	2^{25}	2^{27}
Resolvedor LU (LU)	64^3	102^3	162^3
Resolvedor Pentadiagonal (SP)	64^3	102^3	162^3
Resolvedor tridiagonal bloco (BT)	64^3	102^3	162^3

Ordenação de Inteiros

Este núcleo executa uma operação de ordenação de grandes números inteiros que é importante em alguns códigos de "métodos de partículas". Ele testa ambos a velocidade da computação inteira e o desempenho de comunicação.

O problema específico é gerar um grande vetor usando um determinado esquema e, em seguida, classificá-lo. Qualquer esquema de classificação eficiente paralelo pode ser usado. O teste de verificação é se certificar que o vetor está na ordem de classificação.

Este é um algoritmo de ordenação paralela execução do tipo simples "bucket". Este algoritmo possui três fases. Em primeiro lugar, os elementos são divididos e classificados no local. Em segundo lugar, os elementos são redistribuídos com base na escala utilizada pelo processador. Em terceiro lugar, os elementos que são redistribuídos são mesclados com os elementos atualmente residem no elemento de processamento.

A fase 1 da ordenação de inteiros envolve computação independente por cada um dos elementos de processamento. Se um dos elementos fosse medir o desempenho de apenas esta fase. seria notado que não deve haver quase nenhuma comunicação durante essa ordenação local.

Um dos principais gargalos no desempenho vem da sobrecarga de comunicação na fase 2. e este *benchmark* de ordenação paralela é muitas vezes considerado como um marco de referência para medir o desempenho de comunicação de uma máquina paralela. Durante esta fase 2, para cada subpartição da partição do processador, EPs enviar seções das sub-partições ordenadas que não pertencem ao seu conjunto para os conjuntos apropriados (de propriedade de outros EPs).

Na fase 3, cada conjunto contém apenas ainda seções ordenadas de elementos que estão dentro do intervalo de seu *bucket*, mas os elementos dentro de cada segmento não são ordenados. Assim, todos os EPs localmente reordenar seus elementos, fundindo suas seções ordenadas de elementos juntos. Isso é feito de forma independente por cada EP, normalmente através de um caminho k-direto. Formalmente, o algoritmo de ordenação *bucket* como implementado no *benchmarks* paralelos NAS pode ser descrita em três fases distintas:

1. Fase 1: Local de triagem inicial:
2. Fase 2: Redistribuição:
3. Fase 3: Merge local de partições recebido.

Esta implementação particular do tipo ordenação trabalha especificamente para um vetor de inteiros, embora outros tipos de dados também são possíveis. Este método de classificação não é balanceamento de carga (como é exemplo de classificação). e assim ele pode sofrer alguns problemas de escalabilidade. No entanto, quando benchmarking é importante para expor fraquezas em arquitetura por meio de testes de estresse com mal algoritmos escaláveis. Esta testes de *benchmark* tanto computação inteiro velocidade e desempenho de comunicação. Esse problema é único em que a aritmética de ponto flutuante não está envolvido. Entretanto, uma significativa de comunicação de dados é necessária.

NetPIPE

Conforme citado, o NetPIPE foi escolhido para testar o ambiente assegurando assim a confiabilidade do ambiente de testes. NetPIPE é um avaliador de desempenho independente de protocolo que mapeia o desempenho da rede através de uma larga faixa de valores e apresenta os resultados de uma nova maneira. O avaliador de desempenho de protocolo independente tem sido motivado pela necessidade de se medir os fatores limitantes da comunicação entre aplicações.

O NetPIPE consiste de duas partes: um configurador independente de protocolo e um protocolo de comunicação específico de sessão. A sessão de comunicação contém as

funções necessárias para estabelecer a conexão. Esta parte é diferente para cada protocolo. Entretanto, a *interface* entre o configurador e o módulo do protocolo permanece o mesmo.

O NetPIPE aumenta o bloco de transferência de tamanho k desde um simples *byte* até que o tempo de transmissão exceda 01 (um) segundo. Logo, o NetPIPE é um comparador de tempo variável e que será escalonado para qualquer velocidade da rede.

Para cada bloco de tamanho c três medidas são tomadas: $c-p$ bytes, c bytes e $c+p$ bytes, sendo p um fator de perturbação de valor 3. Este fator de perturbação permite a análise de tamanhos de bloco que são ligeiramente maiores ou menores do que o intervalo de *buffer* da rede.

O NetPIPE utiliza uma transferência *ping-pong* usada para cada bloco. Isto força a rede transmitir somente o bloco de mensagem sem enviar outros blocos de dados naquela mensagem. O resultado é o tempo de transferência de um único bloco, fornecendo a informação necessária para responder qual o melhor bloco, ou qual é a vazão dado um bloco de tamanho k.

O NetPIPE produz um arquivo que contém o tempo de transferência, vazão, tamanho do bloco, e a variância do tempo de transferência para cada ponto de dados e é facilmente plotado por qualquer pacote de gráficos. Por exemplo, a Figura 6.8 representa a vazão versus o tamanho do bloco de transferência. Facilmente se vê que a vazão máxima para esta rede é da ordem de 102 bps. Entretanto, é difícil analisar o atraso, que é uma estatística igualmente importante.

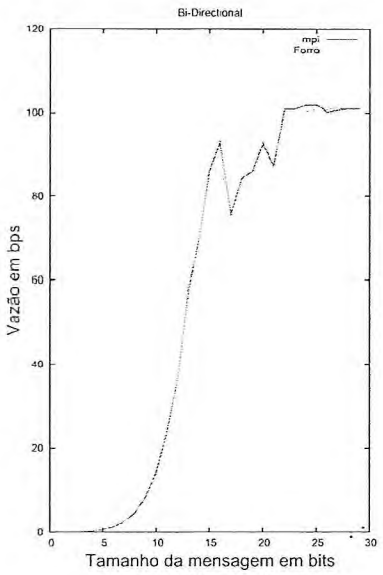


Figura 6.8: *Ping-pong* assíncrono.

Chamamos à atenção que a resposta do gráfico não é uma função do tempo unívoca. Esta característica não é uma anomalia, mas uma indicação de que uma mensagem grande pode, na verdade, levar a um menor tempo de transferência devido ao tamanho do *buffer*

do sistema e a iteração com o sistema operacional. Este fenômeno é repetitivo. Se cogita que em tal caso indica-se uma melhoria no sistema e *software* de mensagem, uma vez que um conjunto grande de tarefas sempre leva mais tempo que uma tarefa isoladamente.

O gráfico do tamanho das mensagens versus o tempo de transferência revela o que chamamos de assinatura do gráfico que podemos ver nas figuras 6.9 e 6.10.

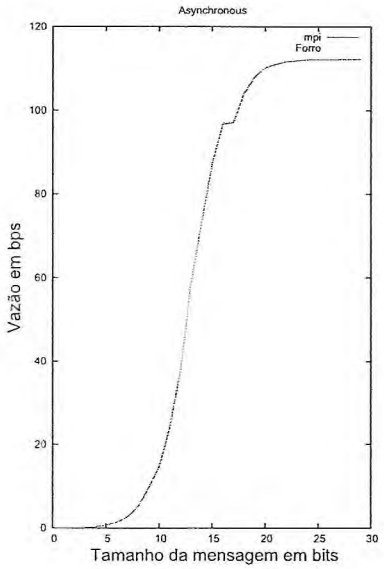


Figura 6.9: *Ping-pong* assíncrono.

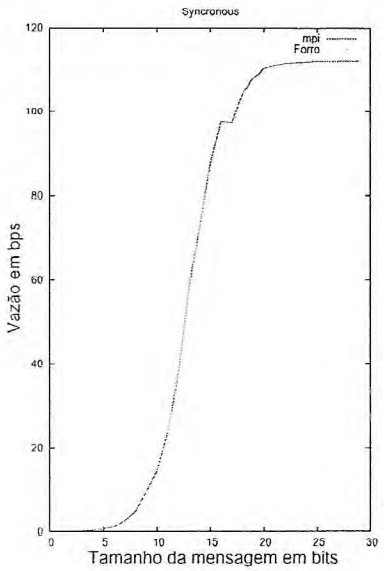


Figura 6.10: *Ping-pong* assíncrono.

6.5.2 Desenvolvimento da Aplicação

O desenvolvimento do estudo de caso envolveu três etapas principais (modelagem, implementação e execução da aplicação), em diferentes áreas de conhecimento, que abrangem as funcionalidades do *Forró*, descritas no Capítulo 5.

Execução

Alguns resultados desses experimentos são apresentados e analisados a seguir. O objetivo principal destes testes foi mostrar o peso do *Forró* em uma aplicação paralela e distribuída. A implementação utilizada nos experimentos inclui componentes CCA em Java, com os métodos implementados do *benchmark* NAS Ordenação de Inteiro na linguagem C.

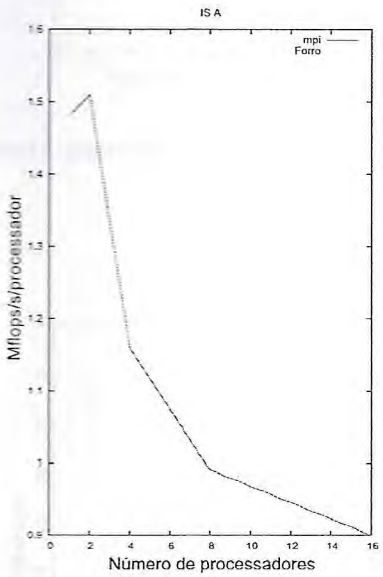


Figura 6.11: *Benchmark* paralelo de ordenação de inteiros na classe A.

Na figura 6.11, mostramos o desempenho da classe A da ordenação de inteiros (IS) do NAS variando o número de processadores em uma aplicação usando somente MPI e outra usando o *Forró*. A figura mostra uma pequena diferença entre o desempenho das aplicações, dando uma indicação do peso do *Forró*. Este desempenho é consistente com os resultados dos *benchmarks* também nas classes B e C como vemos nas figuras 6.12 e 6.13.

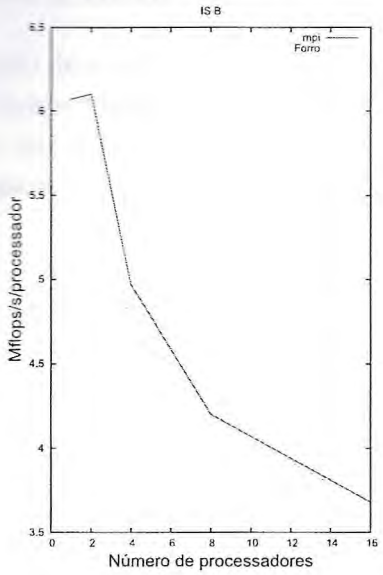


Figura 6.12: *Benchmark* paralelo de ordenação de inteiros na classe B.

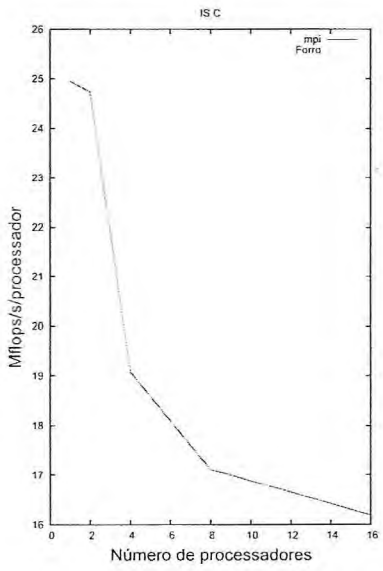


Figura 6.13: *Benchmark* paralelo de ordenação de inteiros na classe C.

Os valores máximos de Mflops/s/processadores obtidos foram:

- Sem o Forró: 25 Mflops/s na classe C do IS 1 processador:
- Com o Forró: 25.2 Mflops/s na classe C do IS 1 processador.

6.6 Considerações Finais

As execuções do primeiro estudo de caso foram bastante satisfatórias. Com os resultados obtidos, com destaque para o tempo médio de implementação e execução de uma aplicação distribuída é possível afirmar que a plataforma proposta pode apresentar-se como uma solução bastante interessante para composição de aplicações a partir de métodos ou código legado.

O segundo estudo de caso serviu de teste do *Forró* para o encapsulamento do código nativo. O terceiro estudo de caso foi uma boa experiência na fase de modelagem. Por último, o quarto estudo de caso garantiu usando o NetPIPE a confiabilidade do ambiente de testes e confirmou o desempenho de um *benchmark* conhecido mesmo com o uso do *Forró*, o que é um bom resultado para nossa plataforma. Portanto, podemos dizer que o primeiro e o quarto estudo de caso validaram o desempenho do uso da plataforma.

De uma maneira geral, podemos concluir que os estudos de caso confirmaram bons resultados com a plataforma *Forró*. Portanto, podemos dizer que os estudos de caso validaram a viabilidade do uso da plataforma *Forró*.

Capítulo 7

Conclusão

Neste capítulo são apresentadas as principais conclusões obtidas durante a elaboração desta tese. Na Seção 7.1, mostramos as principais contribuições do trabalho. Na Seção 7.2, listamos os produtos gerados com a presente tese. Por último, na Seção 7.3 apresentamos algumas possibilidades de trabalhos futuros.

7.1 Contribuições

A Computação de Alto Desempenho (CAD) está se tornando uma área cada vez mais importante para o desenvolvimento de *software* para vários ramos de atividades do conhecimento humano. A evolução de CAD está proporcionando o surgimento de novas aplicações, que advêm sobretudo da sinergia criada pela combinação do alto desempenho com novas tecnologias de armazenamento, de transmissão de dados e de processamento. Para que estas aplicações reutilizem códigos específicos existentes, esta área necessita da cooperação da área de Engenharia de *Software* para permitir que esta interação seja feita de forma a garantir o desempenho e a qualidade.

No contexto descrito, as contribuições do presente trabalho foram: proposta de conceitos ***link*** e ***espaço de conexão*** para componentes distribuídos; e, proposta de utilização de conectores endógenos, conectores exógenos e ***dutos*** no CCA, os quais ainda não tinham sido utilizados em nenhuma outra plataforma CCA.

Com relação aos estudos de caso, esta tese envolveu o estudo de caso de aplicações científicas distintas de CAD, uma voltada para a solução de equações lineares, outra voltada para a simulação de fraturas em dutos, oriunda de um projeto da equipe de Computação Gráfica, e por último, um estudo de problemas de otimização sobre conjuntos independentes em grafos. Inicialmente, o trabalho apresentou os benefícios e os requisitos de aplicações científicas. Em seguida, foi feito um levantamento de características comuns às aplicações de CAD no Capítulo 2. Da mesma forma, a partir dos trabalhos relacionados, também foram relacionados requisitos importantes presentes nas plataformas de *software* existentes no Capítulo 3.

Em seguida, este trabalho também apresentou benefícios da utilização de recursos de

CAD em soluções científicas. Levando-se em consideração as características identificadas nos Capítulos 2 e 3, esta tese definiu um conjunto de requisitos necessários a aplicações de CAD que uma plataforma de componentes deveria respeitar.

Posteriormente, foi definida uma plataforma de componentes orientada a objetos para auxiliar os desenvolvedores: o *Forró*. Entre outras vantagens, a plataforma atende os requisitos estabelecidos neste trabalho, padroniza a construção de aplicações a partir dele e proporciona ganho de produtividade na composição de aplicações com código em linguagens diferentes. A nossa contribuição com o *Forró* foi enriquecer plataformas existentes com o suporte a vários modelos de computação em uma única aplicação, endereçando o problema de alta heterogeneidade de arquiteturas distribuídas contemporâneas que tem emergido para aplicação no domínio científico com o uso de `espaços de conexão`.

A ideia principal é permitir a composição e configuração de novas aplicações a partir de um conjunto de componentes de classes novas ou existentes. O ambiente *Forró* apresenta uma arquitetura dividida em três camadas no modelo MVC. Os elementos do *Forró* abstraem detalhes específicos de implementação. Esta organização possibilita ainda um trabalho em equipe, com profissionais especialistas em diferentes áreas: *interface*, controle e domínio. A plataforma também incorporou outros padrões de projetos para solucionar problemas encontrados durante a codificação. Esta estrutura possibilita aos especialistas concentrarem-se somente na camada que tem mais afinidade, abstraindo-se dos detalhes da implementação e trabalhando na sua área.

Outro benefício da utilização do *Forró* está na sua simplicidade e compatibilidade com o modelo CCA, o que possibilita a utilização dos componentes *Forró* nas plataformas existentes.

Sobre a plataforma proposta, três estudos de caso foram estudados nesta tese: equações lineares, Simfra e problemas de otimização sobre conjuntos independentes em grafos. Através da comparação de suas implementações, observou-se a reutilização de código proporcionada, o que implicou uma redução do tempo de desenvolvimento. A capacidade de extensibilidade do *Forró*, também pode ser notada, o que permitiu a composição de aplicações diferentes com pouca ou nenhuma re-implementação.

Para finalizar a explanação sobre a plataforma *Forró*, podemos citar algumas características diferenciais:

- Possibilidade dos especialistas concentrarem-se somente na camada que tem mais afinidade;
- Simplicidade e compatibilidade com o modelo CCA;
- Extensibilidade de composição de aplicações diferentes.

7.2 Produtos Gerados

Em relação aos produtos gerados com a presente tese, podemos citar:

- Projeto de classes para compatibilidade com o CCA;
- Modelagem UML (do inglês *Unified Modeling Language*) da solução;
- Projeto e implementação dos elementos:
  - CCA *core*;
  - Componentes gerenciadores;
  - Ambiente nativo;
  - Conectores;
- Estudos de caso;
- Artigo publicados [21, 22, 23, 24].

7.3 Perspectivas e Trabalhos Futuros

A partir da limitação do *Forró* com relação à definição do paralelismo, surge a possibilidade de se aprimorar a plataforma através da implementação de classes que estabeleçam e configurem o paralelismo das aplicações. Desta forma, planeja-se proporcionar uma configuração mais específica da aplicação criada e, possivelmente, aumentar o desempenho.

Outro trabalho futuro é a tradução e a recuperação do *Scientific Interface Definition Language* (SIDL) para os componentes escritos em SIDL pudessem ser usados diretamente como componentes *Forró* e o próprio *Forró* gerasse componentes em SIDL. Assim, a interoperabilidade do *Forró* não estaria restrita a códigos escritos nas linguagens Java, C e C++, linguagens as quais o *Forró* tem compatibilidade atualmente.

Uma outra possibilidade de trabalho futuro é a adição de uma funcionalidade de descoberta de novos serviços e componentes. Nesta funcionalidade, o usuário indicaria as máquinas e a plataforma faria uma pesquisa para identificar serviços e componentes disponíveis na máquina, o desenvolvimento das aplicações dar-se-ia de forma mais rápida, fácil e integrada.

As aplicações desenvolvidas no ambiente *Forró* são independentes e funcionam isoladamente. Porém, ao se imaginar aplicações como componentes compostos, então um possível trabalho futuro seria possibilitar a integração de aplicações *Forró* a partir da combinações de outras aplicações *Forró*. Estes sistemas conjugariam funcionalidades de comunicação mediada pela plataforma e as formas de distribuição seria configurável, visando prover integrações entre aplicações que facilitariam a troca de informações entre áreas distintas. Assim, é preciso investigar como seriam componentes compostos sugeridos no modelo Fractal [108] para possibilitar que integrações de componentes com um nível maior de abstração.

Outro trabalho futuro, pensado a partir dos problemas encontrados, seria a definição de uma ferramenta gráfica para composição de aplicações e conexão de componentes.

Esta teria a função de identificar problemas como caminhos críticos, por exemplo. Desta forma, seria evitado que o problema ocorresse durante a execução da aplicação.

Por fim, sugere-se o desenvolvimento de um estudo de caso com um alto volume de comunicação, dados e computação para ser validado junto às equipes a eficiência da plataforma. Neste trabalho, seria necessário o envolvimento de uma equipe específica de acordo com a aplicação alvo escolhida. Um exemplo multidisciplinar que poderia ser testado seria uma aplicação em Bioinformática como o trabalho [132].

Referências Bibliográficas

[1] H. H. Goldstine, *The computer from Pascal to von Neumann.* Princeton, NJ, USA: Princeton University Press, 1980.

[2] C. Eames e R. Eames, *A computer perspective: background to the computer age (new ed.).* Cambridge, MA, USA: Harvard University, 1990.

[3] M. Boden, *Mind as Machine: A History of Cognitive Science.* New York, NY, USA: Oxford University Press, Inc., 2008.

[4] K. Flamm, *Creating the computer: government, industry, and high technology.* Washington, DC, USA: The Brookings Institution, 1988.

[5] P. E. Ceruzzi, *A history of modern computing.* Cambridge, MA, USA: MIT Press, 1998.

[6] K. Dowd e C. Severance, *High Performance Computing.* M. Loukides, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1998.

[7] L. T. Yang e M. Guo, *High-Performance Computing: Paradigm and Infrastructure*, A. Y. Zomaya, Ed. Hoboken, NJ, USA: Wiley-Interscience, 1998.

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, e K. A. Yelick. "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[9] G. Coulouris, J. Dollimore, e T. Kindberg, *Distributed Systems: Concepts and Design.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994.

[10] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, e B. Smolinski, "Toward a common component architecture for high-performance scientific computing," in *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing.* Washington, DC, USA: IEEE Computer Society, 1999, p. 13.

[11] A. M. Law e W. D. Kelton, *Simulation Modeling and Analysis*. MC Graw Hill Science/Engineering/Math, 2000.

[12] G. Eisenhauer, F. E. Bustamante, e K. Schwan, "Event services for high performance computing," in *HPDC '00: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2000, p. 113.

[13] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K.Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, J. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. Mcinnes, J. Nieplocha, B. Norris, J. Ray, T. L. Windus, e S. Zhou, "A component architecture for high-performance scientific computing," *Intl. J. High-Performance Computing Applications*, vol. 20, 2004.

[14] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, e M. V. Zelkowitz, "Understanding the high-performance-computing community: A software engineer's perspective," *IEEE Softw.*, vol. 25, no. 4, pp. 29–36, 2008.

[15] C. C. A. Group. "The common component architecture forum. http://www.cca-forum.org/docs/frameworks.html," 2004.

[16] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. E. Miller e J. W. Thatcher, Eds. Plenum Press, 1972, pp. 85–103.

[17] W. Cook, W. Cunningham, W. Pulleyblank, e A. Schrijver. *Combinatorial optimization*. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[18] D. Avis, A. Hertz, e O. Marcotte, *Graph Theory and Combinatorial Optimization*. Dordrecht: Springer-Verlag New York Inc, 2006.

[19] S. G. Parker, "A component-based architecture for parallel multi-physics pde simulation," *Future Gener. Comput. Syst.*, vol. 22, no. 1, pp. 204–216, 2006.

[20] T. Goodale, G. Allen, G. Lanfermann, J. Mass, T. Radke, E. Seidel, e J. Shalf, "The cactus framework and toolkit: Design and applications," in *High Performance Computing for Computational Science - VECPAR 2002*. Springer, 2003, pp. 15–36.

[21] F. H. Carvalho-Junior, R. D. Lins, R. C. Corrêa, e G. A. Araújo, "Towards an architecture for component-oriented parallel programming," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 5, pp. 697–719, 2007.

[22] F. H. Carvalho-Junior, R. C. Corrêa, G. A. Araújo, J. C. Silva, e R. D. Lins, "High-level service connectors for component-based high performance computing," *Computer Architecture and High Performance Computing, Symposium on*, vol. 0, pp. 237–244, 2007.

[23] F. H. Carvalho-Junior, R. D. Lins, R. C. Corrêa, G. A. Araújo, e J. C. Silva, "On the design of abstract binding connectors for high performance computing component models," in *CompFrame '07: Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing*. New York, NY, USA: ACM, 2007, pp. 67–76.

[24] G. Araújo, F. Carvalho-Jr, e R. Corrêa, "Implementing endogenous and exogenous connectors with the common component architecture," in *CBHPC '09: Proceedings of the 2009 Workshop on Component-Based High Performance Computing*. New York, NY, USA: ACM, 2009, pp. 1–4.

[25] J. Bigot, H. Bouzianc, C. Pérez, e T. Priol, "On abstractions of software component models for scientific applications," in *Proc. of the Abstractions for Distributed Systems workshop*, Las Palmas, Gran Canaria, Spain, 2008.

[26] B. Allan e R. Armstrong, "Ccaffeine framework: Composing and debugging applications iteratively and running them statically. compframe 2005 workshop," 2005.

[27] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, e C. Pérez, "Gcm: a grid extension to fractal for autonomous distributed components," *Annales des Télécommunications*, vol. 64, no. 1-2, pp. 5–24, 2009.

[28] W. Wolf, *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[29] M. D. Smith, M. Johnson, e M. A. Horowitz, "Limits on multiple instruction issue," *SIGARCH Comput. Archit. News*, vol. 17, no. 2, pp. 290–302, 1989.

[30] E. E. Johnson, "Completing an mimd multiprocessor taxonomy," *SIGARCH Comput. Archit. News*, vol. 16, no. 3, pp. 44–47, 1988.

[31] R. S. Morrison, *Cluster Computing Architectures, Operating Systems, Parallel Processing & Programming Languages*. GNU General Public Licence, 2003.

[32] M. J. Flynn e K. W. Rudd, "Parallel architectures," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 67–70, 1996.

[33] M. Baker e R. Buyya, "Cluster computing: The commodity supercomputing," pp. 1–4.

[34] J. Wu, *Distributed System Design*. Boca Raton, FL, USA: CRC Press, Inc., 1998.

[35] H.E.-Rewini e M.A.-E.-Barr, *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2005.

[36] M. V. Wilkes, "Slave memories and dynamic storage allocation," pp. 270–271, 1965.

[37] D. Culler, A. Gupta, e J. P. Singh, *Parallel Computer Architecture: A Hardware-Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann, August 1997.

[38] S. S. S.C. Yadav, *An Introduction to Client/Server Computing*. Sri Lanka, India: New Age International, 2009.

[39] K. Ravindran, "Reliable client-server communication in distributed programs," Ph.D. dissertation, 1987, supervisor-Chanson, Samuel T.

[40] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, e Z. Xu, "Peer-to-peer computing," HP Laboratories Palo Alto HPL-2002-57, Tech. Rep., 2002.

[41] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[42] M. S. Bakshi e S. R. Arora, "The sequencing problem," Management Science - Application Series, pp. B247–B263, December 1969.

[43] F. B. Schneider, "Synchronization in distributed programs." *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 2, pp. 125–148, 1982.

[44] A. Grama, G. Karypis, V. Kumar, e A. Gupta, *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, January 2003.

[45] A. J. Wells, *Grid Application Systems Design*. Boca Raton, FL, USA: CRC Press, 2007.

[46] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[47] W. Gropp, E. Lusk, N. Doss, e A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, 1996.

[48] K. Birman, T. Joseph, T. Raeuchle, e A. E. Abbadi, "Implementing fault-tolerant distributed objects," Ithaca, NY, USA, Tech. Rep., 1984.

[49] W. Emmerich e S. Tai, Eds., *Engineering Distributed Objects, Second International Workshop, EDO 2000, Davis, CA, USA, November 2-3, 2000, Revised Papers*, ser. Lecture Notes in Computer Science, vol. 1999. Springer, 2001.

[50] S. Sechrest, "An introductory 4.4bsd interprocess communication tutorial." In 4.4BSD Programmer's Supplementary Documents, chapter 20, Tech. Rep., 1994.

[51] E. H. Ruspini, "A new approach to clustering," *Information and Control*, vol. 15, no. 1, pp. 22 – 32, 1969.

[52] T. Sterling, D. Becker, e D. F. Savarese, *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters (Scientific and Engineering Computation)*. The MIT Press, May 1999.

[53] M. Baker, "Cluster computing white paper." *CoRR*, vol. cs.DC/0004014, 2000.

[54] A. D. Birrell e B. J. Nelson, "Implementing remote procedure calls," *SIGOPS Oper. Syst. Rev.*, vol. 17, no. 5, p. 3, 1983.

[55] Z. Tari e O. Bukhres, *Fundamentals of distributed object systems: the CORBA perspective*. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[56] F. Berman, G. Fox, e T. Hey, *Grid Computing: Making the Global Infrastructure a Reality*, T. Hey, Ed. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[57] C. Kesselman e I. Foster, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[58] D. L. Eager, J. Zahorjan, e E. D. Lozowska, "Speedup versus efficiency in parallel systems," *IEEE Trans. Comput.*, vol. 38, no. 3, pp. 408–423, 1989.

[59] M. Minsky e S. Papert, "On some associative, parallel, and analog computations," *Associative Information Technologies, American Elsevier North Holand, New York*, 1971.

[60] D. J. K. et al., "The effects of program restructuring, algorithm change and architecture choice on program parallelism." pp. 129–138, 1984.

[61] J. Sametinger, *Software engineering with reusable components*. New York, NY, USA: Springer-Verlag New York, Inc., 1997.

[62] I. Crnkovic, H. Schmidt, J. Stafford, e K. Wallnau, "Anatomy of a research project in predictable assembly," in *5th ICSE Workshop on Component Based Software Engineering, ACM, May*, 2002.

[63] N. Mehta, N. Medvidovic, e S. Phadke, "Towards a taxonomy of software connectors," in *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA: ACM Press, 2000, pp. 178–187.

[64] A. Smeda, M. Oussalah, e T. Khammaci, "Improving Component-Based Software Architecture by Separating Computations from Interactions," in *Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04)*, Oslo, Norway, 2004, in conjunction with ECOOP 2004.

[65] S. Matougui e A. Beugnard, "Two ways of implementing software connections among distributed components," in *OTM Conferences (2)*, ser. Lecture Notes in Computer Science, R. Meersman, Z. Tari, M. Hacid, J. Mylopoulos, B. Pernici, Ö. Babaoglu, H. Jacobsen, J. Loyall, M. Kifer, e S. Spaccapietra, Eds., vol. 3761. Springer, 2005, pp. 997–1014.

[66] K.-K. Lau, M. Ornaghi, e Z. Wang, "A software component model and its preliminary formalisation," in *Proc. 4th International Symposium on Formal Methods for Components and Objects, LNCS 4111*, F. de Boer *et al.*, Ed. Springer-Verlag, 2006, pp. 1–21.

[67] F.Arbab, "What do you mean, coordination," in *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, 1998, pp. 11–22.

[68] B. Whittle, "Models and languages for component description and reuse," *SIGSOFT Softw. Eng. Notes*, vol. 20, no. 2, pp. 76–89, 1995.

[69] I. Sommerville, *Software engineering (5th ed.)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1995.

[70] R. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 1996.

[71] B. Norris, J. Ray, R. C. Armstrong, L. C. McInnes, D. E. Bernholdt, W. R. Elwasif, A. D. Malony, e S. Shende, "Computational quality of service for scientific components." in *CBSE*, ser. Lecture Notes in Computer Science, I. Crnkovic, J. A. Stafford, H. W. Schmidt, e K. C. Wallnau, Eds., vol. 3054. Springer, 2004, pp. 264–271.

[72] M. Govindaraju. S. Krishnan, K. Chiu, A. Slominski, D. Gannon, e R. Bramley, "Merging the CCA component model with the OGSI framework," in *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2003, p. 182.

[73] R. Armstrong, G. Kumfert, L. McInnes, B. A. S. Parker, M. Sottile, T. Epperly, e T. Dahlgreen, "The CCA component model for high-performance scientific computing," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 2, pp. 215–229, 2006.

[74] S. Krishnan e D. Gannon, "XCAT3: A framework for CCA components as OGSA services," *hips*, vol. 00, pp. 90–97, 2004.

[75] S. Lefantzi, J. Ray, e H. N. Najm, "Using the common component architecture to design high performance scientific simulation codes," in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 52.1.

[76] R. Armstrong, G. Kumfert, L. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, e T. Dahlgren, "The cca component model for high-performance scientific computing," *Concurrency Computation : Practice & Experience*, vol. 18, no. 2, pp. 215–229, 2006.

[77] R. L. Jacob, C. Schafer, I. T. Foster, M. Tobis, e J. Anderson, "Computational design and performance of the fast ocean atmosphere model, version one," in *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*. London, UK: Springer-Verlag, 2001, pp. 175–184.

[78] J. Larson, R. Jacob, e E. Ong, "Model coupling toolkit web site," 2005. [Online]. Available: http://www.mcs.anl.gov/mct

[79] G. Geist, J. Kohl, e P. Papadopoulos, "Cumulvs: Providing fault-tolerance, visualization and steering of parallel applications," *International Journal of High Performance Computing Applications*, vol. 11, pp. 224–236, 1996.

[80] B. Allan, R. Armstrong, A. Wolfe, J. Ray, D. Bernholdt, e J. Kohl, "The cca core specification in a distributed memory spmd framework," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 5, pp. 323–345, 2002.

[81] C. F. T. W. Group, "Introduction to the ccaffeine framework," 2003.

[82] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, e R. Bramley, "XCAT 2.0: Design and Implementation of Component based Web Services," Department of Computer Science, Indiana University, Tech. Rep., June 2002, tR562.

[83] M. Govindaraju, M. R. Head, e K. Chiu, "Xcat-c++: Design and performance of a distributed cca framework," *The 12th Annual IEEE International Conference on High Performance Computing (HiPC) 2005*, pp. 18–21, December 2005.

[84] D. C. Erdil, K. Chiu, M. Govindaraju, e M. J. Lewis, "A proteus-mediated communications substrate for legioncca and xcat-c++," in *Workshop on Component Models and Frameworks in High Performance Computing (CompFrame)*, 2005.

[85] K. Czajkowski, I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, e S. Tuecke, "A resource management architecture for metacomputing systems," in *IPPSSPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 1998, pp. 62–82.

[86] F. Bertrand e R. Bramley, "DCA: A distributed CCA framework based on MPI." in *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*. Santa Fe, NM: IEEE Press, April 2004, pp. 80–89.

[87] K. Zhang, K. Damevski, V. Venkatachalapathy, e S. Parker, "SCIRun2: A CCA framework for high performance computing," in *In Proceedings of The 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, April 2004.

[88] S. Parker, D. Beazley, e C. Johnson, "Computational steering software systems and strategies," *IEEE Computational Science and Engineering*, vol. 4, no. 4, pp. 50–59, 1997.

[89] S. Parker, "The scirun problem solving environment and computational steering software systems." Ph.D. dissertation, 1999, university of Utah.

[90] C. Johnson e S. Parker, "Applications in computational medicine using SCIRun: A computational steering programming environment," in *Supercomputer '95*, H. Meuer, Ed. Springer-Verlag, 1995, pp. 2–19.

[91] S. Parker, D. Weinstein, e C. Johnson, "The SCIRun computational steering software system," in *Modern Software Tools in Scientific Computing*, E. Arge, A. Bruaset, e H. Langtangen, Eds. Boston: Birkhauser Press, 1997, pp. 1–40.

[92] M. Malawski, D. Kurzyniec, e V. Sunderam, "MOCCA - towards a distributed CCA framework for metacomputing," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4*. Washington, DC, USA: IEEE Computer Society, 2005, p. 174.1.

[93] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, e V. Sunderam, "Towards self-organizing distributed computing frameworks: The H2O approach," *Parallel Processing Letters*, vol. 13, no. 2, pp. 273–290, 2003.

[94] M. Dyrda, M. Malawski, M. Bubak, e S. Naqvi, "Providing security for mocca component environment," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–7, 2009.

[95] G. Kumfert. *Understanding the CCA Standard Through Decaf*, CASC, Lawrence Livermore National Laboratory, Livermore, CA, May 2002, DRAFT.

[96] F. H. Carvalho-Junior e R. D. Lins., "The # model for parallel programming: From processes to components with insignificant performance penalties," in *In II Workshop on Components and Frameworks for High Performance Computing (CompFrame'2005)*. 2005.

[97] F. H. Carvalho-Junior e R. D. Lins. "An institutional theory for #-components," *Electron. Notes Theor. Comput. Sci.*, vol. 195, pp. 113–132, 2008.

[98] W. L. Hursch e C. V. Lopes, "Separation of concerns." College of Computer Science, Northeastern University, Lincoln City, Oregon, Tech. Rep.

[99] N. Bouraqadi, "Concern oriented programming using reflection," OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Minneapolis, Minnesota, USA, Oct. 2000.

[100] J. C. Silva, "Infra-estrutura de componentes paralelos para aplicações de componentes de alto desempenho," Master's thesis, Universidade Federal do Ceará, 2008.

[101] F. H. Carvalho-Junior e R. D. Lins, "The # model: separation of concerns for reconciling modularity, abstraction and efficiency in distributed parallel programming," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA: ACM Press, 2005, pp. 1357–1364.

[102] F. Arbab. "Reo: a channel-based coordination model for component composition," *Mathematical. Structures in Comp. Sci.*, vol. 14, no. 3, pp. 329–366, June 2004.

[103] A. Wang e K. Qian, *Component-Oriented Programming*. Wiley-Interscience, 2005.

[104] F. H. Carvalho-Junior e R. D. Lins, "Compositional specification of parallel components using circus," *Electronic Notes in Theoretical Computer Science*, vol. 260, pp. 47 – 72, 2009.

[105] ——, "A component model for high level and efficient parallel programming on distributed architectures," in *IADIS AC*, N. Guimarães e P. T. Isaías, Eds. IADIS, 2005, pp. 173–178.

[106] F. H. Carvalho-Junior, J. C. Silva, L. P. Queiroz, e R. C. Corrêa, "A high performance computing platform for component-based parallel programming," in *I Workshop on Languages and Tools for Parallel and Distributed Programming, LTPD 2007 - SBAC Workshops*, October 2007.

[107] F. H. Carvalho-Junior e R. D. Lins, "Haskell#: Parallel programming made simple and efficient," *Journal of Universal Computer Science*, vol. 9, no. 8, pp. 776–794, 2003.

[108] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, e J. Stefani, "The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems," *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[109] E. P. Kasten, P. K. McKinley, S. M. Sadjadi, e K. Stirewalt, "Separating introspection and intercession to support metamorphic distributed systems," in *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 465–472.

[110] F. Baude, D. Caromel, e M. Morel, "From distributed objects to hierarchical grid components," in *On The Move to Meaningful Internet Systems 2003: Coopis, DOA, and ODBASE, Lecture Notes in Computer Science*. Springer Verlag, 2003.

[111] M. Leclercq, V. Quéma, e J. Stefani, "DREAM: a component framework for the construction of resource-aware, reconfigurable MOMs," in *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*. New York, NY, USA: ACM Press, 2004, pp. 250–255.

[112] R. Rouvoy, P. Serrano-Alvarado, e P. Merle, "Towards context-aware transaction services," in *Int. Conf. on Distributed Applications and Interoperable Systems (DAIS)*, ser. LNCS 4025. Springer, 2006.

[113] T. CoreGrid, "Proposals for a grid component model. (http://www.coregrid.net)," 2004.

[114] B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes, e B. Smith, "Parallel components for pdes and optimization: some issues and experiences," *Parallel Comput.*, vol. 28, no. 12, pp. 1811–1831, 2002.

[115] F. Bertrand, R. Bramley, D. Bernholdt, J. K. A. James, A. Sussman, J. Larson, e K. Damevski, "Data redistribution and remote method invocation for coupled components," *J. Parallel Distrib. Comput.*, vol. 66, no. 7, pp. 931–946, 2006.

[116] N. Medvidovic e R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[117] V. Barbosa, *An Introduction to Distributed Algorithms*. Cambridge, MA, USA: MIT Press, 1996.

[118] R. C. Corrêa e V. C. Barbosa, "Partially ordered distributed computations on asynchronous point-to-point networks," *Parallel Comput.*, vol. 35, no. 1, pp. 12–28, 2009.

[119] G. Krasner e S. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, 1988.

[120] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. B. Regnell, e A. Wesslén, *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

[121] J. A. Bondy e U. S. R. Murty, *Graph theory with applications*. London, UK: Macmillan, 1976.

[122] R. E. Tarjan e A. E. Trojanowski, "Finding a maximum independent set," Stanford, CA, USA, Tech. Rep., 1976.

[123] M. Garey e D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[124] J. Moon e L. Moser, "On the correlation function of random binary sequences," *SIAM Journal on Applied Mathematics*, vol. 16, no. 2, pp. 340–343, março 1968.

[125] O. Angelsmark, "Partitioning based algorithms for some colouring problems," in *In Recent Advances in Constraints, volume 3978 of LNAI*. Springer Verlag, 2005, pp. 44–58.

[126] A. S. Manne, "Linear programming and sequential decisions," *Management Science*, vol. 6, no. 3, 1960.

[127] M. Trick, "Appendix: Second dimacs challenge test problems," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 26. American Mathematical Society, 1996, pp. 653–657. [Online]. Available: ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique

[128] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[129] J. B. Soares, D. H. Allen, L. Melo, e J. B. Cavalcante-Neto, "Local and global finite element modeling of asphaltic pavements," *Third International Symposium of 3D Finite Element for Pavement Analysis, Design & Research.*, vol. 1, pp. 305–317, 2002.

[130] E. Silva, J. Cavalcante-Neto, e F. Guimarães, "Manual do simfra," 2004.

[131] E. Barszcz, J. Barton, L. Dagum, P. Frederickson, T. Lasinski, R. Schreiber, V. Venkatakrishnan, S. Weeratunga, D. Bailey, D. Bailey, D. Browning, D. Browning, R. Carter, R. Carter, S. Fineberg, S. Fineberg, H. Simon, e H. Simon, "The nas parallel benchmarks," The International Journal of Supercomputer Applications, Tech. Rep., 1991.

[132] D. Posada, T. J. Maxwell, e A. R. Templeton, "Treescan: a bioinformatic application to search for genotype/phenotype associations using haplotype trees," *Bioinformatics*, vol. 21, no. 9, pp. 2130–2132, 2005.